OpenCV Reference Manual

v2.2

December, 2010

Contents

I C API Reference **47**

1 core. The Core Functionality **49**

1.1 Basic Structures . . . 49
CvPoint . . . 49
CvPoint2D32f . . . 49
CvPoint3D32f . . . 50
CvPoint2D64f . . . 51
CvPoint3D64f . . . 51
CvSize . . . 52
CvSize2D32f . . . 52
CvRect . . . 52
CvScalar . . . 53
CvTermCriteria . . . 54
CvMat . . . 54
CvMatND . . . 55
CvSparseMat . . . 56
IplImage . . . 57
CvArr . . . 60

1.2 Operations on Arrays . . . 60
cv::AbsDiff . . . 60
cv::AbsDiffS . . . 60
cv::Add . . . 61
cv::AddS . . . 61
cv::AddWeighted . . . 62
cv::And . . . 63
cv::AndS . . . 63
cv::Avg . . . 64
cv::AvgSdv . . . 65
cv::CalcCovarMatrix . . . 65

cv::CartToPolar . . . 67
cv::Cbrt . . . 68
cv::ClearND . . . 68
cv::CloneImage . . . 68
cv::CloneMat . . . 69
cv::CloneMatND . . . 69
cv::CloneSparseMat . . . 69
cv::Cmp . . . 69
cv::CmpS . . . 70
cv::ConvertScale . . . 71
cv::ConvertScaleAbs . . . 72
cv::CvtScaleAbs . . . 73
cv::Copy . . . 73
cv::CountNonZero . . . 74
cv::CreateData . . . 74
cv::CreateImage . . . 75
cv::CreateImageHeader . . . 75
cv::CreateMat . . . 76
cv::CreateMatHeader . . . 76
cv::CreateMatND . . . 77
cv::CreateMatNDHeader . . . 77
cv::CreateSparseMat . . . 78
cv::CrossProduct . . . 78
CvtPixToPlane . . . 79
cv::DCT . . . 79
cv::DFT . . . 80
cv::DecRefData . . . 83
cv::Det . . . 83
cv::Div . . . 83
cv::DotProduct . . . 84
cv::EigenVV . . . 85
cv::Exp . . . 86
cv::FastArctan . . . 86
cv::Flip . . . 86
cv::GEMM . . . 87
cv::Get?D . . . 88
cv::GetCol(s) . . . 89
cv::GetDiag . . . 90
cvGetDims, cvGetDimSize . . . 90
cv::GetElemType . . . 91

cv::GetImage . . . 91
cv::GetImageCOI . . . 92
cv::GetImageROI . . . 92
cv::GetMat . . . 93
cv::GetNextSparseNode . . . 93
cv::GetOptimalDFTSize . . . 94
cv::GetRawData . . . 95
cv::GetReal1D . . . 95
cv::GetReal2D . . . 96
cv::GetReal3D . . . 96
cv::GetRealND . . . 97
cv::GetRow(s) . . . 98
cv::GetSize . . . 98
cv::GetSubRect . . . 99
cv::InRange . . . 99
cv::InRangeS . . . 100
cv::IncRefData . . . 100
cv::InitImageHeader . . . 101
cv::InitMatHeader . . . 102
cv::InitMatNDHeader . . . 103
cv::InitSparseMatIterator . . . 103
cv::InvSqrt . . . 104
cv::Inv . . . 104
cv::Invert . . . 104
cv::IsInf . . . 105
cv::IsNaN . . . 105
cv::LUT . . . 105
cv::Log . . . 106
cv::Mahalanobis . . . 107
cv::Mat . . . 107
cv::Max . . . 108
cv::MaxS . . . 108
cv::Merge . . . 109
cv::Min . . . 110
cv::MinMaxLoc . . . 110
cv::MinS . . . 111
Mirror . . . 111
cv::MixChannels . . . 111
MulAddS . . . 112
cv::Mul . . . 112

cv::MulSpectrums . . . 113
cv::MulTransposed . . . 114
cv::Norm . . . 114
cv::Not . . . 115
cv::Or . . . 116
cv::OrS . . . 116
cv::PerspectiveTransform . . . 117
cv::PolarToCart . . . 117
cv::Pow . . . 118
cv::Ptr?D . . . 119
cv::RNG . . . 120
cv::RandArr . . . 120
cv::RandInt . . . 122
cv::RandReal . . . 123
cv::Reduce . . . 124
cv::ReleaseData . . . 124
cv::ReleaseImage . . . 125
cv::ReleaseImageHeader . . . 125
cv::ReleaseMat . . . 126
cv::ReleaseMatND . . . 126
cv::ReleaseSparseMat . . . 127
cv::Repeat . . . 127
cv::ResetImageROI . . . 127
cv::Reshape . . . 128
cv::ReshapeMatND . . . 129
cvRound, cvFloor, cvCeil . . . 130
cv::ScaleAdd . . . 130
cv::Set . . . 131
cv::Set?D . . . 131
cv::SetData . . . 132
cv::SetIdentity . . . 132
cv::SetImageCOI . . . 133
cv::SetImageROI . . . 133
cv::SetReal?D . . . 134
cv::SetZero . . . 134
cv::Solve . . . 135
cv::SolveCubic . . . 136
cv::Split . . . 136
cv::Sqrt . . . 137
cv::Sub . . . 137

cv::SubRS . . . 138
cv::SubS . . . 138
cv::Sum . . . 139
cv::SVBkSb . . . 139
cv::SVD . . . 140
cv::Trace . . . 142
cv::Transform . . . 142
cv::Transpose . . . 143
cv::Xor . . . 143
cv::XorS . . . 144
cv::mGet . . . 145
cv::mSet . . . 145
1.3 Dynamic Structures . . . 146
CvMemStorage . . . 146
CvMemBlock . . . 147
CvMemStoragePos . . . 147
CvSeq . . . 147
CvSeqBlock . . . 150
CvSlice . . . 150
CvSet . . . 151
CvGraph . . . 152
CvGraphScanner . . . 153
CV_TREE_NODE_FIELDS . . . 154
CvTreeNodeIterator . . . 154
cv::ClearGraph . . . 155
cv::ClearMemStorage . . . 155
cv::ClearSeq . . . 156
cv::ClearSet . . . 156
cv::CloneGraph . . . 156
cv::CloneSeq . . . 157
cv::CreateChildMemStorage . . . 157
cv::CreateGraph . . . 159
cv::CreateGraphScanner . . . 159
cv::CreateMemStorage . . . 160
cv::CreateSeq . . . 161
cv::CreateSet . . . 162
cv::CvtSeqToArray . . . 162
cv::EndWriteSeq . . . 163
cv::FindGraphEdge . . . 163
cv::FindGraphEdgeByPtr . . . 164

cv::FlushSeqWriter . . . 164
cv::GetGraphVtx . . . 165
cv::GetSeqElem . . . 165
cv::GetSeqReaderPos . . . 166
cv::GetSetElem . . . 166
cv::GraphAddEdge . . . 166
cv::GraphAddEdgeByPtr . . . 167
cv::GraphAddVtx . . . 168
cv::GraphEdgeIdx . . . 168
cv::GraphRemoveEdge . . . 169
cv::GraphRemoveEdgeByPtr . . . 169
cv::GraphRemoveVtx . . . 170
cv::GraphRemoveVtxByPtr . . . 170
cv::GraphVtxDegree . . . 171
cv::GraphVtxDegreeByPtr . . . 171
cv::GraphVtxIdx . . . 172
cv::InitTreeNodeIterator . . . 172
cv::InsertNodeIntoTree . . . 173
cv::MakeSeqHeaderForArray . . . 173
cv::MemStorageAlloc . . . 174
cv::MemStorageAllocString . . . 174
cv::NextGraphItem . . . 175
cv::NextTreeNode . . . 176
cv::PrevTreeNode . . . 176
cv::ReleaseGraphScanner . . . 176
cv::ReleaseMemStorage . . . 177
cv::RestoreMemStoragePos . . . 177
cv::SaveMemStoragePos . . . 177
cv::SeqElemIdx . . . 178
cv::SeqInsert . . . 178
cv::SeqInsertSlice . . . 179
cv::SeqInvert . . . 180
cv::SeqPop . . . 180
cv::SeqPopFront . . . 180
cv::SeqPopMulti . . . 181
cv::SeqPush . . . 182
cv::SeqPushFront . . . 182
cv::SeqPushMulti . . . 183
cv::SeqRemove . . . 184
cv::SeqRemoveSlice . . . 184

cv::SeqSearch . . . 184
cv::SeqSlice . . . 185
cv::SeqSort . . . 186
cv::SetAdd . . . 187
cv::SetNew . . . 188
cv::SetRemove . . . 188
cv::SetRemoveByPtr . . . 189
cv::SetSeqBlockSize . . . 189
cv::SetSeqReaderPos . . . 190
cv::StartAppendToSeq . . . 190
cv::StartReadSeq . . . 191
cv::StartWriteSeq . . . 192
cv::TreeToNodeSeq . . . 193
1.4 Drawing Functions . . . 193
cv::Circle . . . 194
cv::ClipLine . . . 195
cv::DrawContours . . . 195
cv::Ellipse . . . 197
cv::EllipseBox . . . 198
cv::FillConvexPoly . . . 199
cv::FillPoly . . . 200
cv::GetTextSize . . . 201
cv::InitFont . . . 201
cv::InitLineIterator . . . 202
cv::Line . . . 204
cv::PolyLine . . . 205
cv::PutText . . . 206
cv::Rectangle . . . 206
CV_RGB . . . 207
1.5 XML/YAML Persistence . . . 208
CvFileStorage . . . 208
CvFileNode . . . 209
CvAttrList . . . 211
CvTypeInfo . . . 211
cv::Clone . . . 212
cv::EndWriteStruct . . . 213
cv::FindType . . . 213
cv::FirstType . . . 213
cv::GetFileNode . . . 214
cv::GetFileNodeByName . . . 214

cv::GetFileNodeName . . . 215
cv::GetHashedKey . . . 215
cv::GetRootFileNode . . . 217
cv::Load . . . 218
cv::OpenFileStorage . . . 218
cv::Read . . . 219
cv::ReadByName . . . 220
cv::ReadInt . . . 220
cv::ReadIntByName . . . 221
cv::ReadRawData . . . 221
cv::ReadRawDataSlice . . . 222
cv::ReadReal . . . 222
cv::ReadRealByName . . . 223
cv::ReadString . . . 223
cv::ReadStringByName . . . 224
cv::RegisterType . . . 224
cv::Release . . . 225
cv::ReleaseFileStorage . . . 225
cv::Save . . . 225
cv::StartNextStream . . . 226
cv::StartReadRawData . . . 226
cv::StartWriteStruct . . . 227
cv::TypeOf . . . 228
cv::UnregisterType . . . 228
cv::Write . . . 228
cv::WriteComment . . . 230
cv::WriteFileNode . . . 230
cv::WriteInt . . . 231
cv::WriteRawData . . . 231
cv::WriteReal . . . 233
cv::WriteString . . . 233
1.6 Clustering . . . 234
cv::KMeans2 . . . 234
cv::SeqPartition . . . 237
1.7 Utility and System Functions and Macros . . . 239
Error Handling . . . 239
Example: Use of Error Handling Macros . . . 241
cv::GetErrStatus . . . 243
cv::SetErrStatus . . . 243
cv::GetErrMode . . . 243

cv::SetErrMode . . . 244
cv::Error . . . 244
cv::ErrorStr . . . 245
cv::RedirectError . . . 245
cvNulDevReport cvStdErrReport cvGuiBoxReport . . . 246
cv::Alloc . . . 247
cv::Free . . . 247
cv::GetTickCount . . . 248
cv::GetTickFrequency . . . 248
cv::RegisterModule . . . 248
cv::GetModuleInfo . . . 249
cv::UseOptimized . . . 250
cv::SetMemoryManager . . . 250
cv::SetIPLAllocators . . . 251

2 imgproc. Image Processing 253
2.1 Histograms . . . 253
CvHistogram . . . 253
cv::CalcBackProject . . . 253
cv::CalcBackProjectPatch . . . 254
cv::CalcHist . . . 256
cv::CalcProbDensity . . . 257
cv::ClearHist . . . 258
cv::CompareHist . . . 259
cv::CopyHist . . . 260
cv::CreateHist . . . 260
cv::GetHistValue*D . . . 261
cv::GetMinMaxHistValue . . . 262
cv::MakeHistHeaderForArray . . . 263
cv::NormalizeHist . . . 264
cv::QueryHistValue*D . . . 264
cv::ReleaseHist . . . 265
cv::SetHistBinRanges . . . 265
cv::ThreshHist . . . 265
2.2 Image Filtering . . . 266
IplConvKernel . . . 266
cv::CopyMakeBorder . . . 266
cv::CreateStructuringElementEx . . . 267
cv::Dilate . . . 268
cv::Erode . . . 269

cv::Filter2D . . . 270
cv::Laplace . . . 271
cv::MorphologyEx . . . 271
cv::PyrDown . . . 273
cv::ReleaseStructuringElement . . . 273
cv::Smooth . . . 274
cv::Sobel . . . 275
2.3 Geometric Image Transformations . . . 277
cv::GetRotationMatrix2D . . . 278
cv::GetAffineTransform . . . 278
cv::GetPerspectiveTransform . . . 279
cv::GetQuadrangleSubPix . . . 280
cv::GetRectSubPix . . . 280
cv::LogPolar . . . 281
cv::Remap . . . 283
cv::Resize . . . 284
cv::WarpAffine . . . 285
cv::WarpPerspective . . . 286
2.4 Miscellaneous Image Transformations . . . 287
cv::AdaptiveThreshold . . . 287
cv::CvtColor . . . 288
cv::DistTransform . . . 293
CvConnectedComp . . . 295
cv::FloodFill . . . 295
cv::Inpaint . . . 297
cv::Integral . . . 298
cv::PyrMeanShiftFiltering . . . 299
cv::PyrSegmentation . . . 300
cv::Threshold . . . 301
2.5 Structural Analysis and Shape Descriptors . . . 303
cv::ApproxChains . . . 303
cv::ApproxPoly . . . 304
cv::ArcLength . . . 305
cv::BoundingRect . . . 306
cv::BoxPoints . . . 306
cv::CalcPGH . . . 307
cv::CalcEMD2 . . . 308
cv::CheckContourConvexity . . . 309
CvConvexityDefect . . . 309
cv::ContourArea . . . 310

cv::ContourFromContourTree . . . 311
cv::ConvexHull2 . . . 312
cv::ConvexityDefects . . . 314
cv::CreateContourTree . . . 315
cv::EndFindContours . . . 316
cv::FindContours . . . 316
cv::FindNextContour . . . 318
cv::FitEllipse2 . . . 318
cv::FitLine . . . 318
cv::GetCentralMoment . . . 320
cv::GetHuMoments . . . 320
cv::GetNormalizedCentralMoment . . . 321
cv::GetSpatialMoment . . . 322
cv::MatchContourTrees . . . 322
cv::MatchShapes . . . 323
cv::MinAreaRect2 . . . 324
cv::MinEnclosingCircle . . . 325
cv::Moments . . . 326
cv::PointPolygonTest . . . 326
cv::PointSeqFromMat . . . 327
cv::ReadChainPoint . . . 328
cv::StartFindContours . . . 329
cv::StartReadChainPoints . . . 329
cv::SubstituteContour . . . 330
2.6 Planar Subdivisions . . . 330
CvSubdiv2D . . . 330
CvQuadEdge2D . . . 331
CvSubdiv2DPoint . . . 332
cv::CalcSubdivVoronoi2D . . . 333
cv::ClearSubdivVoronoi2D . . . 333
cv::CreateSubdivDelaunay2D . . . 333
cv::FindNearestPoint2D . . . 334
cv::Subdiv2DEdgeDst . . . 334
cv::Subdiv2DGetEdge . . . 335
cv::Subdiv2DNextEdge . . . 336
cv::Subdiv2DLocate . . . 337
cv::Subdiv2DRotateEdge . . . 338
cv::SubdivDelaunay2DInsert . . . 339
2.7 Motion Analysis and Object Tracking . . . 340
cv::Acc . . . 340

cv::MultiplyAcc . . . 340
cv::RunningAvg . . . 341
cv::SquareAcc . . . 342
2.8 Feature Detection . . . 342
cv::Canny . . . 342
cv::CornerEigenValsAndVecs . . . 343
cv::CornerHarris . . . 344
cv::CornerMinEigenVal . . . 345
cv::FindCornerSubPix . . . 345
cv::GoodFeaturesToTrack . . . 347
cv::HoughLines2 . . . 348
cv::PreCornerDetect . . . 352
cv::SampleLine . . . 353
2.9 Object Detection . . . 354
cv::MatchTemplate . . . 354

3 features2d. Feature Detection and Descriptor Extraction 357
3.1 Feature detection and description . . . 357
cv::ExtractSURF . . . 357
cv::GetStarKeypoints . . . 359

4 flann. Clustering and Search in Multi-Dimensional Spaces 363
4.1 Fast Approximate Nearest Neighbor Search . . . 363

5 objdetect. Object Detection 365
5.1 Cascade Classification . . . 365
Haar Feature-based Cascade Classifier for Object Detection . . . 365
CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade 366
cv::LoadHaarClassifierCascade . . . 369
cv::HaarDetectObjects . . . 369
cv::SetImagesForHaarClassifierCascade . . . 372
cv::ReleaseHaarClassifierCascade . . . 373
cv::RunHaarClassifierCascade . . . 373

6 video. Video Analysis 375
6.1 Motion Analysis and Object Tracking . . . 375
cv::CalcGlobalOrientation . . . 375
cv::CalcMotionGradient . . . 376
cv::CalcOpticalFlowBM . . . 377
cv::CalcOpticalFlowHS . . . 378

cv::CalcOpticalFlowLK . . . 379
cv::CalcOpticalFlowPyrLK . . . 379
cv::CamShift . . . 381
CvConDensation . . . 382
cv::CreateConDensation . . . 382
cv::ConDensInitSampleSet . . . 383
CvKalman . . . 383
cv::CreateKalman . . . 385
cv::KalmanCorrect . . . 385
cv::KalmanPredict . . . 388
KalmanUpdateByMeasurement . . . 389
KalmanUpdateByTime . . . 389
cv::MeanShift . . . 389
cv::ReleaseConDensation . . . 390
cv::ReleaseKalman . . . 390
cv::SegmentMotion . . . 391
cv::SnakeImage . . . 392
cv::UpdateMotionHistory . . . 393

7 highgui. High-level GUI and Media I/O 395
7.1 User Interface . . . 395
cv::ConvertImage . . . 395
cv::CreateTrackbar . . . 396
cv::DestroyAllWindows . . . 397
cv::DestroyWindow . . . 397
cv::GetTrackbarPos . . . 398
cv::GetWindowHandle . . . 398
cv::GetWindowName . . . 399
cv::InitSystem . . . 399
cv::MoveWindow . . . 400
cv::NamedWindow . . . 400
cv::ResizeWindow . . . 401
cv::SetMouseCallback . . . 401
cv::SetTrackbarPos . . . 403
cv::ShowImage . . . 403
cv::WaitKey . . . 404
7.2 Reading and Writing Images and Video . . . 404
cv::LoadImage . . . 404
cv::LoadImageM . . . 405
cv::SaveImage . . . 406

CvCapture . . . 406
cv::CaptureFromCAM . . . 407
cv::CaptureFromFile . . . 407
cv::GetCaptureProperty . . . 408
cv::GrabFrame . . . 409
cv::QueryFrame . . . 409
cv::ReleaseCapture . . . 410
cv::RetrieveFrame . . . 410
cv::SetCaptureProperty . . . 410
cv::CreateVideoWriter . . . 412
cv::ReleaseVideoWriter . . . 412
cv::WriteFrame . . . 413
7.3 Qt new functions . . . 414
cv::SetWindowProperty . . . 415
cv::GetWindowProperty . . . 416
cv::FontQt . . . 417
cv::AddText . . . 418
cv::DisplayOverlay . . . 418
cv::DisplayStatusBar . . . 419
cv::CreateOpenGLCallback . . . 419
cv::SaveWindowParameters . . . 421
cv::LoadWindowParameters . . . 421
cv::CreateButton . . . 421

8 calib3d. Camera Calibration, Pose Estimation and Stereo 423
8.1 Camera Calibration and 3d Reconstruction . . . 423
cv::CalcImageHomography . . . 425
cv::CalibrateCamera2 . . . 425
cv::ComputeCorrespondEpilines . . . 428
cv::ConvertPointsHomogeneous . . . 429
cv::CreatePOSITObject . . . 430
cv::CreateStereoBMState . . . 430
cv::CreateStereoGCState . . . 431
CvStereoBMState . . . 431
CvStereoGCState . . . 433
cv::DecomposeProjectionMatrix . . . 434
cv::DrawChessboardCorners . . . 435
cv::FindChessboardCorners . . . 436
cv::FindExtrinsicCameraParams2 . . . 437
cv::FindFundamentalMat . . . 438

cv::FindHomography . . . 440
cv::FindStereoCorrespondenceBM . . . 442
cv::FindStereoCorrespondenceGC . . . 443
cv::GetOptimalNewCameraMatrix . . . 445
cv::InitIntrinsicParams2D . . . 446
cv::InitUndistortMap . . . 447
cv::InitUndistortRectifyMap . . . 448
cv::POSIT . . . 449
cv::ProjectPoints2 . . . 450
cv::ReprojectImageTo3D . . . 451
cv::RQDecomp3x3 . . . 452
cv::ReleasePOSITObject . . . 453
cv::ReleaseStereoBMState . . . 453
cv::ReleaseStereoGCState . . . 454
cv::Rodrigues2 . . . 454
cv::StereoCalibrate . . . 455
cv::StereoRectify . . . 458
cv::StereoRectifyUncalibrated . . . 461
cv::Undistort2 . . . 462
cv::UndistortPoints . . . 463

9 ml. Machine Learning **465**

II C++ API Reference **467**

10 Introduction **469**
10.1 C++ Cheatsheet . . . 472
10.2 Namespace cv and Function Naming . . . 473
10.3 Memory Management . . . 474
10.4 Memory Management Part II. Automatic Data Allocation . . . 475
10.5 Algebraic Operations . . . 477
10.6 Fast Element Access . . . 477
10.7 Saturation Arithmetics . . . 478
10.8 Error handling . . . 478
10.9 Threading and Reenterability . . . 479

11 core. The Core Functionality **481**
11.1 Basic Structures . . . 481
DataType . . . 481

Point_ . . . 483
Point3_ . . . 484
Size_ . . . 485
Rect_ . . . 486
RotatedRect . . . 488
TermCriteria . . . 488
Matx . . . 489
Vec . . . 490
Scalar_ . . . 491
Range . . . 492
Ptr . . . 492
Mat . . . 495
Matrix Expressions . . . 500
cv::Mat::Mat . . . 501
cv::Mat::Mat . . . 503
cv::Mat::operator = . . . 503
cv::Mat::operator MatExpr . . . 504
cv::Mat::row . . . 504
cv::Mat::col . . . 505
cv::Mat::rowRange . . . 506
cv::Mat::colRange . . . 506
cv::Mat::diag . . . 506
cv::Mat::clone . . . 507
cv::Mat::copyTo . . . 507
cv::Mat::convertTo . . . 508
cv::Mat::assignTo . . . 509
cv::Mat::setTo . . . 509
cv::Mat::reshape . . . 509
cv::Mat::t . . . 510
cv::Mat::inv . . . 510
cv::Mat::mul . . . 511
cv::Mat::cross . . . 512
cv::Mat::dot . . . 512
cv::Mat::zeros . . . 512
cv::Mat::ones . . . 513
cv::Mat::eye . . . 514
cv::Mat::create . . . 514
cv::Mat::addref . . . 515
cv::Mat::release . . . 516
cv::Mat::resize . . . 516

Mat::push_back . . . 517
Mat::pop_back . . . 517
cv::Mat::locateROI . . . 517
cv::Mat::adjustROI . . . 518
cv::Mat::operator() . . . 519
cv::Mat::operator CvMat . . . 519
cv::Mat::operator IplImage . . . 520
cv::Mat::total . . . 520
cv::Mat::isContinuous . . . 520
cv::Mat::elemSize . . . 522
cv::Mat::elemSize1 . . . 522
cv::Mat::type . . . 523
cv::Mat::depth . . . 523
cv::Mat::channels . . . 524
cv::Mat::step1 . . . 524
cv::Mat::size . . . 524
cv::Mat::empty . . . 525
cv::Mat::ptr . . . 525
cv::Mat::at . . . 525
cv::Mat::begin . . . 526
cv::Mat::end . . . 527
Mat_ . . . 528
NAryMatIterator . . . 529
SparseMat . . . 530
SparseMat_ . . . 536
11.2 Operations on Arrays . . . 538
cv::abs . . . 538
cv::absdiff . . . 538
cv::add . . . 539
cv::addWeighted . . . 540
bitwise_and . . . 541
bitwise_not . . . 542
bitwise_or . . . 543
bitwise_xor . . . 544
cv::calcCovarMatrix . . . 545
cv::cartToPolar . . . 546
cv::checkRange . . . 547
cv::compare . . . 548
cv::completeSymm . . . 549
cv::convertScaleAbs . . . 549

cv::countNonZero . . . 550
cv::cubeRoot . . . 551
cv::cvarrToMat . . . 551
cv::dct . . . 553
cv::dft . . . 554
cv::divide . . . 558
cv::determinant . . . 559
cv::eigen . . . 560
cv::exp . . . 560
cv::extractImageCOI . . . 561
cv::fastAtan2 . . . 562
cv::flip . . . 562
cv::gemm . . . 563
cv::getConvertElem . . . 564
cv::getOptimalDFTSize . . . 565
cv::idct . . . 565
cv::idft . . . 566
cv::inRange . . . 566
cv::invert . . . 567
cv::log . . . 568
cv::LUT . . . 568
cv::magnitude . . . 569
cv::Mahalanobis . . . 570
cv::max . . . 570
cv::mean . . . 571
cv::meanStdDev . . . 572
cv::merge . . . 573
cv::min . . . 573
cv::minMaxLoc . . . 574
cv::mixChannels . . . 575
cv::mulSpectrums . . . 577
cv::multiply . . . 578
cv::mulTransposed . . . 578
cv::norm . . . 579
cv::normalize . . . 580
cv::PCA . . . 582
cv::PCA::PCA . . . 583
cv::PCA::operator () . . . 584
cv::PCA::project . . . 585
cv::PCA::backProject . . . 585

cv::perspectiveTransform . . . 586
cv::phase . . . 587
cv::polarToCart . . . 588
cv::pow . . . 588
RNG . . . 589
cv::RNG::RNG . . . 590
cv::RNG::next . . . 591
cv::RNG::operator T . . . 591
cv::RNG::operator () . . . 591
cv::RNG::uniform . . . 592
cv::RNG::gaussian . . . 593
cv::RNG::fill . . . 593
cv::randu . . . 594
cv::randn . . . 594
cv::randShuffle . . . 595
cv::reduce . . . 596
cv::repeat . . . 596
saturate_cast . . . 597
cv::scaleAdd . . . 598
cv::setIdentity . . . 599
cv::solve . . . 599
cv::solveCubic . . . 600
cv::solvePoly . . . 601
cv::sort . . . 601
cv::sortIdx . . . 602
cv::split . . . 603
cv::sqrt . . . 603
cv::subtract . . . 604
cv::SVD . . . 605
cv::SVD::SVD . . . 606
cv::SVD::operator () . . . 607
cv::SVD::solveZ . . . 607
cv::SVD::backSubst . . . 608
cv::sum . . . 608
cv::theRNG . . . 609
cv::trace . . . 609
cv::transform . . . 610
cv::transpose . . . 611
11.3 Dynamic Structures . . . 611
11.4 Drawing Functions . . . 611

cv::circle . . . 612
cv::clipLine . . . 612
cv::ellipse . . . 613
cv::ellipse2Poly . . . 614
cv::fillConvexPoly . . . 615
cv::fillPoly . . . 616
cv::getTextSize . . . 616
cv::line . . . 617
cv::LineIterator . . . 618
cv::rectangle . . . 619
cv::polylines . . . 620
cv::putText . . . 621
11.5 XML/YAML Persistence . . . 621
cv::FileStorage . . . 621
cv::FileNode . . . 623
cv::FileNodeIterator . . . 624
11.6 Clustering . . . 624
cv::kmeans . . . 624
cv::partition . . . 626
11.7 Utility and System Functions and Macros . . . 626
cv::alignPtr . . . 626
cv::alignSize . . . 627
cv::allocate . . . 627
cv::deallocate . . . 627
CV_Assert . . . 628
cv::error . . . 628
cv::Exception . . . 629
cv::fastMalloc . . . 629
cv::fastFree . . . 630
cv::format . . . 630
cv::getNumThreads . . . 631
cv::getThreadNum . . . 631
cv::getTickCount . . . 631
cv::getTickFrequency . . . 632
cv::setNumThreads . . . 632

12 imgproc. Image Processing **633**
12.1 Histograms . . . 633
cv::calcHist . . . 633
cv::calcBackProject . . . 636

cv::compareHist . . . 637
cv::equalizeHist . . . 638
12.2 Image Filtering . . . 639
cv::BaseColumnFilter . . . 640
cv::BaseFilter . . . 640
cv::BaseRowFilter . . . 641
cv::FilterEngine . . . 642
cv::bilateralFilter . . . 647
cv::blur . . . 648
cv::borderInterpolate . . . 648
cv::boxFilter . . . 649
cv::buildPyramid . . . 650
cv::copyMakeBorder . . . 650
cv::createBoxFilter . . . 652
cv::createDerivFilter . . . 653
cv::createGaussianFilter . . . 653
cv::createLinearFilter . . . 654
cv::createMorphologyFilter . . . 655
cv::createSeparableLinearFilter . . . 656
cv::dilate . . . 657
cv::erode . . . 658
cv::filter2D . . . 659
cv::GaussianBlur . . . 660
cv::getDerivKernels . . . 661
cv::getGaussianKernel . . . 662
cv::getKernelType . . . 662
cv::getStructuringElement . . . 663
cv::medianBlur . . . 664
cv::morphologyEx . . . 664
cv::Laplacian . . . 665
cv::pyrDown . . . 666
cv::pyrUp . . . 667
cv::sepFilter2D . . . 668
cv::Sobel . . . 669
cv::Scharr . . . 670
12.3 Geometric Image Transformations . . . 671
cv::convertMaps . . . 672
cv::getAffineTransform . . . 673
cv::getPerspectiveTransform . . . 673
cv::getRectSubPix . . . 674

cv::getRotationMatrix2D . . . 675
cv::invertAffineTransform . . . 675
cv::remap . . . 676
cv::resize . . . 677
cv::warpAffine . . . 678
cv::warpPerspective . . . 679
12.4 Miscellaneous Image Transformations . . . 680
cv::adaptiveThreshold . . . 680
cv::cvtColor . . . 681
cv::distanceTransform . . . 687
cv::floodFill . . . 688
cv::inpaint . . . 690
cv::integral . . . 691
cv::threshold . . . 692
cv::watershed . . . 694
cv::grabCut . . . 695
12.5 Structural Analysis and Shape Descriptors . . . 696
cv::moments . . . 696
cv::HuMoments . . . 697
cv::findContours . . . 698
cv::drawContours . . . 700
cv::approxPolyDP . . . 701
cv::arcLength . . . 702
cv::boundingRect . . . 703
cv::estimateRigidTransform . . . 703
cv::estimateAffine3D . . . 704
cv::contourArea . . . 704
cv::convexHull . . . 705
cv::fitEllipse . . . 706
cv::fitLine . . . 706
cv::isContourConvex . . . 708
cv::minAreaRect . . . 708
cv::minEnclosingCircle . . . 709
cv::matchShapes . . . 709
cv::pointPolygonTest . . . 710
12.6 Planar Subdivisions . . . 711
12.7 Motion Analysis and Object Tracking . . . 711
cv::accumulate . . . 711
cv::accumulateSquare . . . 712
cv::accumulateProduct . . . 713

cv::accumulateWeighted . . . 713
12.8 Feature Detection . . . 714
cv::Canny . . . 714
cv::cornerEigenValsAndVecs . . . 715
cv::cornerHarris . . . 716
cv::cornerMinEigenVal . . . 716
cv::cornerSubPix . . . 717
cv::goodFeaturesToTrack . . . 718
cv::HoughCircles . . . 720
cv::HoughLines . . . 722
cv::HoughLinesP . . . 723
cv::preCornerDetect . . . 725
12.9 Object Detection . . . 726
cv::matchTemplate . . . 726

13 features2d. Feature Detection and Descriptor Extraction 729
13.1 Feature detection and description . . . 729
cv::FAST . . . 729
cv::MSER . . . 729
cv::StarDetector . . . 730
cv::SIFT . . . 731
cv::SURF . . . 733
cv::RandomizedTree . . . 733
cv::RandomizedTree::train . . . 735
cv::RandomizedTree::read . . . 736
cv::RandomizedTree::write . . . 736
cv::RandomizedTree::applyQuantization . . . 737
cv::RTreeNode . . . 737
cv::RTreeClassifier . . . 737
cv::RTreeClassifier::train . . . 739
cv::RTreeClassifier::getSignature . . . 740
cv::RTreeClassifier::getSparseSignature . . . 740
cv::RTreeClassifier::countNonZeroElements . . . 741
cv::RTreeClassifier::read . . . 741
cv::RTreeClassifier::write . . . 741
cv::RTreeClassifier::setQuantization . . . 742
13.2 Common Interfaces of Feature Detectors . . . 744
cv::KeyPoint . . . 744
cv::FeatureDetector . . . 745
cv::FeatureDetector::detect . . . 746

cv::FeatureDetector::read . . . 747
cv::FeatureDetector::write . . . 747
cv::FeatureDetector::create . . . 747
cv::FastFeatureDetector . . . 748
cv::GoodFeaturesToTrackDetector . . . 748
cv::MserFeatureDetector . . . 749
cv::StarFeatureDetector . . . 749
cv::SiftFeatureDetector . . . 750
cv::SurfFeatureDetector . . . 750
cv::GridAdaptedFeatureDetector . . . 751
cv::PyramidAdaptedFeatureDetector . . . 751
cv::DynamicAdaptedFeatureDetector . . . 751
cv::DynamicAdaptedFeatureDetector::DynamicAdaptedFeatureDetector . . . 752
cv::AdjusterAdapter . . . 753
cv::AdjusterAdapter::tooFew . . . 753
cv::AdjusterAdapter::tooMany . . . 754
cv::AdjusterAdapter::good . . . 754
cv::FastAdjuster . . . 755
cv::StarAdjuster . . . 755
cv::SurfAdjuster . . . 755
13.3 Common Interfaces of Descriptor Extractors . . . 755
cv::DescriptorExtractor . . . 756
cv::DescriptorExtractor::compute . . . 756
cv::DescriptorExtractor::read . . . 757
cv::DescriptorExtractor::write . . . 757
cv::DescriptorExtractor::create . . . 758
cv::SiftDescriptorExtractor . . . 758
cv::SurfDescriptorExtractor . . . 759
cv::CalonderDescriptorExtractor . . . 759
cv::OpponentColorDescriptorExtractor . . . 759
cv::BriefDescriptorExtractor . . . 760
13.4 Common Interfaces of Descriptor Matchers . . . 760
cv::DMatch . . . 761
cv::DescriptorMatcher . . . 761
cv::DescriptorMatcher::add . . . 762
cv::DescriptorMatcher::getTrainDescriptors . . . 763
cv::DescriptorMatcher::clear . . . 763
cv::DescriptorMatcher::empty . . . 763
cv::DescriptorMatcher::isMaskSupported . . . 763
cv::DescriptorMatcher::train . . . 764

cv::DescriptorMatcher::match . . . 764
cv::DescriptorMatcher::knnMatch . . . 765
cv::DescriptorMatcher::radiusMatch . . . 766
cv::DescriptorMatcher::clone . . . 766
cv::DescriptorMatcher::create . . . 767
cv::BruteForceMatcher . . . 767
cv::FlannBasedMatcher . . . 769
13.5 Common Interfaces of Generic Descriptor Matchers . . . 769
cv::GenericDescriptorMatcher . . . 770
cv::GenericDescriptorMatcher::add . . . 771
cv::GenericDescriptorMatcher::getTrainImages . . . 772
cv::GenericDescriptorMatcher::getTrainKeypoints . . . 772
cv::GenericDescriptorMatcher::clear . . . 772
cv::GenericDescriptorMatcher::train . . . 772
cv::GenericDescriptorMatcher::isMaskSupported . . . 772
cv::GenericDescriptorMatcher::classify . . . 772
cv::GenericDescriptorMatcher::match . . . 773
cv::GenericDescriptorMatcher::knnMatch . . . 774
cv::GenericDescriptorMatcher::radiusMatch . . . 775
cv::GenericDescriptorMatcher::read . . . 775
cv::GenericDescriptorMatcher::write . . . 776
cv::GenericDescriptorMatcher::clone . . . 776
cv::OneWayDescriptorMatcher . . . 776
cv::FernDescriptorMatcher . . . 777
cv::VectorDescriptorMatcher . . . 779
13.6 Drawing Function of Keypoints and Matches . . . 779
cv::drawMatches . . . 779
cv::drawKeypoints . . . 781
13.7 Object Categorization . . . 782
cv::BOWTrainer . . . 782
cv::BOWTrainer::add . . . 782
cv::BOWTrainer::getDescriptors . . . 783
cv::BOWTrainer::descripotorsCount . . . 783
cv::BOWTrainer::cluster . . . 783
cv::BOWKMeansTrainer . . . 783
cv::BOWImgDescriptorExtractor . . . 784
cv::BOWImgDescriptorExtractor::BOWImgDescriptorExtractor . . . 785
cv::BOWImgDescriptorExtractor::setVocabulary . . . 785
cv::BOWImgDescriptorExtractor::getVocabulary . . . 785
cv::BOWImgDescriptorExtractor::compute . . . 786

cv::BOWImgDescriptorExtractor::descriptorSize . . . 786
cv::BOWImgDescriptorExtractor::descriptorType . . . 786

14 flann. Clustering and Search in Multi-Dimensional Spaces 787
14.1 Fast Approximate Nearest Neighbor Search . . . 787
cv::flann::Index_ . . . 787
cvflann::Index_< T >::Index_ . . . 788
cv::flann::Index_< T >::knnSearch . . . 791
cv::flann::Index_< T >::radiusSearch . . . 792
cv::flann::Index_< T >::save . . . 793
cv::flann::Index_< T >::getIndexParameters . . . 793
14.2 Clustering . . . 793
cv::cv::flann::hierarchicalClustering¡ET,DT¿ . . . 793

15 objdetect. Object Detection 795
15.1 Cascade Classification . . . 795
cv::FeatureEvaluator . . . 795
cv::FeatureEvaluator::read . . . 795
cv::FeatureEvaluator::clone . . . 796
cv::FeatureEvaluator::getFeatureType . . . 796
cv::FeatureEvaluator::setImage . . . 796
cv::FeatureEvaluator::setWindow . . . 797
cv::FeatureEvaluator::calcOrd . . . 797
cv::FeatureEvaluator::calcCat . . . 797
cv::FeatureEvaluator::create . . . 798
cv::CascadeClassifier . . . 798
cv::CascadeClassifier::CascadeClassifier . . . 799
cv::CascadeClassifier::empty . . . 800
cv::CascadeClassifier::load . . . 800
cv::CascadeClassifier::read . . . 800
cv::CascadeClassifier::detectMultiScale . . . 800
cv::CascadeClassifier::setImage . . . 801
cv::CascadeClassifier::runAt . . . 801
cv::groupRectangles . . . 802

16 video. Video Analysis 803
16.1 Motion Analysis and Object Tracking . . . 803
cv::calcOpticalFlowPyrLK . . . 803
cv::calcOpticalFlowFarneback . . . 804
cv::updateMotionHistory . . . 805

cv::calcMotionGradient . . . 806
cv::calcGlobalOrientation . . . 807
cv::CamShift . . . 807
cv::meanShift . . . 808
cv::KalmanFilter . . . 809

17 highgui. High-level GUI and Media I/O 811
17.1 User Interface . . . 811
cv::createTrackbar . . . 811
cv::getTrackbarPos . . . 812
cv::imshow . . . 813
cv::namedWindow . . . 813
cv::setTrackbarPos . . . 815
cv::waitKey . . . 815
17.2 Reading and Writing Images and Video . . . 816
cv::imdecode . . . 816
cv::imencode . . . 816
cv::imread . . . 817
cv::imwrite . . . 818
cv::VideoCapture . . . 819
cv::VideoCapture::VideoCapture . . . 820
cv::VideoCapture::get . . . 821
cv::VideoCapture::set . . . 822
cv::VideoWriter . . . 823
17.3 Qt new functions . . . 824
cv::setWindowProperty . . . 825
cv::getWindowProperty . . . 826
cv::fontQt . . . 827
cv::addText . . . 828
cv::displayOverlay . . . 828
cv::displayStatusBar . . . 829
cv::createOpenGLCallback . . . 829
cv::saveWindowParameters . . . 831
cv::loadWindowParameters . . . 831
cv::createButton . . . 831

18 calib3d. Camera Calibration, Pose Estimation and Stereo 833
18.1 Camera Calibration and 3d Reconstruction . . . 833
cv::calibrateCamera . . . 835
cv::calibrationMatrixValues . . . 837

cv::composeRT . . . 838
cv::computeCorrespondEpilines . . . 839
cv::convertPointsHomogeneous . . . 840
cv::decomposeProjectionMatrix . . . 840
cv::drawChessboardCorners . . . 841
cv::findChessboardCorners . . . 842
cv::findCirclesGrid . . . 843
cv::solvePnP . . . 844
cv::findFundamentalMat . . . 845
cv::findHomography . . . 847
cv::getDefaultNewCameraMatrix . . . 848
cv::getOptimalNewCameraMatrix . . . 849
cv::initCameraMatrix2D . . . 850
cv::initUndistortRectifyMap . . . 851
cv::matMulDeriv . . . 852
cv::projectPoints . . . 853
cv::reprojectImageTo3D . . . 854
cv::RQDecomp3x3 . . . 855
cv::Rodrigues . . . 856
cv::StereoBM . . . 857
cv::StereoSGBM . . . 857
cv::StereoSGBM::StereoSGBM . . . 858
cv::StereoSGBM::operator () . . . 860
cv::stereoCalibrate . . . 860
cv::stereoRectify . . . 863
cv::stereoRectifyUncalibrated . . . 866
cv::undistort . . . 867
cv::undistortPoints . . . 868

19 ml. Machine Learning **871**
19.1 Statistical Models . . . 871
cv::CvStatModel . . . 871
CvStatModel::CvStatModel . . . 872
CvStatModel::CvStatModel(...) . . . 872
CvStatModel:: CvStatModel . . . 873
CvStatModel::clear . . . 873
CvStatModel::save . . . 873
CvStatModel::load . . . 874
CvStatModel::write . . . 874
CvStatModel::read . . . 874

CvStatModel::train . . . 875
CvStatModel::predict . . . 876
19.2 Normal Bayes Classifier . . . 876
cv::CvNormalBayesClassifier . . . 877
CvNormalBayesClassifier::train . . . 877
CvNormalBayesClassifier::predict . . . 878
19.3 K Nearest Neighbors . . . 878
cv::CvKNearest . . . 878
CvKNearest::train . . . 879
CvKNearest::find_nearest . . . 880
19.4 Support Vector Machines . . . 882
cv::CvSVM . . . 883
cv::CvSVMParams . . . 884
CvSVM::train . . . 885
CvSVM::train_auto . . . 885
CvSVM::get_default_grid . . . 886
CvSVM::get_params . . . 887
CvSVM::get_support_vector* . . . 887
19.5 Decision Trees . . . 888
Predicting with Decision Trees . . . 888
Training Decision Trees . . . 888
Variable importance . . . 889
cv::CvDTreeSplit . . . 889
cv::CvDTreeNode . . . 890
cv::CvDTreeParams . . . 890
cv::CvDTreeTrainData . . . 891
cv::CvDTree . . . 894
CvDTree::train . . . 896
CvDTree::predict . . . 896
19.6 Boosting . . . 897
cv::CvBoostParams . . . 899
cv::CvBoostTree . . . 899
cv::CvBoost . . . 900
CvBoost::train . . . 901
CvBoost::predict . . . 902
CvBoost::prune . . . 902
CvBoost::get_weak_predictors . . . 902
19.7 Random Trees . . . 903
cv::CvRTParams . . . 904
cv::CvRTrees . . . 904

CvRTrees::train . . . 905
CvRTrees::predict . . . 906
CvRTrees::get_var_importance . . . 906
CvRTrees::get_proximity . . . 907
19.8 Expectation-Maximization . . . 910
cv::CvEMParams . . . 911
cv::CvEM . . . 912
CvEM::train . . . 913
19.9 Neural Networks . . . 916
cv::CvANN_MLP_TrainParams . . . 919
cv::CvANN_MLP . . . 920
CvANN_MLP::create . . . 922
CvANN_MLP::train . . . 922

20 gpu. GPU-based Functionality 925
20.1 Initalization and Info . . . 925
cv::gpu::hasNativeDoubleSupport . . . 925
cv::gpu::hasAtomicsSupport . . . 925
20.2 Data Structures . . . 926
cv::gpu::createContinuous . . . 926
20.3 Operations on Matrices . . . 926
cv::gpu::merge . . . 926
cv::gpu::split . . . 927
cv::gpu::bitwise_not . . . 927
cv::gpu::bitwise_or . . . 928
cv::gpu::bitwise_and . . . 928
cv::gpu::bitwise_xor . . . 929
20.4 Image Processing . . . 929
cv::gpu::meanShiftFiltering . . . 929
cv::gpu::meanShiftProc . . . 930
cv::gpu::meanShiftSegmentation . . . 930
cv::gpu::integral . . . 931
cv::gpu::sqrIntegral . . . 931
cv::gpu::columnSum . . . 932
cv::gpu::cornerHarris . . . 932
cv::gpu::cornerMinEigenVal . . . 933
cv::gpu::mulSpectrums . . . 933
cv::gpu::mulAndScaleSpectrums . . . 934
cv::gpu::dft . . . 934
cv::gpu::convolve . . . 935

cv::gpu::ConvolveBuf . . . 936
cv::gpu::ConvolveBuf::ConvolveBuf . . . 936
cv::gpu::matchTemplate . . . 937
20.5 Matrix Reductions . . . 938
cv::gpu::sum . . . 938
cv::gpu::sqrSum . . . 938
cv::gpu::minMax . . . 938
cv::gpu::minMaxLoc . . . 939
cv::gpu::countNonZero . . . 940
20.6 Object Detection . . . 940
cv::gpu::HOGDescriptor . . . 940
cv::gpu::HOGDescriptor::HOGDescriptor . . . 941
cv::gpu::HOGDescriptor::getDescriptorSize . . . 942
cv::gpu::HOGDescriptor::getBlockHistogramSize . . . 942
cv::gpu::HOGDescriptor::setSVMDetector . . . 942
cv::gpu::HOGDescriptor::getDefaultPeopleDetector . . . 943
cv::gpu::HOGDescriptor::getPeopleDetector48x96 . . . 943
cv::gpu::HOGDescriptor::getPeopleDetector64x128 . . . 943
cv::gpu::HOGDescriptor::detect . . . 943
cv::gpu::HOGDescriptor::detectMultiScale . . . 944
cv::gpu::HOGDescriptor::getDescriptors . . . 944
20.7 Feature detection and description . . . 945
cv::gpu::SURFParams_GPU . . . 945
cv::gpu::SURF_GPU . . . 946
cv::gpu::BruteForceMatcher_GPU . . . 948
gpu::BruteForceMatcher_GPU::match . . . 950
gpu::BruteForceMatcher_GPU::matchSingle . . . 950
gpu::BruteForceMatcher_GPU::matchCollection . . . 951
gpu::BruteForceMatcher_GPU::makeGpuCollection . . . 952
gpu::BruteForceMatcher_GPU::matchDownload . . . 952
gpu::BruteForceMatcher_GPU::knnMatch . . . 953
gpu::BruteForceMatcher_GPU::knnMatch . . . 953
gpu::BruteForceMatcher_GPU::knnMatchDownload . . . 954
gpu::BruteForceMatcher_GPU::radiusMatch . . . 955
gpu::BruteForceMatcher_GPU::radiusMatch . . . 955
gpu::BruteForceMatcher_GPU::radiusMatchDownload . . . 956
20.8 Image Filtering . . . 957
cv::gpu::BaseRowFilter_GPU . . . 957
cv::gpu::BaseColumnFilter_GPU . . . 957
cv::gpu::BaseFilter_GPU . . . 958

cv::gpu::FilterEngine_GPU . . . 958
gpu::createFilter2D_GPU . . . 959
gpu::createSeparableFilter_GPU . . . 959
gpu::getRowSumFilter_GPU . . . 960
gpu::getColumnSumFilter_GPU . . . 960
gpu::createBoxFilter_GPU . . . 960
cv::gpu::boxFilter . . . 961
cv::gpu::blur . . . 961
gpu::createMorphologyFilter_GPU . . . 961
cv::gpu::erode . . . 962
cv::gpu::dilate . . . 963
cv::gpu::morphologyEx . . . 963
gpu::createLinearFilter_GPU . . . 963
cv::gpu::filter2D . . . 964
cv::gpu::Laplacian . . . 964
gpu::getLinearRowFilter_GPU . . . 964
gpu::getLinearColumnFilter_GPU . . . 965
gpu::createSeparableLinearFilter_GPU . . . 965
cv::gpu::sepFilter2D . . . 966
gpu::createDerivFilter_GPU . . . 966
cv::gpu::Sobel . . . 967
cv::gpu::Scharr . . . 967
gpu::createGaussianFilter_GPU . . . 967
cv::gpu::GaussianBlur . . . 968
gpu::getMaxFilter_GPU . . . 968
gpu::getMinFilter_GPU . . . 968

III Python API Reference **969**

21 Introduction **971**
21.1 Cookbook . . . 971
Convert an image . . . 971
Resize an image . . . 972
Compute the Laplacian . . . 972
Using GoodFeaturesToTrack . . . 972
Using GetSubRect . . . 972
Using CreateMat, and accessing an element . . . 973
ROS image message to OpenCV . . . 973
PIL Image to OpenCV . . . 973

OpenCV to PIL Image . . . 973
NumPy and OpenCV . . . 974
OpenCV to pygame . . . 974
OpenCV and OpenEXR . . . 975

22 core. The Core Functionality 977
22.1 Basic Structures . . . 977
CvPoint . . . 977
CvPoint2D32f . . . 977
CvPoint3D32f . . . 977
CvPoint2D64f . . . 977
CvPoint3D64f . . . 977
CvSize . . . 978
CvSize2D32f . . . 978
CvRect . . . 978
CvScalar . . . 978
CvTermCriteria . . . 978
CvMat . . . 979
CvMatND . . . 979
IplImage . . . 979
CvArr . . . 980
22.2 Operations on Arrays . . . 980
cv::AbsDiff . . . 980
cv::AbsDiffS . . . 981
cv::Add . . . 981
cv::AddS . . . 982
cv::AddWeighted . . . 982
cv::And . . . 983
cv::AndS . . . 983
cv::Avg . . . 984
cv::AvgSdv . . . 984
cv::CalcCovarMatrix . . . 985
cv::CartToPolar . . . 986
cv::Cbrt . . . 987
cv::ClearND . . . 987
cv::CloneImage . . . 988
cv::CloneMat . . . 988
cv::CloneMatND . . . 988
cv::Cmp . . . 988
cv::CmpS . . . 989

cv::Convert . . . 990
cv::ConvertScale . . . 991
cv::ConvertScaleAbs . . . 991
cv::CvtScaleAbs . . . 992
cv::Copy . . . 992
cv::CountNonZero . . . 993
cv::CreateData . . . 993
cv::CreateImage . . . 994
cv::CreateImageHeader . . . 994
cv::CreateMat . . . 995
cv::CreateMatHeader . . . 995
cv::CreateMatND . . . 995
cv::CreateMatNDHeader . . . 996
cv::CrossProduct . . . 996
CvtPixToPlane . . . 997
cv::DCT . . . 997
cv::DFT . . . 998
cv::Det . . . 1000
cv::Div . . . 1000
cv::DotProduct . . . 1001
cv::EigenVV . . . 1001
cv::Exp . . . 1002
cv::FastArctan . . . 1002
cv::Flip . . . 1003
cv::fromarray . . . 1004
cv::GEMM . . . 1005
cv::Get1D . . . 1005
cv::Get2D . . . 1006
cv::Get3D . . . 1006
cv::GetND . . . 1007
cv::GetCol . . . 1007
cv::GetCols . . . 1007
cv::GetDiag . . . 1008
cv::GetDims . . . 1008
cv::GetElemType . . . 1009
cv::GetImage . . . 1009
cv::GetImageCOI . . . 1009
cv::GetImageROI . . . 1010
cv::GetMat . . . 1010
cv::GetOptimalDFTSize . . . 1011

cv::GetReal1D . . . 1011
cv::GetReal2D . . . 1012
cv::GetReal3D . . . 1012
cv::GetRealND . . . 1013
cv::GetRow . . . 1013
cv::GetRows . . . 1013
cv::GetSize . . . 1014
cv::GetSubRect . . . 1014
cv::InRange . . . 1015
cv::InRangeS . . . 1015
cv::InvSqrt . . . 1016
cv::Inv . . . 1016
cv::Invert . . . 1017
cv::IsInf . . . 1017
cv::IsNaN . . . 1018
cv::LUT . . . 1018
cv::Log . . . 1019
cv::Mahalanobis . . . 1019
cv::Max . . . 1020
cv::MaxS . . . 1020
cv::Merge . . . 1021
cv::Min . . . 1021
cv::MinMaxLoc . . . 1022
cv::MinS . . . 1022
Mirror . . . 1023
cv::MixChannels . . . 1023
MulAddS . . . 1024
cv::Mul . . . 1024
cv::MulSpectrums . . . 1024
cv::MulTransposed . . . 1025
cv::Norm . . . 1026
cv::Not . . . 1027
cv::Or . . . 1027
cv::OrS . . . 1028
cv::PerspectiveTransform . . . 1028
cv::PolarToCart . . . 1029
cv::Pow . . . 1030
cv::RNG . . . 1030
cv::RandArr . . . 1031
cv::RandInt . . . 1031

cv::RandReal . . . 1032
cv::Reduce . . . 1032
cv::Repeat . . . 1033
cv::ResetImageROI . . . 1033
cv::Reshape . . . 1034
cv::ReshapeMatND . . . 1034
cv::Round . . . 1035
cv::Floor . . . 1035
cv::Ceil . . . 1036
cv::ScaleAdd . . . 1036
cv::Set . . . 1036
cv::Set1D . . . 1037
cv::Set2D . . . 1037
cv::Set3D . . . 1038
cv::SetND . . . 1038
cv::SetData . . . 1039
cv::SetIdentity . . . 1039
cv::SetImageCOI . . . 1040
cv::SetImageROI . . . 1040
cv::SetReal1D . . . 1041
cv::SetReal2D . . . 1041
cv::SetReal3D . . . 1041
cv::SetRealND . . . 1042
cv::SetZero . . . 1042
cv::Solve . . . 1043
cv::SolveCubic . . . 1043
cv::Split . . . 1044
cv::Sqrt . . . 1045
cv::Sub . . . 1045
cv::SubRS . . . 1045
cv::SubS . . . 1046
cv::Sum . . . 1047
cv::SVBkSb . . . 1047
cv::SVD . . . 1048
cv::Trace . . . 1049
cv::Transform . . . 1050
cv::Transpose . . . 1050
cv::Xor . . . 1051
cv::XorS . . . 1051
cv::mGet . . . 1052

cv::mSet . . . 1052
22.3 Dynamic Structures . . . 1053
CvMemStorage . . . 1053
CvSeq . . . 1053
CvSet . . . 1054
cv::CloneSeq . . . 1054
cv::CreateMemStorage . . . 1054
cv::SeqInvert . . . 1055
cv::SeqRemove . . . 1055
cv::SeqRemoveSlice . . . 1055
22.4 Drawing Functions . . . 1056
cv::Circle . . . 1056
cv::ClipLine . . . 1057
cv::DrawContours . . . 1057
cv::Ellipse . . . 1058
cv::EllipseBox . . . 1059
cv::FillConvexPoly . . . 1060
cv::FillPoly . . . 1060
cv::GetTextSize . . . 1061
cv::InitFont . . . 1061
cv::InitLineIterator . . . 1062
cv::Line . . . 1063
cv::PolyLine . . . 1064
cv::PutText . . . 1065
cv::Rectangle . . . 1065
CV_RGB . . . 1066
22.5 XML/YAML Persistence . . . 1066
cv::Load . . . 1066
cv::Save . . . 1067
22.6 Clustering . . . 1067
cv::KMeans2 . . . 1067
22.7 Utility and System Functions and Macros . . . 1068
Error Handling . . . 1068
cv::GetTickCount . . . 1068
cv::GetTickFrequency . . . 1069

23 imgproc. Image Processing 1071
23.1 Histograms . . . 1071
CvHistogram . . . 1071
cv::CalcBackProject . . . 1071

cv::CalcBackProjectPatch . . . 1072
cv::CalcHist . . . 1073
cv::CalcProbDensity . . . 1075
cv::ClearHist . . . 1075
cv::CompareHist . . . 1076
cv::CreateHist . . . 1077
cv::GetMinMaxHistValue . . . 1078
cv::NormalizeHist . . . 1079
QueryHistValue_1D . . . 1079
QueryHistValue_2D . . . 1079
QueryHistValue_3D . . . 1080
QueryHistValue_nD . . . 1080
cv::ThreshHist . . . 1080
23.2 Image Filtering . . . 1081
IplConvKernel . . . 1081
cv::CopyMakeBorder . . . 1081
cv::CreateStructuringElementEx . . . 1082
cv::Dilate . . . 1083
cv::Erode . . . 1084
cv::Filter2D . . . 1084
cv::Laplace . . . 1085
cv::MorphologyEx . . . 1086
cv::PyrDown . . . 1087
cv::Smooth . . . 1087
cv::Sobel . . . 1089
23.3 Geometric Image Transformations . . . 1090
cv::GetRotationMatrix2D . . . 1091
cv::GetAffineTransform . . . 1092
cv::GetPerspectiveTransform . . . 1092
cv::GetQuadrangleSubPix . . . 1093
cv::GetRectSubPix . . . 1094
cv::LogPolar . . . 1094
cv::Remap . . . 1095
cv::Resize . . . 1096
cv::WarpAffine . . . 1096
cv::WarpPerspective . . . 1097
23.4 Miscellaneous Image Transformations . . . 1098
cv::AdaptiveThreshold . . . 1098
cv::CvtColor . . . 1100
cv::DistTransform . . . 1104

CvConnectedComp . . . 1106
cv::FloodFill . . . 1106
cv::Inpaint . . . 1108
cv::Integral . . . 1109
cv::PyrMeanShiftFiltering . . . 1110
cv::PyrSegmentation . . . 1111
cv::Threshold . . . 1112
23.5 Structural Analysis and Shape Descriptors . . . 1114
cv::ApproxChains . . . 1114
cv::ApproxPoly . . . 1114
cv::ArcLength . . . 1115
cv::BoundingRect . . . 1116
cv::BoxPoints . . . 1116
cv::CalcPGH . . . 1117
cv::CalcEMD2 . . . 1117
cv::CheckContourConvexity . . . 1118
CvConvexityDefect . . . 1119
cv::ContourArea . . . 1119
cv::ContourFromContourTree . . . 1120
cv::ConvexHull2 . . . 1121
cv::ConvexityDefects . . . 1121
cv::CreateContourTree . . . 1122
cv::FindContours . . . 1122
cv::FitEllipse2 . . . 1124
cv::FitLine . . . 1124
cv::GetCentralMoment . . . 1125
cv::GetHuMoments . . . 1126
cv::GetNormalizedCentralMoment . . . 1127
cv::GetSpatialMoment . . . 1127
cv::MatchContourTrees . . . 1128
cv::MatchShapes . . . 1128
cv::MinAreaRect2 . . . 1129
cv::MinEnclosingCircle . . . 1130
cv::Moments . . . 1131
cv::PointPolygonTest . . . 1131
23.6 Planar Subdivisions . . . 1132
CvSubdiv2D . . . 1132
CvSubdiv2DPoint . . . 1133
cv::CalcSubdivVoronoi2D . . . 1133
cv::ClearSubdivVoronoi2D . . . 1134

cv::CreateSubdivDelaunay2D . . . 1134
cv::FindNearestPoint2D . . . 1134
cv::Subdiv2DEdgeDst . . . 1135
cv::Subdiv2DGetEdge . . . 1135
cv::Subdiv2DNextEdge . . . 1136
cv::Subdiv2DLocate . . . 1137
cv::Subdiv2DRotateEdge . . . 1138
cv::SubdivDelaunay2DInsert . . . 1139
23.7 Motion Analysis and Object Tracking . . . 1139
cv::Acc . . . 1139
cv::MultiplyAcc . . . 1140
cv::RunningAvg . . . 1140
cv::SquareAcc . . . 1141
23.8 Feature Detection . . . 1141
cv::Canny . . . 1141
cv::CornerEigenValsAndVecs . . . 1142
cv::CornerHarris . . . 1142
cv::CornerMinEigenVal . . . 1143
cv::FindCornerSubPix . . . 1144
cv::GoodFeaturesToTrack . . . 1145
cv::HoughLines2 . . . 1146
cv::PreCornerDetect . . . 1147
23.9 Object Detection . . . 1148
cv::MatchTemplate . . . 1148

24 features2d. Feature Detection and Descriptor Extraction 1151
24.1 Feature detection and description . . . 1151
CvSURFPoint . . . 1151
cv::ExtractSURF . . . 1152
cv::GetStarKeypoints . . . 1153

25 flann. Clustering and Search in Multi-Dimensional Spaces 1155
25.1 Fast Approximate Nearest Neighbor Search . . . 1155

26 objdetect. Object Detection 1157
26.1 Cascade Classification . . . 1157
Haar Feature-based Cascade Classifier for Object Detection . . . 1157
cv::HaarDetectObjects . . . 1158

27 video. Video Analysis **1161**
27.1 Motion Analysis and Object Tracking . . . 1161
cv::CalcGlobalOrientation . . . 1161
cv::CalcMotionGradient . . . 1162
cv::CalcOpticalFlowBM . . . 1163
cv::CalcOpticalFlowHS . . . 1163
cv::CalcOpticalFlowLK . . . 1164
cv::CalcOpticalFlowPyrLK . . . 1165
cv::CamShift . . . 1166
CvKalman . . . 1167
cv::CreateKalman . . . 1168
cv::KalmanCorrect . . . 1168
cv::KalmanPredict . . . 1169
KalmanUpdateByMeasurement . . . 1170
KalmanUpdateByTime . . . 1170
cv::MeanShift . . . 1170
cv::SegmentMotion . . . 1170
cv::SnakeImage . . . 1171
cv::UpdateMotionHistory . . . 1172

28 highgui. High-level GUI and Media I/O **1173**
28.1 User Interface . . . 1173
cv::CreateTrackbar . . . 1173
cv::DestroyAllWindows . . . 1174
cv::DestroyWindow . . . 1174
cv::GetTrackbarPos . . . 1175
cv::MoveWindow . . . 1175
cv::NamedWindow . . . 1176
cv::ResizeWindow . . . 1177
cv::SetMouseCallback . . . 1177
cv::SetTrackbarPos . . . 1178
cv::ShowImage . . . 1179
cv::WaitKey . . . 1179
28.2 Reading and Writing Images and Video . . . 1180
cv::LoadImage . . . 1180
cv::LoadImageM . . . 1181
cv::SaveImage . . . 1182
CvCapture . . . 1182
cv::CaptureFromCAM . . . 1182
cv::CaptureFromFile . . . 1183

cv::GetCaptureProperty . . . 1183
cv::GrabFrame . . . 1184
cv::QueryFrame . . . 1185
cv::RetrieveFrame . . . 1185
cv::SetCaptureProperty . . . 1185
cv::CreateVideoWriter . . . 1187
cv::WriteFrame . . . 1187

29 calib3d. Camera Calibration, Pose Estimation and Stereo 1189
29.1 Camera Calibration and 3d Reconstruction . . . 1189
cv::CalibrateCamera2 . . . 1191
cv::ComputeCorrespondEpilines . . . 1193
cv::ConvertPointsHomogeneous . . . 1194
cv::CreatePOSITObject . . . 1194
cv::CreateStereoBMState . . . 1195
cv::CreateStereoGCState . . . 1195
CvStereoBMState . . . 1196
CvStereoGCState . . . 1197
cv::DecomposeProjectionMatrix . . . 1198
cv::DrawChessboardCorners . . . 1199
cv::FindChessboardCorners . . . 1199
cv::FindExtrinsicCameraParams2 . . . 1201
cv::FindFundamentalMat . . . 1201
cv::FindHomography . . . 1203
cv::FindStereoCorrespondenceBM . . . 1204
cv::FindStereoCorrespondenceGC . . . 1205
cv::GetOptimalNewCameraMatrix . . . 1207
cv::InitIntrinsicParams2D . . . 1208
cv::InitUndistortMap . . . 1209
cv::InitUndistortRectifyMap . . . 1209
cv::POSIT . . . 1211
cv::ProjectPoints2 . . . 1212
cv::ReprojectImageTo3D . . . 1213
cv::RQDecomp3x3 . . . 1214
cv::Rodrigues2 . . . 1214
cv::StereoCalibrate . . . 1215
cv::StereoRectify . . . 1218
cv::StereoRectifyUncalibrated . . . 1221
cv::Undistort2 . . . 1222
cv::UndistortPoints . . . 1223

30 ml. Machine Learning 1225

Bibliography 1225

Index 1225

Part I

C API Reference

Chapter 1

core. The Core Functionality

1.1 Basic Structures

CvPoint (view/add comments)

2D point with integer coordinates (usually zero-based).

```
typedef struct CvPoint
{
    int x;
    int y;
}
CvPoint;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint cvPoint( int x, int y );

/* Conversion from CvPoint2D32f */
inline CvPoint cvPointFrom32f( CvPoint2D32f point );
```

CvPoint2D32f (view/add comments)

2D point with floating-point coordinates

```
typedef struct CvPoint2D32f
{
    float x;
    float y;
}
CvPoint2D32f;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint2D32f cvPoint2D32f( double x, double y );

/* Conversion from CvPoint */
inline CvPoint2D32f cvPointTo32f( CvPoint point );
```

CvPoint3D32f (view/add comments)

3D point with floating-point coordinates

```
typedef struct CvPoint3D32f
{
    float x;
    float y;
    float z;
}
CvPoint3D32f;
```

x x-coordinate

y y-coordinate

z z-coordinate

```
/* Constructor */
inline CvPoint3D32f cvPoint3D32f( double x, double y, double z );
```

CvPoint2D64f (view/add comments)

2D point with double precision floating-point coordinates

```
typedef struct CvPoint2D64f
{
    double x;
    double y;
}
CvPoint2D64f;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint2D64f cvPoint2D64f( double x, double y );

/* Conversion from CvPoint */
inline CvPoint2D64f cvPointTo64f( CvPoint point );
```

CvPoint3D64f (view/add comments)

3D point with double precision floating-point coordinates

```
typedef struct CvPoint3D64f
{
    double x;
    double y;
    double z;
}
CvPoint3D64f;
```

x x-coordinate

y y-coordinate

z z-coordinate

```
/* Constructor */
inline CvPoint3D64f cvPoint3D64f( double x, double y, double z );
```

CvSize (view/add comments)

Pixel-accurate size of a rectangle.

```
typedef struct CvSize
{
    int width;
    int height;
}
CvSize;
```

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvSize cvSize( int width, int height );
```

CvSize2D32f (view/add comments)

Sub-pixel accurate size of a rectangle.

```
typedef struct CvSize2D32f
{
    float width;
    float height;
}
CvSize2D32f;
```

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvSize2D32f cvSize2D32f( double width, double height );
```

CvRect (view/add comments)

Offset (usually the top-left corner) and size of a rectangle.

```
typedef struct CvRect
{
    int x;
    int y;
    int width;
    int height;
}
CvRect;
```

x x-coordinate of the top-left corner

y y-coordinate of the top-left corner (bottom-left for Windows bitmaps)

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvRect cvRect( int x, int y, int width, int height );
```

CvScalar (view/add comments)

A container for 1-,2-,3- or 4-tuples of doubles.

```
typedef struct CvScalar
{
    double val[4];
}
CvScalar;
```

```
/* Constructor:
initializes val[0] with val0, val[1] with val1, etc.
*/
inline CvScalar cvScalar( double val0, double val1=0,
                          double val2=0, double val3=0 );
/* Constructor:
initializes all of val[0]...val[3] with val0123
*/
inline CvScalar cvScalarAll( double val0123 );

/* Constructor:
initializes val[0] with val0, and all of val[1]...val[3] with zeros
*/
inline CvScalar cvRealScalar( double val0 );
```

CvTermCriteria (view/add comments)

Termination criteria for iterative algorithms.

```
#define CV_TERMCRIT_ITER    1
#define CV_TERMCRIT_NUMBER  CV_TERMCRIT_ITER
#define CV_TERMCRIT_EPS     2

typedef struct CvTermCriteria
{
    int    type;
    int    max_iter;
    double epsilon;
}
CvTermCriteria;
```

type A combination of CV_TERMCRIT_ITER and CV_TERMCRIT_EPS

max_iter Maximum number of iterations

epsilon Required accuracy

```
/* Constructor */
inline CvTermCriteria cvTermCriteria( int type, int max_iter, double epsilon );

/* Check and transform a CvTermCriteria so that
   type=CV_TERMCRIT_ITER+CV_TERMCRIT_EPS
   and both max_iter and epsilon are valid */
CvTermCriteria cvCheckTermCriteria( CvTermCriteria criteria,
                                    double default_eps,
                                    int default_max_iters );
```

CvMat (view/add comments)

A multi-channel matrix.

```
typedef struct CvMat
{
    int type;
    int step;

    int* refcount;

    union
```

```
    {
        uchar* ptr;
        short* s;
        int* i;
        float* fl;
        double* db;
    } data;

#ifdef __cplusplus
    union
    {
        int rows;
        int height;
    };

    union
    {
        int cols;
        int width;
    };
#else
    int rows;
    int cols;
#endif

} CvMat;
```

type A CvMat signature (CV_MAT_MAGIC_VAL) containing the type of elements and flags

step Full row length in bytes

refcount Underlying data reference counter

data Pointers to the actual matrix data

rows Number of rows

cols Number of columns

Matrices are stored row by row. All of the rows are aligned by 4 bytes.

CvMatND (view/add comments)

Multi-dimensional dense multi-channel array.

```
typedef struct CvMatND
{
    int type;
    int dims;

    int* refcount;

    union
    {
        uchar* ptr;
        short* s;
        int* i;
        float* fl;
        double* db;
    } data;

    struct
    {
        int size;
        int step;
    }
    dim[CV_MAX_DIM];

} CvMatND;
```

type A CvMatND signature (CV_MATND_MAGIC_VAL), combining the type of elements and flags

dims The number of array dimensions

refcount Underlying data reference counter

data Pointers to the actual matrix data

dim For each dimension, the pair (number of elements, distance between elements in bytes)

CvSparseMat (view/add comments)

Multi-dimensional sparse multi-channel array.

```
typedef struct CvSparseMat
{
    int type;
    int dims;
    int* refcount;
```

```
    struct CvSet* heap;
    void** hashtable;
    int hashsize;
    int total;
    int valoffset;
    int idxoffset;
    int size[CV_MAX_DIM];

} CvSparseMat;
```

type A CvSparseMat signature (CV_SPARSE_MAT_MAGIC_VAL), combining the type of elements and flags.

dims Number of dimensions

refcount Underlying reference counter. Not used.

heap A pool of hash table nodes

hashtable The hash table. Each entry is a list of nodes.

hashsize Size of the hash table

total Total number of sparse array nodes

valoffset The value offset of the array nodes, in bytes

idxoffset The index offset of the array nodes, in bytes

size Array of dimension sizes

IplImage (view/add comments)

IPL image header

```
typedef struct _IplImage
{
    int  nSize;
    int  ID;
    int  nChannels;
    int  alphaChannel;
    int  depth;
    char colorModel[4];
    char channelSeq[4];
```

```
    int  dataOrder;
    int  origin;
    int  align;
    int  width;
    int  height;
    struct _IplROI *roi;
    struct _IplImage *maskROI;
    void  *imageId;
    struct _IplTileInfo *tileInfo;
    int  imageSize;
    char *imageData;
    int  widthStep;
    int  BorderMode[4];
    int  BorderConst[4];
    char *imageDataOrigin;
}
IplImage;
```

nSize `sizeof(IplImage)`

ID Version, always equals 0

nChannels Number of channels. Most OpenCV functions support 1-4 channels.

alphaChannel Ignored by OpenCV

depth Channel depth in bits + the optional sign bit (`IPL_DEPTH_SIGN`). The supported depths are:

- **IPL_DEPTH_8U** Unsigned 8-bit integer
- **IPL_DEPTH_8S** Signed 8-bit integer
- **IPL_DEPTH_16U** Unsigned 16-bit integer
- **IPL_DEPTH_16S** Signed 16-bit integer
- **IPL_DEPTH_32S** Signed 32-bit integer
- **IPL_DEPTH_32F** Single-precision floating point
- **IPL_DEPTH_64F** Double-precision floating point

colorModel Ignored by OpenCV. The OpenCV function CvtColor requires the source and destination color spaces as parameters.

channelSeq Ignored by OpenCV

dataOrder 0 = `IPL_DATA_ORDER_PIXEL` - interleaved color channels, 1 - separate color channels. CreateImage only creates images with interleaved channels. For example, the usual layout of a color image is: $b_{00}g_{00}r_{00}b_{10}g_{10}r_{10}...$

origin 0 - top-left origin, 1 - bottom-left origin (Windows bitmap style)

align Alignment of image rows (4 or 8). OpenCV ignores this and uses widthStep instead.

width Image width in pixels

height Image height in pixels

roi Region Of Interest (ROI). If not NULL, only this image region will be processed.

maskROI Must be NULL in OpenCV

imageId Must be NULL in OpenCV

tileInfo Must be NULL in OpenCV

imageSize Image data size in bytes. For interleaved data, this equals `image->height·image->widthStep`

imageData A pointer to the aligned image data

widthStep The size of an aligned image row, in bytes

BorderMode Border completion mode, ignored by OpenCV

BorderConst Border completion mode, ignored by OpenCV

imageDataOrigin A pointer to the origin of the image data (not necessarily aligned). This is used for image deallocation.

The IplImage structure was inherited from the Intel Image Processing Library, in which the format is native. OpenCV only supports a subset of possible IplImage formats, as outlined in the parameter list above.

In addition to the above restrictions, OpenCV handles ROIs differently. OpenCV functions require that the image size or ROI size of all source and destination images match exactly. On the other hand, the Intel Image Processing Library processes the area of intersection between the source and destination images (or ROIs), allowing them to vary independently.

CvArr (view/add comments)

Arbitrary array

```
typedef void CvArr;
```

The metatype `CvArr` is used *only* as a function parameter to specify that the function accepts arrays of multiple types, such as IplImage*, CvMat* or even CvSeq* sometimes. The particular array type is determined at runtime by analyzing the first 4 bytes of the header.

1.2 Operations on Arrays

cvAbsDiff (view/add comments)

Calculates absolute difference between two arrays.

```
void cvAbsDiff(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates absolute difference between two arrays.

$$\texttt{dst}(i)_c = |\texttt{src1}(I)_c - \texttt{src2}(I)_c|$$

All the arrays must have the same data type and the same size (or ROI size).

cvAbsDiffS (view/add comments)

Calculates absolute difference between an array and a scalar.

```
void cvAbsDiffS(const CvArr* src, CvArr* dst, CvScalar value);
```

```
#define cvAbs(src, dst) cvAbsDiffS(src, dst, cvScalarAll(0))
```

src The source array

dst The destination array

value The scalar

The function calculates absolute difference between an array and a scalar.

$$\texttt{dst}(i)_c = |\texttt{src}(I)_c - \texttt{value}_c|$$

All the arrays must have the same data type and the same size (or ROI size).

---

cvAdd (view/add comments)

Computes the per-element sum of two arrays.

```
void cvAdd(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds one array to another:

```
dst(I)=src1(I)+src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvAddS (view/add comments)

Computes the sum of an array and a scalar.

```
void cvAddS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Added scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds a scalar `value` to every element in the source array `src1` and stores the result in `dst`. For types that have limited range this operation is saturating.

```
dst(I)=src(I)+value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size).

---

cvAddWeighted (view/add comments)

Computes the weighted sum of two arrays.

```
void cvAddWeighted(const CvArr* src1, double alpha, const CvArr* src2,
double beta, double gamma, CvArr* dst);
```

src1 The first source array

alpha Weight for the first array elements

src2 The second source array

beta Weight for the second array elements

dst The destination array

gamma Scalar, added to each sum

The function calculates the weighted sum of two arrays as follows:

```
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
```

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvAnd (view/add comments)

Calculates per-element bit-wise conjunction of two arrays.

```
void cvAnd(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)&src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cvAndS (view/add comments)

Calculates per-element bit-wise conjunction of an array and a scalar.

```
void cvAndS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)&value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

The following sample demonstrates how to calculate the absolute value of floating-point array elements by clearing the most-significant bit:

```
float a[] = { -1, 2, -3, 4, -5, 6, -7, 8, -9 };
CvMat A = cvMat(3, 3, CV\_32F, &a);
int i, absMask = 0x7fffffff;
cvAndS(&A, cvRealScalar(*(float*)&absMask), &A, 0);
for(i = 0; i < 9; i++ )
    printf("%.1f ", a[i]);
```

The code should print:

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

---

cvAvg (view/add comments)

Calculates average (mean) of array elements.

```
CvScalar cvAvg(const CvArr* arr, const CvArr* mask=NULL);
```

arr The array

mask The optional operation mask

The function calculates the average value `M` of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$M_c = \frac{\sum_{I, \texttt{mask}(I) \neq 0} \texttt{arr}(I)_c}{N}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average to the first scalar component S_0 .

cvAvgSdv (view/add comments)

Calculates average (mean) of array elements.

```
void cvAvgSdv(const CvArr* arr, CvScalar* mean, CvScalar* stdDev, const
CvArr* mask=NULL);
```

arr The array

mean Pointer to the output mean value, may be NULL if it is not needed

stdDev Pointer to the output standard deviation

mask The optional operation mask

The function calculates the average value and standard deviation of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$mean_c = \frac{1}{N} \sum_{I, \texttt{mask}(I) \neq 0} \texttt{arr}(I)_c$$
$$stdDev_c = \sqrt{\frac{1}{N} \sum_{I, \texttt{mask}(I) \neq 0} (\texttt{arr}(I)_c - mean_c)^2}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average and standard deviation to the first components of the output scalars ($mean_0$ and $stdDev_0$).

cvCalcCovarMatrix (view/add comments)

Calculates covariance matrix of a set of vectors.

```
void cvCalcCovarMatrix(
           const CvArr** vects,
           int count,
           CvArr* covMat,
           CvArr* avg,
           int flags);
```

vects The input vectors, all of which must have the same type and the same size. The vectors do not have to be 1D, they can be 2D (e.g., images) and so forth

count The number of input vectors

covMat The output covariance matrix that should be floating-point and square

avg The input or output (depending on the flags) array - the mean (average) vector of the input vectors

flags The operation flags, a combination of the following values

CV_COVAR_SCRAMBLED The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]$$

, that is, the covariance matrix is `count` $\times$ `count`. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces technique for face recognition). Eigenvalues of this "scrambled" matrix will match the eigenvalues of the true covariance matrix and the "true" eigenvectors can be easily calculated from the eigenvectors of the "scrambled" covariance matrix.

CV_COVAR_NORMAL The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...] \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T$$

, that is, `covMat` will be a covariance matrix with the same linear size as the total number of elements in each input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

CV_COVAR_USE_AVG If the flag is specified, the function does not calculate `avg` from the input vectors, but, instead, uses the passed `avg` vector. This is useful if `avg` has been already calculated somehow, or if the covariance matrix is calculated by parts - in this case, `avg` is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.

CV_COVAR_SCALE If the flag is specified, the covariance matrix is scaled. In the "normal" mode `scale` is '1./count'; in the "scrambled" mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled ('scale=1').

CV_COVAR_ROWS Means that all the input vectors are stored as rows of a single matrix, `vects[0].count` is ignored in this case, and `avg` should be a single-row vector of an appropriate size.

CV_COVAR_COLS Means that all the input vectors are stored as columns of a single matrix, `vects[0]. count` is ignored in this case, and `avg` should be a single-column vector of an appropriate size.

The function calculates the covariance matrix and, optionally, the mean vector of the set of input vectors. The function can be used for PCA, for comparing vectors using Mahalanobis distance and so forth.

---

cvCartToPolar (view/add comments)

Calculates the magnitude and/or angle of 2d vectors.

```
void cvCartToPolar(
           const CvArr* x,
           const CvArr* y,
           CvArr* magnitude,
           CvArr* angle=NULL,
           int angleInDegrees=0);
```

x The array of x-coordinates

y The array of y-coordinates

magnitude The destination array of magnitudes, may be set to NULL if it is not needed

angle The destination array of angles, may be set to NULL if it is not needed. The angles are measured in radians (0 to 2π) or in degrees (0 to 360 degrees).

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

```
magnitude(I)=sqrt(x(I)^2^+y(I)^2^ ),
angle(I)=atan(y(I)/x(I) )
```

The angles are calculated with 0.1 degree accuracy. For the (0,0) point, the angle is set to 0.

cvCbrt (view/add comments)

Calculates the cubic root

```
float cvCbrt(float value);
```

value The input floating-point value

The function calculates the cubic root of the argument, and normally it is faster than `pow(value,1./3)`. In addition, negative arguments are handled properly. Special values ($\pm\infty$, NaN) are not handled.

cvClearND (view/add comments)

Clears a specific array element.

```
void cvClearND(CvArr* arr, int* idx);
```

arr Input array

idx Array of the element indices

The function cvClearND clears (sets to zero) a specific element of a dense array or deletes the element of a sparse array. If the sparse array element does not exists, the function does nothing.

cvCloneImage (view/add comments)

Makes a full copy of an image, including the header, data, and ROI.

```
IplImage* cvCloneImage(const IplImage* image);
```

image The original image

The returned `IplImage*` points to the image copy.

cvCloneMat (view/add comments)

Creates a full matrix copy.

```
CvMat* cvCloneMat(const CvMat* mat);
```

mat Matrix to be copied

Creates a full copy of a matrix and returns a pointer to the copy.

cvCloneMatND (view/add comments)

Creates full copy of a multi-dimensional array and returns a pointer to the copy.

```
CvMatND* cvCloneMatND(const CvMatND* mat);
```

mat Input array

cvCloneSparseMat (view/add comments)

Creates full copy of sparse array.

```
CvSparseMat* cvCloneSparseMat(const CvSparseMat* mat);
```

mat Input array

The function creates a copy of the input array and returns pointer to the copy.

cvCmp (view/add comments)

Performs per-element comparison of two arrays.

```
void cvCmp(const CvArr* src1, const CvArr* src2, CvArr* dst, int
cmpOp);
```

src1 The first source array

src2 The second source array. Both source arrays must have a single channel.

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

- **CV_CMP_EQ** src1(I) "equal to" value
- **CV_CMP_GT** src1(I) "greater than" value
- **CV_CMP_GE** src1(I) "greater or equal" value
- **CV_CMP_LT** src1(I) "less than" value
- **CV_CMP_LE** src1(I) "less or equal" value
- **CV_CMP_NE** src1(I) "not equal" value

The function compares the corresponding elements of two arrays and fills the destination mask array:

```
dst(I)=src1(I) op src2(I),
```

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

---

cvCmpS (view/add comments)

Performs per-element comparison of an array and a scalar.

```
void cvCmpS(const CvArr* src, double value, CvArr* dst, int cmpOp);
```

src The source array, must have a single channel

value The scalar value to compare each array element with

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

CV_CMP_EQ src1(I) "equal to" value

CV_CMP_GT src1(I) "greater than" value

CV_CMP_GE src1(I) "greater or equal" value

CV_CMP_LT src1(I) "less than" value

CV_CMP_LE src1(I) "less or equal" value

CV_CMP_NE src1(I) "not equal" value

The function compares the corresponding elements of an array and a scalar and fills the destination mask array:

```
dst(I)=src(I) op scalar
```

where op is $=$, $>$, $\geq$, $<$, $\leq$ or $\neq$.

dst(I) is set to 0xff (all 1-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same size (or ROI size).

---

cvConvertScale (view/add comments)

Converts one array to another with optional linear transformation.

```
void cvConvertScale(const CvArr* src, CvArr* dst, double scale=1,
double shift=0);
```

```
#define cvCvtScale cvConvertScale
#define cvScale  cvConvertScale
#define cvConvert(src, dst )  cvConvertScale((src), (dst), 1, 0 )
```

src Source array

dst Destination array

scale Scale factor

shift Value added to the scaled source array elements

The function has several different purposes, and thus has several different names. It copies one array to another with optional scaling, which is performed first, and/or optional type conversion, performed after:

$$\texttt{dst}(I) = \texttt{scalesrc}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)$$

All the channels of multi-channel arrays are processed independently.

The type of conversion is done with rounding and saturation, that is if the result of scaling + conversion can not be represented exactly by a value of the destination array element type, it is set to the nearest representable value on the real axis.

In the case of `scale=1, shift=0` no prescaling is done. This is a specially optimized case and it has the appropriate cvConvert name. If source and destination array types have equal types, this is also a special case that can be used to scale and shift a matrix or an image and that is caled cvScale.

---

cvConvertScaleAbs (view/add comments)

Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

```
void cvConvertScaleAbs(const CvArr* src, CvArr* dst, double scale=1,
double shift=0);
```

src Source array

dst Destination array (should have 8u depth)

scale ScaleAbs factor

shift Value added to the scaled source array elements

The function is similar to cvConvertScale, but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scalesrc}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of cvConvertScale and cvAbs functions.

cvCvtScaleAbs (view/add comments)

Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

```
void cvCvtScaleAbs(const CvArr* src, CvArr* dst, double scale=1, double
shift=0);
```

src Source array

dst Destination array (should have 8u depth)

scale ScaleAbs factor

shift Value added to the scaled source array elements

The function is similar to cvConvertScale, but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scale}\texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of cvConvertScale and cvAbs functions.

cvCopy (view/add comments)

Copies one array to another.

```
void cvCopy(const CvArr* src, CvArr* dst, const CvArr* mask=NULL);
```

src The source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies selected elements from an input array to an output array:

$$\texttt{dst}(I) = \texttt{src}(I) \quad \text{if} \quad \texttt{mask}(I) \neq 0.$$

If any of the passed arrays is of `IplImage` type, then its ROI and COI fields are used. Both arrays must have the same type, the same number of dimensions, and the same size. The function can also copy sparse arrays (mask is not supported in this case).

---

cvCountNonZero (view/add comments)

Counts non-zero array elements.

```
int cvCountNonZero(const CvArr* arr);
```

arr The array must be a single-channel array or a multi-channel image with COI set

The function returns the number of non-zero elements in arr:

$$\sum_I (\texttt{arr}(I) \neq 0)$$

In the case of `IplImage` both ROI and COI are supported.

---

cvCreateData (view/add comments)

Allocates array data

```
void cvCreateData(CvArr* arr);
```

arr Array header

The function allocates image, matrix or multi-dimensional array data. Note that in the case of matrix types OpenCV allocation functions are used and in the case of IplImage they are used unless `CV_TURN_ON_IPL_COMPATIBILITY` was called. In the latter case IPL functions are used to allocate the data.

cvCreateImage (view/add comments)

Creates an image header and allocates the image data.

```
IplImage* cvCreateImage(CvSize size, int depth, int channels);
```

size Image width and height

depth Bit depth of image elements. See IplImage for valid depths.

channels Number of channels per pixel. See IplImage for details. This function only creates images with interleaved channels.

This call is a shortened form of

```
header = cvCreateImageHeader(size, depth, channels);
cvCreateData(header);
```

cvCreateImageHeader (view/add comments)

Creates an image header but does not allocate the image data.

```
IplImage* cvCreateImageHeader(CvSize size, int depth, int channels);
```

size Image width and height

depth Image depth (see cvCreateImage)

channels Number of channels (see cvCreateImage)

This call is an analogue of

```
hdr=iplCreateImageHeader(channels, 0, depth,
                         channels == 1 ? "GRAY" : "RGB",
                         channels == 1 ? "GRAY" : channels == 3 ? "BGR" :
                         channels == 4 ? "BGRA" : "",
                         IPL_DATA_ORDER_PIXEL, IPL_ORIGIN_TL, 4,
                         size.width, size.height,
                         0,0,0,0);
```

but it does not use IPL functions by default (see the CV_TURN_ON_IPL_COMPATIBILITY macro).

cvCreateMat (view/add comments)

Creates a matrix header and allocates the matrix data.

```
CvMat* cvCreateMat(
            int rows,
            int cols,
            int type);
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type The type of the matrix elements in the form `CV_<bit depth><S|U|F>C<number of channels>`, where S=signed, U=unsigned, F=float. For example, CV_8UC1 means the elements are 8-bit unsigned and the there is 1 channel, and CV_32SC2 means the elements are 32-bit signed and there are 2 channels.

This is the concise form for:

```
CvMat* mat = cvCreateMatHeader(rows, cols, type);
cvCreateData(mat);
```

cvCreateMatHeader (view/add comments)

Creates a matrix header but does not allocate the matrix data.

```
CvMat* cvCreateMatHeader(
            int rows,
            int cols,
            int type);
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements, see cvCreateMat

The function allocates a new matrix header and returns a pointer to it. The matrix data can then be allocated using cvCreateData or set explicitly to user-allocated data via cvSetData.

---

cvCreateMatND (view/add comments)

Creates the header and allocates the data for a multi-dimensional dense array.

```
CvMatND* cvCreateMatND(
            int dims,
            const int* sizes,
            int type);
```

dims Number of array dimensions. This must not exceed CV_MAX_DIM (32 by default, but can be changed at build time).

sizes Array of dimension sizes.

type Type of array elements, see cvCreateMat.

This is a short form for:

```
CvMatND* mat = cvCreateMatNDHeader(dims, sizes, type);
cvCreateData(mat);
```

---

cvCreateMatNDHeader (view/add comments)

Creates a new matrix header but does not allocate the matrix data.

```
CvMatND* cvCreateMatNDHeader(
            int dims,
            const int* sizes,
            int type);
```

dims Number of array dimensions

sizes Array of dimension sizes

type Type of array elements, see cvCreateMat

The function allocates a header for a multi-dimensional dense array. The array data can further be allocated using cvCreateData or set explicitly to user-allocated data via cvSetData.

---

cvCreateSparseMat (view/add comments)

Creates sparse array.

```
CvSparseMat* cvCreateSparseMat(int dims, const int* sizes, int type);
```

dims Number of array dimensions. In contrast to the dense matrix, the number of dimensions is practically unlimited (up to 2^{16}).

sizes Array of dimension sizes

type Type of array elements. The same as for CvMat

The function allocates a multi-dimensional sparse array. Initially the array contain no elements, that is cvGet or cvGetReal returns zero for every index.

---

cvCrossProduct (view/add comments)

Calculates the cross product of two 3D vectors.

```
void cvCrossProduct(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source vector

src2 The second source vector

dst The destination vector

The function calculates the cross product of two 3D vectors:

$$\texttt{dst} = \texttt{src1} \times \texttt{src2}$$

or:

$$\begin{aligned}\texttt{dst}_1 &= \texttt{src1}_2\texttt{src2}_3 - \texttt{src1}_3\texttt{src2}_2 \\ \texttt{dst}_2 &= \texttt{src1}_3\texttt{src2}_1 - \texttt{src1}_1\texttt{src2}_3 \\ \texttt{dst}_3 &= \texttt{src1}_1\texttt{src2}_2 - \texttt{src1}_2\texttt{src2}_1\end{aligned}$$

CvtPixToPlane

Synonym for Split .

---

cvDCT (view/add comments)

Performs a forward or inverse Discrete Cosine transform of a 1D or 2D floating-point array.

```
void cvDCT(const CvArr* src, CvArr* dst, int flags);
```

src Source array, real 1D or 2D array

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

CV_DXT_FORWARD do a forward 1D or 2D transform.

CV_DXT_INVERSE do an inverse 1D or 2D transform.

CV_DXT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

The function performs a forward or inverse transform of a 1D or 2D floating-point array: Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C_{jk}^{(N)} = \sqrt{\alpha_j / N} \cos\left(\frac{\pi(2k+1)j}{2N}\right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.

Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)}\right)^{-1} \cdot Y = \left(C^{(N)}\right)^{T} \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)}\right)^{T} = I$)

Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)}\right)^{T}$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)}\right)^{T} \cdot X \cdot C^{(N)}$$

---

cvDFT (view/add comments)

Performs a forward or inverse Discrete Fourier transform of a 1D or 2D floating-point array.

```
void cvDFT(const CvArr* src, CvArr* dst, int flags, int nonzeroRows=0);
```

src Source array, real or complex

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

- **CV_DXT_FORWARD** do a forward 1D or 2D transform. The result is not scaled.
- **CV_DXT_INVERSE** do an inverse 1D or 2D transform. The result is not scaled. CV_DXT_FORWARD and CV_DXT_INVERSE are mutually exclusive, of course.
- **CV_DXT_SCALE** scale the result: divide it by the number of array elements. Usually, it is combined with CV_DXT_INVERSE, and one may use a shortcut CV_DXT_INV_SCALE.
- **CV_DXT_ROWS** do a forward or inverse transform of every individual row of the input matrix. This flag allows the user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.
- **CV_DXT_INVERSE_SCALE** same as CV_DXT_INVERSE + CV_DXT_SCALE

nonzeroRows Number of nonzero rows in the source array (in the case of a forward 2d transform), or a number of rows of interest in the destination array (in the case of an inverse 2d transform). If the value is negative, zero, or greater than the total number of rows, it is ignored. The parameter can be used to speed up 2d convolution/correlation when computing via DFT. See the example below.

The function performs a forward or inverse transform of a 1D or 2D floating-point array:
Forward Fourier transform of 1D vector of N elements:

$$y = F^{(N)} \cdot x, where F^{(N)}_{jk} = exp(-i \cdot 2\pi \cdot j \cdot k/N)$$

,

$$i = sqrt(-1)$$

Inverse Fourier transform of 1D vector of N elements:

$$x' = (F^{(N)})^{-1} \cdot y = conj(F^(N)) \cdot yx = (1/N) \cdot x$$

Forward Fourier transform of 2D vector of M $\times$ N elements:

$$Y = F^{(M)} \cdot X \cdot F^{(N)}$$

Inverse Fourier transform of 2D vector of M $\times$ N elements:

$$X' = conj(F^{(M)}) \cdot Y \cdot conj(F^{(N)})X = (1/(M \cdot N)) \cdot X'$$

In the case of real (single-channel) data, the packed format, borrowed from IPL, is used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix} ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\ ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\ ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\ \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots \\ ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \hdots & \hdots & \hdots & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\ ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \hdots & \hdots & \hdots & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\ ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \hdots & \hdots & \hdots & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2} \end{bmatrix}$$

Note: the last column is present if `N` is even, the last row is present if `M` is even. In the case of 1D real transform the result looks like the first row of the above matrix.

Here is the example of how to compute 2D convolution using DFT.

```
CvMat* A = cvCreateMat(M1, N1, CVg32F);
CvMat* B = cvCreateMat(M2, N2, A->type);

// it is also possible to have only abs(M2-M1)+1 times abs(N2-N1)+1
// part of the full convolution result
CvMat* conv = cvCreateMat(A->rows + B->rows - 1, A->cols + B->cols - 1,
                          A->type);

// initialize A and B
...

int dftgM = cvGetOptimalDFTSize(A->rows + B->rows - 1);
int dftgN = cvGetOptimalDFTSize(A->cols + B->cols - 1);

CvMat* dftgA = cvCreateMat(dft\_M, dft\_N, A->type);
CvMat* dftgB = cvCreateMat(dft\_M, dft\_N, B->type);
CvMat tmp;

// copy A to dftgA and pad dft\_A with zeros
cvGetSubRect(dftgA, &tmp, cvRect(0,0,A->cols,A->rows));
cvCopy(A, &tmp);
cvGetSubRect(dftgA, &tmp, cvRect(A->cols,0,dft\_A->cols - A->cols,A->rows));
cvZero(&tmp);
// no need to pad bottom part of dftgA with zeros because of
// use nonzerogrows parameter in cvDFT() call below

cvDFT(dftgA, dft\_A, CV\_DXT\_FORWARD, A->rows);

// repeat the same with the second array
cvGetSubRect(dftgB, &tmp, cvRect(0,0,B->cols,B->rows));
cvCopy(B, &tmp);
cvGetSubRect(dftgB, &tmp, cvRect(B->cols,0,dft\_B->cols - B->cols,B->rows));
cvZero(&tmp);
// no need to pad bottom part of dftgB with zeros because of
// use nonzerogrows parameter in cvDFT() call below

cvDFT(dftgB, dft\_B, CV\_DXT\_FORWARD, B->rows);

cvMulSpectrums(dftgA, dft\_B, dft\_A, 0 /* or CV\_DXT\_MUL\_CONJ to get
               correlation rather than convolution */);

cvDFT(dftgA, dft\_A, CV\_DXT\_INV\_SCALE, conv->rows); // calculate only
                                                      // the top part
cvGetSubRect(dftgA, &tmp, cvRect(0,0,conv->cols,conv->rows));
```

```
cvCopy(&tmp, conv);
```

cvDecRefData (view/add comments)

Decrements an array data reference counter.

```
void cvDecRefData(CvArr* arr);
```

arr Pointer to an array header

The function decrements the data reference counter in a CvMat or CvMatND if the reference counter pointer is not NULL. If the counter reaches zero, the data is deallocated. In the current implementation the reference counter is not NULL only if the data was allocated using the cvCreateData function. The counter will be NULL in other cases such as: external data was assigned to the header using cvSetData, the matrix header is part of a larger matrix or image, or the header was converted from an image or n-dimensional matrix header.

cvDet (view/add comments)

Returns the determinant of a matrix.

```
double cvDet(const CvArr* mat);
```

mat The source matrix

The function returns the determinant of the square matrix `mat`. The direct method is used for small matrices and Gaussian elimination is used for larger matrices. For symmetric positive-determined matrices, it is also possible to run cvSVD with $U = V = 0$ and then calculate the determinant as a product of the diagonal elements of W.

cvDiv (view/add comments)

Performs per-element division of two arrays.

```
void cvDiv(const CvArr* src1, const CvArr* src2, CvArr* dst, double
scale=1);
```

src1 The first source array. If the pointer is NULL, the array is assumed to be all 1's.

src2 The second source array

dst The destination array

scale Optional scale factor

The function divides one array by another:

$$\texttt{dst}(I) = \begin{cases} \texttt{scale} \cdot \texttt{src1}(I)/\texttt{src2}(I) & \text{if src1 is not NULL} \\ \texttt{scale}/\texttt{src2}(I) & \text{otherwise} \end{cases}$$

All the arrays must have the same type and the same size (or ROI size).

---

cvDotProduct (view/add comments)

Calculates the dot product of two arrays in Euclidian metrics.

```
double cvDotProduct(const CvArr* src1, const CvArr* src2);
```

src1 The first source array

src2 The second source array

The function calculates and returns the Euclidean dot product of two arrays.

$$src1 \bullet src2 = \sum_I (\texttt{src1}(I)\texttt{src2}(I))$$

In the case of multiple channel arrays, the results for all channels are accumulated. In particular, `cvDotProduct(a,a)` where `a` is a complex vector, will return $||\texttt{a}||^2$. The function can process multi-dimensional arrays, row by row, layer by layer, and so on.

cvEigenVV (view/add comments)

Computes eigenvalues and eigenvectors of a symmetric matrix.

```
void cvEigenVV(
           CvArr* mat,
           CvArr* evects,
           CvArr* evals,
           double eps=0,
           int lowindex = -1,
           int highindex = -1);
```

mat The input symmetric square matrix, modified during the processing

evects The output matrix of eigenvectors, stored as subsequent rows

evals The output vector of eigenvalues, stored in the descending order (order of eigenvalues and eigenvectors is syncronized, of course)

eps Accuracy of diagonalization. Typically, DBL_EPSILON (about 10^{-15}) works well. THIS PARAMETER IS CURRENTLY IGNORED.

lowindex Optional index of largest eigenvalue/-vector to calculate. (See below.)

highindex Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The function computes the eigenvalues and eigenvectors of matrix A:

```
mat*evects(i,:)' = evals(i)*evects(i,:)' (in MATLAB notation)
```

If either low- or highindex is supplied the other is required, too. Indexing is 0-based. Example: To calculate the largest eigenvector/-value set lowindex=highindex=0. To calculate all the eigenvalues, leave lowindex=highindex=-1. For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

The contents of matrix A is destroyed by the function.

Currently the function is slower than cvSVD yet less accurate, so if A is known to be positively-defined (for example, it is a covariance matrix)it is recommended to use cvSVD to find eigenvalues and eigenvectors of A, especially if eigenvectors are not required.

cvExp (view/add comments)

Calculates the exponent of every array element.

```
void cvExp(const CvArr* src, CvArr* dst);
```

src The source array

dst The destination array, it should have `double` type or the same type as the source

The function calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}(I)}$$

The maximum relative error is about 7×10^{-6}. Currently, the function converts denormalized values to zeros on output.

cvFastArctan (view/add comments)

Calculates the angle of a 2D vector.

```
float cvFastArctan(float y, float x);
```

x x-coordinate of 2D vector

y y-coordinate of 2D vector

The function calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0 degrees to 360 degrees. The accuracy is about 0.1 degrees.

cvFlip (view/add comments)

Flip a 2D array around vertical, horizontal or both axes.

```
void cvFlip(const CvArr* src, CvArr* dst=NULL, int flipMode=0);
```

src Source array

dst Destination array. If dst = NULL the flipping is done in place.

flipMode Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas:

The function flips the array in one of three different ways (row and column indices are 0-based):

$$dst(i,j) = \begin{cases} \texttt{src}(rows(\texttt{src}) - i - 1, j) & \text{if } \texttt{flipMode} = 0 \\ \texttt{src}(i, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} > 0 \\ \texttt{src}(rows(\texttt{src}) - i - 1, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (flipMode = 0) to switch between top-left and bottom-left image origin, which is a typical operation in video processing under Win32 systems.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (flipMode $>$ 0)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flipMode $<$ 0)
- reversing the order of 1d point arrays (flipMode ¿ 0)

---

cvGEMM (view/add comments)

Performs generalized matrix multiplication.

```
void cvGEMM(
           const CvArr* src1,
           const CvArr* src2, double alpha,
           const CvArr* src3,
           double beta,
```

```
            CvArr* dst,
            int tABC=0);
#define cvMatMulAdd(src1, src2, src3, dst ) cvGEMM(src1, src2, 1, src3,
1, dst, 0 )
            #define cvMatMul(src1, src2, dst ) cvMatMulAdd(src1, src2,
0, dst )
```

src1 The first source array

src2 The second source array

src3 The third source array (shift). Can be NULL, if there is no shift.

dst The destination array

tABC The operation flags that can be 0 or a combination of the following values

CV_GEMM_A_T transpose src1

CV_GEMM_B_T transpose src2

CV_GEMM_C_T transpose src3

For example, CV_GEMM_A_T+CV_GEMM_C_T corresponds to

$$\texttt{alpha}\,\texttt{src1}^T\,\texttt{src2} + \texttt{beta}\,\texttt{src3}^T$$

The function performs generalized matrix multiplication:

$$\texttt{dst} = \texttt{alpha}\,op(\texttt{src1})\,op(\texttt{src2}) + \texttt{beta}\,op(\texttt{src3}) \quad \text{where } op(X) \text{ is } X \text{ or } X^T$$

All the matrices should have the same data type and coordinated sizes. Real or complex floating-point matrices are supported.

---

cvGet?D (view/add comments)

Return a specific array element.

```
CvScalar cvGet1D(const CvArr* arr, int idx0); CvScalar cvGet2D(const
CvArr* arr, int idx0, int idx1); CvScalar cvGet3D(const CvArr* arr,
int idx0, int idx1, int idx2); CvScalar cvGetND(const CvArr* arr, int*
idx);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

The functions return a specific array element. In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

---

cvGetCol(s) (view/add comments)

Returns array column or column span.

```
CvMat* cvGetCol(const CvArr* arr, CvMat* submat, int col);
```

```
CvMat* cvGetCols(const CvArr* arr, CvMat* submat, int startCol, int
endCol);
```

arr Input array

submat Pointer to the resulting sub-array header

col Zero-based index of the selected column

startCol Zero-based index of the starting column (inclusive) of the span

endCol Zero-based index of the ending column (exclusive) of the span

The functions `GetCol` and `GetCols` return the header, corresponding to a specified column span of the input array. `GetCol` is a shortcut for cvGetCols:

```
cvGetCol(arr, submat, col); // ~ cvGetCols(arr, submat, col, col + 1);
```

cvGetDiag (view/add comments)

Returns one of array diagonals.

```
CvMat* cvGetDiag(const CvArr* arr, CvMat* submat, int diag=0);
```

arr Input array

submat Pointer to the resulting sub-array header

diag Array diagonal. Zero corresponds to the main diagonal, -1 corresponds to the diagonal above the main , 1 corresponds to the diagonal below the main, and so forth.

The function returns the header, corresponding to a specified diagonal of the input array.

cvGetDims, cvGetDimSize

Return number of array dimensions and their sizes or the size of a particular dimension.

```
int cvGetDims(const CvArr* arr, int* sizes=NULL);
```

```
int cvGetDimSize(const CvArr* arr, int index);
```

arr Input array

sizes Optional output vector of the array dimension sizes. For 2d arrays the number of rows (height) goes first, number of columns (width) next.

index Zero-based dimension index (for matrices 0 means number of rows, 1 means number of columns; for images 0 means height, 1 means width)

The function cvGetDims returns the array dimensionality and the array of dimension sizes. In the case of IplImage or CvMat it always returns 2 regardless of number of image/matrix rows. The function cvGetDimSize returns the particular dimension size (number of elements per that dimension). For example, the following code calculates total number of array elements in two ways:

```
// via cvGetDims()
int sizes[CV_MAX_DIM];
int i, total = 1;
int dims = cvGetDims(arr, size);
for(i = 0; i < dims; i++ )
    total *= sizes[i];

// via cvGetDims() and cvGetDimSize()
int i, total = 1;
int dims = cvGetDims(arr);
for(i = 0; i < dims; i++ )
    total *= cvGetDimsSize(arr, i);
```

---

cvGetElemType (view/add comments)

Returns type of array elements.

```
int cvGetElemType(const CvArr* arr);
```

arr Input array

The function returns type of the array elements as described in cvCreateMat discussion: CV_8UC1 ... CV_64FC4.

---

cvGetImage (view/add comments)

Returns image header for arbitrary array.

```
IplImage* cvGetImage(const CvArr* arr, IplImage* imageHeader);
```

arr Input array

imageHeader Pointer to `IplImage` structure used as a temporary buffer

The function returns the image header for the input array that can be a matrix - CvMat , or an image - `IplImage*`. In the case of an image the function simply returns the input pointer. In the case of CvMat it initializes an `imageHeader` structure with the parameters of the input matrix. Note that if we transform `IplImage` to CvMat and then transform CvMat back to IplImage, we can get different headers if the ROI is set, and thus some IPL functions that calculate image stride from its width and align may fail on the resultant image.

---

cvGetImageCOI (view/add comments)

Returns the index of the channel of interest.

```
int cvGetImageCOI(const IplImage* image);
```

image A pointer to the image header

Returns the channel of interest of in an IplImage. Returned values correspond to the `coi` in cvSetImageCOI.

---

cvGetImageROI (view/add comments)

Returns the image ROI.

```
CvRect cvGetImageROI(const IplImage* image);
```

image A pointer to the image header

If there is no ROI set, `cvRect(0,0,image->width,image->height)` is returned.

cvGetMat (view/add comments)

Returns matrix header for arbitrary array.

```
CvMat* cvGetMat(const CvArr* arr, CvMat* header, int* coi=NULL, int
allowND=0);
```

arr Input array

header Pointer to CvMat structure used as a temporary buffer

coi Optional output parameter for storing COI

allowND If non-zero, the function accepts multi-dimensional dense arrays (CvMatND*) and returns 2D (if CvMatND has two dimensions) or 1D matrix (when CvMatND has 1 dimension or more than 2 dimensions). The array must be continuous.

The function returns a matrix header for the input array that can be a matrix - CvMat , an image - `IplImage` or a multi-dimensional dense array - CvMatND (latter case is allowed only if `allowND != 0`) . In the case of matrix the function simply returns the input pointer. In the case of `IplImage*` or CvMatND it initializes the `header` structure with parameters of the current image ROI and returns the pointer to this temporary structure. Because COI is not supported by CvMat , it is returned separately.

The function provides an easy way to handle both types of arrays - `IplImage` and CvMat - using the same code. Reverse transform from CvMat to `IplImage` can be done using the cvGetImage function.

Input array must have underlying data allocated or attached, otherwise the function fails.

If the input array is `IplImage` with planar data layout and COI set, the function returns the pointer to the selected plane and COI = 0. It enables per-plane processing of multi-channel images with planar data layout using OpenCV functions.

cvGetNextSparseNode (view/add comments)

Returns the next sparse matrix element

```
CvSparseNode* cvGetNextSparseNode(CvSparseMatIterator* matIterator);
```

matIterator Sparse array iterator

The function moves iterator to the next sparse matrix element and returns pointer to it. In the current version there is no any particular order of the elements, because they are stored in the hash table. The sample below demonstrates how to iterate through the sparse matrix:

Using cvInitSparseMatIterator and cvGetNextSparseNode to calculate sum of floating-point sparse array.

```
double sum;
int i, dims = cvGetDims(array);
CvSparseMatIterator mat_iterator;
CvSparseNode* node = cvInitSparseMatIterator(array, &mat_iterator);

for(; node != 0; node = cvGetNextSparseNode(&mat_iterator ))
{
    /* get pointer to the element indices */
    int* idx = CV_NODE_IDX(array, node);
    /* get value of the element (assume that the type is CV_32FC1) */
    float val = *(float*)CV_NODE_VAL(array, node);
    printf("(");
    for(i = 0; i < dims; i++ )
        printf("%4d%s", idx[i], i < dims - 1 "," : "): ");
    printf("%g\n", val);

    sum += val;
}

printf("\nTotal sum = %g\n", sum);
```

---

cvGetOptimalDFTSize (view/add comments)

Returns optimal DFT size for a given vector size.

```
int cvGetOptimalDFTSize(int size0);
```

size0 Vector size

The function returns the minimum number N that is greater than or equal to size0, such that the DFT of a vector of size N can be computed fast. In the current implementation $N = 2^p \times 3^q \times 5^r$, for some p, q, r.

The function returns a negative number if size0 is too large (very close to INT_MAX)

cvGetRawData (view/add comments)

Retrieves low-level information about the array.

```
void cvGetRawData(const CvArr* arr, uchar** data, int* step=NULL,
CvSize* roiSize=NULL);
```

arr Array header

data Output pointer to the whole image origin or ROI origin if ROI is set

step Output full row length in bytes

roiSize Output ROI size

The function fills output variables with low-level information about the array data. All output parameters are optional, so some of the pointers may be set to NULL. If the array is IplImage with ROI set, the parameters of ROI are returned.

The following example shows how to get access to array elements. GetRawData calculates the absolute value of the elements in a single-channel, floating-point array.

```
float* data;
int step;

CvSize size;
int x, y;

cvGetRawData(array, (uchar**)&data, &step, &size);
step /= sizeof(data[0]);

for(y = 0; y < size.height; y++, data += step )
    for(x = 0; x < size.width; x++ )
        data[x] = (float)fabs(data[x]);
```

cvGetReal1D (view/add comments)

Return a specific element of single-channel 1D array.

```
double cvGetReal1D(const CvArr* arr, int idx0);
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cvGet function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

---

cvGetReal2D (view/add comments)

Return a specific element of single-channel 2D array.

```
double cvGetReal2D(const CvArr* arr, int idx0, int idx1);
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cvGet function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

---

cvGetReal3D (view/add comments)

Return a specific element of single-channel array.

```
double cvGetReal3D(const CvArr* arr, int idx0, int idx1, int idx2);
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cvGet function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

---

cvGetRealND (view/add comments)

Return a specific element of single-channel array.

```
double cvGetRealND(const CvArr* arr, int* idx)->float;
```

arr Input array. Must have a single channel.

idx Array of the element indices

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cvGet function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

cvGetRow(s) (view/add comments)

Returns array row or row span.

```
CvMat* cvGetRow(const CvArr* arr, CvMat* submat, int row);
```

```
CvMat* cvGetRows(const CvArr* arr, CvMat* submat, int startRow, int
endRow, int deltaRow=1);
```

arr Input array

submat Pointer to the resulting sub-array header

row Zero-based index of the selected row

startRow Zero-based index of the starting row (inclusive) of the span

endRow Zero-based index of the ending row (exclusive) of the span

deltaRow Index step in the row span. That is, the function extracts every `deltaRow`-th row from `startRow` and up to (but not including) `endRow`.

The functions return the header, corresponding to a specified row/row span of the input array. Note that `GetRow` is a shortcut for cvGetRows:

```
cvGetRow(arr, submat, row ) ~ cvGetRows(arr, submat, row, row + 1, 1);
```

cvGetSize (view/add comments)

Returns size of matrix or image ROI.

```
CvSize cvGetSize(const CvArr* arr);
```

arr array header

The function returns number of rows (CvSize::height) and number of columns (CvSize::width) of the input matrix or image. In the case of image the size of ROI is returned.

cvGetSubRect (view/add comments)

Returns matrix header corresponding to the rectangular sub-array of input image or matrix.

```
CvMat* cvGetSubRect(const CvArr* arr, CvMat* submat, CvRect rect);
```

arr Input array

submat Pointer to the resultant sub-array header

rect Zero-based coordinates of the rectangle of interest

The function returns header, corresponding to a specified rectangle of the input array. In other words, it allows the user to treat a rectangular part of input array as a stand-alone array. ROI is taken into account by the function so the sub-array of ROI is actually extracted.

cvInRange (view/add comments)

Checks that array elements lie between the elements of two other arrays.

```
void cvInRange(const CvArr* src, const CvArr* lower, const CvArr*
upper, CvArr* dst);
```

src The first source array

lower The inclusive lower boundary array

upper The exclusive upper boundary array

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0 \wedge \texttt{lower}(I)_1 <= \texttt{src}(I)_1 < \texttt{upper}(I)_1$$

For two-channel arrays and so forth,
dst(I) is set to 0xff (all 1-bits) if src(I) is within the range and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size).

---

cvInRangeS (view/add comments)

Checks that array elements lie between two scalars.

```
void cvInRangeS(const CvArr* src, CvScalar lower, CvScalar upper,
CvArr* dst);
```

src The first source array

lower The inclusive lower boundary

upper The exclusive upper boundary

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0 \wedge \texttt{lower}_1 <= \texttt{src}(I)_1 < \texttt{upper}_1$$

For two-channel arrays nd so forth,
'dst(I)' is set to 0xff (all 1-bits) if 'src(I)' is within the range and 0 otherwise. All the arrays must have the same size (or ROI size).

---

cvIncRefData (view/add comments)

Increments array data reference counter.

```
int cvIncRefData(CvArr* arr);
```

arr Array header

The function increments CvMat or CvMatND data reference counter and returns the new counter value if the reference counter pointer is not NULL, otherwise it returns zero.

---

cvInitImageHeader (view/add comments)

Initializes an image header that was previously allocated.

```
IplImage* cvInitImageHeader(
           IplImage* image,
           CvSize size,
           int depth,
           int channels,
           int origin=0,
           int align=4);
```

image Image header to initialize

size Image width and height

depth Image depth (see cvCreateImage)

channels Number of channels (see cvCreateImage)

origin Top-left `IPL_ORIGIN_TL` or bottom-left `IPL_ORIGIN_BL`

align Alignment for image rows, typically 4 or 8 bytes

The returned `IplImage*` points to the initialized header.

cvInitMatHeader (view/add comments)

Initializes a pre-allocated matrix header.

```
CvMat* cvInitMatHeader(
            CvMat* mat,
            int rows,
            int cols,
            int type,
            void* data=NULL,
            int step=CV_AUTOSTEP);
```

mat A pointer to the matrix header to be initialized

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements, see cvCreateMat.

data Optional: data pointer assigned to the matrix header

step Optional: full row width in bytes of the assigned data. By default, the minimal possible step is used which assumes there are no gaps between subsequent rows of the matrix.

This function is often used to process raw data with OpenCV matrix functions. For example, the following code computes the matrix product of two matrices, stored as ordinary arrays:

```
double a[] = { 1, 2, 3, 4,
               5, 6, 7, 8,
               9, 10, 11, 12 };

double b[] = { 1, 5, 9,
               2, 6, 10,
               3, 7, 11,
               4, 8, 12 };

double c[9];
CvMat Ma, Mb, Mc ;

cvInitMatHeader(&Ma, 3, 4, CV_64FC1, a);
```

```
cvInitMatHeader(&Mb, 4, 3, CV_64FC1, b);
cvInitMatHeader(&Mc, 3, 3, CV_64FC1, c);

cvMatMulAdd(&Ma, &Mb, 0, &Mc);
// the c array now contains the product of a (3x4) and b (4x3)
```

cvInitMatNDHeader (view/add comments)

Initializes a pre-allocated multi-dimensional array header.

```
CvMatND* cvInitMatNDHeader(
            CvMatND* mat,
            int dims,
            const int* sizes,
            int type,
            void* data=NULL);
```

mat A pointer to the array header to be initialized

dims The number of array dimensions

sizes An array of dimension sizes

type Type of array elements, see cvCreateMat

data Optional data pointer assigned to the matrix header

cvInitSparseMatIterator (view/add comments)

Initializes sparse array elements iterator.

```
CvSparseNode* cvInitSparseMatIterator(const CvSparseMat* mat,
CvSparseMatIterator* matIterator);
```

mat Input array

matIterator Initialized iterator

The function initializes iterator of sparse array elements and returns pointer to the first element, or NULL if the array is empty.

---

cvInvSqrt (view/add comments)

Calculates the inverse square root.

```
float cvInvSqrt(float value);
```

value The input floating-point value

The function calculates the inverse square root of the argument, and normally it is faster than `1./sqrt(value)`. If the argument is zero or negative, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

---

cvInv (view/add comments)

Synonym for Invert

---

cvInvert (view/add comments)

Finds the inverse or pseudo-inverse of a matrix.

```
double cvInvert(const CvArr* src, CvArr* dst, int method=CV_LU);
```

src The source matrix

dst The destination matrix

method Inversion method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix

The function inverts matrix `src1` and stores the result in `src2`.

In the case of `LU` method, the function returns the `src1` determinant (src1 must be square). If it is 0, the matrix is not inverted and `src2` is filled with zeros.

In the case of `SVD` methods, the function returns the inversed condition of `src1` (ratio of the smallest singular value to the largest singular value) and 0 if `src1` is all zeros. The SVD methods calculate a pseudo-inverse matrix if `src1` is singular.

---

cvIsInf (view/add comments)

Determines if the argument is Infinity.

```
int cvIsInf(double value);
```

value The input floating-point value

The function returns 1 if the argument is $\pm\infty$ (as defined by IEEE754 standard), 0 otherwise.

---

cvIsNaN (view/add comments)

Determines if the argument is Not A Number.

```
int cvIsNaN(double value);
```

value The input floating-point value

The function returns 1 if the argument is Not A Number (as defined by IEEE754 standard), 0 otherwise.

---

cvLUT (view/add comments)

Performs a look-up table transform of an array.

```
void cvLUT(const CvArr* src, CvArr* dst, const CvArr* lut);
```

src Source array of 8-bit elements

dst Destination array of a given depth and of the same number of channels as the source array

lut Look-up table of 256 elements; should have the same depth as the destination array. In the case of multi-channel source and destination arrays, the table should either have a single-channel (in this case the same table is used for all channels) or the same number of channels as the source/destination array.

The function fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of `src` as follows:

$$\texttt{dst}_i \leftarrow \texttt{lut}_{\texttt{src}_i + d}$$

where

$$d = \begin{cases} 0 & \text{if } \texttt{src} \text{ has depth } \texttt{CV_8U} \\ 128 & \text{if } \texttt{src} \text{ has depth } \texttt{CV_8S} \end{cases}$$

---

cvLog (view/add comments)

Calculates the natural logarithm of every array element's absolute value.

```
void cvLog(const CvArr* src, CvArr* dst);
```

src The source array

dst The destination array, it should have `double` type or the same type as the source

The function calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}[I] = \begin{cases} \log|\texttt{src}(I) & \text{if } \texttt{src}[I] \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where `C` is a large negative number (about -700 in the current implementation).

cvMahalanobis (view/add comments)

Calculates the Mahalanobis distance between two vectors.

```
double cvMahalanobis(
            const CvArr* vec1,
            const CvArr* vec2,
            CvArr* mat);
```

vec1 The first 1D source vector

vec2 The second 1D source vector

mat The inverse covariance matrix

The function calculates and returns the weighted distance between two vectors:

$$d(\texttt{vec1},\texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the cvCalcCovarMatrix function and further inverted using the cvInvert function (CV_SVD method is the prefered one because the matrix might be singular).

cvMat (view/add comments)

Initializes matrix header (lightweight variant).

```
CvMat cvMat(
            int rows,
            int cols,
            int type,
            void* data=NULL);
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements - see cvCreateMat

data Optional data pointer assigned to the matrix header

Initializes a matrix header and assigns data to it. The matrix is filled *row*-wise (the first `cols` elements of data form the first row of the matrix, etc.)

This function is a fast inline substitution for cvInitMatHeader. Namely, it is equivalent to:

```
CvMat mat;
cvInitMatHeader(&mat, rows, cols, type, data, CV\_AUTOSTEP);
```

---

cvMax (view/add comments)

Finds per-element maximum of two arrays.

```
void cvMax(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cvMaxS (view/add comments)

Finds per-element maximum of array and scalar.

```
void cvMaxS(const CvArr* src, double value, CvArr* dst);
```

src The first source array

value The scalar value

dst The destination array

The function calculates per-element maximum of array and scalar:

$$\texttt{dst}(I) = \max(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cvMerge (view/add comments)

Composes a multi-channel array from several single-channel arrays or inserts a single channel into the array.

```
void cvMerge(const CvArr* src0, const CvArr* src1, const CvArr* src2,
const CvArr* src3, CvArr* dst);
```

```
#define cvCvtPlaneToPix cvMerge
```

src0 Input channel 0

src1 Input channel 1

src2 Input channel 2

src3 Input channel 3

dst Destination array

The function is the opposite to cvSplit. If the destination array has N channels then if the first N input channels are not NULL, they all are copied to the destination array; if only a single source channel of the first N is not NULL, this particular channel is copied into the destination array; otherwise an error is raised. The rest of the source channels (beyond the first N) must always be NULL. For IplImage cvCopy with COI set can be also used to insert a single channel into the image.

cvMin (view/add comments)

Finds per-element minimum of two arrays.

```
void cvMin(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

cvMinMaxLoc (view/add comments)

Finds global minimum and maximum in array or subarray.

```
void cvMinMaxLoc(const CvArr* arr, double* minVal, double* maxVal,
CvPoint* minLoc=NULL, CvPoint* maxLoc=NULL, const CvArr* mask=NULL);
```

arr The source array, single-channel or multi-channel with COI set

minVal Pointer to returned minimum value

maxVal Pointer to returned maximum value

minLoc Pointer to returned minimum location

maxLoc Pointer to returned maximum location

mask The optional mask used to select a subarray

The function finds minimum and maximum element values and their positions. The extremums are searched across the whole array, selected `ROI` (in the case of `IplImage`) or, if `mask` is not `NULL`, in the specified array region. If the array has more than one channel, it must be `IplImage` with `COI` set. In the case of multi-dimensional arrays, `minLoc->x` and `maxLoc->x` will contain raw (linear) positions of the extremums.

---

cvMinS (view/add comments)

Finds per-element minimum of an array and a scalar.

```
void cvMinS(const CvArr* src, double value, CvArr* dst);
```

src The first source array

value The scalar value

dst The destination array

The function calculates minimum of an array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

Mirror

Synonym for Flip .

---

cvMixChannels (view/add comments)

Copies several channels from input arrays to certain channels of output arrays

```
void cvMixChannels(const CvArr** src, int srcCount,
           CvArr** dst, int dstCount,
           const int* fromTo, int pairCount);
```

src Input arrays

srcCount The number of input arrays.

dst Destination arrays

dstCount The number of output arrays.

fromTo The array of pairs of indices of the planes copied. `fromTo[k*2]` is the 0-based index of the input channel in `src` and `fromTo[k*2+1]` is the index of the output channel in `dst`. Here the continuous channel numbering is used, that is, the first input image channels are indexed from `0` to `channels(src[0])-1`, the second input image channels are indexed from `channels(src[0])` to `channels(src[0]) + channels(src[1])-1` etc., and the same scheme is used for the output image channels. As a special case, when `fromTo[k*2]` is negative, the corresponding output channel is filled with zero.

The function is a generalized form of cvcvSplit and cvMerge and some forms of CvtColor . It can be used to change the order of the planes, add/remove alpha channel, extract or insert a single plane or multiple planes etc.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B swapped) and separate alpha channel image:

```
CvMat* rgba = cvCreateMat(100, 100, CV_8UC4);
CvMat* bgr = cvCreateMat(rgba->rows, rgba->cols, CV_8UC3);
CvMat* alpha = cvCreateMat(rgba->rows, rgba->cols, CV_8UC1);
cvSet(rgba, cvScalar(1,2,3,4));

CvArr* out[] = { bgr, alpha };
int from_to[] = { 0,2,  1,1,  2,0,  3,3 };
cvMixChannels(&bgra, 1, out, 2, from_to, 4);
```

MulAddS

Synonym for ScaleAdd .

---

cvMul (view/add comments)

Calculates the per-element product of two arrays.

```
void cvMul(const CvArr* src1, const CvArr* src2, CvArr* dst, double
scale=1);
```

src1 The first source array

src2 The second source array

dst The destination array

scale Optional scale factor

The function calculates the per-element product of two arrays:

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I)$$

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvMulSpectrums (view/add comments)

Performs per-element multiplication of two Fourier spectrums.

```
void cvMulSpectrums(
           const CvArr* src1,
           const CvArr* src2,
           CvArr* dst,
           int flags);
```

src1 The first source array

src2 The second source array

dst The destination array of the same type and the same size as the source arrays

flags A combination of the following values;

- **CV_DXT_ROWS** treats each row of the arrays as a separate spectrum (see cvDFT parameters description).
- **CV_DXT_MUL_CONJ** conjugate the second source array before the multiplication.

The function performs per-element multiplication of the two CCS-packed or complex matrices that are results of a real or complex Fourier transform.

The function, together with cvDFT, may be used to calculate convolution of two arrays rapidly.

cvMulTransposed (view/add comments)

Calculates the product of an array and a transposed array.

```
void cvMulTransposed(const CvArr* src, CvArr* dst, int order, const
CvArr* delta=NULL, double scale=1.0);
```

src The source matrix

dst The destination matrix. Must be `CV_32F` or `CV_64F`.

order Order of multipliers

delta An optional array, subtracted from `src` before multiplication

scale An optional scaling

The function calculates the product of src and its transposition:

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})(\texttt{src} - \texttt{delta})^T$$

if $\texttt{order} = 0$, and

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})^T(\texttt{src} - \texttt{delta})$$

otherwise.

cvNorm (view/add comments)

Calculates absolute array norm, absolute difference norm, or relative difference norm.

```
double cvNorm(const CvArr* arr1, const CvArr* arr2=NULL, int
normType=CV_L2, const CvArr* mask=NULL);
```

arr1 The first source image

arr2 The second source image. If it is NULL, the absolute norm of `arr1` is calculated, otherwise the absolute or relative norm of `arr1`-`arr2` is calculated.

normType Type of norm, see the discussion

mask The optional operation mask

The function calculates the absolute norm of arr1 if arr2 is NULL:

$$norm = \begin{cases} ||\texttt{arr1}||_C = \max_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV_C} \\ ||\texttt{arr1}||_{L1} = \sum_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV_L1} \\ ||\texttt{arr1}||_{L2} = \sqrt{\sum_I \texttt{arr1}(I)^2} & \text{if } \texttt{normType} = \texttt{CV_L2} \end{cases}$$

or the absolute difference norm if arr2 is not NULL:

$$norm = \begin{cases} ||\texttt{arr1} - \texttt{arr2}||_C = \max_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV_C} \\ ||\texttt{arr1} - \texttt{arr2}||_{L1} = \sum_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV_L1} \\ ||\texttt{arr1} - \texttt{arr2}||_{L2} = \sqrt{\sum_I (\texttt{arr1}(I) - \texttt{arr2}(I))^2} & \text{if } \texttt{normType} = \texttt{CV_L2} \end{cases}$$

or the relative difference norm if arr2 is not NULL and (normType & CV_RELATIVE) != 0:

$$norm = \begin{cases} \frac{||\texttt{arr1}-\texttt{arr2}||_C}{||\texttt{arr2}||_C} & \text{if } \texttt{normType} = \texttt{CV_RELATIVE_C} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L1}}{||\texttt{arr2}||_{L1}} & \text{if } \texttt{normType} = \texttt{CV_RELATIVE_L1} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L2}}{||\texttt{arr2}||_{L2}} & \text{if } \texttt{normType} = \texttt{CV_RELATIVE_L2} \end{cases}$$

The function returns the calculated norm. A multiple-channel array is treated as a single-channel, that is, the results for all channels are combined.

---

cvNot (view/add comments)

Performs per-element bit-wise inversion of array elements.

```
void cvNot(const CvArr* src, CvArr* dst);
```

src The source array

dst The destination array

The function Not inverses every bit of every array element:

```
dst(I)=~src(I)
```

cvOr (view/add comments)

Calculates per-element bit-wise disjunction of two arrays.

```
void cvOr(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise disjunction of two arrays:

```
dst(I)=src1(I)|src2(I)
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

cvOrS (view/add comments)

Calculates a per-element bit-wise disjunction of an array and a scalar.

```
void cvOrS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function OrS calculates per-element bit-wise disjunction of an array and a scalar:

```
dst(I)=src(I)|value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cvPerspectiveTransform (view/add comments)

Performs perspective matrix transformation of a vector array.

```
void cvPerspectiveTransform(const CvArr* src, CvArr* dst, const CvMat*
mat);
```

src The source three-channel floating-point array

dst The destination three-channel floating-point array

mat 3×3 or 4×4 transformation matrix

The function transforms every element of `src` (by treating it as 2D or 3D vector) in the following way:

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

---

cvPolarToCart (view/add comments)

Calculates Cartesian coordinates of 2d vectors represented in polar form.

```
void cvPolarToCart(
           const CvArr* magnitude,
           const CvArr* angle,
           CvArr* x,
           CvArr* y,
           int angleInDegrees=0);
```

magnitude The array of magnitudes. If it is NULL, the magnitudes are assumed to be all 1's.

angle The array of angles, whether in radians or degrees

x The destination array of x-coordinates, may be set to NULL if it is not needed

y The destination array of y-coordinates, mau be set to NULL if it is not needed

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the x-coodinate, y-coordinate or both of every vector `magnitude(I)*exp(ang`
`j=sqrt(-1)`:

```
x(I)=magnitude(I)*cos(angle(I)),
y(I)=magnitude(I)*sin(angle(I))
```

---

cvPow (view/add comments)

Raises every array element to a power.

```
void cvPow(
           const CvArr* src,
           CvArr* dst,
           double power);
```

src The source array

dst The destination array, should be the same type as the source

power The exponent of power

The function raises every element of the input array to p:

$$\texttt{dst}[I] = \begin{cases} \texttt{src}(I)^p & \text{if p is integer} \\ |\texttt{src}(I)^p| & \text{otherwise} \end{cases}$$

That is, for a non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following example, computing the cube root of array elements, shows:

```
CvSize size = cvGetSize(src);
CvMat* mask = cvCreateMat(size.height, size.width, CV_8UC1);
cvCmpS(src, 0, mask, CV_CMP_LT); /* find negative elements */
cvPow(src, dst, 1./3);
cvSubRS(dst, cvScalarAll(0), dst, mask); /* negate the results of negative inputs */
cvReleaseMat(&mask);
```

For some values of power, such as integer values, 0.5, and -0.5, specialized faster algorithms are used.

---

cvPtr?D (view/add comments)

Return pointer to a particular array element.

```
uchar* cvPtr1D(const CvArr* arr, int idx0, int* type=NULL);
uchar* cvPtr2D(const CvArr* arr, int idx0, int idx1, int* type=NULL);
uchar* cvPtr3D(const CvArr* arr, int idx0, int idx1, int idx2, int*
type=NULL);
uchar* cvPtrND(const CvArr* arr, int* idx, int* type=NULL, int
createNode=1, unsigned* precalcHashval=NULL);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

type Optional output parameter: type of matrix elements

createNode Optional input parameter for sparse matrices. Non-zero value of the parameter means that the requested element is created if it does not exist already.

precalcHashval Optional input parameter for sparse matrices. If the pointer is not NULL, the function does not recalculate the node hash value, but takes it from the specified location. It is useful for speeding up pair-wise operations (TODO: provide an example)

The functions return a pointer to a specific array element. Number of array dimension should match to the number of indices passed to the function except for `cvPtr1D` function that can be used for sequential access to 1D, 2D or nD dense arrays.

The functions can be used for sparse arrays as well - if the requested node does not exist they create it and set it to zero.

All these as well as other functions accessing array elements (cvGet, cvGetReal, cvSet, cvSetReal) raise an error in case if the element index is out of range.

---

cvRNG (view/add comments)

Initializes a random number generator state.

```
CvRNG cvRNG(int64 seed=-1);
```

seed 64-bit value used to initiate a random sequence

The function initializes a random number generator and returns the state. The pointer to the state can be then passed to the cvRandInt, cvRandReal and cvRandArr functions. In the current implementation a multiply-with-carry generator is used.

---

cvRandArr (view/add comments)

Fills an array with random numbers and updates the RNG state.

```
void cvRandArr(
           CvRNG* rng,
           CvArr* arr,
```

```
            int distType,
            CvScalar param1,
            CvScalar param2);
```

rng RNG state initialized by cvRNG

arr The destination array

distType Distribution type

CV_RAND_UNI uniform distribution

CV_RAND_NORMAL normal or Gaussian distribution

param1 The first parameter of the distribution. In the case of a uniform distribution it is the inclusive lower boundary of the random numbers range. In the case of a normal distribution it is the mean value of the random numbers.

param2 The second parameter of the distribution. In the case of a uniform distribution it is the exclusive upper boundary of the random numbers range. In the case of a normal distribution it is the standard deviation of the random numbers.

The function fills the destination array with uniformly or normally distributed random numbers.

In the example below, the function is used to add a few normally distributed floating-point numbers to random locations within a 2d array.

```
/* let noisy_screen be the floating-point 2d array that is to be "crapped" */
CvRNG rng_state = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* allocate the array of coordinates of points */
CvMat* locations = cvCreateMat(pointCount, 1, CV_32SC2);
/* arr of random point values */
CvMat* values = cvCreateMat(pointCount, 1, CV_32FC1);
CvSize size = cvGetSize(noisy_screen);

/* initialize the locations */
cvRandArr(&rng_state, locations, CV_RAND_UNI, cvScalar(0,0,0,0),
          cvScalar(size.width,size.height,0,0));

/* generate values */
cvRandArr(&rng_state, values, CV_RAND_NORMAL,
          cvRealScalar(100), // average intensity
          cvRealScalar(30) // deviation of the intensity
```

```
          );

/* set the points */
for(i = 0; i < pointCount; i++ )
{
    CvPoint pt = *(CvPoint*)cvPtr1D(locations, i, 0);
    float value = *(float*)cvPtr1D(values, i, 0);
    *((float*)cvPtr2D(noisy_screen, pt.y, pt.x, 0 )) += value;
}

/* not to forget to release the temporary arrays */
cvReleaseMat(&locations);
cvReleaseMat(&values);

/* RNG state does not need to be deallocated */
```

---

cvRandInt (view/add comments)

Returns a 32-bit unsigned integer and updates RNG.

```
unsigned cvRandInt(CvRNG* rng);
```

rng RNG state initialized by `RandInit` and, optionally, customized by `RandSetRange` (though, the latter function does not affect the discussed function outcome)

The function returns a uniformly-distributed random 32-bit unsigned integer and updates the RNG state. It is similar to the rand() function from the C runtime library, but it always generates a 32-bit number whereas rand() returns a number in between 0 and `RAND_MAX` which is 2^{16} or 2^{32}, depending on the platform.

The function is useful for generating scalar random numbers, such as points, patch sizes, table indices, etc., where integer numbers of a certain range can be generated using a modulo operation and floating-point numbers can be generated by scaling from 0 to 1 or any other specific range.

Here is the example from the previous function discussion rewritten using cvRandInt:

```
/* the input and the task is the same as in the previous sample. */
CvRNG rnggstate = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* ... - no arrays are allocated here */
CvSize size = cvGetSize(noisygscreen);
```

```
/* make a buffer for normally distributed numbers to reduce call overhead */
#define bufferSize 16
float normalValueBuffer[bufferSize];
CvMat normalValueMat = cvMat(bufferSize, 1, CVg32F, normalValueBuffer);
int valuesLeft = 0;

for(i = 0; i < pointCount; i++ )
{
    CvPoint pt;
    /* generate random point */
    pt.x = cvRandInt(&rnggstate ) % size.width;
    pt.y = cvRandInt(&rnggstate ) % size.height;

    if(valuesLeft <= 0 )
    {
        /* fulfill the buffer with normally distributed numbers
           if the buffer is empty */
        cvRandArr(&rnggstate, &normalValueMat, CV\_RAND\_NORMAL,
                  cvRealScalar(100), cvRealScalar(30));
        valuesLeft = bufferSize;
    }
    *((float*)cvPtr2D(noisygscreen, pt.y, pt.x, 0 ) =
                               normalValueBuffer[--valuesLeft];
}

/* there is no need to deallocate normalValueMat because we have
both the matrix header and the data on stack. It is a common and efficient
practice of working with small, fixed-size matrices */
```

---

cvRandReal (view/add comments)

Returns a floating-point random number and updates RNG.

```
double cvRandReal(CvRNG* rng);
```

rng RNG state initialized by cvRNG

The function returns a uniformly-distributed random floating-point number between 0 and 1 (1 is not included).

cvReduce (view/add comments)

Reduces a matrix to a vector.

```
void cvReduce(const CvArr* src, CvArr* dst, int dim = -1, int
op=CV_REDUCE_SUM);
```

src The input matrix.

dst The output single-row/single-column vector that accumulates somehow all the matrix rows/-columns.

dim The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row, 1 means that the matrix is reduced to a single column and -1 means that the dimension is chosen automatically by analysing the dst size.

op The reduction operation. It can take of the following values:

- **CV_REDUCE_SUM** The output is the sum of all of the matrix's rows/columns.
- **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
- **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/-columns.
- **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/-columns.

The function reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In the case of CV_REDUCE_SUM and CV_REDUCE_AVG the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

cvReleaseData (view/add comments)

Releases array data.

```
void cvReleaseData(CvArr* arr);
```

arr Array header

The function releases the array data. In the case of CvMat or CvMatND it simply calls cvDecRefData(), that is the function can not deallocate external data. See also the note to cvCreateData.

---

cvReleaseImage (view/add comments)

Deallocates the image header and the image data.

```
void cvReleaseImage(IplImage** image);
```

image Double pointer to the image header

This call is a shortened form of

```
if(*image )
{
    cvReleaseData(*image);
    cvReleaseImageHeader(image);
}
```

---

cvReleaseImageHeader (view/add comments)

Deallocates an image header.

```
void cvReleaseImageHeader(IplImage** image);
```

image Double pointer to the image header

This call is an analogue of

```
if(image )
{
    iplDeallocate(*image, IPL_IMAGE_HEADER | IPL_IMAGE_ROI);
    *image = 0;
}
```

but it does not use IPL functions by default (see the `CV_TURN_ON_IPL_COMPATIBILITY` macro).

---

cvReleaseMat (view/add comments)

Deallocates a matrix.

```
void cvReleaseMat(CvMat** mat);
```

mat Double pointer to the matrix

The function decrements the matrix data reference counter and deallocates matrix header. If the data reference counter is 0, it also deallocates the data.

```
if(*mat )
    cvDecRefData(*mat);
cvFree((void**)mat);
```

---

cvReleaseMatND (view/add comments)

Deallocates a multi-dimensional array.

```
void cvReleaseMatND(CvMatND** mat);
```

mat Double pointer to the array

The function decrements the array data reference counter and releases the array header. If the reference counter reaches 0, it also deallocates the data.

```
if(*mat )
    cvDecRefData(*mat);
cvFree((void**)mat);
```

cvReleaseSparseMat (view/add comments)

Deallocates sparse array.

```
void cvReleaseSparseMat(CvSparseMat** mat);
```

mat Double pointer to the array

The function releases the sparse array and clears the array pointer upon exit.

cvRepeat (view/add comments)

Fill the destination array with repeated copies of the source array.

```
void cvRepeat(const CvArr* src, CvArr* dst);
```

src Source array, image or matrix

dst Destination array, image or matrix

The function fills the destination array with repeated copies of the source array:

```
dst(i,j)=src(i mod rows(src), j mod cols(src))
```

So the destination array may be as larger as well as smaller than the source array.

cvResetImageROI (view/add comments)

Resets the image ROI to include the entire image and releases the ROI structure.

```
void cvResetImageROI(IplImage* image);
```

image A pointer to the image header

This produces a similar result to the following , but in addition it releases the ROI structure.

```
cvSetImageROI(image, cvRect(0, 0, image->width, image->height ));
cvSetImageCOI(image, 0);
```

---

cvReshape (view/add comments)

Changes shape of matrix/image without copying data.

```
CvMat* cvReshape(const CvArr* arr, CvMat* header, int newCn, int
newRows=0);
```

arr Input array

header Output header to be filled

newCn New number of channels. 'newCn = 0' means that the number of channels remains unchanged.

newRows New number of rows. 'newRows = 0' means that the number of rows remains unchanged unless it needs to be changed according to `newCn` value.

The function initializes the CvMat header so that it points to the same data as the original array but has a different shape - different number of channels, different number of rows, or both.

The following example code creates one image buffer and two image headers, the first is for a 320x240x3 image and the second is for a 960x240x1 image:

```
IplImage* color_img = cvCreateImage(cvSize(320,240), IPL_DEPTH_8U, 3);
CvMat gray_mat_hdr;
IplImage gray_img_hdr, *gray_img;
cvReshape(color_img, &gray_mat_hdr, 1);
gray_img = cvGetImage(&gray_mat_hdr, &gray_img_hdr);
```

And the next example converts a 3x3 matrix to a single 1x9 vector:

```
CvMat* mat = cvCreateMat(3, 3, CV_32F);
CvMat row_header, *row;
row = cvReshape(mat, &row_header, 0, 1);
```

cvReshapeMatND (view/add comments)

Changes the shape of a multi-dimensional array without copying the data.

```
CvArr* cvReshapeMatND(const CvArr* arr, int sizeofHeader, CvArr*
header, int newCn, int newDims, int* newSizes);
```

```
#define cvReshapeND(arr, header, newCn, newDims, newSizes )    \
      cvReshapeMatND((arr), sizeof(*(header)), (header),          \
                     (newCn), (newDims), (newSizes))
```

arr Input array

sizeofHeader Size of output header to distinguish between IplImage, CvMat and CvMatND output headers

header Output header to be filled

newCn New number of channels. newCn $= 0$ means that the number of channels remains unchanged.

newDims New number of dimensions. newDims $= 0$ means that the number of dimensions remains the same.

newSizes Array of new dimension sizes. Only newDims $- 1$ values are used, because the total number of elements must remain the same. Thus, if newDims $= 1$, newSizes array is not used.

The function is an advanced version of cvReshape that can work with multi-dimensional arrays as well (though it can work with ordinary images and matrices) and change the number of dimensions.

Below are the two samples from the cvReshape description rewritten using cvReshapeMatND:

```
IplImage* color_img = cvCreateImage(cvSize(320,240), IPL_DEPTH_8U, 3);
IplImage gray_img_hdr, *gray_img;
gray_img = (IplImage*)cvReshapeND(color_img, &gray_img_hdr, 1, 0, 0);

...
```

```
/* second example is modified to convert 2x2x2 array to 8x1 vector */
int size[] = { 2, 2, 2 };
CvMatND* mat = cvCreateMatND(3, size, CV_32F);
CvMat row_header, *row;
row = (CvMat*)cvReshapeND(mat, &row_header, 0, 1, 0);
```

---

cvRound, cvFloor, cvCeil (view/add comments)

Converts a floating-point number to an integer.

```
int cvRound(double value); int cvFloor(double value); int cvCeil(double
value);
```

value The input floating-point value

The functions convert the input floating-point number to an integer using one of the rounding modes. `Round` returns the nearest integer value to the argument. `Floor` returns the maximum integer value that is not larger than the argument. `Ceil` returns the minimum integer value that is not smaller than the argument. On some architectures the functions work much faster than the standard cast operations in C. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

---

cvScaleAdd (view/add comments)

Calculates the sum of a scaled array and another array.

```
void cvScaleAdd(const CvArr* src1, CvScalar scale, const CvArr* src2,
CvArr* dst);
```

src1 The first source array

scale Scale factor for the first array

src2 The second source array

dst The destination array

The function calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale}\,\texttt{src1}(I) + \texttt{src2}(I)$$

All array parameters should have the same type and the same size.

---

cvSet (view/add comments)

Sets every element of an array to a given value.

```
void cvSet(CvArr* arr, CvScalar value, const CvArr* mask=NULL);
```

arr The destination array

value Fill value

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies the scalar `value` to every selected element of the destination array:

$$\texttt{arr}(I) = \texttt{value} \quad \text{if} \quad \texttt{mask}(I) \neq 0$$

If array `arr` is of `IplImage` type, then is ROI used, but COI must not be set.

---

cvSet?D (view/add comments)

Change the particular array element.

```
void cvSet1D(CvArr* arr, int idx0, CvScalar value);
void cvSet2D(CvArr* arr, int idx0, int idx1, CvScalar value);
void cvSet3D(CvArr* arr, int idx0, int idx1, int idx2, CvScalar value);
void cvSetND(CvArr* arr, int* idx, CvScalar value);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

value The assigned value

The functions assign the new value to a particular array element. In the case of a sparse array the functions create the node if it does not exist yet.

---

cvSetData (view/add comments)

Assigns user data to the array header.

```
void cvSetData(CvArr* arr, void* data, int step);
```

arr Array header

data User data

step Full row length in bytes

The function assigns user data to the array header. Header should be initialized before using `cvCreate*Header`, `cvInit*Header` or cvMat (in the case of matrix) function.

---

cvSetIdentity (view/add comments)

Initializes a scaled identity matrix.

```
void cvSetIdentity(CvArr* mat, CvScalar value=cvRealScalar(1));
```

mat The matrix to initialize (not necesserily square)

value The value to assign to the diagonal elements

The function initializes a scaled identity matrix:

$$\texttt{arr}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

---

cvSetImageCOI (view/add comments)

Sets the channel of interest in an IplImage.

```
void cvSetImageCOI(
           IplImage* image,
           int coi);
```

image A pointer to the image header

coi The channel of interest. 0 - all channels are selected, 1 - first channel is selected, etc. Note that the channel indices become 1-based.

If the ROI is set to NULL and the coi is *not* 0, the ROI is allocated. Most OpenCV functions do *not* support the COI setting, so to process an individual image/matrix channel one may copy (via cvCopy or cvSplit) the channel to a separate image/matrix, process it and then copy the result back (via cvCopy or cvMerge) if needed.

---

cvSetImageROI (view/add comments)

Sets an image Region Of Interest (ROI) for a given rectangle.

```
void cvSetImageROI(
           IplImage* image,
           CvRect rect);
```

image A pointer to the image header

rect The ROI rectangle

If the original image ROI was NULL and the rect is not the whole image, the ROI structure is allocated.

Most OpenCV functions support the use of ROI and treat the image rectangle as a separate image. For example, all of the pixel coordinates are counted from the top-left (or bottom-left) corner of the ROI, not the original image.

---

cvSetReal?D (view/add comments)

Change a specific array element.

```
void cvSetReal1D(CvArr* arr, int idx0, double value);
void cvSetReal2D(CvArr* arr, int idx0, int idx1, double value);
void cvSetReal3D(CvArr* arr, int idx0, int idx1, int idx2, double
value);
void cvSetRealND(CvArr* arr, int* idx, double value);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

value The assigned value

The functions assign a new value to a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that the cvSet*D function can be used safely for both single-channel and multiple-channel arrays, though they are a bit slower.

In the case of a sparse array the functions create the node if it does not yet exist.

---

cvSetZero (view/add comments)

Clears the array.

```
void cvSetZero(CvArr* arr);
```

```
#define cvZero cvSetZero
```

arr Array to be cleared

The function clears the array. In the case of dense arrays (CvMat, CvMatND or IplImage), cvZero(array) is equivalent to cvSet(array,cvScalarAll(0),0). In the case of sparse arrays all the elements are removed.

---

cvSolve (view/add comments)

Solves a linear system or least-squares problem.

```
int cvSolve(const CvArr* src1, const CvArr* src2, CvArr* dst, int
method=CV_LU);
```

A The source matrix

B The right-hand part of the linear system

X The output solution

method The solution (matrix inversion) method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix.

The function solves a linear system or least-squares problem (the latter is possible with SVD methods):

$$\texttt{dst} = argmin_X ||\texttt{src1}\,\texttt{X} - \texttt{src2}||$$

If CV_LU method is used, the function returns 1 if src1 is non-singular and 0 otherwise; in the latter case dst is not valid.

cvSolveCubic (view/add comments)

Finds the real roots of a cubic equation.

```
void cvSolveCubic(const CvArr* coeffs, CvArr* roots);
```

coeffs The equation coefficients, an array of 3 or 4 elements

roots The output array of real roots which should have 3 elements

The function finds the real roots of a cubic equation:
If coeffs is a 4-element vector:

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or if coeffs is 3-element vector:

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

The function returns the number of real roots found. The roots are stored to `root` array, which is padded with zeros if there is only one root.

cvSplit (view/add comments)

Divides multi-channel array into several single-channel arrays or extracts a single channel from the array.

```
void cvSplit(const CvArr* src, CvArr* dst0, CvArr* dst1, CvArr* dst2,
CvArr* dst3);
```

src Source array

dst0 Destination channel 0

dst1 Destination channel 1

dst2 Destination channel 2

dst3 Destination channel 3

The function divides a multi-channel array into separate single-channel arrays. Two modes are available for the operation. If the source array has N channels then if the first N destination channels are not NULL, they all are extracted from the source array; if only a single destination channel of the first N is not NULL, this particular channel is extracted; otherwise an error is raised. The rest of the destination channels (beyond the first N) must always be NULL. For IplImage cvCopy with COI set can be also used to extract a single channel from the image.

---

cvSqrt (view/add comments)

Calculates the square root.

```
float cvSqrt(float value);
```

value The input floating-point value

The function calculates the square root of the argument. If the argument is negative, the result is not determined.

---

cvSub (view/add comments)

Computes the per-element difference between two arrays.

```
void cvSub(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts one array from another one:

```
dst(I)=src1(I)-src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvSubRS (view/add comments)

Computes the difference between a scalar and an array.

```
void cvSubRS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The first source array

value Scalar to subtract from

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts every element of source array from a scalar:

```
dst(I)=value-src(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvSubS (view/add comments)

Computes the difference between an array and a scalar.

```
void cvSubS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Subtracted scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts a scalar from every element of the source array:

```
dst(I)=src(I)-value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvSum (view/add comments)

Adds up array elements.

```
CvScalar cvSum(const CvArr* arr);
```

arr The array

The function calculates the sum `S` of array elements, independently for each channel:

$$\sum_I \texttt{arr}(I)_c$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the sum to the first scalar component.

---

cvSVBkSb (view/add comments)

Performs singular value back substitution.

```
void cvSVBkSb(
            const CvArr* W,
            const CvArr* U,
            const CvArr* V,
```

```
            const CvArr* B,
            CvArr* X,
            int flags);
```

W Matrix or vector of singular values

U Left orthogonal matrix (tranposed, perhaps)

V Right orthogonal matrix (tranposed, perhaps)

B The matrix to multiply the pseudo-inverse of the original matrix `A` by. This is an optional parameter. If it is omitted then it is assumed to be an identity matrix of an appropriate size (so that `X` will be the reconstructed pseudo-inverse of `A`).

X The destination matrix: result of back substitution

flags Operation flags, should match exactly to the `flags` passed to cvSVD

The function calculates back substitution for decomposed matrix `A` (see cvSVD description) and matrix `B`:

$$\mathtt{X} = \mathtt{V}\mathtt{W}^{-1}\mathtt{U}^T\mathtt{B}$$

where

$$W^{-1}_{(i,i)} = \begin{cases} 1/W_{(i,i)} & \text{if } W_{(i,i)} > \epsilon \sum_i W_{(i,i)} \\ 0 & \text{otherwise} \end{cases}$$

and ϵ is a small number that depends on the matrix data type.

This function together with cvSVD is used inside cvInvert and cvSolve, and the possible reason to use these (svd and bksb) "low-level" function, is to avoid allocation of temporary matrices inside the high-level counterparts (inv and solve).

---

cvSVD (view/add comments)

Performs singular value decomposition of a real floating-point matrix.

```
void cvSVD(
           CvArr* A,
           CvArr* W,
           CvArr* U=NULL,
           CvArr* V=NULL,
           int flags=0);
```

A Source $\mathtt{M} \times \mathtt{N}$ matrix

W Resulting singular value diagonal matrix ($\mathtt{M} \times \mathtt{N}$ or $\min(\mathtt{M}, \mathtt{N}) \times \min(\mathtt{M}, \mathtt{N})$) or $\min(\mathtt{M}, \mathtt{N}) \times 1$ vector of the singular values

U Optional left orthogonal matrix, $\mathtt{M} \times \min(\mathtt{M}, \mathtt{N})$ (when CV_SVD_U_T is not set), or $\min(\mathtt{M}, \mathtt{N}) \times \mathtt{M}$ (when CV_SVD_U_T is set), or $\mathtt{M} \times \mathtt{M}$ (regardless of CV_SVD_U_T flag).

V Optional right orthogonal matrix, $\mathtt{N} \times \min(\mathtt{M}, \mathtt{N})$ (when CV_SVD_V_T is not set), or $\min(\mathtt{M}, \mathtt{N}) \times \mathtt{N}$ (when CV_SVD_V_T is set), or $\mathtt{N} \times \mathtt{N}$ (regardless of CV_SVD_V_T flag).

flags Operation flags; can be 0 or a combination of the following values:

- **CV_SVD_MODIFY_A** enables modification of matrix A during the operation. It speeds up the processing.
- **CV_SVD_U_T** means that the transposed matrix U is returned. Specifying the flag speeds up the processing.
- **CV_SVD_V_T** means that the transposed matrix V is returned. Specifying the flag speeds up the processing.

The function decomposes matrix A into the product of a diagonal matrix and two orthogonal matrices:

$$A = U\, W\, V^T$$

where W is a diagonal matrix of singular values that can be coded as a 1D vector of singular values and U and V. All the singular values are non-negative and sorted (together with U and V columns) in descending order.

An SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix A is a square, symmetric, and positively defined matrix, for example, when it is a covariance matrix. W in this case will be a vector/-matrix of the eigenvalues, and $U = V$ will be a matrix of the eigenvectors.

- accurate solution of a poor-conditioned linear system.
- least-squares solution of an overdetermined linear system. This and the preceeding is done by using the cvSolve function with the `CV_SVD` method.
- accurate calculation of different matrix characteristics such as the matrix rank (the number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), and determinant (absolute value of the determinant is equal to the product of singular values).

---

cvTrace (view/add comments)

Returns the trace of a matrix.

```
CvScalar cvTrace(const CvArr* mat);
```

mat The source matrix

The function returns the sum of the diagonal elements of the matrix `src1`.

$$tr(\texttt{mat}) = \sum_i \texttt{mat}(i,i)$$

---

cvTransform (view/add comments)

Performs matrix transformation of every array element.

```
void cvTransform(const CvArr* src, CvArr* dst, const CvMat* transmat,
const CvMat* shiftvec=NULL);
```

src The first source array

dst The destination array

transmat Transformation matrix

shiftvec Optional shift vector

The function performs matrix transformation of every element of array src and stores the results in dst:

$$dst(I) = transmat \cdot src(I) + shiftvec$$

That is, every element of an N-channel array src is considered as an N-element vector which is transformed using a M × N matrix transmat and shift vector shiftvec into an element of M-channel array dst. There is an option to embedd shiftvec into transmat. In this case transmat should be a M × (N + 1) matrix and the rightmost column is treated as the shift vector.

Both source and destination arrays should have the same depth and the same size or selected ROI size. transmat and shiftvec should be real floating-point matrices.

The function may be used for geometrical transformation of n dimensional point set, arbitrary linear color space transformation, shuffling the channels and so forth.

---

cvTranspose (view/add comments)

Transposes a matrix.

```
void cvTranspose(const CvArr* src, CvArr* dst);
```

src The source matrix

dst The destination matrix

The function transposes matrix src1:

$$\mathtt{dst}(i,j) = \mathtt{src}(j,i)$$

Note that no complex conjugation is done in the case of a complex matrix. Conjugation should be done separately: look at the sample code in cvXorS for an example.

---

cvXor (view/add comments)

Performs per-element bit-wise ”exclusive or” operation on two arrays.

```
void cvXor(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)^src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cvXorS (view/add comments)

Performs per-element bit-wise ”exclusive or” operation on an array and a scalar.

```
void cvXorS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function XorS calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)^value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

The following sample demonstrates how to conjugate complex vector by switching the most-significant bit of imaging part:

```
float a[] = { 1, 0, 0, 1, -1, 0, 0, -1 }; /* 1, j, -1, -j */
CvMat A = cvMat(4, 1, CV\_32FC2, &a);
int i, negMask = 0x80000000;
cvXorS(&A, cvScalar(0, *(float*)&negMask, 0, 0 ), &A, 0);
for(i = 0; i < 4; i++ )
    printf("(\%.1f, \%.1f) ", a[i*2], a[i*2+1]);
```

The code should print:

```
(1.0,0.0) (0.0,-1.0) (-1.0,0.0) (0.0,1.0)
```

---

cvmGet (view/add comments)

Returns the particular element of single-channel floating-point matrix.

```
double cvmGet(const CvMat* mat, int row, int col);
```

mat Input matrix

row The zero-based index of row

col The zero-based index of column

The function is a fast replacement for cvGetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

---

cvmSet (view/add comments)

Returns a specific element of a single-channel floating-point matrix.

```
void cvmSet(CvMat* mat, int row, int col, double value);
```

mat The matrix

row The zero-based index of row

col The zero-based index of column

value The new value of the matrix element

The function is a fast replacement for cvSetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

1.3 Dynamic Structures

CvMemStorage (view/add comments)

Growing memory storage.

```
typedef struct CvMemStorage
{
    struct CvMemBlock* bottom;/* first allocated block */
    struct CvMemBlock* top; /* the current memory block - top of the stack */
    struct CvMemStorage* parent; /* borrows new blocks from */
    int block\_size; /* block size */
    int free\_space; /* free space in the \texttt{top} block (in bytes) */
} CvMemStorage;
```

Memory storage is a low-level structure used to store dynamicly growing data structures such as sequences, contours, graphs, subdivisions, etc. It is organized as a list of memory blocks of equal size - `bottom` field is the beginning of the list of blocks and `top` is the currently used block, but not necessarily the last block of the list. All blocks between `bottom` and `top`, not including the latter, are considered fully occupied; all blocks between `top` and the last block, not including `top`, are considered free and `top` itself is partly ocupied - `free_space` contains the number of free bytes left in the end of `top`.

A new memory buffer that may be allocated explicitly by cvMemStorageAlloc function or implicitly by higher-level functions, such as cvSeqPush, cvGraphAddEdge, etc., `always` starts in the end of the current block if it fits there. After allocation, `free_space` is decremented by the

size of the allocated buffer plus some padding to keep the proper alignment. When the allocated buffer does not fit into the available portion of `top`, the next storage block from the list is taken as `top` and `free_space` is reset to the whole block size prior to the allocation.

If there are no more free blocks, a new block is allocated (or borrowed from the parent, see cvCreateChildMemStorage) and added to the end of list. Thus, the storage behaves as a stack with `bottom` indicating bottom of the stack and the pair (`top`, `free_space`) indicating top of the stack. The stack top may be saved via cvSaveMemStoragePos, restored via cvRestoreMemStoragePos, or reset via cvClearStorage.

CvMemBlock (view/add comments)

Memory storage block.

```
typedef struct CvMemBlock
{
    struct CvMemBlock* prev;
    struct CvMemBlock* next;
} CvMemBlock;
```

The structure CvMemBlock represents a single block of memory storage. The actual data in the memory blocks follows the header, that is, the i_{th} byte of the memory block can be retrieved with the expression `((char*)(mem_block_ptr+1))[i]`. However, there is normally no need to access the storage structure fields directly.

CvMemStoragePos (view/add comments)

Memory storage position.

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
    int free\_space;
} CvMemStoragePos;
```

The structure described above stores the position of the stack top that can be saved via cvSaveMemStoragePos and restored via cvRestoreMemStoragePos.

CvSeq (view/add comments)

Growable sequence of elements.

```
#define CV_SEQUENCE\_FIELDS() \
    int flags; /* micsellaneous flags */ \
```

```
    int header_size; /* size of sequence header */ \
    struct CvSeq* h_prev; /* previous sequence */ \
    struct CvSeq* h_next; /* next sequence */ \
    struct CvSeq* v_prev; /* 2nd previous sequence */ \
    struct CvSeq* v_next; /* 2nd next sequence */ \
    int total; /* total number of elements */ \
    int elem_size;/* size of sequence element in bytes */ \
    char* block_max;/* maximal bound of the last block */ \
    char* ptr; /* current write pointer */ \
    int delta_elems; /* how many elements allocated when the sequence grows
                        (sequence granularity) */ \
    CvMemStorage* storage; /* where the seq is stored */ \
    CvSeqBlock* free_blocks; /* free blocks list */ \
    CvSeqBlock* first; /* pointer to the first sequence block */

typedef struct CvSeq
{
    CV_SEQUENCE_FIELDS()
} CvSeq;
```

The structure CvSeq is a base for all of OpenCV dynamic data structures.

Such an unusual definition via a helper macro simplifies the extension of the structure CvSeq with additional parameters. To extend CvSeq the user may define a new structure and put user-defined fields after all CvSeq fields that are included via the macro CV_SEQUENCE_FIELDS().

There are two types of sequences - dense and sparse. The base type for dense sequences is CvSeq and such sequences are used to represent growable 1d arrays - vectors, stacks, queues, and deques. They have no gaps in the middle - if an element is removed from the middle or inserted into the middle of the sequence, the elements from the closer end are shifted. Sparse sequences have CvSet as a base class and they are discussed later in more detail. They are sequences of nodes; each may be either occupied or free as indicated by the node flag. Such sequences are used for unordered data structures such as sets of elements, graphs, hash tables and so forth.

The field header_size contains the actual size of the sequence header and should be greater than or equal to sizeof(CvSeq).

The fields h_prev, h_next, v_prev, v_next can be used to create hierarchical structures from separate sequences. The fields h_prev and h_next point to the previous and the next sequences on the same hierarchical level, while the fields v_prev and v_next point to the previous and the next sequences in the vertical direction, that is, the parent and its first child. But these are just names and the pointers can be used in a different way.

The field first points to the first sequence block, whose structure is described below.

The field total contains the actual number of dense sequence elements and number of allocated nodes in a sparse sequence.

The field `flags` contains the particular dynamic type signature (`CV_SEQ_MAGIC_VAL` for dense sequences and `CV_SET_MAGIC_VAL` for sparse sequences) in the highest 16 bits and miscellaneous information about the sequence. The lowest `CV_SEQ_ELTYPE_BITS` bits contain the ID of the element type. Most of sequence processing functions do not use element type but rather element size stored in `elem_size`. If a sequence contains the numeric data for one of the CvMat type then the element type matches to the corresponding CvMat element type, e.g., `CV_32SC2` may be used for a sequence of 2D points, `CV_32FC1` for sequences of floating-point values, etc. A `CV_SEQ_ELTYPE(seq_header_ptr)` macro retrieves the type of sequence elements. Processing functions that work with numerical sequences check that `elem_size` is equal to that calculated from the type element size. Besides CvMat compatible types, there are few extra element types defined in the `cvtypes.h` header:

Standard Types of Sequence Elements

```
#define CV_SEQ_ELTYPE_POINT          CV_32SC2  /* (x,y) */
#define CV_SEQ_ELTYPE_CODE           CV_8UC1   /* freeman code: 0..7 */
#define CV_SEQ_ELTYPE_GENERIC        0 /* unspecified type of
                                        sequence elements */
#define CV_SEQ_ELTYPE_PTR            CV_USRTYPE1 /* =6 */
#define CV_SEQ_ELTYPE_PPOINT         CV_SEQ_ELTYPE_PTR  /* &elem: pointer to
                                                  element of other sequence */
#define CV_SEQ_ELTYPE_INDEX          CV_32SC1  /* #elem: index of element of
                                                  some other sequence */
#define CV_SEQ_ELTYPE_GRAPH_EDGE     CV_SEQ_ELTYPE_GENERIC  /* &next_o,
                                                  &next_d, &vtx_o, &vtx_d */
#define CV_SEQ_ELTYPE_GRAPH_VERTEX   CV_SEQ_ELTYPE_GENERIC  /* first_edge,
                                                  &(x,y) */
#define CV_SEQ_ELTYPE_TRIAN_ATR      CV_SEQ_ELTYPE_GENERIC  /* vertex of the
                                                  binary tree   */
#define CV_SEQ_ELTYPE_CONNECTED_COMP CV_SEQ_ELTYPE_GENERIC  /* connected
                                                  component  */
#define CV_SEQ_ELTYPE_POINT3D        CV_32FC3  /* (x,y,z)  */
```

The next `CV_SEQ_KIND_BITS` bits specify the kind of sequence:

Standard Kinds of Sequences

```
/* generic (unspecified) kind of sequence */
#define CV_SEQ_KIND_GENERIC     (0 << CV_SEQ_ELTYPE_BITS)

/* dense sequence suntypes */
#define CV_SEQ_KIND_CURVE       (1 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_BIN_TREE    (2 << CV_SEQ_ELTYPE_BITS)

/* sparse sequence (or set) subtypes */
```

```
#define CV_SEQ_KIND_GRAPH       (3 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_SUBDIV2D    (4 << CV_SEQ_ELTYPE_BITS)
```

The remaining bits are used to identify different features specific to certain sequence kinds and element types. For example, curves made of points `(CV_SEQ_KIND_CURVE|CV_SEQ_ELTYPE_POINT)`, together with the flag `CV_SEQ_FLAG_CLOSED`, belong to the type `CV_SEQ_POLYGON` or, if other flags are used, to its subtype. Many contour processing functions check the type of the input sequence and report an error if they do not support this type. The file `cvtypes.h` stores the complete list of all supported predefined sequence types and helper macros designed to get the sequence type of other properties. The definition of the building blocks of sequences can be found below.

CvSeqBlock (view/add comments)

Continuous sequence block.

```
typedef struct CvSeqBlock
{
    struct CvSeqBlock* prev; /* previous sequence block */
    struct CvSeqBlock* next; /* next sequence block */
    int start_index; /* index of the first element in the block +
    sequence->first->start_index */
    int count; /* number of elements in the block */
    char* data; /* pointer to the first element of the block */
} CvSeqBlock;
```

Sequence blocks make up a circular double-linked list, so the pointers `prev` and `next` are never `NULL` and point to the previous and the next sequence blocks within the sequence. It means that `next` of the last block is the first block and `prev` of the first block is the last block. The fields `startIndex` and `count` help to track the block location within the sequence. For example, if the sequence consists of 10 elements and splits into three blocks of 3, 5, and 2 elements, and the first block has the parameter `startIndex = 2`, then pairs `(startIndex, count)` for the sequence blocks are (2,3), (5, 5), and (10, 2) correspondingly. The parameter `startIndex` of the first block is usually `0` unless some elements have been inserted at the beginning of the sequence.

CvSlice (view/add comments)

A sequence slice.

```
typedef struct CvSlice
{
    int start_index;
    int end_index;
} CvSlice;
```

```
inline CvSlice cvSlice( int start, int end );
#define CV_WHOLE_SEQ_END_INDEX 0x3fffffff
#define CV_WHOLE_SEQ  cvSlice(0, CV_WHOLE_SEQ_END_INDEX)

/* calculates the sequence slice length */
int cvSliceLength( CvSlice slice, const CvSeq* seq );
```

Some of functions that operate on sequences take a `CvSlice slice` parameter that is often set to the whole sequence (CV_WHOLE_SEQ) by default. Either of the `startIndex` and `endIndex` may be negative or exceed the sequence length, `startIndex` is inclusive, and `endIndex` is an exclusive boundary. If they are equal, the slice is considered empty (i.e., contains no elements). Because sequences are treated as circular structures, the slice may select a few elements in the end of a sequence followed by a few elements at the beginning of the sequence. For example, `cvSlice(-2, 3)` in the case of a 10-element sequence will select a 5-element slice, containing the pre-last (8th), last (9th), the very first (0th), second (1th) and third (2nd) elements. The functions normalize the slice argument in the following way: first, cvSliceLength is called to determine the length of the slice, then, `startIndex` of the slice is normalized similarly to the argument of cvGetSeqElem (i.e., negative indices are allowed). The actual slice to process starts at the normalized `startIndex` and lasts cvSliceLength elements (again, assuming the sequence is a circular structure).

If a function does not accept a slice argument, but you want to process only a part of the sequence, the sub-sequence may be extracted using the cvSeqSlice function, or stored into a continuous buffer with CvtSeqToArray (optionally, followed by cvMakeSeqHeaderForArray).

CvSet (view/add comments)

Collection of nodes.

```
typedef struct CvSetElem
{
    int flags; /* it is negative if the node is free and zero or positive otherwise */
    struct CvSetElem* next_free; /* if the node is free, the field is a
                                    pointer to next free node */
}
CvSetElem;

#define CV_SET_FIELDS()    \
    CV_SEQUENCE_FIELDS()   /* inherits from [#CvSeq CvSeq] */ \
    struct CvSetElem* free_elems; /* list of free nodes */

typedef struct CvSet
{
```

```
    CV_SET_FIELDS()
} CvSet;
```

The structure CvSet is a base for OpenCV sparse data structures.

As follows from the above declaration, CvSet inherits from CvSeq and it adds the `free_elems` field, which is a list of free nodes, to it. Every set node, whether free or not, is an element of the underlying sequence. While there are no restrictions on elements of dense sequences, the set (and derived structures) elements must start with an integer field and be able to fit CvSetElem structure, because these two fields (an integer followed by a pointer) are required for the organization of a node set with the list of free nodes. If a node is free, the `flags` field is negative (the most-significant bit, or MSB, of the field is set), and the `next_free` points to the next free node (the first free node is referenced by the `free_elems` field of CvSet). And if a node is occupied, the `flags` field is positive and contains the node index that may be retrieved using the (`set_elem->flags & CV_SET_ELEM_IDX_MASK`) expressions, the rest of the node content is determined by the user. In particular, the occupied nodes are not linked as the free nodes are, so the second field can be used for such a link as well as for some different purpose. The macro `CV_IS_SET_ELEM(set_elem_ptr)` can be used to determined whether the specified node is occupied or not.

Initially the set and the list are empty. When a new node is requested from the set, it is taken from the list of free nodes, which is then updated. If the list appears to be empty, a new sequence block is allocated and all the nodes within the block are joined in the list of free nodes. Thus, the `total` field of the set is the total number of nodes both occupied and free. When an occupied node is released, it is added to the list of free nodes. The node released last will be occupied first.

In OpenCV CvSet is used for representing graphs (CvGraph), sparse multi-dimensional arrays (CvSparseMat), and planar subdivisions CvSubdiv2D .

CvGraph (view/add comments)

Oriented or unoriented weighted graph.

```
#define CV_GRAPH_VERTEX_FIELDS()     \
    int flags; /* vertex flags */    \
    struct CvGraphEdge* first; /* the first incident edge */

typedef struct CvGraphVtx
{
    CV_GRAPH_VERTEX_FIELDS()
}
CvGraphVtx;

#define CV_GRAPH_EDGE_FIELDS()       \
    int flags; /* edge flags */      \
```

```
    float weight; /* edge weight */ \
    struct CvGraphEdge* next[2]; /* the next edges in the incidence lists for staring (0) a
                                    /* and ending (1) vertices */ \
    struct CvGraphVtx* vtx[2]; /* the starting (0) and ending (1) vertices */

typedef struct CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS()
}
CvGraphEdge;

#define  CV_GRAPH_FIELDS()                  \
    CV_SET_FIELDS() /* set of vertices */   \
    CvSet* edges;   /* set of edges */

typedef struct CvGraph
{
    CV_GRAPH_FIELDS()
}
CvGraph;
```

The structure CvGraph is a base for graphs used in OpenCV.

The graph structure inherits from CvSet - which describes common graph properties and the graph vertices, and contains another set as a member - which describes the graph edges.

The vertex, edge, and the graph header structures are declared using the same technique as other extendible OpenCV structures - via macros, which simplify extension and customization of the structures. While the vertex and edge structures do not inherit from CvSetElem explicitly, they satisfy both conditions of the set elements: having an integer field in the beginning and fitting within the CvSetElem structure. The `flags` fields are used as for indicating occupied vertices and edges as well as for other purposes, for example, for graph traversal (see cvCreateGraphScanner et al.), so it is better not to use them directly.

The graph is represented as a set of edges each of which has a list of incident edges. The incidence lists for different vertices are interleaved to avoid information duplication as much as posssible.

The graph may be oriented or unoriented. In the latter case there is no distiction between the edge connecting vertex A with vertex B and the edge connecting vertex B with vertex A - only one of them can exist in the graph at the same moment and it represents both $A \rightarrow B$ and $B \rightarrow A$ edges.

CvGraphScanner (view/add comments)

Graph traversal state.

```
typedef struct CvGraphScanner
{
    CvGraphVtx* vtx;       /* current graph vertex (or current edge origin) */
    CvGraphVtx* dst;       /* current graph edge destination vertex */
    CvGraphEdge* edge;     /* current edge */

    CvGraph* graph;        /* the graph */
    CvSeq*   stack;        /* the graph vertex stack */
    int      index;        /* the lower bound of certainly visited vertices */
    int      mask;         /* event mask */
}
CvGraphScanner;
```

The structure CvGraphScanner is used for depth-first graph traversal. See discussion of the functions below.

CV_TREE_NODE_FIELDS (view/add comments)

Helper macro for a tree node type declaration.

The macro `CV_TREE_NODE_FIELDS()` is used to declare structures that can be organized into hierarchical strucutures (trees), such as CvSeq - the basic type for all dynamic structures. The trees created with nodes declared using this macro can be processed using the functions described below in this section.

CvTreeNodeIterator (view/add comments)

Opens existing or creates new file storage.

```
typedef struct CvTreeNodeIterator
{
    const void* node;
    int level;
    int max_level;
}
CvTreeNodeIterator;
```

```
#define CV_TREE_NODE_FIELDS(node_type)                          \
    int       flags;         /* micsellaneous flags */          \
    int       header_size;   /* size of sequence header */      \
    struct    node_type* h_prev; /* previous sequence */        \
    struct    node_type* h_next; /* next sequence */            \
    struct    node_type* v_prev; /* 2nd previous sequence */    \
    struct    node_type* v_next; /* 2nd next sequence */
```

The structure CvTreeNodeIterator is used to traverse trees. Each tree node should start with the certain fields which are defined by CV_TREE_NODE_FIELDS(...) macro. In C++ terms, each tree node should be a structure "derived" from

```
struct _BaseTreeNode
{
    CV_TREE_NODE_FIELDS(_BaseTreeNode);
}
```

CvSeq, CvSet, CvGraph and other dynamic structures derived from CvSeq comply with the requirement.

---

cvClearGraph (view/add comments)

Clears a graph.

```
void cvClearGraph( CvGraph* graph );
```

graph Graph

The function removes all vertices and edges from a graph. The function has O(1) time complexity.

---

cvClearMemStorage (view/add comments)

Clears memory storage.

```
void cvClearMemStorage( CvMemStorage* storage );
```

storage Memory storage

The function resets the top (free space boundary) of the storage to the very beginning. This function does not deallocate any memory. If the storage has a parent, the function returns all blocks to the parent.

cvClearSeq (view/add comments)

Clears a sequence.

```
void cvClearSeq( CvSeq* seq );
```

seq Sequence

The function removes all elements from a sequence. The function does not return the memory to the storage block, but this memory is reused later when new elements are added to the sequence. The function has 'O(1)' time complexity.

cvClearSet (view/add comments)

Clears a set.

```
void cvClearSet( CvSet* setHeader );
```

setHeader Cleared set

The function removes all elements from set. It has O(1) time complexity.

cvCloneGraph (view/add comments)

Clones a graph.

```
CvGraph* cvCloneGraph(
            const CvGraph* graph,
            CvMemStorage* storage );
```

graph The graph to copy

storage Container for the copy

The function creates a full copy of the specified graph. If the graph vertices or edges have pointers to some external data, it can still be shared between the copies. The vertex and edge indices in the new graph may be different from the original because the function defragments the vertex and edge sets.

---

cvCloneSeq (view/add comments)

Creates a copy of a sequence.

```
CvSeq* cvCloneSeq(
           const CvSeq* seq,
           CvMemStorage* storage=NULL );
```

seq Sequence

storage The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

The function makes a complete copy of the input sequence and returns it.
The call

```
cvCloneSeq( seq, storage )
```

is equivalent to

```
cvSeqSlice( seq, CV_WHOLE_SEQ, storage, 1 )
```

---

cvCreateChildMemStorage (view/add comments)

Creates child memory storage.

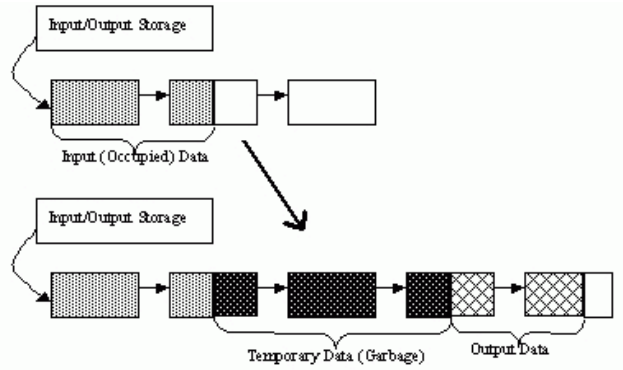
```
CvMemStorage* cvCreateChildMemStorage(CvMemStorage* parent);
```

parent Parent memory storage

The function creates a child memory storage that is similar to simple memory storage except for the differences in the memory allocation/deallocation mechanism. When a child storage needs a new block to add to the block list, it tries to get this block from the parent. The first unoccupied parent block available is taken and excluded from the parent block list. If no blocks are available, the parent either allocates a block or borrows one from its own parent, if any. In other words, the chain, or a more complex structure, of memory storages where every storage is a child/parent of another is possible. When a child storage is released or even cleared, it returns all blocks to the parent. In other aspects, child storage is the same as simple storage.

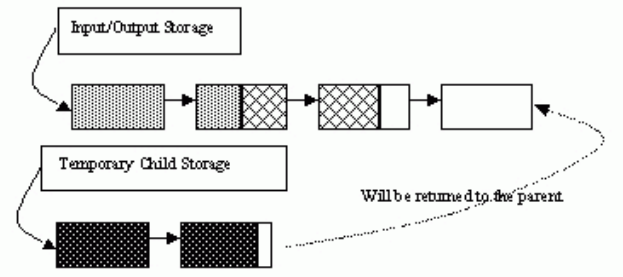
Child storage is useful in the following situation. Imagine that the user needs to process dynamic data residing in a given storage area and put the result back to that same storage area. With the simplest approach, when temporary data is resided in the same storage area as the input and output data, the storage area will look as follows after processing:

Dynamic data processing without using child storage



That is, garbage appears in the middle of the storage. However, if one creates a child memory storage at the beginning of processing, writes temporary data there, and releases the child storage at the end, no garbage will appear in the source/destination storage:

Dynamic data processing using a child storage



cvCreateGraph (view/add comments)

Creates an empty graph.

```
CvGraph* cvCreateGraph(
            int graph_flags,
            int header_size,
            int vtx_size,
            int edge_size,
            CvMemStorage* storage );
```

graph_flags Type of the created graph. Usually, it is either `CV_SEQ_KIND_GRAPH` for generic unoriented graphs and `CV_SEQ_KIND_GRAPH | CV_GRAPH_FLAG_ORIENTED` for generic oriented graphs.

header_size Graph header size; may not be less than `sizeof(CvGraph)`

vtx_size Graph vertex size; the custom vertex structure must start with CvGraphVtx (use `CV_GRAPH_VERTEX_FIELDS()`)

edge_size Graph edge size; the custom edge structure must start with CvGraphEdge (use `CV_GRAPH_EDGE_FIELDS()`)

storage The graph container

The function creates an empty graph and returns a pointer to it.

cvCreateGraphScanner (view/add comments)

Creates structure for depth-first graph traversal.

```
CvGraphScanner* cvCreateGraphScanner(
            CvGraph* graph,
            CvGraphVtx* vtx=NULL,
            int mask=CV_GRAPH_ALL_ITEMS );
```

graph Graph

vtx Initial vertex to start from. If NULL, the traversal starts from the first vertex (a vertex with the minimal index in the sequence of vertices).

mask Event mask indicating which events are of interest to the user (where cvNextGraphItem function returns control to the user) It can be `CV_GRAPH_ALL_ITEMS` (all events are of interest) or a combination of the following flags:

- **CV_GRAPH_VERTEX** stop at the graph vertices visited for the first time
- **CV_GRAPH_TREE_EDGE** stop at tree edges (`tree edge` is the edge connecting the last visited vertex and the vertex to be visited next)
- **CV_GRAPH_BACK_EDGE** stop at back edges (`back edge` is an edge connecting the last visited vertex with some of its ancestors in the search tree)
- **CV_GRAPH_FORWARD_EDGE** stop at forward edges (`forward edge` is an edge conecting the last visited vertex with some of its descendants in the search tree. The forward edges are only possible during oriented graph traversal)
- **CV_GRAPH_CROSS_EDGE** stop at cross edges (`cross edge` is an edge connecting different search trees or branches of the same tree. The `cross edges` are only possible during oriented graph traversal)
- **CV_GRAPH_ANY_EDGE** stop at any edge (`tree, back, forward`, and `cross edges`)
- **CV_GRAPH_NEW_TREE** stop in the beginning of every new search tree. When the traversal procedure visits all vertices and edges reachable from the initial vertex (the visited vertices together with tree edges make up a tree), it searches for some unvisited vertex in the graph and resumes the traversal process from that vertex. Before starting a new tree (including the very first tree when `cvNextGraphItem` is called for the first time) it generates a `CV_GRAPH_NEW_TREE` event. For unoriented graphs, each search tree corresponds to a connected component of the graph.
- **CV_GRAPH_BACKTRACKING** stop at every already visited vertex during backtracking - returning to already visited vertexes of the traversal tree.

The function creates a structure for depth-first graph traversal/search. The initialized structure is used in the cvNextGraphItem function - the incremental traversal procedure.

---

cvCreateMemStorage (view/add comments)

Creates memory storage.

```
CvMemStorage* cvCreateMemStorage( int blockSize=0 );
```

blockSize Size of the storage blocks in bytes. If it is 0, the block size is set to a default value - currently it is about 64K.

The function creates an empty memory storage. See CvMemStorage description.

---

cvCreateSeq (view/add comments)

Creates a sequence.

```
CvSeq* cvCreateSeq(
           int seqFlags,
           int headerSize,
           int elemSize,
           CvMemStorage* storage);
```

seqFlags Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be set to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.

headerSize Size of the sequence header; must be greater than or equal to `sizeof(CvSeq)`. If a specific type or its extension is indicated, this type must fit the base type header.

elemSize Size of the sequence elements in bytes. The size must be consistent with the sequence type. For example, for a sequence of points to be created, the element type `CV_SEQ_ELTYPE_POINT` should be specified and the parameter `elemSize` must be equal to `sizeof(CvPoint)`.

storage Sequence location

The function creates a sequence and returns the pointer to it. The function allocates the sequence header in the storage block as one continuous chunk and sets the structure fields `flags`, `elemSize`, `headerSize`, and `storage` to passed values, sets `delta_elems` to the default value (that may be reassigned using the cvSetSeqBlockSize function), and clears other header fields, including the space following the first `sizeof(CvSeq)` bytes.

cvCreateSet (view/add comments)

Creates an empty set.

```
CvSet* cvCreateSet(
            int set_flags,
            int header_size,
            int elem_size,
            CvMemStorage* storage );
```

set_flags Type of the created set

header_size Set header size; may not be less than `sizeof(CvSet)`

elem_size Set element size; may not be less than CvSetElem

storage Container for the set

The function creates an empty set with a specified header size and element size, and returns the pointer to the set. This function is just a thin layer on top of cvCreateSeq.

cvCvtSeqToArray (view/add comments)

Copies a sequence to one continuous block of memory.

```
void* cvCvtSeqToArray(
            const CvSeq* seq,
            void* elements,
            CvSlice slice=CV_WHOLE_SEQ );
```

seq Sequence

elements Pointer to the destination array that must be large enough. It should be a pointer to data, not a matrix header.

slice The sequence portion to copy to the array

The function copies the entire sequence or subsequence to the specified buffer and returns the pointer to the buffer.

cvEndWriteSeq (view/add comments)

Finishes the process of writing a sequence.

```
CvSeq* cvEndWriteSeq( CvSeqWriter* writer );
```

writer Writer state

The function finishes the writing process and returns the pointer to the written sequence. The function also truncates the last incomplete sequence block to return the remaining part of the block to memory storage. After that, the sequence can be read and modified safely. See cvcvStartWriteSeq and cvcvStartAppendToSeq

cvFindGraphEdge (view/add comments)

Finds an edge in a graph.

```
CvGraphEdge* cvFindGraphEdge( const CvGraph* graph, int start_idx, int
end_idx );
```

```
#define cvGraphFindEdge cvFindGraphEdge
```

graph Graph

start_idx Index of the starting vertex of the edge

end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function finds the graph edge connecting two specified vertices and returns a pointer to it or NULL if the edge does not exist.

cvFindGraphEdgeByPtr (view/add comments)

Finds an edge in a graph by using its pointer.

```
CvGraphEdge* cvFindGraphEdgeByPtr(
            const CvGraph* graph,
            const CvGraphVtx* startVtx,
            const CvGraphVtx* endVtx );
```

```
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

graph Graph

startVtx Pointer to the starting vertex of the edge

endVtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

cvFlushSeqWriter (view/add comments)

Updates sequence headers from the writer.

```
void cvFlushSeqWriter( CvSeqWriter* writer );
```

writer Writer state

The function is intended to enable the user to read sequence elements, whenever required, during the writing process, e.g., in order to check specific conditions. The function updates the sequence headers to make reading from the sequence possible. The writer is not closed, however, so that the writing process can be continued at any time. If an algorithm requires frequent flushes, consider using cvSeqPush instead.

cvGetGraphVtx (view/add comments)

Finds a graph vertex by using its index.

```
CvGraphVtx* cvGetGraphVtx(
            CvGraph* graph,
            int vtx_idx );
```

graph Graph

vtx_idx Index of the vertex

The function finds the graph vertex by using its index and returns the pointer to it or NULL if the vertex does not belong to the graph.

cvGetSeqElem (view/add comments)

Returns a pointer to a sequence element according to its index.

```
char* cvGetSeqElem( const CvSeq* seq, int index );
```

```
#define CV_GET_SEQ_ELEM( TYPE, seq, index )  (TYPE*)cvGetSeqElem( (CvSeq*)(seq), (index) )
```

seq Sequence

index Index of element

The function finds the element with the given index in the sequence and returns the pointer to it. If the element is not found, the function returns 0. The function supports negative indices, where -1 stands for the last sequence element, -2 stands for the one before last, etc. If the sequence is most likely to consist of a single sequence block or the desired element is likely to be located in the first block, then the macro `CV_GET_SEQ_ELEM( elemType, seq, index )` should be used, where the parameter `elemType` is the type of sequence elements (CvPoint for example), the parameter `seq` is a sequence, and the parameter `index` is the index of the desired element. The macro checks first whether the desired element belongs to the first block of the sequence and returns it if it does; otherwise the macro calls the main function `GetSeqElem`. Negative indices always cause the cvGetSeqElem call. The function has O(1) time complexity assuming that the number of blocks is much smaller than the number of elements.

cvGetSeqReaderPos (view/add comments)

Returns the current reader position.

```
int cvGetSeqReaderPos( CvSeqReader* reader );
```

reader Reader state

The function returns the current reader position (within 0 ... `reader->seq->total` - 1).

cvGetSetElem (view/add comments)

Finds a set element by its index.

```
CvSetElem* cvGetSetElem(
           const CvSet* setHeader,
           int index );
```

setHeader Set

index Index of the set element within a sequence

The function finds a set element by its index. The function returns the pointer to it or 0 if the index is invalid or the corresponding node is free. The function supports negative indices as it uses cvGetSeqElem to locate the node.

cvGraphAddEdge (view/add comments)

Adds an edge to a graph.

```
int cvGraphAddEdge(
           CvGraph* graph,
           int start_idx,
```

```
            int end_idx,
            const CvGraphEdge* edge=NULL,
            CvGraphEdge** inserted_edge=NULL );
```

graph Graph

start_idx Index of the starting vertex of the edge

end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

edge Optional input parameter, initialization data for the edge

inserted_edge Optional output parameter to contain the address of the inserted edge

The function connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same, or there is some other critical situation. In the latter case (i.e., when the result is negative), the function also reports an error by default.

---

cvGraphAddEdgeByPtr (view/add comments)

Adds an edge to a graph by using its pointer.

```
int cvGraphAddEdgeByPtr(
            CvGraph* graph,
            CvGraphVtx* start_vtx,
            CvGraphVtx* end_vtx,
            const CvGraphEdge* edge=NULL,
            CvGraphEdge** inserted_edge=NULL );
```

graph Graph

start_vtx Pointer to the starting vertex of the edge

end_vtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

edge Optional input parameter, initialization data for the edge

inserted_edge Optional output parameter to contain the address of the inserted edge within the edge set

The function connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already, and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e., when the result is negative), the function also reports an error by default.

---

cvGraphAddVtx (view/add comments)

Adds a vertex to a graph.

```
int cvGraphAddVtx(
            CvGraph* graph,
            const CvGraphVtx* vtx=NULL,
            CvGraphVtx** inserted_vtx=NULL );
```

graph Graph

vtx Optional input argument used to initialize the added vertex (only user-defined fields beyond `sizeof(CvGraphVtx)` are copied)

inserted_vertex Optional output argument. If not `NULL`, the address of the new vertex is written here.

The function adds a vertex to the graph and returns the vertex index.

---

cvGraphEdgeIdx (view/add comments)

Returns the index of a graph edge.

```
int cvGraphEdgeIdx(
            CvGraph* graph,
            CvGraphEdge* edge );
```

graph Graph

edge Pointer to the graph edge

The function returns the index of a graph edge.

---

cvGraphRemoveEdge (view/add comments)

Removes an edge from a graph.

```
void cvGraphRemoveEdge(
            CvGraph* graph,
            int start_idx,
            int end_idx );
```

graph Graph

start_idx Index of the starting vertex of the edge

end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

---

cvGraphRemoveEdgeByPtr (view/add comments)

Removes an edge from a graph by using its pointer.

```
void cvGraphRemoveEdgeByPtr(
            CvGraph* graph,
            CvGraphVtx* start_vtx,
            CvGraphVtx* end_vtx );
```

graph Graph

start_vtx Pointer to the starting vertex of the edge

end_vtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

---

cvGraphRemoveVtx (view/add comments)

Removes a vertex from a graph.

```
int cvGraphRemoveVtx(
            CvGraph* graph,
            int index );
```

graph Graph

vtx_idx Index of the removed vertex

The function removes a vertex from a graph together with all the edges incident to it. The function reports an error if the input vertex does not belong to the graph. The return value is the number of edges deleted, or -1 if the vertex does not belong to the graph.

---

cvGraphRemoveVtxByPtr (view/add comments)

Removes a vertex from a graph by using its pointer.

```
int cvGraphRemoveVtxByPtr(
            CvGraph* graph,
            CvGraphVtx* vtx );
```

graph Graph

vtx Pointer to the removed vertex

The function removes a vertex from the graph by using its pointer together with all the edges incident to it. The function reports an error if the vertex does not belong to the graph. The return value is the number of edges deleted, or -1 if the vertex does not belong to the graph.

---

cvGraphVtxDegree (view/add comments)

Counts the number of edges indicent to the vertex.

```
int cvGraphVtxDegree( const CvGraph* graph, int vtxIdx );
```

graph Graph

vtxIdx Index of the graph vertex

The function returns the number of edges incident to the specified vertex, both incoming and outgoing. To count the edges, the following code is used:

```
CvGraphEdge* edge = vertex->first; int count = 0;
while( edge )
{
    edge = CV_NEXT_GRAPH_EDGE( edge, vertex );
    count++;
}
```

The macro `CV_NEXT_GRAPH_EDGE( edge, vertex )` returns the edge incident to `vertex` that follows after `edge`.

---

cvGraphVtxDegreeByPtr (view/add comments)

Finds an edge in a graph.

```
int cvGraphVtxDegreeByPtr(
           const CvGraph* graph,
           const CvGraphVtx* vtx );
```

graph Graph

vtx Pointer to the graph vertex

The function returns the number of edges incident to the specified vertex, both incoming and outcoming.

---

cvGraphVtxIdx (view/add comments)

Returns the index of a graph vertex.

```
int cvGraphVtxIdx(
            CvGraph* graph,
            CvGraphVtx* vtx );
```

graph Graph

vtx Pointer to the graph vertex

The function returns the index of a graph vertex.

---

cvInitTreeNodeIterator (view/add comments)

Initializes the tree node iterator.

```
void cvInitTreeNodeIterator(
            CvTreeNodeIterator* tree_iterator,
            const void* first,
            int max_level );
```

tree_iterator Tree iterator initialized by the function

first The initial node to start traversing from

max_level The maximal level of the tree (`first` node assumed to be at the first level) to traverse up to. For example, 1 means that only nodes at the same level as `first` should be visited, 2 means that the nodes on the same level as `first` and their direct children should be visited, and so forth.

The function initializes the tree iterator. The tree is traversed in depth-first order.

cvInsertNodeIntoTree (view/add comments)

Adds a new node to a tree.

```
void cvInsertNodeIntoTree(
           void* node,
           void* parent,
           void* frame );
```

node The inserted node

parent The parent node that is already in the tree

frame The top level node. If `parent` and `frame` are the same, the `v_prev` field of `node` is set to NULL rather than `parent`.

The function adds another node into tree. The function does not allocate any memory, it can only modify links of the tree nodes.

cvMakeSeqHeaderForArray (view/add comments)

Constructs a sequence header for an array.

```
CvSeq* cvMakeSeqHeaderForArray(
           int seq_type,
           int header_size,
           int elem_size,
           void* elements,
           int total,
           CvSeq* seq,
           CvSeqBlock* block );
```

seq_type Type of the created sequence

header_size Size of the header of the sequence. Parameter sequence must point to the structure of that size or greater

elem_size Size of the sequence elements

elements Elements that will form a sequence

total Total number of elements in the sequence. The number of array elements must be equal to the value of this parameter.

seq Pointer to the local variable that is used as the sequence header

block Pointer to the local variable that is the header of the single sequence block

The function initializes a sequence header for an array. The sequence header as well as the sequence block are allocated by the user (for example, on stack). No data is copied by the function. The resultant sequence will consists of a single block and have NULL storage pointer; thus, it is possible to read its elements, but the attempts to add elements to the sequence will raise an error in most cases.

---

cvMemStorageAlloc (view/add comments)

Allocates a memory buffer in a storage block.

```
void* cvMemStorageAlloc(
          CvMemStorage* storage,
          size_t size );
```

storage Memory storage

size Buffer size

The function allocates a memory buffer in a storage block. The buffer size must not exceed the storage block size, otherwise a runtime error is raised. The buffer address is aligned by `CV_STRUCT_ALIGN=sizeof(double)` (for the moment) bytes.

---

cvMemStorageAllocString (view/add comments)

Allocates a text string in a storage block.

```
CvString cvMemStorageAllocString(CvMemStorage* storage, const char*
ptr, int len=-1);
```

```
typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;
```

storage Memory storage

ptr The string

len Length of the string (not counting the ending NUL) . If the parameter is negative, the function computes the length.

The function creates copy of the string in memory storage. It returns the structure that contains user-passed or computed length of the string and pointer to the copied string.

---

cvNextGraphItem (view/add comments)

Executes one or more steps of the graph traversal procedure.

```
int cvNextGraphItem( CvGraphScanner* scanner );
```

scanner Graph traversal state. It is updated by this function.

The function traverses through the graph until an event of interest to the user (that is, an event, specified in the mask in the cvCreateGraphScanner call) is met or the traversal is completed. In the first case, it returns one of the events listed in the description of the mask parameter above and with the next call it resumes the traversal. In the latter case, it returns CV_GRAPH_OVER (-1). When the event is CV_GRAPH_VERTEX, CV_GRAPH_BACKTRACKING, or CV_GRAPH_NEW_TREE, the currently observed vertex is stored in scanner->vtx. And if the event is edge-related, the edge itself is stored at scanner->edge, the previously visited vertex - at scanner->vtx and the other ending vertex of the edge - at scanner->dst.

cvNextTreeNode (view/add comments)

Returns the currently observed node and moves the iterator toward the next node.

```
void* cvNextTreeNode( CvTreeNodeIterator* tree_iterator );
```

tree_iterator Tree iterator initialized by the function

The function returns the currently observed node and then updates the iterator - moving it toward the next node. In other words, the function behavior is similar to the *p++ expression on a typical C pointer or C++ collection iterator. The function returns NULL if there are no more nodes.

cvPrevTreeNode (view/add comments)

Returns the currently observed node and moves the iterator toward the previous node.

```
void* cvPrevTreeNode( CvTreeNodeIterator* tree_iterator );
```

tree_iterator Tree iterator initialized by the function

The function returns the currently observed node and then updates the iterator - moving it toward the previous node. In other words, the function behavior is similar to the *p-- expression on a typical C pointer or C++ collection iterator. The function returns NULL if there are no more nodes.

cvReleaseGraphScanner (view/add comments)

Completes the graph traversal procedure.

```
void cvReleaseGraphScanner( CvGraphScanner** scanner );
```

scanner Double pointer to graph traverser

The function completes the graph traversal procedure and releases the traverser state.

cvReleaseMemStorage (view/add comments)

Releases memory storage.

```
void cvReleaseMemStorage( CvMemStorage** storage );
```

storage Pointer to the released storage

The function deallocates all storage memory blocks or returns them to the parent, if any. Then it deallocates the storage header and clears the pointer to the storage. All child storage associated with a given parent storage block must be released before the parent storage block is released.

cvRestoreMemStoragePos (view/add comments)

Restores memory storage position.

```
void cvRestoreMemStoragePos(
           CvMemStorage* storage,
           CvMemStoragePos* pos);
```

storage Memory storage

pos New storage top position

The function restores the position of the storage top from the parameter `pos`. This function and the function `cvClearMemStorage` are the only methods to release memory occupied in memory blocks. Note again that there is no way to free memory in the middle of an occupied portion of a storage block.

cvSaveMemStoragePos (view/add comments)

Saves memory storage position.

```
void cvSaveMemStoragePos(
           const CvMemStorage* storage,
           CvMemStoragePos* pos);
```

storage Memory storage

pos The output position of the storage top

The function saves the current position of the storage top to the parameter `pos`. The function `cvRestoreMemStoragePos` can further retrieve this position.

---

cvSeqElemIdx (view/add comments)

Returns the index of a specific sequence element.

```
int cvSeqElemIdx(
           const CvSeq* seq,
           const void* element,
           CvSeqBlock** block=NULL );
```

seq Sequence

element Pointer to the element within the sequence

block Optional argument. If the pointer is not `NULL`, the address of the sequence block that contains the element is stored in this location.

The function returns the index of a sequence element or a negative number if the element is not found.

---

cvSeqInsert (view/add comments)

Inserts an element in the middle of a sequence.

```
char* cvSeqInsert(
           CvSeq* seq,
           int beforeIndex,
           void* element=NULL );
```

seq Sequence

beforeIndex Index before which the element is inserted. Inserting before 0 (the minimal allowed value of the parameter) is equal to cvSeqPushFront and inserting before `seq->total` (the maximal allowed value of the parameter) is equal to cvSeqPush.

element Inserted element

The function shifts the sequence elements from the inserted position to the nearest end of the sequence and copies the `element` content there if the pointer is not NULL. The function returns a pointer to the inserted element.

---

cvSeqInsertSlice (view/add comments)

Inserts an array in the middle of a sequence.

```
void cvSeqInsertSlice(
           CvSeq* seq,
           int beforeIndex,
           const CvArr* fromArr );
```

seq Sequence

slice The part of the sequence to remove

fromArr The array to take elements from

The function inserts all `fromArr` array elements at the specified position of the sequence. The array `fromArr` can be a matrix or another sequence.

cvSeqInvert (view/add comments)

Reverses the order of sequence elements.

```
void cvSeqInvert( CvSeq* seq );
```

seq Sequence

The function reverses the sequence in-place - makes the first element go last, the last element go first and so forth.

cvSeqPop (view/add comments)

Removes an element from the end of a sequence.

```
void cvSeqPop(
           CvSeq* seq,
           void* element=NULL );
```

seq Sequence

element Optional parameter . If the pointer is not zero, the function copies the removed element to this location.

The function removes an element from a sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

cvSeqPopFront (view/add comments)

Removes an element from the beginning of a sequence.

```
void cvSeqPopFront(

            CvSeq* seq,

            void* element=NULL );
```

seq Sequence

element Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

The function removes an element from the beginning of a sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

---

cvSeqPopMulti (view/add comments)

Removes several elements from either end of a sequence.

```
void cvSeqPopMulti(
            CvSeq* seq,
            void* elements,
            int count,
            int in_front=0 );
```

seq Sequence

elements Removed elements

count Number of elements to pop

in_front The flags specifying which end of the modified sequence.

- **CV_BACK** the elements are added to the end of the sequence
- **CV_FRONT** the elements are added to the beginning of the sequence

The function removes several elements from either end of the sequence. If the number of the elements to be removed exceeds the total number of elements in the sequence, the function removes as many elements as possible.

cvSeqPush (view/add comments)

Adds an element to the end of a sequence.

```
char* cvSeqPush(
            CvSeq* seq,
            void* element=NULL );
```

seq Sequence

element Added element

The function adds an element to the end of a sequence and returns a pointer to the allocated element. If the input `element` is NULL, the function simply allocates a space for one more element.

The following code demonstrates how to create a new sequence using this function:

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, /* sequence of integer elements */
                          sizeof(CvSeq), /* header size - no extra fields */
                          sizeof(int), /* element size */
                          storage /* the container storage */ );
int i;
for( i = 0; i < 100; i++ )
{
    int* added = (int*)cvSeqPush( seq, &i );
    printf( "%d is added\n", *added );
}

...
/* release memory storage in the end */
cvReleaseMemStorage( &storage );
```

The function has O(1) complexity, but there is a faster method for writing large sequences (see cvStartWriteSeq and related functions).

cvSeqPushFront (view/add comments)

Adds an element to the beginning of a sequence.

```
char* cvSeqPushFront( CvSeq* seq, void* element=NULL );
```

seq Sequence

element Added element

The function is similar to cvSeqPush but it adds the new element to the beginning of the sequence. The function has O(1) complexity.

---

cvSeqPushMulti (view/add comments)

Pushes several elements to either end of a sequence.

```
void cvSeqPushMulti(
           CvSeq* seq,
           void* elements,
           int count,
           int in_front=0 );
```

seq Sequence

elements Added elements

count Number of elements to push

in_front The flags specifying which end of the modified sequence.

- **CV_BACK** the elements are added to the end of the sequence
- **CV_FRONT** the elements are added to the beginning of the sequence

The function adds several elements to either end of a sequence. The elements are added to the sequence in the same order as they are arranged in the input array but they can fall into different sequence blocks.

cvSeqRemove (view/add comments)

Removes an element from the middle of a sequence.

```
void cvSeqRemove(
            CvSeq* seq,
            int index );
```

seq Sequence

index Index of removed element

The function removes elements with the given index. If the index is out of range the function reports an error. An attempt to remove an element from an empty sequence is a special case of this situation. The function removes an element by shifting the sequence elements between the nearest end of the sequence and the `index`-th position, not counting the latter.

cvSeqRemoveSlice (view/add comments)

Removes a sequence slice.

```
void cvSeqRemoveSlice( CvSeq* seq, CvSlice slice );
```

seq Sequence

slice The part of the sequence to remove

The function removes a slice from the sequence.

cvSeqSearch (view/add comments)

Searches for an element in a sequence.

```
char* cvSeqSearch( CvSeq* seq, const void* elem, CvCmpFunc func, int
is_sorted, int* elem_idx, void* userdata=NULL );
```

seq The sequence

elem The element to look for

func The comparison function that returns negative, zero or positive value depending on the relationships among the elements (see also cvSeqSort)

is_sorted Whether the sequence is sorted or not

elem_idx Output parameter; index of the found element

userdata The user parameter passed to the compasion function; helps to avoid global variables in some cases

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
```

The function searches for the element in the sequence. If the sequence is sorted, a binary O(log(N)) search is used; otherwise, a simple linear search is used. If the element is not found, the function returns a NULL pointer and the index is set to the number of sequence elements if a linear search is used, or to the smallest index `i, seq(i)>elem`.

---

cvSeqSlice (view/add comments)

Makes a separate header for a sequence slice.

```
CvSeq* cvSeqSlice(
            const CvSeq* seq,
            CvSlice slice,
            CvMemStorage* storage=NULL,
            int copy_data=0 );
```

seq Sequence

slice The part of the sequence to be extracted

storage The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

copy_data The flag that indicates whether to copy the elements of the extracted slice (`copy_data!=0`) or not (`copy_data=0`)

The function creates a sequence that represents the specified slice of the input sequence. The new sequence either shares the elements with the original sequence or has its own copy of the elements. So if one needs to process a part of sequence but the processing function does not have a slice parameter, the required sub-sequence may be extracted using this function.

---

cvSeqSort (view/add comments)

Sorts sequence element using the specified comparison function.

```
void cvSeqSort( CvSeq* seq, CvCmpFunc func, void* userdata=NULL );
```

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
```

seq The sequence to sort

func The comparison function that returns a negative, zero, or positive value depending on the relationships among the elements (see the above declaration and the example below) - a similar function is used by `qsort` from C runline except that in the latter, `userdata` is not used

userdata The user parameter passed to the compasion function; helps to avoid global variables in some cases

The function sorts the sequence in-place using the specified criteria. Below is an example of using this function:

```
/* Sort 2d points in top-to-bottom left-to-right order */
static int cmp_func( const void* _a, const void* _b, void* userdata )
{
    CvPoint* a = (CvPoint*)_a;
```

```
    CvPoint* b = (CvPoint*)_b;
    int y_diff = a->y - b->y;
    int x_diff = a->x - b->x;
    return y_diff ? y_diff : x_diff;
}

...

CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq), sizeof(CvPoint), storage );
int i;

for( i = 0; i < 10; i++ )
{
    CvPoint pt;
    pt.x = rand() % 1000;
    pt.y = rand() % 1000;
    cvSeqPush( seq, &pt );
}

cvSeqSort( seq, cmp_func, 0 /* userdata is not used here */ );

/* print out the sorted sequence */
for( i = 0; i < seq->total; i++ )
{
    CvPoint* pt = (CvPoint*)cvSeqElem( seq, i );
    printf( "(%d,%d)\n", pt->x, pt->y );
}

cvReleaseMemStorage( &storage );
```

---

cvSetAdd (view/add comments)

Occupies a node in the set.

```
int cvSetAdd(
            CvSet* setHeader,
            CvSetElem* elem=NULL,
            CvSetElem** inserted_elem=NULL );
```

setHeader Set

elem Optional input argument, an inserted element. If not NULL, the function copies the data to the allocated node (the MSB of the first integer field is cleared after copying).

inserted_elem Optional output argument; the pointer to the allocated cell

The function allocates a new node, optionally copies input element data to it, and returns the pointer and the index to the node. The index value is taken from the lower bits of the `flags` field of the node. The function has O(1) complexity; however, there exists a faster function for allocating set nodes (see cvSetNew).

---

cvSetNew (view/add comments)

Adds an element to a set (fast variant).

```
CvSetElem* cvSetNew( CvSet* setHeader );
```

setHeader Set

The function is an inline lightweight variant of cvSetAdd. It occupies a new node and returns a pointer to it rather than an index.

---

cvSetRemove (view/add comments)

Removes an element from a set.

```
void cvSetRemove(
            CvSet* setHeader,
            int index );
```

setHeader Set

index Index of the removed element

The function removes an element with a specified index from the set. If the node at the specified location is not occupied, the function does nothing. The function has O(1) complexity; however, cvSetRemoveByPtr provides a quicker way to remove a set element if it is located already.

cvSetRemoveByPtr (view/add comments)

Removes a set element based on its pointer.

```
void cvSetRemoveByPtr(
           CvSet* setHeader,
           void* elem );
```

setHeader Set

elem Removed element

The function is an inline lightweight variant of cvSetRemove that requires an element pointer. The function does not check whether the node is occupied or not - the user should take care of that.

cvSetSeqBlockSize (view/add comments)

Sets up sequence block size.

```
void cvSetSeqBlockSize(
           CvSeq* seq,
           int deltaElems );
```

seq Sequence

deltaElems Desirable sequence block size for elements

The function affects memory allocation granularity. When the free space in the sequence buffers has run out, the function allocates the space for `deltaElems` sequence elements. If this block immediately follows the one previously allocated, the two blocks are concatenated; otherwise, a new sequence block is created. Therefore, the bigger the parameter is, the lower the possible sequence fragmentation, but the more space in the storage block is wasted. When the sequence is created, the parameter `deltaElems` is set to the default value of about 1K. The function can be called any time after the sequence is created and affects future allocations. The function can modify the passed value of the parameter to meet memory storage constraints.

cvSetSeqReaderPos (view/add comments)

Moves the reader to the specified position.

```
void cvSetSeqReaderPos(
            CvSeqReader* reader,
            int index,
            int is_relative=0 );
```

reader Reader state

index The destination position. If the positioning mode is used (see the next parameter), the actual position will be `index` mod `reader->seq->total`.

is_relative If it is not zero, then `index` is a relative to the current position

The function moves the read position to an absolute position or relative to the current position.

cvStartAppendToSeq (view/add comments)

Initializes the process of writing data to a sequence.

```
void cvStartAppendToSeq(
            CvSeq* seq,
            CvSeqWriter* writer );
```

seq Pointer to the sequence

writer Writer state; initialized by the function

The function initializes the process of writing data to a sequence. Written elements are added to the end of the sequence by using the `CV_WRITE_SEQ_ELEM( written_elem, writer )` macro. Note that during the writing process, other operations on the sequence may yield an incorrect result or even corrupt the sequence (see description of cvFlushSeqWriter, which helps to avoid some of these problems).

cvStartReadSeq (view/add comments)

Initializes the process of sequential reading from a sequence.

```
void cvStartReadSeq(
           const CvSeq* seq,
           CvSeqReader* reader,
           int reverse=0 );
```

seq Sequence

reader Reader state; initialized by the function

reverse Determines the direction of the sequence traversal. If `reverse` is 0, the reader is positioned at the first sequence element; otherwise it is positioned at the last element.

The function initializes the reader state. After that, all the sequence elements from the first one down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM( read_elem, reader )` in the case of forward reading and by using `CV_REV_READ_SEQ_ELEM( read_elem, reader )` in the case of reverse reading. Both macros put the sequence element to `read_elem` and move the reading pointer toward the next element. A circular structure of sequence blocks is used for the reading process, that is, after the last element has been read by the macro `CV_READ_SEQ_ELEM`, the first element is read when the macro is called again. The same applies to `CV_REV_READ_SEQ_ELEM`. There is no function to finish the reading process, since it neither changes the sequence nor creates any temporary buffers. The reader field `ptr` points to the current element of the sequence that is to be read next. The code below demonstrates how to use the sequence writer and reader.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, sizeof(CvSeq), sizeof(int), storage );
CvSeqWriter writer;
CvSeqReader reader;
int i;

cvStartAppendToSeq( seq, &writer );
for( i = 0; i < 10; i++ )
{
    int val = rand()%100;
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("%d is written\n", val );
```

```
}
cvEndWriteSeq( &writer );

cvStartReadSeq( seq, &reader, 0 );
for( i = 0; i < seq->total; i++ )
{
    int val;
#if 1
    CV_READ_SEQ_ELEM( val, reader );
    printf("%d is read\n", val );
#else /* alternative way, that is prefferable if sequence elements are large,
         or their size/type is unknown at compile time */
    printf("%d is read\n", *(int*)reader.ptr );
    CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
#endif
}
...

cvReleaseStorage( &storage );
```

---

cvStartWriteSeq (view/add comments)

Creates a new sequence and initializes a writer for it.

```
void cvStartWriteSeq(
           int seq_flags,
           int header_size,
           int elem_size,
           CvMemStorage* storage,
           CvSeqWriter* writer );
```

seq_flags Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be equal to 0; otherwise the appropriate type must be selected from the list of predefined sequence types.

header_size Size of the sequence header. The parameter value may not be less than `sizeof(CvSeq)`. If a certain type or extension is specified, it must fit within the base type header.

elem_size Size of the sequence elements in bytes; must be consistent with the sequence type. For example, if a sequence of points is created (element type `CV_SEQ_ELTYPE_POINT`), then the parameter `elem_size` must be equal to `sizeof(CvPoint)`.

storage Sequence location

writer Writer state; initialized by the function

The function is a combination of cvCreateSeq and cvStartAppendToSeq. The pointer to the created sequence is stored at `writer->seq` and is also returned by the cvEndWriteSeq function that should be called at the end.

---

cvTreeToNodeSeq (view/add comments)

Gathers all node pointers to a single sequence.

```
CvSeq* cvTreeToNodeSeq(
           const void* first,
           int header_size,
           CvMemStorage* storage );
```

first The initial tree node

header_size Header size of the created sequence (sizeof(CvSeq) is the most frequently used value)

storage Container for the sequence

The function puts pointers of all nodes reacheable from `first` into a single sequence. The pointers are written sequentially in the depth-first order.

1.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with `CV_RGB` macro or the cv function) for color images and brightness for grayscale images. For color images the

order channel is normally *Blue, Green, Red*, this is what cv, cv and cv expect , so if you form a color using cv, it should look like:

$$\texttt{cvScalar}(blue_component, green_component, red_component[, alpha_component])$$

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using cv.

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}(x*2^{-shift}, y*2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

---

cvCircle (view/add comments)

Draws a circle.

```
void cvCircle(
           CvArr* img,
           CvPoint center,
           int radius,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img Image where the circle is drawn

center Center of the circle

radius Radius of the circle

color Circle color

thickness Thickness of the circle outline if positive, otherwise this indicates that a filled circle is to be drawn

lineType Type of the circle boundary, see Line description

shift Number of fractional bits in the center coordinates and radius value

The function draws a simple or filled circle with a given center and radius.

---

cvClipLine (view/add comments)

Clips the line against the image rectangle.

```
int cvClipLine(
           CvSize imgSize,
           CvPoint* pt1,
           CvPoint* pt2 );
```

imgSize Size of the image

pt1 First ending point of the line segment. It is modified by the function.

pt2 Second ending point of the line segment. It is modified by the function.

The function calculates a part of the line segment which is entirely within the image. It returns 0 if the line segment is completely outside the image and 1 otherwise.

---

cvDrawContours (view/add comments)

Draws contour outlines or interiors in an image.

```
void cvDrawContours(
           CvArr *img,
           CvSeq* contour,
           CvScalar external_color,
```

```
            CvScalar hole_color,
            int max_level,
            int thickness=1,
            int lineType=8 );
```

img Image where the contours are to be drawn. As with any other drawing function, the contours are clipped with the ROI.

contour Pointer to the first contour

external_color Color of the external contours

hole_color Color of internal contours (holes)

max_level Maximal level for drawn contours. If 0, only `contour` is drawn. If 1, the contour and all contours following it on the same level are drawn. If 2, all contours following and all contours one level below the contours are drawn, and so forth. If the value is negative, the function does not draw the contours following after `contour` but draws the child contours of `contour` up to the $|\texttt{max_level}| - 1$ level.

thickness Thickness of lines the contours are drawn with. If it is negative (For example, =CV_FILLED), the contour interiors are drawn.

lineType Type of the contour segments, see Line description

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$.

Example: Connected component detection via contour functions

```
#include "cv.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    IplImage* src;
    // the first command line parameter must be file name of binary
    // (black-n-white) image
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* contour = 0;
```

```
        cvThreshold( src, src, 1, 255, CV_THRESH_BINARY );
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvFindContours( src, storage, &contour, sizeof(CvContour),
           CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );
        cvZero( dst );

        for( ; contour != 0; contour = contour->h_next )
        {
            CvScalar color = CV_RGB( rand()&255, rand()&255, rand()&255 );
            /* replace CV_FILLED with 1 to see the outlines */
            cvDrawContours( dst, contour, color, color, -1, CV_FILLED, 8 );
        }

        cvNamedWindow( "Components", 1 );
        cvShowImage( "Components", dst );
        cvWaitKey(0);
    }
}
```

---

cvEllipse (view/add comments)

Draws a simple or thick elliptic arc or an fills ellipse sector.

```
void cvEllipse(
           CvArr* img,
           CvPoint center,
           CvSize axes,
           double angle,
           double start_angle,
           double end_angle,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img The image

center Center of the ellipse

axes Length of the ellipse axes

angle Rotation angle

start_angle Starting angle of the elliptic arc

end_angle Ending angle of the elliptic arc.

color Ellipse color

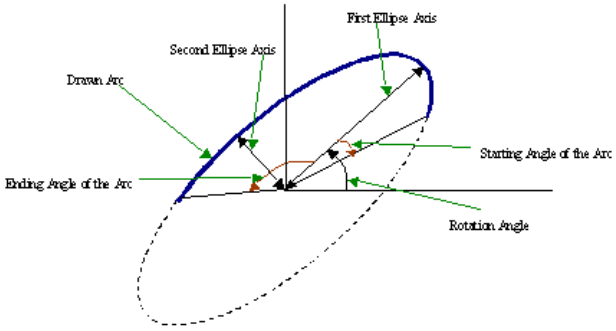
thickness Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the center coordinates and axes' values

The function draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by the ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are given in degrees. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



---

cvEllipseBox (view/add comments)

Draws a simple or thick elliptic arc or fills an ellipse sector.

```
void cvEllipseBox(
           CvArr* img,
           CvBox2D box,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img Image

box The enclosing box of the ellipse drawn

thickness Thickness of the ellipse boundary

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the box vertex coordinates

The function draws a simple or thick ellipse outline, or fills an ellipse. The functions provides a convenient way to draw an ellipse approximating some shape; that is what CamShift and FitEllipse do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

---

cvFillConvexPoly (view/add comments)

Fills a convex polygon.

```
void cvFillConvexPoly(
           CvArr* img,
           CvPoint* pts,
           int npts,
           CvScalar color,
           int lineType=8,
           int shift=0 );
```

img Image

pts Array of pointers to a single polygon

npts Polygon vertex counter

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills a convex polygon's interior. This function is much faster than the function `cvFillPoly` and can fill not only convex polygons but any monotonic polygon, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most.

---

cvFillPoly (view/add comments)

Fills a polygon's interior.

```
void cvFillPoly(
           CvArr* img,
           CvPoint** pts,
           int* npts,
           int contours,
           CvScalar color,
           int lineType=8,
           int shift=0 );
```

img Image

pts Array of pointers to polygons

npts Array of polygon vertex counters

contours Number of contours that bind the filled region

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, and so forth.

cvGetTextSize (view/add comments)

Retrieves the width and height of a text string.

```
void cvGetTextSize(
           const char* textString,
           const CvFont* font,
           CvSize* textSize,
           int* baseline );
```

font Pointer to the font structure

textString Input string

textSize Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.

baseline y-coordinate of the baseline relative to the bottom-most text point

The function calculates the dimensions of a rectangle to enclose a text string when a specified font is used.

cvInitFont (view/add comments)

Initializes font structure.

```
void cvInitFont(
           CvFont* font,
           int fontFace,
           double hscale,
           double vscale,
           double shear=0,
           int thickness=1,
           int lineType=8 );
```

font Pointer to the font structure initialized by the function

fontFace Font name identifier. Only a subset of Hershey fonts http://sources.isc.org/utils/misc/hershey-font.txt are supported now:

- **CV_FONT_HERSHEY_SIMPLEX** normal size sans-serif font
- **CV_FONT_HERSHEY_PLAIN** small size sans-serif font
- **CV_FONT_HERSHEY_DUPLEX** normal size sans-serif font (more complex than CV_FONT_HERSHEY_SIMPLEX)
- **CV_FONT_HERSHEY_COMPLEX** normal size serif font
- **CV_FONT_HERSHEY_TRIPLEX** normal size serif font (more complex than CV_FONT_HERSHEY_COMPLEX)
- **CV_FONT_HERSHEY_COMPLEX_SMALL** smaller version of CV_FONT_HERSHEY_COMPLEX
- **CV_FONT_HERSHEY_SCRIPT_SIMPLEX** hand-writing style font
- **CV_FONT_HERSHEY_SCRIPT_COMPLEX** more complex variant of CV_FONT_HERSHEY_SCRIPT_SIMPLEX

The parameter can be composited from one of the values above and an optional CV_FONT_ITALIC flag, which indicates italic or oblique font.

hscale Horizontal scale. If equal to 1.0f, the characters have the original width depending on the font type. If equal to 0.5f, the characters are of half the original width.

vscale Vertical scale. If equal to 1.0f, the characters have the original height depending on the font type. If equal to 0.5f, the characters are of half the original height.

shear Approximate tangent of the character slope relative to the vertical line. A zero value means a non-italic font, 1.0f means about a 45 degree slope, etc.

thickness Thickness of the text strokes

lineType Type of the strokes, see Line description

The function initializes the font structure that can be passed to text rendering functions.

---

cvInitLineIterator (view/add comments)

Initializes the line iterator.

```
int cvInitLineIterator(
           const CvArr* image,
           CvPoint pt1,
```

```
            CvPoint pt2,
            CvLineIterator* line_iterator,
            int connectivity=8,
            int left_to_right=0 );
```

image Image to sample the line from

pt1 First ending point of the line segment

pt2 Second ending point of the line segment

line_iterator Pointer to the line iterator state structure

connectivity The scanned line connectivity, 4 or 8.

left_to_right If (`left_to_right` $= 0$) then the line is scanned in the specified order, from `pt1` to `pt2`. If (`left_to_right` $\neq 0$) the line is scanned from left-most point to right-most.

The function initializes the line iterator and returns the number of pixels between the two end points. Both points must be inside the image. After the iterator has been initialized, all the points on the raster line that connects the two ending points may be retrieved by successive calls of `CV_NEXT_LINE_POINT` point.

The points on the line are calculated one by one using a 4-connected or 8-connected Bresenham algorithm.

Example: Using line iterator to calculate the sum of pixel values along the color line.

```
CvScalar sum_line_pixels( IplImage* image, CvPoint pt1, CvPoint pt2 )
{
    CvLineIterator iterator;
    int blue_sum = 0, green_sum = 0, red_sum = 0;
    int count = cvInitLineIterator( image, pt1, pt2, &iterator, 8, 0 );

    for( int i = 0; i < count; i++ ){
        blue_sum += iterator.ptr[0];
        green_sum += iterator.ptr[1];
        red_sum += iterator.ptr[2];
        CV_NEXT_LINE_POINT(iterator);

        /* print the pixel coordinates: demonstrates how to calculate the
                                                    coordinates */
        {
```

```
        int offset, x, y;
        /* assume that ROI is not set, otherwise need to take it
                                                    into account. */
        offset = iterator.ptr - (uchar*)(image->imageData);
        y = offset/image->widthStep;
        x = (offset - y*image->widthStep)/(3*sizeof(uchar)
                                        /* size of pixel */);
        printf("(%d,%d)\n", x, y );
        }
    }
    return cvScalar( blue_sum, green_sum, red_sum );
}
```

---

cvLine (view/add comments)

Draws a line segment connecting two points.

```
void cvLine(
            CvArr* img,
            CvPoint pt1,
            CvPoint pt2,
            CvScalar color,
            int thickness=1,
            int lineType=8,
            int shift=0 );
```

img The image

pt1 First point of the line segment

pt2 Second point of the line segment

color Line color

thickness Line thickness

lineType Type of the line:

- **8** (or omitted) 8-connected line.
- **4** 4-connected line.

CV_AA antialiased line.

shift Number of fractional bits in the point coordinates

The function draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB( r, g, b )`.

---

cvPolyLine (view/add comments)

Draws simple or thick polygons.

```
void cvPolyLine(
           CvArr* img,
           CvPoint** pts,
           int* npts,
           int contours,
           int is_closed,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

pts Array of pointers to polygons

npts Array of polygon vertex counters

contours Number of contours that bind the filled region

img Image

is_closed Indicates whether the polylines must be drawn closed. If closed, the function draws the line from the last vertex of every contour to the first vertex.

color Polyline color

thickness Thickness of the polyline edges

lineType Type of the line segments, see Line description

shift Number of fractional bits in the vertex coordinates

The function draws single or multiple polygonal curves.

---

cvPutText (view/add comments)

Draws a text string.

```
void cvPutText(
           CvArr* img,
           const char* text,
           CvPoint org,
           const CvFont* font,
           CvScalar color );
```

img Input image

text String to print

org Coordinates of the bottom-left corner of the first letter

font Pointer to the font structure

color Text color

The function renders the text in the image with the specified font and color. The printed text is clipped by the ROI rectangle. Symbols that do not belong to the specified font are replaced with the symbol for a rectangle.

---

cvRectangle (view/add comments)

Draws a simple, thick, or filled rectangle.

```
void cvRectangle(
           CvArr* img,
           CvPoint pt1,
           CvPoint pt2,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img Image

pt1 One of the rectangle's vertices

pt2 Opposite rectangle vertex

color Line color (RGB) or brightness (grayscale image)

thickness Thickness of lines that make up the rectangle. Negative values, e.g., CV_FILLED, cause the function to draw a filled rectangle.

lineType Type of the line, see Line description

shift Number of fractional bits in the point coordinates

The function draws a rectangle with two opposite corners `pt1` and `pt2`.

CV_RGB (view/add comments)

Constructs a color value.

```
#define CV_RGB( r, g, b ) cvScalar( (b), (g), (r) )
```

red Red component

grn Green component

blu Blue component

1.5 XML/YAML Persistence

CvFileStorage (view/add comments)

File Storage.

```
typedef struct CvFileStorage
{
    ...       // hidden fields
} CvFileStorage;
```

The structure CvFileStorage is a "black box" representation of the file storage associated with a file on disk. Several functions that are described below take `CvFileStorage` as inputs and allow theuser to save or to load hierarchical collections that consist of scalar values, standard CXCore objects (such as matrices, sequences, graphs), and user-defined objects.

CXCore can read and write data in XML (http://www.w3c.org/XML) or YAML (http://www.yaml.org) formats. Below is an example of 3×3 floating-point identity matrix `A`, stored in XML and YAML files using CXCore functions:

XML:

```
<?xml version="1.0">
<opencv_storage>
<A type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>f</dt>
  <data>1. 0. 0. 0. 1. 0. 0. 0. 1.</data>
</A>
</opencv_storage>
```

YAML:

```
%YAML:1.0
A: !!opencv-matrix
  rows: 3
  cols: 3
  dt: f
  data: [ 1., 0., 0., 0., 1., 0., 0., 0., 1.]
```

As it can be seen from the examples, XML uses nested tags to represent hierarchy, while YAML uses indentation for that purpose (similar to the Python programming language).

The same CXCore functions can read and write data in both formats; the particular format is determined by the extension of the opened file, .xml for XML files and .yml or .yaml for YAML.

CvFileNode (view/add comments)

File Storage Node.

```
/* file node type */
#define CV_NODE_NONE        0
#define CV_NODE_INT         1
#define CV_NODE_INTEGER     CV_NODE_INT
#define CV_NODE_REAL        2
#define CV_NODE_FLOAT       CV_NODE_REAL
#define CV_NODE_STR         3
#define CV_NODE_STRING      CV_NODE_STR
#define CV_NODE_REF         4 /* not used */
#define CV_NODE_SEQ         5
#define CV_NODE_MAP         6
#define CV_NODE_TYPE_MASK   7

/* optional flags */
#define CV_NODE_USER        16
#define CV_NODE_EMPTY       32
#define CV_NODE_NAMED       64

#define CV_NODE_TYPE(tag)   ((tag) & CV_NODE_TYPE_MASK)

#define CV_NODE_IS_INT(tag)         (CV_NODE_TYPE(tag) == CV_NODE_INT)
#define CV_NODE_IS_REAL(tag)        (CV_NODE_TYPE(tag) == CV_NODE_REAL)
#define CV_NODE_IS_STRING(tag)      (CV_NODE_TYPE(tag) == CV_NODE_STRING)
#define CV_NODE_IS_SEQ(tag)         (CV_NODE_TYPE(tag) == CV_NODE_SEQ)
#define CV_NODE_IS_MAP(tag)         (CV_NODE_TYPE(tag) == CV_NODE_MAP)
#define CV_NODE_IS_COLLECTION(tag)  (CV_NODE_TYPE(tag) >= CV_NODE_SEQ)
#define CV_NODE_IS_FLOW(tag)        (((tag) & CV_NODE_FLOW) != 0)
#define CV_NODE_IS_EMPTY(tag)       (((tag) & CV_NODE_EMPTY) != 0)
#define CV_NODE_IS_USER(tag)        (((tag) & CV_NODE_USER) != 0)
#define CV_NODE_HAS_NAME(tag)       (((tag) & CV_NODE_NAMED) != 0)

#define CV_NODE_SEQ_SIMPLE 256
#define CV_NODE_SEQ_IS_SIMPLE(seq) (((seq)->flags & CV_NODE_SEQ_SIMPLE) != 0)

typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;
```

```
/* all the keys (names) of elements in the readed file storage
   are stored in the hash to speed up the lookup operations */
typedef struct CvStringHashNode
{
    unsigned hashval;
    CvString str;
    struct CvStringHashNode* next;
}
CvStringHashNode;

/* basic element of the file storage - scalar or collection */
typedef struct CvFileNode
{
    int tag;
    struct CvTypeInfo* info; /* type information
            (only for user-defined object, for others it is 0) */
    union
    {
        double f; /* scalar floating-point number */
        int i;    /* scalar integer number */
        CvString str; /* text string */
        CvSeq* seq; /* sequence (ordered collection of file nodes) */
        struct CvMap* map; /* map (collection of named file nodes) */
    } data;
}
CvFileNode;
```

The structure is used only for retrieving data from file storage (i.e., for loading data from the file). When data is written to a file, it is done sequentially, with minimal buffering. No data is stored in the file storage.

In opposite, when data is read from a file, the whole file is parsed and represented in memory as a tree. Every node of the tree is represented by CvFileNode . The type of file node `N` can be retrieved as `CV_NODE_TYPE(N->tag)`. Some file nodes (leaves) are scalars: text strings, integers, or floating-point numbers. Other file nodes are collections of file nodes, which can be scalars or collections in their turn. There are two types of collections: sequences and maps (we use YAML notation, however, the same is true for XML streams). Sequences (do not mix them with CvSeq) are ordered collections of unnamed file nodes; maps are unordered collections of named file nodes. Thus, elements of sequences are accessed by index (GetSeqElem), while elements of maps are accessed by name (GetFileNodeByName). The table below describes the different types of file nodes:

Type	CV_NODE_TYPE(node->tag)	Value
Integer	CV_NODE_INT	node->data.i
Floating-point	CV_NODE_REAL	node->data.f
Text string	CV_NODE_STR	node->data.str.ptr
Sequence	CV_NODE_SEQ	node->data.seq
Map	CV_NODE_MAP	node->data.map (see below)

There is no need to access the map field directly (by the way, CvMap is a hidden structure). The elements of the map can be retrieved with the GetFileNodeByName function that takes a pointer to the "map" file node.

A user (custom) object is an instance of either one of the standard CxCore types, such as CvMat , CvSeq etc., or any type registered with RegisterTypeInfo . Such an object is initially represented in a file as a map (as shown in XML and YAML example files above) after the file storage has been opened and parsed. Then the object can be decoded (coverted to native representation) by request - when a user calls the Read or ReadByName functions.

CvAttrList (view/add comments)

List of attributes.

```
typedef struct CvAttrList
{
    const char** attr; /* NULL-terminated array of (attribute\_name,attribute\_value) pairs
    struct CvAttrList* next; /* pointer to next chunk of the attributes list */
}
CvAttrList;

/* initializes CvAttrList structure */
inline CvAttrList cvAttrList( const char** attr=NULL, CvAttrList* next=NULL );

/* returns attribute value or 0 (NULL) if there is no such attribute */
const char* cvAttrValue( const CvAttrList* attr, const char* attr\_name );
```

In the current implementation, attributes are used to pass extra parameters when writing user objects (see Write). XML attributes inside tags are not supported, aside from the object type specification (type_id attribute).

CvTypeInfo (view/add comments)

Type information.

```
typedef int (CV_CDECL *CvIsInstanceFunc)( const void* structPtr );
typedef void (CV_CDECL *CvReleaseFunc)( void** structDblPtr );
typedef void* (CV_CDECL *CvReadFunc)( CvFileStorage* storage, CvFileNode* node );
```

```
typedef void (CV_CDECL *CvWriteFunc)( CvFileStorage* storage,
                                      const char* name,
                                      const void* structPtr,
                                      CvAttrList attributes );
typedef void* (CV_CDECL *CvCloneFunc)( const void* structPtr );

typedef struct CvTypeInfo
{
    int flags; /* not used */
    int header_size; /* sizeof(CvTypeInfo) */
    struct CvTypeInfo* prev; /* previous registered type in the list */
    struct CvTypeInfo* next; /* next registered type in the list */
    const char* type_name; /* type name, written to file storage */

    /* methods */
    CvIsInstanceFunc is_instance; /* checks if the passed object belongs to the type */
    CvReleaseFunc release; /* releases object (memory etc.) */
    CvReadFunc read; /* reads object from file storage */
    CvWriteFunc write; /* writes object to file storage */
    CvCloneFunc clone; /* creates a copy of the object */
}
CvTypeInfo;
```

The structure CvTypeInfo contains information about one of the standard or user-defined types. Instances of the type may or may not contain a pointer to the corresponding CvTypeInfo structure. In any case, there is a way to find the type info structure for a given object using the TypeOf function. Aternatively, type info can be found by type name using FindType , which is used when an object is read from file storage. The user can register a new type with RegisterType that adds the type information structure into the beginning of the type list. Thus, it is possible to create specialized types from generic standard types and override the basic methods.

---

cvClone (view/add comments)

Makes a clone of an object.

```
void* cvClone( const void* structPtr );
```

structPtr The object to clone

The function finds the type of a given object and calls `clone` with the passed object.

cvEndWriteStruct (view/add comments)

Ends the writing of a structure.

```
void cvEndWriteStruct(CvFileStorage* fs);
```

fs File storage

The function finishes the currently written structure.

cvFindType (view/add comments)

Finds a type by its name.

```
CvTypeInfo* cvFindType(const char* typeName);
```

typeName Type name

The function finds a registered type by its name. It returns NULL if there is no type with the specified name.

cvFirstType (view/add comments)

Returns the beginning of a type list.

```
CvTypeInfo* cvFirstType(void);
```

The function returns the first type in the list of registered types. Navigation through the list can be done via the `prev` and `next` fields of the CvTypeInfo structure.

cvGetFileNode (view/add comments)

Finds a node in a map or file storage.

```
CvFileNode* cvGetFileNode(
           CvFileStorage* fs,
           CvFileNode* map,
           const CvStringHashNode* key,
           int createMissing=0 );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node. If both `map` and `key` are NULLs, the function returns the root file node - a map that contains top-level nodes.

key Unique pointer to the node name, retrieved with GetHashedKey

createMissing Flag that specifies whether an absent node should be added to the map

The function finds a file node. It is a faster version of GetFileNodeByName (see GetHashedKey discussion). Also, the function can insert a new node, if it is not in the map yet.

cvGetFileNodeByName (view/add comments)

Finds a node in a map or file storage.

```
CvFileNode* cvGetFileNodeByName(
           const CvFileStorage* fs,
           const CvFileNode* map,
           const char* name);
```

fs File storage

map The parent map. If it is NULL, the function searches in all the top-level nodes (streams), starting with the first one.

name The file node name

The function finds a file node by `name`. The node is searched either in `map` or, if the pointer is NULL, among the top-level file storage nodes. Using this function for maps and GetSeqElem (or sequence reader) for sequences, it is possible to nagivate through the file storage. To speed up multiple queries for a certain key (e.g., in the case of an array of structures) one may use a combination of GetHashedKey and GetFileNode .

---

cvGetFileNodeName (view/add comments)

Returns the name of a file node.

```
const char* cvGetFileNodeName( const CvFileNode* node );
```

node File node

The function returns the name of a file node or NULL, if the file node does not have a name or if `node` is `NULL`.

---

cvGetHashedKey (view/add comments)

Returns a unique pointer for a given name.

```
CvStringHashNode* cvGetHashedKey(
            CvFileStorage* fs,
            const char* name,
            int len=-1,
            int createMissing=0 );
```

fs File storage

name Literal node name

len Length of the name (if it is known apriori), or -1 if it needs to be calculated

createMissing Flag that specifies, whether an absent key should be added into the hash table

The function returns a unique pointer for each particular file node name. This pointer can be then passed to the GetFileNode function that is faster than GetFileNodeByName because it compares text strings by comparing pointers rather than the strings' content.

Consider the following example where an array of points is encoded as a sequence of 2-entry maps:

```
%YAML:1.0
points:
  - { x: 10, y: 10 }
  - { x: 20, y: 20 }
  - { x: 30, y: 30 }
  # ...
```

Then, it is possible to get hashed "x" and "y" pointers to speed up decoding of the points.

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvFileStorage* fs = cvOpenFileStorage( "points.yml", 0, CV\_STORAGE\_READ );
    CvStringHashNode* x\_key = cvGetHashedNode( fs, "x", -1, 1 );
    CvStringHashNode* y\_key = cvGetHashedNode( fs, "y", -1, 1 );
    CvFileNode* points = cvGetFileNodeByName( fs, 0, "points" );

    if( CV\_NODE\_IS\_SEQ(points->tag) )
    {
        CvSeq* seq = points->data.seq;
        int i, total = seq->total;
        CvSeqReader reader;
        cvStartReadSeq( seq, &reader, 0 );
        for( i = 0; i < total; i++ )
        {
            CvFileNode* pt = (CvFileNode*)reader.ptr;
#if 1 /* faster variant */
            CvFileNode* xnode = cvGetFileNode( fs, pt, x\_key, 0 );
            CvFileNode* ynode = cvGetFileNode( fs, pt, y\_key, 0 );
            assert( xnode && CV\_NODE\_IS\_INT(xnode->tag) &&
                    ynode && CV\_NODE\_IS\_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#elif 1 /* slower variant; does not use x\_key & y\_key */
            CvFileNode* xnode = cvGetFileNodeByName( fs, pt, "x" );
            CvFileNode* ynode = cvGetFileNodeByName( fs, pt, "y" );
            assert( xnode && CV\_NODE\_IS\_INT(xnode->tag) &&
```

```
                    ynode && CV\_NODE\_IS\_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#else /* the slowest yet the easiest to use variant */
            int x = cvReadIntByName( fs, pt, "x", 0 /* default value */ );
            int y = cvReadIntByName( fs, pt, "y", 0 /* default value */ );
#endif
            CV\_NEXT\_SEQ\_ELEM( seq->elem\_size, reader );
            printf("%d: (%d, %d)\n", i, x, y );
        }
    }
    cvReleaseFileStorage( &fs );
    return 0;
}
```

Please note that whatever method of accessing a map you are using, it is still much slower than using plain sequences; for example, in the above example, it is more efficient to encode the points as pairs of integers in a single numeric sequence.

---

cvGetRootFileNode (view/add comments)

Retrieves one of the top-level nodes of the file storage.

```
CvFileNode* cvGetRootFileNode(
            const CvFileStorage* fs,
            int stream_index=0 );
```

fs File storage

stream_index Zero-based index of the stream. See StartNextStream . In most cases, there is only one stream in the file; however, there can be several.

The function returns one of the top-level file nodes. The top-level nodes do not have a name, they correspond to the streams that are stored one after another in the file storage. If the index is out of range, the function returns a NULL pointer, so all the top-level nodes may be iterated by subsequent calls to the function with `stream_index=0,1,...`, until the NULL pointer is returned. This function may be used as a base for recursive traversal of the file storage.

cvLoad (view/add comments)

Loads an object from a file.

```
void* cvLoad(
           const char* filename,
           CvMemStorage* storage=NULL,
           const char* name=NULL,
           const char** realName=NULL );
```

filename File name

storage Memory storage for dynamic structures, such as CvSeq or CvGraph . It is not used for matrices or images.

name Optional object name. If it is NULL, the first top-level object in the storage will be loaded.

realName Optional output parameter that will contain the name of the loaded object (useful if name=NULL)

The function loads an object from a file. It provides a simple interface to cvRead. After the object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as a sequence, contour, or graph, one should pass a valid memory storage destination to the function.

cvOpenFileStorage (view/add comments)

Opens file storage for reading or writing data.

```
CvFileStorage* cvOpenFileStorage(
           const char* filename,
           CvMemStorage* memstorage,
           int flags);
```

filename Name of the file associated with the storage

memstorage Memory storage used for temporary data and for storing dynamic structures, such as CvSeq or CvGraph . If it is NULL, a temporary memory storage is created and used.

flags Can be one of the following:

CV_STORAGE_READ the storage is open for reading

CV_STORAGE_WRITE the storage is open for writing

The function opens file storage for reading or writing data. In the latter case, a new file is created or an existing file is rewritten. The type of the read or written file is determined by the filename extension: `.xml` for `XML` and `.yml` or `.yaml` for `YAML`. The function returns a pointer to the CvFileStorage structure.

---

cvRead (view/add comments)

Decodes an object and returns a pointer to it.

```
void* cvRead(
           CvFileStorage* fs,
           CvFileNode* node,
           CvAttrList* attributes=NULL );
```

fs File storage

node The root object node

attributes Unused parameter

The function decodes a user object (creates an object in a native representation from the file storage subtree) and returns it. The object to be decoded must be an instance of a registered type that supports the `read` method (see CvTypeInfo). The type of the object is determined by the type name that is encoded in the file. If the object is a dynamic structure, it is created either in memory storage and passed to OpenFileStorage or, if a NULL pointer was passed, in temporary memory storage, which is released when ReleaseFileStorage is called. Otherwise, if the object is not a dynamic structure, it is created in a heap and should be released with a specialized function or by using the generic Release .

cvReadByName (view/add comments)

Finds an object by name and decodes it.

```
void* cvReadByName(
            CvFileStorage* fs,
            const CvFileNode* map,
            const char* name,
            CvAttrList* attributes=NULL );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

attributes Unused parameter

The function is a simple superposition of GetFileNodeByName and Read .

cvReadInt (view/add comments)

Retrieves an integer value from a file node.

```
int cvReadInt(
            const CvFileNode* node,
            int defaultValue=0 );
```

node File node

defaultValue The value that is returned if `node` is NULL

The function returns an integer that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after GetFileNode without checking for a NULL pointer). If the file node has type `CV_NODE_INT`, then `node->data.i` is returned. If the file node has type `CV_NODE_REAL`, then `node->data.f` is converted to an integer and returned. Otherwise the result is not determined.

cvReadIntByName (view/add comments)

Finds a file node and returns its value.

```
int cvReadIntByName(
            const CvFileStorage* fs,
            const CvFileNode* map,
            const char* name,
            int defaultValue=0 );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

defaultValue The value that is returned if the file node is not found

The function is a simple superposition of GetFileNodeByName and ReadInt .

cvReadRawData (view/add comments)

Reads multiple numbers.

```
void cvReadRawData(
            const CvFileStorage* fs,
            const CvFileNode* src,
            void* dst,
            const char* dt);
```

fs File storage

src The file node (a sequence) to read numbers from

dst Pointer to the destination array

dt Specification of each array element. It has the same format as in WriteRawData .

The function reads elements from a file node that represents a sequence of scalars.

cvReadRawDataSlice (view/add comments)

Initializes file node sequence reader.

```
void cvReadRawDataSlice(
           const CvFileStorage* fs,
           CvSeqReader* reader,
           int count,
           void* dst,
           const char* dt );
```

fs File storage

reader The sequence reader. Initialize it with StartReadRawData .

count The number of elements to read

dst Pointer to the destination array

dt Specification of each array element. It has the same format as in WriteRawData .

The function reads one or more elements from the file node, representing a sequence, to a user-specified array. The total number of read sequence elements is a product of `total` and the number of components in each array element. For example, if dt=`2if`, the function will read `total` $\times 3$ sequence elements. As with any sequence, some parts of the file node sequence may be skipped or read repeatedly by repositioning the reader using SetSeqReaderPos .

cvReadReal (view/add comments)

Retrieves a floating-point value from a file node.

```
double cvReadReal(
           const CvFileNode* node,
           double defaultValue=0. );
```

node File node

defaultValue The value that is returned if `node` is NULL

The function returns a floating-point value that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after GetFileNode without checking for a NULL pointer). If the file node has type `CV_NODE_REAL`, then `node->data.f` is returned. If the file node has type `CV_NODE_INT`, then `node->data.f` is converted to floating-point and returned. Otherwise the result is not determined.

---

cvReadRealByName (view/add comments)

Finds a file node and returns its value.

```
double cvReadRealByName(
           const CvFileStorage* fs,
           const CvFileNode* map,
           const char* name,
           double defaultValue=0.);
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

defaultValue The value that is returned if the file node is not found

The function is a simple superposition of GetFileNodeByName and ReadReal .

---

cvReadString (view/add comments)

Retrieves a text string from a file node.

```
const char* cvReadString(
           const CvFileNode* node,
           const char* defaultValue=NULL );
```

node File node

defaultValue The value that is returned if `node` is NULL

The function returns a text string that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after GetFileNode without checking for a NULL pointer). If the file node has type `CV_NODE_STR`, then `node->data.str.ptr` is returned. Otherwise the result is not determined.

---

cvReadStringByName (view/add comments)

Finds a file node by its name and returns its value.

```
const char* cvReadStringByName(
            const CvFileStorage* fs,
            const CvFileNode* map,
            const char* name,
            const char* defaultValue=NULL );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

defaultValue The value that is returned if the file node is not found

The function is a simple superposition of GetFileNodeByName and ReadString .

---

cvRegisterType (view/add comments)

Registers a new type.

```
void cvRegisterType(const CvTypeInfo* info);
```

info Type info structure

The function registers a new type, which is described by `info`. The function creates a copy of the structure, so the user should delete it after calling the function.

cvRelease (view/add comments)

Releases an object.

```
void cvRelease( void** structPtr );
```

structPtr Double pointer to the object

The function finds the type of a given object and calls `release` with the double pointer.

cvReleaseFileStorage (view/add comments)

Releases file storage.

```
void cvReleaseFileStorage(CvFileStorage** fs);
```

fs Double pointer to the released file storage

The function closes the file associated with the storage and releases all the temporary structures. It must be called after all I/O operations with the storage are finished.

cvSave (view/add comments)

Saves an object to a file.

```
void cvSave(
           const char* filename,
           const void* structPtr,
           const char* name=NULL,
           const char* comment=NULL,
           CvAttrList attributes=cvAttrList());
```

filename File name

structPtr Object to save

name Optional object name. If it is NULL, the name will be formed from `filename`.

comment Optional comment to put in the beginning of the file

attributes Optional attributes passed to Write

The function saves an object to a file. It provides a simple interface to Write .

---

cvStartNextStream (view/add comments)

Starts the next stream.

```
void cvStartNextStream(CvFileStorage* fs);
```

fs File storage

The function starts the next stream in file storage. Both YAML and XML support multiple "streams." This is useful for concatenating files or for resuming the writing process.

---

cvStartReadRawData (view/add comments)

Initializes the file node sequence reader.

```
void cvStartReadRawData(
           const CvFileStorage* fs,
           const CvFileNode* src,
           CvSeqReader* reader);
```

fs File storage

src The file node (a sequence) to read numbers from

reader Pointer to the sequence reader

The function initializes the sequence reader to read data from a file node. The initialized reader can be then passed to ReadRawDataSlice .

cvStartWriteStruct (view/add comments)

Starts writing a new structure.

```
void cvStartWriteStruct( CvFileStorage* fs,
            const char* name,
            int struct_flags,
            const char* typeName=NULL,
            CvAttrList attributes=cvAttrList(
            ));
```

fs File storage

name Name of the written structure. The structure can be accessed by this name when the storage is read.

struct_flags A combination one of the following values:

- **CV_NODE_SEQ** the written structure is a sequence (see discussion of CvFileStorage), that is, its elements do not have a name.
- **CV_NODE_MAP** the written structure is a map (see discussion of CvFileStorage), that is, all its elements have names.

One and only one of the two above flags must be specified

CV_NODE_FLOW the optional flag that makes sense only for YAML streams. It means that the structure is written as a flow (not as a block), which is more compact. It is recommended to use this flag for structures or arrays whose elements are all scalars.

typeName Optional parameter - the object type name. In case of XML it is written as a `type_id` attribute of the structure opening tag. In the case of YAML it is written after a colon following the structure name (see the example in CvFileStorage description). Mainly it is used with user objects. When the storage is read, the encoded type name is used to determine the object type (see CvTypeInfo and FindTypeInfo).

attributes This parameter is not used in the current implementation

The function starts writing a compound structure (collection) that can be a sequence or a map. After all the structure fields, which can be scalars or structures, are written, EndWriteStruct should be called. The function can be used to group some objects or to implement the `write` function for a some user object (see CvTypeInfo).

cvTypeOf (view/add comments)

Returns the type of an object.

```
CvTypeInfo* cvTypeOf( const void* structPtr );
```

structPtr The object pointer

The function finds the type of a given object. It iterates through the list of registered types and calls the `is_instance` function/method for every type info structure with that object until one of them returns non-zero or until the whole list has been traversed. In the latter case, the function returns NULL.

cvUnregisterType (view/add comments)

Unregisters the type.

```
void cvUnregisterType( const char* typeName );
```

typeName Name of an unregistered type

The function unregisters a type with a specified name. If the name is unknown, it is possible to locate the type info by an instance of the type using TypeOf or by iterating the type list, starting from FirstType , and then calling `cvUnregisterType(info->typeName)`.

cvWrite (view/add comments)

Writes a user object.

```
void cvWrite( CvFileStorage* fs,
              const char* name,
              const void* ptr,
              CvAttrList attributes=cvAttrList(
              ) );
```

fs File storage

name Name of the written object. Should be NULL if and only if the parent structure is a sequence.

ptr Pointer to the object

attributes The attributes of the object. They are specific for each particular type (see the dicsussion below).

The function writes an object to file storage. First, the appropriate type info is found using TypeOf . Then, the `write` method associated with the type info is called.

Attributes are used to customize the writing procedure. The standard types support the following attributes (all the `*dt` attributes have the same format as in WriteRawData):

1. CvSeq

   header_dt description of user fields of the sequence header that follow CvSeq, or CvChain (if the sequence is a Freeman chain) or CvContour (if the sequence is a contour or point sequence)

   dt description of the sequence elements.

   recursive if the attribute is present and is not equal to ”0” or ”false”, the whole tree of sequences (contours) is stored.

2. Cvgraph

   header_dt description of user fields of the graph header that follows CvGraph;

   vertex_dt description of user fields of graph vertices

   edge_dt description of user fields of graph edges (note that the edge weight is always written, so there is no need to specify it explicitly)

Below is the code that creates the YAML file shown in the `CvFileStorage` description:

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvMat* mat = cvCreateMat( 3, 3, CV\_32F );
    CvFileStorage* fs = cvOpenFileStorage( "example.yml", 0, CV\_STORAGE\_WRITE );

    cvSetIdentity( mat );
    cvWrite( fs, "A", mat, cvAttrList(0,0) );

    cvReleaseFileStorage( &fs );
```

```
    cvReleaseMat( &mat );
    return 0;
}
```

cvWriteComment (view/add comments)

Writes a comment.

```
void cvWriteComment(
            CvFileStorage* fs,
            const char* comment,
            int eolComment);
```

fs File storage

comment The written comment, single-line or multi-line

eolComment If non-zero, the function tries to put the comment at the end of current line. If the flag is zero, if the comment is multi-line, or if it does not fit at the end of the current line, the comment starts a new line.

The function writes a comment into file storage. The comments are skipped when the storage is read, so they may be used only for debugging or descriptive purposes.

cvWriteFileNode (view/add comments)

Writes a file node to another file storage.

```
void cvWriteFileNode(
            CvFileStorage* fs,
            const char* new_node_name,
            const CvFileNode* node,
            int embed );
```

fs Destination file storage

new_file_node New name of the file node in the destination file storage. To keep the existing name, use cvGetFileNodeName

node The written node

embed If the written node is a collection and this parameter is not zero, no extra level of hiararchy is created. Instead, all the elements of `node` are written into the currently written structure. Of course, map elements may be written only to a map, and sequence elements may be written only to a sequence.

The function writes a copy of a file node to file storage. Possible applications of the function are merging several file storages into one and conversion between XML and YAML formats.

---

cvWriteInt (view/add comments)

Writes an integer value.

```
void cvWriteInt(
           CvFileStorage* fs,
           const char* name,
           int value);
```

fs File storage

name Name of the written value. Should be NULL if and only if the parent structure is a sequence.

value The written value

The function writes a single integer value (with or without a name) to the file storage.

---

cvWriteRawData (view/add comments)

Writes multiple numbers.

```
void cvWriteRawData(
           CvFileStorage* fs,
           const void* src,
           int len,
           const char* dt );
```

fs File storage

src Pointer to the written array

len Number of the array elements to write

dt Specification of each array element that has the following format `([count]{'u'|'c'|'w'|'s'|'i'|'f'|'d'})...` where the characters correspond to fundamental C types:

- **u** 8-bit unsigned number
- **c** 8-bit signed number
- **w** 16-bit unsigned number
- **s** 16-bit signed number
- **i** 32-bit signed number
- **f** single precision floating-point number
- **d** double precision floating-point number
- **r** pointer, 32 lower bits of which are written as a signed integer. The type can be used to store structures with links between the elements. `count` is the optional counter of values of a given type. For example, `2if` means that each array element is a structure of 2 integers, followed by a single-precision floating-point number. The equivalent notations of the above specification are '`iif`', '`2i1f`' and so forth. Other examples: `u` means that the array consists of bytes, and `2d` means the array consists of pairs of doubles.

The function writes an array, whose elements consist of single or multiple numbers. The function call can be replaced with a loop containing a few WriteInt and WriteReal calls, but a single call is more efficient. Note that because none of the elements have a name, they should be written to a sequence rather than a map.

cvWriteReal (view/add comments)

Writes a floating-point value.

```
void cvWriteReal(
           CvFileStorage* fs,
           const char* name,
           double value );
```

fs File storage

name Name of the written value. Should be NULL if and only if the parent structure is a sequence.

value The written value

The function writes a single floating-point value (with or without a name) to file storage. Special values are encoded as follows: NaN (Not A Number) as .NaN, $\pm\infty$ as +.Inf (-.Inf).

The following example shows how to use the low-level writing functions to store custom structures, such as termination criteria, without registering a new type.

```
void write_termcriteria( CvFileStorage* fs, const char* struct_name,
                         CvTermCriteria* termcrit )
{
    cvStartWriteStruct( fs, struct_name, CV_NODE_MAP, NULL, cvAttrList(0,0));
    cvWriteComment( fs, "termination criteria", 1 ); // just a description
    if( termcrit->type & CV_TERMCRIT_ITER )
        cvWriteInteger( fs, "max_iterations", termcrit->max_iter );
    if( termcrit->type & CV_TERMCRIT_EPS )
        cvWriteReal( fs, "accuracy", termcrit->epsilon );
    cvEndWriteStruct( fs );
}
```

cvWriteString (view/add comments)

Writes a text string.

```
void cvWriteString(
           CvFileStorage* fs,
           const char* name,
           const char* str,
           int quote=0 );
```

fs File storage

name Name of the written string . Should be NULL if and only if the parent structure is a sequence.

str The written text string

quote If non-zero, the written string is put in quotes, regardless of whether they are required. Otherwise, if the flag is zero, quotes are used only when they are required (e.g. when the string starts with a digit or contains spaces).

The function writes a text string to file storage.

1.6 Clustering

---

cvKMeans2 (view/add comments)

Splits set of vectors by a given number of clusters.

```
int cvKMeans2(const CvArr* samples, int nclusters,
           CvArr* labels, CvTermCriteria termcrit,
           int attempts=1, CvRNG* rng=0,
           int flags=0, CvArr* centers=0,
           double* compactness=0);
```

samples Floating-point matrix of input samples, one row per sample

nclusters Number of clusters to split the set by

labels Output integer vector storing cluster indices for every sample

termcrit Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

attempts How many times the algorithm is executed using different initial labelings. The algorithm returns labels that yield the best compactness (see the last function parameter)

rng Optional external random number generator; can be used to fully control the function behaviour

flags Can be 0 or `CV_KMEANS_USE_INITIAL_LABELS`. The latter value means that during the first (and possibly the only) attempt, the function uses the user-supplied labels as the initial approximation instead of generating random labels. For the second and further attempts, the function will use randomly generated labels in any case

centers The optional output array of the cluster centers

compactness The optional output parameter, which is computed as $\sum_i ||\texttt{samples}_i - \texttt{centers}_{\texttt{labels}_i}||^2$ after every attempt; the best (minimum) value is chosen and the corresponding labels are returned by the function. Basically, the user can use only the core of the function, set the number of attempts to 1, initialize labels each time using a custom algorithm (`flags=CV_KMEAN_USE_INITIAL_I` and, based on the output compactness or any other criteria, choose the best clustering.

The function `cvKMeans2` implements a k-means algorithm that finds the centers of `nclusters` clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a cluster index for samples stored in the i-th row of the `samples` matrix.

```
#include "cxcore.h"
#include "highgui.h"

void main( int argc, char** argv )
{
    #define MAX_CLUSTERS 5
    CvScalar color_tab[MAX_CLUSTERS];
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG rng = cvRNG(0xffffffff);

    color_tab[0] = CV_RGB(255,0,0);
    color_tab[1] = CV_RGB(0,255,0);
    color_tab[2] = CV_RGB(100,100,255);
    color_tab[3] = CV_RGB(255,0,255);
    color_tab[4] = CV_RGB(255,255,0);

    cvNamedWindow( "clusters", 1 );
```

```
    for(;;)
    {
        int k, cluster_count = cvRandInt(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandInt(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

        /* generate random sample from multigaussian distribution */
        for( k = 0; k < cluster_count; k++ )
        {
            CvPoint center;
            CvMat point_chunk;
            center.x = cvRandInt(&rng)%img->width;
            center.y = cvRandInt(&rng)%img->height;
            cvGetRows( points,
                       &point_chunk,
                       k*sample_count/cluster_count,
                       (k == (cluster_count - 1)) ?
                           sample_count :
                           (k+1)*sample_count/cluster_count );
            cvRandArr( &rng, &point_chunk, CV_RAND_NORMAL,
                       cvScalar(center.x,center.y,0,0),
                       cvScalar(img->width/6, img->height/6,0,0) );
        }

        /* shuffle samples */
        for( i = 0; i < sample_count/2; i++ )
        {
            CvPoint2D32f* pt1 =
                (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)%sample_count;
            CvPoint2D32f* pt2 =
                (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)%sample_count;
            CvPoint2D32f temp;
            CV_SWAP( *pt1, *pt2, temp );
        }

        cvKMeans2( points, cluster_count, clusters,
                   cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

        cvZero( img );

        for( i = 0; i < sample_count; i++ )
        {
            CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
            int cluster_idx = clusters->data.i[i];
```

```
            cvCircle( img,
                      cvPointFrom32f(pt),
                      2,
                      color_tab[cluster_idx],
                      CV_FILLED );
        }

        cvReleaseMat( &points );
        cvReleaseMat( &clusters );

        cvShowImage( "clusters", img );

        int key = cvWaitKey(0);
        if( key == 27 )
            break;
    }
}
```

cvSeqPartition (view/add comments)

Splits a sequence into equivalency classes.

```
int cvSeqPartition(
            const CvSeq* seq,
            CvMemStorage* storage,
            CvSeq** labels,
            CvCmpFunc is_equal,
            void* userdata );
```

seq The sequence to partition

storage The storage block to store the sequence of equivalency classes. If it is NULL, the function uses `seq->storage` for output labels

labels Ouput parameter. Double pointer to the sequence of 0-based labels of input sequence elements

is_equal The relation function that should return non-zero if the two particular sequence elements are from the same class, and zero otherwise. The partitioning algorithm uses transitive closure of the relation function as an equivalency critria

userdata Pointer that is transparently passed to the `is_equal` function

```
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
```

The function `cvSeqPartition` implements a quadratic algorithm for splitting a set into one or more equivalancy classes. The function returns the number of equivalency classes.

```
#include "cxcore.h"
#include "highgui.h"
#include <stdio.h>

CvSeq* point_seq = 0;
IplImage* canvas = 0;
CvScalar* colors = 0;
int pos = 10;

int is_equal( const void* _a, const void* _b, void* userdata )
{
    CvPoint a = *(const CvPoint*)_a;
    CvPoint b = *(const CvPoint*)_b;
    double threshold = *(double*)userdata;
    return (double)((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y)) <=
        threshold;
}

void on_track( int pos )
{
    CvSeq* labels = 0;
    double threshold = pos*pos;
    int i, class_count = cvSeqPartition( point_seq,
                                         0,
                                         &labels,
                                         is_equal,
                                         &threshold );
    printf("%4d classes\n", class_count );
    cvZero( canvas );

    for( i = 0; i < labels->total; i++ )
    {
        CvPoint pt = *(CvPoint*)cvGetSeqElem( point_seq, i );
        CvScalar color = colors[*(int*)cvGetSeqElem( labels, i )];
        cvCircle( canvas, pt, 1, color, -1 );
    }
```

```
    cvShowImage( "points", canvas );
}

int main( int argc, char** argv )
{
    CvMemStorage* storage = cvCreateMemStorage(0);
    point_seq = cvCreateSeq( CV_32SC2,
                             sizeof(CvSeq),
                             sizeof(CvPoint),
                             storage );
    CvRNG rng = cvRNG(0xffffffff);

    int width = 500, height = 500;
    int i, count = 1000;
    canvas = cvCreateImage( cvSize(width,height), 8, 3 );

    colors = (CvScalar*)cvAlloc( count*sizeof(colors[0]) );
    for( i = 0; i < count; i++ )
    {
        CvPoint pt;
        int icolor;
        pt.x = cvRandInt( &rng ) % width;
        pt.y = cvRandInt( &rng ) % height;
        cvSeqPush( point_seq, &pt );
        icolor = cvRandInt( &rng ) | 0x00404040;
        colors[i] = CV_RGB(icolor & 255,
                           (icolor >> 8)&255,
                           (icolor >> 16)&255);
    }

    cvNamedWindow( "points", 1 );
    cvCreateTrackbar( "threshold", "points", &pos, 50, on_track );
    on_track(pos);
    cvWaitKey(0);
    return 0;
}
```

1.7 Utility and System Functions and Macros

Error Handling

Error handling in OpenCV is similar to IPL (Image Processing Library). In the case of an error, functions do not return the error code. Instead, they raise an error using `CV_ERROR` macro that

calls cvError that, in its turn, sets the error status with cvSetErrStatus and calls a standard or user-defined error handler (that can display a message box, write to log, etc., see cvRedirectError). There is a global variable, one per each program thread, that contains current error status (an integer value). The status can be retrieved with the cvGetErrStatus function.

There are three modes of error handling (see cvSetErrMode and cvGetErrMode):

- **Leaf**. The program is terminated after the error handler is called. This is the default value. It is useful for debugging, as the error is signalled immediately after it occurs. However, for production systems, other two methods may be preferable as they provide more control.
- **Parent**. The program is not terminated, but the error handler is called. The stack is unwound (it is done w/o using a C++ exception mechanism). The user may check error code after calling the `CxCore` function with cvGetErrStatus and react.
- **Silent**. Similar to `Parent` mode, but no error handler is called.

Actually, the semantics of the `Leaf` and `Parent` modes are implemented by error handlers and the above description is true for them. cvGuiBoxReport behaves slightly differently, and some custom error handlers may implement quite different semantics.

Macros for raising an error, checking for errors, etc.

```
/* special macros for enclosing processing statements within a function and separating
   them from prologue (resource initialization) and epilogue (guaranteed resource release)
#define __BEGIN__       {
#define __END__         goto exit; exit: ; }
/* proceeds to "resource release" stage */
#define EXIT            goto exit

/* Declares locally the function name for CV_ERROR() use */
#define CV_FUNCNAME( Name )  \
    static char cvFuncName[] = Name

/* Raises an error within the current context */
#define CV_ERROR( Code, Msg )                                           \
{                                                                       \
     cvError( (Code), cvFuncName, Msg, __FILE__, __LINE__ );            \
     EXIT;                                                              \
}

/* Checks status after calling CXCORE function */
#define CV_CHECK()                                                      \
{                                                                       \
    if( cvGetErrStatus() < 0 )                                      \
```

```
        CV_ERROR( CV_StsBackTrace, "Inner function failed." );      \
}

/* Provies shorthand for CXCORE function call and CV_CHECK() */
#define CV_CALL( Statement )                                        \
{                                                                   \
    Statement;                                                      \
    CV_CHECK();                                                     \
}

/* Checks some condition in both debug and release configurations */
#define CV_ASSERT( Condition )                                        \
{                                                                     \
    if( !(Condition) )                                                \
        CV_ERROR( CV_StsInternal, "Assertion: " #Condition " failed" ); \
}

/* these macros are similar to their CV_... counterparts, but they
   do not need exit label nor cvFuncName to be defined */
#define OPENCV_ERROR(status,func_name,err_msg) ...
#define OPENCV_ERRCHK(func_name,err_msg) ...
#define OPENCV_ASSERT(condition,func_name,err_msg) ...
#define OPENCV_CALL(statement) ...
```

Instead of a discussion, below is a documented example of a typical CXCORE function and an example of the function use.

Example: Use of Error Handling Macros

```
#include "cxcore.h"
#include <stdio.h>

void cvResizeDCT( CvMat* input_array, CvMat* output_array )
{
    CvMat* temp_array = 0; // declare pointer that should be released anyway.

    CV_FUNCNAME( "cvResizeDCT" ); // declare cvFuncName

    __BEGIN__; // start processing. There may be some declarations just after
              // this macro, but they could not be accessed from the epilogue.

    if( !CV_IS_MAT(input_array) || !CV_IS_MAT(output_array) )
        // use CV_ERROR() to raise an error
```

```
        CV_ERROR( CV_StsBadArg,
        "input_array or output_array are not valid matrices" );

    // some restrictions that are going to be removed later, may be checked
    // with CV_ASSERT()
    CV_ASSERT( input_array->rows == 1 && output_array->rows == 1 );

    // use CV_CALL for safe function call
    CV_CALL( temp_array = cvCreateMat( input_array->rows,
                                       MAX(input_array->cols,
                                       output_array->cols),
                                       input_array->type ));

    if( output_array->cols > input_array->cols )
        CV_CALL( cvZero( temp_array ));

    temp_array->cols = input_array->cols;
    CV_CALL( cvDCT( input_array, temp_array, CV_DXT_FORWARD ));
    temp_array->cols = output_array->cols;
    CV_CALL( cvDCT( temp_array, output_array, CV_DXT_INVERSE ));
    CV_CALL( cvScale( output_array,
                      output_array,
                      1./sqrt((double)input_array->cols*output_array->cols), 0 ));

    __END__; // finish processing. Epilogue follows after the macro.

    // release temp_array. If temp_array has not been allocated
    // before an error occured, cvReleaseMat
    // takes care of it and does nothing in this case.
    cvReleaseMat( &temp_array );
}

int main( int argc, char** argv )
{
    CvMat* src = cvCreateMat( 1, 512, CV_32F );
#if 1 /* no errors */
    CvMat* dst = cvCreateMat( 1, 256, CV_32F );
#else
    CvMat* dst = 0; /* test error processing mechanism */
#endif
    cvSet( src, cvRealScalar(1.), 0 );
#if 0 /* change 0 to 1 to suppress error handler invocation */
    cvSetErrMode( CV_ErrModeSilent );
#endif
    cvResizeDCT( src, dst ); // if some error occurs, the message
```

```
                                // box will popup, or a message will be
                                // written to log, or some user-defined
                                // processing will be done
    if( cvGetErrStatus() < 0 )
        printf("Some error occured" );
    else
        printf("Everything is OK" );
    return 0;
}
```

cvGetErrStatus (view/add comments)

Returns the current error status.

```
int cvGetErrStatus( void );
```

The function returns the current error status - the value set with the last cvSetErrStatus call. Note that in `Leaf` mode, the program terminates immediately after an error occurs, so to always gain control after the function call, one should call cvSetErrMode and set the `Parent` or `Silent` error mode.

cvSetErrStatus (view/add comments)

Sets the error status.

```
void cvSetErrStatus( int status );
```

`status` The error status

The function sets the error status to the specified value. Mostly, the function is used to reset the error status (set to it `CV_StsOk`) to recover after an error. In other cases it is more natural to call cvError or `CV_ERROR`.

cvGetErrMode (view/add comments)

Returns the current error mode.

```
int cvGetErrMode(void);
```

The function returns the current error mode - the value set with the last cvSetErrMode call.

---

cvSetErrMode (view/add comments)

Sets the error mode.

```
#define CV_ErrModeLeaf    0
#define CV_ErrModeParent  1
#define CV_ErrModeSilent  2
```

```
int cvSetErrMode( int mode );
```

mode The error mode

The function sets the specified error mode. For descriptions of different error modes, see the beginning of the error section.

---

cvError (view/add comments)

Raises an error.

```
int cvError(
            int status,
            const char* func_name,
            const char* err_msg,
            const char* filename,
            int line );
```

status The error status

func_name Name of the function where the error occured

err_msg Additional information/diagnostics about the error

filename Name of the file where the error occured

line Line number, where the error occured

The function sets the error status to the specified value (via cvSetErrStatus) and, if the error mode is not `Silent`, calls the error handler.

---

cvErrorStr (view/add comments)

Returns textual description of an error status code.

```
const char* cvErrorStr( int status );
```

status The error status

The function returns the textual description for the specified error status code. In the case of unknown status, the function returns a NULL pointer.

---

cvRedirectError (view/add comments)

Sets a new error handler.

```
CvErrorCallback cvRedirectError(
           CvErrorCallback error_handler,
           void* userdata=NULL,
           void** prevUserdata=NULL );
```

error_handler The new error_handler

userdata Arbitrary pointer that is transparently passed to the error handler

prevUserdata Pointer to the previously assigned user data pointer

```
typedef int (CV_CDECL *CvErrorCallback)( int status, const char* func_name,
                    const char* err_msg, const char* file_name, int line );
```

The function sets a new error handler that can be one of the standard handlers or a custom handler that has a specific interface. The handler takes the same parameters as the cvError function. If the handler returns a non-zero value, the program is terminated; otherwise, it continues. The error handler may check the current error mode with cvGetErrMode to make a decision.

---

cvNulDevReport cvStdErrReport cvGuiBoxReport (view/add comments)

Provide standard error handling.

```
int cvNulDevReport( int status, const char* func_name, const char*
err_msg, const char* file_name, int line, void* userdata );

           int cvStdErrReport( int status, const char* func_name, const
char* err_msg, const char* file_name, int line, void* userdata );

           int cvGuiBoxReport( int status, const char* func_name, const
char* err_msg, const char* file_name, int line, void* userdata );
```

status The error status

func_name Name of the function where the error occured

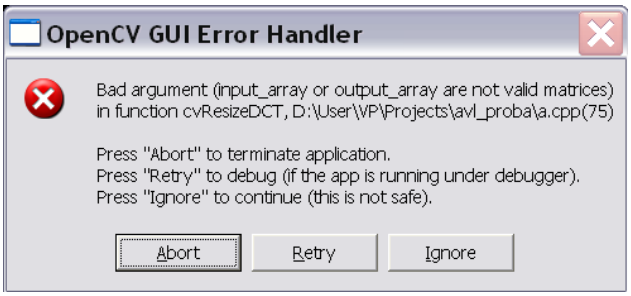
err_msg Additional information/diagnostics about the error

filename Name of the file where the error occured

line Line number, where the error occured

userdata Pointer to the user data. Ignored by the standard handlers

The functions `cvNullDevReport`, `cvStdErrReport`, and `cvGuiBoxReport` provide standard error handling. `cvGuiBoxReport` is the default error handler on Win32 systems, `cvStdErrReport` is the default on other systems. `cvGuiBoxReport` pops up a message box with the error description and suggest a few options. Below is an example message box that may be recieved with the sample code above, if one introduces an error as described in the sample.



Error Message Box

If the error handler is set to `cvStdErrReport`, the above message will be printed to standard error output and the program will be terminated or continued, depending on the current error mode.

Error Message printed to Standard Error Output (in `Leaf` mode)

```
OpenCV ERROR: Bad argument (input_array or output_array are not valid matrices)
        in function cvResizeDCT, D:\User\VP\Projects\avl\_proba\a.cpp(75)
Terminating the application...
```

---

cvAlloc (view/add comments)

Allocates a memory buffer.

```
void* cvAlloc( size_t size );
```

`size` Buffer size in bytes

The function allocates `size` bytes and returns a pointer to the allocated buffer. In the case of an error the function reports an error and returns a NULL pointer. By default, `cvAlloc` calls `icvAlloc` which itself calls `malloc`. However it is possible to assign user-defined memory allocation/deallocation functions using the cvSetMemoryManager function.

---

cvFree (view/add comments)

Deallocates a memory buffer.

```
void cvFree( void** ptr );
```

ptr Double pointer to released buffer

The function deallocates a memory buffer allocated by cvAlloc. It clears the pointer to buffer upon exit, which is why the double pointer is used. If the `*buffer` is already NULL, the function does nothing.

---

cvGetTickCount (view/add comments)

Returns the number of ticks.

```
int64 cvGetTickCount( void );
```

The function returns number of the ticks starting from some platform-dependent event (number of CPU ticks from the startup, number of milliseconds from 1970th year, etc.). The function is useful for accurate measurement of a function/user-code execution time. To convert the number of ticks to time units, use cvGetTickFrequency.

---

cvGetTickFrequency (view/add comments)

Returns the number of ticks per microsecond.

```
double cvGetTickFrequency( void );
```

The function returns the number of ticks per microsecond. Thus, the quotient of cvGetTickCount and cvGetTickFrequency will give the number of microseconds starting from the platform-dependent event.

---

cvRegisterModule (view/add comments)

Registers another module.

```
typedef struct CvPluginFuncInfo
{
    void** func_addr;
    void* default_func_addr;
    const char* func_names;
```

```
    int search_modules;
    int loaded_from;
}
CvPluginFuncInfo;

typedef struct CvModuleInfo
{
    struct CvModuleInfo* next;
    const char* name;
    const char* version;
    CvPluginFuncInfo* func_tab;
}
CvModuleInfo;
```

```
int cvRegisterModule( const CvModuleInfo* moduleInfo );
```

moduleInfo Information about the module

The function adds a module to the list of registered modules. After the module is registered, information about it can be retrieved using the cvGetModuleInfo function. Also, the registered module makes full use of optimized plugins (IPP, MKL, ...), supported by CXCORE. CXCORE itself, CV (computer vision), CVAUX (auxilary computer vision), and HIGHGUI (visualization and image/video acquisition) are examples of modules. Registration is usually done when the shared library is loaded. See `cxcore/src/cxswitcher.cpp` and `cv/src/cvswitcher.cpp` for details about how registration is done and look at `cxcore/src/cxswitcher.cpp`, `cxcore/src/_cxipp.h` on how IPP and MKL are connected to the modules.

---

cvGetModuleInfo (view/add comments)

Retrieves information about registered module(s) and plugins.

```
void cvGetModuleInfo(
           const char* moduleName,
           const char** version,
           const char** loadedAddonPlugins);
```

moduleName Name of the module of interest, or NULL, which means all the modules

version The output parameter. Information about the module(s), including version

loadedAddonPlugins The list of names and versions of the optimized plugins that CXCORE was able to find and load

The function returns information about one or all of the registered modules. The returned information is stored inside the libraries, so the user should not deallocate or modify the returned text strings.

---

cvUseOptimized (view/add comments)

Switches between optimized/non-optimized modes.

```
int cvUseOptimized( int onoff );
```

onoff Use optimized ($\neq 0$) or not ($= 0$)

The function switches between the mode, where only pure C implementations from cxcore, OpenCV, etc. are used, and the mode, where IPP and MKL functions are used if available. When `cvUseOptimized(0)` is called, all the optimized libraries are unloaded. The function may be useful for debugging, IPP and MKL upgrading on the fly, online speed comparisons, etc. It returns the number of optimized functions loaded. Note that by default, the optimized plugins are loaded, so it is not necessary to call `cvUseOptimized(1)` in the beginning of the program (actually, it will only increase the startup time).

---

cvSetMemoryManager (view/add comments)

Accesses custom/default memory managing functions.

```
typedef void* (CV_CDECL *CvAllocFunc)(size_t size, void* userdata);
typedef int (CV_CDECL *CvFreeFunc)(void* pptr, void* userdata);
```

```
void cvSetMemoryManager(
            CvAllocFunc allocFunc=NULL,
            CvFreeFunc freeFunc=NULL,
            void* userdata=NULL );
```

allocFunc Allocation function; the interface is similar to malloc, except that userdata may be used to determine the context

freeFunc Deallocation function; the interface is similar to free

userdata User data that is transparently passed to the custom functions

The function sets user-defined memory managment functions (substitutes for malloc and free) that will be called by cvAlloc, cvFree and higher-level functions (e.g., cvCreateImage). Note that the function should be called when there is data allocated using cvAlloc. Also, to avoid infinite recursive calls, it is not allowed to call cvAlloc and cvFree from the custom allocation/deallocation functions.

If the alloc_func and free_func pointers are NULL, the default memory managing functions are restored.

---

cvSetIPLAllocators (view/add comments)

Switches to IPL functions for image allocation/deallocation.

```
typedef IplImage* (CV_STDCALL* Cv_iplCreateImageHeader)
                            (int,int,int,char*,char*,int,int,int,int,int,
                            IplROI*,IplImage*,void*,IplTileInfo*);
typedef void (CV_STDCALL* Cv_iplAllocateImageData)(IplImage*,int,int);
typedef void (CV_STDCALL* Cv_iplDeallocate)(IplImage*,int);
typedef IplROI* (CV_STDCALL* Cv_iplCreateROI)(int,int,int,int,int);
typedef IplImage* (CV_STDCALL* Cv_iplCloneImage)(const IplImage*);

#define CV_TURN_ON_IPL_COMPATIBILITY()                                    \
    cvSetIPLAllocators( iplCreateImageHeader, iplAllocateImage,          \
                        iplDeallocate, iplCreateROI, iplCloneImage )
```

```
void cvSetIPLAllocators(
           Cv_iplCreateImageHeader create_header,
           Cv_iplAllocateImageData allocate_data,
           Cv_iplDeallocate deallocate,
           Cv_iplCreateROI create_roi,
           Cv_iplCloneImage clone_image );
```

create_header Pointer to iplCreateImageHeader

allocate_data Pointer to iplAllocateImage

deallocate Pointer to iplDeallocate

create_roi Pointer to iplCreateROI

clone_image Pointer to iplCloneImage

The function causes CXCORE to use IPL functions for image allocation/deallocation operations. For convenience, there is the wrapping macro `CV_TURN_ON_IPL_COMPATIBILITY`. The function is useful for applications where IPL and CXCORE/OpenCV are used together and still there are calls to `iplCreateImageHeader`, etc. The function is not necessary if IPL is called only for data processing and all the allocation/deallocation is done by CXCORE, or if all the allocation/deallocation is done by IPL and some of OpenCV functions are used to process the data.

Chapter 2

imgproc. Image Processing

2.1 Histograms

CvHistogram (view/add comments)

Multi-dimensional histogram.

```
typedef struct CvHistogram
{
    int     type;
    CvArr*  bins;
    float   thresh[CV_MAX_DIM][2]; /* for uniform histograms */
    float** thresh2; /* for non-uniform histograms */
    CvMatND mat; /* embedded matrix header for array histograms */
}
CvHistogram;
```

cvCalcBackProject (view/add comments)

Calculates the back projection.

```
void cvCalcBackProject(
           IplImage** image,
           CvArr* back_project,
           const CvHistogram* hist );
```

image Source images (though you may pass CvMat** as well)

back_project Destination back projection image of the same type as the source images

hist Histogram

The function calculates the back project of the histogram. For each tuple of pixels at the same position of all input single-channel images the function puts the value of the histogram bin, corresponding to the tuple in the destination image. In terms of statistics, the value of each output image pixel is the probability of the observed tuple given the distribution (histogram). For example, to find a red object in the picture, one may do the following:

1. Calculate a hue histogram for the red object assuming the image contains only this object. The histogram is likely to have a strong maximum, corresponding to red color.
2. Calculate back projection of a hue plane of input image where the object is searched, using the histogram. Threshold the image.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

That is the approximate algorithm of Camshift color object tracker, except for the 3rd step, instead of which CAMSHIFT algorithm is used to locate the object on the back projection given the previous object position.

---

cvCalcBackProjectPatch (view/add comments)

Locates a template within an image by using a histogram comparison.

```
void cvCalcBackProjectPatch(
           IplImage** images,
           CvArr* dst,
           CvSize patch_size,
           CvHistogram* hist,
           int method,
           double factor );
```

images Source images (though, you may pass CvMat** as well)

dst Destination image

patch_size Size of the patch slid though the source image

hist Histogram

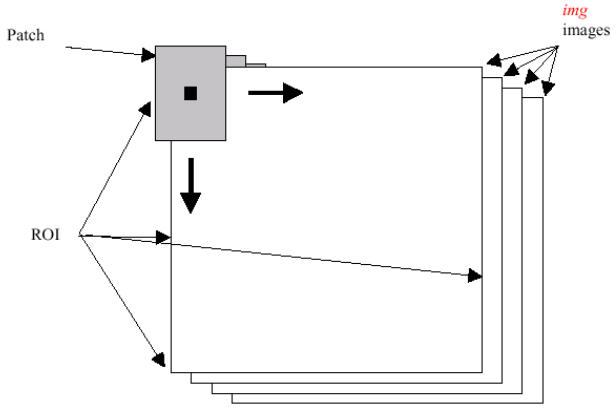
method Comparison method, passed to cvCompareHist (see description of that function)

factor Normalization factor for histograms, will affect the normalization scale of the destination image, pass 1 if unsure

The function calculates the back projection by comparing histograms of the source image patches with the given histogram. Taking measurement results from some image at each location over ROI creates an array `image`. These results might be one or more of hue, `x` derivative, `y` derivative, Laplacian filter, oriented Gabor filter, etc. Each measurement output is collected into its own separate image. The `image` image array is a collection of these measurement images. A multi-dimensional histogram `hist` is constructed by sampling from the `image` image array. The final histogram is normalized. The `hist` histogram has as many dimensions as the number of elements in `image` array.

Each new image is measured and then converted into an `image` image array over a chosen ROI. Histograms are taken from this `image` image in an area covered by a "patch" with an anchor at center as shown in the picture below. The histogram is normalized using the parameter `norm_factor` so that it may be compared with `hist`. The calculated histogram is compared to the model histogram; `hist` uses The function `cvCompareHist` with the comparison method=`method`). The resulting output is placed at the location corresponding to the patch anchor in the probability image `dst`. This process is repeated as the patch is slid over the ROI. Iterative histogram update by subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can save a lot of operations, though it is not implemented yet.

Back Project Calculation by Patches



cvCalcHist (view/add comments)

Calculates the histogram of image(s).

```
void cvCalcHist(
           IplImage** image,
           CvHistogram* hist,
           int accumulate=0,
           const CvArr* mask=NULL );
```

image Source images (though you may pass CvMat** as well)

hist Pointer to the histogram

accumulate Accumulation flag. If it is set, the histogram is not cleared in the beginning. This feature allows user to compute a single histogram from several images, or to update the histogram online

mask The operation mask, determines what pixels of the source images are counted

The function calculates the histogram of one or more single-channel images. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input images.

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv )
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* v_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* planes[] = { h_plane, s_plane };
        IplImage* hsv = cvCreateImage( cvGetSize(src), 8, 3 );
        int h_bins = 30, s_bins = 32;
        int hist_size[] = {h_bins, s_bins};
        /* hue varies from 0 (~0 deg red) to 180 (~360 deg red again) */
        float h_ranges[] = { 0, 180 };
```

```
        /* saturation varies from 0 (black-gray-white) to
           255 (pure spectrum color) */
        float s_ranges[] = { 0, 255 };
        float* ranges[] = { h_ranges, s_ranges };
        int scale = 10;
        IplImage* hist_img =
            cvCreateImage( cvSize(h_bins*scale,s_bins*scale), 8, 3 );
        CvHistogram* hist;
        float max_value = 0;
        int h, s;

        cvCvtColor( src, hsv, CV_BGR2HSV );
        cvCvtPixToPlane( hsv, h_plane, s_plane, v_plane, 0 );
        hist = cvCreateHist( 2, hist_size, CV_HIST_ARRAY, ranges, 1 );
        cvCalcHist( planes, hist, 0, 0 );
        cvGetMinMaxHistValue( hist, 0, &max_value, 0, 0 );
        cvZero( hist_img );

        for( h = 0; h < h_bins; h++ )
        {
            for( s = 0; s < s_bins; s++ )
            {
                float bin_val = cvQueryHistValue\_2D( hist, h, s );
                int intensity = cvRound(bin_val*255/max_value);
                cvRectangle( hist_img, cvPoint( h*scale, s*scale ),
                             cvPoint( (h+1)*scale - 1, (s+1)*scale - 1),
                             CV_RGB(intensity,intensity,intensity),
                             CV_FILLED );
            }
        }

        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "H-S Histogram", 1 );
        cvShowImage( "H-S Histogram", hist_img );

        cvWaitKey(0);
    }
}
```

---

cvCalcProbDensity (view/add comments)

Divides one histogram by another.

```
void cvCalcProbDensity(
           const CvHistogram* hist1,
           const CvHistogram* hist2,
           CvHistogram* dst_hist,
           double scale=255 );
```

hist1 first histogram (the divisor)

hist2 second histogram

dst_hist destination histogram

scale scale factor for the destination histogram

The function calculates the object probability density from the two histograms as:

$$\texttt{dist_hist}(I) = \begin{cases} 0 & \text{if } \texttt{hist1}(I) = 0 \\ \texttt{scale} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) > \texttt{hist1}(I) \\ \frac{\texttt{hist2}(I) \cdot \texttt{scale}}{\texttt{hist1}(I)} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) \leq \texttt{hist1}(I) \end{cases}$$

So the destination histogram bins are within less than `scale`.

---

cvClearHist (view/add comments)

Clears the histogram.

```
void cvClearHist( CvHistogram* hist );
```

hist Histogram

The function sets all of the histogram bins to 0 in the case of a dense histogram and removes all histogram bins in the case of a sparse array.

cvCompareHist (view/add comments)

Compares two dense histograms.

```
double cvCompareHist(
           const CvHistogram* hist1,
           const CvHistogram* hist2,
           int method );
```

hist1 The first dense histogram

hist2 The second dense histogram

method Comparison method, one of the following:

CV_COMP_CORREL Correlation

CV_COMP_CHISQR Chi-Square

CV_COMP_INTERSECT Intersection

CV_COMP_BHATTACHARYYA Bhattacharyya distance

The function compares two dense histograms using the specified method (H_1 denotes the first histogram, H_2 the second):

Correlation (method=CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H'_1(I) \cdot H'_2(I))}{\sqrt{\sum_I (H'_1(I)^2) \cdot \sum_I (H'_2(I)^2)}}$$

where

$$H'_k(I) = \frac{H_k(I) - 1}{N \cdot \sum_J H_k(J)}$$

where N is the number of histogram bins.

Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \sum_I \frac{\sqrt{H_1(I) \cdot H_2(I)}}{\sqrt{\sum_I H_1(I) \cdot \sum_I H_2(I)}}}$$

The function returns $d(H_1, H_2)$.

Note: the method `CV_COMP_BHATTACHARYYA` only works with normalized histograms.

To compare a sparse histogram or more general sparse configurations of weighted points, consider using the cvCalcEMD2 function.

---

cvCopyHist (view/add comments)

Copies a histogram.

```
void cvCopyHist( const CvHistogram* src, CvHistogram** dst );
```

src Source histogram

dst Pointer to destination histogram

The function makes a copy of the histogram. If the second histogram pointer `*dst` is NULL, a new histogram of the same size as `src` is created. Otherwise, both histograms must have equal types and sizes. Then the function copies the source histogram's bin values to the destination histogram and sets the same bin value ranges as in `src`.

---

cvCreateHist (view/add comments)

Creates a histogram.

```
CvHistogram* cvCreateHist(
            int dims,
            int* sizes,
            int type,
            float** ranges=NULL,
            int uniform=1 );
```

dims Number of histogram dimensions

sizes Array of the histogram dimension sizes

type Histogram representation format: CV_HIST_ARRAY means that the histogram data is represented as a multi-dimensional dense array CvMatND; CV_HIST_SPARSE means that histogram data is represented as a multi-dimensional sparse array CvSparseMat

ranges Array of ranges for the histogram bins. Its meaning depends on the uniform parameter value. The ranges are used for when the histogram is calculated or backprojected to determine which histogram bin corresponds to which value/tuple of values from the input image(s)

uniform Uniformity flag; if not 0, the histogram has evenly spaced bins and for every $0 <= i < cDims$ ranges[i] is an array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is then split into dims[i] equal parts to determine the i-th input tuple value ranges for every histogram bin. And if uniform=0, then i-th element of ranges array contains dims[i]+1 elements: $\mathtt{lower}_0, \mathtt{upper}_0, \mathtt{lower}_1, \mathtt{upper}_1 = \mathtt{lower}_2, ...\mathtt{upper}_{dims[i]-1}$ where $\mathtt{lower}_j$ and $\mathtt{upper}_j$ are lower and upper boundaries of i-th input tuple value for j-th bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin are not counted by cvCalcHist and filled with 0 by cvCalcBackProject

The function creates a histogram of the specified size and returns a pointer to the created histogram. If the array ranges is 0, the histogram bin ranges must be specified later via the function cvSetHistBinRanges. Though cvCalcHist and cvCalcBackProject may process 8-bit images without setting bin ranges, they assume thy are equally spaced in 0 to 255 bins.

---

cvGetHistValue*D (view/add comments)

Returns a pointer to the histogram bin.

```
float cvGetHistValue_1D(hist, idx0)
float cvGetHistValue_2D(hist, idx0, idx1)
float cvGetHistValue_3D(hist, idx0, idx1, idx2)
float cvGetHistValue_nD(hist, idx)
```

hist Histogram

idx0, idx1, idx2, idx3 Indices of the bin

idx Array of indices

```
#define cvGetHistValue_1D( hist, idx0 )
    ((float*)(cvPtr1D( (hist)->bins, (idx0), 0 ))
#define cvGetHistValue_2D( hist, idx0, idx1 )
    ((float*)(cvPtr2D( (hist)->bins, (idx0), (idx1), 0 )))
#define cvGetHistValue_3D( hist, idx0, idx1, idx2 )
    ((float*)(cvPtr3D( (hist)->bins, (idx0), (idx1), (idx2), 0 )))
#define cvGetHistValue_nD( hist, idx )
    ((float*)(cvPtrND( (hist)->bins, (idx), 0 )))
```

The macros `GetHistValue` return a pointer to the specified bin of the 1D, 2D, 3D or N-D histogram. In the case of a sparse histogram the function creates a new bin and sets it to 0, unless it exists already.

---

cvGetMinMaxHistValue (view/add comments)

Finds the minimum and maximum histogram bins.

```
void cvGetMinMaxHistValue(
           const CvHistogram* hist,
           float* min_value,
           float* max_value,
           int* min_idx=NULL,
           int* max_idx=NULL );
```

hist Histogram

min_value Pointer to the minimum value of the histogram

max_value Pointer to the maximum value of the histogram

min_idx Pointer to the array of coordinates for the minimum

max_idx Pointer to the array of coordinates for the maximum

The function finds the minimum and maximum histogram bins and their positions. All of output arguments are optional. Among several extremas with the same value the ones with the minimum index (in lexicographical order) are returned. In the case of several maximums or minimums, the earliest in lexicographical order (extrema locations) is returned.

---

cvMakeHistHeaderForArray (view/add comments)

Makes a histogram out of an array.

```
CvHistogram* cvMakeHistHeaderForArray(
            int dims,
            int* sizes,
            CvHistogram* hist,
            float* data,
            float** ranges=NULL,
            int uniform=1 );
```

dims Number of histogram dimensions

sizes Array of the histogram dimension sizes

hist The histogram header initialized by the function

data Array that will be used to store histogram bins

ranges Histogram bin ranges, see cvCreateHist

uniform Uniformity flag, see cvCreateHist

The function initializes the histogram, whose header and bins are allocated by th user. cvReleaseHist does not need to be called afterwards. Only dense histograms can be initialized this way. The function returns `hist`.

cvNormalizeHist (view/add comments)

Normalizes the histogram.

```
void cvNormalizeHist( CvHistogram* hist, double factor );
```

hist Pointer to the histogram

factor Normalization factor

The function normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor`.

cvQueryHistValue*D (view/add comments)

Queries the value of the histogram bin.

```
float QueryHistValue_1D(CvHistogram hist, int idx0)
```

hist Histogram

idx0, idx1, idx2, idx3 Indices of the bin

idx Array of indices

```
#define cvQueryHistValue\_1D( hist, idx0 ) \
    cvGetReal1D( (hist)->bins, (idx0) )
#define cvQueryHistValue\_2D( hist, idx0, idx1 ) \
    cvGetReal2D( (hist)->bins, (idx0), (idx1) )
#define cvQueryHistValue\_3D( hist, idx0, idx1, idx2 ) \
    cvGetReal3D( (hist)->bins, (idx0), (idx1), (idx2) )
#define cvQueryHistValue\_nD( hist, idx ) \
    cvGetRealND( (hist)->bins, (idx) )
```

The macros return the value of the specified bin of the 1D, 2D, 3D or N-D histogram. In the case of a sparse histogram the function returns 0, if the bin is not present in the histogram no new bin is created.

cvReleaseHist (view/add comments)

Releases the histogram.

```
void cvReleaseHist( CvHistogram** hist );
```

hist Double pointer to the released histogram

The function releases the histogram (header and the data). The pointer to the histogram is cleared by the function. If `*hist` pointer is already `NULL`, the function does nothing.

cvSetHistBinRanges (view/add comments)

Sets the bounds of the histogram bins.

```
void cvSetHistBinRanges(
           CvHistogram* hist,
           float** ranges,
           int uniform=1 );
```

hist Histogram

ranges Array of bin ranges arrays, see cvCreateHist

uniform Uniformity flag, see cvCreateHist

The function is a stand-alone function for setting bin ranges in the histogram. For a more detailed description of the parameters `ranges` and `uniform` see the cvCalcHist function, that can initialize the ranges as well. Ranges for the histogram bins must be set before the histogram is calculated or the backproject of the histogram is calculated.

cvThreshHist (view/add comments)

Thresholds the histogram.

```
void cvThreshHist( CvHistogram* hist, double threshold );
```

hist Pointer to the histogram

threshold Threshold level

The function clears histogram bins that are below the specified threshold.

2.2 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as cv's), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y). It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc.

IplConvKernel (view/add comments)

An IplConvKernel is a rectangular convolution kernel, created by function CreateStructuringElementEx .

---

cvCopyMakeBorder (view/add comments)

Copies an image and makes a border around it.

```
void cvCopyMakeBorder(
           const CvArr* src,
           CvArr* dst,
           CvPoint offset,
           int bordertype,
           CvScalar value=cvScalarAll(0) );
```

src The source image

dst The destination image

offset Coordinates of the top-left corner (or bottom-left in the case of images with bottom-left origin) of the destination image rectangle where the source image (or its ROI) is copied. Size of the rectanlge matches the source image size/ROI size

bordertype Type of the border to create around the copied source image rectangle; types include:

IPL_BORDER_CONSTANT border is filled with the fixed value, passed as last parameter of the function.

IPL_BORDER_REPLICATE the pixels from the top and bottom rows, the left-most and right-most columns are replicated to fill the border.

(The other two border types from IPL, `IPL_BORDER_REFLECT` and `IPL_BORDER_WRAP`, are currently unsupported)

value Value of the border pixels if `bordertype` is `IPL_BORDER_CONSTANT`

The function copies the source 2D array into the interior of the destination array and makes a border of the specified type around the copied area. The function is useful when one needs to emulate border type that is different from the one embedded into a specific algorithm implementation. For example, morphological functions, as well as most of other filtering functions in OpenCV, internally use replication border type, while the user may need a zero border or a border, filled with 1's or 255's.

---

cvCreateStructuringElementEx (view/add comments)

Creates a structuring element.

```
IplConvKernel* cvCreateStructuringElementEx(
            int cols,
            int rows,
            int anchorX,
            int anchorY,
            int shape,
            int* values=NULL );
```

cols Number of columns in the structuring element

rows Number of rows in the structuring element

anchorX Relative horizontal offset of the anchor point

anchorY Relative vertical offset of the anchor point

shape Shape of the structuring element; may have the following values:

CV_SHAPE_RECT a rectangular element

CV_SHAPE_CROSS a cross-shaped element

CV_SHAPE_ELLIPSE an elliptic element

CV_SHAPE_CUSTOM a user-defined element. In this case the parameter `values` specifies the mask, that is, which neighbors of the pixel must be considered

values Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non-zero values indicate points that belong to the element. If the pointer is `NULL`, then all values are considered non-zero, that is, the element is of a rectangular shape. This parameter is considered only if the shape is `CV_SHAPE_CUSTOM`

The function CreateStructuringElementEx allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the morphological operations.

---

cvDilate (view/add comments)

Dilates an image by using a specific structuring element.

```
void cvDilate(
           const CvArr* src,
           CvArr* dst,
           IplConvKernel* element=NULL,
           int iterations=1 );
```

src Source image

dst Destination image

element Structuring element used for dilation. If it is `NULL`, a 3×3 rectangular structuring element is used

iterations Number of times dilation is applied

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\max_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. For color images, each channel is processed independently.

---

cvErode (view/add comments)

Erodes an image by using a specific structuring element.

```
void cvErode(
           const CvArr* src,
           CvArr* dst,
           IplConvKernel* element=NULL,
           int iterations=1);
```

src Source image

dst Destination image

element Structuring element used for erosion. If it is `NULL`, a 3×3 rectangular structuring element is used

iterations Number of times erosion is applied

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\min_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. For color images, each channel is processed independently.

---

cvFilter2D (view/add comments)

Convolves an image with the kernel.

```
void cvFilter2D(
           const CvArr* src,
           CvArr* dst,
           const CvMat* kernel,
           CvPoint anchor=cvPoint(-1,-1));
```

src The source image

dst The destination image

kernel Convolution kernel, a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cvSplit and process them individually

anchor The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor shoud lie within the kernel. The special default value (-1,-1) means that it is at the kernel center

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values from the nearest pixels that are inside the image.

cvLaplace (view/add comments)

Calculates the Laplacian of an image.

```
void cvLaplace(
            const CvArr* src,
            CvArr* dst,
            int apertureSize=3);
```

src Source image

dst Destination image

apertureSize Aperture size (it has the same meaning as cvSobel)

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst}(x,y) = \frac{d^2\texttt{src}}{dx^2} + \frac{d^2\texttt{src}}{dy^2}$$

Setting `apertureSize` = 1 gives the fastest variant that is equal to convolving the image with the following kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Similar to the cvSobel function, no scaling is done and the same combinations of input and output formats are supported.

cvMorphologyEx (view/add comments)

Performs advanced morphological transformations.

```
void cvMorphologyEx(
           const CvArr* src,
           CvArr* dst,
           CvArr* temp,
           IplConvKernel* element,
           int operation,
           int iterations=1 );
```

src Source image

dst Destination image

temp Temporary image, required in some cases

element Structuring element

operation Type of morphological operation, one of the following:

- **CV_MOP_OPEN** opening
- **CV_MOP_CLOSE** closing
- **CV_MOP_GRADIENT** morphological gradient
- **CV_MOP_TOPHAT** "top hat"
- **CV_MOP_BLACKHAT** "black hat"

iterations Number of times erosion and dilation are applied

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$dst = open(src, element) = dilate(erode(src, element), element)$$

Closing:

$$dst = close(src, element) = erode(dilate(src, element), element)$$

Morphological gradient:

$$dst = morph_grad(src, element) = dilate(src, element) - erode(src, element)$$

"Top hat":

$$dst = tophat(src, element) = src - open(src, element)$$

"Black hat":

$$dst = blackhat(src, element) = close(src, element) - src$$

The temporary image `temp` is required for a morphological gradient and, in the case of in-place operation, for "top hat" and "black hat".

---

cvPyrDown (view/add comments)

Downsamples an image.

```
void cvPyrDown(
            const CvArr* src,
            CvArr* dst,
            int filter=CV_GAUSSIAN_5x5 );
```

src The source image

dst The destination image, should have a half as large width and height than the source

filter Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported

The function performs the downsampling step of the Gaussian pyramid decomposition. First it convolves the source image with the specified filter and then downsamples the image by rejecting even rows and columns.

---

cvReleaseStructuringElement (view/add comments)

Deletes a structuring element.

```
void cvReleaseStructuringElement( IplConvKernel** element );
```

element Pointer to the deleted structuring element

The function releases the structure `IplConvKernel` that is no longer needed. If `*element` is `NULL`, the function has no effect.

---

cvSmooth (view/add comments)

Smooths the image in one of several ways.

```
void cvSmooth(
           const CvArr* src,
           CvArr* dst,
           int smoothtype=CV_GAUSSIAN,
           int param1=3,
           int param2=0,
           double param3=0,
           double param4=0);
```

src The source image

dst The destination image

smoothtype Type of the smoothing:

- **CV_BLUR_NO_SCALE** linear convolution with `param1` × `param2` box kernel (all 1's). If you want to smooth different pixels with different-size box kernels, you can use the integral image that is computed using cvIntegral
- **CV_BLUR** linear convolution with `param1` × `param2` box kernel (all 1's) with subsequent scaling by $1/(\texttt{param1} \cdot \texttt{param2})$
- **CV_GAUSSIAN** linear convolution with a `param1` × `param2` Gaussian kernel
- **CV_MEDIAN** median filter with a `param1` × `param1` square aperture
- **CV_BILATERAL** bilateral filter with a `param1`×`param1` square aperture, color sigma=`param3` and spatial sigma=`param4`. If `param1=0`, the aperture square side is set to `cvRound(param4*1.5)*` Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1 The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)

param2 The second parameter of the smoothing operation, the aperture height. Ignored by `CV_MEDIAN` and `CV_BILATERAL` methods. In the case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1`. Otherwise it must be a positive odd number.

param3 In the case of a Gaussian parameter this parameter may specify Gaussian σ (standard deviation). If it is zero, it is calculated from the kernel size:

$$\sigma = 0.3(n/2 - 1) + 0.8 \quad \text{where} \quad n = \begin{array}{l} \texttt{param1} \text{ for horizontal kernel} \\ \texttt{param2} \text{ for vertical kernel} \end{array}$$

Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function smooths an image using one of several methods. Every of the methods has some features and restrictions listed below

Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to cvSobel and cvLaplace) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

---

cvSobel (view/add comments)

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator.

```
void cvSobel(
            const CvArr* src,
            CvArr* dst,
            int xorder,
            int yorder,
            int apertureSize=3 );
```

src Source image of type CvArr*

dst Destination image

xorder Order of the derivative x

yorder Order of the derivative y

apertureSize Size of the extended Sobel kernel, must be 1, 3, 5 or 7

In all cases except 1, an `apertureSize` $\times$ `apertureSize` separable kernel will be used to calculate the derivative. For `apertureSize` $= 1$ a 3×1 or 1×3 a kernel is used (Gaussian smoothing is not done). There is also the special value `CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\texttt{dst}(x,y) = \frac{d^{xorder+yorder}\texttt{src}}{dx^{xorder} \cdot dy^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `apertureSize` = 3) or (`xorder` = 0, `yorder` = 1, `apertureSize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and the second one corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

or a kernel of:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix}$$

depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger numbers (in absolute values) than the source image does. To avoid overflow, the function requires a 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using the cvConvertScale or the cvConvertScaleAbs function. Besides 8-bit images the function can process 32-bit floating-point images. Both the source and the destination must be single-channel images of equal size or equal ROI size.

2.3 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x, y) = \texttt{src}(f_x(x, y), f_y(x, y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic cvRemap and to the simplest and the fastest cvResize, need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x, y) one of $f_x(x, y)$ or $f_y(x, y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method `BORDER_TRANSPARENT`, which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x, y)$ and $f_y(x, y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x, y), f_y(x, y))$ and then the value of the polynomial at $(f_x(x, y), f_y(x, y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see cvResize.

cvGetRotationMatrix2D (view/add comments)

Calculates the affine matrix of 2d rotation.

```
CvMat* cv2DRotationMatrix(
           CvPoint2D32f center,
           double angle,
           double scale,
           CvMat* mapMatrix );
```

center Center of the rotation in the source image

angle The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)

scale Isotropic scale factor

mapMatrix Pointer to the destination 2×3 matrix

The function `cv2DRotationMatrix` calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} - (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot cos(\texttt{angle}), \beta = \texttt{scale} \cdot sin(\texttt{angle})$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

cvGetAffineTransform (view/add comments)

Calculates the affine transform from 3 corresponding points.

```
CvMat* cvGetAffineTransform(
           const CvPoint2D32f* src,
           const CvPoint2D32f* dst,
           CvMat* mapMatrix );
```

src Coordinates of 3 triangle vertices in the source image

dst Coordinates of the 3 corresponding triangle vertices in the destination image

mapMatrix Pointer to the destination 2×3 matrix

The function cvGetAffineTransform calculates the matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

---

cvGetPerspectiveTransform (view/add comments)

Calculates the perspective transform from 4 corresponding points.

```
CvMat* cvGetPerspectiveTransform(
           const CvPoint2D32f* src,
           const CvPoint2D32f* dst,
           CvMat* mapMatrix );
```

src Coordinates of 4 quadrangle vertices in the source image

dst Coordinates of the 4 corresponding quadrangle vertices in the destination image

mapMatrix Pointer to the destination 3×3 matrix

The function `cvGetPerspectiveTransform` calculates a matrix of perspective transforms such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

cvGetQuadrangleSubPix (view/add comments)

Retrieves the pixel quadrangle from an image with sub-pixel accuracy.

```
void cvGetQuadrangleSubPix(
           const CvArr* src,
           CvArr* dst,
           const CvMat* mapMatrix );
```

src Source image

dst Extracted quadrangle

mapMatrix The transformation 2×3 matrix $[A|b]$ (see the discussion)

The function `cvGetQuadrangleSubPix` extracts pixels from `src` at sub-pixel accuracy and stores them to `dst` as follows:

$$dst(x, y) = src(A_{11}x' + A_{12}y' + b_1, A_{21}x' + A_{22}y' + b_2)$$

where

$$x' = x - \frac{(width(dst) - 1)}{2}, y' = y - \frac{(height(dst) - 1)}{2}$$

and

$$\texttt{mapMatrix} = \begin{bmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{bmatrix}$$

The values of pixels at non-integer coordinates are retrieved using bilinear interpolation. When the function needs pixels outside of the image, it uses replication border mode to reconstruct the values. Every channel of multiple-channel images is processed independently.

cvGetRectSubPix (view/add comments)

Retrieves the pixel rectangle from an image with sub-pixel accuracy.

```
void cvGetRectSubPix(
           const CvArr* src,
           CvArr* dst,
           CvPoint2D32f center );
```

src Source image

dst Extracted rectangle

center Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image

The function `cvGetRectSubPix` extracts pixels from `src`:

$$dst(x,y) = src(x + \texttt{center.x} - (width(\texttt{dst}) - 1) * 0.5, y + \texttt{center.y} - (height(\texttt{dst}) - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

---

cvLogPolar (view/add comments)

Remaps an image to log-polar space.

```
void cvLogPolar(
           const CvArr* src,
           CvArr* dst,
           CvPoint2D32f center,
           double M,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
```

src Source image

dst Destination image

center The transformation center; where the output precision is maximal

M Magnitude scale parameter. See below

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero

CV_WARP_INVERSE_MAP See below

The function `cvLogPolar` transforms the source image using the following transformation: Forward transformation (`CV_WARP_INVERSE_MAP` is not set):

$$dst(\phi, \rho) = src(x, y)$$

Inverse transformation (`CV_WARP_INVERSE_MAP` is set):

$$dst(x, y) = src(\phi, \rho)$$

where

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

```
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv)
{
    IplImage* src;

    if( argc == 2 && (src=cvLoadImage(argv[1],1) != 0 )
    {
        IplImage* dst = cvCreateImage( cvSize(256,256), 8, 3 );
        IplImage* src2 = cvCreateImage( cvGetSize(src), 8, 3 );
        cvLogPolar( src, dst, cvPoint2D32f(src->width/2,src->height/2), 40,
        CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
        cvLogPolar( dst, src2, cvPoint2D32f(src->width/2,src->height/2), 40,
        CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS+CV_WARP_INVERSE_MAP );
        cvNamedWindow( "log-polar", 1 );
        cvShowImage( "log-polar", dst );
        cvNamedWindow( "inverse log-polar", 1 );
        cvShowImage( "inverse log-polar", src2 );
```

```
        cvWaitKey();
    }
    return 0;
}
```

And this is what the program displays when `opencv/samples/c/fruits.jpg` is passed to it

---

cvRemap (view/add comments)

Applies a generic geometrical transformation to the image.

```
void cvRemap(
            const CvArr* src,
            CvArr* dst,
            const CvArr* mapx,
            const CvArr* mapy,
            int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
            CvScalar fillval=cvScalarAll(0) );
```

src Source image

dst Destination image

mapx The map of x-coordinates (CV_32FC1 image)

mapy The map of y-coordinates (CV_32FC1 image)

flags A combination of interpolation method and the following optional flag(s):

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`

fillval A value used to fill outliers

The function `cvRemap` transforms the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(\texttt{mapx}(x,y), \texttt{mapy}(x,y))$$

Similar to other geometrical transformations, some interpolation method (specified by user) is used to extract pixels with non-integer coordinates. Note that the function can not operate in-place.

---

cvResize (view/add comments)

Resizes an image.

```
void cvResize(
           const CvArr* src,
           CvArr* dst,
           int interpolation=CV_INTER_LINEAR );
```

src Source image

dst Destination image

interpolation Interpolation method:

CV_INTER_NN nearest-neigbor interpolation

CV_INTER_LINEAR bilinear interpolation (used by default)

CV_INTER_AREA resampling using pixel area relation. It is the preferred method for image decimation that gives moire-free results. In terms of zooming it is similar to the `CV_INTER_NN` method

CV_INTER_CUBIC bicubic interpolation

The function `cvResize` resizes an image `src` so that it fits exactly into `dst`. If ROI is set, the function considers the ROI as supported.

cvWarpAffine (view/add comments)

Applies an affine transformation to an image.

```
void cvWarpAffine(
           const CvArr* src,
           CvArr* dst,
           const CvMat* mapMatrix,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
           CvScalar fillval=cvScalarAll(0) );
```

src Source image

dst Destination image

mapMatrix 2×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

- **CV_WARP_FILL_OUTLIERS** fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`
- **CV_WARP_INVERSE_MAP** indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpAffine` transforms the source image using the specified matrix:

$$dst(x', y') = src(x, y)$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV_WARP_INVERSE_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

The function is similar to cvGetQuadrangleSubPix but they are not exactly the same. cvWarpAffine requires input and output image have the same data type, has larger overhead (so it is not quite suitable for small images) and can leave part of destination image unchanged. While cvGetQuadrangleSubPix may extract quadrangles from 8-bit images into floating-point buffer, has smaller overhead and always changes the whole destination image content. Note that the function can not operate in-place.

To transform a sparse set of points, use the cvTransform function from cxcore.

---

cvWarpPerspective (view/add comments)

Applies a perspective transformation to an image.

```
void cvWarpPerspective(
           const CvArr* src,
           CvArr* dst,
           const CvMat* mapMatrix,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
           CvScalar fillval=cvScalarAll(0) );
```

src Source image

dst Destination image

mapMatrix 3×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

- **CV_WARP_FILL_OUTLIERS** fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`
- **CV_WARP_INVERSE_MAP** indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpPerspective` transforms the source image using the specified matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV_WARP_INVERSE_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

Note that the function can not operate in-place. For a sparse set of points use the cvPerspectiveTransform function from CxCore.

2.4 Miscellaneous Image Transformations

cvAdaptiveThreshold (view/add comments)

Applies an adaptive threshold to an array.

```
void cvAdaptiveThreshold(
           const CvArr* src,
           CvArr* dst,
           double maxValue,
           int adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C,
           int thresholdType=CV_THRESH_BINARY,
           int blockSize=3,
           double param1=5 );
```

src Source image

dst Destination image

maxValue Maximum value that is used with CV_THRESH_BINARY and CV_THRESH_BINARY_INV

adaptive_method Adaptive thresholding algorithm to use: CV_ADAPTIVE_THRESH_MEAN_C or CV_ADAPTIVE_THRESH_GAUSSIAN_C (see the discussion)

thresholdType Thresholding type; must be one of

CV_THRESH_BINARY xxx

CV_THRESH_BINARY_INV xxx

blockSize The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

param1 The method-dependent parameter. For the methods CV_ADAPTIVE_THRESH_MEAN_C and CV_ADAPTIVE_THRESH_GAUSSIAN_C it is a constant subtracted from the mean or weighted mean (see the discussion), though it may be negative

The function transforms a grayscale image to a binary image according to the formulas:

CV_THRESH_BINARY

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

For the method CV_ADAPTIVE_THRESH_MEAN_C it is the mean of a blockSize × blockSize pixel neighborhood, minus param1.

For the method CV_ADAPTIVE_THRESH_GAUSSIAN_C it is the weighted sum (gaussian) of a blockSize × blockSize pixel neighborhood, minus param1.

---

cvCvtColor (view/add comments)

Converts an image from one color space to another.

```
void cvCvtColor(
           const CvArr* src,
           CvArr* dst,
           int code );
```

src The source 8-bit (8u), 16-bit (16u) or single-precision floating-point (32f) image

dst The destination image of the same data type as the source. The number of channels may be different

code Color conversion operation that can be specifed using CV_ *src_color_space* 2 *dst_color_space* constants (see below)

The function converts the input image from one color space to another. The function ignores the `colorModel` and `channelSeq` fields of the `IplImage` header, so the source image color space should be specified correctly (including order of the channels in the case of RGB space. For example, BGR means 24-bit format with $B_0, G_0, R_0, B_1, G_1, R_1, ...$ layout whereas RGB means 24-format with $R_0, G_0, B_0, R_1, G_1, B_1, ...$ layout).

The conventional range for R,G,B channel values is:

- 0 to 255 for 8-bit images
- 0 to 65535 for 16-bit images and
- 0 to 1 for floating-point images.

Of course, in the case of linear transformations the range can be specific, but in order to get correct results in the case of non-linear transformations, the input image should be scaled.

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

  ```
  cvCvtColor(src ,bwsrc, CV_RGB2GRAY)
  ```

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB`):

  $$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

  $$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB ↔ YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb`, `CV_RGB2YCrCb`, `CV_YCrCb2BGR`, `CV_YCrCb2RGB`)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

  where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

  Y, Cr and Cb cover the whole value range.

- RGB ↔ HSV (`CV_BGR2HSV`, `CV_RGB2HSV`, `CV_HSV2BGR`, `CV_HSV2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V = R \\ 120 + 60(B - R)/S & \text{if } V = G \\ 240 + 60(R - G)/S & \text{if } V = B \end{cases}$$

  if $H < 0$ then $H \leftarrow H + 360$

  On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

  The values are then converted to the destination data type:

  8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

  16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

  32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R,G,B)$$

$$V_{min} \leftarrow min(R,G,B)$$

$$L \leftarrow \frac{V_{max} + V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max}-V_{min}}{V_{max}+V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max}-V_{min}}{2-(V_{max}+V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V_{max} = R \\ 120 + 60(B-R)/S & \text{if } V_{max} = G \\ 240 + 60(R-G)/S & \text{if } V_{max} = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$ On output $0 \leq L \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t <= 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \leq L \leq 100$, $-127 \leq a \leq 127$, $-127 \leq b \leq 127$

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow L * 255/100, a \leftarrow a + 128, b \leftarrow b + 128$$

16-bit images currently not supported

32-bit images L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y <= 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow 255/100L, u \leftarrow 255/354(u + 134), v \leftarrow 255/256(v + 140)$$

16-bit images currently not supported

32-bit images L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Ford98 at the Charles Poynton site.

- Bayer $\rightarrow$ RGB (`CV_BayerBG2BGR`, `CV_BayerGB2BGR`, `CV_BayerRG2BGR`, `CV_BayerGR2BGR`, `CV_BayerBG2RGB`, `CV_BayerGB2RGB`, `CV_BayerRG2RGB`, `CV_BayerGR2RGB`) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{matrix} R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants `CV_Bayer` C_1C_2 `2BGR` and `CV_Bayer` C_1C_2 `2RGB` indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

---

cvDistTransform (view/add comments)

Calculates the distance to the closest zero pixel for all non-zero pixels of the source image.

```
void cvDistTransform(
            const CvArr* src,
            CvArr* dst,
            int distance_type=CV_DIST_L2,
            int mask_size=3,
            const float* mask=NULL,
            CvArr* labels=NULL );
```

src 8-bit, single-channel (binary) source image

dst Output image with calculated distances (32-bit floating-point, single-channel)

distance_type Type of distance; can be CV_DIST_L1, CV_DIST_L2, CV_DIST_C or CV_DIST_USER

mask_size Size of the distance transform mask; can be 3 or 5. in the case of CV_DIST_L1 or CV_DIST_C the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 yet it is faster

mask User-defined mask in the case of a user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in the case ofa 3×3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight's move cost) in the case of a 5×5 mask

labels The optional output 2d array of integer type labels, the same size as src and dst

The function calculates the approximated distance from every binary image pixel to the nearest zero pixel. For zero pixels the function sets the zero distance, for others it finds the shortest path consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (that is denoted as a), all the diagonal shifts must have the same cost (denoted b), and all knight's moves must have the same cost (denoted c). For CV_DIST_C and CV_DIST_L1 types the distance is calculated precisely, whereas for CV_DIST_L2 (Euclidian distance) the distance can be calculated only with some relative error (a 5×5 mask gives more accurate results), OpenCV uses the values suggested in [**?**]:

CV_DIST_C	(3×3)	a = 1, b = 1
CV_DIST_L1	(3×3)	a = 1, b = 2
CV_DIST_L2	(3×3)	a=0.955, b=1.3693
CV_DIST_L2	(5×5)	a=1, b=1.4, c=2.1969

And below are samples of the distance field (black (0) pixel is in the middle of white square) in the case of a user-defined distance:

User-defined 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1		1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

User-defined 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1		1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

Typically, for a fast, coarse distance estimation `CV_DIST_L2`, a 3×3 mask is used, and for a more accurate distance estimation `CV_DIST_L2`, a 5×5 mask is used.

When the output parameter `labels` is not `NULL`, for every non-zero pixel the function also finds the nearest connected component consisting of zero pixels. The connected components themselves are found as contours in the beginning of the function.

In this mode the processing time is still O(N), where N is the number of pixels. Thus, the function provides a very fast way to compute approximate Voronoi diagram for the binary image.

CvConnectedComp (view/add comments)

```
typedef struct CvConnectedComp
{
    double area;    /* area of the segmented component */
    CvScalar value; /* average color of the connected component */
    CvRect rect;    /* ROI of the segmented component */
    CvSeq* contour; /* optional component boundary
                       (the contour might have child contours corresponding to the holes) */
} CvConnectedComp;
```

cvFloodFill (view/add comments)

Fills a connected component with the given color.

```
void cvFloodFill(
            CvArr* image,
            CvPoint seed_point,
            CvScalar new_val,
            CvScalar lo_diff=cvScalarAll(0),
            CvScalar up_diff=cvScalarAll(0),
            CvConnectedComp* comp=NULL,
            int flags=4,
            CvArr* mask=NULL );
```

image Input 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the CV_FLOODFILL_MASK_ONLY flag is set (see below)

seed_point The starting point

new_val New value of the repainted domain pixels

lo_diff Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

up_diff Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

comp Pointer to the structure that the function fills with the information about the repainted domain. Note that the function does not fill comp->contour field. The boundary of the filled component can be retrieved from the output mask image using cvFindContours

flags The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:

CV_FLOODFILL_FIXED_RANGE if set, the difference between the current pixel and seed pixel is considered, otherwise the difference between neighbor pixels is considered (the range is floating)

CV_FLOODFILL_MASK_ONLY if set, the function does not fill the image (new_val is ignored), but fills the mask (that must be non-NULL in this case)

mask Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller than image. If not NULL, the function uses and updates the mask, so the user takes responsibility of initializing the mask content. Floodfilling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. It is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap. **Note**: because the mask is larger than the filled image, a pixel in mask that corresponds to (x, y) pixel in image will have coordinates $(x + 1, y + 1)$

The function fills a connected component starting from the seed point with the specified color. The connectivity is determined by the closeness of pixel values. The pixel at (x, y) is considered to belong to the repainted domain if:

grayscale image, floating range

$$src(x', y') - \texttt{lo_diff} <= src(x, y) <= src(x', y') + \texttt{up_diff}$$

grayscale image, fixed range

$$src(seed.x, seed.y) - \texttt{lo_diff} <= src(x, y) <= src(seed.x, seed.y) + \texttt{up_diff}$$

color image, floating range

$$src(x', y')_r - \texttt{lo_diff}_r <= src(x, y)_r <= src(x', y')_r + \texttt{up_diff}_r$$

$$src(x', y')_g - \texttt{lo_diff}_g <= src(x, y)_g <= src(x', y')_g + \texttt{up_diff}_g$$

$$src(x', y')_b - \texttt{lo_diff}_b <= src(x, y)_b <= src(x', y')_b + \texttt{up_diff}_b$$

color image, fixed range

$$src(seed.x, seed.y)_r - \texttt{lo_diff}_r <= src(x, y)_r <= src(seed.x, seed.y)_r + \texttt{up_diff}_r$$

$$src(seed.x, seed.y)_g - \texttt{lo_diff}_g <= src(x, y)_g <= src(seed.x, seed.y)_g + \texttt{up_diff}_g$$

$$src(seed.x, seed.y)_b - \texttt{lo_diff}_b <= src(x, y)_b <= src(seed.x, seed.y)_b + \texttt{up_diff}_b$$

where $src(x', y')$ is the value of one of pixel neighbors. That is, to be added to the connected component, a pixel's color/brightness should be close enough to the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

---

cvInpaint (view/add comments)

Inpaints the selected region in the image.

```
void cvInpaint(
           const CvArr* src,
           const CvArr* mask,
           CvArr* dst,
           double inpaintRadius,
           int flags);
```

src The input 8-bit 1-channel or 3-channel image.

mask The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.

dst The output image of the same format and the same size as input.

inpaintRadius The radius of circlular neighborhood of each point inpainted that is considered by the algorithm.

flags The inpainting method, one of the following:

CV_INPAINT_NS Navier-Stokes based method.

CV_INPAINT_TELEA The method by Alexandru Telea [**?**]

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video.

---

cvIntegral (view/add comments)

Calculates the integral of an image.

```
void cvIntegral(
           const CvArr* image,
           CvArr* sum,
           CvArr* sqsum=NULL,
           CvArr* tiltedSum=NULL );
```

image The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

sum The integral image, $(W + 1) \times (H + 1)$, 32-bit integer or double precision floating-point (64f)

sqsum The integral image for squared pixel values, $(W + 1) \times (H + 1)$, double precision floating-point (64f)

tiltedSum The integral for the image rotated by 45 degrees, $(W + 1) \times (H + 1)$, the same data type as `sum`

The function calculates one or more integral images for the source image as following:

$$\texttt{sum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)$$

$$\texttt{sqsum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)^2$$

$$\texttt{tiltedSum}(X,Y) = \sum_{y<Y,abs(x-X+1)\leq Y-y-1} \texttt{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum_{x_1<=x<x_2,\, y_1<=y<y_2} = \texttt{sum}(x_2,y_2) - \texttt{sum}(x_1,y_2) - \texttt{sum}(x_2,y_1) + \texttt{sum}(x_1,x_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

---

cvPyrMeanShiftFiltering (view/add comments)

Does meanshift image segmentation

```
void cvPyrMeanShiftFiltering(
           const CvArr* src,
           CvArr* dst,
           double sp,
           double sr,
           int max_level=1,
           CvTermCriteria termcrit=
           cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,5,1));
```

src The source 8-bit, 3-channel image.

dst The destination image of the same format and the same size as the source.

sp The spatial window radius.

sr The color window radius.

max_level Maximum level of the pyramid for the segmentation.

termcrit Termination criteria: when to stop meanshift iterations.

The function implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X,Y) neighborhood in the joint space-color hyperspace is considered:

$$(x,y): X-\texttt{sp} \le x \le X+\texttt{sp}, Y-\texttt{sp} \le y \le Y+\texttt{sp}, ||(R,G,B)-(r,g,b)|| \le \texttt{sr}$$

where `(R,G,B)` and `(r,g,b)` are the vectors of color components at `(X,Y)` and `(x,y)`, respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value `(X',Y')` and average color vector `(R',G',B')` are found and they act as the neighborhood center on the next iteration:

(X,Y) (X',Y'), (R,G,B) (R',G',B').

After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration):

$I(X,Y) <- (R*,G*,B*)$

Then `max_level` > 0 , the gaussian pyramid of `max_level` $+ 1$ levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much ($>$ `sr`) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when `max_level` $== 0$).

---

cvPyrSegmentation (view/add comments)

Implements image segmentation by pyramids.

```
void cvPyrSegmentation(
           IplImage* src,
           IplImage* dst,
           CvMemStorage* storage,
           CvSeq** comp,
           int level,
           double threshold1,
           double threshold2 );
```

src The source image

dst The destination image

storage Storage; stores the resulting sequence of connected components

comp Pointer to the output sequence of the segmented components

level Maximum level of the pyramid for the segmentation

threshold1 Error threshold for establishing the links

threshold2 Error threshold for the segments clustering

The function implements image segmentation by pyramids. The pyramid builds up to the level `level`. The links between any pixel `a` on level `i` and its candidate father pixel `b` on the adjacent level are established if $p(c(a), c(b)) < threshold1$. After the connected components are defined, they are joined into several clusters. Any two segments A and B belong to the same cluster, if $p(c(A), c(B)) < threshold2$. If the input image has only one channel, then $p(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then

$$p(c^1, c^2) = 0.30(c_r^1 - c_r^2) + 0.59(c_g^1 - c_g^2) + 0.11(c_b^1 - c_b^2).$$

There may be more than one connected component per a cluster. The images `src` and `dst` should be 8-bit single-channel or 3-channel images or equal size.

---

cvThreshold (view/add comments)

Applies a fixed-level threshold to array elements.

```
double cvThreshold(
           const CvArr* src,
           CvArr* dst,
           double threshold,
           double maxValue,
           int thresholdType );
```

src Source array (single-channel, 8-bit or 32-bit floating point)

dst Destination array; must be either the same type as `src` or 8-bit

threshold Threshold value

maxValue Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types

thresholdType Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (cvCmpS could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by `thresholdType`:

CV_THRESH_BINARY

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxValue} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

CV_THRESH_TRUNC

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

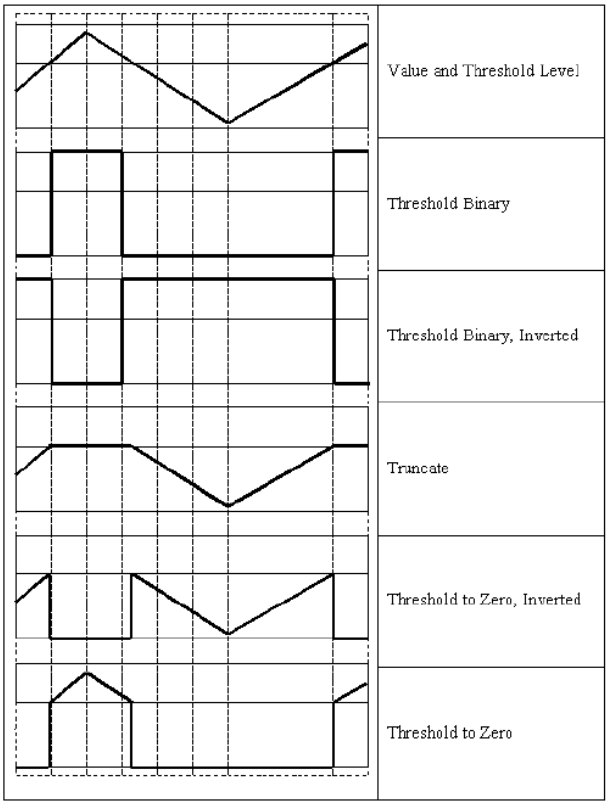
CV_THRESH_TOZERO

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_TOZERO_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Also, the special value `CV_THRESH_OTSU` may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified `thresh`. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.



2.5 Structural Analysis and Shape Descriptors

---

cvApproxChains (view/add comments)

Approximates Freeman chain(s) with a polygonal curve.

```
CvSeq* cvApproxChains(
           CvSeq* src_seq,
           CvMemStorage* storage,
           int method=CV_CHAIN_APPROX_SIMPLE,
```

```
            double parameter=0,
            int minimal_perimeter=0,
            int recursive=0 );
```

src_seq Pointer to the chain that can refer to other chains

storage Storage location for the resulting polylines

method Approximation method (see the description of the function cvFindContours)

parameter Method parameter (not used now)

minimal_perimeter Approximates only those contours whose perimeters are not less than `minimal_perimeter`. Other chains are removed from the resulting structure

recursive If not 0, the function approximates all chains that access can be obtained to from `src_seq` by using the `h_next` or `v_next links`. If 0, the single chain is approximated

This is a stand-alone approximation routine. The function `cvApproxChains` works exactly in the same way as cvFindContours with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via the `v_next` or `h_next` fields of the returned structure.

---

cvApproxPoly (view/add comments)

Approximates polygonal curve(s) with the specified precision.

```
CvSeq* cvApproxPoly(
            const void* src_seq,
            int header_size,
            CvMemStorage* storage,
            int method,
            double parameter,
            int parameter2=0 );
```

src_seq Sequence of an array of points

header_size Header size of the approximated curve[s]

storage Container for the approximated contours. If it is NULL, the input sequences' storage is used

method Approximation method; only CV_POLY_APPROX_DP is supported, that corresponds to the Douglas-Peucker algorithm

parameter Method-specific parameter; in the case of CV_POLY_APPROX_DP it is a desired approximation accuracy

parameter2 If case if src_seq is a sequence, the parameter determines whether the single sequence should be approximated or all sequences on the same level or below src_seq (see cvFindContours for description of hierarchical contour structures). If src_seq is an array CvMat* of points, the parameter specifies whether the curve is closed (parameter2!=0) or not (parameter2 =0)

The function approximates one or more curves and returns the approximation result[s]. In the case of multiple curves, the resultant tree will have the same structure as the input one (1:1 correspondence).

---

cvArcLength (view/add comments)

Calculates the contour perimeter or the curve length.

```
double cvArcLength(
           const void* curve,
           CvSlice slice=CV_WHOLE_SEQ,
           int isClosed=-1 );
```

curve Sequence or array of the curve points

slice Starting and ending points of the curve, by default, the whole curve length is calculated

isClosed Indicates whether the curve is closed or not. There are 3 cases:

- isClosed $= 0$ the curve is assumed to be unclosed.
- isClosed > 0 the curve is assumed to be closed.

- `isClosed` < 0 if curve is sequence, the flag `CV_SEQ_FLAG_CLOSED` of `((CvSeq*)curve)->flags` is checked to determine if the curve is closed or not, otherwise (curve is represented by array (CvMat*) of points) it is assumed to be unclosed.

The function calculates the length or curve as the sum of lengths of segments between subsequent points

---

cvBoundingRect (view/add comments)

Calculates the up-right bounding rectangle of a point set.

```
CvRect cvBoundingRect( CvArr* points, int update=0 );
```

points 2D point set, either a sequence or vector (`CvMat`) of points

update The update flag. See below.

The function returns the up-right bounding rectangle for a 2d point set. Here is the list of possible combination of the flag values and type of `points`:

update	points	action
0	`CvContour`	the bounding rectangle is not calculated, but it is taken from `rect` field of the contour header.
1	`CvContour`	the bounding rectangle is calculated and written to `rect` field of the contour header.
0	`CvSeq` or `CvMat`	the bounding rectangle is calculated and returned.
1	`CvSeq` or `CvMat`	runtime error is raised.

---

cvBoxPoints (view/add comments)

Finds the box vertices.

```
void cvBoxPoints(
           CvBox2D box,
           CvPoint2D32f pt[4] );
```

box Box

points Array of vertices

The function calculates the vertices of the input 2d box.
Here is the function code:

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    float a = (float)cos(box.angle)*0.5f;
    float b = (float)sin(box.angle)*0.5f;

    pt[0].x = box.center.x - a*box.size.height - b*box.size.width;
    pt[0].y = box.center.y + b*box.size.height - a*box.size.width;
    pt[1].x = box.center.x + a*box.size.height - b*box.size.width;
    pt[1].y = box.center.y - b*box.size.height - a*box.size.width;
    pt[2].x = 2*box.center.x - pt[0].x;
    pt[2].y = 2*box.center.y - pt[0].y;
    pt[3].x = 2*box.center.x - pt[1].x;
    pt[3].y = 2*box.center.y - pt[1].y;
}
```

---

cvCalcPGH (view/add comments)

Calculates a pair-wise geometrical histogram for a contour.

```
void cvCalcPGH( const CvSeq* contour, CvHistogram* hist );
```

contour Input contour. Currently, only integer point coordinates are allowed

hist Calculated histogram; must be two-dimensional

The function calculates a 2D pair-wise geometrical histogram (PGH), described in cvlivarinen97 for the contour. The algorithm considers every pair of contour edges. The angle between the edges and the minimum/maximum distances are determined for every pair. To do this each of the edges in turn is taken as the base, while the function loops through all the other edges. When the base edge and any other edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented (that is, the histogram

is transposed relatively to the cvlivarninen97 definition). The histogram can be used for contour matching.

---

cvCalcEMD2 (view/add comments)

Computes the "minimal work" distance between two weighted point configurations.

```
float cvCalcEMD2(
            const CvArr* signature1,
            const CvArr* signature2,
            int distance_type,
            CvDistanceFunction distance_func=NULL,
            const CvArr* cost_matrix=NULL,
            CvArr* flow=NULL,
            float* lower_bound=NULL,
            void* userdata=NULL );
```

signature1 First signature, a `size1` $\times$ `dims` $+ 1$ floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used

signature2 Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different, in this case an extra "dummy" point is added to either `signature1` or `signature2`

distance_type Metrics used; `CV_DIST_L1`, `CV_DIST_L2`, and `CV_DIST_C` stand for one of the standard metrics; `CV_DIST_USER` means that a user-defined function `distance_func` or pre-calculated `cost_matrix` is used

distance_func The user-supplied distance function. It takes coordinates of two points and returns the distance between the points `typedef float (*CvDistanceFunction)(const float* f1, const float* f2, void* userdata);`

cost_matrix The user-defined `size1` $\times$ `size2` cost matrix. At least one of `cost_matrix` and `distance_func` must be NULL. Also, if a cost matrix is used, lower boundary (see below) can not be calculated, because it needs a metric function

flow The resultant `size1` $\times$ `size2` flow matrix: $\texttt{flow}_{i,j}$ is a flow from i th point of `signature1` to j th point of `signature2`

lower_bound Optional input/output parameter: lower boundary of distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or if the signatures consist of weights only (i.e. the signature matrices have a single column). The user **must** initialize *lower_bound. If the calculated distance between mass centers is greater or equal to *lower_bound (it means that the signatures are far enough) the function does not calculate EMD. In any case *lower_bound is set to the calculated distance between mass centers on return. Thus, if user wants to calculate both distance between mass centers and EMD, *lower_bound should be set to 0

userdata Pointer to optional data that is passed into the user-defined distance function

The function computes the earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the applications described in cvRubnerSept98 is multi-dimensional histogram comparison for image retrieval. EMD is a a transportation problem that is solved using some modification of a simplex algorithm, thus the complexity is exponential in the worst case, though, on average it is much faster. In the case of a real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

---

cvCheckContourConvexity (view/add comments)

Tests contour convexity.

```
int cvCheckContourConvexity( const CvArr* contour );
```

contour Tested contour (sequence or array of points)

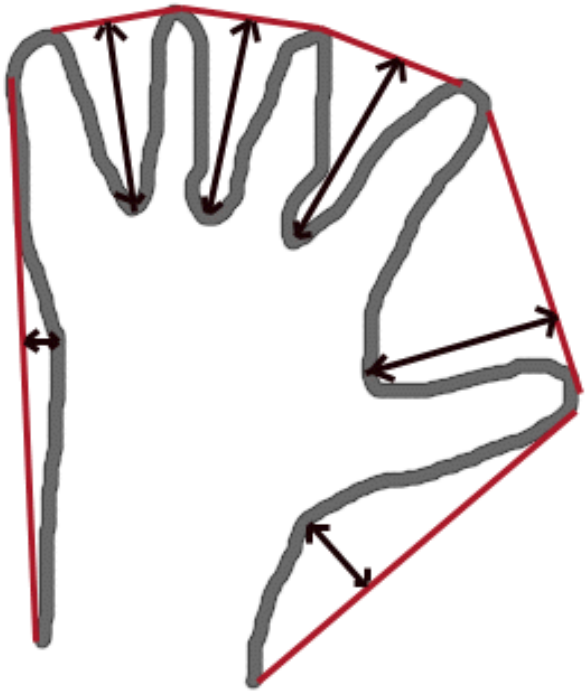
The function tests whether the input contour is convex or not. The contour must be simple, without self-intersections.

CvConvexityDefect (view/add comments)

Structure describing a single contour convexity defect.

```
typedef struct CvConvexityDefect
{
```

```
    CvPoint* start; /* point of the contour where the defect begins */
    CvPoint* end; /* point of the contour where the defect ends */
    CvPoint* depth_point; /* the farthest from the convex hull point within the defect */
    float depth; /* distance between the farthest point and the convex hull */
} CvConvexityDefect;
```



cvContourArea (view/add comments)

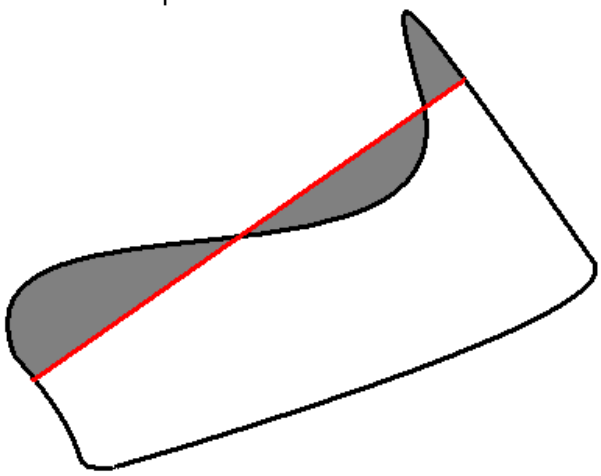
Calculates the area of a whole contour or a contour section.

```
double cvContourArea(
           const CvArr* contour,
           CvSlice slice=CV_WHOLE_SEQ );
```

contour Contour (sequence or array of vertices)

slice Starting and ending points of the contour section of interest, by default, the area of the whole contour is calculated

The function calculates the area of a whole contour or a contour section. In the latter case the total area bounded by the contour arc and the chord connecting the 2 selected points is calculated as shown on the picture below:



Orientation of the contour affects the area sign, thus the function may return a *negative* result. Use the `fabs()` function from C runtime to get the absolute value of the area.

---

cvContourFromContourTree (view/add comments)

Restores a contour from the tree.

```
CvSeq* cvContourFromContourTree(
           const CvContourTree* tree,
           CvMemStorage* storage,
           CvTermCriteria criteria );
```

tree Contour tree

storage Container for the reconstructed contour

criteria Criteria, where to stop reconstruction

The function restores the contour from its binary tree representation. The parameter `criteria` determines the accuracy and/or the number of tree levels used for reconstruction, so it is possible to build an approximated contour. The function returns the reconstructed contour.

cvConvexHull2 (view/add comments)

Finds the convex hull of a point set.

```
CvSeq* cvConvexHull2(
           const CvArr* input,
           void* storage=NULL,
           int orientation=CV_CLOCKWISE,
           int return_points=0 );
```

points Sequence or array of 2D points with 32-bit integer or floating-point coordinates

storage The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is an array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified as to truncate the array down to the hull size. If `storage` is NULL then the convex hull will be stored in the same storage as the input sequence

orientation Desired orientation of convex hull: `CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`

return_points If non-zero, the points themselves will be stored in the hull instead of indices if `storage` is an array, or pointers if `storage` is memory storage

The function finds the convex hull of a 2D point set using Sklansky's algorithm. If `storage` is memory storage, the function creates a sequence containing the hull points or pointers to them, depending on `return_points` value and returns the sequence on output. If `storage` is a CvMat, the function returns NULL.

Example. Building convex hull for a sequence or array of points

```
#include "cv.h"
#include "highgui.h"
#include <stdlib.h>

#define ARRAY  0 /* switch between array/sequence method by replacing 0<=>1 */

void main( int argc, char** argv )
{
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    cvNamedWindow( "hull", 1 );
```

```
#if !ARRAY
        CvMemStorage* storage = cvCreateMemStorage();
#endif

    for(;;)
    {
        int i, count = rand()%100 + 1, hullcount;
        CvPoint pt0;
#if !ARRAY
        CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2,
                                    sizeof(CvContour),
                                    sizeof(CvPoint),
                                    storage );
        CvSeq* hull;

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            cvSeqPush( ptseq, &pt0 );
        }
        hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
        hullcount = hull->total;
#else
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
        int* hull = (int*)malloc( count * sizeof(hull[0]));
        CvMat point_mat = cvMat( 1, count, CV_32SC2, points );
        CvMat hull_mat = cvMat( 1, count, CV_32SC1, hull );

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            points[i] = pt0;
        }
        cvConvexHull2( &point_mat, &hull_mat, CV_CLOCKWISE, 0 );
        hullcount = hull_mat.cols;
#endif
        cvZero( img );
        for( i = 0; i < count; i++ )
        {
#if !ARRAY
            pt0 = *CV_GET_SEQ_ELEM( CvPoint, ptseq, i );
#else
            pt0 = points[i];
```

```
#endif
            cvCircle( img, pt0, 2, CV_RGB( 255, 0, 0 ), CV_FILLED );
        }

#if !ARRAY
        pt0 = **CV_GET_SEQ_ELEM( CvPoint*, hull, hullcount - 1 );
#else
        pt0 = points[hull[hullcount-1]];
#endif

        for( i = 0; i < hullcount; i++ )
        {
#if !ARRAY
            CvPoint pt = **CV_GET_SEQ_ELEM( CvPoint*, hull, i );
#else
            CvPoint pt = points[hull[i]];
#endif
            cvLine( img, pt0, pt, CV_RGB( 0, 255, 0 ));
            pt0 = pt;
        }

        cvShowImage( "hull", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;

#if !ARRAY
        cvClearMemStorage( storage );
#else
        free( points );
        free( hull );
#endif
    }
}
```

---

cvConvexityDefects (view/add comments)

Finds the convexity defects of a contour.

```
CvSeq* cvConvexityDefects(
```

```
            const CvArr* contour,
            const CvArr* convexhull,
            CvMemStorage* storage=NULL );
```

contour Input contour

convexhull Convex hull obtained using cvConvexHull2 that should contain pointers or indices to the contour points, not the hull points themselves (the `return_points` parameter in cvConvexHull2 should be 0)

storage Container for the output sequence of convexity defects. If it is NULL, the contour or hull (in that order) storage is used

The function finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect structures.

---

cvCreateContourTree (view/add comments)

Creates a hierarchical representation of a contour.

```
CvContourTree* cvCreateContourTree(
            const CvSeq* contour,
            CvMemStorage* storage,
            double threshold );
```

contour Input contour

storage Container for output tree

threshold Approximation accuracy

The function creates a binary tree representation for the input `contour` and returns the pointer to its root. If the parameter `threshold` is less than or equal to 0, the function creates a full binary tree representation. If the threshold is greater than 0, the function creates a representation with the precision `threshold`: if the vertices with the interceptive area of its base line are less than `threshold`, the tree should not be built any further. The function returns the created tree.

cvEndFindContours (view/add comments)

Finishes the scanning process.

```
CvSeq* cvEndFindContours(
           CvContourScanner* scanner );
```

scanner Pointer to the contour scanner

The function finishes the scanning process and returns a pointer to the first contour on the highest level.

cvFindContours (view/add comments)

Finds the contours in a binary image.

```
int cvFindContours(
           CvArr* image,
           CvMemStorage* storage,
           CvSeq** first_contour,
           int header_size=sizeof(CvContour),
           int mode=CV_RETR_LIST,
           int method=CV_CHAIN_APPROX_SIMPLE,
           CvPoint offset=cvPoint(0,0) );
```

image The source, an 8-bit single channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary`. To get such a binary image from grayscale, one may use cvThreshold, cvAdaptiveThreshold or cvCanny. The function modifies the source image's content

storage Container of the retrieved contours

first_contour Output parameter, will contain the pointer to the first outer contour

header_size Size of the sequence header, $\geq$ `sizeof(CvChain)` if `method` = `CV_CHAIN_CODE`, and $\geq$ `sizeof(CvContour)` otherwise

mode Retrieval mode

CV_RETR_EXTERNAL retrives only the extreme outer contours

CV_RETR_LIST retrieves all of the contours and puts them in the list

CV_RETR_CCOMP retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes

CV_RETR_TREE retrieves all of the contours and reconstructs the full hierarchy of nested contours

method Approximation method (for all the modes, except `CV_LINK_RUNS`, which uses built-in approximation)

CV_CHAIN_CODE outputs contours in the Freeman chain code. All other methods output polygons (sequences of vertices)

CV_CHAIN_APPROX_NONE translates all of the points from the chain code into points

CV_CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and diagonal segments and leaves only their end points

CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS applies one of the flavors of the Teh-Chin chain approximation algorithm.

CV_LINK_RUNS uses a completely different contour retrieval algorithm by linking horizontal segments of 1's. Only the `CV_RETR_LIST` retrieval mode can be used with this method.

offset Offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context

The function retrieves contours from the binary image using the algorithm [**?**]. The contours are a useful tool for shape analysis and object detection and recognition.

The function retrieves contours from the binary image and returns the number of retrieved contours. The pointer `first_contour` is filled by the function. It will contain a pointer to the first outermost contour or `NULL` if no contours are detected (if the image is completely black). Other contours may be reached from `first_contour` using the `h_next` and `v_next` links. The sample in the cvDrawContours discussion shows how to use contours for connected component detection. Contours can be also used for shape analysis and object recognition - see `squares.c` in the OpenCV sample directory.

Note: the source `image` is modified by this function.

cvFindNextContour (view/add comments)

Finds the next contour in the image.

```
CvSeq* cvFindNextContour(
            CvContourScanner scanner );
```

scanner Contour scanner initialized by cvStartFindContours

The function locates and retrieves the next contour in the image and returns a pointer to it. The function returns NULL if there are no more contours.

cvFitEllipse2 (view/add comments)

Fits an ellipse around a set of 2D points.

```
CvBox2D cvFitEllipse2(
            const CvArr* points );
```

points Sequence or array of points

The function calculates the ellipse that fits best (in least-squares sense) around a set of 2D points. The meaning of the returned structure fields is similar to those in cvEllipse except that `size` stores the full lengths of the ellipse axises, not half-lengths.

cvFitLine (view/add comments)

Fits a line to a 2D or 3D point set.

```
void cvFitLine(
            const CvArr* points,
            int dist_type,
            double param,
```

```
            double reps,
            double aeps,
            float* line );
```

points Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates

dist_type The distance used for fitting (see the discussion)

param Numerical parameter (`C`) for some types of distances, if 0 then some optimal value is chosen

reps Sufficient accuracy for the radius (distance between the coordinate origin and the line). 0.01 is a good default value.

aeps Sufficient accuracy for the angle. 0.01 is a good default value.

line The output line parameters. In the case of a 2d fitting, it is an array of 4 floats `(vx, vy, x0, y0)` where `(vx, vy)` is a normalized vector collinear to the line and `(x0, y0)` is some point on the line. in the case of a 3D fitting it is an array of 6 floats `(vx, vy, vz, x0, y0, z0)` where `(vx, vy, vz)` is a normalized vector collinear to the line and `(x0, y0, z0)` is some point on the line

The function fits a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i th point and the line and $\rho(r)$ is a distance function, one of:

dist_type=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

dist_type=CV_DIST_L1

$$\rho(r) = r$$

dist_type=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

dist_type=CV_DIST_FAIR

$$\rho\left(r\right) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right) \quad \text{where} \quad C = 1.3998$$

dist_type=CV_DIST_WELSCH

$$\rho\left(r\right) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

dist_type=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

---

cvGetCentralMoment (view/add comments)

Retrieves the central moment from the moment state structure.

```
double cvGetCentralMoment(
           CvMoments* moments,
           int x_order,
           int y_order );
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, x_order >= 0

y_order y order of the retrieved moment, y_order >= 0 and x_order + y_order <= 3

The function retrieves the central moment, which in the case of image moments is defined as:

$$\mu_{x_order, y_order} = \sum_{x,y}(I(x,y) \cdot (x - x_c)^{x_order} \cdot (y - y_c)^{y_order})$$

where x_c, y_c are the coordinates of the gravity center:

$$x_c = \frac{M_{10}}{M_{00}}, y_c = \frac{M_{01}}{M_{00}}$$

---

cvGetHuMoments (view/add comments)

Calculates the seven Hu invariants.

```
void cvGetHuMoments( const CvMoments* moments,CvHuMoments* hu );
```

moments The input moments, computed with cvMoments

hu The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment, that are defined as:

$$
\begin{aligned}
hu_1 &= \eta_{20} + \eta_{02} \\
hu_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
hu_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
hu_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
hu_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
hu_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
hu_7 &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}
$$

where η_{ji} denote the normalized central moments.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

---

cvGetNormalizedCentralMoment (view/add comments)

Retrieves the normalized central moment from the moment state structure.

```
double cvGetNormalizedCentralMoment(
           CvMoments* moments,
           int x_order,
           int y_order );
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, $\texttt{x_order} >= 0$

y_order y order of the retrieved moment, y_order $>= 0$ and x_order + y_order $<= 3$

The function retrieves the normalized central moment:

$$\eta_{x_order,\, y_order} = \frac{\mu_{x_order,\, y_order}}{M_{00}^{(y_order+x_order)/2+1}}$$

---

cvGetSpatialMoment (view/add comments)

Retrieves the spatial moment from the moment state structure.

```
double cvGetSpatialMoment(
           CvMoments* moments,
           int x_order,
           int y_order );
```

moments The moment state, calculated by cvMoments

x_order x order of the retrieved moment, x_order $>= 0$

y_order y order of the retrieved moment, y_order $>= 0$ and x_order + y_order $<= 3$

The function retrieves the spatial moment, which in the case of image moments is defined as:

$$M_{x_order,\, y_order} = \sum_{x,y} (I(x,y) \cdot x^{x_order} \cdot y^{y_order})$$

where $I(x,y)$ is the intensity of the pixel (x,y).

---

cvMatchContourTrees (view/add comments)

Compares two contours using their tree representations.

```
double cvMatchContourTrees(
           const CvContourTree* tree1,
           const CvContourTree* tree2,
           int method,
           double threshold );
```

tree1 First contour tree

tree2 Second contour tree

method Similarity measure, only CV_CONTOUR_TREES_MATCH_I1 is supported

threshold Similarity threshold

The function calculates the value of the matching measure for two contour trees. The similarity measure is calculated level by level from the binary tree roots. If at a certain level the difference between contours becomes less than threshold, the reconstruction process is interrupted and the current difference is returned.

---

cvMatchShapes (view/add comments)

Compares two shapes.

```
double cvMatchShapes(
           const void* object1,
           const void* object2,
           int method,
           double parameter=0 );
```

object1 First contour or grayscale image

object2 Second contour or grayscale image

method Comparison method; CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3

parameter Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu moments (see cvGetHuMoments) (A is object1, B is object2):

method=CV_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1\ldots7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1...7} \left| m_i^A - m_i^B \right|$$

method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1...7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

where

$$m_i^A = sign(h_i^A) \cdot \log h_i^A m_i^B = sign(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

---

cvMinAreaRect2 (view/add comments)

Finds the circumscribed rectangle of minimal area for a given 2D point set.

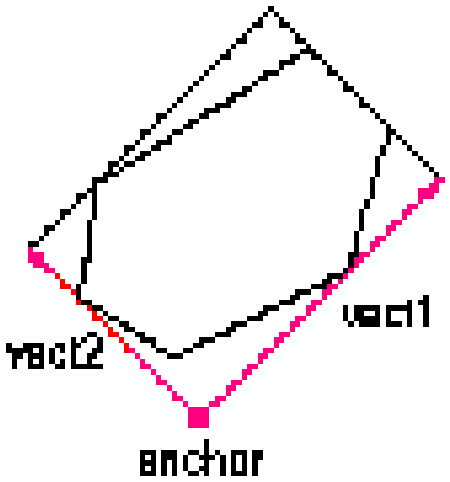
```
CvBox2D cvMinAreaRect2(
           const CvArr* points,
           CvMemStorage* storage=NULL );
```

points Sequence or array of points

storage Optional temporary memory storage

The function finds a circumscribed rectangle of the minimal area for a 2D point set by building a convex hull for the set and applying the rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour



cvMinEnclosingCircle (view/add comments)

Finds the circumscribed circle of minimal area for a given 2D point set.

```
int cvMinEnclosingCircle(
           const CvArr* points,
           CvPoint2D32f* center,
           float* radius );
```

points Sequence or array of 2D points

center Output parameter; the center of the enclosing circle

radius Output parameter; the radius of the enclosing circle

The function finds the minimal circumscribed circle for a 2D point set using an iterative algorithm. It returns nonzero if the resultant circle contains all the input points and zero otherwise (i.e. the algorithm failed).

cvMoments (view/add comments)

Calculates all of the moments up to the third order of a polygon or rasterized shape.

```
void cvMoments(
           const CvArr* arr,
           CvMoments* moments,
           int binary=0 );
```

arr Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points or a vector of points)

moments Pointer to returned moment's state structure

binary (For images only) If the flag is non-zero, all of the zero pixel values are treated as zeroes, and all of the others are treated as 1's

The function calculates spatial and central moments up to the third order and writes them to `moments`. The moments may then be used then to calculate the gravity center of the shape, its area, main axises and various shape characeteristics including 7 Hu invariants.

cvPointPolygonTest (view/add comments)

Point in contour test.

```
double cvPointPolygonTest(
           const CvArr* contour,
           CvPoint2D32f pt,
           int measure_dist );
```

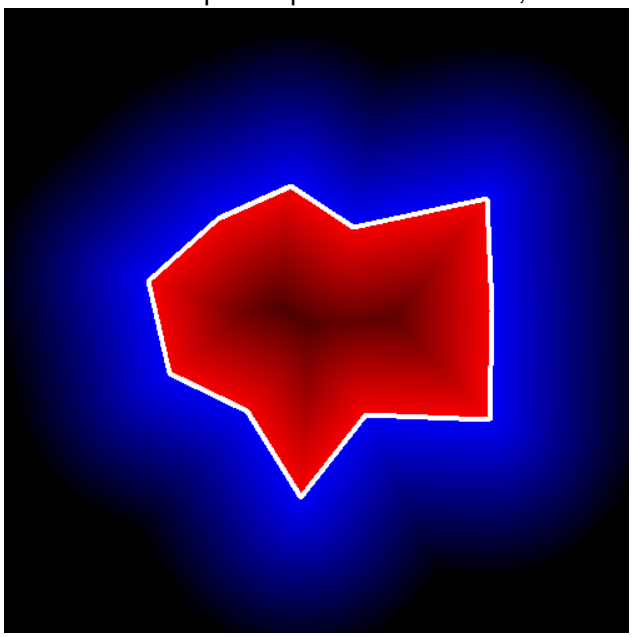
contour Input contour

pt The point tested against the contour

measure_dist If it is non-zero, the function estimates the distance from the point to the nearest contour edge

The function determines whether the point is inside a contour, outside, or lies on an edge (or coinsides with a vertex). It returns positive, negative or zero value, correspondingly. When measure_dist $= 0$, the return value is +1, -1 and 0, respectively. When measure_dist $\neq 0$, it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



---

cvPointSeqFromMat (view/add comments)

Initializes a point sequence header from a point vector.

```
CvSeq* cvPointSeqFromMat(
           int seq_kind,
           const CvArr* mat,
           CvContour* contour_header,
           CvSeqBlock* block );
```

seq_kind Type of the point sequence: point set (0), a curve (CV_SEQ_KIND_CURVE), closed curve (CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED) etc.

mat Input matrix. It should be a continuous, 1-dimensional vector of points, that is, it should have type CV_32SC2 or CV_32FC2

contour_header Contour header, initialized by the function

block Sequence block header, initialized by the function

The function initializes a sequence header to create a "virtual" sequence in which elements reside in the specified matrix. No data is copied. The initialized sequence header may be passed to any function that takes a point sequence on input. No extra elements can be added to the sequence, but some may be removed. The function is a specialized variant of cvMakeSeqHeaderForArray and uses the latter internally. It returns a pointer to the initialized contour header. Note that the bounding rectangle (field `rect` of `CvContour` strucuture) is not initialized by the function. If you need one, use cvBoundingRect.

Here is a simple usage example.

```
CvContour header;
CvSeqBlock block;
CvMat* vector = cvCreateMat( 1, 3, CV_32SC2 );

CV_MAT_ELEM( *vector, CvPoint, 0, 0 ) = cvPoint(100,100);
CV_MAT_ELEM( *vector, CvPoint, 0, 1 ) = cvPoint(100,200);
CV_MAT_ELEM( *vector, CvPoint, 0, 2 ) = cvPoint(200,100);

IplImage* img = cvCreateImage( cvSize(300,300), 8, 3 );
cvZero(img);

cvDrawContours( img,
    cvPointSeqFromMat(CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED,
                      vector,
                      &header,
                      &block),
                CV_RGB(255,0,0),
                CV_RGB(255,0,0),
                0, 3, 8, cvPoint(0,0));
```

cvReadChainPoint (view/add comments)

Gets the next chain point.

```
CvPoint cvReadChainPoint( CvChainPtReader* reader );
```

reader Chain reader state

The function returns the current chain point and updates the reader position.

cvStartFindContours (view/add comments)

Initializes the contour scanning process.

```
CvContourScanner cvStartFindContours(
            CvArr* image,
            CvMemStorage* storage,
            int header_size=sizeof(CvContour),
            int mode=CV_RETR_LIST,
            int method=CV_CHAIN_APPROX_SIMPLE,
            CvPoint offset=cvPoint(0,
            0) );
```

image The 8-bit, single channel, binary source image

storage Container of the retrieved contours

header_size Size of the sequence header, $>= sizeof(CvChain)$ if `method` =CV_CHAIN_CODE, and $>= sizeof(CvContour)$ otherwise

mode Retrieval mode; see cvFindContours

method Approximation method. It has the same meaning in cvFindContours, but `CV_LINK_RUNS` can not be used here

offset ROI offset; see cvFindContours

The function initializes and returns a pointer to the contour scanner. The scanner is used in cvFindNextContour to retrieve the rest of the contours.

cvStartReadChainPoints (view/add comments)

Initializes the chain reader.

```
void cvStartReadChainPoints( CvChain* chain, CvChainPtReader* reader );
```

The function initializes a special reader.

cvSubstituteContour (view/add comments)

Replaces a retrieved contour.

```
void cvSubstituteContour(
           CvContourScanner scanner,
           CvSeq* new_contour );
```

scanner Contour scanner initialized by cvStartFindContours

new_contour Substituting contour

The function replaces the retrieved contour, that was returned from the preceding call of cvFindNextContour and stored inside the contour scanner state, with the user-specified contour. The contour is inserted into the resulting structure, list, two-level hierarchy, or tree, depending on the retrieval mode. If the parameter `new_contour` is `NULL`, the retrieved contour is not included in the resulting structure, nor are any of its children that might be added to this structure later.

2.6 Planar Subdivisions

CvSubdiv2D (view/add comments)

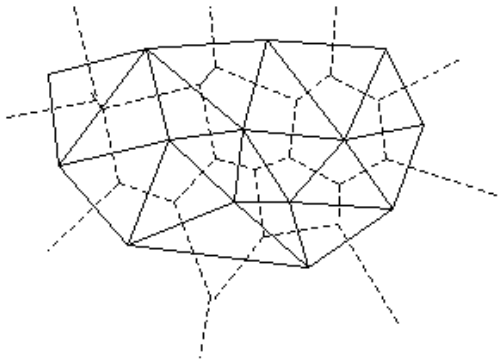
Planar subdivision.

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
CvSubdiv2D;
```

Planar subdivision is the subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on a 2d point set,

where the points are linked together and form a planar graph, which, together with a few edges connecting the exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists a dual subdivision in which facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called a virtual point below) of the dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dotted lines.



OpenCV subdivides a plane into triangles using Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case the dual subdivision is a Voronoi diagram of the input 2d point set. The subdivisions can be used for the 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) and so forth.

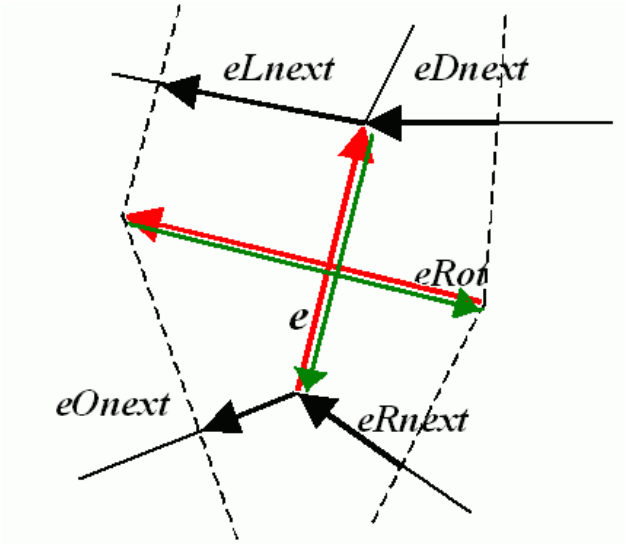
CvQuadEdge2D (view/add comments)

Quad-edge of planar subdivision.

```
/* one of edges within quad-edge, lower 2 bits is index (0..3)
   and upper bits are quad-edge pointer */
typedef long CvSubdiv2DEdge;

/* quad-edge structure fields */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
```

```
CvQuadEdge2D;
```

Quad-edge is a basic element of subdivision containing four edges (e, eRot, reversed e and reversed eRot):



CvSubdiv2DPoint (view/add comments)

Point of original or dual subdivision.

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int            flags;      \
    CvSubdiv2DEdge first;      \
    CvPoint2D32f   pt;         \
    int id;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

id This integer can be used to index auxillary data associated with each vertex of the planar subdivision

cvCalcSubdivVoronoi2D (view/add comments)

Calculates the coordinates of Voronoi diagram cells.

```
void cvCalcSubdivVoronoi2D(
            CvSubdiv2D* subdiv );
```

subdiv Delaunay subdivision, in which all the points are already added

The function calculates the coordinates of virtual points. All virtual points corresponding to some vertex of the original subdivision form (when connected together) a boundary of the Voronoi cell at that point.

cvClearSubdivVoronoi2D (view/add comments)

Removes all virtual points.

```
void cvClearSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv Delaunay subdivision

The function removes all of the virtual points. It is called internally in cvCalcSubdivVoronoi2D if the subdivision was modified after previous call to the function.

cvCreateSubdivDelaunay2D (view/add comments)

Creates an empty Delaunay triangulation.

```
CvSubdiv2D* cvCreateSubdivDelaunay2D(
            CvRect rect,
            CvMemStorage* storage );
```

rect Rectangle that includes all of the 2d points that are to be added to the subdivision

storage Container for subdivision

The function creates an empty Delaunay subdivision, where 2d points can be added using the function cvSubdivDelaunay2DInsert. All of the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

Note that the triangulation is a single large triangle that covers the given rectangle. Hence the three vertices of this triangle are outside the rectangle `rect`.

---

cvFindNearestPoint2D (view/add comments)

Finds the closest subdivision vertex to the given point.

```
CvSubdiv2DPoint* cvFindNearestPoint2D(
            CvSubdiv2D* subdiv,
            CvPoint2D32f pt );
```

subdiv Delaunay or another subdivision

pt Input point

The function is another function that locates the input point within the subdivision. It finds the subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using cvSubdiv2DLocate) is used as a starting point. The function returns a pointer to the found subdivision vertex.

---

cvSubdiv2DEdgeDst (view/add comments)

Returns the edge destination.

```
CvSubdiv2DPoint* cvSubdiv2DEdgeDst(
            CvSubdiv2DEdge edge );
```

edge Subdivision edge (not a quad-edge)

The function returns the edge destination. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using the function cvCalcSubdivVoronoi2D.

---

cvSubdiv2DGetEdge (view/add comments)

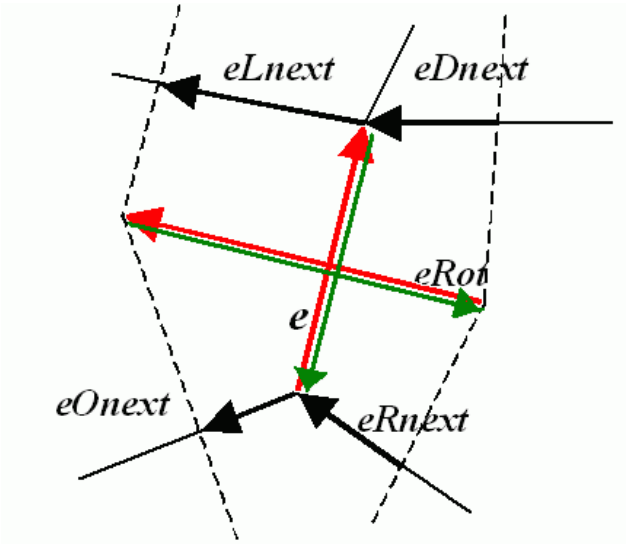
Returns one of the edges related to the given edge.

```
CvSubdiv2DEdge cvSubdiv2DGetEdge( CvSubdiv2DEdge edge, CvNextEdgeType
type );
```

edge Subdivision edge (not a quad-edge)

type Specifies which of the related edges to return, one of the following:

- **CV_NEXT_AROUND_ORG** next around the edge origin (`eOnext` on the picture below if `e` is the input edge)
- **CV_NEXT_AROUND_DST** next around the edge vertex (`eDnext`)
- **CV_PREV_AROUND_ORG** previous around the edge origin (reversed `eRnext`)
- **CV_PREV_AROUND_DST** previous around the edge destination (reversed `eLnext`)
- **CV_NEXT_AROUND_LEFT** next around the left facet (`eLnext`)
- **CV_NEXT_AROUND_RIGHT** next around the right facet (`eRnext`)
- **CV_PREV_AROUND_LEFT** previous around the left facet (reversed `eOnext`)
- **CV_PREV_AROUND_RIGHT** previous around the right facet (reversed `eDnext`)



The function returns one of the edges related to the input edge.

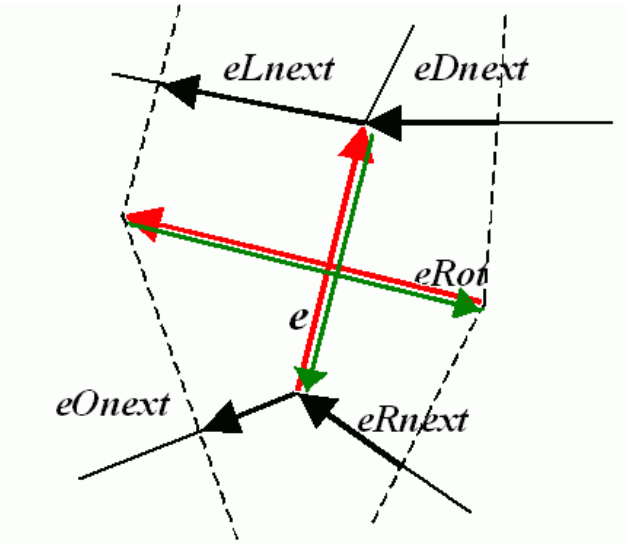
---

cvSubdiv2DNextEdge (view/add comments)

Returns next edge around the edge origin

```
CvSubdiv2DEdge cvSubdiv2DNextEdge( CvSubdiv2DEdge edge );
```

edge Subdivision edge (not a quad-edge)



The function returns the next edge around the edge origin: `eOnext` on the picture above if `e` is the input edge)

---

cvSubdiv2DLocate (view/add comments)

Returns the location of a point within a Delaunay triangulation.

```
CvSubdiv2DPointLocation cvSubdiv2DLocate(
            CvSubdiv2D* subdiv,
            CvPoint2D32f pt,
            CvSubdiv2DEdge* edge,
            CvSubdiv2DPoint** vertex=NULL );
```

subdiv Delaunay or another subdivision

pt The point to locate

edge The output edge the point falls onto or right to

vertex Optional output vertex double pointer the input point coinsides with

The function locates the input point within the subdivision. There are 5 cases:

- The point falls into some facet. The function returns `CV_PTLOC_INSIDE` and `*edge` will contain one of edges of the facet.
- The point falls onto the edge. The function returns `CV_PTLOC_ON_EDGE` and `*edge` will contain this edge.
- The point coincides with one of the subdivision vertices. The function returns `CV_PTLOC_VERTEX` and `*vertex` will contain a pointer to the vertex.
- The point is outside the subdivsion reference rectangle. The function returns `CV_PTLOC_OUTSIDE_RECT` and no pointers are filled.
- One of input arguments is invalid. A runtime error is raised or, if silent or "parent" error processing mode is selected, `CV_PTLOC_ERROR` is returnd.

---

cvSubdiv2DRotateEdge (view/add comments)

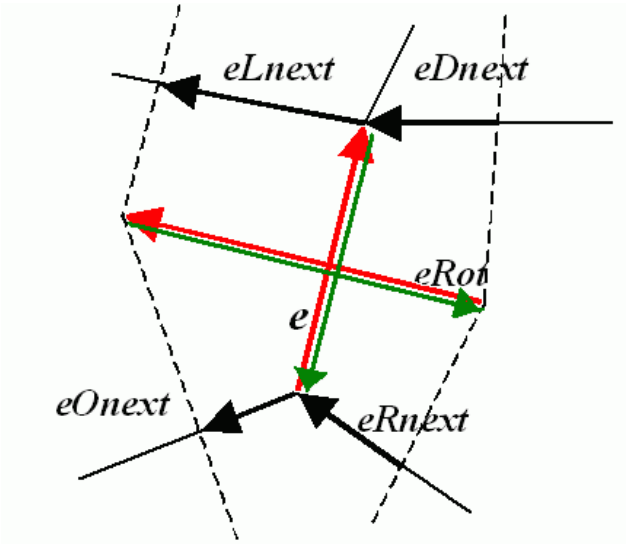
Returns another edge of the same quad-edge.

```
CvSubdiv2DEdge cvSubdiv2DRotateEdge(
            CvSubdiv2DEdge edge,
            int rotate );
```

edge Subdivision edge (not a quad-edge)

rotate Specifies which of the edges of the same quad-edge as the input one to return, one of the following:

- **0** the input edge (`e` on the picture below if `e` is the input edge)
- **1** the rotated edge (`eRot`)
- **2** the reversed edge (reversed `e` (in green))
- **3** the reversed rotated edge (reversed `eRot` (in green))



The function returns one of the edges of the same quad-edge as the input edge.

---

cvSubdivDelaunay2DInsert (view/add comments)

Inserts a single point into a Delaunay triangulation.

```
CvSubdiv2DPoint* cvSubdivDelaunay2DInsert(
            CvSubdiv2D* subdiv,
            CvPoint2D32f pt);
```

subdiv Delaunay subdivision created by the function cvCreateSubdivDelaunay2D

pt Inserted point

The function inserts a single point into a subdivision and modifies the subdivision topology appropriately. If a point with the same coordinates exists already, no new point is added. The function returns a pointer to the allocated point. No virtual point coordinates are calculated at this stage.

2.7 Motion Analysis and Object Tracking

cvAcc (view/add comments)

Adds a frame to an accumulator.

```
void cvAcc(
            const CvArr* image,
            CvArr* sum,
            const CvArr* mask=NULL );
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently)

sum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the whole image `image` or its selected region to the accumulator `sum`:

$$\texttt{sum}(x,y) \leftarrow \texttt{sum}(x,y) + \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

cvMultiplyAcc (view/add comments)

Adds the product of two input images to the accumulator.

```
void cvMultiplyAcc(
            const CvArr* image1,
            const CvArr* image2,
            CvArr* acc,
            const CvArr* mask=NULL );
```

image1 First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

image2 Second input image, the same format as the first one

acc Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `acc`:

$$\texttt{acc}(x,y) \leftarrow \texttt{acc}(x,y) + \texttt{image1}(x,y) \cdot \texttt{image2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cvRunningAvg (view/add comments)

Updates the running average.

```
void cvRunningAvg(
            const CvArr* image,
            CvArr* acc,
            double alpha,
            const CvArr* mask=NULL );
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

acc Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

alpha Weight of input image

mask Optional operation mask

The function calculates the weighted sum of the input image `image` and the accumulator `acc` so that `acc` becomes a running average of frame sequence:

$$\texttt{acc}(x,y) \leftarrow (1-\alpha) \cdot \texttt{acc}(x,y) + \alpha \cdot \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

where α regulates the update speed (how fast the accumulator forgets about previous frames).

cvSquareAcc (view/add comments)

Adds the square of the source image to the accumulator.

```
void cvSquareAcc(
           const CvArr* image,
           CvArr* sqsum,
           const CvArr* mask=NULL );
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

sqsum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum`:

$$\texttt{sqsum}(x,y) \leftarrow \texttt{sqsum}(x,y) + \texttt{image}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

2.8 Feature Detection

cvCanny (view/add comments)

Implements the Canny algorithm for edge detection.

```
void cvCanny(
           const CvArr* image,
           CvArr* edges,
           double threshold1,
           double threshold2,
           int aperture_size=3 );
```

image Single-channel input image

edges Single-channel image to store the edges found by the function

threshold1 The first threshold

threshold2 The second threshold

aperture_size Aperture parameter for the Sobel operator (see cvSobel)

The function finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges.

---

cvCornerEigenValsAndVecs (view/add comments)

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
void cvCornerEigenValsAndVecs(
           const CvArr* image,
           CvArr* eigenvv,
           int blockSize,
           int aperture_size=3 );
```

image Input image

eigenvv Image to store the results. It must be 6 times wider than the input image

blockSize Neighborhood size (see discussion)

aperture_size Aperture parameter for the Sobel operator (see cvSobel)

For every pixel, the function `cvCornerEigenValsAndVecs` considers a `blockSize×blockSize` neigborhood S(p). It calcualtes the covariation matrix of derivatives over the neigborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dx \cdot dI/dy)^2 \\ \sum_{S(p)}(dI/dx \cdot dI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the eigenvalues of M; not sorted

x_1, y_1 are the eigenvectors corresponding to λ_1

x_2, y_2 are the eigenvectors corresponding to λ_2

---

cvCornerHarris (view/add comments)

Harris edge detector.

```
void cvCornerHarris(
           const CvArr* image,
           CvArr* harris_dst,
           int blockSize,
           int aperture_size=3,
           double k=0.04 );
```

image Input image

harris_dst Image to store the Harris detector responses. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cvCornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cvSobel).

k Harris detector free parameter. See the formula below

The function runs the Harris edge detector on the image. Similarly to cvCornerMinEigenVal and cvCornerEigenValsAndVecs, for each pixel it calculates a 2×2 gradient covariation matrix M over a `blockSize` $\times$ `blockSize` neighborhood. Then, it stores

$$det(M) - k\, trace(M)^2$$

to the destination image. Corners in the image can be found as the local maxima of the destination image.

cvCornerMinEigenVal (view/add comments)

Calculates the minimal eigenvalue of gradient matrices for corner detection.

```
void cvCornerMinEigenVal(
           const CvArr* image,
           CvArr* eigenval,
           int blockSize,
           int aperture_size=3 );
```

image Input image

eigenval Image to store the minimal eigenvalues. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cvCornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cvSobel).

The function is similar to cvCornerEigenValsAndVecs but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. $min(\lambda_1, \lambda_2)$ in terms of the previous function.

cvFindCornerSubPix (view/add comments)

Refines the corner locations.

```
void cvFindCornerSubPix(
           const CvArr* image,
           CvPoint2D32f* corners,
           int count,
           CvSize win,
           CvSize zero_zone,
           CvTermCriteria criteria );
```

image Input image

corners Initial coordinates of the input corners; refined coordinates on output

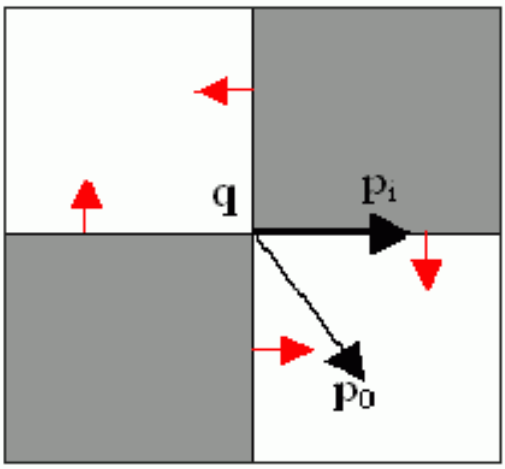
count Number of corners

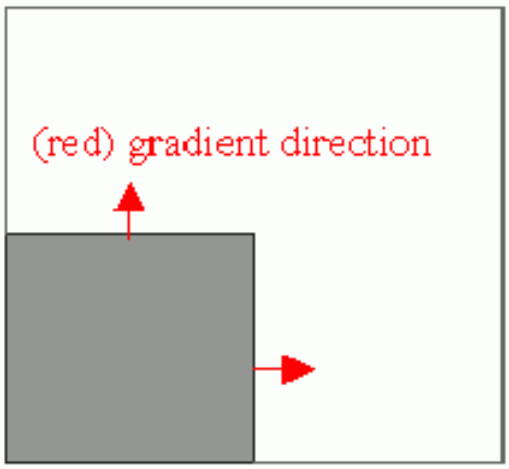
win Half of the side length of the search window. For example, if `win`=(5,5), then a $5*2+1\times 5*2+1=11\times 11$ search window would be used

zero_zone Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size

criteria Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below.





Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider the expression:

$$\epsilon_i = {DI_{p_i}}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot {DI_{p_i}}^T) q = \sum_i (DI_{p_i} \cdot {DI_{p_i}}^T \cdot p_i)$$

where the gradients are summed within a neighborhood ("search window") of q. Calling the first gradient term G and the second gradient term b gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center q and then iterates until the center keeps within a set threshold.

---

cvGoodFeaturesToTrack (view/add comments)

Determines strong corners on an image.

```
void cvGoodFeaturesToTrack(
           const CvArr* image
           CvArr* eigImage, CvArr* tempImage
           CvPoint2D32f* corners
           int* cornerCount
           double qualityLevel
           double minDistance
           const CvArr* mask=NULL
           int blockSize=3
           int useHarris=0
           double k=0.04 );
```

image The source 8-bit or floating-point 32-bit, single-channel image

eigImage Temporary floating-point 32-bit image, the same size as `image`

tempImage Another temporary image, the same size and format as `eigImage`

corners Output parameter; detected corners

cornerCount Output parameter; number of detected corners

qualityLevel Multiplier for the max/min eigenvalue; specifies the minimal accepted quality of image corners

minDistance Limit, specifying the minimum possible distance between the returned corners; Euclidian distance is used

mask Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL

blockSize Size of the averaging block, passed to the underlying cvCornerMinEigenVal or cvCornerHarris used by the function

useHarris If nonzero, Harris operator (cvCornerHarris) is used instead of default cvCornerMinEigenVal

k Free parameter of Harris detector; used only if ($\texttt{useHarris} != 0$)

The function finds the corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using the cvCornerMinEigenVal function and stores them in `eigImage`. Then it performs non-maxima suppression (only the local maxima in 3×3 neighborhood are retained). The next step rejects the corners with the minimal eigenvalue less than $\texttt{qualityLevel} \cdot max(\texttt{eigImage}(x, y))$. Finally, the function ensures that the distance between any two corners is not smaller than `minDistance`. The weaker corners (with a smaller min eigenvalue) that are too close to the stronger corners are rejected.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel`, and `A` ¿ B, the array of returned corners with `qualityLevel=A` will be the prefix of the output corners array with `qualityLevel=B`.

---

cvHoughLines2 (view/add comments)

Finds lines in a binary image using a Hough transform.

```
CvSeq* cvHoughLines2(
           CvArr* image,
           void* storage,
           int method,
           double rho,
           double theta,
           int threshold,
           double param1=0,
           double param2=0 );
```

image The 8-bit, single-channel, binary source image. In the case of a probabilistic method, the image is modified by the function

storage The storage for the lines that are detected. It can be a memory storage (in this case a sequence of lines is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of a particular type (see below) to which the lines' parameters are written. The matrix header is modified by the function so its `cols` or `rows` will contain the number of lines detected. If `storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (in the case of standard hough transform the lines are sorted by the accumulator value)

method The Hough transform variant, one of the following:

CV_HOUGH_STANDARD classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ), where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of `CV_32FC2` type

CV_HOUGH_PROBABILISTIC probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole line. Each segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of `CV_32SC4` type

CV_HOUGH_MULTI_SCALE multi-scale variant of the classical Hough transform. The lines are encoded the same way as `CV_HOUGH_STANDARD`

rho Distance resolution in pixel-related units

theta Angle resolution measured in radians

threshold Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`

param1 The first method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the minimum line length.
- For the multi-scale Hough transform it is the divisor for the distance resolution ρ. (The coarse distance resolution will be ρ and the accurate resolution will be $(\rho/\texttt{param1})$).

param2 The second method-dependent parameter:

- For the classical Hough transform it is not used (0).

- For the probabilistic Hough transform it is the maximum gap between line segments lying on the same line to treat them as a single line segment (i.e. to join them).
- For the multi-scale Hough transform it is the divisor for the angle resolution θ. (The coarse angle resolution will be θ and the accurate resolution will be $(\theta/\texttt{param2})$).

The function implements a few variants of the Hough transform for line detection.

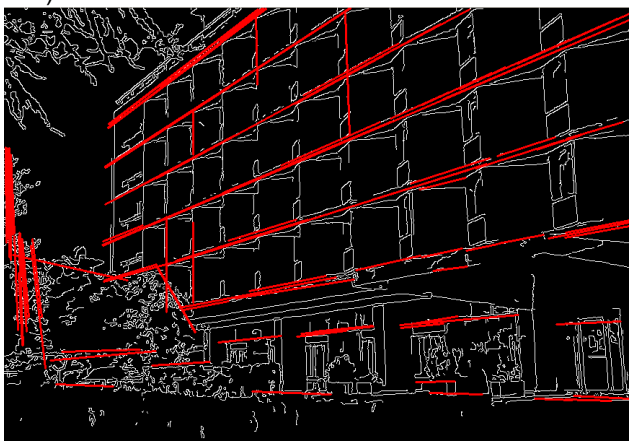
Example. Detecting lines with Hough transform.

```
/* This is a standalone program. Pass an image name as a first parameter
of the program.  Switch between standard and probabilistic Hough transform
by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* lines = 0;
        int i;
        cvCanny( src, dst, 50, 200, 3 );
        cvCvtColor( dst, color_dst, CV_GRAY2BGR );
#if 1
        lines = cvHoughLines2( dst,
                               storage,
                               CV_HOUGH_STANDARD,
                               1,
                               CV_PI/180,
                               100,
                               0,
                               0 );

        for( i = 0; i < MIN(lines->total,100); i++ )
        {
            float* line = (float*)cvGetSeqElem(lines,i);
            float rho = line[0];
            float theta = line[1];
            CvPoint pt1, pt2;
            double a = cos(theta), b = sin(theta);
            double x0 = a*rho, y0 = b*rho;
```

```
            pt1.x = cvRound(x0 + 1000*(-b));
            pt1.y = cvRound(y0 + 1000*(a));
            pt2.x = cvRound(x0 - 1000*(-b));
            pt2.y = cvRound(y0 - 1000*(a));
            cvLine( color_dst, pt1, pt2, CV_RGB(255,0,0), 3, 8 );
        }
#else
        lines = cvHoughLines2( dst,
                               storage,
                               CV_HOUGH_PROBABILISTIC,
                               1,
                               CV_PI/180,
                               80,
                               30,
                               10 );
        for( i = 0; i < lines->total; i++ )
        {
            CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
            cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3, 8 );
        }
#endif
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "Hough", 1 );
        cvShowImage( "Hough", color_dst );

        cvWaitKey(0);
    }
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in the case of probabilistic Hough transform (`#if 0` case):



---

cvPreCornerDetect (view/add comments)

Calculates the feature map for corner detection.

```
void cvPreCornerDetect(
           const CvArr* image,
           CvArr* corners,
           int apertureSize=3 );
```

image Input image

corners Image to store the corner candidates

apertureSize Aperture parameter for the Sobel operator (see cvSobel)

The function calculates the function

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}$$

where $D_?$ denotes one of the first image derivatives and $D_{??}$ denotes a second image derivative.

The corners can be found as local maximums of the function below:

```
// assume that the image is floating-point
IplImage* corners = cvCloneImage(image);
IplImage* dilated_corners = cvCloneImage(image);
IplImage* corner_mask = cvCreateImage( cvGetSize(image), 8, 1 );
cvPreCornerDetect( image, corners, 3 );
cvDilate( corners, dilated_corners, 0, 1 );
cvSubS( corners, dilated_corners, corners );
cvCmpS( corners, 0, corner_mask, CV_CMP_GE );
cvReleaseImage( &corners );
cvReleaseImage( &dilated_corners );
```

---

cvSampleLine (view/add comments)

Reads the raster line to the buffer.

```
int cvSampleLine(
           const CvArr* image
           CvPoint pt1
           CvPoint pt2
           void* buffer
           int connectivity=8 );
```

image Image to sample the line from

pt1 Starting line point

pt2 Ending line point

buffer Buffer to store the line points; must have enough size to store $max(|\texttt{pt2.x} - \texttt{pt1.x}| + 1, |\texttt{pt2.y} - \texttt{pt1.y}| + 1)$ points in the case of an 8-connected line and $(|\texttt{pt2.x} - \texttt{pt1.x}| + |\texttt{pt2.y} - \texttt{pt1.y}| + 1)$ in the case of a 4-connected line

connectivity The line connectivity, 4 or 8

The function implements a particular application of line iterators. The function reads all of the image points lying on the line between `pt1` and `pt2`, including the end points, and stores them into the buffer.

2.9 Object Detection

---

cvMatchTemplate (view/add comments)

Compares a template against overlapped image regions.

```
void cvMatchTemplate(
           const CvArr* image,
           const CvArr* templ,
           CvArr* result,
           int method );
```

image Image where the search is running; should be 8-bit or 32-bit floating-point

templ Searched template; must be not greater than the source image and the same data type as the image

result A map of comparison results; single-channel 32-bit floating-point. If `image` is $W \times H$ and `templ` is $w \times h$ then `result` must be $(W - w + 1) \times (H - h + 1)$

method Specifies the way the template must be compared with the image regions (see below)

The function is similar to cvCalcBackProjectPatch. It slides through `image`, compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result`. Here are the formulas for the different comparison methods one may use (I denotes `image`, T `template`, R `result`). The summation is done over template and/or the image patch: $x' = 0...w - 1, y' = 0...h - 1$

method=CV_TM_SQDIFF

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

method=CV_TM_SQDIFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCORR

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I(x+x',y+y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

After the function finishes the comparison, the best matches can be found as global minimums (`CV_TM_SQDIFF`) or maximums (`CV_TM_CCORR` and `CV_TM_CCOEFF`) using the cvMinMaxLoc function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel).

Chapter 3

features2d. Feature Detection and Descriptor Extraction

3.1 Feature detection and description

image The image. Keypoints (corners) will be detected on this.

keypoints Keypoints detected on the image.

threshold Threshold on difference between intensity of center pixel and pixels on circle around this pixel. See description of the algorithm.

nonmaxSupression If it is true then non-maximum supression will be applied to detected corners (keypoints).

---

cvExtractSURF (view/add comments)

Extracts Speeded Up Robust Features from an image.

```
void cvExtractSURF(
            const CvArr* image,
            const CvArr* mask,
            CvSeq** keypoints,
            CvSeq** descriptors,
            CvMemStorage* storage,
            CvSURFParams params );
```

image The input 8-bit grayscale image

mask The optional input 8-bit mask. The features are only found in the areas that contain more than 50% of non-zero mask pixels

keypoints The output parameter; double pointer to the sequence of keypoints. The sequence of CvSURFPoint structures is as follows:

```
typedef struct CvSURFPoint
{
   CvPoint2D32f pt; // position of the feature within the image
   int laplacian;   // -1, 0 or +1. sign of the laplacian at the point.
                    // can be used to speedup feature comparison
                    // (normally features with laplacians of different
            // signs can not match)
   int size;        // size of the feature
   float dir;       // orientation of the feature: 0..360 degrees
   float hessian;   // value of the hessian (can be used to
            // approximately estimate the feature strengths;
                    // see also params.hessianThreshold)
}
CvSURFPoint;
```

descriptors The optional output parameter; double pointer to the sequence of descriptors. Depending on the params.extended value, each element of the sequence will be either a 64-element or a 128-element floating-point (`CV_32F`) vector. If the parameter is NULL, the descriptors are not computed

storage Memory storage where keypoints and descriptors will be stored

params Various algorithm parameters put to the structure CvSURFParams:

```
typedef struct CvSURFParams
{
   int extended; // 0 means basic descriptors (64 elements each),
                 // 1 means extended descriptors (128 elements each)
   double hessianThreshold; // only features with keypoint.hessian
         // larger than that are extracted.
                 // good default value is ~300-500 (can depend on the
         // average local contrast and sharpness of the image).
                 // user can further filter out some features based on
         // their hessian values and other characteristics.
   int nOctaves; // the number of octaves to be used for extraction.
                 // With each next octave the feature size is doubled
         // (3 by default)
   int nOctaveLayers; // The number of layers within each octave
```

```
        // (4 by default)
}
CvSURFParams;

CvSURFParams cvSURFParams(double hessianThreshold, int extended=0);
        // returns default parameters
```

The function cvExtractSURF finds robust features in the image, as described in [**?**]. For each feature it returns its location, size, orientation and optionally the descriptor, basic or extended. The function can be used for object tracking and localization, image stitching etc.

See the `find_obj.cpp` demo in OpenCV samples directory.

---

cvGetStarKeypoints (view/add comments)

Retrieves keypoints using the StarDetector algorithm.

```
CvSeq* cvGetStarKeypoints(
           const CvArr* image,
           CvMemStorage* storage,
           CvStarDetectorParams params=cvStarDetectorParams() );
```

image The input 8-bit grayscale image

storage Memory storage where the keypoints will be stored

params Various algorithm parameters given to the structure CvStarDetectorParams:

```
typedef struct CvStarDetectorParams
{
   int maxSize; // maximal size of the features detected. The following
                // values of the parameter are supported:
                // 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
   int responseThreshold; // threshold for the approximatd laplacian,
                          // used to eliminate weak features
   int lineThresholdProjected; // another threshold for laplacian to
               // eliminate edges
   int lineThresholdBinarized; // another threshold for the feature
               // scale to eliminate edges
   int suppressNonmaxSize; // linear size of a pixel neighborhood
               // for non-maxima suppression
```

```
}
CvStarDetectorParams;
```

The function GetStarKeypoints extracts keypoints that are local scale-space extremas. The scale-space is constructed by computing approximate values of laplacians with different sigma's at each pixel. Instead of using pyramids, a popular approach to save computing time, all of the laplacians are computed at each pixel of the original high-resolution image. But each approximate laplacian value is computed in O(1) time regardless of the sigma, thanks to the use of integral images. The algorithm is based on the paper Agrawal08 , but instead of a square, hexagon or octagon it uses an 8-end star shape, hence the name, consisting of overlapping upright and tilted squares.

Each computed feature is represented by the following structure:

```
typedef struct CvStarKeypoint
{
    CvPoint pt; // coordinates of the feature
    int size; // feature size, see CvStarDetectorParams::maxSize
    float response; // the approximated laplacian value at that point.
}
CvStarKeypoint;

inline CvStarKeypoint cvStarKeypoint(CvPoint pt, int size, float response);
```

Below is the small usage sample:

```
#include "cv.h"
#include "highgui.h"

int main(int argc, char** argv)
{
    const char* filename = argc > 1 ? argv[1] : "lena.jpg";
    IplImage* img = cvLoadImage( filename, 0 ), *cimg;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* keypoints = 0;
    int i;

    if( !img )
        return 0;
    cvNamedWindow( "image", 1 );
    cvShowImage( "image", img );
    cvNamedWindow( "features", 1 );
    cimg = cvCreateImage( cvGetSize(img), 8, 3 );
    cvCvtColor( img, cimg, CV_GRAY2BGR );

    keypoints = cvGetStarKeypoints( img, storage, cvStarDetectorParams(45) );
```

```
    for( i = 0; i < (keypoints ? keypoints->total : 0); i++ )
    {
        CvStarKeypoint kpt = *(CvStarKeypoint*)cvGetSeqElem(keypoints, i);
        int r = kpt.size/2;
        cvCircle( cimg, kpt.pt, r, CV_RGB(0,255,0));
        cvLine( cimg, cvPoint(kpt.pt.x + r, kpt.pt.y + r),
            cvPoint(kpt.pt.x - r, kpt.pt.y - r), CV_RGB(0,255,0));
        cvLine( cimg, cvPoint(kpt.pt.x - r, kpt.pt.y + r),
            cvPoint(kpt.pt.x + r, kpt.pt.y - r), CV_RGB(0,255,0));
    }
    cvShowImage( "features", cimg );
    cvWaitKey();
}
```

Chapter 4

flann. Clustering and Search in Multi-Dimensional Spaces

4.1 Fast Approximate Nearest Neighbor Search

Chapter 5

objdetect. Object Detection

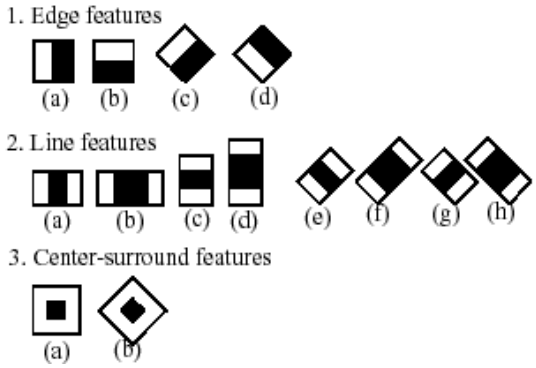
5.1 Cascade Classification

Haar Feature-based Cascade Classifier for Object Detection

The object detector described below has been initially proposed by Paul Viola cvViola01 and improved by Rainer Lienhart cvLienhart02 . First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundred sample views of a particular object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.

After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a ”1” if the region is likely to show the object (i.e., face/car), and ”0” otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily ”resized” in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word ”cascade” in the classifier name means that the resultant classifier consists of several simpler classifiers (*stages*) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word ”boosted” means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different `boosting` techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifers, and are calculated as described below. The current algorithm uses the following Haar-like features:



The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in the case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and the cvIntegral description).

To see the object detector at work, have a look at the HaarFaceDetect demo.

The following reference is for the detection part only. There is a separate application called `haartraining` that can train a cascade of boosted classifiers from a set of samples. See `opencv/apps/haartraining` for details.

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade (view/add comments)

Boosted Haar classifier structures.

```
#define CV_HAAR_FEATURE_MAX  3

/* a haar feature consists of 2-3 rectangles with appropriate weights */
typedef struct CvHaarFeature
{
    int  tilted;  /* 0 means up-right feature, 1 means 45--rotated feature */

    /* 2-3 rectangles with weights of opposite signs and
       with absolute values inversely proportional to the areas of the
       rectangles.  If rect[2].weight !=0, then
       the feature consists of 3 rectangles, otherwise it consists of 2 */
    struct
    {
```

```
        CvRect r;
        float weight;
    } rect[CV_HAAR_FEATURE_MAX];
}
CvHaarFeature;

/* a single tree classifier (stump in the simplest case) that returns the
   response for the feature at the particular image location (i.e. pixel
   sum over subrectangles of the window) and gives out a value depending
   on the response */
typedef struct CvHaarClassifier
{
    int count;  /* number of nodes in the decision tree */

    /* these are "parallel" arrays. Every index \texttt{i}
       corresponds to a node of the decision tree (root has 0-th index).

       left[i] - index of the left child (or negated index if the
         left child is a leaf)
       right[i] - index of the right child (or negated index if the
          right child is a leaf)
       threshold[i] - branch threshold. if feature responce is <= threshold,
                      left branch is chosen, otherwise right branch is chosen.
       alpha[i] - output value correponding to the leaf. */
    CvHaarFeature* haar_feature;
    float* threshold;
    int* left;
    int* right;
    float* alpha;
}
CvHaarClassifier;

/* a boosted battery of classifiers(=stage classifier):
   the stage classifier returns 1
   if the sum of the classifiers responses
   is greater than \texttt{threshold} and 0 otherwise */
typedef struct CvHaarStageClassifier
{
    int  count;  /* number of classifiers in the battery */
    float threshold; /* threshold for the boosted classifier */
    CvHaarClassifier* classifier; /* array of classifiers */

    /* these fields are used for organizing trees of stage classifiers,
       rather than just stright cascades */
    int next;
```

```
    int child;
    int parent;
}
CvHaarStageClassifier;

typedef struct CvHidHaarClassifierCascade CvHidHaarClassifierCascade;

/* cascade or tree of stage classifiers */
typedef struct CvHaarClassifierCascade
{
    int  flags; /* signature */
    int  count; /* number of stages */
    CvSize orig_window_size; /* original object size (the cascade is
                            trained for) */

    /* these two parameters are set by cvSetImagesForHaarClassifierCascade */
    CvSize real_window_size; /* current object size */
    double scale; /* current scale */
    CvHaarStageClassifier* stage_classifier; /* array of stage classifiers */
    CvHidHaarClassifierCascade* hid_cascade; /* hidden optimized
                        representation of the
                        cascade, created by
                cvSetImagesForHaarClassifierCascade */
}
CvHaarClassifierCascade;
```

All the structures are used for representing a cascaded of boosted Haar classifiers. The cascade has the following hierarchical structure:

```
    Cascade:
        Stage,,1,,:
            Classifier,,11,,:
                Feature,,11,,
            Classifier,,12,,:
                Feature,,12,,
            ...
        Stage,,2,,:
            Classifier,,21,,:
                Feature,,21,,
            ...
        ...
```

The whole hierarchy can be constructed manually or loaded from a file or an embedded base using the function cvLoadHaarClassifierCascade.

cvLoadHaarClassifierCascade (view/add comments)

Loads a trained cascade classifier from a file or the classifier database embedded in OpenCV.

```
CvHaarClassifierCascade* cvLoadHaarClassifierCascade(
            const char* directory,
            CvSize orig_window_size );
```

directory Name of the directory containing the description of a trained cascade classifier

orig_window_size Original size of the objects the cascade has been trained on. Note that it is not stored in the cascade and therefore must be specified separately

The function loads a trained cascade of haar classifiers from a file or the classifier database embedded in OpenCV. The base can be trained using the `haartraining` application (see opencv/apps/haartraining for details).

The function is obsolete. Nowadays object detection classifiers are stored in XML or YAML files, rather than in directories. To load a cascade from a file, use the cvLoad function.

cvHaarDetectObjects (view/add comments)

Detects objects in the image.

```
typedef struct CvAvgComp
{
    CvRect rect; /* bounding rectangle for the object (average rectangle of a group) */
    int neighbors; /* number of neighbor rectangles in the group */
}
CvAvgComp;
```

```
CvSeq* cvHaarDetectObjects(
            const CvArr* image,
            CvHaarClassifierCascade* cascade,
            CvMemStorage* storage,
            double scaleFactor=1.1,
            int minNeighbors=3,
```

```
            int flags=0,
            CvSize minSize=cvSize(0, 0),
            CvSize maxSize=cvSize(0,0) );
```

image Image to detect objects in

cascade Haar classifier cascade in internal representation

storage Memory storage to store the resultant sequence of the object candidate rectangles

scaleFactor The factor by which the search window is scaled between the subsequent scans, 1.1 means increasing window by 10%

minNeighbors Minimum number (minus 1) of neighbor rectangles that makes up an object. All the groups of a smaller number of rectangles than `min_neighbors`-1 are rejected. If `minNeighbors` is 0, the function does not any grouping at all and returns all the detected candidate rectangles, which may be useful if the user wants to apply a customized grouping procedure

flags Mode of operation. Currently the only flag that may be specified is `CV_HAAR_DO_CANNY_PRUNING`. If it is set, the function uses Canny edge detector to reject some image regions that contain too few or too much edges and thus can not contain the searched object. The particular threshold values are tuned for face detection and in this case the pruning speeds up the processing

minSize Minimum window size. By default, it is set to the size of samples the classifier has been trained on ($\sim 20 \times 20$ for face detection)

maxSize Maximum window size to use. By default, it is set to the size of the image.

The function finds rectangular regions in the given image that are likely to contain objects the cascade has been trained for and returns those regions as a sequence of rectangles. The function scans the image several times at different scales (see cvSetImagesForHaarClassifierCascade). Each time it considers overlapping regions in the image and applies the classifiers to the regions using cvRunHaarClassifierCascade. It may also apply some heuristics to reduce number of analyzed regions, such as Canny prunning. After it has proceeded and collected the candidate rectangles (regions that passed the classifier cascade), it groups them and returns a sequence of average rectangles for each large enough group. The default parameters (`scale_factor` =1.1, `min_neighbors` =3, `flags` =0) are tuned for accurate yet slow object detection. For a faster operation on real video images the settings are: `scale_factor` =1.2, `min_neighbors` =2, `flags`

=CV_HAAR_DO_CANNY_PRUNING, min_size =*minimum possible face size* (for example, $\sim$ 1/4 to 1/16 of the image area in the case of video conferencing).

```
#include "cv.h"
#include "highgui.h"

CvHaarClassifierCascade* load_object_detector( const char* cascade_path )
{
    return (CvHaarClassifierCascade*)cvLoad( cascade_path );
}

void detect_and_draw_objects( IplImage* image,
                              CvHaarClassifierCascade* cascade,
                              int do_pyramids )
{
    IplImage* small_image = image;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* faces;
    int i, scale = 1;

    /* if the flag is specified, down-scale the input image to get a
       performance boost w/o loosing quality (perhaps) */
    if( do_pyramids )
    {
        small_image = cvCreateImage( cvSize(image->width/2,image->height/2), IPL_DEPTH_8U,
        cvPyrDown( image, small_image, CV_GAUSSIAN_5x5 );
        scale = 2;
    }

    /* use the fastest variant */
    faces = cvHaarDetectObjects( small_image, cascade, storage, 1.2, 2, CV_HAAR_DO_CANNY_PR

    /* draw all the rectangles */
    for( i = 0; i < faces->total; i++ )
    {
        /* extract the rectanlges only */
        CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i );
        cvRectangle( image, cvPoint(face_rect.x*scale,face_rect.y*scale),
                     cvPoint((face_rect.x+face_rect.width)*scale,
                             (face_rect.y+face_rect.height)*scale),
                     CV_RGB(255,0,0), 3 );
    }

    if( small_image != image )
        cvReleaseImage( &small_image );
```

```
    cvReleaseMemStorage( &storage );
}

/* takes image filename and cascade path from the command line */
int main( int argc, char** argv )
{
    IplImage* image;
    if( argc==3 && (image = cvLoadImage( argv[1], 1 )) != 0 )
    {
        CvHaarClassifierCascade* cascade = load_object_detector(argv[2]);
        detect_and_draw_objects( image, cascade, 1 );
        cvNamedWindow( "test", 0 );
        cvShowImage( "test", image );
        cvWaitKey(0);
        cvReleaseHaarClassifierCascade( &cascade );
        cvReleaseImage( &image );
    }

    return 0;
}
```

---

cvSetImagesForHaarClassifierCascade (view/add comments)

Assigns images to the hidden cascade.

```
void cvSetImagesForHaarClassifierCascade(
           CvHaarClassifierCascade* cascade,
           const CvArr* sum,
           const CvArr* sqsum,
           const CvArr* tilted_sum,
           double scale );
```

cascade Hidden Haar classifier cascade, created by cvCreateHidHaarClassifierCascade

sum Integral (sum) single-channel image of 32-bit integer format. This image as well as the two subsequent images are used for fast feature evaluation and brightness/contrast normalization. They all can be retrieved from input 8-bit or floating point single-channel image using the function cvIntegral

sqsum Square sum single-channel image of 64-bit floating-point format

tilted_sum Tilted sum single-channel image of 32-bit integer format

scale Window scale for the cascade. If `scale` =1, the original window size is used (objects of that size are searched) - the same size as specified in cvLoadHaarClassifierCascade (24x24 in the case of `default_face_cascade`), if `scale` =2, a two times larger window is used (48x48 in the case of default face cascade). While this will speed-up search about four times, faces smaller than 48x48 cannot be detected

The function assigns images and/or window scale to the hidden classifier cascade. If image pointers are NULL, the previously set images are used further (i.e. NULLs mean "do not change images"). Scale parameter has no such a "protection" value, but the previous value can be retrieved by the cvGetHaarClassifierCascadeScale function and reused again. The function is used to prepare cascade for detecting object of the particular size in the particular image. The function is called internally by cvHaarDetectObjects, but it can be called by the user if they are using the lower-level function cvRunHaarClassifierCascade.

---

cvReleaseHaarClassifierCascade (view/add comments)

Releases the haar classifier cascade.

```
void cvReleaseHaarClassifierCascade(
           CvHaarClassifierCascade** cascade );
```

cascade Double pointer to the released cascade. The pointer is cleared by the function

The function deallocates the cascade that has been created manually or loaded using cvLoadHaarClassifierCascade or cvLoad.

---

cvRunHaarClassifierCascade (view/add comments)

Runs a cascade of boosted classifiers at the given image location.

```
int cvRunHaarClassifierCascade(
           CvHaarClassifierCascade* cascade,
           CvPoint pt,
           int start_stage=0 );
```

cascade Haar classifier cascade

pt Top-left corner of the analyzed region. Size of the region is a original window size scaled by the currenly set scale. The current window size may be retrieved using the cvGetHaarClassifierCascadeWindowSize function

start_stage Initial zero-based index of the cascade stage to start from. The function assumes that all the previous stages are passed. This feature is used internally by cvHaarDetectObjects for better processor cache utilization

The function runs the Haar classifier cascade at a single image location. Before using this function the integral images and the appropriate scale (window size) should be set using cvSetImagesForHaarClassifierCascade. The function returns a positive value if the analyzed rectangle passed all the classifier stages (it is a candidate) and a zero or negative value otherwise.

Chapter 6

video. Video Analysis

6.1 Motion Analysis and Object Tracking

cvCalcGlobalOrientation (view/add comments)

Calculates the global motion orientation of some selected region.

```
double cvCalcGlobalOrientation(
           const CvArr* orientation,
           const CvArr* mask,
           const CvArr* mhi,
           double timestamp,
           double duration );
```

orientation Motion gradient orientation image; calculated by the function cvCalcMotionGradient

mask Mask image. It may be a conjunction of a valid gradient mask, obtained with cvCalcMotionGradient and the mask of the region, whose direction needs to be calculated

mhi Motion history image

timestamp Current time in milliseconds or other units, it is better to store time passed to cvUpdateMotionHistory before and reuse it here, because running cvUpdateMotionHistory and cvCalcMotionGradient on large images may take some time

duration Maximal duration of motion track in milliseconds, the same as cvUpdateMotionHistory

The function calculates the general motion direction in the selected region and returns the angle between 0 degrees and 360 degrees . At first the function builds the orientation histogram and finds the basic orientation as a coordinate of the histogram maximum. After that the function calculates the shift relative to the basic orientation as a weighted sum of all of the orientation vectors: the more recent the motion, the greater the weight. The resultant angle is a circular sum of the basic orientation and the shift.

---

cvCalcMotionGradient (view/add comments)

Calculates the gradient orientation of a motion history image.

```
void cvCalcMotionGradient(
           const CvArr* mhi,
           CvArr* mask,
           CvArr* orientation,
           double delta1,
           double delta2,
           int apertureSize=3 );
```

mhi Motion history image

mask Mask image; marks pixels where the motion gradient data is correct; output parameter

orientation Motion gradient orientation image; contains angles from 0 to 360 degrees

delta1 See below

delta2 See below

apertureSize Aperture size of derivative operators used by the function: CV_SCHARR, 1, 3, 5 or 7 (see cvSobel)

The function calculates the derivatives Dx and Dy of `mhi` and then calculates gradient orientation as:

$$\texttt{orientation}(x,y) = \arctan \frac{Dy(x,y)}{Dx(x,y)}$$

where both $Dx(x,y)$ and $Dy(x,y)$ signs are taken into account (as in the cvCartToPolar function). After that `mask` is filled to indicate where the orientation is valid (see the `delta1` and `delta2` description).

The function finds the minimum ($m(x,y)$) and maximum ($M(x,y)$) mhi values over each pixel (x,y) neighborhood and assumes the gradient is valid only if

$$\min(\texttt{delta1},\texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1},\texttt{delta2}).$$

---

cvCalcOpticalFlowBM (view/add comments)

Calculates the optical flow for two images by using the block matching method.

```
void cvCalcOpticalFlowBM(
           const CvArr* prev,
           const CvArr* curr,
           CvSize blockSize,
           CvSize shiftSize,
           CvSize max_range,
           int usePrevious,
           CvArr* velx,
           CvArr* vely );
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

blockSize Size of basic blocks that are compared

shiftSize Block coordinate increments

max_range Size of the scanned neighborhood in pixels around the block

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of

$$\left\lfloor \frac{\texttt{prev->width} - \texttt{blockSize.width}}{\texttt{shiftSize.width}} \right\rfloor \times \left\lfloor \frac{\texttt{prev->height} - \texttt{blockSize.height}}{\texttt{shiftSize.height}} \right\rfloor$$

size, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size `velx`, 32-bit floating-point, single-channel

The function calculates the optical flow for overlapped blocks `blockSize.width×blockSize.height` pixels each, thus the velocity fields are smaller than the original images. For every block in `prev` the functions tries to find a similar block in `curr` in some neighborhood of the original block or shifted by (velx(x0,y0),vely(x0,y0)) block as has been calculated by previous function call (if `usePrevious=1`)

---

cvCalcOpticalFlowHS (view/add comments)

Calculates the optical flow for two images.

```
void cvCalcOpticalFlowHS(
           const CvArr* prev,
           const CvArr* curr,
           int usePrevious,
           CvArr* velx,
           CvArr* vely,
           double lambda,
           CvTermCriteria criteria );
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

lambda Lagrangian multiplier

criteria Criteria of termination of velocity computing

The function computes the flow for every pixel of the first input image using the Horn and Schunck algorithm [**?**].

cvCalcOpticalFlowLK (view/add comments)

Calculates the optical flow for two images.

```
void cvCalcOpticalFlowLK(
           const CvArr* prev,
           const CvArr* curr,
           CvSize winSize,
           CvArr* velx,
           CvArr* vely );
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

winSize Size of the averaging window used for grouping pixels

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

The function computes the flow for every pixel of the first input image using the Lucas and Kanade algorithm [**?**].

cvCalcOpticalFlowPyrLK (view/add comments)

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

```
void cvCalcOpticalFlowPyrLK(
           const CvArr* prev,
           const CvArr* curr,
           CvArr* prevPyr,
           CvArr* currPyr,
```

```
        const CvPoint2D32f* prevFeatures,
        CvPoint2D32f* currFeatures,
        int count,
        CvSize winSize,
        int level,
        char* status,
        float* track_error,
        CvTermCriteria criteria,
        int flags );
```

prev First frame, at time `t`

curr Second frame, at time `t + dt`

prevPyr Buffer for the pyramid for the first frame. If the pointer is not `NULL` , the buffer must have a sufficient size to store the pyramid from level `1` to level `level` ; the total size of `(image_width+8)*image_height/3` bytes is sufficient

currPyr Similar to `prevPyr`, used for the second frame

prevFeatures Array of points for which the flow needs to be found

currFeatures Array of 2D points containing the calculated new positions of the input features in the second image

count Number of feature points

winSize Size of the search window of each pyramid level

level Maximal pyramid level number. If `0` , pyramids are not used (single level), if `1` , two levels are used, etc

status Array. Every element of the array is set to `1` if the flow for the corresponding feature has been found, `0` otherwise

track_error Array of double numbers containing the difference between patches around the original and moved points. Optional parameter; can be `NULL`

criteria Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped

flags Miscellaneous flags:

CV_LKFLOWPyr_A_READY pyramid for the first frame is precalculated before the call

CV_LKFLOWPyr_B_READY pyramid for the second frame is precalculated before the call

CV_LKFLOW_INITIAL_GUESSES array B contains initial coordinates of features before the function call

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids [**?**] . It calculates the coordinates of the feature points on the current video frame given their coordinates on the previous frame. The function finds the coordinates with sub-pixel accuracy.

Both parameters `prevPyr` and `currPyr` comply with the following rules: if the image pointer is 0, the function allocates the buffer internally, calculates the pyramid, and releases the buffer after processing. Otherwise, the function calculates the pyramid and stores it in the buffer unless the flag `CV_LKFLOWPyr_A[B]_READY` is set. The image should be large enough to fit the Gaussian pyramid data. After the function call both pyramids are calculated and the readiness flag for the corresponding image can be set in the next call (i.e., typically, for all the image pairs except the very first one `CV_LKFLOWPyr_A_READY` is set).

---

cvCamShift (view/add comments)

Finds the object center, size, and orientation.

```
int cvCamShift(
            const CvArr* prob_image,
            CvRect window,
            CvTermCriteria criteria,
            CvConnectedComp* comp,
            CvBox2D* box=NULL );
```

prob_image Back projection of object histogram (see cvCalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

box Circumscribed box for the object. If not `NULL`, it contains object size and orientation

The function implements the CAMSHIFT object tracking algrorithm [**?**]. First, it finds an object center using cvMeanShift and, after that, calculates the object size and orientation. The function returns number of iterations made within cvMeanShift.

The `CamShiftTracker` class declared in cv.hpp implements the color object tracker that uses the function.

CvConDensation

ConDenstation state.

```
    typedef struct CvConDensation
    {
        int MP;     //Dimension of measurement vector
        int DP;     // Dimension of state vector
        float* DynamMatr;       // Matrix of the linear Dynamics system
        float* State;           // Vector of State
        int SamplesNum;         // Number of the Samples
        float** flSamples;      // array of the Sample Vectors
        float** flNewSamples;   // temporary array of the Sample Vectors
        float* flConfidence;    // Confidence for each Sample
        float* flCumulative;    // Cumulative confidence
        float* Temp;            // Temporary vector
        float* RandomSample;    // RandomVector to update sample set
        CvRandState* RandS;     // Array of structures to generate random vectors
    } CvConDensation;
```

The structure `CvConDensation` stores the CONditional DENSity propagATION tracker state. The information about the algorithm can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/ISARD1/condensation.html.

---

cvCreateConDensation (view/add comments)

Allocates the ConDensation filter structure.

```
CvConDensation* cvCreateConDensation(
            int dynam_params,
            int measure_params,
            int sample_count );
```

dynam_params Dimension of the state vector

measure_params Dimension of the measurement vector

sample_count Number of samples

The function creates a `CvConDensation` structure and returns a pointer to the structure.

---

cvConDensInitSampleSet (view/add comments)

Initializes the sample set for the ConDensation algorithm.

```
void cvConDensInitSampleSet( CvConDensation* condens,
            CvMat* lower_bound,
            CvMat* upper_bound );
```

condens Pointer to a structure to be initialized

lower_bound Vector of the lower boundary for each dimension

upper_bound Vector of the upper boundary for each dimension

The function fills the samples arrays in the structure `condens` with values within the specified ranges.

CvKalman (view/add comments)

Kalman filter state.

```
typedef struct CvKalman
{
    int MP;                     /* number of measurement vector dimensions */
    int DP;                     /* number of state vector dimensions */
    int CP;                     /* number of control vector dimensions */

    /* backward compatibility fields */
#if 1
    float* PosterState;         /* =state_pre->data.fl */
    float* PriorState;          /* =state_post->data.fl */
    float* DynamMatr;           /* =transition_matrix->data.fl */
    float* MeasurementMatr;     /* =measurement_matrix->data.fl */
    float* MNCovariance;        /* =measurement_noise_cov->data.fl */
```

```
    float* PNCovariance;        /* =process_noise_cov->data.fl */
    float* KalmGainMatr;        /* =gain->data.fl */
    float* PriorErrorCovariance;/* =error_cov_pre->data.fl */
    float* PosterErrorCovariance;/* =error_cov_post->data.fl */
    float* Temp1;               /* temp1->data.fl */
    float* Temp2;               /* temp2->data.fl */
#endif

    CvMat* state_pre;           /* predicted state (x'(k)):
                                    x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* corrected state (x(k)):
                                    x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* state transition matrix (A) */
    CvMat* control_matrix;      /* control matrix (B)
                                   (it is not used if there is no control)*/
    CvMat* measurement_matrix;  /* measurement matrix (H) */
    CvMat* process_noise_cov;   /* process noise covariance matrix (Q) */
    CvMat* measurement_noise_cov; /* measurement noise covariance matrix (R) */
    CvMat* error_cov_pre;       /* priori error estimate covariance matrix (P'(k)):
                                    P'(k)=A*P(k-1)*At + Q*/
    CvMat* gain;                /* Kalman gain matrix (K(k)):
                                    K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* posteriori error estimate covariance matrix (P(k)):
                                    P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

The structure `CvKalman` is used to keep the Kalman filter state. It is created by the cvCreateKalman function, updated by the cvKalmanPredict and cvKalmanCorrect functions and released by the cvReleaseKalman function . Normally, the structure is used for the standard Kalman filter (notation and the formulas below are borrowed from the excellent Kalman tutorial [**?**])

$$x_k = A \cdot x_{k-1} + B \cdot u_k + w_k$$
$$z_k = H \cdot x_k + v_k$$

where:

x_k (x_{k-1})	state of the system at the moment *k* (*k-1*)
z_k	measurement of the system state at the moment *k*
u_k	external control applied at the moment *k*

w_k and v_k are normally-distributed process and measurement noise, respectively:

$$p(w) \sim N(0, Q)$$
$$p(v) \sim N(0, R)$$

that is,
Q process noise covariance matrix, constant or variable,
R measurement noise covariance matrix, constant or variable

In the case of the standard Kalman filter, all of the matrices: A, B, H, Q and R are initialized once after the cvCvKalman structure is allocated via cvCreateKalman. However, the same structure and the same functions may be used to simulate the extended Kalman filter by linearizing the extended Kalman filter equation in the current system state neighborhood, in this case A, B, H (and, probably, Q and R) should be updated on every step.

---

cvCreateKalman (view/add comments)

Allocates the Kalman filter structure.

```
CvKalman* cvCreateKalman(
            int dynam_params,
            int measure_params,
            int control_params=0 );
```

dynam_params dimensionality of the state vector

measure_params dimensionality of the measurement vector

control_params dimensionality of the control vector

The function allocates cvCvKalman and all its matrices and initializes them somehow.

---

cvKalmanCorrect (view/add comments)

Adjusts the model state.

```
const CvMat* cvKalmanCorrect( CvKalman* kalman, const CvMat*
measurement );
```

kalman Pointer to the structure to be updated

measurement CvMat containing the measurement vector

The function adjusts the stochastic model state on the basis of the given measurement of the model state:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

where

z_k	given measurement (`mesurement` parameter)
K_k	Kalman "gain" matrix.

The function stores the adjusted state at `kalman->state_post` and returns it on output.

Example. Using Kalman filter to track a rotating point

```
#include "cv.h"
#include "highgui.h"
#include <math.h>

int main(int argc, char** argv)
{
    /* A matrix data */
    const float A[] = { 1, 1, 0, 1 };

    IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
    CvKalman* kalman = cvCreateKalman( 2, 1, 0 );
    /* state is (phi, delta_phi) - angle and angle increment */
    CvMat* state = cvCreateMat( 2, 1, CV_32FC1 );
    CvMat* process_noise = cvCreateMat( 2, 1, CV_32FC1 );
    /* only phi (angle) is measured */
    CvMat* measurement = cvCreateMat( 1, 1, CV_32FC1 );
    CvRandState rng;
    int code = -1;

    cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

    cvZero( measurement );
    cvNamedWindow( "Kalman", 1 );

    for(;;)
    {
        cvRandSetRange( &rng, 0, 0.1, 0 );
        rng.disttype = CV_RAND_NORMAL;
```

```
    cvRand( &rng, state );

    memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
    cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1) );
    cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-5) );
    cvSetIdentity( kalman->measurement_noise_cov, cvRealScalar(1e-1) );
    cvSetIdentity( kalman->error_cov_post, cvRealScalar(1));
    /* choose random initial state */
    cvRand( &rng, kalman->state_post );

    rng.disttype = CV_RAND_NORMAL;

    for(;;)
    {
        #define calc_point(angle)                                      \
            cvPoint( cvRound(img->width/2 + img->width/3*cos(angle)),  \
                     cvRound(img->height/2 - img->width/3*sin(angle)))

        float state_angle = state->data.fl[0];
        CvPoint state_pt = calc_point(state_angle);

        /* predict point position */
        const CvMat* prediction = cvKalmanPredict( kalman, 0 );
        float predict_angle = prediction->data.fl[0];
        CvPoint predict_pt = calc_point(predict_angle);
        float measurement_angle;
        CvPoint measurement_pt;

        cvRandSetRange( &rng,
                        0,
                        sqrt(kalman->measurement_noise_cov->data.fl[0]),
                        0 );
        cvRand( &rng, measurement );

        /* generate measurement */
        cvMatMulAdd( kalman->measurement_matrix, state, measurement, measurement );

        measurement_angle = measurement->data.fl[0];
        measurement_pt = calc_point(measurement_angle);

        /* plot points */
        #define draw_cross( center, color, d )                         \
            cvLine( img, cvPoint( center.x - d, center.y - d ),        \
                         cvPoint( center.x + d, center.y + d ),        \
```

```
                            color, 1, 0 );                                  \
                cvLine( img, cvPoint( center.x + d, center.y - d ),         \
                            cvPoint( center.x - d, center.y + d ),          \
                            color, 1, 0 )

            cvZero( img );
            draw_cross( state_pt, CV_RGB(255,255,255), 3 );
            draw_cross( measurement_pt, CV_RGB(255,0,0), 3 );
            draw_cross( predict_pt, CV_RGB(0,255,0), 3 );
            cvLine( img, state_pt, predict_pt, CV_RGB(255,255,0), 3, 0 );

            /* adjust Kalman filter state */
            cvKalmanCorrect( kalman, measurement );

            cvRandSetRange( &rng,
                            0,
                            sqrt(kalman->process_noise_cov->data.fl[0]),
                            0 );
            cvRand( &rng, process_noise );
            cvMatMulAdd( kalman->transition_matrix,
                         state,
                         process_noise,
                         state );

            cvShowImage( "Kalman", img );
            code = cvWaitKey( 100 );

            if( code > 0 ) /* break current simulation by pressing a key */
                break;
        }
        if( code == 27 ) /* exit by ESCAPE */
            break;
    }

    return 0;
}
```

---

cvKalmanPredict (view/add comments)

Estimates the subsequent model state.

```
const CvMat* cvKalmanPredict(
            CvKalman* kalman,
            const CvMat* control=NULL);
```

kalman Kalman filter state

control Control vector u_k, should be NULL iff there is no external control (control_params =0)

The function estimates the subsequent stochastic model state by its current state and stores it at kalman->state_pre:

$$x'_k = Ax_{k-1} + Bu_k$$
$$P'_k = AP_{k-1}A^T + Q$$

where

x'_k	is predicted state kalman->state_pre,
x_{k-1}	is corrected state on the previous step kalman->state_post (should be initialized somehow in the beginning, zero vector by default),
u_k	is external control (control parameter),
P'_k	is priori error covariance matrix kalman->error_cov_pre
P_{k-1}	is posteriori error covariance matrix on the previous step kalman->error_cov_post (should be initialized somehow in the beginning, identity matrix by default),

The function returns the estimated state.

KalmanUpdateByMeasurement

Synonym for KalmanCorrect

KalmanUpdateByTime

Synonym for KalmanPredict

cvMeanShift (view/add comments)

Finds the object center on back projection.

```
int cvMeanShift(
            const CvArr* prob_image,
            CvRect window,
            CvTermCriteria criteria,
            CvConnectedComp* comp );
```

prob_image Back projection of the object histogram (see cvCalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

The function iterates to find the object center given its back projection and initial position of search window. The iterations are made until the search window center moves by less than the given value and/or until the function has done the maximum number of iterations. The function returns the number of iterations made.

---

cvReleaseConDensation (view/add comments)

Deallocates the ConDensation filter structure.

```
void cvReleaseConDensation( CvConDensation** condens );
```

condens Pointer to the pointer to the structure to be released

The function releases the structure `condens`) and frees all memory previously allocated for the structure.

---

cvReleaseKalman (view/add comments)

Deallocates the Kalman filter structure.

```
void cvReleaseKalman(
           CvKalman** kalman );
```

kalman double pointer to the Kalman filter structure

The function releases the structure cvCvKalman and all of the underlying matrices.

---

cvSegmentMotion (view/add comments)

Segments a whole motion into separate moving parts.

```
CvSeq* cvSegmentMotion(
           const CvArr* mhi,
           CvArr* seg_mask,
           CvMemStorage* storage,
           double timestamp,
           double seg_thresh );
```

mhi Motion history image

seg_mask Image where the mask found should be stored, single-channel, 32-bit floating-point

storage Memory storage that will contain a sequence of motion connected components

timestamp Current time in milliseconds or other units

seg_thresh Segmentation threshold; recommended to be equal to the interval between motion history ”steps” or greater

The function finds all of the motion segments and marks them in `seg_mask` with individual values (1,2,...). It also returns a sequence of cvCvConnectedComp structures, one for each motion component. After that the motion direction for every component can be calculated with cvCalcGlobalOrientation using the extracted mask of the particular component cvCmp.

cvSnakeImage (view/add comments)

Changes the contour position to minimize its energy.

```
void cvSnakeImage(
           const IplImage* image,
           CvPoint* points,
           int length,
           float* alpha,
           float* beta,
           float* gamma,
           int coeff_usage,
           CvSize win,
           CvTermCriteria criteria,
           int calc_gradient=1 );
```

image The source image or external energy field

points Contour points (snake)

length Number of points in the contour

alpha Weight[s] of continuity energy, single float or array of `length` floats, one for each contour point

beta Weight[s] of curvature energy, similar to `alpha`

gamma Weight[s] of image energy, similar to `alpha`

coeff_usage Different uses of the previous three parameters:

- **CV_VALUE** indicates that each of `alpha, beta, gamma` is a pointer to a single value to be used for all points;
- **CV_ARRAY** indicates that each of `alpha, beta, gamma` is a pointer to an array of coefficients different for all the points of the snake. All the arrays must have the size equal to the contour size.

win Size of neighborhood of every point used to search the minimum, both `win.width` and `win.height` must be odd

criteria Termination criteria

calc_gradient Gradient flag; if not 0, the function calculates the gradient magnitude for every image pixel and consideres it as the energy field, otherwise the input image itself is considered

The function updates the snake in order to minimize its total energy that is a sum of internal energy that depends on the contour shape (the smoother contour is, the smaller internal energy is) and external energy that depends on the energy field and reaches minimum at the local energy extremums that correspond to the image edges in the case of using an image gradient.

The parameter `criteria.epsilon` is used to define the minimal number of points that must be moved during any iteration to keep the iteration process running.

If at some iteration the number of moved points is less than `criteria.epsilon` or the function performed `criteria.max_iter` iterations, the function terminates.

---

cvUpdateMotionHistory (view/add comments)

Updates the motion history image by a moving silhouette.

```
void cvUpdateMotionHistory(
           const CvArr* silhouette,
           CvArr* mhi,
           double timestamp,
           double duration );
```

silhouette Silhouette mask that has non-zero pixels where the motion occurs

mhi Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp Current time in milliseconds or other units

duration Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

Chapter 7

highgui. High-level GUI and Media I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and ”remember” their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

7.1 User Interface

---

cvConvertImage (view/add comments)

Converts one image to another with an optional vertical flip.

```
void cvConvertImage( const CvArr* src, CvArr* dst, int flags=0 );
```

src Source image.

dst Destination image. Must be single-channel or 3-channel 8-bit image.

flags The operation flags:

CV_CVTIMG_FLIP Flips the image vertically

CV_CVTIMG_SWAP_RB Swaps the red and blue channels. In OpenCV color images have `BGR` channel order, however on some systems the order needs to be reversed before displaying the image (ShowImage does this automatically).

The function `cvConvertImage` converts one image to another and flips the result vertically if desired. The function is used by ShowImage .

---

cvCreateTrackbar (view/add comments)

Creates a trackbar and attaches it to the specified window

```
int cvCreateTrackbar(
           const char* trackbarName,
           const char* windowName,
           int* value,
           int count,
           CvTrackbarCallback onChange );
```

trackbarName Name of the created trackbar.

windowName Name of the window which will be used as a parent for created trackbar.

value Pointer to an integer variable, whose value will reflect the position of the slider. Upon creation, the slider position is defined by this variable.

count Maximal position of the slider. Minimal position is always 0.

onChange Pointer to the function to be called every time the slider changes position. This function should be prototyped as `void Foo(int);` Can be NULL if callback is not required.

The function `cvCreateTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable to be syncronized with trackbar position and specifies a callback function to be called on trackbar position change. The created trackbar is displayed on the top of the given window.

[Qt Backend Only] qt-specific details:

windowName Name of the window which will be used as a parent for created trackbar. Can be NULL if the trackbar should be attached to the control panel.

The created trackbar is displayed at the bottom of the given window if *windowName* is correctly provided, or displayed on the control panel if *windowName* is NULL.

By clicking on the label of each trackbar, it is possible to edit the trackbar's value manually for a more accurate control of it.

```
CV_EXTERN_C_FUNCPTR( void (*CvTrackbarCallback)(int pos) );
```

---

cvDestroyAllWindows (view/add comments)

Destroys all of the HighGUI windows.

```
void cvDestroyAllWindows(void);
```

The function `cvDestroyAllWindows` destroys all of the opened HighGUI windows.

---

cvDestroyWindow (view/add comments)

Destroys a window.

```
void cvDestroyWindow( const char* name );
```

name Name of the window to be destroyed.

The function `cvDestroyWindow` destroys the window with the given name.

cvGetTrackbarPos (view/add comments)

Returns the trackbar position.

```
int cvGetTrackbarPos(
            const char* trackbarName,
            const char* windowName );
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of the trackbar.

The function `cvGetTrackbarPos` returns the current position of the specified trackbar.

[Qt Backend Only] qt-specific details:

windowName Name of the window which is the parent of the trackbar. Can be NULL if the trackbar is attached to the control panel.

cvGetWindowHandle (view/add comments)

Gets the window's handle by its name.

```
void* cvGetWindowHandle( const char* name );
```

name Name of the window .

The function `cvGetWindowHandle` returns the native window handle (HWND in case of Win32 and GtkWidget in case of GTK+).

[Qt Backend Only] qt-specific details: The function `cvGetWindowHandle` returns the native window handle inheriting from the Qt class QWidget.

cvGetWindowName (view/add comments)

Gets the window's name by its handle.

```
const char* cvGetWindowName( void* windowHandle );
```

windowHandle Handle of the window.

The function `cvGetWindowName` returns the name of the window given its native handle (HWND in case of Win32 and GtkWidget in case of GTK+).

[Qt Backend Only] qt-specific details: The function `cvGetWindowName` returns the name of the window given its native handle (QWidget).

cvInitSystem (view/add comments)

Initializes HighGUI.

```
int cvInitSystem( int argc, char** argv );
```

argc Number of command line arguments

argv Array of command line arguments

The function `cvInitSystem` initializes HighGUI. If it wasn't called explicitly by the user before the first window was created, it is called implicitly then with `argc=0`, `argv=NULL`. Under Win32 there is no need to call it explicitly. Under X Window the arguments may be used to customize a look of HighGUI windows and controls.

[Qt Backend Only] qt-specific details: The function `cvInitSystem` is automatically called at the first cvNameWindow call.

cvMoveWindow (view/add comments)

Sets the position of the window.

```
void cvMoveWindow( const char* name, int x, int y );
```

name Name of the window to be moved.

x New x coordinate of the top-left corner

y New y coordinate of the top-left corner

The function `cvMoveWindow` changes the position of the window.

cvNamedWindow (view/add comments)

Creates a window.

```
int cvNamedWindow( const char* name, int flags );
```

name Name of the window in the window caption that may be used as a window identifier.

flags Flags of the window. Currently the only supported flag is `CV_WINDOW_AUTOSIZE`. If this is set, window size is automatically adjusted to fit the displayed image (see ShowImage), and the user can not change the window size manually.

The function `cvNamedWindow` creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing.

[Qt Backend Only] qt-specific details:

flags Flags of the window. Currently the supported flags are:

CV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE: CV_WINDOW_NORMAL let the user resize the window, whereas CV_WINDOW_AUTOSIZE adjusts automatically the window's size to fit the displayed image (see ShowImage), and the user can not change the window size manually.

CV_WINDOW_FREERATIO or CV_WINDOW_KEEPRATIO: CV_WINDOW_FREERATIO adjust the image without respect the its ration, whereas CV_WINDOW_KEEPRATIO keep the image's ratio.

CV_GUI_NORMAL or CV_GUI_EXPANDED: CV_GUI_NORMAL is the old way to draw the window without statusbar and toolbar, whereas CV_GUI_EXPANDED is the new enhance GUI.

This parameter is optional. The default flags set for a new window are CV_WINDOW_AUTOSIZE, CV_WINDOW_KEEPRATIO, and CV_GUI_EXPANDED.

However, if you want to modify the flags, you can combine them using OR operator, ie:

```
V_WINDOW_NORMALV_GUI_NORMAL
```

---

cvResizeWindow (view/add comments)

Sets the window size.

```
void cvResizeWindow( const char* name, int width, int height );
```

name Name of the window to be resized.

width New width

height New height

The function cvResizeWindow changes the size of the window.

---

cvSetMouseCallback (view/add comments)

Assigns callback for mouse events.

```
void cvSetMouseCallback( const char* windowName, CvMouseCallback
onMouse, void* param=NULL );
```

windowName Name of the window.

onMouse Pointer to the function to be called every time a mouse event occurs in the specified window. This function should be prototyped as `void Foo(int event, int x, int y, int flags, void* param);` where `event` is one of `CV_EVENT_*`, `x` and `y` are the coordinates of the mouse pointer in image coordinates (not window coordinates), `flags` is a combination of `CV_EVENT_FLAG_*`, and `param` is a user-defined parameter passed to the `cvSetMouseCallback` function call.

param User-defined parameter to be passed to the callback function.

The function `cvSetMouseCallback` sets the callback function for mouse events occuring within the specified window.

The `event` parameter is one of:

CV_EVENT_MOUSEMOVE Mouse movement

CV_EVENT_LBUTTONDOWN Left button down

CV_EVENT_RBUTTONDOWN Right button down

CV_EVENT_MBUTTONDOWN Middle button down

CV_EVENT_LBUTTONUP Left button up

CV_EVENT_RBUTTONUP Right button up

CV_EVENT_MBUTTONUP Middle button up

CV_EVENT_LBUTTONDBLCLK Left button double click

CV_EVENT_RBUTTONDBLCLK Right button double click

CV_EVENT_MBUTTONDBLCLK Middle button double click

The `flags` parameter is a combination of :

CV_EVENT_FLAG_LBUTTON Left button pressed

CV_EVENT_FLAG_RBUTTON Right button pressed

CV_EVENT_FLAG_MBUTTON Middle button pressed

CV_EVENT_FLAG_CTRLKEY Control key pressed

CV_EVENT_FLAG_SHIFTKEY Shift key pressed

CV_EVENT_FLAG_ALTKEY Alt key pressed

cvSetTrackbarPos (view/add comments)

Sets the trackbar position.

```
void cvSetTrackbarPos(
            const char* trackbarName,
            const char* windowName,
            int pos );
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of trackbar.

pos New position.

The function cvSetTrackbarPos sets the position of the specified trackbar.

[Qt Backend Only] qt-specific details:

windowName Name of the window which is the parent of trackbar. Can be NULL if the trackbar is attached to the control panel.

cvShowImage (view/add comments)

Displays the image in the specified window

```
void cvShowImage( const char* name, const CvArr* image );
```

name Name of the window.

image Image to be shown.

The function cvShowImage displays the image in the specified window. If the window was created with the CV_WINDOW_AUTOSIZE flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

---

cvWaitKey (view/add comments)

Waits for a pressed key.

```
int cvWaitKey( int delay=0 );
```

delay Delay in milliseconds.

The function `cvWaitKey` waits for key event infinitely (`delay` $<= 0$) or for `delay` milliseconds. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

[Qt Backend Only] qt-specific details: With this current Qt implementation, this is the only way to process event such as repaint for the windows, and so on ...

7.2 Reading and Writing Images and Video

---

cvLoadImage (view/add comments)

Loads an image from a file as an IplImage.

```
IplImage* cvLoadImage(
           const char* filename,
           int iscolor=CV_LOAD_IMAGE_COLOR );
```

filename Name of file to be loaded.

iscolor Specific color type of the loaded image:

CV_LOAD_IMAGE_COLOR the loaded image is forced to be a 3-channel color image

CV_LOAD_IMAGE_GRAYSCALE the loaded image is forced to be grayscale

CV_LOAD_IMAGE_UNCHANGED the loaded image will be loaded as is.

The function `cvLoadImage` loads an image from the specified file and returns the pointer to the loaded image. Currently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

Note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB.

---

cvLoadImageM (view/add comments)

Loads an image from a file as a CvMat.

```
CvMat* cvLoadImageM(
           const char* filename,
           int iscolor=CV_LOAD_IMAGE_COLOR );
```

filename Name of file to be loaded.

iscolor Specific color type of the loaded image:

CV_LOAD_IMAGE_COLOR the loaded image is forced to be a 3-channel color image

CV_LOAD_IMAGE_GRAYSCALE the loaded image is forced to be grayscale

CV_LOAD_IMAGE_UNCHANGED the loaded image will be loaded as is.

The function `cvLoadImageM` loads an image from the specified file and returns the pointer to the loaded image. urrently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

Note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB.

---

cvSaveImage (view/add comments)

Saves an image to a specified file.

```
int cvSaveImage( const char* filename, const CvArr* image );
```

filename Name of the file.

image Image to be saved.

The function `cvSaveImage` saves the image to the specified file. The image format is chosen based on the `filename` extension, see LoadImage . Only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use `cvCvtScale` and `cvCvtColor` to convert it before saving, or use universal `cvSave` to save the image to XML or YAML format.

CvCapture (view/add comments)

Video capturing structure.

```
typedef struct CvCapture CvCapture;
```

The structure `CvCapture` does not have a public interface and is used only as a parameter for video capturing functions.

---

cvCaptureFromCAM (view/add comments)

Initializes capturing a video from a camera.

```
CvCapture* cvCaptureFromCAM( int index );
```

index Index of the camera to be used. If there is only one camera or it does not matter what camera is used -1 may be passed.

The function `cvCaptureFromCAM` allocates and initializes the CvCapture structure for reading a video stream from the camera. Currently two camera interfaces can be used on Windows: Video for Windows (VFW) and Matrox Imaging Library (MIL); and two on Linux: V4L and FireWire (IEEE1394).

To release the structure, use ReleaseCapture .

---

cvCaptureFromFile (view/add comments)

Initializes capturing a video from a file.

```
CvCapture* cvCaptureFromFile( const char* filename );
```

filename Name of the video file.

The function `cvCaptureFromFile` allocates and initializes the CvCapture structure for reading the video stream from the specified file. Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg

is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect and how to prepare your video files.

After the allocated structure is not used any more it should be released by the ReleaseCapture function.

---

cvGetCaptureProperty (view/add comments)

Gets video capturing properties.

```
double cvGetCaptureProperty( CvCapture* capture, int property_id );
```

capture video capturing structure.

property_id Property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
- **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
- **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
- **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
- **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
- **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
- **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB

CV_CAP_PROP_WHITE_BALANCE Currently unsupported

CV_CAP_PROP_RECTIFICATION TOWRITE (note: only supported by DC1394 v 2.x backend currently)

The function `cvGetCaptureProperty` retrieves the specified property of the camera or video file.

---

cvGrabFrame (view/add comments)

Grabs the frame from a camera or file.

```
int cvGrabFrame( CvCapture* capture );
```

capture video capturing structure.

The function `cvGrabFrame` grabs the frame from a camera or file. The grabbed frame is stored internally. The purpose of this function is to grab the frame *quickly* so that syncronization can occur if it has to read from several cameras simultaneously. The grabbed frames are not exposed because they may be stored in a compressed format (as defined by the camera/driver). To retrieve the grabbed frame, RetrieveFrame should be used.

---

cvQueryFrame (view/add comments)

Grabs and returns a frame from a camera or file.

```
IplImage* cvQueryFrame( CvCapture* capture );
```

capture video capturing structure.

The function `cvQueryFrame` grabs a frame from a camera or video file, decompresses it and returns it. This function is just a combination of GrabFrame and RetrieveFrame , but in one call. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cvReleaseCapture (view/add comments)

Releases the CvCapture structure.

```
void cvReleaseCapture( CvCapture** capture );
```

capture Pointer to video the capturing structure.

The function `cvReleaseCapture` releases the CvCapture structure allocated by CaptureFromFile or CaptureFromCAM .

cvRetrieveFrame (view/add comments)

Gets the image grabbed with cvGrabFrame.

```
IplImage* cvRetrieveFrame( CvCapture* capture );
```

capture video capturing structure.

The function `cvRetrieveFrame` returns the pointer to the image grabbed with the GrabFrame function. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cvSetCaptureProperty (view/add comments)

Sets video capturing properties.

```
int cvSetCaptureProperty(
            CvCapture* capture,
            int property_id,
            double value );
```

capture video capturing structure.

property_id property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
- **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
- **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
- **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
- **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
- **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
- **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB
- **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
- **CV_CAP_PROP_RECTIFICATION** TOWRITE (note: only supported by DC1394 v 2.x backend currently)

value value of the property.

The function `cvSetCaptureProperty` sets the specified property of video capturing. Currently the function supports only video files: `CV_CAP_PROP_POS_MSEC, CV_CAP_PROP_POS_FRAMES, CV_CAP_PROP_POS_AVI_RATIO`.

NB This function currently does nothing when using the latest CVS download on linux with FFMPEG (the function contents are hidden if 0 is used and returned).

cvCreateVideoWriter (view/add comments)

Creates the video file writer.

```
typedef struct CvVideoWriter CvVideoWriter; CvVideoWriter*
cvCreateVideoWriter(
            const char* filename,
            int fourcc,
            double fps,
            CvSize frame_size,
            int is_color=1 );
```

filename Name of the output video file.

fourcc 4-character code of codec used to compress the frames. For example, `CV_FOURCC('P','I','M','1')` is a MPEG-1 codec, `CV_FOURCC('M','J','P','G')` is a motion-jpeg codec etc. Under Win32 it is possible to pass -1 in order to choose compression method and additional compression parameters from dialog. Under Win32 if 0 is passed while using an avi filename it will create a video writer that creates an uncompressed avi file.

fps Framerate of the created video stream.

frame_size Size of the video frames.

is_color If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only).

The function `cvCreateVideoWriter` creates the video writer structure.

Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect.

cvReleaseVideoWriter (view/add comments)

Releases the AVI writer.

```
void cvReleaseVideoWriter( CvVideoWriter** writer );
```

writer Pointer to the video file writer structure.

The function `cvReleaseVideoWriter` finishes writing to the video file and releases the structure.

---

cvWriteFrame (view/add comments)

Writes a frame to a video file.

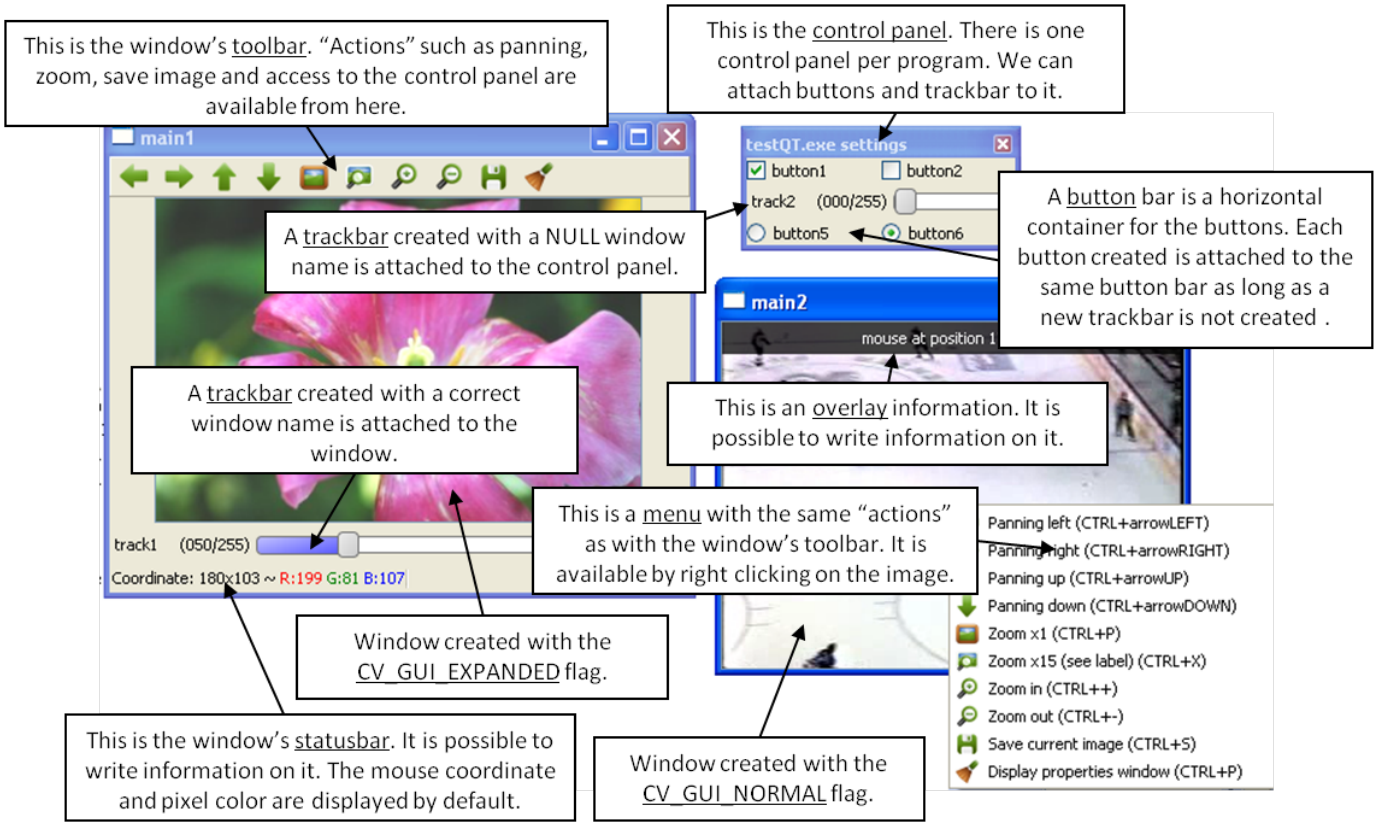
```
int cvWriteFrame( CvVideoWriter* writer, const IplImage* image );
```

writer Video writer structure

image The written frame

The function `cvWriteFrame` writes/appends one frame to a video file.

7.3 Qt new functions



This figure explains the new functionalities implemented with Qt GUI. As we can see, the new GUI provides a statusbar, a toolbar, and a control panel. The control panel can have trackbars and buttonbars attached to it.

- To attach a trackbar, the window_name parameter must be NULL.
- To attach a buttonbar, a button must be created. If the last bar attached to the control panel is a buttonbar, the new button is added on the right of the last button. If the last bar attached to the control panel is a trackbar, or the control panel is empty, a new buttonbar is created. Then a new button is attached to it.

The following code is an example used to generate the figure.

```
int main(int argc, char *argv[])
    int value = 50;
    int value2 = 0;
```

```
    cvNamedWindow("main1",CV_WINDOW_NORMAL);
    cvNamedWindow("main2",CV_WINDOW_AUTOSIZE | CV_GUI_NORMAL);

    cvCreateTrackbar( "track1", "main1", &value, 255,  NULL);//OK tested
    char* nameb1 = "button1";
    char* nameb2 = "button2";
    cvCreateButton(nameb1,callbackButton,nameb1,CV_CHECKBOX,1);

    cvCreateButton(nameb2,callbackButton,nameb2,CV_CHECKBOX,0);
    cvCreateTrackbar( "track2", NULL, &value2, 255, NULL);
    cvCreateButton("button5",callbackButton1,NULL,CV_RADIOBOX,0);
    cvCreateButton("button6",callbackButton2,NULL,CV_RADIOBOX,1);

    cvSetMouseCallback( "main2",on_mouse,NULL );

    IplImage* img1 = cvLoadImage("files/flower.jpg");
    IplImage* img2 = cvCreateImage(cvGetSize(img1),8,3);
    CvCapture* video = cvCaptureFromFile("files/hockey.avi");
    IplImage* img3 = cvCreateImage(cvGetSize(cvQueryFrame(video)),8,3);

    while(cvWaitKey(33) != 27)
    {
        cvAddS(img1,cvScalarAll(value),img2);
        cvAddS(cvQueryFrame(video),cvScalarAll(value2),img3);
        cvShowImage("main1",img2);
        cvShowImage("main2",img3);
    }

    cvDestroyAllWindows();
    cvReleaseImage(&img1);
    cvReleaseImage(&img2);
    cvReleaseImage(&img3);
    cvReleaseCapture(&video);
    return 0;
}
```

---

cvSetWindowProperty (view/add comments)

Change the parameters of the window dynamically.

```
void cvSetWindowProperty(const char* name, int prop_id, double
prop_value);
```

name Name of the window.

prop_id Window's property to edit. The operation flags:

- **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (CV_WINDOW_NORMAL or CV_WINDOW_FULLSCREEN).
- **CV_WND_PROP_AUTOSIZE** Change if the user can resize the window (texttttCV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE).
- **CV_WND_PROP_ASPECTRATIO** Change if the image's aspect ratio is preserved (texttttCV_WINDOW_FREERA or CV_WINDOW_KEEPRATIO).

prop_value New value of the Window's property. The operation flags:

- **CV_WINDOW_NORMAL** Change the window in normal size, or allows the user to resize the window.
- **CV_WINDOW_AUTOSIZE** The user cannot resize the window, the size is constrainted by the image displayed.
- **CV_WINDOW_FULLSCREEN** Change the window to fullscreen.
- **CV_WINDOW_FREERATIO** The image expends as much as it can (no ratio constraint)
- **CV_WINDOW_KEEPRATIO** The ration image is respected.

The function cvSetWindowProperty allows to change the window's properties.

---

cvGetWindowProperty (view/add comments)

Get the parameters of the window.

```
void cvGetWindowProperty(const char* name, int prop_id);
```

name Name of the window.

prop_id Window's property to retrive. The operation flags:

- **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (CV_WINDOW_NORMAL or CV_WINDOW_FULLSCREEN).

CV_WND_PROP_AUTOSIZE Change if the user can resize the window (texttttCV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE).

CV_WND_PROP_ASPECTRATIO Change if the image's aspect ratio is preserved (texttttCV_WINDOW_FREERA or CV_WINDOW_KEEPRATIO).

See SetWindowProperty to know the meaning of the returned values.
The function cvGetWindowProperty return window's properties.

---

cvFontQt (view/add comments)

Create the font to be used to draw text on an image (with addText).

```
CvFont cvFontQt(const char* nameFont, int pointSize = -1, CvScalar
color = cvScalarAll(0), int weight = CV_FONT_NORMAL, int style =
CV_STYLE_NORMAL, int spacing = 0);
```

nameFont Name of the font. The name should match the name of a system font (such as "Times"). If the font is not found, a default one will be used.

pointSize Size of the font. If not specified, equal zero or negative, the point size of the font is set to a system-dependent default value. Generally, this is 12 points.

color Color of the font in BGRA – A = 255 is fully transparent. Use the macro CV_RGB for simplicity.

weight The operation flags:

CV_FONT_LIGHT Weight of 25

CV_FONT_NORMAL Weight of 50

CV_FONT_DEMIBOLD Weight of 63

CV_FONT_BOLD Weight of 75

CV_FONT_BLACK Weight of 87 You can also specify a positive integer for more control.

style The operation flags:

CV_STYLE_NORMAL Font is normal

CV_STYLE_ITALIC Font is in italic

CV_STYLE_OBLIQUE Font is oblique

spacing Spacing between characters. Can be negative or positive

The function `cvFontQt` creates a CvFont object to be used with addText . This CvFont is not compatible with cvPutText.

A basic usage of this function is:

```
CvFont font = cvFontQt(''Times'');
cvAddText( img1, ``Hello World !'', cvPoint(50,50), font);
```

---

cvAddText (view/add comments)

Create the font to be used to draw text on an image

```
void cvAddText(const CvArr* img, const char* text, CvPoint location,
CvFont *font);
```

img Image where the text should be drawn

text Text to write on the image

location Point(x,y) where the text should start on the image

font Font to use to draw the text

The function `cvAddText` draw *text* on the image *img* using a specific font *font* (see example FontQt)

---

cvDisplayOverlay (view/add comments)

Display text on the window's image as an overlay for delay milliseconds. This is not editing the image's data. The text is display on the top of the image.

```
void cvDisplayOverlay(const char* name, const char* text, int delay);
```

name Name of the window

text Overlay text to write on the window's image

delay Delay to display the overlay text. If this function is called before the previous overlay text time out, the timer is restarted and the text updated. . If this value is zero, the text never disapers.

The function `cvDisplayOverlay` aims at displaying useful information/tips on the window for a certain amount of time *delay*. This information is display on the top of the window.

---

cvDisplayStatusBar (view/add comments)

Display text on the window's statusbar as for delay milliseconds.

```
void cvDisplayStatusBar(const char* name, const char* text, int
delayms);
```

name Name of the window

text Text to write on the window's statusbar

delay Delay to display the text. If this function is called before the previous text time out, the timer is restarted and the text updated. If this value is zero, the text never disapers.

The function `cvDisplayOverlay` aims at displaying useful information/tips on the window for a certain amount of time *delay*. This information is displayed on the window's statubar (the window must be created with `CV_GUI_EXPANDED` flags).

---

cvCreateOpenGLCallback (view/add comments)

Create a callback function called to draw OpenGL on top the the image display by *window_name*.

```
void cvCreateOpenGLCallback( const char* window_name, CvOpenGLCallback
callbackOpenGL, void* userdata CV_DEFAULT(NULL), double angle
CV_DEFAULT(-1), double zmin CV_DEFAULT(-1), double zmax CV_DEFAULT(-1);
```

window_name Name of the window

callbackOpenGL Pointer to the function to be called every frame. This function should be prototyped as `void Foo(*void);`.

userdata pointer passed to the callback function. *(Optional)*

angle Specifies the field of view angle, in degrees, in the y direction.. *(Optional - Default 45 degree)*

zmin Specifies the distance from the viewer to the near clipping plane (always positive). *(Optional - Default 0.01)*

zmax Specifies the distance from the viewer to the far clipping plane (always positive). *(Optional - Default 1000)*

The function `cvCreateOpenGLCallback` can be used to draw 3D data on the window. An example of callback could be:

```
void on_opengl(void* param)
{
    //draw scene here
    glLoadIdentity();

    glTranslated(0.0, 0.0, -1.0);

    glRotatef( 55, 1, 0, 0 );
    glRotatef( 45, 0, 1, 0 );
    glRotatef( 0, 0, 0, 1 );

    static const int coords[6][4][3] = {
        { { +1, -1, -1 }, { -1, -1, -1 }, { -1, +1, -1 }, { +1, +1, -1 } },
        { { +1, +1, -1 }, { -1, +1, -1 }, { -1, +1, +1 }, { +1, +1, +1 } },
        { { +1, -1, +1 }, { +1, -1, -1 }, { +1, +1, -1 }, { +1, +1, +1 } },
        { { -1, -1, -1 }, { -1, -1, +1 }, { -1, +1, +1 }, { -1, +1, -1 } },
        { { +1, -1, +1 }, { -1, -1, +1 }, { -1, -1, -1 }, { +1, -1, -1 } },
        { { -1, -1, +1 }, { +1, -1, +1 }, { +1, +1, +1 }, { -1, +1, +1 } }
    };

    for (int i = 0; i < 6; ++i) {
                glColor3ub( i*20, 100+i*10, i*42 );
                glBegin(GL_QUADS);
                for (int j = 0; j < 4; ++j) {
                        glVertex3d(0.2 * coords[i][j][0], 0.2 * coords[i][j][1], 0.2 * coo
                }
```

```
                glEnd();
    }
}
```

```
CV_EXTERN_C_FUNCPTR( *CvOpenGLCallback)(void* userdata));
```

---

cvSaveWindowParameters (view/add comments)

Save parameters of the window *window_name*.

```
void cvSaveWindowParameters(const char* name);
```

name Name of the window

The function `cvSaveWindowParameters` saves size, location, flags, trackbars' value, zoom and panning location of the window *window_name*

---

cvLoadWindowParameters (view/add comments)

Load parameters of the window *window_name*.

```
void cvLoadWindowParameters(const char* name);
```

name Name of the window

The function `cvLoadWindowParameters` load size, location, flags, trackbars' value, zoom and panning location of the window *window_name*

---

cvCreateButton (view/add comments)

Create a callback function called to draw OpenGL on top the the image display by *window_name*.

```
cvCreateButton( const char* button_name CV_DEFAULT(NULL),CvButtonCallback
on_change CV_DEFAULT(NULL), void* userdata CV_DEFAULT(NULL) , int
button_type CV_DEFAULT(CV_PUSH_BUTTON), int initial_button_state
CV_DEFAULT(0);
```

button_name Name of the button *(if NULL, the name will be "button ¡number of boutton¿")*

on_change Pointer to the function to be called every time the button changed its state. This function should be prototyped as `void Foo(int state,*void);`. *state* is the current state of the button. It could be -1 for a push button, 0 or 1 for a check/radio box button.

userdata pointer passed to the callback function. *(Optional)*

The `button_type` parameter can be : *(Optional – Will be a push button by default.)*

CV_PUSH_BUTTON *The button will be a push button.*

CV_CHECKBOX *The button will be a checkbox button.*

CV_RADIOBOX *The button will be a radiobox button. The radiobox on the same buttonbar (same line) are exclusive; one on can be select at the time.*

initial_button_state Default state of the button. Use for checkbox and radiobox, its value could be 0 or 1. *(Optional)*

The function `cvCreateButton` attach button to the control panel. Each button is added to a buttonbar on the right of the last button. A new buttonbar is create if nothing was attached to the control panel before, or if the last element attached to the control panel was a trackbar.

Here are various example of `cvCreateButton` function call:

```
cvCreateButton(NULL,callbackButton);//create a push button "button 0", that will call callb
cvCreateButton("button2",callbackButton,NULL,CV\_CHECKBOX,0);
cvCreateButton("button3",callbackButton,&value);
cvCreateButton("button5",callbackButton1,NULL,CV\_RADIOBOX);
cvCreateButton("button6",callbackButton2,NULL,CV\_PUSH\_BUTTON,1);
```

```
CV_EXTERN_C_FUNCPTR( *CvButtonCallback)(int state, void* userdata));
```

Chapter 8

calib3d. Camera Calibration, Pose Estimation and Stereo

8.1 Camera Calibration and 3d Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re-used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distortion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$
where $r^2 = x'^2 + y'^2$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

k_1, k_2, k_3, k_4, k_5, k_6 are radial distortion coefficients, p_1, p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters. *And they remain the same regardless of the captured image resolution.* That is, if, for example, a camera has been calibrated on images of 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x, f_y, c_x and c_y need to be scaled appropriately).

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspondences).

- Estimate the relative position and orientation of the stereo camera ”heads” and compute the *rectification* transformation that makes the camera optical axes parallel.

---

cvCalcImageHomography (view/add comments)

Calculates the homography matrix for an oblong planar object (e.g. arm).

```
void cvCalcImageHomography(
           float* line,
           CvPoint3D32f* center,
           float* intrinsic,
           float* homography );
```

line the main object axis direction (vector (dx,dy,dz))

center object center ((cx,cy,cz))

intrinsic intrinsic camera parameters (3x3 matrix)

homography output homography matrix (3x3)

The function calculates the homography matrix for the initial image transformation from image plane to the plane, defined by a 3D oblong object line (See __Figure 6-10__ in the OpenCV Guide 3D Reconstruction Chapter).

---

cvCalibrateCamera2 (view/add comments)

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
double cvCalibrateCamera2(
           const CvMat* objectPoints,
           const CvMat* imagePoints,
           const CvMat* pointCounts,
           CvSize imageSize,
           CvMat* cameraMatrix,
           CvMat* distCoeffs,
```

```
            CvMat* rvecs=NULL,
            CvMat* tvecs=NULL,
            int flags=0 );
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints The joint matrix of object points projections in the camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints` (=N).

imageSize Size of the image, used only to initialize the intrinsic camera matrix

cameraMatrix The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized before calling the function

distCoeffs The output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements

rvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of rotation vectors (see cvRodrigues2), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0..*M*-1)

tvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of translation vectors, estimated for each pattern view.

flags Different flags, may be 0 or combination of the following values:

CV_CALIB_USE_INTRINSIC_GUESS `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the

image center (`imageSize` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate the extrinsic parameters. Use cvFindExtrinsicCameraParams2 instead.

CV_CALIB_FIX_PRINCIPAL_POINT The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.

CV_CALIB_FIX_ASPECT_RATIO The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix`.
When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.

CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6 Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.

CV_CALIB_RATIONAL_MODEL Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute only 5 distortion coefficients.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see cvFindChessboardCorners). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).

2. The initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using cvFindExtrinsicCameraParams2

3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points

imagePoints and the projected (using the current estimates for camera parameters and the poses) object points objectPoints; see cvProjectPoints2.

The function returns the final re-projection error.

Note: if you're using a non-square (=non-NxN) grid and cv for calibration, and calibrateCamera returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2-0.5, h/2-0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probably used patternSize=cvSize(rows,cols), but should use patternSize=cvSize(cols,rows) in cvFindChessboardCorners.

See also: cvFindChessboardCorners, cvFindExtrinsicCameraParams2, cv, cvStereoCalibrate, cvUndistort2

---

cvComputeCorrespondEpilines (view/add comments)

For points in one image of a stereo pair, computes the corresponding epilines in the other image.

```
void cvComputeCorrespondEpilines(
            const CvMat* points,
            int whichImage,
            const CvMat* F,
            CvMat* lines);
```

points The input points. 2xN, Nx2, 3xN or Nx3 array (where N number of points). Multi-channel 1xN or Nx1 array is also acceptable

whichImage Index of the image (1 or 2) that contains the points

F The fundamental matrix that can be estimated using cvFindFundamentalMat or cvStereoRectify.

lines The output epilines, a 3xN or Nx3 array. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see cvFindFundamentalMat), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when whichImage=1) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

---

cvConvertPointsHomogeneous (view/add comments)

Convert points to/from homogeneous coordinates.

```
void cvConvertPointsHomogeneous(
           const CvMat* src,
           CvMat* dst );
```

src The input point array, `2xN, Nx2, 3xN, Nx3, 4xN or Nx4 (where N is the number of points)`. Multi-channel `1xN` or `Nx1` array is also acceptable

dst The output point array, must contain the same number of points as the input; The dimensionality must be the same, 1 less or 1 more than the input, and also within 2 to 4

The function converts 2D or 3D points from/to homogeneous coordinates, or simply copies or transposes the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$x' = x/w$
$y' = y/w$
$z' = z/w$ (if output is 3D)

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

Note because the function accepts a large variety of array layouts, it may report an error when input/output array dimensionality is ambiguous. It is always safe to use the function with number of points $\texttt{N} \geq 5$, or to use multi-channel `Nx1` or `1xN` arrays.

cvCreatePOSITObject (view/add comments)

Initializes a structure containing object information.

```
CvPOSITObject* cvCreatePOSITObject(
           CvPoint3D32f* points,
           int point_count );
```

points Pointer to the points of the 3D object model

point_count Number of object points

The function allocates memory for the object structure and computes the object inverse matrix.

The preprocessed object data is stored in the structure cvCvPOSITObject, internal for OpenCV, which means that the user cannot directly access the structure data. The user may only create this structure and pass its pointer to the function.

An object is defined as a set of points given in a coordinate system. The function cvPOSIT computes a vector that begins at a camera-related coordinate system center and ends at the `points[0]` of the object.

Once the work with a given object is finished, the function cvReleasePOSITObject must be called to free memory.

cvCreateStereoBMState (view/add comments)

Creates block matching stereo correspondence structure.

```
CvStereoBMState* cvCreateStereoBMState( int preset=CV_STEREO_BM_BASIC,
int numberOfDisparities=0 );
```

preset ID of one of the pre-defined parameter sets. Any of the parameters can be overridden after creating the structure. Values are

CV_STEREO_BM_BASIC Parameters suitable for general cameras

CV_STEREO_BM_FISH_EYE Parameters suitable for wide-angle cameras

CV_STEREO_BM_NARROW Parameters suitable for narrow-angle cameras

numberOfDisparities The number of disparities. If the parameter is 0, it is taken from the preset, otherwise the supplied value overrides the one from preset.

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cvFindStereoCorrespondenceBM.

---

cvCreateStereoGCState (view/add comments)

Creates the state of graph cut-based stereo correspondence algorithm.

```
CvStereoGCState* cvCreateStereoGCState( int numberOfDisparities, int
maxIters );
```

numberOfDisparities The number of disparities. The disparity search range will be `state->minDisparity` $\leq disparity <$ `state->minDisparity` + `state->numberOfDisparities`

maxIters Maximum number of iterations. On each iteration all possible (or reasonable) alpha-expansions are tried. The algorithm may terminate earlier if it could not find an alpha-expansion that decreases the overall cost function value. See [**?**] for details.

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cvFindStereoCorrespondenceGC.

CvStereoBMState (view/add comments)

The structure for block matching stereo correspondence algorithm.

```
typedef struct CvStereoBMState
{
    //pre filters (normalize input images):
    int       preFilterType; // 0 for now
    int       preFilterSize; // ~5x5..21x21
    int       preFilterCap;  // up to ~31
    //correspondence using Sum of Absolute Difference (SAD):
    int       SADWindowSize; // Could be 5x5..21x21
    int       minDisparity;  // minimum disparity (=0)
    int       numberOfDisparities; // maximum disparity - minimum disparity
    //post filters (knock out bad matches):
    int       textureThreshold; // areas with no texture are ignored
    int       uniquenessRatio;// invalidate disparity at pixels where there are other close
```

```
                                // with different disparity
    int       speckleWindowSize; // the maximum area of speckles to remove
                                 // (set to 0 to disable speckle filtering)
    int       speckleRange; // acceptable range of disparity variation in each connected co

    int trySmallerWindows; // not used
    CvRect roi1, roi2; // clipping ROIs

    int disp12MaxDiff; // maximum allowed disparity difference in the left-right check

    // internal data
    ...
}
CvStereoBMState;
```

preFilterType type of the prefilter, CV_STEREO_BM_NORMALIZED_RESPONSE or the default and the recommended CV_STEREO_BM_XSOBEL, int

preFilterSize 5x5..21x21, int

preFilterCap up to 31, int

SADWindowSize Could be 5x5..21x21 or higher, but with 21x21 or smaller windows the processing speed is much higher, int

minDisparity minimum disparity (=0), int

numberOfDisparities maximum disparity - minimum disparity, int

textureThreshold the textureness threshold. That is, if the sum of absolute values of x-derivatives computed over SADWindowSize by SADWindowSize pixel neighborhood is smaller than the parameter, no disparity is computed at the pixel, int

uniquenessRatio the minimum margin in percents between the best (minimum) cost function value and the second best value to accept the computed disparity, int

speckleWindowSize the maximum area of speckles to remove (set to 0 to disable speckle filtering), int

speckleRange acceptable range of disparity variation in each connected component, int

trySmallerWindows not used currently (0), int

roi1, roi2 These are the clipping ROIs for the left and the right images. The function cvStereoRectify returns the largest rectangles in the left and right images where after the rectification all the pixels are valid. If you copy those rectangles to the `CvStereoBMState` structure, the stereo correspondence function will automatically clear out the pixels outside of the "valid" disparity rectangle computed by cvGetValidDisparityROI. Thus you will get more "invalid disparity" pixels than usual, but the remaining pixels are more probable to be valid.

disp12MaxDiff The maximum allowed difference between the explicitly computed left-to-right disparity map and the implicitly (by cvValidateDisparity) computed right-to-left disparity. If for some pixel the difference is larger than the specified threshold, the disparity at the pixel is invalidated. By default this parameter is set to (-1), which means that the left-right check is not performed.

The block matching stereo correspondence algorithm, by Kurt Konolige, is very fast single-pass stereo matching algorithm that uses sliding sums of absolute differences between pixels in the left image and the pixels in the right image, shifted by some varying amount of pixels (from `minDisparity` to `minDisparity+numberOfDisparities`). On a pair of images WxH the algorithm computes disparity in `O(W*H*numberOfDisparities)` time. In order to improve quality and readability of the disparity map, the algorithm includes pre-filtering and post-filtering procedures.

Note that the algorithm searches for the corresponding blocks in x direction only. It means that the supplied stereo pair should be rectified. Vertical stereo layout is not directly supported, but in such a case the images could be transposed by user.

CvStereoGCState (view/add comments)

The structure for graph cuts-based stereo correspondence algorithm

```
typedef struct CvStereoGCState
{
    int Ithreshold; // threshold for piece-wise linear data cost function (5 by default)
    int interactionRadius; // radius for smoothness cost function (1 by default; means Pott
    float K, lambda, lambda1, lambda2; // parameters for the cost function
                                       // (usually computed adaptively from the input data)
    int occlusionCost; // 10000 by default
    int minDisparity; // 0 by default; see CvStereoBMState
    int numberOfDisparities; // defined by user; see CvStereoBMState
    int maxIters; // number of iterations; defined by user.

    // internal buffers
    CvMat* left;
    CvMat* right;
```

```
    CvMat* dispLeft;
    CvMat* dispRight;
    CvMat* ptrLeft;
    CvMat* ptrRight;
    CvMat* vtxBuf;
    CvMat* edgeBuf;
}
CvStereoGCState;
```

The graph cuts stereo correspondence algorithm, described in [**?**] (as **KZ1**), is non-realtime stereo correspondence algorithm that usually gives very accurate depth map with well-defined object boundaries. The algorithm represents stereo problem as a sequence of binary optimization problems, each of those is solved using maximum graph flow algorithm. The state structure above should not be allocated and initialized manually; instead, use cvCreateStereoGCState and then override necessary parameters if needed.

---

cvDecomposeProjectionMatrix (view/add comments)

Decomposes the projection matrix into a rotation matrix and a camera matrix.

```
void cvDecomposeProjectionMatrix(
            const CvMat *projMatrix,
            CvMat *cameraMatrix,
            CvMat *rotMatrix,
            CvMat *transVect,
            CvMat *rotMatrX=NULL,
            CvMat *rotMatrY=NULL,
            CvMat *rotMatrZ=NULL,
            CvPoint3D64f *eulerAngles=NULL);
```

projMatrix The 3x4 input projection matrix P

cameraMatrix The output 3x3 camera matrix K

rotMatrix The output 3x3 external rotation matrix R

transVect The output 4x1 translation vector T

rotMatrX Optional 3x3 rotation matrix around x-axis

rotMatrY Optional 3x3 rotation matrix around y-axis

rotMatrZ Optional 3x3 rotation matrix around z-axis

eulerAngles Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on cvRQDecomp3x3.

---

cvDrawChessboardCorners (view/add comments)

Renders the detected chessboard corners.

```
void cvDrawChessboardCorners(
           CvArr* image,
           CvSize patternSize,
           CvPoint2D32f* corners,
           int count,
           int patternWasFound );
```

image The destination image; it must be an 8-bit color image

patternSize The number of inner corners per chessboard row and column. (patternSize = cv::Size(points_per_row,points_per_column) = cv::Size(rows,columns))

corners The array of corners detected, this should be the output from findChessboardCorners wrapped in a cv::Mat().

count The number of corners

patternWasFound Indicates whether the complete board was found $(\neq 0)$ or not $(= 0)$. One may just pass the return value cvFindChessboardCornersfindChessboardCorners here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

cvFindChessboardCorners (view/add comments)

Finds the positions of the internal corners of the chessboard.

```
int cvFindChessboardCorners(
           const void* image,
           CvSize patternSize,
           CvPoint2D32f* corners,
           int* cornerCount=NULL,
           int flags=CV_CALIB_CB_ADAPTIVE_THRESH );
```

image Source chessboard view; it must be an 8-bit grayscale or color image

patternSize The number of inner corners per chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The output array of corners detected

cornerCount The output corner counter. If it is not NULL, it stores the number of corners found

flags Various operation flags, can be 0 or a combination of the following values:

- **CV_CALIB_CB_ADAPTIVE_THRESH** use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
- **CV_CALIB_CB_NORMALIZE_IMAGE** normalize the image gamma with cvEqualizeHist before applying fixed or adaptive thresholding.
- **CV_CALIB_CB_FILTER_QUADS** use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.
- **CALIB_CB_FAST_CHECK** Runs a fast check on the image that looks for chessboard corners, and shortcuts the call if none are found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where

the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cvFindCornerSubPix.

Sample usage of detecting and drawing chessboard corners:

```
Size patternsize(8,6); //interior number of corners
Mat gray = ....; //source image
vector<Point2f> corners; //this will be filled by the detected corners

//CALIB_CB_FAST_CHECK saves a lot of time on images
//that don't contain any chessboard corners
bool patternfound = findChessboardCorners(gray, patternsize, corners,
        CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE
        + CALIB_CB_FAST_CHECK);

if(patternfound)
  cornerSubPix(gray, corners, Size(11, 11), Size(-1, -1),
    TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

drawChessboardCorners(img, patternsize, Mat(corners), patternfound);
```

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

---

cvFindExtrinsicCameraParams2 (view/add comments)

Finds the object pose from the 3D-2D point correspondences

```
void cvFindExtrinsicCameraParams2(
           const CvMat* objectPoints,
           const CvMat* imagePoints,
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           CvMat* rvec,
           CvMat* tvec,
           int useExtrinsicGuess=0);
```

objectPoints The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points.

imagePoints The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points.

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

rvec The output rotation vector (see cvRodrigues2) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system

tvec The output translation vector

useExtrinsicGuess If true (1), the function will use the provided `rvec` and `tvec` as the initial approximations of the rotation and translation vectors, respectively, and will further optimize them.

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using cvProjectPoints2) `objectPoints`.

The function's counterpart in the C++ API is cv::solvePnP

---

cvFindFundamentalMat (view/add comments)

Calculates the fundamental matrix from the corresponding points in two images.

```
int cvFindFundamentalMat(
           const CvMat* points1,
           const CvMat* points2,
           CvMat* fundamentalMatrix,
           int method=CV_FM_RANSAC,
           double param1=1.,
           double param2=0.99,
           CvMat* status=NULL);
```

points1 Array of `N` points from the first image. It can be `2xN, Nx2, 3xN` or `Nx3` 1-channel array or `1xN` or `Nx1` 2- or 3-channel array . The point coordinates should be floating-point (single or double precision)

points2 Array of the second image points of the same size and format as `points1`

fundamentalMatrix The output fundamental matrix or matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices)

method Method for computing the fundamental matrix

CV_FM_7POINT for a 7-point algorithm. $N = 7$

CV_FM_8POINT for an 8-point algorithm. $N \geq 8$

CV_FM_RANSAC for the RANSAC algorithm. $N \geq 8$

CV_FM_LMEDS for the LMedS algorithm. $N \geq 8$

param1 The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise

param2 The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct

status The optional output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1's

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the number of fundamental matrices found (1 or 3) and 0, if no matrix is found . Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to cvComputeCorrespondEpilines that finds the epipolar lines corresponding to the specified points. It can also be passed to cvStereoRectifyUncalibrated to compute the rectification transformation.

```
int point_count = 100;
CvMat* points1;
CvMat* points2;
CvMat* status;
CvMat* fundamental_matrix;

points1 = cvCreateMat(1,point_count,CV_32FC2);
points2 = cvCreateMat(1,point_count,CV_32FC2);
status = cvCreateMat(1,point_count,CV_8UC1);

/* Fill the points here ... */
for( i = 0; i < point_count; i++ )
{
    points1->data.fl[i*2] = <x,,1,i,,>;
    points1->data.fl[i*2+1] = <y,,1,i,,>;
    points2->data.fl[i*2] = <x,,2,i,,>;
    points2->data.fl[i*2+1] = <y,,2,i,,>;
}

fundamental_matrix = cvCreateMat(3,3,CV_32FC1);
int fm_count = cvFindFundamentalMat( points1,points2,fundamental_matrix,
                                     CV_FM_RANSAC,1.0,0.99,status );
```

---

cvFindHomography (view/add comments)

Finds the perspective transformation between two planes.

```
void cvFindHomography(
            const CvMat* srcPoints,
            const CvMat* dstPoints,
            CvMat* H
            int method=0,
            double ransacReprojThreshold=3,
            CvMat* status=NULL);
```

srcPoints Coordinates of the points in the original plane, 2xN, Nx2, 3xN or Nx3 1-channel array (the latter two are for representation in homogeneous coordinates), where N is the number of points. 1xN or Nx1 2- or 3-channel array can also be passed.

dstPoints Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 1-channel, or 1xN or Nx1 2- or 3-channel array.

H The output 3x3 homography matrix

method The method used to computed homography matrix; one of the following:

0 a regular method using all the points

CV_RANSAC RANSAC-based robust method

CV_LMEDS Least-Median robust method

ransacReprojThreshold The maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.

status The optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). *Note that the input mask values are ignored.*

The function finds the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}} \right)^2$$

is minimized. If the parameter `method` is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, `RANSAC` and `LMeDS`, try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection error

for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method `RANSAC` can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method `LMeDS` does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features, where can be only some small noise present, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized so that $h_{33} = 1$.

See also: cvGetAffineTransform, cvGetPerspectiveTransform, cvEstimateRigidMotion, cvWarpPerspective, cvPerspectiveTransform

---

cvFindStereoCorrespondenceBM (view/add comments)

Computes the disparity map using block matching algorithm.

```
void cvFindStereoCorrespondenceBM(
           const CvArr* left,
           const CvArr* right,
           CvArr* disparity,
           CvStereoBMState* state );
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

disparity The output single-channel 16-bit signed, or 32-bit floating-point disparity map of the same size as input images. In the first case the computed disparities are represented as fixed-point numbers with 4 fractional bits (i.e. the computed disparity values are multiplied by 16 and rounded to integers).

state Stereo correspondence structure.

The function cvFindStereoCorrespondenceBM computes disparity map for the input rectified stereo pair. Invalid pixels (for which disparity can not be computed) are set to `state->minDisparity - 1` (or to `(state->minDisparity-1)*16` in the case of 16-bit fixed-point disparity map)

cvFindStereoCorrespondenceGC (view/add comments)

Computes the disparity map using graph cut-based algorithm.

```
void cvFindStereoCorrespondenceGC(
           const CvArr* left,
           const CvArr* right,
           CvArr* dispLeft,
           CvArr* dispRight,
           CvStereoGCState* state,
           int useDisparityGuess = CV_DEFAULT(0) );
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

dispLeft The optional output single-channel 16-bit signed left disparity map of the same size as input images.

dispRight The optional output single-channel 16-bit signed right disparity map of the same size as input images.

state Stereo correspondence structure.

useDisparityGuess If the parameter is not zero, the algorithm will start with pre-defined disparity maps. Both dispLeft and dispRight should be valid disparity maps. Otherwise, the function starts with blank disparity maps (all pixels are marked as occlusions).

The function computes disparity maps for the input rectified stereo pair. Note that the left disparity image will contain values in the following range:

$$-\texttt{state->numberOfDisparities}-\texttt{state->minDisparity} < dispLeft(x,y) \leq -\texttt{state->minDisparity}$$

or

$$dispLeft(x,y) == \texttt{CV_STEREO_GC_OCCLUSION}$$

and for the right disparity image the following will be true:

$$\texttt{state->minDisparity} \leq dispRight(x,y) < \texttt{state->minDisparity}+\texttt{state->numberOfDisparities}$$

or

$$dispRight(x,y) == \texttt{CV_STEREO_GC_OCCLUSION}$$

that is, the range for the left disparity image will be inversed, and the pixels for which no good match has been found, will be marked as occlusions.

Here is how the function can be used:

```
// image_left and image_right are the input 8-bit single-channel images
// from the left and the right cameras, respectively
CvSize size = cvGetSize(image_left);
CvMat* disparity_left = cvCreateMat( size.height, size.width, CV_16S );
CvMat* disparity_right = cvCreateMat( size.height, size.width, CV_16S );
CvStereoGCState* state = cvCreateStereoGCState( 16, 2 );
cvFindStereoCorrespondenceGC( image_left, image_right,
    disparity_left, disparity_right, state, 0 );
cvReleaseStereoGCState( &state );
// now process the computed disparity images as you want ...
```

and this is the output left disparity image computed from the well-known Tsukuba stereo pair and multiplied by -16 (because the values in the left disparity images are usually negative):

```
CvMat* disparity_left_visual = cvCreateMat( size.height, size.width, CV_8U );
cvConvertScale( disparity_left, disparity_left_visual, -16 );
cvSave( "disparity.pgm", disparity_left_visual );
```

cvGetOptimalNewCameraMatrix (view/add comments)

Returns the new camera matrix based on the free scaling parameter

```
void cvGetOptimalNewCameraMatrix(
           const CvMat* cameraMatrix, const CvMat* distCoeffs,
           CvSize imageSize, double alpha,
           CvMat* newCameraMatrix,
           CvSize newImageSize=cvSize(0,0),
           CvRect* validPixROI=0 );
```

cameraMatrix The input camera matrix

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imageSize The original image size

alpha The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see cvStereoRectify

newCameraMatrix The output new camera matrix.

newImageSize The image size after rectification. By default it will be set to `imageSize`.

validPixROI The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in cvStereoRectify

The function computes the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0`, keep all the original image pixels if there is valuable information in the corners `alpha=1`, or get something in between. When `alpha>0`, the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to cvInitUndistortRectifyMap to produce the maps for cvRemap.

---

cvInitIntrinsicParams2D (view/add comments)

Finds the initial camera matrix from the 3D-2D point correspondences

```
void cvInitIntrinsicParams2D(
           const CvMat* objectPoints,
           const CvMat* imagePoints,
           const CvMat* npoints, CvSize imageSize,
           CvMat* cameraMatrix,
           double aspectRatio=1.);
```

objectPoints The joint array of object points; see cvCalibrateCamera2

imagePoints The joint array of object point projections; see cvCalibrateCamera2

npoints The array of point counts; see cvCalibrateCamera2

imageSize The image size in pixels; used to initialize the principal point

cameraMatrix The output camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

aspectRatio If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

---

cvInitUndistortMap (view/add comments)

Computes an undistortion map.

```
void cvInitUndistortMap(
            const CvMat* cameraMatrix,
            const CvMat* distCoeffs,
            CvArr* map1,
            CvArr* map2 );
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function is a simplified variant of cvInitUndistortRectifyMap where the rectification transformation `R` is identity matrix and `newCameraMatrix=cameraMatrix`.

cvInitUndistortRectifyMap (view/add comments)

Computes the undistortion and rectification transformation map.

```
void cvInitUndistortRectifyMap(
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           const CvMat* R,
           const CvMat* newCameraMatrix,
           CvArr* map1,
           CvArr* map2 );
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R The optional rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cvStereoRectify can be passed here. If the matrix is NULL , the identity transformation is assumed

newCameraMatrix The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for cvRemap. The undistorted image will look like the original, as if it was captured with a camera with camera matrix `=newCameraMatrix` and zero distortion. In the case of monocular camera `newCameraMatrix` is usually equal to `cameraMatrix`, or it can be computed by cvGetOptimalNewCameraMatrix for a better control over scaling. In the case of stereo camera `newCameraMatrix` is normally set to `P1` or `P2` computed by cvStereoRectify.

Also, this new camera will be oriented differently in the coordinate space, according to `R`. That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images

become horizontal and have the same y- coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by cvRemap. That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$
\begin{aligned}
&x \leftarrow (u - {c'}_x)/{f'}_x \\
&y \leftarrow (v - {c'}_y)/{f'}_y \\
&[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\
&x' \leftarrow X/W \\
&y' \leftarrow Y/W \\
&x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
&y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
&map_x(u, v) \leftarrow x" f_x + c_x \\
&map_y(u, v) \leftarrow y" f_y + c_y
\end{aligned}
$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after cvStereoRectify, which in its turn is called after cvStereoCalibrate. But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using cvStereoRectifyUncalibrated. For each camera the function computes homography `H` as the rectification transformation in pixel domain, not a rotation matrix `R` in 3D space. The `R` can be computed from `H` as

$$\texttt{R} = \texttt{cameraMatrix}^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

---

cvPOSIT (view/add comments)

Implements the POSIT algorithm.

```
void cvPOSIT(
           CvPOSITObject* posit_object,
           CvPoint2D32f* imagePoints,
           double focal_length,
           CvTermCriteria criteria,
           CvMatr32f rotationMatrix,
           CvVect32f translation_vector );
```

posit_object Pointer to the object structure

imagePoints Pointer to the object points projections on the 2D image plane

focal_length Focal length of the camera used

criteria Termination criteria of the iterative POSIT algorithm

rotationMatrix Matrix of rotations

translation_vector Translation vector

The function implements the POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using the camera calibration functions. At every iteration of the algorithm a new perspective projection of the estimated pose is computed.

Difference norm between two projections is the maximal distance between corresponding points. The parameter `criteria.epsilon` serves to stop the algorithm if the difference is small.

---

cvProjectPoints2 (view/add comments)

Project 3D points on to an image plane.

```
void cvProjectPoints2(
           const CvMat* objectPoints,
           const CvMat* rvec,
           const CvMat* tvec,
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           CvMat* imagePoints,
           CvMat* dpdrot=NULL,
           CvMat* dpdt=NULL,
           CvMat* dpdf=NULL,
           CvMat* dpdc=NULL,
           CvMat* dpddist=NULL );
```

objectPoints The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel , where N is the number of points in the view

rvec The rotation vector, see cvRodrigues2

tvec The translation vector

cameraMatrix The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imagePoints The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel

dpdrot Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector

dpdt Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector

dpdf Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y

dpdc Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y

dpddist Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in cvCalibrateCamera2, cvFindExtrinsicCameraParams2 and cvStereoCalibrate. The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

---

cvReprojectImageTo3D (view/add comments)

Reprojects disparity image to 3D space.

```
void cvReprojectImageTo3D( const CvArr* disparity,
           CvArr* _3dImage, const CvMat* Q,
           int handleMissingValues=0);
```

disparity The input single-channel 16-bit signed or 32-bit floating-point disparity image

_3dImage The output 3-channel floating-point image of the same size as `disparity`. Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)`, computed from the disparity map.

Q The 4×4 perspective transformation matrix that can be obtained with cvStereoRectify

handleMissingValues If true, when the pixels with the minimal disparity (that corresponds to the outliers; see cvFindStereoCorrespondenceBM) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \mathtt{Q} * [x\ y\ \mathtt{disparity}(x,y)\ 1]^T$$
$$\mathtt{_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by cvStereoRectify. To reproject a sparse set of points (x,y,d),... to 3D space, use cvPerspectiveTransform.

---

cvRQDecomp3x3 (view/add comments)

Computes the 'RQ' decomposition of 3x3 matrices.

```
void cvRQDecomp3x3(
           const CvMat *M,
           CvMat *R,
           CvMat *Q,
           CvMat *Qx=NULL,
           CvMat *Qy=NULL,
           CvMat *Qz=NULL,
           CvPoint3D64f *eulerAngles=NULL);
```

M The 3x3 input matrix

R The output 3x3 upper-triangular matrix

Q The output 3x3 orthogonal matrix

Qx Optional 3x3 rotation matrix around x-axis

Qy Optional 3x3 rotation matrix around y-axis

Qz Optional 3x3 rotation matrix around z-axis

eulerAngles Optional three Euler angles of rotation

The function computes a RQ decomposition using the given rotations. This function is used in cvDecomposeProjectionMatrix to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

---

cvReleasePOSITObject (view/add comments)

Deallocates a 3D object structure.

```
void cvReleasePOSITObject(
           CvPOSITObject** posit_object );
```

posit_object Double pointer to `CvPOSIT` structure

The function releases memory previously allocated by the function cvCreatePOSITObject.

---

cvReleaseStereoBMState (view/add comments)

Releases block matching stereo correspondence structure.

```
void cvReleaseStereoBMState( CvStereoBMState** state );
```

state Double pointer to the released structure.

The function releases the stereo correspondence structure and all the associated internal buffers.

---

cvReleaseStereoGCState (view/add comments)

Releases the state structure of the graph cut-based stereo correspondence algorithm.

```
void cvReleaseStereoGCState( CvStereoGCState** state );
```

state Double pointer to the released structure.

The function releases the stereo correspondence structure and all the associated internal buffers.

---

cvRodrigues2 (view/add comments)

Converts a rotation matrix to a rotation vector or vice versa.

```
int cvRodrigues2(
            const CvMat* src,
            CvMat* dst,
            CvMat* jacobian=0 );
```

src The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)

dst The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively

jacobian Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like cvCalibrateCamera2, cvStereoCalibrate or cvFindExtrinsicCameraParams2.

---

cvStereoCalibrate (view/add comments)

Calibrates stereo camera.

```
double cvStereoCalibrate(
           const CvMat* objectPoints,
           const CvMat* imagePoints1,
           const CvMat* imagePoints2,
           const CvMat* pointCounts,
           CvMat* cameraMatrix1,
           CvMat* distCoeffs1,
           CvMat* cameraMatrix2,
           CvMat* distCoeffs2,
           CvSize imageSize,
           CvMat* R,
           CvMat* T,
           CvMat* E=0,
           CvMat* F=0,
           CvTermCriteria term_crit=cvTermCriteria(
           CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,30,1e-6),
           int flags=CV_CALIB_FIX_INTRINSIC );
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints1 The joint matrix of object points projections in the first camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

imagePoints2 The joint matrix of object points projections in the second camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints*` (=N).

cameraMatrix1 The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of `CV_CALIB_USE_INTRINSIC_GUESS`, `CV_CALIB_FIX_ASPECT_RATIO`, `CV_CALIB_FIX_INTRINSIC` or `CV_CALIB_FIX_FOCAL_LENGTH` are specified, some or all of the matrices' components must be initialized; see the flags description

distCoeffs The input/output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements.

cameraMatrix2 The input/output second camera matrix, as cameraMatrix1.

distCoeffs2 The input/output lens distortion coefficients for the second camera, as `distCoeffs1`.

imageSize Size of the image, used only to initialize intrinsic camera matrix.

R The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The output translation vector between the cameras' coordinate systems.

E The optional output essential matrix.

F The optional output fundamental matrix.

term_crit The termination criteria for the iterative optimization algorithm.

flags Different flags, may be 0 or combination of the following values:

CV_CALIB_FIX_INTRINSIC If it is set, `cameraMatrix?`, as well as `distCoeffs?` are fixed, so that only `R, T, E` and `F` are estimated.

CV_CALIB_USE_INTRINSIC_GUESS The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.

CV_CALIB_FIX_PRINCIPAL_POINT The principal points are fixed during the optimization.

CV_CALIB_FIX_FOCAL_LENGTH $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.

CV_CALIB_FIX_ASPECT_RATIO $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.

CV_CALIB_SAME_FOCAL_LENGTH Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients for each camera are set to zeros and fixed there.

CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6 Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.

CV_CALIB_RATIONAL_MODEL Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute only 5 distortion coefficients.

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with cvFindExtrinsicCameraParams2), obviously, those poses will relate to each other, i.e. given (R_1, T_1) it should be possible to compute (R_2, T_2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector T: $T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform full calibration of each of the 2 cameras. However, because of the high dimensionality of the parameter space and noise in

the input data the function can diverge from the correct solution. Thus, if intrinsic parameters can be estimated with high accuracy for each of the cameras individually (e.g. using cvCalibrateCamera2), it is recommended to do so and then pass `CV_CALIB_FIX_INTRINSIC` flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are estimated at once, it makes sense to restrict some parameters, e.g. pass `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which are usually reasonable assumptions.

Similarly to cvCalibrateCamera2, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

---

cvStereoRectify (view/add comments)

Computes rectification transforms for each head of a calibrated stereo camera.

```
void cvStereoRectify(
           const CvMat* cameraMatrix1, const CvMat* cameraMatrix2,
           const CvMat* distCoeffs1, const CvMat* distCoeffs2,
           CvSize imageSize, const CvMat* R, const CvMat* T,
           CvMat* R1, CvMat* R2, CvMat* P1, CvMat* P2,
           CvMat* Q=0, int flags=CV_CALIB_ZERO_DISPARITY,
           double alpha=-1, CvSize newImageSize=cvSize(0,0),
           CvRect* roi1=0, CvRect* roi2=0);
```

cameraMatrix1, cameraMatrix2 The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs1, distCoeffs2

distCoeffs The input vectors of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements each. If the vectors are NULL/empty, the zero distortion coefficients are assumed.

imageSize Size of the image used for stereo calibration.

R The rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The translation vector between the cameras' coordinate systems.

R1, R2 The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.

P1, P2 The output 3×4 projection matrices in the new (rectified) coordinate systems.

Q The output 4×4 disparity-to-depth mapping matrix, see cv.

flags The operation flags; may be 0 or `CV_CALIB_ZERO_DISPARITY`. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.

alpha The free scaling parameter. If it is -1 , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.

newImageSize The new image resolution after rectification. The same size should be passed to cvInitUndistortRectifyMap, see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.

roi1, roi2 The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0`, the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by cv and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

1. Horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. Vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

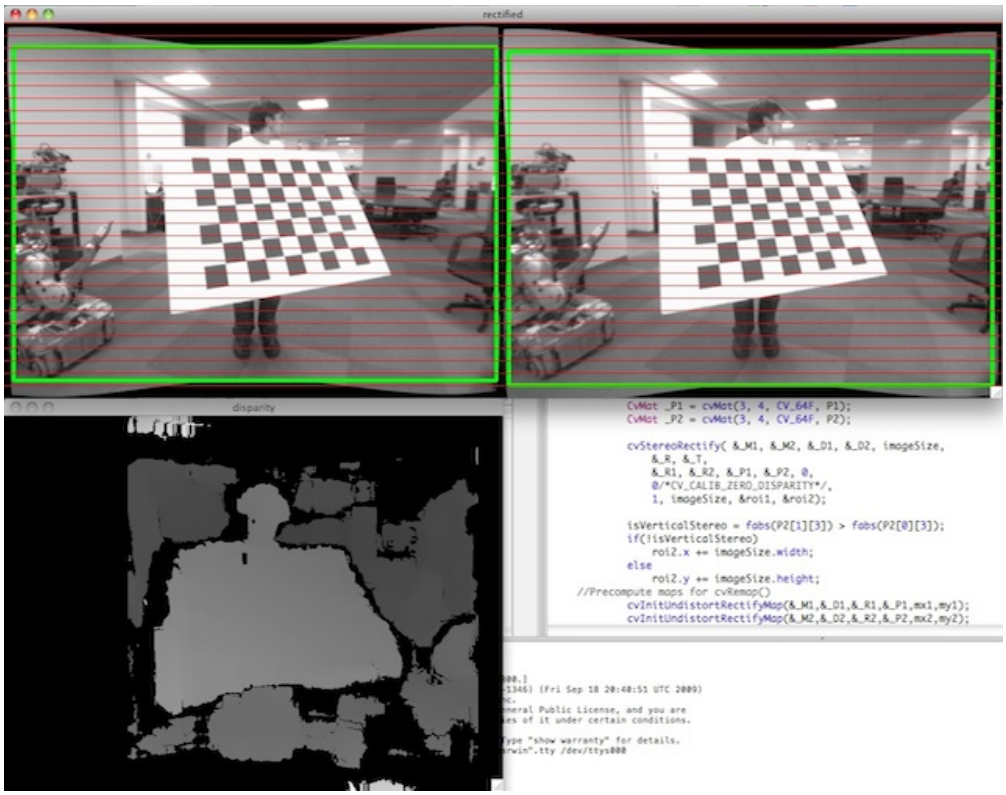
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of `P1` and `P2` will effectively be the new "rectified" camera matrices. The matrices, together with `R1` and `R2`, can then be passed to cvInitUndistortRectifyMap to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



cvStereoRectifyUncalibrated (view/add comments)

Computes rectification transform for uncalibrated stereo camera.

```
void cvStereoRectifyUncalibrated(
           const CvMat* points1,
           const CvMat* points2,
           const CvMat* F,
           CvSize imageSize,
           CvMat* H1,
           CvMat* H2,
           double threshold=5 );
```

points1, points2 The 2 arrays of corresponding 2D points. The same formats as in cvFindFundamentalMat are supported

F The input fundamental matrix. It can be computed from the same set of point pairs using cvFindFundamentalMat.

imageSize Size of the image.

H1, H2 The output rectification homography matrices for the first and for the second images.

threshold The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the points pairs that do not comply the epipolar geometry well enough (that is, the points for which $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from cvStereoRectify is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the algorithm [**?**].

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using cvCalibrateCamera2 and then the images can be corrected using cvUndistort2, or just the point coordinates can be corrected with cvUndistortPoints.

---

cvUndistort2 (view/add comments)

Transforms an image to compensate for lens distortion.

```
void cvUndistort2(
            const CvArr* src,
            CvArr* dst,
            const CvMat* cameraMatrix,
            const CvMat* distCoeffs,
            const CvMat* newCameraMatrix=0 );
```

src The input (distorted) image

dst The output (corrected) image; will have the same size and the same type as `src`

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of cvInitUndistortRectifyMap (with unity `R`) and cvRemap (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix`. You can use cvGetOptimalNewCameraMatrix to compute the appropriate `newCameraMatrix`, depending on your requirements.

The camera matrix and the distortion parameters can be determined using cvCalibrateCamera2. If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

---

cvUndistortPoints (view/add comments)

Computes the ideal point coordinates from the observed point coordinates.

```
void cvUndistortPoints(
           const CvMat* src,
           CvMat* dst,
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           const CvMat* R=NULL,
           const CvMat* P=NULL);
```

src The observed point coordinates, 1xN or Nx1 2-channel (CV_32FC2 or CV_64FC2).

dst The output ideal point coordinates, after undistortion and reverse perspective transformation , same format as `src` .

cameraMatrix The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv can be passed here. If the matrix is empty, the identity transformation is used

P The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2`, computed by cv can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to cvUndistort2 and cvInitUndistortRectifyMap, but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to cvProjectPoints2 (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used both for a stereo camera head or for monocular camera (when R is NULL).

Chapter 9

ml. Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different seta of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class ‘CvStatModel‘ that all the other ML classes are derived from.

Part II

C++ API Reference

Chapter 10

Introduction

Starting from OpenCV 2.0 the new modern C++ interface has been introduced. It is crisp (less typing is needed to code the same thing), type-safe (no more `CvArr*` a.k.a. `void*`) and, in general, more convenient to use. Here is a short example of what it looks like:

```
//
// Simple retro-style photo effect done by adding noise to
// the luminance channel and reducing intensity of the chroma channels
//

// include standard OpenCV headers, same as before
#include "cv.h"
#include "highgui.h"

// all the new API is put into "cv" namespace. Export its content
using namespace cv;

// enable/disable use of mixed API in the code below.
#define DEMO_MIXED_API_USE 1

int main( int argc, char** argv )
{
    const char* imagename = argc > 1 ? argv[1] : "lena.jpg";
#if DEMO_MIXED_API_USE
    // Ptr<T> is safe ref-conting pointer class
    Ptr<IplImage> iplimg = cvLoadImage(imagename);

    // cv::Mat replaces the CvMat and IplImage, but it's easy to convert
    // between the old and the new data structures
    // (by default, only the header is converted and the data is shared)
    Mat img(iplimg);
```

```
#else
    // the newer cvLoadImage alternative with MATLAB-style name
    Mat img = imread(imagename);
#endif

    if( !img.data ) // check if the image has been loaded properly
        return -1;

    Mat img_yuv;
    // convert image to YUV color space.
    // The output image will be allocated automatically
    cvtColor(img, img_yuv, CV_BGR2YCrCb);

    // split the image into separate color planes
    vector<Mat> planes;
    split(img_yuv, planes);

    // another Mat constructor; allocates a matrix of the specified
        // size and type
    Mat noise(img.size(), CV_8U);

    // fills the matrix with normally distributed random values;
    // there is also randu() for uniformly distributed random numbers.
    // Scalar replaces CvScalar, Scalar::all() replaces cvScalarAll().
    randn(noise, Scalar::all(128), Scalar::all(20));

    // blur the noise a bit, kernel size is 3x3 and both sigma's
        // are set to 0.5
    GaussianBlur(noise, noise, Size(3, 3), 0.5, 0.5);

    const double brightness_gain = 0;
    const double contrast_gain = 1.7;
#if DEMO_MIXED_API_USE
    // it's easy to pass the new matrices to the functions that
    // only work with IplImage or CvMat:
    // step 1) - convert the headers, data will not be copied
    IplImage cv_planes_0 = planes[0], cv_noise = noise;
    // step 2) call the function; do not forget unary "&" to form pointers
    cvAddWeighted(&cv_planes_0, contrast_gain, &cv_noise, 1,
                 -128 + brightness_gain, &cv_planes_0);
#else
    addWeighted(planes[0], constrast_gain, noise, 1,
                -128 + brightness_gain, planes[0]);
#endif
    const double color_scale = 0.5;
```

```
    // Mat::convertTo() replaces cvConvertScale.
    // One must explicitly specify the output matrix type
    // (we keep it intact, i.e. pass planes[1].type())
    planes[1].convertTo(planes[1], planes[1].type(),
                        color_scale, 128*(1-color_scale));

    // alternative form of convertTo if we know the datatype
    // at compile time ("uchar" here).
    // This expression will not create any temporary arrays
    // and should be almost as fast as the above variant
    planes[2] = Mat_<uchar>(planes[2]*color_scale + 128*(1-color_scale));

    // Mat::mul replaces cvMul(). Again, no temporary arrays are
    // created in the case of simple expressions.
    planes[0] = planes[0].mul(planes[0], 1./255);

    // now merge the results back
    merge(planes, img_yuv);
    // and produce the output RGB image
    cvtColor(img_yuv, img, CV_YCrCb2BGR);

    // this is counterpart for cvNamedWindow
    namedWindow("image with grain", CV_WINDOW_AUTOSIZE);
#if DEMO_MIXED_API_USE
    // this is to demonstrate that img and iplimg really share the data -
    // the result of the above processing is stored to img and thus
        // in iplimg too.
    cvShowImage("image with grain", iplimg);
#else
    imshow("image with grain", img);
#endif
    waitKey();

    return 0;
    // all the memory will automatically be released
    // by vector<>, Mat and Ptr<> destructors.
}
```

Following a summary ”cheatsheet” below, the rest of the introduction will discuss the key features of the new interface in more detail.

10.1 C++ Cheatsheet

The section is just a summary ”cheatsheet” of common things you may want to do with cv::Mat:. The code snippets below all assume the correct namespace is used:

```
using namespace cv;
using namespace std;
```

Convert an IplImage or CvMat to an cv::Mat and a cv::Mat to an IplImage or CvMat:

```
// Assuming somewhere IplImage *iplimg; exists
// and has been allocated and cv::Mat Mimg has been defined
Mat imgMat(iplimg);  //Construct an Mat image "img" out of an IplImage
Mimg = iplimg;       //Or just set the header of pre existing cv::Mat
                     //Ming to iplimg's data (no copying is done)

//Convert to IplImage or CvMat, no data copying
IplImage ipl_img = img;
CvMat cvmat = img; // convert cv::Mat -> CvMat
```

A very simple way to operate on a rectanglular sub-region of an image (ROI – ”Region of Interest”):

```
//Make a rectangle
Rect roi(10, 20, 100, 50);
//Point a cv::Mat header at it (no allocation is done)
Mat image_roi = image(roi);
```

A bit advanced, but should you want efficiently to sample from a circular region in an image (below, instead of sampling, we just draw into a BGR image) :

```
// the function returns x boundary coordinates of
// the circle for each y. RxV[y1] = x1 means that
// when y=y1, -x1 <=x<=x1 is inside the circle
void getCircularROI(int R, vector < int > & RxV)
{
    RxV.resize(R+1);
    for( int y = 0; y <= R; y++ )
        RxV[y] = cvRound(sqrt((double)R*R - y*y));
}

// This draws a circle in the green channel
// (note the "[1]" for a BGR" image,
// blue and red channels are not modified),
// but is really an example of how to *sample* from a circular region.
void drawCircle(Mat &image, int R, Point center)
{
    vector<int> RxV;
```

```
    getCircularROI(R, RxV);

    Mat_<Vec3b>& img = (Mat_<Vec3b>&)image; //3 channel pointer to image
    for( int dy = -R; dy <= R; dy++ )
    {
        int Rx = RxV[abs(dy)];
        for( int dx = -Rx; dx <= Rx; dx++ )
            img(center.y+dy, center.x+dx)[1] = 255;
    }
}
```

10.2 Namespace cv and Function Naming

All the newly introduced classes and functions are placed into `cv` namespace. Therefore, to access this functionality from your code, use `cv::` specifier or "`using namespace cv;`" directive:

```
#include "cv.h"

...
cv::Mat H = cv::findHomography(points1, points2, cv::RANSAC, 5);
...
```

or

```
#include "cv.h"

using namespace cv;

...
Mat H = findHomography(points1, points2, RANSAC, 5 );
...
```

It is probable that some of the current or future OpenCV external names conflict with STL or other libraries, in this case use explicit namespace specifiers to resolve the name conflicts:

```
Mat a(100, 100, CV_32F);
randu(a, Scalar::all(1), Scalar::all(std::rand()%256+1));
cv::log(a, a);
a /= std::log(2.);
```

For the most of the C functions and structures from OpenCV 1.x you may find the direct counterparts in the new C++ interface. The name is usually formed by omitting `cv` or `Cv` prefix and turning the first letter to the low case (unless it's a own name, like Canny, Sobel etc). In case when there is no the new-style counterpart, it's possible to use the old functions with the new structures, as shown the first sample in the chapter.

10.3 Memory Management

When using the new interface, the most of memory deallocation and even memory allocation operations are done automatically when needed.

First of all, Mat , SparseMat and other classes have destructors that deallocate memory buffers occupied by the structures when needed.

Secondly, this "when needed" means that the destructors do not always deallocate the buffers, they take into account possible data sharing. That is, in a destructor the reference counter associated with the underlying data is decremented and the data is deallocated if and only if the reference counter becomes zero, that is, when no other structures refer to the same buffer. When such a structure containing a reference counter is copied, usually just the header is duplicated, while the underlying data is not; instead, the reference counter is incremented to memorize that there is another owner of the same data. Also, some structures, such as `Mat`, can refer to the user-allocated data. In this case the reference counter is `NULL` pointer and then no reference counting is done - the data is not deallocated by the destructors and should be deallocated manually by the user. We saw this scheme in the first example in the chapter:

```
// allocates IplImages and wraps it into shared pointer class.
Ptr<IplImage> iplimg = cvLoadImage(...);

// constructs Mat header for IplImage data;
// does not copy the data;
// the reference counter will be NULL
Mat img(iplimg);
...
// in the end of the block img destructor is called,
// which does not try to deallocate the data because
// of NULL pointer to the reference counter.
//
// Then Ptr<IplImage> destructor is called that decrements
// the reference counter and, as the counter becomes 0 in this case,
// the destructor calls cvReleaseImage().
```

The copying semantics was mentioned in the above paragraph, but deserves a dedicated discussion. By default, the new OpenCV structures implement shallow, so called O(1) (i.e. constant-time) assignment operations. It gives user possibility to pass quite big data structures to functions (though, e.g. passing `const Mat&` is still faster than passing `Mat`), return them (e.g. see the example with findHomography above), store them in OpenCV and STL containers etc. - and do all of this very efficiently. On the other hand, most of the new data structures provide `clone()` method that creates a full copy of an object. Here is the sample:

```
// create a big 8Mb matrix
Mat A(1000, 1000, CV_64F);
```

```
// create another header for the same matrix;
// this is instant operation, regardless of the matrix size.
Mat B = A;
// create another header for the 3-rd row of A; no data is copied either
Mat C = B.row(3);
// now create a separate copy of the matrix
Mat D = B.clone();
// copy the 5-th row of B to C, that is, copy the 5-th row of A
// to the 3-rd row of A.
B.row(5).copyTo(C);
// now let A and D share the data; after that the modified version
// of A is still referenced by B and C.
A = D;
// now make B an empty matrix (which references no memory buffers),
// but the modified version of A will still be referenced by C,
// despite that C is just a single row of the original A
B.release();

// finally, make a full copy of C. In result, the big modified
// matrix will be deallocated, since it's not referenced by anyone
C = C.clone();
```

Memory management of the new data structures is automatic and thus easy. If, however, your code uses IplImage , CvMat or other C data structures a lot, memory management can still be automated without immediate migration to Mat by using the already mentioned template class Ptr , similar to `shared_ptr` from Boost and C++ TR1. It wraps a pointer to an arbitrary object, provides transparent access to all the object fields and associates a reference counter with it. Instance of the class can be passed to any function that expects the original pointer. For correct deallocation of the object, you should specialize `Ptr<T>::delete_obj()` method. Such specialized methods already exist for the classical OpenCV structures, e.g.:

```
// cxoperations.hpp:
...
template<> inline Ptr<IplImage>::delete_obj() {
    cvReleaseImage(&obj);
}
...
```

See Ptr description for more details and other usage scenarios.

10.4 Memory Management Part II. Automatic Data Allocation

With the new interface not only explicit memory deallocation is not needed anymore, but the memory allocation is often done automatically too. That was demonstrated in the example in the be-

ginning of the chapter when `cvtColor` was called, and here are some more details.

Mat and other array classes provide method `create` that allocates a new buffer for array data if and only if the currently allocated array is not of the required size and type. If a new buffer is needed, the previously allocated buffer is released (by engaging all the reference counting mechanism described in the previous section). Now, since it is very quick to check whether the needed memory buffer is already allocated, most new OpenCV functions that have arrays as output parameters call the `create` method and this way the *automatic data allocation* concept is implemented. Here is the example:

```
#include "cv.h"
#include "highgui.h"

using namespace cv;

int main(int, char**)
{
    VideoCapture cap(0);
    if(!cap.isOpened()) return -1;

    Mat edges;
    namedWindow("edges",1);
    for(;;)
    {
        Mat frame;
        cap >> frame;
        cvtColor(frame, edges, CV_BGR2GRAY);
        GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
        Canny(edges, edges, 0, 30, 3);
        imshow("edges", edges);
        if(waitKey(30) >= 0) break;
    }
    return 0;
}
```

The matrix `edges` is allocated during the first frame processing and unless the resolution will suddenly change, the same buffer will be reused for every next frame's edge map.

In many cases the output array type and size can be inferenced from the input arrays' respective characteristics, but not always. In these rare cases the corresponding functions take separate input parameters that specify the data type and/or size of the output arrays, like resize . Anyway, a vast majority of the new-style array processing functions call `create` for each of the output array, with just a few exceptions like `mixChannels`, `RNG::fill` and some others.

Note that this output array allocation semantic is only implemented in the new functions. If you want to pass the new structures to some old OpenCV function, you should first allocate the output arrays using `create` method, then make `CvMat` or `IplImage` headers and after that call

the function.

10.5 Algebraic Operations

Just like in v1.x, OpenCV 2.x provides some basic functions operating on matrices, like `add`, `subtract`, `gemm` etc. In addition, it introduces overloaded operators that give the user a convenient algebraic notation, which is nearly as fast as using the functions directly. For example, here is how the least squares problem $Ax = b$ can be solved using normal equations:

```
Mat x = (A.t()*A).inv()*(A.t()*b);
```

The complete list of overloaded operators can be found in Matrix Expressions .

10.6 Fast Element Access

Historically, OpenCV provided many different ways to access image and matrix elements, and none of them was both fast and convenient. With the new data structures, OpenCV 2.x introduces a few more alternatives, hopefully more convenient than before. For detailed description of the operations, please, check Mat and Mat_ description. Here is part of the retro-photo-styling example rewritten (in simplified form) using the element access operations:

```
...
// split the image into separate color planes
vector<Mat> planes;
split(img_yuv, planes);

// method 1. process Y plane using an iterator
MatIterator_<uchar> it = planes[0].begin<uchar>(),
                    it_end = planes[0].end<uchar>();
for(; it != it_end; ++it)
{
    double v = *it*1.7 + rand()%21-10;
    *it = saturate_cast<uchar>(v*v/255.);
}

// method 2. process the first chroma plane using pre-stored row pointer.
// method 3. process the second chroma plane using
//           individual element access operations
for( int y = 0; y < img_yuv.rows; y++ )
{
    uchar* Uptr = planes[1].ptr<uchar>(y);
    for( int x = 0; x < img_yuv.cols; x++ )
    {
        Uptr[x] = saturate_cast<uchar>((Uptr[x]-128)/2 + 128);
```

```
        uchar& Vxy = planes[2].at<uchar>(y, x);
        Vxy = saturate_cast<uchar>((Vxy-128)/2 + 128);
    }
}

merge(planes, img_yuv);
...
```

10.7 Saturation Arithmetics

In the above sample you may have noticed saturate_cast operator, and that's how all the pixel processing is done in OpenCV. When a result of image operation is 8-bit image with pixel values ranging from 0 to 255, each output pixel value is clipped to this available range:

$$I(x, y) = \min(\max(value, 0), 255)$$

and the similar rules are applied to 8-bit signed and 16-bit signed and unsigned types. This "saturation" semantics (different from usual C language "wrapping" semantics, where lowest bits are taken, is implemented in every image processing function, from the simple `cv::add` to `cv::cvtColor`, `cv::resize`, `cv::filter2D` etc. It is not a new feature of OpenCV v2.x, it was there from very beginning. In the new version this special saturate_cast template operator is introduced to simplify implementation of this semantic in your own functions.

10.8 Error handling

The modern error handling mechanism in OpenCV uses exceptions, as opposite to the manual stack unrolling used in previous versions.

If you to check some conditions in your code and raise an error if they are not satisfied, use CV_Assert() or CV_Error().

```
CV_Assert(mymat.type() == CV_32FC1);
...

if( scaleValue < 0 || scaleValue > 1000 )
   CV_Error(CV_StsOutOfRange, "The scale value is out of range");
```

There is also CV_DbgAssert that yields no code in the Release configuration.

To handle the errors, use the standard exception handling mechanism:

```
try
{
    ...
}
```

```
catch( cv::Exception& e )
{
    const char* err_msg = e.what();
    ...
}
```

instead of cv::Exception you can write `std::exception`, since the former is derived from the latter.

Then, obviously, to make it all work and do not worry about the object destruction, try not to use `IplImage*`, `CvMat*` and plain pointers in general. Use cv::Mat, `std::vector<>`, cv::Ptr etc.

10.9 Threading and Reenterability

OpenCV uses OpenMP to run some time-consuming operations in parallel. Threading can be explicitly controlled by setNumThreads function. Also, functions and ”const” methods of the classes are generally re-enterable, that is, they can be called from different threads asynchronously.

Chapter 11

core. The Core Functionality

11.1 Basic Structures

DataType

Template ”traits” class for other OpenCV primitive data types

```
template<typename _Tp> class DataType
{
    // value_type is always a synonym for _Tp.
    typedef _Tp value_type;

    // intermediate type used for operations on _Tp.
    // it is int for uchar, signed char, unsigned short, signed short and int,
    // float for float, double for double, ...
    typedef <...> work_type;
    // in the case of multi-channel data it is the data type of each channel
    typedef <...> channel_type;
    enum
    {
        // CV_8U ... CV_64F
        depth = DataDepth<channel_type>::value,
        // 1 ...
        channels = <...>,
        // ’1u’, ’4i’, ’3f’, ’2d’ etc.
        fmt=<...>,
        // CV_8UC3, CV_32FC2 ...
        type = CV_MAKETYPE(depth, channels)
    };
};
```

The template class `DataType` is descriptive class for OpenCV primitive data types and other types that comply with the following definition. A primitive OpenCV data type is one of `unsigned char, bool, signed char, unsigned short, signed short, int, float, double` or a tuple of values of one of these types, where all the values in the tuple have the same type. If you are familiar with OpenCV CvMat 's type notation, CV_8U ... CV_32FC3, CV_64FC2 etc., then a primitive type can be defined as a type for which you can give a unique identifier in a form `CV_<bit-depth>U|S|FC<number_of_channels>`. A universal OpenCV structure able to store a single instance of such primitive data type is Vec . Multiple instances of such a type can be stored to a `std::vector`, `Mat`, `Mat_`, `SparseMat`, `SparseMat_` or any other container that is able to store Vec instances.

The class `DataType` is basically used to provide some description of such primitive data types without adding any fields or methods to the corresponding classes (and it is actually impossible to add anything to primitive C/C++ data types). This technique is known in C++ as class traits. It's not `DataType` itself that is used, but its specialized versions, such as:

```
template<> class DataType<uchar>
{
    typedef uchar value_type;
    typedef int work_type;
    typedef uchar channel_type;
    enum { channel_type = CV_8U, channels = 1, fmt='u', type = CV_8U };
};
...
template<typename _Tp> DataType<std::complex<_Tp> >
{
    typedef std::complex<_Tp> value_type;
    typedef std::complex<_Tp> work_type;
    typedef _Tp channel_type;
    // DataDepth is another helper trait class
    enum { depth = DataDepth<_Tp>::value, channels=2,
        fmt=(channels-1)*256+DataDepth<_Tp>::fmt,
        type=CV_MAKETYPE(depth, channels) };
};
...
```

The main purpose of the classes is to convert compile-time type information to OpenCV-compatible data type identifier, for example:

```
// allocates 30x40 floating-point matrix
Mat A(30, 40, DataType<float>::type);

Mat B = Mat_<std::complex<double> >(3, 3);
// the statement below will print 6, 2 /* i.e. depth == CV_64F, channels == 2 */
cout << B.depth() << ", " << B.channels() << endl;
```

that is, such traits are used to tell OpenCV which data type you are working with, even if such a type is not native to OpenCV (the matrix `B` intialization above compiles because OpenCV defines the proper specialized template class `DataType<complex<_Tp> >`). Also, this mechanism is useful (and used in OpenCV this way) for generic algorithms implementations.

Point_

Template class for 2D points

```
template<typename _Tp> class Point_
{
public:
    typedef _Tp value_type;

    Point_();
    Point_(_Tp _x, _Tp _y);
    Point_(const Point_& pt);
    Point_(const CvPoint& pt);
    Point_(const CvPoint2D32f& pt);
    Point_(const Size_<_Tp>& sz);
    Point_(const Vec<_Tp, 2>& v);
    Point_& operator = (const Point_& pt);
    template<typename _Tp2> operator Point_<_Tp2>() const;
    operator CvPoint() const;
    operator CvPoint2D32f() const;
    operator Vec<_Tp, 2>() const;

    // computes dot-product (this->x*pt.x + this->y*pt.y)
    _Tp dot(const Point_& pt) const;
    // computes dot-product using double-precision arithmetics
    double ddot(const Point_& pt) const;
    // returns true if the point is inside the rectangle "r".
    bool inside(const Rect_<_Tp>& r) const;

    _Tp x, y;
};
```

The class represents a 2D point, specified by its coordinates x and y. Instance of the class is interchangeable with C structures `CvPoint` and `CvPoint2D32f`. There is also cast operator to convert point coordinates to the specified type. The conversion from floating-point coordinates to integer coordinates is done by rounding; in general case the conversion uses saturate_cast operation on each of the coordinates. Besides the class members listed in the declaration above, the following operations on points are implemented:

```
    pt1 = pt2 + pt3;
```

```
    pt1 = pt2 - pt3;
    pt1 = pt2 * a;
    pt1 = a * pt2;
    pt1 += pt2;
    pt1 -= pt2;
    pt1 *= a;
    double value = norm(pt); // L2 norm
    pt1 == pt2;
    pt1 != pt2;
```

For user convenience, the following type aliases are defined:

```
typedef Point_<int> Point2i;
typedef Point2i Point;
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;
```

Here is a short example:

```
Point2f a(0.3f, 0.f), b(0.f, 0.4f);
Point pt = (a + b)*10.f;
cout << pt.x << ", " << pt.y << endl;
```

Point3_

Template class for 3D points

```
template<typename _Tp> class Point3_
{
public:
    typedef _Tp value_type;

    Point3_();
    Point3_(_Tp _x, _Tp _y, _Tp _z);
    Point3_(const Point3_& pt);
    explicit Point3_(const Point_<_Tp>& pt);
    Point3_(const CvPoint3D32f& pt);
    Point3_(const Vec<_Tp, 3>& v);
    Point3_& operator = (const Point3_& pt);
    template<typename _Tp2> operator Point3_<_Tp2>() const;
    operator CvPoint3D32f() const;
    operator Vec<_Tp, 3>() const;

    _Tp dot(const Point3_& pt) const;
    double ddot(const Point3_& pt) const;
```

```
    _Tp x, y, z;
};
```

The class represents a 3D point, specified by its coordinates x, y and z. Instance of the class is interchangeable with C structure `CvPoint2D32f`. Similarly to `Point_`, the 3D points' coordinates can be converted to another type, and the vector arithmetic and comparison operations are also supported.

The following type aliases are available:

```
typedef Point3_<int> Point3i;
typedef Point3_<float> Point3f;
typedef Point3_<double> Point3d;
```

Size_

Template class for specfying image or rectangle size.

```
template<typename _Tp> class Size_
{
public:
    typedef _Tp value_type;

    Size_();
    Size_(_Tp _width, _Tp _height);
    Size_(const Size_& sz);
    Size_(const CvSize& sz);
    Size_(const CvSize2D32f& sz);
    Size_(const Point_<_Tp>& pt);
    Size_& operator = (const Size_& sz);
    _Tp area() const;

    operator Size_<int>() const;
    operator Size_<float>() const;
    operator Size_<double>() const;
    operator CvSize() const;
    operator CvSize2D32f() const;

    _Tp width, height;
};
```

The class `Size_` is similar to `Point_`, except that the two members are called `width` and `height` instead of `x` and `y`. The structure can be converted to and from the old OpenCV structures CvSize and CvSize2D32f . The same set of arithmetic and comparison operations as for `Point_` is available.

OpenCV defines the following type aliases:

```
typedef Size_<int> Size2i;
typedef Size2i Size;
typedef Size_<float> Size2f;
```

Rect_

Template class for 2D rectangles

```
template<typename _Tp> class Rect_
{
public:
    typedef _Tp value_type;

    Rect_();
    Rect_(_Tp _x, _Tp _y, _Tp _width, _Tp _height);
    Rect_(const Rect_& r);
    Rect_(const CvRect& r);
    // (x, y) <- org, (width, height) <- sz
    Rect_(const Point_<_Tp>& org, const Size_<_Tp>& sz);
    // (x, y) <- min(pt1, pt2), (width, height) <- max(pt1, pt2) - (x, y)
    Rect_(const Point_<_Tp>& pt1, const Point_<_Tp>& pt2);
    Rect_& operator = ( const Rect_& r );
    // returns Point_<_Tp>(x, y)
    Point_<_Tp> tl() const;
    // returns Point_<_Tp>(x+width, y+height)
    Point_<_Tp> br() const;

    // returns Size_<_Tp>(width, height)
    Size_<_Tp> size() const;
    // returns width*height
    _Tp area() const;

    operator Rect_<int>() const;
    operator Rect_<float>() const;
    operator Rect_<double>() const;
    operator CvRect() const;

    // x <= pt.x && pt.x < x + width &&
    // y <= pt.y && pt.y < y + height ? true : false
    bool contains(const Point_<_Tp>& pt) const;

    _Tp x, y, width, height;
};
```

The rectangle is described by the coordinates of the top-left corner (which is the default interpretation of `Rect_::x` and `Rect_::y` in OpenCV; though, in your algorithms you may count `x` and `y` from the bottom-left corner), the rectangle width and height.

Another assumption OpenCV usually makes is that the top and left boundary of the rectangle are inclusive, while the right and bottom boundaries are not, for example, the method `Rect_::contains` returns true if

$$x \leq pt.x < x + width, y \leq pt.y < y + height$$

And virtually every loop over an image ROI in OpenCV (where ROI is specified by `Rect_<int>`) is implemented as:

```
for(int y = roi.y; y < roi.y + rect.height; y++)
    for(int x = roi.x; x < roi.x + rect.width; x++)
    {
        // ...
    }
```

In addition to the class members, the following operations on rectangles are implemented:

- `rect = rect ± point` (shifting rectangle by a certain offset)
- `rect = rect ± size` (expanding or shrinking rectangle by a certain amount)
- `rect += point, rect -= point, rect += size, rect -= size` (augmenting operations)
- `rect = rect1 & rect2` (rectangle intersection)
- `rect = rect1 | rect2` (minimum area rectangle containing `rect2` and `rect3`)
- `rect &= rect1, rect |= rect1` (and the corresponding augmenting operations)
- `rect == rect1, rect != rect1` (rectangle comparison)

Example. Here is how the partial ordering on rectangles can be established (rect1 $\subseteq$ rect2):

```
template<typename _Tp> inline bool
operator <= (const Rect_<_Tp>& r1, const Rect_<_Tp>& r2)
{
    return (r1 & r2) == r1;
}
```

For user convenience, the following type alias is available:

```
typedef Rect_<int> Rect;
```

RotatedRect

Possibly rotated rectangle

```
class RotatedRect
{
public:
    // constructors
    RotatedRect();
    RotatedRect(const Point2f& _center, const Size2f& _size, float _angle);
    RotatedRect(const CvBox2D& box);

    // returns minimal up-right rectangle that contains the rotated rectangle
    Rect boundingRect() const;
    // backward conversion to CvBox2D
    operator CvBox2D() const;

    // mass center of the rectangle
    Point2f center;
    // size
    Size2f size;
    // rotation angle in degrees
    float angle;
};
```

The class `RotatedRect` replaces the old CvBox2D and fully compatible with it.

TermCriteria

Termination criteria for iterative algorithms

```
class TermCriteria
{
public:
    enum { COUNT=1, MAX_ITER=COUNT, EPS=2 };

    // constructors
    TermCriteria();
    // type can be MAX_ITER, EPS or MAX_ITER+EPS.
    // type = MAX_ITER means that only the number of iterations does matter;
    // type = EPS means that only the required precision (epsilon) does matter
    //    (though, most algorithms put some limit on the number of iterations anyway)
    // type = MAX_ITER + EPS means that algorithm stops when
    // either the specified number of iterations is made,
    // or when the specified accuracy is achieved - whatever happens first.
```

```
    TermCriteria(int _type, int _maxCount, double _epsilon);
    TermCriteria(const CvTermCriteria& criteria);
    operator CvTermCriteria() const;

    int type;
    int maxCount;
    double epsilon;
};
```

The class `TermCriteria` replaces the old CvTermCriteria and fully compatible with it.

Matx

Template class for small matrices

```
template<typename T, int m, int n> class Matx
{
public:
    typedef T value_type;
    enum { depth = DataDepth<T>::value, channels = m*n,
           type = CV_MAKETYPE(depth, channels) };

    // various methods
    ...

    Tp val[m*n];
};

typedef Matx<float, 1, 2> Matx12f;
typedef Matx<double, 1, 2> Matx12d;
...
typedef Matx<float, 1, 6> Matx16f;
typedef Matx<double, 1, 6> Matx16d;

typedef Matx<float, 2, 1> Matx21f;
typedef Matx<double, 2, 1> Matx21d;
...
typedef Matx<float, 6, 1> Matx61f;
typedef Matx<double, 6, 1> Matx61d;

typedef Matx<float, 2, 2> Matx22f;
typedef Matx<double, 2, 2> Matx22d;
...
typedef Matx<float, 6, 6> Matx66f;
typedef Matx<double, 6, 6> Matx66d;
```

The class represents small matrices, which type and size are known at compile time. If you need more flexible type, use Mat . The elements of a matrix `M` are accessible using `M(i,j)` notation, and most of the common matrix operations (see also MatrixExpressions) are available. If you need to do some operation on `Matx` that is not implemented, it is easy to convert the matrix to Mat and backwards.

```
Matx33f m(1, 2, 3,
          4, 5, 6,
          7, 8, 9);
cout << sum(Mat(m*m.t())) << endl;
```

Vec

Template class for short numerical vectors

```
template<typename T, int cn> class Vec : public Matx<T, cn, 1>
{
public:
    typedef T value_type;
    enum { depth = DataDepth<T>::value, channels = cn,
           type = CV_MAKETYPE(depth, channels) };

    // various methods ...
};

typedef Vec<uchar, 2> Vec2b;
typedef Vec<uchar, 3> Vec3b;
typedef Vec<uchar, 4> Vec4b;

typedef Vec<short, 2> Vec2s;
typedef Vec<short, 3> Vec3s;
typedef Vec<short, 4> Vec4s;

typedef Vec<int, 2> Vec2i;
typedef Vec<int, 3> Vec3i;
typedef Vec<int, 4> Vec4i;

typedef Vec<float, 2> Vec2f;
typedef Vec<float, 3> Vec3f;
typedef Vec<float, 4> Vec4f;
typedef Vec<float, 6> Vec6f;

typedef Vec<double, 2> Vec2d;
typedef Vec<double, 3> Vec3d;
```

```
typedef Vec<double, 4> Vec4d;
typedef Vec<double, 6> Vec6d;
```

Vec is a partial case of Matx. It is possible to convert Vec<T,2> to/from Point_, Vec<T,3> to/from Point3_, and Vec<T,4> to CvScalar or Scalar. The elements of Vec are accessed using operator[]. All the expected vector operations are implemented too:

- v1 = v2 ± v3, v1 = v2 ∗ α, v1 = α ∗ v2 (plus the corresponding augmenting operations; note that these operations apply saturate_cast.3C.3E to the each computed vector component)
- v1 == v2, v1 != v2
- norm(v1) (L_2-norm)

The class Vec is commonly used to describe pixel types of multi-channel arrays, see Mat_ description.

Scalar_

4-element vector

```
template<typename _Tp> class Scalar_ : public Vec<_Tp, 4>
{
public:
    Scalar_();
    Scalar_(_Tp v0, _Tp v1, _Tp v2=0, _Tp v3=0);
    Scalar_(const CvScalar& s);
    Scalar_(_Tp v0);
    static Scalar_<_Tp> all(_Tp v0);
    operator CvScalar() const;

    template<typename T2> operator Scalar_<T2>() const;

    Scalar_<_Tp> mul(const Scalar_<_Tp>& t, double scale=1 ) const;
    template<typename T2> void convertTo(T2* buf, int channels, int unroll_to=0) const;
};

typedef Scalar_<double> Scalar;
```

The template class Scalar_ and it's double-precision instantiation Scalar represent 4-element vector. Being derived from Vec<_Tp, 4>, they can be used as typical 4-element vectors, but in addition they can be converted to/from CvScalar. The type Scalar is widely used in OpenCV for passing pixel values and it is a drop-in replacement for CvScalar that was used for the same purpose in the earlier versions of OpenCV.

Range

Specifies a continuous subsequence (a.k.a. slice) of a sequence.

```
class Range
{
public:
    Range();
    Range(int _start, int _end);
    Range(const CvSlice& slice);
    int size() const;
    bool empty() const;
    static Range all();
    operator CvSlice() const;

    int start, end;
};
```

The class is used to specify a row or column span in a matrix (Mat), and for many other purposes. `Range(a,b)` is basically the same as `a:b` in Matlab or `a..b` in Python. As in Python, `start` is inclusive left boundary of the range, and `end` is exclusive right boundary of the range. Such a half-opened interval is usually denoted as $[start, end)$.

The static method `Range::all()` returns some special variable that means "the whole sequence" or "the whole range", just like ":" in Matlab or "..." in Python. All the methods and functions in OpenCV that take `Range` support this special `Range::all()` value, but of course, in the case of your own custom processing you will probably have to check and handle it explicitly:

```
void my_function(..., const Range& r, ....)
{
    if(r == Range::all()) {
        // process all the data
    }
    else {
        // process [r.start, r.end)
    }
}
```

Ptr

A template class for smart reference-counting pointers

```
template<typename _Tp> class Ptr
{
public:
```

```
    // default constructor
    Ptr();
    // constructor that wraps the object pointer
    Ptr(_Tp* _obj);
    // destructor: calls release()
    ~Ptr();
    // copy constructor; increments ptr's reference counter
    Ptr(const Ptr& ptr);
    // assignment operator; decrements own reference counter
    // (with release()) and increments ptr's reference counter
    Ptr& operator = (const Ptr& ptr);
    // increments reference counter
    void addref();
    // decrements reference counter; when it becomes 0,
    // delete_obj() is called
    void release();
    // user-specified custom object deletion operation.
    // by default, "delete obj;" is called
    void delete_obj();
    // returns true if obj == 0;
    bool empty() const;

    // provide access to the object fields and methods
    _Tp* operator -> ();
    const _Tp* operator -> () const;

    // return the underlying object pointer;
    // thanks to the methods, the Ptr<_Tp> can be
    // used instead of _Tp*
    operator _Tp* ();
    operator const _Tp*() const;
protected:
    // the encapsulated object pointer
    _Tp* obj;
    // the associated reference counter
    int* refcount;
};
```

The class `Ptr<_Tp>` is a template class that wraps pointers of the corresponding type. It is similar to `shared_ptr` that is a part of Boost library (http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/shared_ptr.htm) and also a part of the C++0x standard.

By using this class you can get the following capabilities:

- default constructor, copy constructor and assignment operator for an arbitrary C++ class or a C structure. For some objects, like files, windows, mutexes, sockets etc, copy constructor

or assignment operator are difficult to define. For some other objects, like complex classifiers in OpenCV, copy constructors are absent and not easy to implement. Finally, some of complex OpenCV and your own data structures may have been written in C. However, copy constructors and default constructors can simplify programming a lot; besides, they are often required (e.g. by STL containers). By wrapping a pointer to such a complex object `TObj` to `Ptr<TObj>` you will automatically get all of the necessary constructors and the assignment operator.

- all the above-mentioned operations running very fast, regardless of the data size, i.e. as "O(1)" operations. Indeed, while some structures, like `std::vector` provide a copy constructor and an assignment operator, the operations may take considerable time if the data structures are big. But if the structures are put into `Ptr<>`, the overhead becomes small and independent of the data size.
- automatic destruction, even for C structures. See the example below with `FILE*`.
- heterogeneous collections of objects. The standard STL and most other C++ and OpenCV containers can only store objects of the same type and the same size. The classical solution to store objects of different types in the same container is to store pointers to the base class `base_class_t*` instead, but when you loose the automatic memory management. Again, by using `Ptr<base_class_t>()` instead of the raw pointers, you can solve the problem.

The class `Ptr` treats the wrapped object as a black box, the reference counter is allocated and managed separately. The only thing the pointer class needs to know about the object is how to deallocate it. This knowledge is incapsulated in `Ptr::delete_obj()` method, which is called when the reference counter becomes 0. If the object is a C++ class instance, no additional coding is needed, because the default implementation of this method calls `delete obj;`. However, if the object is deallocated in a different way, then the specialized method should be created. For example, if you want to wrap `FILE`, the `delete_obj` may be implemented as following:

```
template<> inline void Ptr<FILE>::delete_obj()
{
    fclose(obj); // no need to clear the pointer afterwards,
                 // it is done externally.
}
...

// now use it:
Ptr<FILE> f(fopen("myfile.txt", "r"));
if(f.empty())
    throw ...;
fprintf(f, ....);
...
// the file will be closed automatically by the Ptr<FILE> destructor.
```

Note: The reference increment/decrement operations are implemented as atomic operations, and therefore it is normally safe to use the classes in multi-threaded applications. The same is true for Mat and other C++ OpenCV classes that operate on the reference counters.

Mat

OpenCV C++ n-dimensional dense array class.

```
class CV_EXPORTS Mat
{
public:
    // ... a lot of methods ...
    ...

    /*! includes several bit-fields:
         - the magic signature
         - continuity flag
         - depth
         - number of channels
     */
    int flags;
    //! the array dimensionality, >= 2
    int dims;
    //! the number of rows and columns or (-1, -1) when the array has more than 2 dimensio
    int rows, cols;
    //! pointer to the data
    uchar* data;

    //! pointer to the reference counter;
    // when array points to user-allocated data, the pointer is NULL
    int* refcount;

    // other members
    ...
};
```

The class `Mat` represents an n-dimensional dense numerical single-channel or multi-channel array. It can be used to store real or complex-valued vectors and matrices, grayscale or color images, voxel volumes, vector fields, point clouds, tensors, histograms (though, very high-dimensional histograms may be better stored in a `SparseMat`). The data layout of array M is defined by the array `M.step[]`, so that the address of element $(i_0, ..., i_{M.dims-1})$, where $0 \leq i_k < M.size[k]$ is computed as:

$$addr(M_{i_0,...,i_{M.dims-1}}) = M.data + M.step[0]*i_0 + M.step[1]*i_1 + ... + M.step[M.dims-1]*i_{M.dims-1}$$

In the case of 2-dimensional array the above formula is reduced to:

$$addr(M_{i,j}) = M.data + M.step[0] * i + M.step[1] * j$$

Note that `M.step[i] >= M.step[i+1]` (in fact, `M.step[i] >= M.step[i+1]*M.size[i+1]`), that is, 2-dimensional matrices are stored row-by-row, 3-dimensional matrices are stored plane-by-plane etc. `M.step[M.dims-1]` is minimal and always equal to the element size `M.elemSize()`.

That is, the data layout in `Mat` is fully compatible with `CvMat`, `IplImage` and `CvMatND` types from OpenCV 1.x, as well as with majority of dense array types from the standard toolkits and SDKs, such as Numpy (ndarray), Win32 (independent device bitmaps) etc, i.e. any other array that uses "steps", a.k.a. "strides", to compute position of a pixel. Because of such compatibility, it is possible to make a `Mat` header for user-allocated data and process it in-place using OpenCV functions.

There are many different ways to create `Mat` object. Here are the some popular ones:

- using `create(nrows, ncols, type)` method or the similar constructor `Mat(nrows, ncols, type[, fillValue])` constructor. A new array of the specified size and specifed type will be allocated. `type` has the same meaning as in cv::cvCreateMat method, e.g. `CV_8UC1` means 8-bit single-channel array, `CV_32FC2` means 2-channel (i.e. complex) floating-point array etc:

```
// make 7x7 complex matrix filled with 1+3j.
cv::Mat M(7,7,CV_32FC2,Scalar(1,3));
// and now turn M to 100x60 15-channel 8-bit matrix.
// The old content will be deallocated
M.create(100,60,CV_8UC(15));
```

  As noted in the introduction of this chapter, `create()` will only allocate a new array when the current array shape or type are different from the specified.

- similarly to above, you can create a multi-dimensional array:

```
// create 100x100x100 8-bit array
int sz[] = {100, 100, 100};
cv::Mat bigCube(3, sz, CV_8U, Scalar::all(0));
```

  note that it is pass number of dimensions =1 to the `Mat` constructor, but the created array will be 2-dimensional, with the number of columns set to 1. That's why `Mat::dims` is always ¿= 2 (can also be 0 when the array is empty)

- by using a copy constructor or assignment operator, where on the right side it can be a array or expression, see below. Again, as noted in the introduction, array assignment is O(1) operation because it only copies the header and increases the reference counter. `Mat::clone()` method can be used to get a full (a.k.a. deep) copy of the array when you need it.

- by constructing a header for a part of another array. It can be a single row, single column, several rows, several columns, rectangular region in the array (called a minor in algebra) or a diagonal. Such operations are also O(1), because the new header will reference the same data. You can actually modify a part of the array using this feature, e.g.

```
// add 5-th row, multiplied by 3 to the 3rd row
M.row(3) = M.row(3) + M.row(5)*3;

// now copy 7-th column to the 1-st column
// M.col(1) = M.col(7); // this will not work
Mat M1 = M.col(1);
M.col(7).copyTo(M1);

// create new 320x240 image
cv::Mat img(Size(320,240),CV_8UC3);
// select a roi
cv::Mat roi(img, Rect(10,10,100,100));
// fill the ROI with (0,255,0) (which is green in RGB space);
// the original 320x240 image will be modified
roi = Scalar(0,255,0);
```

  Thanks to the additional `datastart` and `dataend` members, it is possible to compute the relative sub-array position in the main *"container"* array using `locateROI()`:

```
Mat A = Mat::eye(10, 10, CV_32S);
// extracts A columns, 1 (inclusive) to 3 (exclusive).
Mat B = A(Range::all(), Range(1, 3));
// extracts B rows, 5 (inclusive) to 9 (exclusive).
// that is, C ~ A(Range(5, 9), Range(1, 3))
Mat C = B(Range(5, 9), Range::all());
Size size; Point ofs;
C.locateROI(size, ofs);
// size will be (width=10,height=10) and the ofs will be (x=1, y=5)
```

  As in the case of whole matrices, if you need a deep copy, use `clone()` method of the extracted sub-matrices.

- by making a header for user-allocated-data. It can be useful for

  1. processing "foreign" data using OpenCV (e.g. when you implement a DirectShow filter or a processing module for gstreamer etc.), e.g.

```
void process_video_frame(const unsigned char* pixels,
                         int width, int height, int step)
{
    cv::Mat img(height, width, CV_8UC3, pixels, step);
```

```
    cv::GaussianBlur(img, img, cv::Size(7,7), 1.5, 1.5);
}
```

2. for quick initialization of small matrices and/or super-fast element access

```
double m[3][3] = {{a, b, c}, {d, e, f}, {g, h, i}};
cv::Mat M = cv::Mat(3, 3, CV_64F, m).inv();
```

partial yet very common cases of this "user-allocated data" case are conversions from CvMat and IplImage to `Mat`. For this purpose there are special constructors taking pointers to `CvMat` or `IplImage` and the optional flag indicating whether to copy the data or not.

Backward conversion from `Mat` to `CvMat` or `IplImage` is provided via cast operators `Mat::operator CvMat() const` an `Mat::operator IplImage()`. The operators do *not* copy the data.

```
IplImage* img = cvLoadImage("greatwave.jpg", 1);
Mat mtx(img); // convert IplImage* -> cv::Mat
CvMat oldmat = mtx; // convert cv::Mat -> CvMat
CV_Assert(oldmat.cols == img->width && oldmat.rows == img->height &&
    oldmat.data.ptr == (uchar*)img->imageData && oldmat.step == img->widthStep);
```

- by using MATLAB-style array initializers, `zeros(), ones(), eye()`, e.g.:

```
// create a double-precision identity martix and add it to M.
M += Mat::eye(M.rows, M.cols, CV_64F);
```

- by using comma-separated initializer:

```
// create 3x3 double-precision identity matrix
Mat M = (Mat_<double>(3,3) << 1, 0, 0, 0, 1, 0, 0, 0, 1);
```

here we first call constructor of `Mat_` class (that we describe further) with the proper parameters, and then we just put `<<` operator followed by comma-separated values that can be constants, variables, expressions etc. Also, note the extra parentheses that are needed to avoid compiler errors.

Once array is created, it will be automatically managed by using reference-counting mechanism (unless the array header is built on top of user-allocated data, in which case you should handle the data by yourself). The array data will be deallocated when no one points to it; if you want to release the data pointed by a array header before the array destructor is called, use `Mat::release()`.

The next important thing to learn about the array class is element access. Earlier it was shown how to compute address of each array element. Normally, it's not needed to use the formula directly in your code. If you know the array element type (which can be retrieved using the method `Mat::type()`), you can access element M_{ij} of 2-dimensional array as:

```
M.at<double>(i,j) += 1.f;
```

assuming that M is double-precision floating-point array. There are several variants of the method `at` for different number of dimensions.

If you need to process a whole row of a 2d array, the most efficient way is to get the pointer to the row first, and then just use plain C operator `[]`:

```
// compute sum of positive matrix elements
// (assuming that M is double-precision matrix)
double sum=0;
for(int i = 0; i < M.rows; i++)
{
    const double* Mi = M.ptr<double>(i);
    for(int j = 0; j < M.cols; j++)
        sum += std::max(Mi[j], 0.);
}
```

Some operations, like the above one, do not actually depend on the array shape, they just process elements of an array one by one (or elements from multiple arrays that have the same coordinates, e.g. array addition). Such operations are called element-wise and it makes sense to check whether all the input/output arrays are continuous, i.e. have no gaps in the end of each row, and if yes, process them as a single long row:

```
// compute sum of positive matrix elements, optimized variant
double sum=0;
int cols = M.cols, rows = M.rows;
if(M.isContinuous())
{
    cols *= rows;
    rows = 1;
}
for(int i = 0; i < rows; i++)
{
    const double* Mi = M.ptr<double>(i);
    for(int j = 0; j < cols; j++)
        sum += std::max(Mi[j], 0.);
}
```

in the case of continuous matrix the outer loop body will be executed just once, so the overhead will be smaller, which will be especially noticeable in the case of small matrices.

Finally, there are STL-style iterators that are smart enough to skip gaps between successive rows:

```
// compute sum of positive matrix elements, iterator-based variant
double sum=0;
MatConstIterator_<double> it = M.begin<double>(), it_end = M.end<double>();
```

```
for(; it != it_end; ++it)
    sum += std::max(*it, 0.);
```

The matrix iterators are random-access iterators, so they can be passed to any STL algorithm, including `std::sort()`.

Matrix Expressions

This is a list of implemented matrix operations that can be combined in arbitrary complex expressions (here *A*, *B* stand for matrices (`Mat`), *s* for a scalar (`Scalar`), α for a real-valued scalar (`double`)):

- addition, subtraction, negation: $A \pm B,\ A \pm s,\ s \pm A,\ -A$
- scaling: $A * \alpha$, $A * \alpha$
- per-element multiplication and division: $A.mul(B), A/B, \alpha/A$
- matrix multiplication: $A * B$
- transposition: $A.t() \sim A^t$
- matrix inversion and pseudo-inversion, solving linear systems and least-squares problems: $A.inv([method]) \sim A^{-1}, A.inv([method]) * B \sim X : AX = B$
- comparison: $A \gtrless B,\ A \neq B,\ A \gtrless \alpha,\ A \neq \alpha$. The result of comparison is 8-bit single channel mask, which elements are set to 255 (if the particular element or pair of elements satisfy the condition) and 0 otherwise.
- bitwise logical operations: `A & B, A & s, A | B, A | s, A ^ B, A ^ s, ~A`
- element-wise minimum and maximum: $min(A, B), min(A, \alpha), max(A, B), max(A, \alpha)$
- element-wise absolute value: $abs(A)$
- cross-product, dot-product: $A.cross(B), A.dot(B)$
- any function of matrix or matrices and scalars that returns a matrix or a scalar, such as cv::norm, cv::mean, cv::sum, cv::countNonZero, cv::trace, cv::determinant, cv::repeat etc.
- matrix initializers (`eye(), zeros(), ones()`), matrix comma-separated initializers, matrix constructors and operators that extract sub-matrices (see Mat description).
- `Mat_<destination_type>()` constructors to cast the result to the proper type.

Note, however, that comma-separated initializers and probably some other operations may require additional explicit `Mat()` or `Mat_<T>()` constuctor calls to resolve possible ambiguity.

Below is the formal description of the `Mat` methods.

---

cv::Mat::Mat (view/add comments)

Various array constructors

```
(1) Mat::Mat();
(2) Mat::Mat(int rows, int cols, int type);
(3) Mat::Mat(Size size, int type);
(4) Mat::Mat(int rows, int cols, int type, const Scalar& s);
(5) Mat::Mat(Size size, int type, const Scalar& s);
(6) Mat::Mat(const Mat& m);
(7) Mat::Mat(int rows, int cols, int type, void* data, size_t
step=AUTO_STEP);
(8) Mat::Mat(Size size, int type, void* data, size_t step=AUTO_STEP);
(9) Mat::Mat(const Mat& m, const Range& rowRange, const Range&
colRange);
(10) Mat::Mat(const Mat& m, const Rect& roi);
(11) Mat::Mat(const CvMat* m, bool copyData=false);
(12) Mat::Mat(const IplImage* img, bool copyData=false);
(13) template<typename T, int n> explicit Mat::Mat(const Vec<T, n>&
vec, bool copyData=true);
(14) template<typename T, int m, int n> explicit Mat::Mat(const Matx<T,
m, n>& vec, bool copyData=true);
(15) template<typename T> explicit Mat::Mat(const vector<T>& vec, bool
copyData=false);
(16) Mat::Mat(const MatExpr& expr);
(17) Mat::Mat(int ndims, const int* sizes, int type);
(18) Mat::Mat(int ndims, const int* sizes, int type, const Scalar& s);
(19) Mat::Mat(int ndims, const int* sizes, int type, void* data, const
size_t* steps=0);
(20) Mat::Mat(const Mat& m, const Range* ranges);
```

ndims The array dimensionality

rows The number of rows in 2D array

cols The number of columns in 2D array

size The 2D array size: `Size(cols, rows)`. Note that in the `Size()` constructor the number of rows and the number of columns go in the reverse order.

sizes The array of integers, specifying the n-dimensional array shape

type The array type, use `CV_8UC1, ..., CV_64FC4` to create 1-4 channel matrices, or `CV_8UC(n), ..., CV_64FC(n)` to create multi-channel (up to `CV_MAX_CN` channels) matrices

s The optional value to initialize each matrix element with. To set all the matrix elements to the particular value after the construction, use the assignment operator `Mat::operator=(const Scalar& value)`.

data Pointer to the user data. Matrix constructors that take `data` and `step` parameters do not allocate matrix data. Instead, they just initialize the matrix header that points to the specified data, i.e. no data is copied. This operation is very efficient and can be used to process external data using OpenCV functions. The external data is not automatically deallocated, user should take care of it.

step The `data` buddy. This optional parameter specifies the number of bytes that each matrix row occupies. The value should include the padding bytes in the end of each row, if any. If the parameter is missing (set to `cv::AUTO_STEP`), no padding is assumed and the actual step is calculated as `cols*elemSize()`, see Mat::elemSize ().

steps The array of `ndims-1` steps in the case of multi-dimensional array (the last step is always set to the element size). If not specified, the matrix is assumed to be continuous.

m The array that (in whole, a partly) is assigned to the constructed matrix. No data is copied by these constructors. Instead, the header pointing to `m` data, or its sub-array, is constructed and the associated with it reference counter, if any, is incremented. That is, when you modify the matrix formed using such a constructor, you will also modify the corresponding elements of `m`. If you want to have an independent copy of the sub-array, use `Mat::clone()`.

img Pointer to the old-style `IplImage` image structure. By default, the data is shared between the original image and the new matrix, but when `copyData` is set, the full copy of the image data is created.

vec STL vector, which elements will form the matrix. The matrix will have a single column and the number of rows equal to the number of vector elements. Type of the matrix will match the type of vector elements. The constructor can handle arbitrary types, for which there is properly declared DataType , i.e. the vector elements must be primitive numbers or uni-type numerical tuples of numbers. Mixed-type structures are not supported, of course. Note that

the corresponding constructor is explicit, meaning that STL vectors are not automatically converted to `Mat` instances, you should write `Mat(vec)` explicitly. Another obvious note: unless you copied the data into the matrix (`copyData=true`), no new elements should be added to the vector, because it can potentially yield vector data reallocation, and thus the matrix data pointer will become invalid.

copyData Specifies, whether the underlying data of the STL vector, or the old-style `CvMat` or `IplImage` should be copied to (true) or shared with (false) the newly constructed matrix. When the data is copied, the allocated buffer will be managed using `Mat`'s reference counting mechanism. While when the data is shared, the reference counter will be NULL, and you should not deallocate the data until the matrix is not destructed.

rowRange The range of the `m`'s rows to take. As usual, the range start is inclusive and the range end is exclusive. Use `Range::all()` to take all the rows.

colRange The range of the `m`'s columns to take. Use `Range::all()` to take all the columns.

ranges The array of selected ranges of `m` along each dimensionality .

expr Matrix expression. See Matrix Expressions .

These are various constructors that form a matrix. As noticed in the often the default constructor is enough, and the proper matrix will be allocated by an OpenCV function. The constructed matrix can further be assigned to another matrix or matrix expression, in which case the old content is dereferenced, or be allocated with Mat::create .

---

cv::Mat::Mat (view/add comments)

Matrix destructor

```
Mat::~Mat();
```

The matrix destructor calls Mat::release .

---

cv::Mat::operator = (view/add comments)

Matrix assignment operators

```
Mat& Mat::operator = (const Mat& m);
Mat& Mat::operator = (const MatExpr_Base& expr);
Mat& operator = (const Scalar& s);
```

m The assigned, right-hand-side matrix. Matrix assignment is O(1) operation, that is, no data is copied. Instead, the data is shared and the reference counter, if any, is incremented. Before assigning new data, the old data is dereferenced via Mat::release .

expr The assigned matrix expression object. As opposite to the first form of assignment operation, the second form can reuse already allocated matrix if it has the right size and type to fit the matrix expression result. It is automatically handled by the real function that the matrix expressions is expanded to. For example, `C=A+B` is expanded to `cv::add(A, B, C)`, and cv::add will take care of automatic `C` reallocation.

s The scalar, assigned to each matrix element. The matrix size or type is not changed.

These are the available assignment operators, and they all are very different, so, please, look at the operator parameters description.

---

cv::Mat::operator MatExpr (view/add comments)

Mat-to-MatExpr cast operator

```
Mat::operator MatExpr_<Mat, Mat>() const;
```

The cast operator should not be called explicitly. It is used internally by the Matrix Expressions engine.

---

cv::Mat::row (view/add comments)

Makes a matrix header for the specified matrix row

```
Mat Mat::row(int i) const;
```

i the 0-based row index

The method makes a new header for the specified matrix row and returns it. This is O(1) operation, regardless of the matrix size. The underlying data of the new matrix will be shared with the original matrix. Here is the example of one of the classical basic matrix processing operations, axpy, used by LU and many other algorithms:

```
inline void matrix_axpy(Mat& A, int i, int j, double alpha)
{
    A.row(i) += A.row(j)*alpha;
}
```

Important note. In the current implementation the following code will not work as expected:

```
Mat A;
...
A.row(i) = A.row(j); // will not work
```

This is because `A.row(i)` forms a temporary header, which is further assigned another header. Remember, each of these operations is O(1), i.e. no data is copied. Thus, the above assignment will have absolutely no effect, while you may have expected j-th row being copied to i-th row. To achieve that, you should either turn this simple assignment into an expression, or use Mat::copyTo method:

```
Mat A;
...
// works, but looks a bit obscure.
A.row(i) = A.row(j) + 0;

// this is a bit longer, but the recommended method.
Mat Ai = A.row(i); M.row(j).copyTo(Ai);
```

---

cv::Mat::col (view/add comments)

Makes a matrix header for the specified matrix column

```
Mat Mat::col(int j) const;
```

j the 0-based column index

The method makes a new header for the specified matrix column and returns it. This is O(1) operation, regardless of the matrix size. The underlying data of the new matrix will be shared with the original matrix. See also Mat::row description.

cv::Mat::rowRange (view/add comments)

Makes a matrix header for the specified row span

```
Mat Mat::rowRange(int startrow, int endrow) const;
Mat Mat::rowRange(const Range& r) const;
```

startrow the 0-based start index of the row span

endrow the 0-based ending index of the row span

r The cv::Range structure containing both the start and the end indices

The method makes a new header for the specified row span of the matrix. Similarly to cv::Mat::row and cv::Mat::col, this is O(1) operation.

cv::Mat::colRange (view/add comments)

Makes a matrix header for the specified row span

```
Mat Mat::colRange(int startcol, int endcol) const;
Mat Mat::colRange(const Range& r) const;
```

startcol the 0-based start index of the column span

endcol the 0-based ending index of the column span

r The cv::Range structure containing both the start and the end indices

The method makes a new header for the specified column span of the matrix. Similarly to cv::Mat::row and cv::Mat::col, this is O(1) operation.

cv::Mat::diag (view/add comments)

Extracts diagonal from a matrix, or creates a diagonal matrix.

```
Mat Mat::diag(int d) const; static Mat Mat::diag(const Mat& matD);
```

d index of the diagonal, with the following meaning:

- **d=0** the main diagonal
- **d>0** a diagonal from the lower half, e.g. `d=1` means the diagonal immediately below the main one
- **d<0** a diagonal from the upper half, e.g. `d=1` means the diagonal immediately above the main one

matD single-column matrix that will form the diagonal matrix.

The method makes a new header for the specified matrix diagonal. The new matrix will be represented as a single-column matrix. Similarly to cv::Mat::row and cv::Mat::col, this is O(1) operation.

---

cv::Mat::clone (view/add comments)

Creates full copy of the array and the underlying data.

```
Mat Mat::clone() const;
```

The method creates full copy of the array. The original `step[]` are not taken into the account. That is, the array copy will be a continuous array occupying `total()*elemSize()` bytes.

---

cv::Mat::copyTo (view/add comments)

Copies the matrix to another one.

```
void Mat::copyTo( Mat& m ) const; void Mat::copyTo( Mat& m, const Mat&
mask ) const;
```

m The destination matrix. If it does not have a proper size or type before the operation, it will be reallocated

mask The operation mask. Its non-zero elements indicate, which matrix elements need to be copied

The method copies the matrix data to another matrix. Before copying the data, the method invokes

```
m.create(this->size(), this->type);
```

so that the destination matrix is reallocated if needed. While `m.copyTo(m);` will work as expected, i.e. will have no effect, the function does not handle the case of a partial overlap between the source and the destination matrices.

When the operation mask is specified, and the `Mat::create` call shown above reallocated the matrix, the newly allocated matrix is initialized with all 0's before copying the data.

---

cv::Mat::convertTo (view/add comments)

Converts array to another datatype with optional scaling.

```
void Mat::convertTo( Mat& m, int rtype, double alpha=1, double beta=0 )
const;
```

m The destination matrix. If it does not have a proper size or type before the operation, it will be reallocated

rtype The desired destination matrix type, or rather, the depth (since the number of channels will be the same with the source one). If `rtype` is negative, the destination matrix will have the same type as the source.

alpha The optional scale factor

beta The optional delta, added to the scaled values.

The method converts source pixel values to the target datatype. `saturate_cast<>` is applied in the end to avoid possible overflows:

$$m(x,y) = saturate_cast < rType > (\alpha(*this)(x,y) + \beta)$$

cv::Mat::assignTo (view/add comments)

Functional form of convertTo

```
void Mat::assignTo( Mat& m, int type=-1 ) const;
```

m The destination array

type The desired destination array depth (or -1 if it should be the same as the source one).

This is internal-use method called by the Matrix Expressions engine.

cv::Mat::setTo (view/add comments)

Sets all or some of the array elements to the specified value.

```
Mat& Mat::setTo(const Scalar& s, const Mat& mask=Mat());
```

s Assigned scalar, which is converted to the actual array type

mask The operation mask of the same size as `*this`

This is the advanced variant of `Mat::operator=(const Scalar& s)` operator.

cv::Mat::reshape (view/add comments)

Changes the 2D matrix's shape and/or the number of channels without copying the data.

```
Mat Mat::reshape(int cn, int rows=0) const;
```

cn The new number of channels. If the parameter is 0, the number of channels remains the same.

rows The new number of rows. If the parameter is 0, the number of rows remains the same.

The method makes a new matrix header for `*this` elements. The new matrix may have different size and/or different number of channels. Any combination is possible, as long as:

1. No extra elements is included into the new matrix and no elements are excluded. Consequently, the product `rows*cols*channels()` must stay the same after the transformation.
2. No data is copied, i.e. this is O(1) operation. Consequently, if you change the number of rows, or the operation changes elements' row indices in some other way, the matrix must be continuous. See cv::Mat::isContinuous.

Here is some small example. Assuming, there is a set of 3D points that are stored as STL vector, and you want to represent the points as `3xN` matrix. Here is how it can be done:

```
std::vector<cv::Point3f> vec;
...

Mat pointMat = Mat(vec). // convert vector to Mat, O(1) operation
                  reshape(1). // make Nx3 1-channel matrix out of Nx1 3-channel.
                              // Also, an O(1) operation
                     t(); // finally, transpose the Nx3 matrix.
                          // This involves copying of all the elements
```

---

cv::Mat::t (view/add comments)

Transposes the matrix

```
MatExpr Mat::t() const;
```

The method performs matrix transposition by means of matrix expressions. It does not perform the actual transposition, but returns a temporary "matrix transposition" object that can be further used as a part of more complex matrix expression or be assigned to a matrix:

```
Mat A1 = A + Mat::eye(A.size(), A.type)*lambda;
Mat C = A1.t()*A1; // compute (A + lambda*I)^t * (A + lamda*I)
```

---

cv::Mat::inv (view/add comments)

Inverses the matrix

```
MatExpr Mat::inv(int method=DECOMP_LU) const;
```

method The matrix inversion method, one of

DECOMP_LU LU decomposition. The matrix must be non-singular

DECOMP_CHOLESKY Cholesky LL^T decomposition, for symmetrical positively defined matrices only. About twice faster than LU on big matrices.

DECOMP_SVD SVD decomposition. The matrix can be a singular or even non-square, then the pseudo-inverse is computed

The method performs matrix inversion by means of matrix expressions, i.e. a temporary ”matrix inversion” object is returned by the method, and can further be used as a part of more complex matrix expression or be assigned to a matrix.

---

cv::Mat::mul (view/add comments)

Performs element-wise multiplication or division of the two matrices

```
MatExpr Mat::mul(const Mat& m, double scale=1) const;
MatExpr Mat::mul(const MatExpr& m, double scale=1) const;
```

m Another matrix, of the same type and the same size as `*this`, or a matrix expression

scale The optional scale factor

The method returns a temporary object encoding per-element array multiplication, with optional scale. Note that this is not a matrix multiplication, which corresponds to a simpler ”*” operator.

Here is a example:

```
Mat C = A.mul(5/B); // equivalent to divide(A, B, C, 5)
```

cv::Mat::cross (view/add comments)

Computes cross-product of two 3-element vectors

```
Mat Mat::cross(const Mat& m) const;
```

m Another cross-product operand

The method computes cross-product of the two 3-element vectors. The vectors must be 3-elements floating-point vectors of the same shape and the same size. The result will be another 3-element vector of the same shape and the same type as operands.

cv::Mat::dot (view/add comments)

Computes dot-product of two vectors

```
double Mat::dot(const Mat& m) const;
```

m Another dot-product operand.

The method computes dot-product of the two matrices. If the matrices are not single-column or single-row vectors, the top-to-bottom left-to-right scan ordering is used to treat them as 1D vectors. The vectors must have the same size and the same type. If the matrices have more than one channel, the dot products from all the channels are summed together.

cv::Mat::zeros (view/add comments)

Returns zero array of the specified size and type

```
static MatExpr Mat::zeros(int rows, int cols, int type); static MatExpr
Mat::zeros(Size size, int type); static MatExpr Mat::zeros(int ndims,
const int* sizes, int type);
```

ndims The array dimensionality

rows The number of rows

cols The number of columns

size Alternative matrix size specification: `Size(cols, rows)`

sizes The array of integers, specifying the array shape

type The created matrix type

The method returns Matlab-style zero array initializer. It can be used to quickly form a constant array and use it as a function parameter, as a part of matrix expression, or as a matrix initializer.

```
Mat A;
A = Mat::zeros(3, 3, CV_32F);
```

Note that in the above sample a new matrix will be allocated only if `A` is not 3x3 floating-point matrix, otherwise the existing matrix `A` will be filled with 0's.

---

cv::Mat::ones (view/add comments)

Returns array of all 1's of the specified size and type

```
static MatExpr Mat::ones(int rows, int cols, int type); static MatExpr
Mat::ones(Size size, int type); static MatExpr Mat::ones(int ndims,
const int* sizes, int type);
```

ndims The array dimensionality

rows The number of rows

cols The number of columns

size Alternative matrix size specification: `Size(cols, rows)`

sizes The array of integers, specifying the array shape

type The created matrix type

The method returns Matlab-style ones' array initializer, similarly to cv::Mat::zeros. Note that using this method you can initialize an array with arbitrary value, using the following Matlab idiom:

```
Mat A = Mat::ones(100, 100, CV_8U)*3; // make 100x100 matrix filled with 3.
```

The above operation will not form 100x100 matrix of ones and then multiply it by 3. Instead, it will just remember the scale factor (3 in this case) and use it when actually invoking the matrix initializer.

---

cv::Mat::eye (view/add comments)

Returns identity matrix of the specified size and type

```
static MatExpr Mat::eye(int rows, int cols, int type); static MatExpr
Mat::eye(Size size, int type);
```

rows The number of rows

cols The number of columns

size Alternative matrix size specification: `Size(cols, rows)`

type The created matrix type

The method returns Matlab-style identity matrix initializer, similarly to cv::Mat::zeros. Similarly to `Mat::ones`, you can use a scale operation to create a scaled identity matrix efficiently:

```
// make a 4x4 diagonal matrix with 0.1's on the diagonal.
Mat A = Mat::eye(4, 4, CV_32F)*0.1;
```

---

cv::Mat::create (view/add comments)

Allocates new array data if needed.

```
void Mat::create(int rows, int cols, int type); void Mat::create(Size
size, int type); void Mat::create(int ndims, const int* sizes, int
type);
```

ndims The new array dimensionality

rows The new number of rows

cols The new number of columns

size Alternative new matrix size specification: `Size(cols, rows)`

sizes The array of integers, specifying the new array shape

type The new matrix type

This is one of the key `Mat` methods. Most new-style OpenCV functions and methods that produce arrays call this method for each output array. The method uses the following algorithm:

1. if the current array shape and the type match the new ones, return immediately.
2. otherwise, dereference the previous data by calling cv::Mat::release
3. initialize the new header
4. allocate the new data of `total()*elemSize()` bytes
5. allocate the new, associated with the data, reference counter and set it to 1.

Such a scheme makes the memory management robust and efficient at the same time, and also saves quite a bit of typing for the user, i.e. usually there is no need to explicitly allocate output arrays. That is, instead of writing:

```
Mat color;
...
Mat gray(color.rows, color.cols, color.depth());
cvtColor(color, gray, CV_BGR2GRAY);
```

you can simply write:

```
Mat color;
...
Mat gray;
cvtColor(color, gray, CV_BGR2GRAY);
```

because `cvtColor`, as well as most of OpenCV functions, calls Mat::create() for the output array internally.

---

cv::Mat::addref (view/add comments)

Increments the reference counter

```
void Mat::addref();
```

The method increments the reference counter, associated with the matrix data. If the matrix header points to an external data (see cv::Mat::Mat), the reference counter is NULL, and the method has no effect in this case. Normally, the method should not be called explicitly, to avoid memory leaks. It is called implicitly by the matrix assignment operator. The reference counter increment is the atomic operation on the platforms that support it, thus it is safe to operate on the same matrices asynchronously in different threads.

---

cv::Mat::release (view/add comments)

Decrements the reference counter and deallocates the matrix if needed

```
void Mat::release();
```

The method decrements the reference counter, associated with the matrix data. When the reference counter reaches 0, the matrix data is deallocated and the data and the reference counter pointers are set to NULL's. If the matrix header points to an external data (see cv::Mat::Mat), the reference counter is NULL, and the method has no effect in this case.

This method can be called manually to force the matrix data deallocation. But since this method is automatically called in the destructor, or by any other method that changes the data pointer, it is usually not needed. The reference counter decrement and check for 0 is the atomic operation on the platforms that support it, thus it is safe to operate on the same matrices asynchronously in different threads.

---

cv::Mat::resize (view/add comments)

Changes the number of matrix rows

```
void Mat::resize( size_t sz ) const;
```

sz The new number of rows

The method changes the number of matrix rows. If the matrix is reallocated, the first `min(Mat::rows, sz)` rows are preserved. The method emulates the corresponding method of STL vector class.

Mat::push_back (view/add comments)

Adds elements to the bottom of the matrix

```
template<typename T> void Mat::push_back(const T& elem);
template<typename T> void Mat::push_back(const Mat_<T>& elem);
```

elem The added element(s).

The methods add one or more elements to the bottom of the matrix. They emulate the corresponding method of STL vector class. When `elem` is `Mat`, its type and the number of columns must be the same as in the container matrix.

Mat::pop_back (view/add comments)

Removes elements from the bottom of the matrix.

```
template<typename T> void Mat::pop_back(size_t nelems=1);
```

nelems The number of rows removed. If it is greater than the total number of rows, an exception is thrown.

The method removes one or more rows from the bottom of the matrix.

---

cv::Mat::locateROI (view/add comments)

Locates matrix header within a parent matrix

```
void Mat::locateROI( Size& wholeSize, Point& ofs ) const;
```

wholeSize The output parameter that will contain size of the whole matrix, which `*this` is a part of.

ofs The output parameter that will contain offset of `*this` inside the whole matrix

After you extracted a submatrix from a matrix using cv::Mat::row, cv::Mat::col, cv::Mat::rowRange, cv::Mat::colRange etc., the result submatrix will point just to the part of the original big matrix. However, each submatrix contains some information (represented by `datastart` and `dataend` fields), using which it is possible to reconstruct the original matrix size and the position of the extracted submatrix within the original matrix. The method `locateROI` does exactly that.

---

cv::Mat::adjustROI (view/add comments)

Adjust submatrix size and position within the parent matrix

```
Mat& Mat::adjustROI( int dtop, int dbottom, int dleft, int dright );
```

dtop The shift of the top submatrix boundary upwards

dbottom The shift of the bottom submatrix boundary downwards

dleft The shift of the left submatrix boundary to the left

dright The shift of the right submatrix boundary to the right

The method is complimentary to the cv::Mat::locateROI. Indeed, the typical use of these functions is to determine the submatrix position within the parent matrix and then shift the position somehow. Typically it can be needed for filtering operations, when pixels outside of the ROI should be taken into account. When all the method's parameters are positive, it means that the ROI needs to grow in all directions by the specified amount, i.e.

```
A.adjustROI(2, 2, 2, 2);
```

increases the matrix size by 4 elements in each direction and shifts it by 2 elements to the left and 2 elements up, which brings in all the necessary pixels for the filtering with 5x5 kernel.

It's user responsibility to make sure that adjustROI does not cross the parent matrix boundary. If it does, the function will signal an error.

The function is used internally by the OpenCV filtering functions, like cv::filter2D, morphological operations etc.

See also cv::copyMakeBorder.

cv::Mat::operator() (view/add comments)

Extracts a rectangular submatrix

```
Mat Mat::operator()( Range rowRange, Range colRange ) const;
Mat Mat::operator()( const Rect& roi ) const; Mat Mat::operator()(
const Ranges* ranges ) const;
```

rowRange The start and the end row of the extracted submatrix. The upper boundary is not included. To select all the rows, use `Range::all()`

colRange The start and the end column of the extracted submatrix. The upper boundary is not included. To select all the columns, use `Range::all()`

roi The extracted submatrix specified as a rectangle

ranges The array of selected ranges along each array dimension

The operators make a new header for the specified sub-array of `*this`. They are the most generalized forms of cv::Mat::row, cv::Mat::col, cv::Mat::rowRange and cv::Mat::colRange. For example, `A(Range(0, 10), Range::all())` is equivalent to `A.rowRange(0, 10)`. Similarly to all of the above, the operators are O(1) operations, i.e. no matrix data is copied.

cv::Mat::operator CvMat (view/add comments)

Creates CvMat header for the matrix

```
Mat::operator CvMat() const;
```

The operator makes CvMat header for the matrix without copying the underlying data. The reference counter is not taken into account by this operation, thus you should make sure than the original matrix is not deallocated while the `CvMat` header is used. The operator is useful for intermixing the new and the old OpenCV API's, e.g:

```
Mat img(Size(320, 240), CV_8UC3);
...
```

```
CvMat cvimg = img;
mycvOldFunc( &cvimg, ...);
```

where `mycvOldFunc` is some function written to work with OpenCV 1.x data structures.

---

cv::Mat::operator IplImage (view/add comments)

Creates IplImage header for the matrix

```
Mat::operator IplImage() const;
```

The operator makes IplImage header for the matrix without copying the underlying data. You should make sure than the original matrix is not deallocated while the `IplImage` header is used. Similarly to `Mat::operator CvMat`, the operator is useful for intermixing the new and the old OpenCV API's.

---

cv::Mat::total (view/add comments)

Returns the total number of array elements.

```
size_t Mat::total() const;
```

The method returns the number of array elements (e.g. number of pixels if the array represents an image).

---

cv::Mat::isContinuous (view/add comments)

Reports whether the matrix is continuous or not

```
bool Mat::isContinuous() const;
```

The method returns true if the matrix elements are stored continuously, i.e. without gaps in the end of each row, and false otherwise. Obviously, `1x1` or `1xN` matrices are always continuous. Matrices created with cv::Mat::create are always continuous, but if you extract a part of the matrix using cv::Mat::col, cv::Mat::diag etc. or constructed a matrix header for externally allocated data, such matrices may no longer have this property.

The continuity flag is stored as a bit in `Mat::flags` field, and is computed automatically when you construct a matrix header, thus the continuity check is very fast operation, though it could be, in theory, done as following:

```
// alternative implementation of Mat::isContinuous()
bool myCheckMatContinuity(const Mat& m)
{
    //return (m.flags & Mat::CONTINUOUS_FLAG) != 0;
    return m.rows == 1 || m.step == m.cols*m.elemSize();
}
```

The method is used in a quite a few of OpenCV functions, and you are welcome to use it as well. The point is that element-wise operations (such as arithmetic and logical operations, math functions, alpha blending, color space transformations etc.) do not depend on the image geometry, and thus, if all the input and all the output arrays are continuous, the functions can process them as very long single-row vectors. Here is the example of how alpha-blending function can be implemented.

```
template<typename T>
void alphaBlendRGBA(const Mat& src1, const Mat& src2, Mat& dst)
{
    const float alpha_scale = (float)std::numeric_limits<T>::max(),
                inv_scale = 1.f/alpha_scale;

    CV_Assert( src1.type() == src2.type() &&
               src1.type() == CV_MAKETYPE(DataType<T>::depth, 4) &&
               src1.size() == src2.size());
    Size size = src1.size();
    dst.create(size, src1.type());

    // here is the idiom: check the arrays for continuity and,
    // if this is the case,
    // treat the arrays as 1D vectors
    if( src1.isContinuous() && src2.isContinuous() && dst.isContinuous() )
    {
        size.width *= size.height;
        size.height = 1;
    }
    size.width *= 4;
```

```
    for( int i = 0; i < size.height; i++ )
    {
        // when the arrays are continuous,
        // the outer loop is executed only once
        const T* ptr1 = src1.ptr<T>(i);
        const T* ptr2 = src2.ptr<T>(i);
        T* dptr = dst.ptr<T>(i);

        for( int j = 0; j < size.width; j += 4 )
        {
            float alpha = ptr1[j+3]*inv_scale, beta = ptr2[j+3]*inv_scale;
            dptr[j] = saturate_cast<T>(ptr1[j]*alpha + ptr2[j]*beta);
            dptr[j+1] = saturate_cast<T>(ptr1[j+1]*alpha + ptr2[j+1]*beta);
            dptr[j+2] = saturate_cast<T>(ptr1[j+2]*alpha + ptr2[j+2]*beta);
            dptr[j+3] = saturate_cast<T>((1 - (1-alpha)*(1-beta))*alpha_scale);
        }
    }
}
```

This trick, while being very simple, can boost performance of a simple element-operation by 10-20 percents, especially if the image is rather small and the operation is quite simple.

Also, note that we use another OpenCV idiom in this function - we call cv::Mat::create for the destination array instead of checking that it already has the proper size and type. And while the newly allocated arrays are always continuous, we still check the destination array, because cv::create does not always allocate a new matrix.

---

cv::Mat::elemSize (view/add comments)

Returns matrix element size in bytes

```
size_t Mat::elemSize() const;
```

The method returns the matrix element size in bytes. For example, if the matrix type is `CV_16SC3`, the method will return `3*sizeof(short)` or 6.

---

cv::Mat::elemSize1 (view/add comments)

Returns size of each matrix element channel in bytes

```
size_t Mat::elemSize1() const;
```

The method returns the matrix element channel size in bytes, that is, it ignores the number of channels. For example, if the matrix type is `CV_16SC3`, the method will return `sizeof(short)` or 2.

---

cv::Mat::type (view/add comments)

Returns matrix element type

```
int Mat::type() const;
```

The method returns the matrix element type, an id, compatible with the `CvMat` type system, like `CV_16SC3` or 16-bit signed 3-channel array etc.

---

cv::Mat::depth (view/add comments)

Returns matrix element depth

```
int Mat::depth() const;
```

The method returns the matrix element depth id, i.e. the type of each individual channel. For example, for 16-bit signed 3-channel array the method will return `CV_16S`. The complete list of matrix types:

- `CV_8U` - 8-bit unsigned integers (`0..255`)
- `CV_8S` - 8-bit signed integers (`-128..127`)
- `CV_16U` - 16-bit unsigned integers (`0..65535`)
- `CV_16S` - 16-bit signed integers (`-32768..32767`)
- `CV_32S` - 32-bit signed integers (`-2147483648..2147483647`)

- `CV_32F` - 32-bit floating-point numbers (`-FLT_MAX..FLT_MAX, INF, NAN`)
- `CV_64F` - 64-bit floating-point numbers (`-DBL_MAX..DBL_MAX, INF, NAN`)

---

cv::Mat::channels (view/add comments)

Returns matrix element depth

```
int Mat::channels() const;
```

The method returns the number of matrix channels.

---

cv::Mat::step1 (view/add comments)

Returns normalized step

```
size_t Mat::step1() const;
```

The method returns the matrix step, divided by cv::Mat::elemSize1(). It can be useful for fast access to arbitrary matrix element.

---

cv::Mat::size (view/add comments)

Returns the matrix size

```
Size Mat::size() const;
```

The method returns the matrix size: `Size(cols, rows)`.

cv::Mat::empty (view/add comments)

Returns true if the array has no elemens

```
bool Mat::empty() const;
```

The method returns true if `Mat::total()` is 0 or if `Mat::data` is NULL. Because of `pop_back()` and `resize()` methods `M.total() == 0` does not imply that `M.data == NULL`.

cv::Mat::ptr (view/add comments)

Return pointer to the specified matrix row

```
uchar* Mat::ptr(int i=0);
const uchar* Mat::ptr(int i=0) const;
template<typename _Tp> _Tp* Mat::ptr(int i=0);
template<typename _Tp> const _Tp* Mat::ptr(int i=0) const;
```

i The 0-based row index

The methods return `uchar*` or typed pointer to the specified matrix row. See the sample in cv::Mat::isContinuous() on how to use these methods.

cv::Mat::at (view/add comments)

Return reference to the specified array element

```
template<typename T> T& Mat::at(int i) const;
template<typename T> const T& Mat::at(int i) const;

          template<typename T> T& Mat::at(int i, int j);
template<typename T> const T& Mat::at(int i, int j) const;
```

```
          template<typename T> T& Mat::at(Point pt);
template<typename T> const T& Mat::at(Point pt) const;

          template<typename T> T& Mat::at(int i, int j, int k);
template<typename T> const T& Mat::at(int i, int j, int k) const;

          template<typename T> T& Mat::at(const int* idx);
template<typename T> const T& Mat::at(const int* idx) const;
```

i, j, k Indices along the dimensions 0, 1 and 2, respectively

pt The element position specified as `Point(j,i)`

idx The array of `Mat::dims` indices

The template methods return reference to the specified array element. For the sake of higher performance the index range checks are only performed in Debug configuration.

Note that the variants with a single index (i) can be used to access elements of single-row or single-column 2-dimensional arrays. That is, if, for example, `A` is `1 x N` floating-point matrix and `B` is `M x 1` integer matrix, you can simply write `A.at<float>(k+4)` and `B.at<int>(2*i+1)` instead of `A.at<float>(0,k+4)` and `B.at<int>(2*i+1,0)`, respectively.

Here is an example of initialization of a Hilbert matrix:

```
Mat H(100, 100, CV_64F);
for(int i = 0; i < H.rows; i++)
    for(int j = 0; j < H.cols; j++)
        H.at<double>(i,j)=1./(i+j+1);
```

---

cv::Mat::begin (view/add comments)

Return the matrix iterator, set to the first matrix element

```
template<typename _Tp> MatIterator_<_Tp> Mat::begin(); template<typename
_Tp> MatConstIterator_<_Tp> Mat::begin() const;
```

The methods return the matrix read-only or read-write iterators. The use of matrix iterators is very similar to the use of bi-directional STL iterators. Here is the alpha blending function rewritten using the matrix iterators:

```
template<typename T>
void alphaBlendRGBA(const Mat& src1, const Mat& src2, Mat& dst)
{
    typedef Vec<T, 4> VT;

    const float alpha_scale = (float)std::numeric_limits<T>::max(),
                inv_scale = 1.f/alpha_scale;

    CV_Assert( src1.type() == src2.type() &&
               src1.type() == DataType<VT>::type &&
               src1.size() == src2.size());
    Size size = src1.size();
    dst.create(size, src1.type());

    MatConstIterator_<VT> it1 = src1.begin<VT>(), it1_end = src1.end<VT>();
    MatConstIterator_<VT> it2 = src2.begin<VT>();
    MatIterator_<VT> dst_it = dst.begin<VT>();

    for( ; it1 != it1_end; ++it1, ++it2, ++dst_it )
    {
        VT pix1 = *it1, pix2 = *it2;
        float alpha = pix1[3]*inv_scale, beta = pix2[3]*inv_scale;
        *dst_it = VT(saturate_cast<T>(pix1[0]*alpha + pix2[0]*beta),
                     saturate_cast<T>(pix1[1]*alpha + pix2[1]*beta),
                     saturate_cast<T>(pix1[2]*alpha + pix2[2]*beta),
                     saturate_cast<T>((1 - (1-alpha)*(1-beta))*alpha_scale));
    }
}
```

---

cv::Mat::end (view/add comments)

Return the matrix iterator, set to the after-last matrix element

```
template<typename _Tp> MatIterator_<_Tp> Mat::end(); template<typename
_Tp> MatConstIterator_<_Tp> Mat::end() const;
```

The methods return the matrix read-only or read-write iterators, set to the point following the last matrix element.

Mat_

Template matrix class derived from Mat

```
template<typename _Tp> class Mat_ : public Mat
{
public:
    // ... some specific methods
    //         and
    // no new extra fields
};
```

The class Mat_<_Tp> is a ”thin” template wrapper on top of Mat class. It does not have any extra data fields, nor it or Mat have any virtual methods and thus references or pointers to these two classes can be freely converted one to another. But do it with care, e.g.:

```
// create 100x100 8-bit matrix
Mat M(100,100,CV_8U);
// this will compile fine. no any data conversion will be done.
Mat_<float>& M1 = (Mat_<float>&)M;
// the program will likely crash at the statement below
M1(99,99) = 1.f;
```

While Mat is sufficient in most cases, Mat_ can be more convenient if you use a lot of element access operations and if you know matrix type at compile time. Note that Mat::at<_Tp>(int y, int x) and Mat_<_Tp>::operator ()(int y, int x) do absolutely the same and run at the same speed, but the latter is certainly shorter:

```
Mat_<double> M(20,20);
for(int i = 0; i < M.rows; i++)
    for(int j = 0; j < M.cols; j++)
        M(i,j) = 1./(i+j+1);
Mat E, V;
eigen(M,E,V);
cout << E.at<double>(0,0)/E.at<double>(M.rows-1,0);
```

How to use Mat_ for multi-channel images/matrices?

This is simple - just pass Vec as Mat_ parameter:

```
// allocate 320x240 color image and fill it with green (in RGB space)
Mat_<Vec3b> img(240, 320, Vec3b(0,255,0));
// now draw a diagonal white line
for(int i = 0; i < 100; i++)
    img(i,i)=Vec3b(255,255,255);
// and now scramble the 2nd (red) channel of each pixel
for(int i = 0; i < img.rows; i++)
    for(int j = 0; j < img.cols; j++)
        img(i,j)[2] ^= (uchar)(i ^ j);
```

NAryMatIterator

n-ary multi-dimensional array iterator

```
class CV_EXPORTS NAryMatIterator
{
public:
    //! the default constructor
    NAryMatIterator();
    //! the full constructor taking arbitrary number of n-dim matrices
    NAryMatIterator(const Mat** arrays, Mat* planes, int narrays=-1);
    //! the separate iterator initialization method
    void init(const Mat** arrays, Mat* planes, int narrays=-1);

    //! proceeds to the next plane of every iterated matrix
    NAryMatIterator& operator ++();
    //! proceeds to the next plane of every iterated matrix (postfix increment operator)
    NAryMatIterator operator ++(int);

    ...
    int nplanes; // the total number of planes
};
```

The class is used for implementation of unary, binary and, generally, n-ary element-wise operations on multi-dimensional arrays. Some of the arguments of n-ary function may be continuous arrays, some may be not. It is possible to use conventional MatIterator 's for each array, but it can be a big overhead to increment all of the iterators after each small operations. That's where `NAryMatIterator` can be used. Using it, you can iterate though several matrices simultaneously as long as they have the same geometry (dimensionality and all the dimension sizes are the same). On each iteration `it.planes[0]`, `it.planes[1]`, ... will be the slices of the corresponding matrices.

Here is an example of how you can compute a normalized and thresholded 3D color histogram:

```
void computeNormalizedColorHist(const Mat& image, Mat& hist, int N, double minProb)
{
    const int histSize[] = {N, N, N};

    // make sure that the histogram has proper size and type
    hist.create(3, histSize, CV_32F);

    // and clear it
    hist = Scalar(0);

    // the loop below assumes that the image
    // is 8-bit 3-channel, so let's check it.
```

```
    CV_Assert(image.type() == CV_8UC3);
    MatConstIterator_<Vec3b> it = image.begin<Vec3b>(),
                             it_end = image.end<Vec3b>();
    for( ; it != it_end; ++it )
    {
        const Vec3b& pix = *it;
        hist.at<float>(pix[0]*N/256, pix[1]*N/256, pix[2]*N/256) += 1.f;
    }

    minProb *= image.rows*image.cols;
    Mat plane;
    NAryMatIterator it(&hist, &plane, 1);
    double s = 0;
    // iterate through the matrix. on each iteration
    // it.planes[*] (of type Mat) will be set to the current plane.
    for(int p = 0; p < it.nplanes; p++, ++it)
    {
        threshold(it.planes[0], it.planes[0], minProb, 0, THRESH_TOZERO);
        s += sum(it.planes[0])[0];
    }

    s = 1./s;
    it = NAryMatIterator(&hist, &plane, 1);
    for(int p = 0; p < it.nplanes; p++, ++it)
        it.planes[0] *= s;
}
```

SparseMat

Sparse n-dimensional array.

```
class SparseMat
{
public:
    typedef SparseMatIterator iterator;
    typedef SparseMatConstIterator const_iterator;

    // internal structure - sparse matrix header
    struct Hdr
    {
        ...
    };

    // sparse matrix node - element of a hash table
```

```
struct Node
{
    size_t hashval;
    size_t next;
    int idx[CV_MAX_DIM];
};

////////// constructors and destructor //////////
// default constructor
SparseMat();
// creates matrix of the specified size and type
SparseMat(int dims, const int* _sizes, int _type);
// copy constructor
SparseMat(const SparseMat& m);
// converts dense array to the sparse form,
// if try1d is true and matrix is a single-column matrix (Nx1),
// then the sparse matrix will be 1-dimensional.
SparseMat(const Mat& m, bool try1d=false);
// converts old-style sparse matrix to the new-style.
// all the data is copied, so that "m" can be safely
// deleted after the conversion
SparseMat(const CvSparseMat* m);
// destructor
~SparseMat();

///////// assignment operations ///////////

// this is O(1) operation; no data is copied
SparseMat& operator = (const SparseMat& m);
// (equivalent to the corresponding constructor with try1d=false)
SparseMat& operator = (const Mat& m);

// creates full copy of the matrix
SparseMat clone() const;

// copy all the data to the destination matrix.
// the destination will be reallocated if needed.
void copyTo( SparseMat& m ) const;
// converts 1D or 2D sparse matrix to dense 2D matrix.
// If the sparse matrix is 1D, then the result will
// be a single-column matrix.
void copyTo( Mat& m ) const;
// converts arbitrary sparse matrix to dense matrix.
// multiplies all the matrix elements by the specified scalar
void convertTo( SparseMat& m, int rtype, double alpha=1 ) const;
```

```
    // converts sparse matrix to dense matrix with optional type conversion and scaling.
    // When rtype=-1, the destination element type will be the same
    // as the sparse matrix element type.
    // Otherwise rtype will specify the depth and
    // the number of channels will remain the same is in the sparse matrix
    void convertTo( Mat& m, int rtype, double alpha=1, double beta=0 ) const;

    // not used now
    void assignTo( SparseMat& m, int type=-1 ) const;

    // reallocates sparse matrix. If it was already of the proper size and type,
    // it is simply cleared with clear(), otherwise,
    // the old matrix is released (using release()) and the new one is allocated.
    void create(int dims, const int* _sizes, int _type);
    // sets all the matrix elements to 0, which means clearing the hash table.
    void clear();
    // manually increases reference counter to the header.
    void addref();
    // decreses the header reference counter, when it reaches 0,
    // the header and all the underlying data are deallocated.
    void release();

    // converts sparse matrix to the old-style representation.
    // all the elements are copied.
    operator CvSparseMat*() const;
    // size of each element in bytes
    // (the matrix nodes will be bigger because of
    //  element indices and other SparseMat::Node elements).
    size_t elemSize() const;
    // elemSize()/channels()
    size_t elemSize1() const;

    // the same is in Mat
    int type() const;
    int depth() const;
    int channels() const;

    // returns the array of sizes and 0 if the matrix is not allocated
    const int* size() const;
    // returns i-th size (or 0)
    int size(int i) const;
    // returns the matrix dimensionality
    int dims() const;
    // returns the number of non-zero elements
    size_t nzcount() const;
```

```
// compute element hash value from the element indices:
// 1D case
size_t hash(int i0) const;
// 2D case
size_t hash(int i0, int i1) const;
// 3D case
size_t hash(int i0, int i1, int i2) const;
// n-D case
size_t hash(const int* idx) const;

// low-level element-acccess functions,
// special variants for 1D, 2D, 3D cases and the generic one for n-D case.
//
// return pointer to the matrix element.
//  if the element is there (it's non-zero), the pointer to it is returned
//  if it's not there and createMissing=false, NULL pointer is returned
//  if it's not there and createMissing=true, then the new element
//    is created and initialized with 0. Pointer to it is returned
//  If the optional hashval pointer is not NULL, the element hash value is
//  not computed, but *hashval is taken instead.
uchar* ptr(int i0, bool createMissing, size_t* hashval=0);
uchar* ptr(int i0, int i1, bool createMissing, size_t* hashval=0);
uchar* ptr(int i0, int i1, int i2, bool createMissing, size_t* hashval=0);
uchar* ptr(const int* idx, bool createMissing, size_t* hashval=0);

// higher-level element access functions:
// ref<_Tp>(i0,...[,hashval]) - equivalent to *(_Tp*)ptr(i0,...true[,hashval]).
//    always return valid reference to the element.
//    If it's did not exist, it is created.
// find<_Tp>(i0,...[,hashval]) - equivalent to (_const Tp*)ptr(i0,...false[,hashval]).
//    return pointer to the element or NULL pointer if the element is not there.
// value<_Tp>(i0,...[,hashval]) - equivalent to
//    { const _Tp* p = find<_Tp>(i0,...[,hashval]); return p ? *p : _Tp(); }
//    that is, 0 is returned when the element is not there.
// note that _Tp must match the actual matrix type -
// the functions do not do any on-fly type conversion

// 1D case
template<typename _Tp> _Tp& ref(int i0, size_t* hashval=0);
template<typename _Tp> _Tp value(int i0, size_t* hashval=0) const;
template<typename _Tp> const _Tp* find(int i0, size_t* hashval=0) const;

// 2D case
template<typename _Tp> _Tp& ref(int i0, int i1, size_t* hashval=0);
```

```
    template<typename _Tp> _Tp value(int i0, int i1, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(int i0, int i1, size_t* hashval=0) const;

    // 3D case
    template<typename _Tp> _Tp& ref(int i0, int i1, int i2, size_t* hashval=0);
    template<typename _Tp> _Tp value(int i0, int i1, int i2, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(int i0, int i1, int i2, size_t* hashval=0) const

    // n-D case
    template<typename _Tp> _Tp& ref(const int* idx, size_t* hashval=0);
    template<typename _Tp> _Tp value(const int* idx, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(const int* idx, size_t* hashval=0) const;

    // erase the specified matrix element.
    // When there is no such element, the methods do nothing
    void erase(int i0, int i1, size_t* hashval=0);
    void erase(int i0, int i1, int i2, size_t* hashval=0);
    void erase(const int* idx, size_t* hashval=0);

    // return the matrix iterators,
    //   pointing to the first sparse matrix element,
    SparseMatIterator begin();
    SparseMatConstIterator begin() const;
    //   ... or to the point after the last sparse matrix element
    SparseMatIterator end();
    SparseMatConstIterator end() const;

    // and the template forms of the above methods.
    // _Tp must match the actual matrix type.
    template<typename _Tp> SparseMatIterator_<_Tp> begin();
    template<typename _Tp> SparseMatConstIterator_<_Tp> begin() const;
    template<typename _Tp> SparseMatIterator_<_Tp> end();
    template<typename _Tp> SparseMatConstIterator_<_Tp> end() const;

    // return value stored in the sparse martix node
    template<typename _Tp> _Tp& value(Node* n);
    template<typename _Tp> const _Tp& value(const Node* n) const;

    /////////////// some internal-use methods ///////////////
    ...

    // pointer to the sparse matrix header
    Hdr* hdr;
};
```

The class `SparseMat` represents multi-dimensional sparse numerical arrays. Such a sparse array can store elements of any type that Mat can store. "Sparse" means that only non-zero elements are stored (though, as a result of operations on a sparse matrix, some of its stored elements can actually become 0. It's up to the user to detect such elements and delete them using `SparseMat::erase`). The non-zero elements are stored in a hash table that grows when it's filled enough, so that the search time is O(1) in average (regardless of whether element is there or not). Elements can be accessed using the following methods:

1. query operations (`SparseMat::ptr` and the higher-level `SparseMat::ref`, `SparseMat::value` and `SparseMat::find`), e.g.:

```
const int dims = 5;
int size[] = {10, 10, 10, 10, 10};
SparseMat sparse_mat(dims, size, CV_32F);
for(int i = 0; i < 1000; i++)
{
    int idx[dims];
    for(int k = 0; k < dims; k++)
        idx[k] = rand()%sparse_mat.size(k);
    sparse_mat.ref<float>(idx) += 1.f;
}
```

2. sparse matrix iterators. Like Mat iterators and unlike MatND iterators, the sparse matrix iterators are STL-style, that is, the iteration loop is familiar to C++ users:

```
// prints elements of a sparse floating-point matrix
// and the sum of elements.
SparseMatConstIterator_<float>
    it = sparse_mat.begin<float>(),
    it_end = sparse_mat.end<float>();
double s = 0;
int dims = sparse_mat.dims();
for(; it != it_end; ++it)
{
    // print element indices and the element value
    const Node* n = it.node();
    printf("(")
    for(int i = 0; i < dims; i++)
        printf("%3d%c", n->idx[i], i < dims-1 ? ',' : ')');
    printf(": %f\n", *it);
    s += *it;
}
printf("Element sum is %g\n", s);
```

If you run this loop, you will notice that elements are enumerated in no any logical order (lexicographical etc.), they come in the same order as they stored in the hash table, i.e. semi-randomly. You may collect pointers to the nodes and sort them to get the proper ordering. Note, however, that pointers to the nodes may become invalid when you add more elements to the matrix; this is because of possible buffer reallocation.

3. a combination of the above 2 methods when you need to process 2 or more sparse matrices simultaneously, e.g. this is how you can compute unnormalized cross-correlation of the 2 floating-point sparse matrices:

```
double cross_corr(const SparseMat& a, const SparseMat& b)
{
    const SparseMat *_a = &a, *_b = &b;
    // if b contains less elements than a,
    // it's faster to iterate through b
    if(_a->nzcount() > _b->nzcount())
        std::swap(_a, _b);
    SparseMatConstIterator_<float> it = _a->begin<float>(),
                                   it_end = _a->end<float>();
    double ccorr = 0;
    for(; it != it_end; ++it)
    {
        // take the next element from the first matrix
        float avalue = *it;
        const Node* anode = it.node();
        // and try to find element with the same index in the second matrix.
        // since the hash value depends only on the element index,
        // we reuse hashvalue stored in the node
        float bvalue = _b->value<float>(anode->idx,&anode->hashval);
        ccorr += avalue*bvalue;
    }
    return ccorr;
}
```

SparseMat_

Template sparse n-dimensional array class derived from SparseMat

```
template<typename _Tp> class SparseMat_ : public SparseMat
{
public:
    typedef SparseMatIterator_<_Tp> iterator;
    typedef SparseMatConstIterator_<_Tp> const_iterator;
```

```
    // constructors;
    // the created matrix will have data type = DataType<_Tp>::type
    SparseMat_();
    SparseMat_(int dims, const int* _sizes);
    SparseMat_(const SparseMat& m);
    SparseMat_(const SparseMat_& m);
    SparseMat_(const Mat& m);
    SparseMat_(const CvSparseMat* m);
    // assignment operators; data type conversion is done when necessary
    SparseMat_& operator = (const SparseMat& m);
    SparseMat_& operator = (const SparseMat_& m);
    SparseMat_& operator = (const Mat& m);
    SparseMat_& operator = (const MatND& m);

    // equivalent to the correspoding parent class methods
    SparseMat_ clone() const;
    void create(int dims, const int* _sizes);
    operator CvSparseMat*() const;

    // overriden methods that do extra checks for the data type
    int type() const;
    int depth() const;
    int channels() const;

    // more convenient element access operations.
    // ref() is retained (but <_Tp> specification is not need anymore);
    // operator () is equivalent to SparseMat::value<_Tp>
    _Tp& ref(int i0, size_t* hashval=0);
    _Tp operator()(int i0, size_t* hashval=0) const;
    _Tp& ref(int i0, int i1, size_t* hashval=0);
    _Tp operator()(int i0, int i1, size_t* hashval=0) const;
    _Tp& ref(int i0, int i1, int i2, size_t* hashval=0);
    _Tp operator()(int i0, int i1, int i2, size_t* hashval=0) const;
    _Tp& ref(const int* idx, size_t* hashval=0);
    _Tp operator()(const int* idx, size_t* hashval=0) const;

    // iterators
    SparseMatIterator_<_Tp> begin();
    SparseMatConstIterator_<_Tp> begin() const;
    SparseMatIterator_<_Tp> end();
    SparseMatConstIterator_<_Tp> end() const;
};
```

`SparseMat_` is a thin wrapper on top of SparseMat , made in the same way as `Mat_`. It simplifies notation of some operations, and that's it.

```
int sz[] = {10, 20, 30};
SparseMat_<double> M(3, sz);
...
M.ref(1, 2, 3) = M(4, 5, 6) + M(7, 8, 9);
```

11.2 Operations on Arrays

---

cv::abs (view/add comments)

Computes absolute value of each matrix element

```
MatExpr<...> abs(const Mat& src);
MatExpr<...> abs(const MatExpr<...>& src);
```

src matrix or matrix expression

`abs` is a meta-function that is expanded to one of cv::absdiff forms:

- `C = abs(A-B)` is equivalent to `absdiff(A, B, C)` and
- `C = abs(A)` is equivalent to `absdiff(A, Scalar::all(0), C)`.
- `C = Mat_<Vec<uchar,n> >(abs(A*alpha + beta))` is equivalent to `convertScaleAbs(A, C, alpha, beta)`

The output matrix will have the same size and the same type as the input one (except for the last case, where `C` will be `depth=CV_8U`).

See also: Matrix Expressions , cv::absdiff, saturate_cast

---

cv::absdiff (view/add comments)

Computes per-element absolute difference between 2 arrays or between array and a scalar.

```
void absdiff(const Mat& src1, const Mat& src2, Mat& dst);
void absdiff(const Mat& src1, const Scalar& sc, Mat& dst);
void absdiff(const MatND& src1, const MatND& src2, MatND& dst);
void absdiff(const MatND& src1, const Scalar& sc, MatND& dst);
```

src1 The first input array

src2 The second input array; Must be the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

The functions `absdiff` compute:

- absolute difference between two arrays

$$\texttt{dst}(I) = \texttt{saturate}(|\texttt{src1}(I) - \texttt{src2}(I)|)$$

- or absolute difference between array and a scalar:

$$\texttt{dst}(I) = \texttt{saturate}(|\texttt{src1}(I) - \texttt{sc}|)$$

where `I` is multi-dimensional index of array elements. in the case of multi-channel arrays each channel is processed independently.

See also: cv::abs, saturate_cast

---

cv::add (view/add comments)

Computes the per-element sum of two arrays or an array and a scalar.

```
void add(const Mat& src1, const Mat& src2, Mat& dst);
void add(const Mat& src1, const Mat& src2,
         Mat& dst, const Mat& mask);
void add(const Mat& src1, const Scalar& sc,
         Mat& dst, const Mat& mask=Mat());
void add(const MatND& src1, const MatND& src2, MatND& dst);
void add(const MatND& src1, const MatND& src2,
         MatND& dst, const MatND& mask);
void add(const MatND& src1, const Scalar& sc,
         MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `add` compute:

- the sum of two arrays:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) + \texttt{src2}(I)) \quad \texttt{if mask}(I) \neq 0$$

- or the sum of array and a scalar:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) + \texttt{sc}) \quad \texttt{if mask}(I) \neq 0$$

where `I` is multi-dimensional index of array elements.

The first function in the above list can be replaced with matrix expressions:

```
dst = src1 + src2;
dst += src1; // equivalent to add(dst, src1, dst);
```

in the case of multi-channel arrays each channel is processed independently.

See also: cv::subtract, cv::addWeighted, cv::scaleAdd, cv::convertScale, Matrix Expressions , saturate_cast.

---

cv::addWeighted (view/add comments)

Computes the weighted sum of two arrays.

```
void addWeighted(const Mat& src1, double alpha, const Mat& src2,
           double beta, double gamma, Mat& dst);
void addWeighted(const MatND& src1, double alpha, const MatND& src2,
           double beta, double gamma, MatND& dst);
```

src1 The first source array

alpha Weight for the first array elements

src2 The second source array; must have the same size and same type as `src1`

beta Weight for the second array elements

dst The destination array; it will have the same size and same type as `src1`

gamma Scalar, added to each sum

The functions `addWeighted` calculate the weighted sum of two arrays as follows:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) * \texttt{alpha} + \texttt{src2}(I) * \texttt{beta} + \texttt{gamma})$$

where `I` is multi-dimensional index of array elements.

The first function can be replaced with a matrix expression:

```
dst = src1*alpha + src2*beta + gamma;
```

In the case of multi-channel arrays each channel is processed independently.

See also: cv::add, cv::subtract, cv::scaleAdd, cv::convertScale, Matrix Expressions , saturate_cast.

---

bitwise_and (view/add comments)

Calculates per-element bit-wise conjunction of two arrays and an array and a scalar.

```
void bitwise_and(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat());
void bitwise_and(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void bitwise_and(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask=MatND());
void bitwise_and(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_and` compute per-element bit-wise logical conjunction:

- of two arrays

$$\texttt{dst}(I) = \texttt{src1}(I) \wedge \texttt{src2}(I) \quad \texttt{if mask}(I) \neq 0$$

- or array and a scalar:

$$\texttt{dst}(I) = \texttt{src1}(I) \wedge \texttt{sc} \quad \texttt{if mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation, and in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_not, bitwise_xor

---

bitwise_not (view/add comments)

Inverts every bit of array

```
void bitwise_not(const Mat& src, Mat& dst);
void bitwise_not(const MatND& src, MatND& dst);
```

src1 The source array

dst The destination array; it is reallocated to be of the same size and the same type as `src`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_not` compute per-element bit-wise inversion of the source array:

$$\texttt{dst}(I) = \neg \texttt{src}(I)$$

In the case of floating-point source array its machine-specific bit representation (usually IEEE754-compliant) is used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_or, bitwise_xor

bitwise_or (view/add comments)

Calculates per-element bit-wise disjunction of two arrays and an array and a scalar.

```
void bitwise_or(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat());
void bitwise_or(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void bitwise_or(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask=MatND());
void bitwise_or(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it is reallocated to be of the same size and the same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_or` compute per-element bit-wise logical disjunction

- of two arrays

$$\texttt{dst}(I) = \texttt{src1}(I) \vee \texttt{src2}(I) \quad \texttt{if mask}(I) \neq 0$$

- or array and a scalar:

$$\texttt{dst}(I) = \texttt{src1}(I) \vee \texttt{sc} \quad \texttt{if mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_not, bitwise_or

bitwise_xor (view/add comments)

Calculates per-element bit-wise "exclusive or" operation on two arrays and an array and a scalar.

```
void bitwise_xor(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat());
void bitwise_xor(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void bitwise_xor(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask=MatND());
void bitwise_xor(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it is reallocated to be of the same size and the same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_xor` compute per-element bit-wise logical "exclusive or" operation

- on two arrays
$$\texttt{dst}(I) = \texttt{src1}(I) \oplus \texttt{src2}(I) \quad \texttt{if mask}(I) \neq 0$$
- or array and a scalar:
$$\texttt{dst}(I) = \texttt{src1}(I) \oplus \texttt{sc} \quad \texttt{if mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_not, bitwise_or

cv::calcCovarMatrix (view/add comments)

Calculates covariation matrix of a set of vectors

```
void calcCovarMatrix( const Mat* samples, int nsamples,
           Mat& covar, Mat& mean,
           int flags, int ctype=CV_64F);
void calcCovarMatrix( const Mat& samples, Mat& covar, Mat& mean,
           int flags, int ctype=CV_64F);
```

samples The samples, stored as separate matrices, or as rows or columns of a single matrix

nsamples The number of samples when they are stored separately

covar The output covariance matrix; it will have type=`ctype` and square size

mean The input or output (depending on the flags) array - the mean (average) vector of the input vectors

flags The operation flags, a combination of the following values

CV_COVAR_SCRAMBLED The output covariance matrix is calculated as:

$$\texttt{scale} \cdot [\texttt{vects}[0]-\texttt{mean}, \texttt{vects}[1]-\texttt{mean}, ...]^T \cdot [\texttt{vects}[0]-\texttt{mean}, \texttt{vects}[1]-\texttt{mean}, ...]$$

, that is, the covariance matrix will be `nsamples` × `nsamples`. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces technique for face recognition). Eigenvalues of this ”scrambled” matrix will match the eigenvalues of the true covariance matrix and the ”true” eigenvectors can be easily calculated from the eigenvectors of the ”scrambled” covariance matrix.

CV_COVAR_NORMAL The output covariance matrix is calculated as:

$$\texttt{scale} \cdot [\texttt{vects}[0]-\texttt{mean}, \texttt{vects}[1]-\texttt{mean}, ...] \cdot [\texttt{vects}[0]-\texttt{mean}, \texttt{vects}[1]-\texttt{mean}, ...]^T$$

, that is, `covar` will be a square matrix of the same size as the total number of elements in each input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

CV_COVAR_USE_AVG If the flag is specified, the function does not calculate `mean` from the input vectors, but, instead, uses the passed `mean` vector. This is useful if `mean` has been pre-computed or known a-priori, or if the covariance matrix is calculated by parts - in this case, `mean` is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.

CV_COVAR_SCALE If the flag is specified, the covariance matrix is scaled. In the "normal" mode `scale` is `1./nsamples`; in the "scrambled" mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled (i.e. `scale=1`).

CV_COVAR_ROWS [Only useful in the second variant of the function] The flag means that all the input vectors are stored as rows of the `samples` matrix. `mean` should be a single-row vector in this case.

CV_COVAR_COLS [Only useful in the second variant of the function] The flag means that all the input vectors are stored as columns of the `samples` matrix. `mean` should be a single-column vector in this case.

The functions `calcCovarMatrix` calculate the covariance matrix and, optionally, the mean vector of the set of input vectors.

See also: cv::PCA, cv::mulTransposed, cv::Mahalanobis

---

cv::cartToPolar (view/add comments)

Calculates the magnitude and angle of 2d vectors.

```
void cartToPolar(const Mat& x, const Mat& y,
           Mat& magnitude, Mat& angle,
           bool angleInDegrees=false);
```

x The array of x-coordinates; must be single-precision or double-precision floating-point array

y The array of y-coordinates; it must have the same size and same type as `x`

magnitude The destination array of magnitudes of the same size and same type as `x`

angle The destination array of angles of the same size and same type as `x`. The angles are measured in radians (0 to 2π) or in degrees (0 to 360 degrees).

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function `cartToPolar` calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

$$\texttt{magnitude}(I) = \sqrt{\texttt{x}(I)^2 + \texttt{y}(I)^2},$$
$$\texttt{angle}(I) = \texttt{atan2}(\texttt{y}(I), \texttt{x}(I))[\cdot 180/\pi]$$

The angles are calculated with $\sim 0.3^\circ$ accuracy. For the (0,0) point, the angle is set to 0.

---

cv::checkRange (view/add comments)

Checks every element of an input array for invalid values.

```
bool checkRange(const Mat& src, bool quiet=true, Point* pos=0,
           double minVal=-DBL_MAX, double maxVal=DBL_MAX);
bool checkRange(const MatND& src, bool quiet=true, int* pos=0,
           double minVal=-DBL_MAX, double maxVal=DBL_MAX);
```

src The array to check

quiet The flag indicating whether the functions quietly return false when the array elements are out of range, or they throw an exception.

pos The optional output parameter, where the position of the first outlier is stored. In the second function `pos`, when not NULL, must be a pointer to array of `src.dims` elements

minVal The inclusive lower boundary of valid values range

maxVal The exclusive upper boundary of valid values range

The functions `checkRange` check that every array element is neither NaN nor $\pm\infty$. When `minVal < -DBL_MAX` and `maxVal < DBL_MAX`, then the functions also check that each value is between `minVal` and `maxVal`. in the case of multi-channel arrays each channel is processed independently. If some values are out of range, position of the first outlier is stored in `pos` (when `pos` $\neq 0$), and then the functions either return false (when `quiet=true`) or throw an exception.

cv::compare (view/add comments)

Performs per-element comparison of two arrays or an array and scalar value.

```
void compare(const Mat& src1, const Mat& src2, Mat& dst, int cmpop);
void compare(const Mat& src1, double value,
             Mat& dst, int cmpop);
void compare(const MatND& src1, const MatND& src2,
             MatND& dst, int cmpop);
void compare(const MatND& src1, double value,
             MatND& dst, int cmpop);
```

src1 The first source array

src2 The second source array; must have the same size and same type as `src1`

value The scalar value to compare each array element with

dst The destination array; will have the same size as `src1` and type=`CV_8UC1`

cmpop The flag specifying the relation between the elements to be checked

- **CMP_EQ** $\texttt{src1}(I) = \texttt{src2}(I)$ or $\texttt{src1}(I) = \texttt{value}$
- **CMP_GT** $\texttt{src1}(I) > \texttt{src2}(I)$ or $\texttt{src1}(I) > \texttt{value}$
- **CMP_GE** $\texttt{src1}(I) \geq \texttt{src2}(I)$ or $\texttt{src1}(I) \geq \texttt{value}$
- **CMP_LT** $\texttt{src1}(I) < \texttt{src2}(I)$ or $\texttt{src1}(I) < \texttt{value}$
- **CMP_LE** $\texttt{src1}(I) \leq \texttt{src2}(I)$ or $\texttt{src1}(I) \leq \texttt{value}$
- **CMP_NE** $\texttt{src1}(I) \neq \texttt{src2}(I)$ or $\texttt{src1}(I) \neq \texttt{value}$

The functions `compare` compare each element of `src1` with the corresponding element of `src2` or with real scalar `value`. When the comparison result is true, the corresponding element of destination array is set to 255, otherwise it is set to 0:

- `dst(I) = src1(I) cmpop src2(I) ? 255 : 0`
- `dst(I) = src1(I) cmpop value ? 255 : 0`

The comparison operations can be replaced with the equivalent matrix expressions:

```
Mat dst1 = src1 >= src2;
Mat dst2 = src1 < 8;
...
```

See also: cv::checkRange, cv::min, cv::max, cv::threshold, Matrix Expressions

---

cv::completeSymm (view/add comments)

Copies the lower or the upper half of a square matrix to another half.

```
void completeSymm(Mat& mtx, bool lowerToUpper=false);
```

mtx Input-output floating-point square matrix

lowerToUpper If true, the lower half is copied to the upper half, otherwise the upper half is copied to the lower half

The function `completeSymm` copies the lower half of a square matrix to its another half; the matrix diagonal remains unchanged:

- $\texttt{mtx}_{ij} = \texttt{mtx}_{ji}$ for $i > j$ if `lowerToUpper=false`
- $\texttt{mtx}_{ij} = \texttt{mtx}_{ji}$ for $i < j$ if `lowerToUpper=true`

See also: cv::flip, cv::transpose

---

cv::convertScaleAbs (view/add comments)

Scales, computes absolute values and converts the result to 8-bit.

```
void convertScaleAbs(const Mat& src, Mat& dst, double alpha=1, double
beta=0);
```

src The source array

dst The destination array

alpha The optional scale factor

beta The optional delta added to the scaled values

On each element of the input array the function `convertScaleAbs` performs 3 operations sequentially: scaling, taking absolute value, conversion to unsigned 8-bit type:

$$\texttt{dst}(I) = \texttt{saturate_cast<uchar>}(|\texttt{src}(I) * \texttt{alpha} + \texttt{beta}|)$$

in the case of multi-channel arrays the function processes each channel independently. When the output is not 8-bit, the operation can be emulated by calling `Mat::convertTo` method (or by using matrix expressions) and then by computing absolute value of the result, for example:

```
Mat_<float> A(30,30);
randu(A, Scalar(-100), Scalar(100));
Mat_<float> B = A*5 + 3;
B = abs(B);
// Mat_<float> B = abs(A*5+3) will also do the job,
// but it will allocate a temporary matrix
```

See also: cv::Mat::convertTo, cv::abs

---

cv::countNonZero (view/add comments)

Counts non-zero array elements.

```
int countNonZero( const Mat& mtx );
int countNonZero( const MatND& mtx );
```

mtx Single-channel array

The function `cvCountNonZero` returns the number of non-zero elements in mtx:

$$\sum_{I:\, \texttt{mtx}(I) \neq 0} 1$$

See also: cv::mean, cv::meanStdDev, cv::norm, cv::minMaxLoc, cv::calcCovarMatrix

cv::cubeRoot (view/add comments)

Computes cube root of the argument

```
float cubeRoot(float val);
```

val The function argument

The function `cubeRoot` computes $\sqrt[3]{\texttt{val}}$. Negative arguments are handled correctly, *NaN* and $\pm\infty$ are not handled. The accuracy approaches the maximum possible accuracy for single-precision data.

cv::cvarrToMat (view/add comments)

Converts CvMat, IplImage or CvMatND to cv::Mat.

```
Mat cvarrToMat(const CvArr* src, bool copyData=false, bool
allowND=true, int coiMode=0);
```

src The source `CvMat`, `IplImage` or `CvMatND`

copyData When it is false (default value), no data is copied, only the new header is created. In this case the original array should not be deallocated while the new matrix header is used. The the parameter is true, all the data is copied, then user may deallocate the original array right after the conversion

allowND When it is true (default value), then `CvMatND` is converted to `Mat` if it's possible (e.g. then the data is contiguous). If it's not possible, or when the parameter is false, the function will report an error

coiMode The parameter specifies how the IplImage COI (when set) is handled.

- If `coiMode=0`, the function will report an error if COI is set.
- If `coiMode=1`, the function will never report an error; instead it returns the header to the whole original image and user will have to check and process COI manually, see cv::extractImageCOI.

The function `cvarrToMat` converts CvMat , IplImage or CvMatND header to cv::Mat header, and optionally duplicates the underlying data. The constructed header is returned by the function.

When `copyData=false`, the conversion is done really fast (in O(1) time) and the newly created matrix header will have `refcount=0`, which means that no reference counting is done for the matrix data, and user has to preserve the data until the new header is destructed. Otherwise, when `copyData=true`, the new buffer will be allocated and managed as if you created a new matrix from scratch and copy the data there. That is, `cvarrToMat(src, true)` ∼ `cvarrToMat(src, false).clone()` (assuming that COI is not set). The function provides uniform way of supporting CvArr paradigm in the code that is migrated to use new-style data structures internally. The reverse transformation, from cv::Mat to CvMat or IplImage can be done by simple assignment:

```
CvMat* A = cvCreateMat(10, 10, CV_32F);
cvSetIdentity(A);
IplImage A1; cvGetImage(A, &A1);
Mat B = cvarrToMat(A);
Mat B1 = cvarrToMat(&A1);
IplImage C = B;
CvMat C1 = B1;
// now A, A1, B, B1, C and C1 are different headers
// for the same 10x10 floating-point array.
// note, that you will need to use "&"
// to pass C & C1 to OpenCV functions, e.g:
printf("%g", cvDet(&C1));
```

Normally, the function is used to convert an old-style 2D array (CvMat or IplImage) to `Mat`, however, the function can also take CvMatND on input and create cv::Mat for it, if it's possible. And for `CvMatND A` it is possible if and only if `A.dim[i].size*A.dim.step[i] == A.dim.step[i-1]` for all or for all but one `i, 0 < i < A.dims`. That is, the matrix data should be continuous or it should be representable as a sequence of continuous matrices. By using this function in this way, you can process CvMatND using arbitrary element-wise function. But for more complex operations, such as filtering functions, it will not work, and you need to convert CvMatND to cv::MatND using the corresponding constructor of the latter.

The last parameter, `coiMode`, specifies how to react on an image with COI set: by default it's 0, and then the function reports an error when an image with COI comes in. And `coiMode=1` means that no error is signaled - user has to check COI presence and handle it manually. The modern structures, such as cv::Mat and cv::MatND do not support COI natively. To process individual channel of an new-style array, you will need either to organize loop over the array (e.g. using matrix iterators) where the channel of interest will be processed, or extract the COI using cv::mixChannels (for new-style arrays) or cv::extractImageCOI (for old-style arrays), process this individual channel and insert it back to the destination array if need (using cv::mixChannel or

cv::insertImageCOI, respectively).

See also: cv::cvGetImage, cv::cvGetMat, cv::cvGetMatND, cv::extractImageCOI, cv::insertImageCOI, cv::mixChannels

---

cv::dct (view/add comments)

Performs a forward or inverse discrete cosine transform of 1D or 2D array

```
void dct(const Mat& src, Mat& dst, int flags=0);
```

src The source floating-point array

dst The destination array; will have the same size and same type as `src`

flags Transformation flags, a combination of the following values

DCT_INVERSE do an inverse 1D or 2D transform instead of the default forward transform.

DCT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

The function `dct` performs a forward or inverse discrete cosine transform (DCT) of a 1D or 2D floating-point array:

Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C_{jk}^{(N)} = \sqrt{\alpha_j/N} \cos\left(\frac{\pi(2k+1)j}{2N}\right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.

Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)}\right)^{-1} \cdot Y = \left(C^{(N)}\right)^{T} \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)}\right)^{T} = I$)

Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)}\right)^T$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)}\right)^T \cdot X \cdot C^{(N)}$$

The function chooses the mode of operation by looking at the flags and size of the input array:

- if `(flags & DCT_INVERSE) == 0`, the function does forward 1D or 2D transform, otherwise it is inverse 1D or 2D transform.
- if `(flags & DCT_ROWS)` $\neq$ `0`, the function performs 1D transform of each row.
- otherwise, if the array is a single column or a single row, the function performs 1D transform
- otherwise it performs 2D transform.

Important note: currently cv::dct supports even-size arrays (2, 4, 6 ...). For data analysis and approximation you can pad the array when necessary.

Also, the function's performance depends very much, and not monotonically, on the array size, see cv::getOptimalDFTSize. In the current implementation DCT of a vector of size `N` is computed via DFT of a vector of size `N/2`, thus the optimal DCT size `N*` $\geq$ `N` can be computed as:

```
size_t getOptimalDCTSize(size_t N) { return 2*getOptimalDFTSize((N+1)/2); }
```

See also: cv::dft, cv::getOptimalDFTSize, cv::idct

---

cv::dft (view/add comments)

Performs a forward or inverse Discrete Fourier transform of 1D or 2D floating-point array.

```
void dft(const Mat& src, Mat& dst, int flags=0, int nonzeroRows=0);
```

src The source array, real or complex

dst The destination array, which size and type depends on the `flags`

flags Transformation flags, a combination of the following values

DFT_INVERSE do an inverse 1D or 2D transform instead of the default forward transform.

DFT_SCALE scale the result: divide it by the number of array elements. Normally, it is combined with DFT_INVERSE .

DFT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows the user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

DFT_COMPLEX_OUTPUT then the function performs forward transformation of 1D or 2D real array, the result, though being a complex array, has complex-conjugate symmetry (*CCS*), see the description below. Such an array can be packed into real array of the same size as input, which is the fastest option and which is what the function does by default. However, you may wish to get the full complex array (for simpler spectrum analysis etc.). Pass the flag to tell the function to produce full-size complex output array.

DFT_REAL_OUTPUT then the function performs inverse transformation of 1D or 2D complex array, the result is normally a complex array of the same size. However, if the source array has conjugate-complex symmetry (for example, it is a result of forward transformation with DFT_COMPLEX_OUTPUT flag), then the output is real array. While the function itself does not check whether the input is symmetrical or not, you can pass the flag and then the function will assume the symmetry and produce the real output array. Note that when the input is packed real array and inverse transformation is executed, the function treats the input as packed complex-conjugate symmetrical array, so the output will also be real array

nonzeroRows When the parameter $\neq 0$, the function assumes that only the first nonzeroRows rows of the input array (DFT_INVERSE is not set) or only the first nonzeroRows of the output array (DFT_INVERSE is set) contain non-zeros, thus the function can handle the rest of the rows more efficiently and thus save some time. This technique is very useful for computing array cross-correlation or convolution using DFT

Forward Fourier transform of 1D vector of N elements:

$$Y = F^{(N)} \cdot X,$$

where $F^{(N)}_{jk} = \exp(-2\pi ijk/N)$ and $i = \sqrt{-1}$

Inverse Fourier transform of 1D vector of N elements:

$$X' = \left(F^{(N)}\right)^{-1} \cdot Y = \left(F^{(N)}\right)^{*} \cdot y$$
$$X = (1/N) \cdot X,$$

where $F^{*} = \left(\mathrm{Re}(F^{(N)}) - \mathrm{Im}(F^{(N)})\right)^{T}$

Forward Fourier transform of 2D vector of $M \times N$ elements:

$$Y = F^{(M)} \cdot X \cdot F^{(N)}$$

Inverse Fourier transform of 2D vector of $M \times N$ elements:

$$X' = \left(F^{(M)}\right)^* \cdot Y \cdot \left(F^{(N)}\right)^*$$
$$X = \frac{1}{M \cdot N} \cdot X'$$

In the case of real (single-channel) data, the packed format called *CCS* (complex-conjugate-symmetrical) that was borrowed from IPL and used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix} ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\ ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\ ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\ \hdots & & & & & & & & \\ ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \hdots & & & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\ ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \hdots & & & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\ ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \hdots & & & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2} \end{bmatrix}$$

in the case of 1D transform of real vector, the output will look as the first row of the above matrix.

So, the function chooses the operation mode depending on the flags and size of the input array:

- if `DFT_ROWS` is set or the input array has single row or single column then the function performs 1D forward or inverse transform (of each row of a matrix when `DFT_ROWS` is set, otherwise it will be 2D transform.
- if input array is real and `DFT_INVERSE` is not set, the function does forward 1D or 2D transform:
  - when `DFT_COMPLEX_OUTPUT` is set then the output will be complex matrix of the same size as input.
  - otherwise the output will be a real matrix of the same size as input. in the case of 2D transform it will use the packed format as shown above; in the case of single 1D transform it will look as the first row of the above matrix; in the case of multiple 1D transforms (when using `DCT_ROWS` flag) each row of the output matrix will look like the first row of the above matrix.

- otherwise, if the input array is complex and either DFT_INVERSE or DFT_REAL_OUTPUT are not set then the output will be a complex array of the same size as input and the function will perform the forward or inverse 1D or 2D transform of the whole input array or each row of the input array independently, depending on the flags DFT_INVERSE and DFT_ROWS.

- otherwise, i.e. when DFT_INVERSE is set, the input array is real, or it is complex but DFT_REAL_OUTPUT is set, the output will be a real array of the same size as input, and the function will perform 1D or 2D inverse transformation of the whole input array or each individual row, depending on the flags DFT_INVERSE and DFT_ROWS.

The scaling is done after the transformation if DFT_SCALE is set.

Unlike cv::dct, the function supports arrays of arbitrary size, but only those arrays are processed efficiently, which sizes can be factorized in a product of small prime numbers (2, 3 and 5 in the current implementation). Such an efficient DFT size can be computed using cv::getOptimalDFTSize method.

Here is the sample on how to compute DFT-based convolution of two 2D real arrays:

```
void convolveDFT(const Mat& A, const Mat& B, Mat& C)
{
    // reallocate the output array if needed
    C.create(abs(A.rows - B.rows)+1, abs(A.cols - B.cols)+1, A.type());
    Size dftSize;
    // compute the size of DFT transform
    dftSize.width = getOptimalDFTSize(A.cols + B.cols - 1);
    dftSize.height = getOptimalDFTSize(A.rows + B.rows - 1);

    // allocate temporary buffers and initialize them with 0's
    Mat tempA(dftSize, A.type(), Scalar::all(0));
    Mat tempB(dftSize, B.type(), Scalar::all(0));

    // copy A and B to the top-left corners of tempA and tempB, respectively
    Mat roiA(tempA, Rect(0,0,A.cols,A.rows));
    A.copyTo(roiA);
    Mat roiB(tempB, Rect(0,0,B.cols,B.rows));
    B.copyTo(roiB);

    // now transform the padded A & B in-place;
    // use "nonzeroRows" hint for faster processing
    dft(tempA, tempA, 0, A.rows);
    dft(tempB, tempB, 0, B.rows);

    // multiply the spectrums;
    // the function handles packed spectrum representations well
    mulSpectrums(tempA, tempB, tempA);
```

```
    // transform the product back from the frequency domain.
    // Even though all the result rows will be non-zero,
    // we need only the first C.rows of them, and thus we
    // pass nonzeroRows == C.rows
    dft(tempA, tempA, DFT_INVERSE + DFT_SCALE, C.rows);

    // now copy the result back to C.
    tempA(Rect(0, 0, C.cols, C.rows)).copyTo(C);

    // all the temporary buffers will be deallocated automatically
}
```

What can be optimized in the above sample?

- since we passed `nonzeroRows` $\neq 0$ to the forward transform calls and since we copied `A`/`B` to the top-left corners of `tempA`/`tempB`, respectively, it's not necessary to clear the whole `tempA` and `tempB`; it is only necessary to clear the `tempA.cols - A.cols` (`tempB.cols - B.cols`) rightmost columns of the matrices.
- this DFT-based convolution does not have to be applied to the whole big arrays, especially if `B` is significantly smaller than `A` or vice versa. Instead, we can compute convolution by parts. For that we need to split the destination array `C` into multiple tiles and for each tile estimate, which parts of `A` and `B` are required to compute convolution in this tile. If the tiles in `C` are too small, the speed will decrease a lot, because of repeated work - in the ultimate case, when each tile in `C` is a single pixel, the algorithm becomes equivalent to the naive convolution algorithm. If the tiles are too big, the temporary arrays `tempA` and `tempB` become too big and there is also slowdown because of bad cache locality. So there is optimal tile size somewhere in the middle.
- if the convolution is done by parts, since different tiles in `C` can be computed in parallel, the loop can be threaded.

All of the above improvements have been implemented in cv::matchTemplate and cv::filter2D, therefore, by using them, you can get even better performance than with the above theoretically optimal implementation (though, those two functions actually compute cross-correlation, not convolution, so you will need to "flip" the kernel or the image around the center using cv::flip).

See also: cv::dct, cv::getOptimalDFTSize, cv::mulSpectrums, cv::filter2D, cv::matchTemplate, cv::flip, cv::cartToPolar, cv::magnitude, cv::phase

---

cv::divide (view/add comments)

Performs per-element division of two arrays or a scalar by an array.

```
void divide(const Mat& src1, const Mat& src2,
            Mat& dst, double scale=1);
void divide(double scale, const Mat& src2, Mat& dst);
void divide(const MatND& src1, const MatND& src2,
            MatND& dst, double scale=1);
void divide(double scale, const MatND& src2, MatND& dst);
```

src1 The first source array

src2 The second source array; should have the same size and same type as `src1`

scale Scale factor

dst The destination array; will have the same size and same type as `src2`

The functions `divide` divide one array by another:

```
dst(I) = saturate(src1(I)*scale/src2(I))
```

or a scalar by array, when there is no `src1`:

```
dst(I) = saturate(scale/src2(I))
```

The result will have the same type as `src1`. When `src2(I)=0`, `dst(I)=0` too.
See also: cv::multiply, cv::add, cv::subtract, Matrix Expressions

---

cv::determinant (view/add comments)

Returns determinant of a square floating-point matrix.

```
double determinant(const Mat& mtx);
```

mtx The input matrix; must have `CV_32FC1` or `CV_64FC1` type and square size

The function `determinant` computes and returns determinant of the specified matrix. For small matrices (`mtx.cols=mtx.rows<=3`) the direct method is used; for larger matrices the function uses LU factorization.

For symmetric positive-determined matrices, it is also possible to compute cv::SVD: `mtx` $= U \cdot W \cdot V^T$ and then calculate the determinant as a product of the diagonal elements of W.

See also: cv::SVD, cv::trace, cv::invert, cv::solve, Matrix Expressions

cv::eigen (view/add comments)

Computes eigenvalues and eigenvectors of a symmetric matrix.

```
bool eigen(const Mat& src, Mat& eigenvalues,
           int lowindex=-1, int highindex=-1);
bool eigen(const Mat& src, Mat& eigenvalues,
           Mat& eigenvectors, int lowindex=-1,
           int highindex=-1);
```

src The input matrix; must have `CV_32FC1` or `CV_64FC1` type, square size and be symmetric: $\texttt{src}^T = \texttt{src}$

eigenvalues The output vector of eigenvalues of the same type as `src`; The eigenvalues are stored in the descending order.

eigenvectors The output matrix of eigenvectors; It will have the same size and the same type as `src`; The eigenvectors are stored as subsequent matrix rows, in the same order as the corresponding eigenvalues

lowindex Optional index of largest eigenvalue/-vector to calculate. (See below.)

highindex Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The functions `eigen` compute just eigenvalues, or eigenvalues and eigenvectors of symmetric matrix `src`:

```
src*eigenvectors(i,:)' = eigenvalues(i)*eigenvectors(i,:)' (in MATLAB notation)
```

If either low- or highindex is supplied the other is required, too. Indexing is 0-based. Example: To calculate the largest eigenvector/-value set lowindex = highindex = 0. For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

See also: cv::SVD, cv::completeSymm, cv::PCA

cv::exp (view/add comments)

Calculates the exponent of every array element.

```
void exp(const Mat& src, Mat& dst);
void exp(const MatND& src, MatND& dst);
```

src The source array

dst The destination array; will have the same size and same type as `src`

The function `exp` calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}}(I)$$

The maximum relative error is about 7×10^{-6} for single-precision and less than 10^{-10} for double-precision. Currently, the function converts denormalized values to zeros on output. Special values (NaN, $\pm\infty$) are not handled.

See also: cv::log, cv::cartToPolar, cv::polarToCart, cv::phase, cv::pow, cv::sqrt, cv::magnitude

---

cv::extractImageCOI (view/add comments)

Extract the selected image channel

```
void extractImageCOI(const CvArr* src, Mat& dst, int coi=-1);
```

src The source array. It should be a pointer to CvMat or IplImage

dst The destination array; will have single-channel, and the same size and the same depth as `src`

coi If the parameter is `>=0`, it specifies the channel to extract; If it is `<0`, `src` must be a pointer to `IplImage` with valid COI set - then the selected COI is extracted.

The function `extractImageCOI` is used to extract image COI from an old-style array and put the result to the new-style C++ matrix. As usual, the destination matrix is reallocated using `Mat::create` if needed.

To extract a channel from a new-style matrix, use cv::mixChannels or cv::split

See also: cv::mixChannels, cv::split, cv::merge, cv::cvarrToMat, cv::cvSetImageCOI, cv::cvGetImageCOI

cv::fastAtan2 (view/add comments)

Calculates the angle of a 2D vector in degrees

```
float fastAtan2(float y, float x);
```

x x-coordinate of the vector

y y-coordinate of the vector

The function `fastAtan2` calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0° to 360°. The accuracy is about 0.3°.

cv::flip (view/add comments)

Flips a 2D array around vertical, horizontal or both axes.

```
void flip(const Mat& src, Mat& dst, int flipCode);
```

src The source array

dst The destination array; will have the same size and same type as `src`

flipCode Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas.

The function `flip` flips the array in one of three different ways (row and column indices are 0-based):

$$\texttt{dst}_{ij} = \begin{cases} \texttt{src}_{\texttt{src.rows}-i-1,j} & \text{if } \texttt{flipCode} = 0 \\ \texttt{src}_{i,\texttt{src.cols}-j-1} & \text{if } \texttt{flipCode} > 0 \\ \texttt{src}_{\texttt{src.rows}-i-1,\texttt{src.cols}-j-1} & \text{if } \texttt{flipCode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (`flipCode` $= 0$) to switch between top-left and bottom-left image origin, which is a typical operation in video processing in Windows.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (`flipCode` > 0)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (`flipCode` < 0)
- reversing the order of 1d point arrays (`flipCode` > 0 or `flipCode` $= 0$)

See also: cv::transpose, cv::repeat, cv::completeSymm

---

cv::gemm (view/add comments)

Performs generalized matrix multiplication.

```
void gemm(const Mat& src1, const Mat& src2, double alpha,
          const Mat& src3, double beta, Mat& dst, int flags=0);
```

src1 The first multiplied input matrix; should have `CV_32FC1`, `CV_64FC1`, `CV_32FC2` or `CV_64FC2` type

src2 The second multiplied input matrix; should have the same type as `src1`

alpha The weight of the matrix product

src3 The third optional delta matrix added to the matrix product; should have the same type as `src1` and `src2`

beta The weight of `src3`

dst The destination matrix; It will have the proper size and the same type as input matrices

flags Operation flags:

GEMM_1_T transpose `src1`

GEMM_2_T transpose `src2`

GEMM_3_T transpose `src3`

The function performs generalized matrix multiplication and similar to the corresponding functions `*gemm` in BLAS level 3. For example, `gemm(src1, src2, alpha, src3, beta, dst, GEMM_1_T + GEMM_3_T)` corresponds to

$$\texttt{dst} = \texttt{alpha} \cdot \texttt{src1}^T \cdot \texttt{src2} + \texttt{beta} \cdot \texttt{src3}^T$$

The function can be replaced with a matrix expression, e.g. the above call can be replaced with:

```
dst = alpha*src1.t()*src2 + beta*src3.t();
```

See also: cv::mulTransposed, cv::transform, Matrix Expressions

---

cv::getConvertElem (view/add comments)

Returns conversion function for a single pixel

```
ConvertData getConvertElem(int fromType, int toType);
ConvertScaleData getConvertScaleElem(int fromType, int toType);
typedef void (*ConvertData)(const void* from, void* to, int cn);
typedef void (*ConvertScaleData)(const void* from, void* to,
           int cn, double alpha, double beta);
```

fromType The source pixel type

toType The destination pixel type

from Callback parameter: pointer to the input pixel

to Callback parameter: pointer to the output pixel

cn Callback parameter: the number of channels; can be arbitrary, 1, 100, 100000, ...

alpha ConvertScaleData callback optional parameter: the scale factor

beta ConvertScaleData callback optional parameter: the delta or offset

The functions `getConvertElem` and `getConvertScaleElem` return pointers to the functions for converting individual pixels from one type to another. While the main function purpose is to convert single pixels (actually, for converting sparse matrices from one type to another), you can use them to convert the whole row of a dense matrix or the whole matrix at once, by setting `cn = matrix.cols*matrix.rows*matrix.channels()` if the matrix data is continuous.

See also: cv::Mat::convertTo, cv::MatND::convertTo, cv::SparseMat::convertTo

cv::getOptimalDFTSize (view/add comments)

Returns optimal DFT size for a given vector size.

```
int getOptimalDFTSize(int vecsize);
```

vecsize Vector size

DFT performance is not a monotonic function of a vector size, therefore, when you compute convolution of two arrays or do a spectral analysis of array, it usually makes sense to pad the input data with zeros to get a bit larger array that can be transformed much faster than the original one. Arrays, which size is a power-of-two (2, 4, 8, 16, 32, ...) are the fastest to process, though, the arrays, which size is a product of 2's, 3's and 5's (e.g. 300 = 5*5*3*2*2), are also processed quite efficiently.

The function `getOptimalDFTSize` returns the minimum number `N` that is greater than or equal to `vecsize`, such that the DFT of a vector of size `N` can be computed efficiently. In the current implementation $N = 2^p \times 3^q \times 5^r$, for some p, q, r.

The function returns a negative number if `vecsize` is too large (very close to `INT_MAX`).

While the function cannot be used directly to estimate the optimal vector size for DCT transform (since the current DCT implementation supports only even-size vectors), it can be easily computed as `getOptimalDFTSize((vecsize+1)/2)*2`.

See also: cv::dft, cv::dct, cv::idft, cv::idct, cv::mulSpectrums

cv::idct (view/add comments)

Computes inverse Discrete Cosine Transform of a 1D or 2D array

```
void idct(const Mat& src, Mat& dst, int flags=0);
```

src The source floating-point single-channel array

dst The destination array. Will have the same size and same type as `src`

flags The operation flags.

`idct(src, dst, flags)` is equivalent to `dct(src, dst, flags | DCT_INVERSE)`. See cv::dct for details.

See also: cv::dct, cv::dft, cv::idft, cv::getOptimalDFTSize

---

cv::idft (view/add comments)

Computes inverse Discrete Fourier Transform of a 1D or 2D array

```
void idft(const Mat& src, Mat& dst, int flags=0, int outputRows=0);
```

src The source floating-point real or complex array

dst The destination array, which size and type depends on the `flags`

flags The operation flags. See cv::dft

nonzeroRows The number of `dst` rows to compute. The rest of the rows will have undefined content. See the convolution sample in cv::dft description

`idft(src, dst, flags)` is equivalent to `dct(src, dst, flags | DFT_INVERSE)`. See cv::dft for details. Note, that none of `dft` and `idft` scale the result by default. Thus, you should pass `DFT_SCALE` to one of `dft` or `idft` explicitly to make these transforms mutually inverse.

See also: cv::dft, cv::dct, cv::idct, cv::mulSpectrums, cv::getOptimalDFTSize

---

cv::inRange (view/add comments)

Checks if array elements lie between the elements of two other arrays.

```
void inRange(const Mat& src, const Mat& lowerb,
             const Mat& upperb, Mat& dst);
void inRange(const Mat& src, const Scalar& lowerb,
             const Scalar& upperb, Mat& dst);
void inRange(const MatND& src, const MatND& lowerb,
             const MatND& upperb, MatND& dst);
void inRange(const MatND& src, const Scalar& lowerb,
             const Scalar& upperb, MatND& dst);
```

src The first source array

lowerb The inclusive lower boundary array of the same size and type as src

upperb The exclusive upper boundary array of the same size and type as src

dst The destination array, will have the same size as src and CV_8U type

The functions inRange do the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lowerb}(I)_0 \leq \texttt{src}(I)_0 < \texttt{upperb}(I)_0$$

for single-channel arrays,

$$\texttt{dst}(I) = \texttt{lowerb}(I)_0 \leq \texttt{src}(I)_0 < \texttt{upperb}(I)_0 \wedge \texttt{lowerb}(I)_1 \leq \texttt{src}(I)_1 < \texttt{upperb}(I)_1$$

for two-channel arrays and so forth. dst(I) is set to 255 (all 1-bits) if src(I) is within the specified range and 0 otherwise.

---

cv::invert (view/add comments)

Finds the inverse or pseudo-inverse of a matrix

```
double invert(const Mat& src, Mat& dst, int method=DECOMP_LU);
```

src The source floating-point $M \times N$ matrix

dst The destination matrix; will have $N \times M$ size and the same type as src

flags The inversion method :

DECOMP_LU Gaussian elimination with optimal pivot element chosen

DECOMP_SVD Singular value decomposition (SVD) method

DECOMP_CHOLESKY Cholesky decomposion. The matrix must be symmetrical and positively defined

The function `invert` inverts matrix `src` and stores the result in `dst`. When the matrix `src` is singular or non-square, the function computes the pseudo-inverse matrix, i.e. the matrix `dst`, such that $\|\texttt{src} \cdot \texttt{dst} - I\|$ is minimal.

In the case of `DECOMP_LU` method, the function returns the `src` determinant (`src` must be square). If it is 0, the matrix is not inverted and `dst` is filled with zeros.

In the case of `DECOMP_SVD` method, the function returns the inversed condition number of `src` (the ratio of the smallest singular value to the largest singular value) and 0 if `src` is singular. The SVD method calculates a pseudo-inverse matrix if `src` is singular.

Similarly to `DECOMP_LU`, the method `DECOMP_CHOLESKY` works only with non-singular square matrices. In this case the function stores the inverted matrix in `dst` and returns non-zero, otherwise it returns 0.

See also: cv::solve, cv::SVD

---

cv::log (view/add comments)

Calculates the natural logarithm of every array element.

```
void log(const Mat& src, Mat& dst);
void log(const MatND& src, MatND& dst);
```

src The source array

dst The destination array; will have the same size and same type as `src`

The function `log` calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}(I) = \begin{cases} \log|\texttt{src}(I)| & \text{if } \texttt{src}(I) \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where `C` is a large negative number (about -700 in the current implementation). The maximum relative error is about 7×10^{-6} for single-precision input and less than 10^{-10} for double-precision input. Special values (NaN, $\pm\infty$) are not handled.

See also: cv::exp, cv::cartToPolar, cv::polarToCart, cv::phase, cv::pow, cv::sqrt, cv::magnitude

---

cv::LUT (view/add comments)

Performs a look-up table transform of an array.

```
void LUT(const Mat& src, const Mat& lut, Mat& dst);
```

src Source array of 8-bit elements

lut Look-up table of 256 elements. In the case of multi-channel source array, the table should either have a single channel (in this case the same table is used for all channels) or the same number of channels as in the source array

dst Destination array; will have the same size and the same number of channels as `src`, and the same depth as `lut`

The function `LUT` fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of `src` as follows:

$$\texttt{dst}(I) \leftarrow \texttt{lut(src(I) + d)}$$

where

$$d = \begin{cases} 0 & \text{if } \texttt{src} \text{ has depth } \texttt{CV_8U} \\ 128 & \text{if } \texttt{src} \text{ has depth } \texttt{CV_8S} \end{cases}$$

See also: cv::convertScaleAbs, `Mat::convertTo`

---

cv::magnitude (view/add comments)

Calculates magnitude of 2D vectors.

```
void magnitude(const Mat& x, const Mat& y, Mat& magnitude);
```

x The floating-point array of x-coordinates of the vectors

y The floating-point array of y-coordinates of the vectors; must have the same size as `x`

dst The destination array; will have the same size and same type as `x`

The function `magnitude` calculates magnitude of 2D vectors formed from the corresponding elements of `x` and `y` arrays:

$$\texttt{dst}(I) = \sqrt{\texttt{x}(I)^2 + \texttt{y}(I)^2}$$

See also: cv::cartToPolar, cv::polarToCart, cv::phase, cv::sqrt

---

cv::Mahalanobis (view/add comments)

Calculates the Mahalanobis distance between two vectors.

```
double Mahalanobis(const Mat& vec1, const Mat& vec2,
           const Mat& icovar);
```

vec1 The first 1D source vector

vec2 The second 1D source vector

icovar The inverse covariance matrix

The function `cvMahalonobis` calculates and returns the weighted distance between two vectors:

$$d(\texttt{vec1}, \texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the cv::calcCovarMatrix function and then inverted using the cv::invert function (preferably using DECOMP_SVD method, as the most accurate).

---

cv::max (view/add comments)

Calculates per-element maximum of two arrays or array and a scalar

```
Mat_Expr<...> max(const Mat& src1, const Mat& src2);
Mat_Expr<...> max(const Mat& src1, double value);
Mat_Expr<...> max(double value, const Mat& src1);
void max(const Mat& src1, const Mat& src2, Mat& dst);
void max(const Mat& src1, double value, Mat& dst);
void max(const MatND& src1, const MatND& src2, MatND& dst);
void max(const MatND& src1, double value, MatND& dst);
```

src1 The first source array

src2 The second source array of the same size and type as `src1`

value The real scalar value

dst The destination array; will have the same size and type as `src1`

The functions `max` compute per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

or array and a scalar:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{value})$$

In the second variant, when the source array is multi-channel, each channel is compared with `value` independently.

The first 3 variants of the function listed above are actually a part of Matrix Expressions , they return the expression object that can be further transformed, or assigned to a matrix, or passed to a function etc.

See also: cv::min, cv::compare, cv::inRange, cv::minMaxLoc, Matrix Expressions

---

cv::mean (view/add comments)

Calculates average (mean) of array elements

```
Scalar mean(const Mat& mtx);
Scalar mean(const Mat& mtx, const Mat& mask);
Scalar mean(const MatND& mtx);
Scalar mean(const MatND& mtx, const MatND& mask);
```

mtx The source array; it should have 1 to 4 channels (so that the result can be stored in cv::Scalar)

mask The optional operation mask

The functions `mean` compute mean value `M` of array elements, independently for each channel, and return it:

$$N = \sum_{I:\, \texttt{mask}(I) \neq 0} 1$$
$$M_c = \left(\sum_{I:\, \texttt{mask}(I) \neq 0} \texttt{mtx}(I)_c \right) / N$$

When all the mask elements are 0's, the functions return `Scalar::all(0)`.
See also: cv::countNonZero, cv::meanStdDev, cv::norm, cv::minMaxLoc

---

cv::meanStdDev (view/add comments)

Calculates mean and standard deviation of array elements

```
void meanStdDev(const Mat& mtx, Scalar& mean,
           Scalar& stddev, const Mat& mask=Mat());
void meanStdDev(const MatND& mtx, Scalar& mean,
           Scalar& stddev, const MatND& mask=MatND());
```

mtx The source array; it should have 1 to 4 channels (so that the results can be stored in cv::Scalar's)

mean The output parameter: computed mean value

stddev The output parameter: computed standard deviation

mask The optional operation mask

The functions `meanStdDev` compute the mean and the standard deviation `M` of array elements, independently for each channel, and return it via the output parameters:

$$N = \sum_{I, \texttt{mask}(I) \neq 0} 1$$
$$\texttt{mean}_c = \frac{\sum_{I:\, \texttt{mask}(I) \neq 0} \texttt{src}(I)_c}{N}$$
$$\texttt{stddev}_c = \sqrt{\sum_{I:\, \texttt{mask}(I) \neq 0} \left(\texttt{src}(I)_c - \texttt{mean}_c \right)^2}$$

When all the mask elements are 0's, the functions return `mean=stddev=Scalar::all(0)`. Note that the computed standard deviation is only the diagonal of the complete normalized covariance matrix. If the full matrix is needed, you can reshape the multi-channel array $M \times N$ to the single-channel array $M*N \times \texttt{mtx.channels}()$ (only possible when the matrix is continuous) and then pass the matrix to cv::calcCovarMatrix.

See also: cv::countNonZero, cv::mean, cv::norm, cv::minMaxLoc, cv::calcCovarMatrix

---

cv::merge (view/add comments)

Composes a multi-channel array from several single-channel arrays.

```
void merge(const Mat* mv, size_t count, Mat& dst);
void merge(const vector<Mat>& mv, Mat& dst);
void merge(const MatND* mv, size_t count, MatND& dst);
void merge(const vector<MatND>& mv, MatND& dst);
```

mv The source array or vector of the single-channel matrices to be merged. All the matrices in `mv` must have the same size and the same type

count The number of source matrices when `mv` is a plain C array; must be greater than zero

dst The destination array; will have the same size and the same depth as `mv[0]`, the number of channels will match the number of source matrices

The functions `merge` merge several single-channel arrays (or rather interleave their elements) to make a single multi-channel array.

$$\texttt{dst}(I)_c = \texttt{mv}[c](I)$$

The function cv::split does the reverse operation and if you need to merge several multi-channel images or shuffle channels in some other advanced way, use cv::mixChannels

See also: cv::mixChannels, cv::split, cv::reshape

---

cv::min (view/add comments)

Calculates per-element minimum of two arrays or array and a scalar

```
Mat_Expr<...> min(const Mat& src1, const Mat& src2);
Mat_Expr<...> min(const Mat& src1, double value);
Mat_Expr<...> min(double value, const Mat& src1);
void min(const Mat& src1, const Mat& src2, Mat& dst);
void min(const Mat& src1, double value, Mat& dst);
void min(const MatND& src1, const MatND& src2, MatND& dst);
void min(const MatND& src1, double value, MatND& dst);
```

src1 The first source array

src2 The second source array of the same size and type as `src1`

value The real scalar value

dst The destination array; will have the same size and type as `src1`

The functions `min` compute per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

or array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{value})$$

In the second variant, when the source array is multi-channel, each channel is compared with `value` independently.

The first 3 variants of the function listed above are actually a part of Matrix Expressions , they return the expression object that can be further transformed, or assigned to a matrix, or passed to a function etc.

See also: cv::max, cv::compare, cv::inRange, cv::minMaxLoc, Matrix Expressions

---

cv::minMaxLoc (view/add comments)

Finds global minimum and maximum in a whole array or sub-array

```
void minMaxLoc(const Mat& src, double* minVal,
               double* maxVal=0, Point* minLoc=0,
```

```
            Point* maxLoc=0, const Mat& mask=Mat());
void minMaxLoc(const MatND& src, double* minVal,
            double* maxVal, int* minIdx=0, int* maxIdx=0,
            const MatND& mask=MatND());
void minMaxLoc(const SparseMat& src, double* minVal,
            double* maxVal, int* minIdx=0, int* maxIdx=0);
```

src The source single-channel array

minVal Pointer to returned minimum value; NULL if not required

maxVal Pointer to returned maximum value; NULL if not required

minLoc Pointer to returned minimum location (in 2D case); NULL if not required

maxLoc Pointer to returned maximum location (in 2D case); NULL if not required

minIdx Pointer to returned minimum location (in nD case); NULL if not required, otherwise must point to an array of src.dims elements and the coordinates of minimum element in each dimensions will be stored sequentially there.

maxIdx Pointer to returned maximum location (in nD case); NULL if not required

mask The optional mask used to select a sub-array

The functions ninMaxLoc find minimum and maximum element values and their positions. The extremums are searched across the whole array, or, if mask is not an empty array, in the specified array region.

The functions do not work with multi-channel arrays. If you need to find minimum or maximum elements across all the channels, use cv::reshape first to reinterpret the array as single-channel. Or you may extract the particular channel using cv::extractImageCOI or cv::mixChannels or cv::split.

in the case of a sparse matrix the minimum is found among non-zero elements only.

See also: cv::max, cv::min, cv::compare, cv::inRange, cv::extractImageCOI, cv::mixChannels, cv::split, cv::reshape.

---

cv::mixChannels (view/add comments)

Copies specified channels from input arrays to the specified channels of output arrays

```
void mixChannels(const Mat* srcv, int nsrc, Mat* dstv, int ndst,
           const int* fromTo, size_t npairs);
void mixChannels(const MatND* srcv, int nsrc, MatND* dstv, int ndst,
           const int* fromTo, size_t npairs);
void mixChannels(const vector<Mat>& srcv, vector<Mat>& dstv,
           const int* fromTo, int npairs);
void mixChannels(const vector<MatND>& srcv, vector<MatND>& dstv,
           const int* fromTo, int npairs);
```

srcv The input array or vector of matrices. All the matrices must have the same size and the same depth

nsrc The number of elements in `srcv`

dstv The output array or vector of matrices. All the matrices *must be allocated*, their size and depth must be the same as in `srcv[0]`

ndst The number of elements in `dstv`

fromTo The array of index pairs, specifying which channels are copied and where. `fromTo[k*2]` is the 0-based index of the input channel in `srcv` and `fromTo[k*2+1]` is the index of the output channel in `dstv`. Here the continuous channel numbering is used, that is, the first input image channels are indexed from `0` to `srcv[0].channels()-1`, the second input image channels are indexed from `srcv[0].channels()` to `srcv[0].channels() + srcv[1].channels()-1` etc., and the same scheme is used for the output image channels. As a special case, when `fromTo[k*2]` is negative, the corresponding output channel is filled with zero. `npairs`The number of pairs. In the latter case the parameter is not passed explicitly, but computed as `srcv.size()` (=`dstv.size()`)

The functions `mixChannels` provide an advanced mechanism for shuffling image channels. cv::split and cv::merge and some forms of cv::cvtColor are partial cases of `mixChannels`.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B channels swapped) and separate alpha channel image:

```
Mat rgba( 100, 100, CV_8UC4, Scalar(1,2,3,4) );
Mat bgr( rgba.rows, rgba.cols, CV_8UC3 );
Mat alpha( rgba.rows, rgba.cols, CV_8UC1 );

// forming array of matrices is quite efficient operations,
```

```
// because the matrix data is not copied, only the headers
Mat out[] = { bgr, alpha };
// rgba[0] -> bgr[2], rgba[1] -> bgr[1],
// rgba[2] -> bgr[0], rgba[3] -> alpha[0]
int from_to[] = { 0,2,  1,1,  2,0,  3,3 };
mixChannels( &rgba, 1, out, 2, from_to, 4 );
```

Note that, unlike many other new-style C++ functions in OpenCV (see the introduction section and cv::Mat::create), `mixChannels` requires the destination arrays be pre-allocated before calling the function.

See also: cv::split, cv::merge, cv::cvtColor

---

cv::mulSpectrums (view/add comments)

Performs per-element multiplication of two Fourier spectrums.

```
void mulSpectrums(const Mat& src1, const Mat& src2, Mat& dst,
            int flags, bool conj=false);
```

src1 The first source array

src2 The second source array; must have the same size and the same type as `src1`

dst The destination array; will have the same size and the same type as `src1`

flags The same flags as passed to cv::dft; only the flag `DFT_ROWS` is checked for

conj The optional flag that conjugate the second source array before the multiplication (true) or not (false)

The function `mulSpectrums` performs per-element multiplication of the two CCS-packed or complex matrices that are results of a real or complex Fourier transform.

The function, together with cv::dft and cv::idft, may be used to calculate convolution (pass `conj=false`) or correlation (pass `conj=false`) of two arrays rapidly. When the arrays are complex, they are simply multiplied (per-element) with optional conjugation of the second array elements. When the arrays are real, they assumed to be CCS-packed (see cv::dft for details).

cv::multiply (view/add comments)

Calculates the per-element scaled product of two arrays

```
void multiply(const Mat& src1, const Mat& src2,
           Mat& dst, double scale=1);
void multiply(const MatND& src1, const MatND& src2,
           MatND& dst, double scale=1);
```

src1 The first source array

src2 The second source array of the same size and the same type as `src1`

dst The destination array; will have the same size and the same type as `src1`

scale The optional scale factor

The function `multiply` calculates the per-element product of two arrays:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I))$$

There is also Matrix Expressions -friendly variant of the first function, see cv::Mat::mul.
If you are looking for a matrix product, not per-element product, see cv::gemm.
See also: cv::add, cv::substract, cv::divide, Matrix Expressions , cv::scaleAdd, cv::addWeighted, cv::accumulate, cv::accumulateProduct, cv::accumulateSquare, cv::Mat::convertTo

cv::mulTransposed (view/add comments)

Calculates the product of a matrix and its transposition.

```
void mulTransposed( const Mat& src, Mat& dst, bool aTa,
           const Mat& delta=Mat(),
           double scale=1, int rtype=-1 );
```

src The source matrix

dst The destination square matrix

aTa Specifies the multiplication ordering; see the description below

delta The optional delta matrix, subtracted from `src` before the multiplication. When the matrix is empty (`delta=Mat()`), it's assumed to be zero, i.e. nothing is subtracted, otherwise if it has the same size as `src`, then it's simply subtracted, otherwise it is "repeated" (see cv::repeat) to cover the full `src` and then subtracted. Type of the delta matrix, when it's not empty, must be the same as the type of created destination matrix, see the `rtype` description

scale The optional scale factor for the matrix product

rtype When it's negative, the destination matrix will have the same type as `src`. Otherwise, it will have `type=CV_MAT_DEPTH(rtype)`, which should be either `CV_32F` or `CV_64F`

The function `mulTransposed` calculates the product of `src` and its transposition:

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})^T(\texttt{src} - \texttt{delta})$$

if `aTa=true`, and

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})(\texttt{src} - \texttt{delta})^T$$

otherwise. The function is used to compute covariance matrix and with zero delta can be used as a faster substitute for general matrix product $A * B$ when $B = A^T$.

See also: cv::calcCovarMatrix, cv::gemm, cv::repeat, cv::reduce

---

cv::norm (view/add comments)

Calculates absolute array norm, absolute difference norm, or relative difference norm.

```
double norm(const Mat& src1, int normType=NORM_L2);
double norm(const Mat& src1, const Mat& src2, int normType=NORM_L2);
double norm(const Mat& src1, int normType, const Mat& mask);
double norm(const Mat& src1, const Mat& src2,
            int normType, const Mat& mask);
double norm(const MatND& src1, int normType=NORM_L2,
            const MatND& mask=MatND());
double norm(const MatND& src1, const MatND& src2,
            int normType=NORM_L2, const MatND& mask=MatND());
double norm( const SparseMat& src, int normType );
```

src1 The first source array

src2 The second source array of the same size and the same type as src1

normType Type of the norm; see the discussion below

mask The optional operation mask

The functions norm calculate the absolute norm of src1 (when there is no src2):

$$norm = \begin{cases} \|\texttt{src1}\|_{L_\infty} = \max_I |\texttt{src1}(I)| & \text{if } \texttt{normType} = \texttt{NORM_INF} \\ \|\texttt{src1}\|_{L_1} = \sum_I |\texttt{src1}(I)| & \text{if } \texttt{normType} = \texttt{NORM_L1} \\ \|\texttt{src1}\|_{L_2} = \sqrt{\sum_I \texttt{src1}(I)^2} & \text{if } \texttt{normType} = \texttt{NORM_L2} \end{cases}$$

or an absolute or relative difference norm if src2 is there:

$$norm = \begin{cases} \|\texttt{src1} - \texttt{src2}\|_{L_\infty} = \max_I |\texttt{src1}(I) - \texttt{src2}(I)| & \text{if } \texttt{normType} = \texttt{NORM_INF} \\ \|\texttt{src1} - \texttt{src2}\|_{L_1} = \sum_I |\texttt{src1}(I) - \texttt{src2}(I)| & \text{if } \texttt{normType} = \texttt{NORM_L1} \\ \|\texttt{src1} - \texttt{src2}\|_{L_2} = \sqrt{\sum_I (\texttt{src1}(I) - \texttt{src2}(I))^2} & \text{if } \texttt{normType} = \texttt{NORM_L2} \end{cases}$$

or

$$norm = \begin{cases} \frac{\|\texttt{src1} - \texttt{src2}\|_{L_\infty}}{\|\texttt{src2}\|_{L_\infty}} & \text{if } \texttt{normType} = \texttt{NORM_RELATIVE_INF} \\ \frac{\|\texttt{src1} - \texttt{src2}\|_{L_1}}{\|\texttt{src2}\|_{L_1}} & \text{if } \texttt{normType} = \texttt{NORM_RELATIVE_L1} \\ \frac{\|\texttt{src1} - \texttt{src2}\|_{L_2}}{\|\texttt{src2}\|_{L_2}} & \text{if } \texttt{normType} = \texttt{NORM_RELATIVE_L2} \end{cases}$$

The functions norm return the calculated norm.

When there is mask parameter, and it is not empty (then it should have type CV_8U and the same size as src1), the norm is computed only over the specified by the mask region.

A multiple-channel source arrays are treated as a single-channel, that is, the results for all channels are combined.

---

cv::normalize (view/add comments)

Normalizes array's norm or the range

```
void normalize( const Mat& src, Mat& dst,
           double alpha=1, double beta=0,
           int normType=NORM_L2, int rtype=-1,
```

```
            const Mat& mask=Mat());
void normalize( const MatND& src, MatND& dst,
            double alpha=1, double beta=0,
            int normType=NORM_L2, int rtype=-1,
            const MatND& mask=MatND());
void normalize( const SparseMat& src, SparseMat& dst,
            double alpha, int normType );
```

src The source array

dst The destination array; will have the same size as `src`

alpha The norm value to normalize to or the lower range boundary in the case of range normalization

beta The upper range boundary in the case of range normalization; not used for norm normalization

normType The normalization type, see the discussion

rtype When the parameter is negative, the destination array will have the same type as `src`, otherwise it will have the same number of channels as `src` and the depth=`CV_MAT_DEPTH(rtype)`

mask The optional operation mask

The functions `normalize` scale and shift the source array elements, so that

$$\|\texttt{dst}\|_{L_p} = \texttt{alpha}$$

(where $p = \infty$, 1 or 2) when `normType=NORM_INF`, `NORM_L1` or `NORM_L2`, or so that

$$\min_I \texttt{dst}(I) = \texttt{alpha}, \ \max_I \texttt{dst}(I) = \texttt{beta}$$

when `normType=NORM_MINMAX` (for dense arrays only).

The optional mask specifies the sub-array to be normalize, that is, the norm or min-n-max are computed over the sub-array and then this sub-array is modified to be normalized. If you want to only use the mask to compute the norm or min-max, but modify the whole array, you can use cv::norm and cv::Mat::convertScale/ cv::MatND::convertScale/crossSparseMat::convertScale separately.

in the case of sparse matrices, only the non-zero values are analyzed and transformed. Because of this, the range transformation for sparse matrices is not allowed, since it can shift the zero level.

See also: cv::norm, cv::Mat::convertScale, cv::MatND::convertScale, cv::SparseMat::convertScale

cv::PCA (view/add comments)

Class for Principal Component Analysis

```
class PCA
{
public:
    // default constructor
    PCA();
    // computes PCA for a set of vectors stored as data rows or columns.
    PCA(const Mat& data, const Mat& mean, int flags, int maxComponents=0);
    // computes PCA for a set of vectors stored as data rows or columns
    PCA& operator()(const Mat& data, const Mat& mean, int flags, int maxComponents=0);
    // projects vector into the principal components space
    Mat project(const Mat& vec) const;
    void project(const Mat& vec, Mat& result) const;
    // reconstructs the vector from its PC projection
    Mat backProject(const Mat& vec) const;
    void backProject(const Mat& vec, Mat& result) const;

    // eigenvectors of the PC space, stored as the matrix rows
    Mat eigenvectors;
    // the corresponding eigenvalues; not used for PCA compression/decompression
    Mat eigenvalues;
    // mean vector, subtracted from the projected vector
    // or added to the reconstructed vector
    Mat mean;
};
```

The class `PCA` is used to compute the special basis for a set of vectors. The basis will consist of eigenvectors of the covariance matrix computed from the input set of vectors. And also the class `PCA` can transform vectors to/from the new coordinate space, defined by the basis. Usually, in this new coordinate system each vector from the original set (and any linear combination of such vectors) can be quite accurately approximated by taking just the first few its components, corresponding to the eigenvectors of the largest eigenvalues of the covariance matrix. Geometrically it means that we compute projection of the vector to a subspace formed by a few eigenvectors corresponding to the dominant eigenvalues of the covariation matrix. And usually such a projection is very close to the original vector. That is, we can represent the original vector from a high-dimensional space with a much shorter vector consisting of the projected vector's coordinates in the subspace. Such a transformation is also known as Karhunen-Loeve Transform, or KLT. See http://en.wikipedia.org/wiki/Principal_component_analysis

The following sample is the function that takes two matrices. The first one stores the set of vectors (a row per vector) that is used to compute PCA, the second one stores another "test" set of vectors (a row per vector) that are first compressed with PCA, then reconstructed back and then

the reconstruction error norm is computed and printed for each vector.

```
PCA compressPCA(const Mat& pcaset, int maxComponents,
                const Mat& testset, Mat& compressed)
{
    PCA pca(pcaset, // pass the data
            Mat(), // we do not have a pre-computed mean vector,
                   // so let the PCA engine to compute it
            CV_PCA_DATA_AS_ROW, // indicate that the vectors
                                // are stored as matrix rows
                                // (use CV_PCA_DATA_AS_COL if the vectors are
                                // the matrix columns)
            maxComponents // specify, how many principal components to retain
            );
    // if there is no test data, just return the computed basis, ready-to-use
    if( !testset.data )
        return pca;
    CV_Assert( testset.cols == pcaset.cols );

    compressed.create(testset.rows, maxComponents, testset.type());

    Mat reconstructed;
    for( int i = 0; i < testset.rows; i++ )
    {
        Mat vec = testset.row(i), coeffs = compressed.row(i);
        // compress the vector, the result will be stored
        // in the i-th row of the output matrix
        pca.project(vec, coeffs);
        // and then reconstruct it
        pca.backProject(coeffs, reconstructed);
        // and measure the error
        printf("%d. diff = %g\n", i, norm(vec, reconstructed, NORM_L2));
    }
    return pca;
}
```

See also: cv::calcCovarMatrix, cv::mulTransposed, cv::SVD, cv::dft, cv::dct

---

cv::PCA::PCA (view/add comments)

PCA constructors

```
PCA::PCA();
PCA::PCA(const Mat& data, const Mat& mean, int flags, int
maxComponents=0);
```

data the input samples, stored as the matrix rows or as the matrix columns

mean the optional mean value. If the matrix is empty (`Mat()`), the mean is computed from the data.

flags operation flags. Currently the parameter is only used to specify the data layout.

CV_PCA_DATA_AS_ROWS Indicates that the input samples are stored as matrix rows.

CV_PCA_DATA_AS_COLS Indicates that the input samples are stored as matrix columns.

maxComponents The maximum number of components that PCA should retain. By default, all the components are retained.

The default constructor initializes empty PCA structure. The second constructor initializes the structure and calls cv::PCA::operator ().

---

cv::PCA::operator () (view/add comments)

Performs Principal Component Analysis of the supplied dataset.

```
PCA& PCA::operator()(const Mat& data, const Mat& mean, int flags, int
maxComponents=0);
```

data the input samples, stored as the matrix rows or as the matrix columns

mean the optional mean value. If the matrix is empty (`Mat()`), the mean is computed from the data.

flags operation flags. Currently the parameter is only used to specify the data layout.

CV_PCA_DATA_AS_ROWS Indicates that the input samples are stored as matrix rows.

CV_PCA_DATA_AS_COLS Indicates that the input samples are stored as matrix columns.

maxComponents The maximum number of components that PCA should retain. By default, all the components are retained.

The operator performs PCA of the supplied dataset. It is safe to reuse the same PCA structure for multiple dataset. That is, if the structure has been previously used with another dataset, the existing internal data is reclaimed and the new `eigenvalues`, `eigenvectors` and `mean` are allocated and computed.

The computed eigenvalues are sorted from the largest to the smallest and the corresponding eigenvectors are stored as `PCA::eigenvectors` rows.

---

cv::PCA::project (view/add comments)

Project vector(s) to the principal component subspace

```
Mat PCA::project(const Mat& vec) const;
void PCA::project(const Mat& vec, Mat& result) const;
```

vec the input vector(s). They have to have the same dimensionality and the same layout as the input data used at PCA phase. That is, if `CV_PCA_DATA_AS_ROWS` had been specified, then `vec.cols==data.cols` (that's vectors' dimensionality) and `vec.rows` is the number of vectors to project; and similarly for the `CV_PCA_DATA_AS_COLS` case.

result the output vectors. Let's now consider `CV_PCA_DATA_AS_COLS` case. In this case the output matrix will have as many columns as the number of input vectors, i.e. `result.cols==vec.cols` and the number of rows will match the number of principal components (e.g. `maxComponents` parameter passed to the constructor).

The methods project one or more vectors to the principal component subspace, where each vector projection is represented by coefficients in the principal component basis. The first form of the method returns the matrix that the second form writes to the result. So the first form can be used as a part of expression, while the second form can be more efficient in a processing loop.

---

cv::PCA::backProject (view/add comments)

Reconstruct vectors from their PC projections.

```
Mat PCA::backProject(const Mat& vec) const;
void PCA::backProject(const Mat& vec, Mat& result) const;
```

vec Coordinates of the vectors in the principal component subspace. The layout and size are the same as of `PCA::project` output vectors.

result The reconstructed vectors. The layout and size are the same as of `PCA::project` input vectors.

The methods are inverse operations to cv::PCA::project. They take PC coordinates of projected vectors and reconstruct the original vectors. Of course, unless all the principal components have been retained, the reconstructed vectors will be different from the originals, but typically the difference will be small is if the number of components is large enough (but still much smaller than the original vector dimensionality) - that's why PCA is used after all.

---

cv::perspectiveTransform (view/add comments)

Performs perspective matrix transformation of vectors.

```
void perspectiveTransform(const Mat& src,
           Mat& dst, const Mat& mtx );
```

src The source two-channel or three-channel floating-point array; each element is 2D/3D vector to be transformed

dst The destination array; it will have the same size and same type as `src`

mtx 3×3 or 4×4 transformation matrix

The function `perspectiveTransform` transforms every element of `src`, by treating it as 2D or 3D vector, in the following way (here 3D vector transformation is shown; in the case of 2D vector transformation the z component is omitted):

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Note that the function transforms a sparse set of 2D or 3D vectors. If you want to transform an image using perspective transformation, use cv::warpPerspective. If you have an inverse task, i.e. want to compute the most probable perspective transformation out of several pairs of corresponding points, you can use cv::getPerspectiveTransform or cv::findHomography.

See also: cv::transform, cv::warpPerspective, cv::getPerspectiveTransform, cv::findHomography

---

cv::phase (view/add comments)

Calculates the rotation angle of 2d vectors

```
void phase(const Mat& x, const Mat& y, Mat& angle,
           bool angleInDegrees=false);
```

x The source floating-point array of x-coordinates of 2D vectors

y The source array of y-coordinates of 2D vectors; must have the same size and the same type as x

angle The destination array of vector angles; it will have the same size and same type as x

angleInDegrees When it is true, the function will compute angle in degrees, otherwise they will be measured in radians

The function phase computes the rotation angle of each 2D vector that is formed from the corresponding elements of x and y:

$$\texttt{angle}(I) = \texttt{atan2}(\texttt{y}(I), \texttt{x}(I))$$

The angle estimation accuracy is $\sim 0.3^\circ$, when x(I)=y(I)=0, the corresponding angle(I) is set to 0.

See also:

cv::polarToCart (view/add comments)

Computes x and y coordinates of 2D vectors from their magnitude and angle.

```
void polarToCart(const Mat& magnitude, const Mat& angle,
           Mat& x, Mat& y, bool angleInDegrees=false);
```

magnitude The source floating-point array of magnitudes of 2D vectors. It can be an empty matrix (=Mat()) - in this case the function assumes that all the magnitudes are =1. If it's not empty, it must have the same size and same type as angle

angle The source floating-point array of angles of the 2D vectors

x The destination array of x-coordinates of 2D vectors; will have the same size and the same type as angle

y The destination array of y-coordinates of 2D vectors; will have the same size and the same type as angle

angleInDegrees When it is true, the input angles are measured in degrees, otherwise they are measured in radians

The function polarToCart computes the cartesian coordinates of each 2D vector represented by the corresponding elements of magnitude and angle:

$$\begin{aligned} \texttt{x}(I) &= \texttt{magnitude}(I) \cos(\texttt{angle}(I)) \\ \texttt{y}(I) &= \texttt{magnitude}(I) \sin(\texttt{angle}(I)) \end{aligned}$$

The relative accuracy of the estimated coordinates is $\sim 10^{-6}$.

See also: cv::cartToPolar, cv::magnitude, cv::phase, cv::exp, cv::log, cv::pow, cv::sqrt

cv::pow (view/add comments)

Raises every array element to a power.

```
void pow(const Mat& src, double p, Mat& dst);
void pow(const MatND& src, double p, MatND& dst);
```

src The source array

p The exponent of power

dst The destination array; will have the same size and the same type as `src`

The function `pow` raises every element of the input array to `p`:

$$\texttt{dst}(I) = \begin{cases} \texttt{src}(I)^p & \text{if } \texttt{p} \text{ is integer} \\ |\texttt{src}(I)|^p & \text{otherwise} \end{cases}$$

That is, for a non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following example, computing the 5th root of array `src`, shows:

```
Mat mask = src < 0;
pow(src, 1./5, dst);
subtract(Scalar::all(0), dst, dst, mask);
```

For some values of `p`, such as integer values, 0.5, and -0.5, specialized faster algorithms are used.

See also: cv::sqrt, cv::exp, cv::log, cv::cartToPolar, cv::polarToCart

---

RNG

Random number generator class.

```
class CV_EXPORTS RNG
{
public:
    enum { UNIFORM=0, NORMAL=1 };

    // constructors
    RNG();
    RNG(uint64 state);

    // returns 32-bit unsigned random number
    unsigned next();

    // return random numbers of the specified type
    operator uchar();
    operator schar();
    operator ushort();
    operator short();
    operator unsigned();
        // returns a random integer sampled uniformly from [0, N).
```

```
        unsigned operator()(unsigned N);
        unsigned operator()();
    operator int();
    operator float();
    operator double();
    // returns a random number sampled uniformly from [a, b) range
    int uniform(int a, int b);
    float uniform(float a, float b);
    double uniform(double a, double b);

    // returns Gaussian random number with zero mean.
        double gaussian(double sigma);

    // fills array with random numbers sampled from the specified distribution
    void fill( Mat& mat, int distType, const Scalar& a, const Scalar& b );
    void fill( MatND& mat, int distType, const Scalar& a, const Scalar& b );

    // internal state of the RNG (could change in the future)
    uint64 state;
};
```

The class `RNG` implements random number generator. It encapsulates the RNG state (currently, a 64-bit integer) and has methods to return scalar random values and to fill arrays with random values. Currently it supports uniform and Gaussian (normal) distributions. The generator uses Multiply-With-Carry algorithm, introduced by G. Marsaglia (http://en.wikipedia.org/wiki/Multiply-with-carry). Gaussian-distribution random numbers are generated using Ziggurat algorithm (http://en.wikipedia.org/wiki/Ziggurat_algorithm), introduced by G. Marsaglia and W. W. Tsang.

---

cv::RNG::RNG (view/add comments)

RNG constructors

```
RNG::RNG();
RNG::RNG(uint64 state);
```

state the 64-bit value used to initialize the RNG

These are the RNG constructors. The first form sets the state to some pre-defined value, equal to `2**32-1` in the current implementation. The second form sets the state to the specified value.

If the user passed `state=0`, the constructor uses the above default value instead, to avoid the singular random number sequence, consisting of all zeros.

---

cv::RNG::next (view/add comments)

Returns the next random number

```
unsigned RNG::next();
```

The method updates the state using MWC algorithm and returns the next 32-bit random number.

---

cv::RNG::operator T (view/add comments)

Returns the next random number of the specified type

```
RNG::operator uchar(); RNG::operator schar(); RNG::operator ushort();
RNG::operator short(); RNG::operator unsigned(); RNG::operator int();
RNG::operator float(); RNG::operator double();
```

Each of the methods updates the state using MWC algorithm and returns the next random number of the specified type. In the case of integer types the returned number is from the whole available value range for the specified type. In the case of floating-point types the returned value is from `[0,1)` range.

---

cv::RNG::operator () (view/add comments)

Returns the next random number

```
unsigned RNG::operator ()();
unsigned RNG::operator ()(unsigned N);
```

N The upper non-inclusive boundary of the returned random number

The methods transforms the state using MWC algorithm and returns the next random number. The first form is equivalent to cv::RNG::next, the second form returns the random number modulo `N`, i.e. the result will be in the range `[0, N)`.

---

cv::RNG::uniform (view/add comments)

Returns the next random number sampled from the uniform distribution

```
int RNG::uniform(int a, int b);
float RNG::uniform(float a, float b);
double RNG::uniform(double a, double b);
```

a The lower inclusive boundary of the returned random numbers

b The upper non-inclusive boundary of the returned random numbers

The methods transforms the state using MWC algorithm and returns the next uniformly-distributed random number of the specified type, deduced from the input parameter type, from the range `[a, b)`. There is one nuance, illustrated by the following sample:

```
cv::RNG rng;

// will always produce 0
double a = rng.uniform(0, 1);

// will produce double from [0, 1)
double a1 = rng.uniform((double)0, (double)1);

// will produce float from [0, 1)
double b = rng.uniform(0.f, 1.f);

// will produce double from [0, 1)
double c = rng.uniform(0., 1.);

// will likely cause compiler error because of ambiguity:
//  RNG::uniform(0, (int)0.999999)? or RNG::uniform((double)0, 0.99999)?
double d = rng.uniform(0, 0.999999);
```

That is, the compiler does not take into account type of the variable that you assign the result of `RNG::uniform` to, the only thing that matters to it is the type of `a` and `b` parameters. So if you want a floating-point random number, but the range boundaries are integer numbers, either put dots in the end, if they are constants, or use explicit type cast operators, as in `a1` initialization above.

---

cv::RNG::gaussian (view/add comments)

Returns the next random number sampled from the Gaussian distribution

```
double RNG::gaussian(double sigma);
```

sigma The standard deviation of the distribution

The methods transforms the state using MWC algorithm and returns the next random number from the Gaussian distribution `N(0,sigma)`. That is, the mean value of the returned random numbers will be zero and the standard deviation will be the specified `sigma`.

---

cv::RNG::fill (view/add comments)

Fill arrays with random numbers

```
void RNG::fill( Mat& mat, int distType, const Scalar& a, const Scalar&
b );
void RNG::fill( MatND& mat, int distType, const Scalar& a, const
Scalar& b );
```

mat 2D or N-dimensional matrix. Currently matrices with more than 4 channels are not supported by the methods. Use cv::reshape as a possible workaround.

distType The distribution type, `RNG::UNIFORM` or `RNG::NORMAL`

a The first distribution parameter. In the case of uniform distribution this is inclusive lower boundary. In the case of normal distribution this is mean value.

b The second distribution parameter. In the case of uniform distribution this is non-inclusive upper boundary. In the case of normal distribution this is standard deviation.

Each of the methods fills the matrix with the random values from the specified distribution. As the new numbers are generated, the RNG state is updated accordingly. In the case of multiple-channel images every channel is filled independently, i.e. RNG can not generate samples from multi-dimensional Gaussian distribution with non-diagonal covariation matrix directly. To do that, first, generate matrix from the distribution $N(0, I_n)$, i.e. Gaussian distribution with zero mean and identity covariation matrix, and then transform it using cv::transform and the specific covariation matrix.

---

cv::randu (view/add comments)

Generates a single uniformly-distributed random number or array of random numbers

```
template<typename _Tp> _Tp randu();
void randu(Mat& mtx, const Scalar& low, const Scalar& high);
```

mtx The output array of random numbers. The array must be pre-allocated and have 1 to 4 channels

low The inclusive lower boundary of the generated random numbers

high The exclusive upper boundary of the generated random numbers

The template functions `randu` generate and return the next uniformly-distributed random value of the specified type. `randu<int>()` is equivalent to `(int)theRNG();` etc. See cv::RNG description.

The second non-template variant of the function fills the matrix `mtx` with uniformly-distributed random numbers from the specified range:

$$\texttt{low}_c \leq \texttt{mtx}(I)_c < \texttt{high}_c$$

See also: cv::RNG, cv::randn, cv::theRNG.

---

cv::randn (view/add comments)

Fills array with normally distributed random numbers

```
void randn(Mat& mtx, const Scalar& mean, const Scalar& stddev);
```

mtx The output array of random numbers. The array must be pre-allocated and have 1 to 4 channels

mean The mean value (expectation) of the generated random numbers

stddev The standard deviation of the generated random numbers

The function `randn` fills the matrix `mtx` with normally distributed random numbers with the specified mean and standard deviation. saturate_cast is applied to the generated numbers (i.e. the values are clipped)

See also: cv::RNG, cv::randu

---

cv::randShuffle (view/add comments)

Shuffles the array elements randomly

```
void randShuffle(Mat& mtx, double iterFactor=1., RNG* rng=0);
```

mtx The input/output numerical 1D array

iterFactor The scale factor that determines the number of random swap operations. See the discussion

rng The optional random number generator used for shuffling. If it is zero, cv::theRNG() is used instead

The function `randShuffle` shuffles the specified 1D array by randomly choosing pairs of elements and swapping them. The number of such swap operations will be `mtx.rows*mtx.cols*iterFactor`

See also: cv::RNG, cv::sort

cv::reduce (view/add comments)

Reduces a matrix to a vector

```
void reduce(const Mat& mtx, Mat& vec,
            int dim, int reduceOp, int dtype=-1);
```

mtx The source 2D matrix

vec The destination vector. Its size and type is defined by `dim` and `dtype` parameters

dim The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row and 1 means that the matrix is reduced to a single column

reduceOp The reduction operation, one of:

- **CV_REDUCE_SUM** The output is the sum of all of the matrix's rows/columns.
- **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
- **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/-columns.
- **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/-columns.

dtype When it is negative, the destination vector will have the same type as the source matrix, otherwise, its type will be `CV_MAKE_TYPE(CV_MAT_DEPTH(dtype), mtx.channels())`

The function `reduce` reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In the case of `CV_REDUCE_SUM` and `CV_REDUCE_AVG` the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

See also: cv::repeat

cv::repeat (view/add comments)

Fill the destination array with repeated copies of the source array.

```
void repeat(const Mat& src, int ny, int nx, Mat& dst);
Mat repeat(const Mat& src, int ny, int nx);
```

src The source array to replicate

dst The destination array; will have the same type as `src`

ny How many times the `src` is repeated along the vertical axis

nx How many times the `src` is repeated along the horizontal axis

The functions cv::repeat duplicate the source array one or more times along each of the two axes:

$$\texttt{dst}_{ij} = \texttt{src}_{i \mod \texttt{src.rows}, j \mod \texttt{src.cols}}$$

The second variant of the function is more convenient to use with Matrix Expressions
See also: cv::reduce, Matrix Expressions

---

saturate_cast

Template function for accurate conversion from one primitive type to another

```
template<typename _Tp> inline _Tp saturate_cast(unsigned char v);
template<typename _Tp> inline _Tp saturate_cast(signed char v);
template<typename _Tp> inline _Tp saturate_cast(unsigned short v);
template<typename _Tp> inline _Tp saturate_cast(signed short v);
template<typename _Tp> inline _Tp saturate_cast(int v);
template<typename _Tp> inline _Tp saturate_cast(unsigned int v);
template<typename _Tp> inline _Tp saturate_cast(float v);
template<typename _Tp> inline _Tp saturate_cast(double v);
```

v The function parameter

The functions `saturate_cast` resembles the standard C++ cast operations, such as `static_cast<T>()` etc. They perform an efficient and accurate conversion from one primitive type to another, see the introduction. "saturate" in the name means that when the input value `v` is out of range of the target type, the result will not be formed just by taking low bits of the input, but instead the value will be clipped. For example:

```
uchar a = saturate_cast<uchar>(-100); // a = 0 (UCHAR_MIN)
short b = saturate_cast<short>(33333.33333); // b = 32767 (SHRT_MAX)
```

Such clipping is done when the target type is `unsigned char, signed char, unsigned short or signed short` - for 32-bit integers no clipping is done.

When the parameter is floating-point value and the target type is an integer (8-, 16- or 32-bit), the floating-point value is first rounded to the nearest integer and then clipped if needed (when the target type is 8- or 16-bit).

This operation is used in most simple or complex image processing functions in OpenCV.

See also: cv::add, cv::subtract, cv::multiply, cv::divide, cv::Mat::convertTo

---

cv::scaleAdd (view/add comments)

Calculates the sum of a scaled array and another array.

```
void scaleAdd(const Mat& src1, double scale,
              const Mat& src2, Mat& dst);
void scaleAdd(const MatND& src1, double scale,
              const MatND& src2, MatND& dst);
```

src1 The first source array

scale Scale factor for the first array

src2 The second source array; must have the same size and the same type as `src1`

dst The destination array; will have the same size and the same type as `src1`

The function `cvScaleAdd` is one of the classical primitive linear algebra operations, known as `DAXPY` or `SAXPY` in BLAS. It calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) + \texttt{src2}(I)$$

The function can also be emulated with a matrix expression, for example:

```
Mat A(3, 3, CV_64F);
...
A.row(0) = A.row(1)*2 + A.row(2);
```

See also: cv::add, cv::addWeighted, cv::subtract, cv::Mat::dot, cv::Mat::convertTo, Matrix Expressions

---

cv::setIdentity (view/add comments)

Initializes a scaled identity matrix

```
void setIdentity(Mat& dst, const Scalar& value=Scalar(1));
```

dst The matrix to initialize (not necessarily square)

value The value to assign to the diagonal elements

The function cv::setIdentity initializes a scaled identity matrix:

$$\texttt{dst}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The function can also be emulated using the matrix initializers and the matrix expressions:

```
Mat A = Mat::eye(4, 3, CV_32F)*5;
// A will be set to [[5, 0, 0], [0, 5, 0], [0, 0, 5], [0, 0, 0]]
```

See also: cv::Mat::zeros, cv::Mat::ones, Matrix Expressions , cv::Mat::setTo, cv::Mat::operator=,

---

cv::solve (view/add comments)

Solves one or more linear systems or least-squares problems.

```
bool solve(const Mat& src1, const Mat& src2,
           Mat& dst, int flags=DECOMP_LU);
```

src1 The input matrix on the left-hand side of the system

src2 The input matrix on the right-hand side of the system

dst The output solution

flags The solution (matrix inversion) method

- **DECOMP_LU** Gaussian elimination with optimal pivot element chosen
- **DECOMP_CHOLESKY** Cholesky LL^T factorization; the matrix `src1` must be symmetrical and positively defined
- **DECOMP_EIG** Eigenvalue decomposition; the matrix `src1` must be symmetrical
- **DECOMP_SVD** Singular value decomposition (SVD) method; the system can be over-defined and/or the matrix `src1` can be singular
- **DECOMP_QR** QR factorization; the system can be over-defined and/or the matrix `src1` can be singular
- **DECOMP_NORMAL** While all the previous flags are mutually exclusive, this flag can be used together with any of the previous. It means that the normal equations $\texttt{src1}^T \cdot \texttt{src1} \cdot \texttt{dst} = \texttt{src1}^T \texttt{src2}$ are solved instead of the original system $\texttt{src1} \cdot \texttt{dst} = \texttt{src2}$

The function `solve` solves a linear system or least-squares problem (the latter is possible with SVD or QR methods, or by specifying the flag `DECOMP_NORMAL`):

$$\texttt{dst} = \arg\min_X \|\texttt{src1} \cdot \texttt{X} - \texttt{src2}\|$$

If `DECOMP_LU` or `DECOMP_CHOLESKY` method is used, the function returns 1 if `src1` (or $\texttt{src1}^T\texttt{src1}$) is non-singular and 0 otherwise; in the latter case `dst` is not valid. Other methods find some pseudo-solution in the case of singular left-hand side part.

Note that if you want to find unity-norm solution of an under-defined singular system $\texttt{src1} \cdot \texttt{dst} = 0$, the function `solve` will not do the work. Use cv::SVD::solveZ instead.

See also: cv::invert, cv::SVD, cv::eigen

---

cv::solveCubic (view/add comments)

Finds the real roots of a cubic equation.

```
void solveCubic(const Mat& coeffs, Mat& roots);
```

coeffs The equation coefficients, an array of 3 or 4 elements

roots The destination array of real roots which will have 1 or 3 elements

The function `solveCubic` finds the real roots of a cubic equation:
(if coeffs is a 4-element vector)

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or (if coeffs is 3-element vector):

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

The roots are stored to `roots` array.

---

cv::solvePoly (view/add comments)

Finds the real or complex roots of a polynomial equation

```
void solvePoly(const Mat& coeffs, Mat& roots,
           int maxIters=20, int fig=100);
```

coeffs The array of polynomial coefficients

roots The destination (complex) array of roots

maxIters The maximum number of iterations the algorithm does

fig

The function `solvePoly` finds real and complex roots of a polynomial equation:

$$\texttt{coeffs}[0]x^n + \texttt{coeffs}[1]x^{n-1} + ... + \texttt{coeffs}[n-1]x + \texttt{coeffs}[n] = 0$$

---

cv::sort (view/add comments)

Sorts each row or each column of a matrix

```
void sort(const Mat& src, Mat& dst, int flags);
```

src The source single-channel array

dst The destination array of the same size and the same type as `src`

flags The operation flags, a combination of the following values:

- **CV_SORT_EVERY_ROW** Each matrix row is sorted independently
- **CV_SORT_EVERY_COLUMN** Each matrix column is sorted independently. This flag and the previous one are mutually exclusive
- **CV_SORT_ASCENDING** Each matrix row is sorted in the ascending order
- **CV_SORT_DESCENDING** Each matrix row is sorted in the descending order. This flag and the previous one are also mutually exclusive

The function `sort` sorts each matrix row or each matrix column in ascending or descending order. If you want to sort matrix rows or columns lexicographically, you can use STL `std::sort` generic function with the proper comparison predicate.

See also: cv::sortIdx, cv::randShuffle

---

cv::sortIdx (view/add comments)

Sorts each row or each column of a matrix

```
void sortIdx(const Mat& src, Mat& dst, int flags);
```

src The source single-channel array

dst The destination integer array of the same size as `src`

flags The operation flags, a combination of the following values:

- **CV_SORT_EVERY_ROW** Each matrix row is sorted independently
- **CV_SORT_EVERY_COLUMN** Each matrix column is sorted independently. This flag and the previous one are mutually exclusive
- **CV_SORT_ASCENDING** Each matrix row is sorted in the ascending order
- **CV_SORT_DESCENDING** Each matrix row is sorted in the descending order. This flag and the previous one are also mutually exclusive

The function `sortIdx` sorts each matrix row or each matrix column in ascending or descending order. Instead of reordering the elements themselves, it stores the indices of sorted elements in the destination array. For example:

```
Mat A = Mat::eye(3,3,CV_32F), B;
sortIdx(A, B, CV_SORT_EVERY_ROW + CV_SORT_ASCENDING);
// B will probably contain
// (because of equal elements in A some permutations are possible):
// [[1, 2, 0], [0, 2, 1], [0, 1, 2]]
```

See also: cv::sort, cv::randShuffle

---

cv::split (view/add comments)

Divides multi-channel array into several single-channel arrays

```
void split(const Mat& mtx, Mat* mv);
void split(const Mat& mtx, vector<Mat>& mv);
void split(const MatND& mtx, MatND* mv);
void split(const MatND& mtx, vector<MatND>& mv);
```

mtx The source multi-channel array

mv The destination array or vector of arrays; The number of arrays must match `mtx.channels()`. The arrays themselves will be reallocated if needed

The functions `split` split multi-channel array into separate single-channel arrays:

$$\texttt{mv}[c](I) = \texttt{mtx}(I)_c$$

If you need to extract a single-channel or do some other sophisticated channel permutation, use cv::mixChannels

See also: cv::merge, cv::mixChannels, cv::cvtColor

---

cv::sqrt (view/add comments)

Calculates square root of array elements

```
void sqrt(const Mat& src, Mat& dst);
void sqrt(const MatND& src, MatND& dst);
```

src The source floating-point array

dst The destination array; will have the same size and the same type as `src`

The functions `sqrt` calculate square root of each source array element. in the case of multi-channel arrays each channel is processed independently. The function accuracy is approximately the same as of the built-in `std::sqrt`.

See also: cv::pow, cv::magnitude

---

cv::subtract (view/add comments)

Calculates per-element difference between two arrays or array and a scalar

```
void subtract(const Mat& src1, const Mat& src2, Mat& dst);
void subtract(const Mat& src1, const Mat& src2,
              Mat& dst, const Mat& mask);
void subtract(const Mat& src1, const Scalar& sc,
              Mat& dst, const Mat& mask=Mat());
void subtract(const Scalar& sc, const Mat& src2,
              Mat& dst, const Mat& mask=Mat());
void subtract(const MatND& src1, const MatND& src2, MatND& dst);
void subtract(const MatND& src1, const MatND& src2,
              MatND& dst, const MatND& mask);
void subtract(const MatND& src1, const Scalar& sc,
              MatND& dst, const MatND& mask=MatND());
void subtract(const Scalar& sc, const MatND& src2,
              MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the first or the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `subtract` compute

- the difference between two arrays

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) - \texttt{src2}(I)) \quad \texttt{if mask}(I) \neq 0$$

- the difference between array and a scalar:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) - \texttt{sc}) \quad \texttt{if mask}(I) \neq 0$$

- the difference between scalar and an array:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{sc} - \texttt{src2}(I)) \quad \texttt{if mask}(I) \neq 0$$

where `I` is multi-dimensional index of array elements.
The first function in the above list can be replaced with matrix expressions:

```
dst = src1 - src2;
dst -= src2; // equivalent to subtract(dst, src2, dst);
```

See also: cv::add, cv::addWeighted, cv::scaleAdd, cv::convertScale, Matrix Expressions , saturate_cast.

---

cv::SVD (view/add comments)

Class for computing Singular Value Decomposition

```
class SVD
{
public:
    enum { MODIFY_A=1, NO_UV=2, FULL_UV=4 };
    // default empty constructor
    SVD();
    // decomposes A into u, w and vt: A = u*w*vt;
    // u and vt are orthogonal, w is diagonal
    SVD( const Mat& A, int flags=0 );
    // decomposes A into u, w and vt.
    SVD& operator ()( const Mat& A, int flags=0 );
```

```
    // finds such vector x, norm(x)=1, so that A*x = 0,
    // where A is singular matrix
    static void solveZ( const Mat& A, Mat& x );
    // does back-subsitution:
    // x = vt.t()*inv(w)*u.t()*rhs ~ inv(A)*rhs
    void backSubst( const Mat& rhs, Mat& x ) const;

    Mat u; // the left orthogonal matrix
    Mat w; // vector of singular values
    Mat vt; // the right orthogonal matrix
};
```

The class SVD is used to compute Singular Value Decomposition of a floating-point matrix and then use it to solve least-square problems, under-determined linear systems, invert matrices, compute condition numbers etc. For a bit faster operation you can pass flags=SVD::MODIFY_A|... to modify the decomposed matrix when it is not necessarily to preserve it. If you want to compute condition number of a matrix or absolute value of its determinant - you do not need u and vt, so you can pass flags=SVD::NO_UV|.... Another flag FULL_UV indicates that full-size u and vt must be computed, which is not necessary most of the time.

See also: cv::invert, cv::solve, cv::eigen, cv::determinant

---

cv::SVD::SVD (view/add comments)

SVD constructors

```
SVD::SVD();
SVD::SVD( const Mat& A, int flags=0 );
```

A The decomposed matrix

flags Operation flags

- **SVD::MODIFY_A** The algorithm can modify the decomposed matrix. It can save some space and speed-up processing a bit
- **SVD::NO_UV** Indicates that only the vector of singular values w is to be computed, while u and vt will be set to empty matrices
- **SVD::FULL_UV** When the matrix is not square, by default the algorithm produces u and vt matrices of sufficiently large size for the further A reconstruction. If, however, FULL_UV flag is specified, u and vt will be full-size square orthogonal matrices.

The first constructor initializes empty SVD structure. The second constructor initializes empty SVD structure and then calls cv::SVD::operator ().

cv::SVD::operator () (view/add comments)

Performs SVD of a matrix

```
SVD& SVD::operator ()( const Mat& A, int flags=0 );
```

A The decomposed matrix

flags Operation flags

- **SVD::MODIFY_A** The algorithm can modify the decomposed matrix. It can save some space and speed-up processing a bit
- **SVD::NO_UV** Only singular values are needed. The algorithm will not compute u and vt matrices
- **SVD::FULL_UV** When the matrix is not square, by default the algorithm produces u and vt matrices of sufficiently large size for the further A reconstruction. If, however, FULL_UV flag is specified, u and vt will be full-size square orthogonal matrices.

The operator performs singular value decomposition of the supplied matrix. The u, vt and the vector of singular values w are stored in the structure. The same SVD structure can be reused many times with different matrices. Each time, if needed, the previous u, vt and w are reclaimed and the new matrices are created, which is all handled by cv::Mat::create.

cv::SVD::solveZ (view/add comments)

Solves under-determined singular linear system

```
static void SVD::solveZ( const Mat& A, Mat& x );
```

A The left-hand-side matrix.

x The found solution

The method finds unit-length solution **x** of the under-determined system $Ax = 0$. Theory says that such system has infinite number of solutions, so the algorithm finds the unit-length solution as the right singular vector corresponding to the smallest singular value (which should be 0). In practice, because of round errors and limited floating-point accuracy, the input matrix can appear to be close-to-singular rather than just singular. So, strictly speaking, the algorithm solves the following problem:

$$x^* = \arg \min_{x:\|x\|=1} \|A \cdot x\|$$

---

cv::SVD::backSubst (view/add comments)

Performs singular value back substitution

```
void SVD::backSubst( const Mat& rhs, Mat& x ) const;
```

rhs The right-hand side of a linear system `Ax` = `rhs` being solved, where `A` is the matrix passed to cv::SVD::SVD or cv::SVD::operator ()

x The found solution of the system

The method computes back substitution for the specified right-hand side:

$$\mathtt{x} = \mathtt{vt}^T \cdot diag(\mathtt{w})^{-1} \cdot \mathtt{u}^T \cdot \mathtt{rhs} \sim \mathtt{A}^{-1} \cdot \mathtt{rhs}$$

Using this technique you can either get a very accurate solution of convenient linear system, or the best (in the least-squares terms) pseudo-solution of an overdetermined linear system. Note that explicit SVD with the further back substitution only makes sense if you need to solve many linear systems with the same left-hand side (e.g. `A`). If all you need is to solve a single system (possibly with multiple `rhs` immediately available), simply call cv::solve add pass `cv::DECOMP_SVD` there - it will do absolutely the same thing.

---

cv::sum (view/add comments)

Calculates sum of array elements

```
Scalar sum(const Mat& mtx);
Scalar sum(const MatND& mtx);
```

mtx The source array; must have 1 to 4 channels

The functions `sum` calculate and return the sum of array elements, independently for each channel.

See also: cv::countNonZero, cv::mean, cv::meanStdDev, cv::norm, cv::minMaxLoc, cv::reduce

---

cv::theRNG (view/add comments)

Returns the default random number generator

```
RNG& theRNG();
```

The function `theRNG` returns the default random number generator. For each thread there is separate random number generator, so you can use the function safely in multi-thread environments. If you just need to get a single random number using this generator or initialize an array, you can use cv::randu or cv::randn instead. But if you are going to generate many random numbers inside a loop, it will be much faster to use this function to retrieve the generator and then use `RNG::operator _Tp()`.

See also: cv::RNG, cv::randu, cv::randn

---

cv::trace (view/add comments)

Returns the trace of a matrix

```
Scalar trace(const Mat& mtx);
```

mtx The source matrix

The function `trace` returns the sum of the diagonal elements of the matrix `mtx`.

$$\mathrm{tr}(\mathtt{mtx}) = \sum_i \mathtt{mtx}(i,i)$$

---

cv::transform (view/add comments)

Performs matrix transformation of every array element.

```
void transform(const Mat& src,
               Mat& dst, const Mat& mtx );
```

src The source array; must have as many channels (1 to 4) as `mtx.cols` or `mtx.cols-1`

dst The destination array; will have the same size and depth as `src` and as many channels as `mtx.rows`

mtx The transformation matrix

The function `transform` performs matrix transformation of every element of array `src` and stores the results in `dst`:

$$\mathtt{dst}(I) = \mathtt{mtx} \cdot \mathtt{src}(I)$$

(when `mtx.cols=src.channels()`), or

$$\mathtt{dst}(I) = \mathtt{mtx} \cdot [\mathtt{src}(I); 1]$$

(when `mtx.cols=src.channels()+1`)

That is, every element of an `N`-channel array `src` is considered as `N`-element vector, which is transformed using a `M` × `N` or `M` × `N+1` matrix `mtx` into an element of `M`-channel array `dst`.

The function may be used for geometrical transformation of N-dimensional points, arbitrary linear color space transformation (such as various kinds of RGB→YUV transforms), shuffling the image channels and so forth.

See also: cv::perspectiveTransform, cv::getAffineTransform, cv::estimateRigidTransform, cv::warpAffine, cv::warpPerspective

cv::transpose (view/add comments)

Transposes a matrix

```
void transpose(const Mat& src, Mat& dst);
```

src The source array

dst The destination array of the same type as src

The function cv::transpose transposes the matrix src:

$$\texttt{dst}(i,j) = \texttt{src}(j,i)$$

Note that no complex conjugation is done in the case of a complex matrix, it should be done separately if needed.

11.3 Dynamic Structures

11.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with CV_RGB or the Scalar constructor) for color images and brightness for grayscale images. For color images the order channel is normally *Blue, Green, Red*, this is what cv::imshow, cv::imread and cv::imwrite expect , so if you form a color using Scalar constructor, it should look like:

$$\texttt{Scalar}(blue_component, green_component, red_component[, alpha_component])$$

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using cv::cvtColor.

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the shift parameter and the real point coordinates are calculated as

`Point`$(x, y) \rightarrow$ `Point2f`$(x*2^{-shift}, y*2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

---

cv::circle (view/add comments)

Draws a circle

```
void circle(Mat& img, Point center, int radius,
            const Scalar& color, int thickness=1,
            int lineType=8, int shift=0);
```

`img` Image where the circle is drawn

`center` Center of the circle

`radius` Radius of the circle

`color` Circle color

`thickness` Thickness of the circle outline if positive; negative thickness means that a filled circle is to be drawn

`lineType` Type of the circle boundary, see cv::line description

`shift` Number of fractional bits in the center coordinates and radius value

The function `circle` draws a simple or filled circle with a given center and radius.

---

cv::clipLine (view/add comments)

Clips the line against the image rectangle

```
bool clipLine(Size imgSize, Point& pt1, Point& pt2);
bool clipLine(Rect imgRect, Point& pt1, Point& pt2);
```

imgSize The image size; the image rectangle will be `Rect(0, 0, imgSize.width, imgSize.height)`

imgSize The image rectangle

pt1 The first line point

pt2 The second line point

The functions `clipLine` calculate a part of the line segment which is entirely within the specified rectangle. They return `false` if the line segment is completely outside the rectangle and `true` otherwise.

---

cv::ellipse (view/add comments)

Draws a simple or thick elliptic arc or an fills ellipse sector.

```
void ellipse(Mat& img, Point center, Size axes,
             double angle, double startAngle, double endAngle,
             const Scalar& color, int thickness=1,
             int lineType=8, int shift=0);
void ellipse(Mat& img, const RotatedRect& box, const Scalar& color,
             int thickness=1, int lineType=8);
```

img The image

center Center of the ellipse

axes Length of the ellipse axes

angle The ellipse rotation angle in degrees

startAngle Starting angle of the elliptic arc in degrees

endAngle Ending angle of the elliptic arc in degrees

box Alternative ellipse representation via a RotatedRect , i.e. the function draws an ellipse inscribed in the rotated rectangle

color Ellipse color

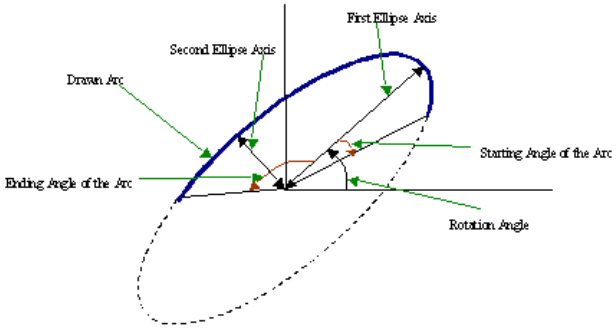
thickness Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn

lineType Type of the ellipse boundary, see cv::line description

shift Number of fractional bits in the center coordinates and axes' values

The functions `ellipse` with less parameters draw an ellipse outline, a filled ellipse, an elliptic arc or a filled ellipse sector. A piecewise-linear curve is used to approximate the elliptic arc boundary. If you need more control of the ellipse rendering, you can retrieve the curve using cv::ellipse2Poly and then render it with cv::polylines or fill it with cv::fillPoly. If you use the first variant of the function and want to draw the whole ellipse, not an arc, pass `startAngle=0` and `endAngle=360`. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



---

cv::ellipse2Poly (view/add comments)

Approximates an elliptic arc with a polyline

```
void ellipse2Poly( Point center, Size axes, int angle,
           int startAngle, int endAngle, int delta,
           vector<Point>& pts );
```

center Center of the arc

axes Half-sizes of the arc. See cv::ellipse

angle Rotation angle of the ellipse in degrees. See cv::ellipse

startAngle Starting angle of the elliptic arc in degrees

endAngle Ending angle of the elliptic arc in degrees

delta Angle between the subsequent polyline vertices. It defines the approximation accuracy.

pts The output vector of polyline vertices

The function `ellipse2Poly` computes the vertices of a polyline that approximates the specified elliptic arc. It is used by cv::ellipse.

---

cv::fillConvexPoly (view/add comments)

Fills a convex polygon.

```
void fillConvexPoly(Mat& img, const Point* pts, int npts,
           const Scalar& color, int lineType=8,
           int shift=0);
```

img Image

pts The polygon vertices

npts The number of polygon vertices

color Polygon color

lineType Type of the polygon boundaries, see cv::line description

shift The number of fractional bits in the vertex coordinates

The function `fillConvexPoly` draws a filled convex polygon. This function is much faster than the function `fillPoly` and can fill not only convex polygons but any monotonic polygon without self-intersections, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most (though, its top-most and/or the bottom edge could be horizontal).

cv::fillPoly (view/add comments)

Fills the area bounded by one or more polygons

```
void fillPoly(Mat& img, const Point** pts,
              const int* npts, int ncontours,
              const Scalar& color, int lineType=8,
              int shift=0, Point offset=Point() );
```

img Image

pts Array of polygons, each represented as an array of points

npts The array of polygon vertex counters

ncontours The number of contours that bind the filled region

color Polygon color

lineType Type of the polygon boundaries, see cv::line description

shift The number of fractional bits in the vertex coordinates

The function `fillPoly` fills an area bounded by several polygonal contours. The function can fills complex areas, for example, areas with holes, contours with self-intersections (some of thier parts), and so forth.

cv::getTextSize (view/add comments)

Calculates the width and height of a text string.

```
Size getTextSize(const string& text, int fontFace,
              double fontScale, int thickness,
              int* baseLine);
```

text The input text string

fontFace The font to use; see cv::putText

fontScale The font scale; see cv::putText

thickness The thickness of lines used to render the text; see cv::putText

baseLine The output parameter - y-coordinate of the baseline relative to the bottom-most text point

The function `getTextSize` calculates and returns size of the box that contain the specified text. That is, the following code will render some text, the tight box surrounding it and the baseline:

```
// Use "y" to show that the baseLine is about
string text = "Funny text inside the box";
int fontFace = FONT_HERSHEY_SCRIPT_SIMPLEX;
double fontScale = 2;
int thickness = 3;

Mat img(600, 800, CV_8UC3, Scalar::all(0));

int baseline=0;
Size textSize = getTextSize(text, fontFace,
                            fontScale, thickness, &baseline);
baseline += thickness;

// center the text
Point textOrg((img.cols - textSize.width)/2,
              (img.rows + textSize.height)/2);

// draw the box
rectangle(img, textOrg + Point(0, baseline),
          textOrg + Point(textSize.width, -textSize.height),
          Scalar(0,0,255));
// ... and the baseline first
line(img, textOrg + Point(0, thickness),
     textOrg + Point(textSize.width, thickness),
     Scalar(0, 0, 255));

// then put the text itself
putText(img, text, textOrg, fontFace, fontScale,
        Scalar::all(255), thickness, 8);
```

---

cv::line (view/add comments)

Draws a line segment connecting two points

```
void line(Mat& img, Point pt1, Point pt2, const Scalar& color,
          int thickness=1, int lineType=8, int shift=0);
```

img The image

pt1 First point of the line segment

pt2 Second point of the line segment

color Line color

thickness Line thickness

lineType Type of the line:

- **8** (or omitted) 8-connected line.
- **4** 4-connected line.
- **CV_AA** antialiased line.

shift Number of fractional bits in the point coordinates

The function `line` draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image boundaries. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB(r, g, b)`.

---

cv::LineIterator (view/add comments)

Class for iterating pixels on a raster line

```
class LineIterator
{
public:
    // creates iterators for the line connecting pt1 and pt2
    // the line will be clipped on the image boundaries
    // the line is 8-connected or 4-connected
    // If leftToRight=true, then the iteration is always done
    // from the left-most point to the right most,
```

```
    // not to depend on the ordering of pt1 and pt2 parameters
    LineIterator(const Mat& img, Point pt1, Point pt2,
                 int connectivity=8, bool leftToRight=false);
    // returns pointer to the current line pixel
    uchar* operator *();
    // move the iterator to the next pixel
    LineIterator& operator ++();
    LineIterator operator ++(int);

    // internal state of the iterator
    uchar* ptr;
    int err, count;
    int minusDelta, plusDelta;
    int minusStep, plusStep;
};
```

The class `LineIterator` is used to get each pixel of a raster line. It can be treated as versatile implementation of the Bresenham algorithm, where you can stop at each pixel and do some extra processing, for example, grab pixel values along the line, or draw a line with some effect (e.g. with XOR operation).

The number of pixels along the line is store in `LineIterator::count`.

```
// grabs pixels along the line (pt1, pt2)
// from 8-bit 3-channel image to the buffer
LineIterator it(img, pt1, pt2, 8);
vector<Vec3b> buf(it.count);

for(int i = 0; i < it.count; i++, ++it)
    buf[i] = *(const Vec3b)*it;
```

---

cv::rectangle (view/add comments)

Draws a simple, thick, or filled up-right rectangle.

```
void rectangle(Mat& img, Point pt1, Point pt2,
           const Scalar& color, int thickness=1,
           int lineType=8, int shift=0);
```

img Image

pt1 One of the rectangle's vertices

pt2 Opposite to `pt1` rectangle vertex

color Rectangle color or brightness (grayscale image)

thickness Thickness of lines that make up the rectangle. Negative values, e.g. `CV_FILLED`, mean that the function has to draw a filled rectangle.

lineType Type of the line, see cv::line description

shift Number of fractional bits in the point coordinates

The function `rectangle` draws a rectangle outline or a filled rectangle, which two opposite corners are `pt1` and `pt2`.

---

cv::polylines (view/add comments)

Draws several polygonal curves

```
void polylines(Mat& img, const Point** pts, const int* npts,
           int ncontours, bool isClosed, const Scalar& color,
           int thickness=1, int lineType=8, int shift=0 );
```

img The image

pts Array of polygonal curves

npts Array of polygon vertex counters

ncontours The number of curves

isClosed Indicates whether the drawn polylines are closed or not. If they are closed, the function draws the line from the last vertex of each curve to its first vertex

color Polyline color

thickness Thickness of the polyline edges

lineType Type of the line segments, see cv::line description

shift The number of fractional bits in the vertex coordinates

The function `polylines` draws one or more polygonal curves.

cv::putText (view/add comments)

Draws a text string

```
void putText( Mat& img, const string& text, Point org,
            int fontFace, double fontScale, Scalar color,
            int thickness=1, int lineType=8,
            bool bottomLeftOrigin=false );
```

img The image

text The text string to be drawn

org The bottom-left corner of the text string in the image

fontFace The font type, one of FONT_HERSHEY_SIMPLEX, FONT_HERSHEY_PLAIN, FONT_HERSHEY_DUPLEX, FONT_HERSHEY_COMPLEX, FONT_HERSHEY_TRIPLEX, FONT_HERSHEY_COMPLEX_SMALL, FONT_HERSHEY_S or FONT_HERSHEY_SCRIPT_COMPLEX, where each of the font id's can be combined with FONT_HERSHEY_ITALIC to get the slanted letters.

fontScale The font scale factor that is multiplied by the font-specific base size

color The text color

thickness Thickness of the lines used to draw the text

lineType The line type; see line for details

bottomLeftOrigin When true, the image data origin is at the bottom-left corner, otherwise it's at the top-left corner

The function putText renders the specified text string in the image. Symbols that can not be rendered using the specified font are replaced by question marks. See cv::getTextSize for a text rendering code example.

11.5 XML/YAML Persistence

cv::FileStorage (view/add comments)

The XML/YAML file storage class

```
class FileStorage
{
public:
    enum { READ=0, WRITE=1, APPEND=2 };
    enum { UNDEFINED=0, VALUE_EXPECTED=1, NAME_EXPECTED=2, INSIDE_MAP=4 };
    // the default constructor
    FileStorage();
    // the constructor that opens the file for reading
    // (flags=FileStorage::READ) or writing (flags=FileStorage::WRITE)
    FileStorage(const string& filename, int flags);
    // wraps the already opened CvFileStorage*
    FileStorage(CvFileStorage* fs);
    // the destructor; closes the file if needed
    virtual ~FileStorage();

    // opens the specified file for reading (flags=FileStorage::READ)
    // or writing (flags=FileStorage::WRITE)
    virtual bool open(const string& filename, int flags);
    // checks if the storage is opened
    virtual bool isOpened() const;
    // closes the file
    virtual void release();

    // returns the first top-level node
    FileNode getFirstTopLevelNode() const;
    // returns the root file node
    // (it's the parent of the first top-level node)
    FileNode root(int streamidx=0) const;
    // returns the top-level node by name
    FileNode operator[](const string& nodename) const;
    FileNode operator[](const char* nodename) const;

    // returns the underlying CvFileStorage*
    CvFileStorage* operator *() { return fs; }
    const CvFileStorage* operator *() const { return fs; }

    // writes the certain number of elements of the specified format
    // (see DataType) without any headers
    void writeRaw( const string& fmt, const uchar* vec, size_t len );

    // writes an old-style object (CvMat, CvMatND etc.)
    void writeObj( const string& name, const void* obj );

    // returns the default object name from the filename
    // (used by cvSave() with the default object name etc.)
```

```
    static string getDefaultObjectName(const string& filename);

    Ptr<CvFileStorage> fs;
    string elname;
    vector<char> structs;
    int state;
};
```

---

cv::FileNode (view/add comments)

The XML/YAML file node class

```
class CV_EXPORTS FileNode
{
public:
    enum { NONE=0, INT=1, REAL=2, FLOAT=REAL, STR=3,
        STRING=STR, REF=4, SEQ=5, MAP=6, TYPE_MASK=7,
        FLOW=8, USER=16, EMPTY=32, NAMED=64 };
    FileNode();
    FileNode(const CvFileStorage* fs, const CvFileNode* node);
    FileNode(const FileNode& node);
    FileNode operator[](const string& nodename) const;
    FileNode operator[](const char* nodename) const;
    FileNode operator[](int i) const;
    int type() const;
    int rawDataSize(const string& fmt) const;
    bool empty() const;
    bool isNone() const;
    bool isSeq() const;
    bool isMap() const;
    bool isInt() const;
    bool isReal() const;
    bool isString() const;
    bool isNamed() const;
    string name() const;
    size_t size() const;
    operator int() const;
    operator float() const;
    operator double() const;
    operator string() const;

    FileNodeIterator begin() const;
    FileNodeIterator end() const;
```

```
    void readRaw( const string& fmt, uchar* vec, size_t len ) const;
    void* readObj() const;

    // do not use wrapper pointer classes for better efficiency
    const CvFileStorage* fs;
    const CvFileNode* node;
};
```

cv::FileNodeIterator (view/add comments)

The XML/YAML file node iterator class

```
class CV_EXPORTS FileNodeIterator
{
public:
    FileNodeIterator();
    FileNodeIterator(const CvFileStorage* fs,
        const CvFileNode* node, size_t ofs=0);
    FileNodeIterator(const FileNodeIterator& it);
    FileNode operator *() const;
    FileNode operator ->() const;

    FileNodeIterator& operator ++();
    FileNodeIterator operator ++(int);
    FileNodeIterator& operator --();
    FileNodeIterator operator --(int);
    FileNodeIterator& operator += (int);
    FileNodeIterator& operator -= (int);

    FileNodeIterator& readRaw( const string& fmt, uchar* vec,
                               size_t maxCount=(size_t)INT_MAX );

    const CvFileStorage* fs;
    const CvFileNode* container;
    CvSeqReader reader;
    size_t remaining;
};
```

11.6 Clustering

cv::kmeans (view/add comments)

Finds the centers of clusters and groups the input samples around the clusters.

```
double kmeans( const Mat& samples, int clusterCount, Mat& labels,
           TermCriteria termcrit, int attempts,
           int flags, Mat* centers );
```

samples Floating-point matrix of input samples, one row per sample

clusterCount The number of clusters to split the set by

labels The input/output integer array that will store the cluster indices for every sample

termcrit Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

attempts How many times the algorithm is executed using different initial labelings. The algorithm returns the labels that yield the best compactness (see the last function parameter)

flags It can take the following values:

KMEANS_RANDOM_CENTERS Random initial centers are selected in each attempt

KMEANS_PP_CENTERS Use kmeans++ center initialization by Arthur and Vassilvitskii

KMEANS_USE_INITIAL_LABELS During the first (and possibly the only) attempt, the function uses the user-supplied labels instaed of computing them from the initial centers. For the second and further attempts, the function will use the random or semi-random centers (use one of `KMEANS_*_CENTERS` flag to specify the exact method)

centers The output matrix of the cluster centers, one row per each cluster center

The function `kmeans` implements a k-means algorithm that finds the centers of `clusterCount` clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a 0-based cluster index for the sample stored in the i^{th} row of the `samples` matrix.

The function returns the compactness measure, which is computed as

$$\sum_i \|\texttt{samples}_i - \texttt{centers}_{\texttt{labels}_i}\|^2$$

after every attempt; the best (minimum) value is chosen and the corresponding labels and the compactness value are returned by the function. Basically, the user can use only the core of the function, set the number of attempts to 1, initialize labels each time using some custom algorithm and pass them with

(`flags=KMEANS_USE_INITIAL_LABELS`) flag, and then choose the best (most-compact) clustering.

cv::partition (view/add comments)

Splits an element set into equivalency classes.

```
template<typename _Tp, class _EqPredicate> int
partition( const vector<_Tp>& vec, vector<int>& labels,
           _EqPredicate predicate=_EqPredicate());
```

vec The set of elements stored as a vector

labels The output vector of labels; will contain as many elements as `vec`. Each label `labels[i]` is 0-based cluster index of `vec[i]`

predicate The equivalence predicate (i.e. pointer to a boolean function of two arguments or an instance of the class that has the method `bool operator()(const _Tp& a, const _Tp& b)`. The predicate returns true when the elements are certainly if the same class, and false if they may or may not be in the same class

The generic function `partition` implements an $O(N^2)$ algorithm for splitting a set of N elements into one or more equivalency classes, as described in http://en.wikipedia.org/wiki/Disjoint-set_data_structure. The function returns the number of equivalency classes.

11.7 Utility and System Functions and Macros

cv::alignPtr (view/add comments)

Aligns pointer to the specified number of bytes

```
template<typename _Tp> _Tp* alignPtr(_Tp* ptr, int n=sizeof(_Tp));
```

ptr The aligned pointer

n The alignment size; must be a power of two

The function returns the aligned pointer of the same type as the input pointer:

```
(_Tp*)(((size_t)ptr + n-1) & -n)
```

cv::alignSize (view/add comments)

Aligns a buffer size to the specified number of bytes

```
size_t alignSize(size_t sz, int n);
```

sz The buffer size to align

n The alignment size; must be a power of two

The function returns the minimum number that is greater or equal to `sz` and is divisble by `n`:

```
(sz + n-1) & -n
```

cv::allocate (view/add comments)

Allocates an array of elements

```
template<typename _Tp> _Tp* allocate(size_t n);
```

n The number of elements to allocate

The generic function `allocate` allocates buffer for the specified number of elements. For each element the default constructor is called.

cv::deallocate (view/add comments)

Allocates an array of elements

```
template<typename _Tp> void deallocate(_Tp* ptr, size_t n);
```

ptr Pointer to the deallocated buffer

n The number of elements in the buffer

The generic function `deallocate` deallocates the buffer allocated with cv::allocate. The number of elements must match the number passed to cv::allocate.

CV_Assert (view/add comments)

Checks a condition at runtime.

```
CV_Assert(expr)
```

```
#define CV_Assert( expr ) ...
#define CV_DbgAssert(expr) ...
```

expr The checked expression

The macros `CV_Assert` and `CV_DbgAssert` evaluate the specified expression and if it is 0, the macros raise an error (see cv::error). The macro `CV_Assert` checks the condition in both Debug and Release configurations, while `CV_DbgAssert` is only retained in the Debug configuration.

cv::error (view/add comments)

Signals an error and raises the exception

```
void error( const Exception& exc );
#define CV_Error( code, msg ) <...>
#define CV_Error_( code, args ) <...>
```

exc The exception to throw

code The error code, normally, a negative value. The list of pre-defined error codes can be found in `cxerror.h`

msg Text of the error message

args printf-like formatted error message in parantheses

The function and the helper macros `CV_Error` and `CV_Error_` call the error handler. Currently, the error handler prints the error code (`exc.code`), the context (`exc.file`, `exc.line` and the error message `exc.err` to the standard error stream `stderr`. In Debug configuration it then

provokes memory access violation, so that the execution stack and all the parameters can be analyzed in debugger. In Release configuration the exception `exc` is thrown.

The macro `CV_Error_` can be used to construct the error message on-fly to include some dynamic information, for example:

```
// note the extra parentheses around the formatted text message
CV_Error_(CV_StsOutOfRange,
    ("the matrix element (%d,%d)=%g is out of range",
    i, j, mtx.at<float>(i,j)))
```

---

cv::Exception (view/add comments)

The exception class passed to error

```
class  Exception
{
public:
    // various constructors and the copy operation
    Exception() { code = 0; line = 0; }
    Exception(int _code, const string& _err,
              const string& _func, const string& _file, int _line);
    Exception(const Exception& exc);
    Exception& operator = (const Exception& exc);

    // the error code
    int code;
    // the error text message
    string err;
    // function name where the error happened
    string func;
    // the source file name where the error happened
    string file;
    // the source file line where the error happened
    int line;
};
```

The class `Exception` encapsulates all or almost all the necessary information about the error happened in the program. The exception is usually constructed and thrown implicitly, via `CV_Error` and `CV_Error_` macros, see cv::error.

---

cv::fastMalloc (view/add comments)

Allocates aligned memory buffer

```
void* fastMalloc(size_t size);
```

size The allocated buffer size

The function allocates buffer of the specified size and returns it. When the buffer size is 16 bytes or more, the returned buffer is aligned on 16 bytes.

---

cv::fastFree (view/add comments)

Deallocates memory buffer

```
void fastFree(void* ptr);
```

ptr Pointer to the allocated buffer

The function deallocates the buffer, allocated with cv::fastMalloc. If NULL pointer is passed, the function does nothing.

---

cv::format (view/add comments)

Returns a text string formatted using printf-like expression

```
string format( const char* fmt, ... );
```

fmt The printf-compatible formatting specifiers

The function acts like `sprintf`, but forms and returns STL string. It can be used for form the error message in cv::Exception constructor.

cv::getNumThreads (view/add comments)

Returns the number of threads used by OpenCV

```
int getNumThreads();
```

The function returns the number of threads that is used by OpenCV.
See also: cv::setNumThreads, cv::getThreadNum.

cv::getThreadNum (view/add comments)

Returns index of the currently executed thread

```
int getThreadNum();
```

The function returns 0-based index of the currently executed thread. The function is only valid inside a parallel OpenMP region. When OpenCV is built without OpenMP support, the function always returns 0.
See also: cv::setNumThreads, cv::getNumThreads.

cv::getTickCount (view/add comments)

Returns the number of ticks

```
int64 getTickCount();
```

The function returns the number of ticks since the certain event (e.g. when the machine was turned on). It can be used to initialize cv::RNG or to measure a function execution time by reading the tick count before and after the function call. See also the tick frequency.

cv::getTickFrequency (view/add comments)

Returns the number of ticks per second

```
double getTickFrequency();
```

The function returns the number of ticks per second. That is, the following code computes the execution time in seconds.

```
double t = (double)getTickCount();
// do something ...
t = ((double)getTickCount() - t)/getTickFrequency();
```

cv::setNumThreads (view/add comments)

Sets the number of threads used by OpenCV

```
void setNumThreads(int nthreads);
```

nthreads The number of threads used by OpenCV

The function sets the number of threads used by OpenCV in parallel OpenMP regions. If `nthreads=0`, the function will use the default number of threads, which is usually equal to the number of the processing cores.

See also: cv::getNumThreads, cv::getThreadNum

Chapter 12

imgproc. Image Processing

12.1 Histograms

cv::calcHist (view/add comments)

Calculates histogram of a set of arrays

```
void calcHist( const Mat* arrays, int narrays,
           const int* channels, const Mat& mask,
           MatND& hist, int dims, const int* histSize,
           const float** ranges, bool uniform=true,
           bool accumulate=false );
void calcHist( const Mat* arrays, int narrays,
           const int* channels, const Mat& mask,
           SparseMat& hist, int dims, const int* histSize,
           const float** ranges, bool uniform=true,
           bool accumulate=false );
```

arrays Source arrays. They all should have the same depth, `CV_8U` or `CV_32F`, and the same size. Each of them can have an arbitrary number of channels

narrays The number of source arrays

channels The list of `dims` channels that are used to compute the histogram. The first array channels are numerated from 0 to `arrays[0].channels()-1`, the second array channels are

counted from `arrays[0].channels()` to `arrays[0].channels() + arrays[1].channels()-1` etc.

mask The optional mask. If the matrix is not empty, it must be 8-bit array of the same size as `arrays[i]`. The non-zero mask elements mark the array elements that are counted in the histogram

hist The output histogram, a dense or sparse `dims`-dimensional array

dims The histogram dimensionality; must be positive and not greater than `CV_MAX_DIMS`(=32 in the current OpenCV version)

histSize The array of histogram sizes in each dimension

ranges The array of `dims` arrays of the histogram bin boundaries in each dimension. When the histogram is uniform (`uniform`=true), then for each dimension `i` it's enough to specify the lower (inclusive) boundary L_0 of the 0-th histogram bin and the upper (exclusive) boundary $U_{\texttt{histSize}[i]-1}$ for the last histogram bin `histSize[i]-1`. That is, in the case of uniform histogram each of `ranges[i]` is an array of 2 elements. When the histogram is not uniform (`uniform=false`), then each of `ranges[i]` contains `histSize[i]+1` elements: $L_0, U_0 = L_1, U_1 = L_2, ..., U_{\texttt{histSize[i]}-2} = L_{\texttt{histSize[i]}-1}, U_{\texttt{histSize[i]}-1}$. The array elements, which are not between L_0 and $U_{\texttt{histSize[i]}-1}$, are not counted in the histogram

uniform Indicates whether the histogram is uniform or not, see above

accumulate Accumulation flag. If it is set, the histogram is not cleared in the beginning (when it is allocated). This feature allows user to compute a single histogram from several sets of arrays, or to update the histogram in time

The functions `calcHist` calculate the histogram of one or more arrays. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input arrays. The sample below shows how to compute 2D Hue-Saturation histogram for a color imag

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat src, hsv;
    if( argc != 2 || !(src=imread(argv[1], 1)).data )
        return -1;
```

```
    cvtColor(src, hsv, CV_BGR2HSV);

    // let's quantize the hue to 30 levels
    // and the saturation to 32 levels
    int hbins = 30, sbins = 32;
    int histSize[] = {hbins, sbins};
    // hue varies from 0 to 179, see cvtColor
    float hranges[] = { 0, 180 };
    // saturation varies from 0 (black-gray-white) to
    // 255 (pure spectrum color)
    float sranges[] = { 0, 256 };
    const float* ranges[] = { hranges, sranges };
    MatND hist;
    // we compute the histogram from the 0-th and 1-st channels
    int channels[] = {0, 1};

    calcHist( &hsv, 1, channels, Mat(), // do not use mask
              hist, 2, histSize, ranges,
              true, // the histogram is uniform
              false );
    double maxVal=0;
    minMaxLoc(hist, 0, &maxVal, 0, 0);

    int scale = 10;
    Mat histImg = Mat::zeros(sbins*scale, hbins*10, CV_8UC3);

    for( int h = 0; h < hbins; h++ )
        for( int s = 0; s < sbins; s++ )
        {
            float binVal = hist.at<float>(h, s);
            int intensity = cvRound(binVal*255/maxVal);
            rectangle( histImg, Point(h*scale, s*scale),
                        Point( (h+1)*scale - 1, (s+1)*scale - 1),
                        Scalar::all(intensity),
                        CV_FILLED );
        }

    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "H-S Histogram", 1 );
    imshow( "H-S Histogram", histImg );
    waitKey();
}
```

cv::calcBackProject (view/add comments)

Calculates the back projection of a histogram.

```
void calcBackProject( const Mat* arrays, int narrays,
           const int* channels, const MatND& hist,
           Mat& backProject, const float** ranges,
           double scale=1, bool uniform=true );
void calcBackProject( const Mat* arrays, int narrays,
           const int* channels, const SparseMat& hist,
           Mat& backProject, const float** ranges,
           double scale=1, bool uniform=true );
```

arrays Source arrays. They all should have the same depth, `CV_8U` or `CV_32F`, and the same size. Each of them can have an arbitrary number of channels

narrays The number of source arrays

channels The list of channels that are used to compute the back projection. The number of channels must match the histogram dimensionality. The first array channels are numerated from 0 to `arrays[0].channels()-1`, the second array channels are counted from `arrays[0].channels()` to `arrays[0].channels() + arrays[1].channels()-1` etc.

hist The input histogram, a dense or sparse

backProject Destination back projection aray; will be a single-channel array of the same size and the same depth as `arrays[0]`

ranges The array of arrays of the histogram bin boundaries in each dimension. See cv::calcHist

scale The optional scale factor for the output back projection

uniform Indicates whether the histogram is uniform or not, see above

The functions `calcBackProject` calculate the back project of the histogram. That is, similarly to `calcHist`, at each location `(x, y)` the function collects the values from the selected channels in the input images and finds the corresponding histogram bin. But instead of incrementing it, the function reads the bin value, scales it by `scale` and stores in `backProject(x,y)`. In terms of statistics, the function computes probability of each element value in respect with the empirical

probability distribution represented by the histogram. Here is how, for example, you can find and track a bright-colored object in a scene:

1. Before the tracking, show the object to the camera such that covers almost the whole frame. Calculate a hue histogram. The histogram will likely have a strong maximums, corresponding to the dominant colors in the object.
2. During the tracking, calculate back projection of a hue plane of each input video frame using that pre-computed histogram. Threshold the back projection to suppress weak colors. It may also have sense to suppress pixels with non sufficient color saturation and too dark or too bright pixels.
3. Find connected components in the resulting picture and choose, for example, the largest component.

That is the approximate algorithm of cv::CAMShift color object tracker.
See also: cv::calcHist

---

cv::compareHist (view/add comments)

Compares two histograms

```
double compareHist( const MatND& H1, const MatND& H2, int method );
double compareHist( const SparseMat& H1,
           const SparseMat& H2, int method );
```

H1 The first compared histogram

H2 The second compared histogram of the same size as `H1`

method The comparison method, one of the following:

CV_COMP_CORREL Correlation

CV_COMP_CHISQR Chi-Square

CV_COMP_INTERSECT Intersection

CV_COMP_BHATTACHARYYA Bhattacharyya distance

The functions `compareHist` compare two dense or two sparse histograms using the specified method:

Correlation (method=CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H_1})(H_2(I) - \bar{H_2})}{\sqrt{\sum_I (H_1(I) - \bar{H_1})^2 \sum_I (H_2(I) - \bar{H_2})^2}}$$

where

$$\bar{H_k} = \frac{1}{N} \sum_J H_k(J)$$

and N is the total number of histogram bins.

Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H_1} \bar{H_2} N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

The function returns $d(H_1, H_2)$.

While the function works well with 1-, 2-, 3-dimensional dense histograms, it may not be suitable for high-dimensional sparse histograms, where, because of aliasing and sampling problems the coordinates of non-zero histogram bins can slightly shift. To compare such histograms or more general sparse configurations of weighted points, consider using the cv::calcEMD function.

---

cv::equalizeHist (view/add comments)

Equalizes the histogram of a grayscale image.

```
void equalizeHist( const Mat& src, Mat& dst );
```

src The source 8-bit single channel image

dst The destination image; will have the same size and the same type as `src`

The function equalizes the histogram of the input image using the following algorithm:

1. calculate the histogram H for `src`.
2. normalize the histogram so that the sum of histogram bins is 255.
3. compute the integral of the histogram:

$$H'_i = \sum_{0 \le j < i} H(j)$$

4. transform the image using H' as a look-up table: $\texttt{dst}(x,y) = H'(\texttt{src}(x,y))$

The algorithm normalizes the brightness and increases the contrast of the image.

12.2 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as cv::Mat's), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y). It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc. OpenCV let the user to specify the extrapolation method; see the function cv::borderInterpolate and discussion of `borderType` parameter in various functions below.

cv::BaseColumnFilter (view/add comments)

Base class for filters with single-column kernels

```
class BaseColumnFilter
{
public:
    virtual ~BaseColumnFilter();

    // To be overriden by the user.
    //
    // runs filtering operation on the set of rows,
    // "dstcount + ksize - 1" rows on input,
    // "dstcount" rows on output,
    // each input and output row has "width" elements
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, int dststep,
                            int dstcount, int width) = 0;
    // resets the filter state (may be needed for IIR filters)
    virtual void reset();

    int ksize; // the aperture size
    int anchor; // position of the anchor point,
                // normally not used during the processing
};
```

The class `BaseColumnFilter` is the base class for filtering data using single-column kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\texttt{dst}(x,y) = F(\texttt{src}[y](x),\ \texttt{src}[y+1](x),\ ...,\ \texttt{src}[y+\texttt{ksize}-1](x)$$

where F is the filtering function, but, as it is represented as a class, it can produce any side effects, memorize previously processed data etc. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to cv::FilterEngine constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: cv::BaseRowFilter, cv::BaseFilter, cv::FilterEngine, cv::getColumnSumFilter, cv::getLinearColumnFilter, cv::getMorphologyColumnFilter

cv::BaseFilter (view/add comments)

Base class for 2D image filters

```
class BaseFilter
{
public:
    virtual ~BaseFilter();

    // To be overriden by the user.
    //
    // runs filtering operation on the set of rows,
    // "dstcount + ksize.height - 1" rows on input,
    // "dstcount" rows on output,
    // each input row has "(width + ksize.width-1)*cn" elements
    // each output row has "width*cn" elements.
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, int dststep,
                            int dstcount, int width, int cn) = 0;
    // resets the filter state (may be needed for IIR filters)
    virtual void reset();
    Size ksize;
    Point anchor;
};
```

The class `BaseFilter` is the base class for filtering data using 2D kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\begin{aligned}&\texttt{dst}(x,y) = F(\texttt{src}[y](x),\ \texttt{src}[y](x+1),\ ...,\ \texttt{src}[y](x+\texttt{ksize.width}-1),\\&\texttt{src}[y+1](x),\ \texttt{src}[y+1](x+1),\ ...,\ \texttt{src}[y+1](x+\texttt{ksize.width}-1),\\&.....\\&\texttt{src}[y+\texttt{ksize.height}-1](x),\\&\texttt{src}[y+\texttt{ksize.height}-1](x+1),\\&...\texttt{src}[y+\texttt{ksize.height}-1](x+\texttt{ksize.width}-1))\end{aligned}$$

where F is the filtering function. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to cv::FilterEngine constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: cv::BaseColumnFilter, cv::BaseRowFilter, cv::FilterEngine, cv::getLinearFilter, cv::getMorphologyFilter

---

cv::BaseRowFilter (view/add comments)

Base class for filters with single-row kernels

```
class BaseRowFilter
```

```
{
public:
    virtual ~BaseRowFilter();

    // To be overriden by the user.
    //
    // runs filtering operation on the single input row
    // of "width" element, each element is has "cn" channels.
    // the filtered row is written into "dst" buffer.
    virtual void operator()(const uchar* src, uchar* dst,
                            int width, int cn) = 0;
    int ksize, anchor;
};
```

The class `BaseRowFilter` is the base class for filtering data using single-row kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\texttt{dst}(x,y) = F(\texttt{src}[y](x),\ \texttt{src}[y](x+1),\ ...,\ \texttt{src}[y](x+\texttt{ksize.width}-1))$$

where F is the filtering function. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to cv::FilterEngine constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: cv::BaseColumnFilter, cv::Filter, cv::FilterEngine, cv::getLinearRowFilter, cv::getMorphologyRow cv::getRowSumFilter

---

cv::FilterEngine (view/add comments)

Generic image filtering class

```
class FilterEngine
{
public:
    // empty constructor
    FilterEngine();
    // builds a 2D non-separable filter (!_filter2D.empty()) or
    // a separable filter (!_rowFilter.empty() && !_columnFilter.empty())
    // the input data type will be "srcType", the output data type will be "dstType",
    // the intermediate data type is "bufType".
    // _rowBorderType and _columnBorderType determine how the image
    // will be extrapolated beyond the image boundaries.
    // _borderValue is only used when _rowBorderType and/or _columnBorderType
    // == cv::BORDER_CONSTANT
```

```
FilterEngine(const Ptr<BaseFilter>& _filter2D,
             const Ptr<BaseRowFilter>& _rowFilter,
             const Ptr<BaseColumnFilter>& _columnFilter,
             int srcType, int dstType, int bufType,
             int _rowBorderType=BORDER_REPLICATE,
             int _columnBorderType=-1, // use _rowBorderType by default
             const Scalar& _borderValue=Scalar());
virtual ~FilterEngine();
// separate function for the engine initialization
void init(const Ptr<BaseFilter>& _filter2D,
          const Ptr<BaseRowFilter>& _rowFilter,
          const Ptr<BaseColumnFilter>& _columnFilter,
          int srcType, int dstType, int bufType,
          int _rowBorderType=BORDER_REPLICATE, int _columnBorderType=-1,
          const Scalar& _borderValue=Scalar());
// starts filtering of the ROI in an image of size "wholeSize".
// returns the starting y-position in the source image.
virtual int start(Size wholeSize, Rect roi, int maxBufRows=-1);
// alternative form of start that takes the image
// itself instead of "wholeSize". Set isolated to true to pretend that
// there are no real pixels outside of the ROI
// (so that the pixels will be extrapolated using the specified border modes)
virtual int start(const Mat& src, const Rect& srcRoi=Rect(0,0,-1,-1),
                  bool isolated=false, int maxBufRows=-1);
// processes the next portion of the source image,
// "srcCount" rows starting from "src" and
// stores the results to "dst".
// returns the number of produced rows
virtual int proceed(const uchar* src, int srcStep, int srcCount,
                    uchar* dst, int dstStep);
// higher-level function that processes the whole
// ROI or the whole image with a single call
virtual void apply( const Mat& src, Mat& dst,
                    const Rect& srcRoi=Rect(0,0,-1,-1),
                    Point dstOfs=Point(0,0),
                    bool isolated=false);
bool isSeparable() const { return filter2D.empty(); }
// how many rows from the input image are not yet processed
int remainingInputRows() const;
// how many output rows are not yet produced
int remainingOutputRows() const;
...
// the starting and the ending rows in the source image
int startY, endY;
```

```
    // pointers to the filters
    Ptr<BaseFilter> filter2D;
    Ptr<BaseRowFilter> rowFilter;
    Ptr<BaseColumnFilter> columnFilter;
};
```

The class `FilterEngine` can be used to apply an arbitrary filtering operation to an image. It contains all the necessary intermediate buffers, it computes extrapolated values of the "virtual" pixels outside of the image etc. Pointers to the initialized `FilterEngine` instances are returned by various `create*Filter` functions, see below, and they are used inside high-level functions such as cv::filter2D, cv::erode, cv::dilate etc, that is, the class is the workhorse in many of OpenCV filtering functions.

This class makes it easier (though, maybe not very easy yet) to combine filtering operations with other operations, such as color space conversions, thresholding, arithmetic operations, etc. By combining several operations together you can get much better performance because your data will stay in cache. For example, below is the implementation of Laplace operator for a floating-point images, which is a simplified implementation of cv::Laplacian:

```
void laplace_f(const Mat& src, Mat& dst)
{
    CV_Assert( src.type() == CV_32F );
    dst.create(src.size(), src.type());

    // get the derivative and smooth kernels for d2I/dx2.
    // for d2I/dy2 we could use the same kernels, just swapped
    Mat kd, ks;
    getSobelKernels( kd, ks, 2, 0, ksize, false, ktype );

    // let's process 10 source rows at once
    int DELTA = std::min(10, src.rows);
    Ptr<FilterEngine> Fxx = createSeparableLinearFilter(src.type(),
        dst.type(), kd, ks, Point(-1,-1), 0, borderType, borderType, Scalar() );
    Ptr<FilterEngine> Fyy = createSeparableLinearFilter(src.type(),
        dst.type(), ks, kd, Point(-1,-1), 0, borderType, borderType, Scalar() );

    int y = Fxx->start(src), dsty = 0, dy = 0;
    Fyy->start(src);
    const uchar* sptr = src.data + y*src.step;

    // allocate the buffers for the spatial image derivatives;
    // the buffers need to have more than DELTA rows, because at the
    // last iteration the output may take max(kd.rows-1,ks.rows-1)
    // rows more than the input.
    Mat Ixx( DELTA + kd.rows - 1, src.cols, dst.type() );
    Mat Iyy( DELTA + kd.rows - 1, src.cols, dst.type() );
```

```
    // inside the loop we always pass DELTA rows to the filter
    // (note that the "proceed" method takes care of possibe overflow, since
    // it was given the actual image height in the "start" method)
    // on output we can get:
    //  * < DELTA rows (the initial buffer accumulation stage)
    //  * = DELTA rows (settled state in the middle)
    //  * > DELTA rows (then the input image is over, but we generate
    //                  "virtual" rows using the border mode and filter them)
    // this variable number of output rows is dy.
    // dsty is the current output row.
    // sptr is the pointer to the first input row in the portion to process
    for( ; dsty < dst.rows; sptr += DELTA*src.step, dsty += dy )
    {
        Fxx->proceed( sptr, (int)src.step, DELTA, Ixx.data, (int)Ixx.step );
        dy = Fyy->proceed( sptr, (int)src.step, DELTA, d2y.data, (int)Iyy.step );
        if( dy > 0 )
        {
            Mat dstripe = dst.rowRange(dsty, dsty + dy);
            add(Ixx.rowRange(0, dy), Iyy.rowRange(0, dy), dstripe);
        }
    }
}
```

If you do not need that much control of the filtering process, you can simply use the `FilterEngine::apply` method. Here is how the method is actually implemented:

```
void FilterEngine::apply(const Mat& src, Mat& dst,
    const Rect& srcRoi, Point dstOfs, bool isolated)
{
    // check matrix types
    CV_Assert( src.type() == srcType && dst.type() == dstType );

    // handle the "whole image" case
    Rect _srcRoi = srcRoi;
    if( _srcRoi == Rect(0,0,-1,-1) )
        _srcRoi = Rect(0,0,src.cols,src.rows);

    // check if the destination ROI is inside the dst.
    // and FilterEngine::start will check if the source ROI is inside src.
    CV_Assert( dstOfs.x >= 0 && dstOfs.y >= 0 &&
        dstOfs.x + _srcRoi.width <= dst.cols &&
        dstOfs.y + _srcRoi.height <= dst.rows );

    // start filtering
    int y = start(src, _srcRoi, isolated);
```

```
    // process the whole ROI. Note that "endY - startY" is the total number
    // of the source rows to process
    // (including the possible rows outside of srcRoi but inside the source image)
    proceed( src.data + y*src.step,
             (int)src.step, endY - startY,
             dst.data + dstOfs.y*dst.step +
             dstOfs.x*dst.elemSize(), (int)dst.step );
}
```

Unlike the earlier versions of OpenCV, now the filtering operations fully support the notion of image ROI, that is, pixels outside of the ROI but inside the image can be used in the filtering operations. For example, you can take a ROI of a single pixel and filter it - that will be a filter response at that particular pixel (however, it's possible to emulate the old behavior by passing `isolated=false` to `FilterEngine::start` or `FilterEngine::apply`). You can pass the ROI explicitly to `FilterEngine::apply`, or construct a new matrix headers:

```
// compute dI/dx derivative at src(x,y)

// method 1:
// form a matrix header for a single value
float val1 = 0;
Mat dst1(1,1,CV_32F,&val1);

Ptr<FilterEngine> Fx = createDerivFilter(CV_32F, CV_32F,
                        1, 0, 3, BORDER_REFLECT_101);
Fx->apply(src, Rect(x,y,1,1), Point(), dst1);

// method 2:
// form a matrix header for a single value
float val2 = 0;
Mat dst2(1,1,CV_32F,&val2);

Mat pix_roi(src, Rect(x,y,1,1));
Sobel(pix_roi, dst2, dst2.type(), 1, 0, 3, 1, 0, BORDER_REFLECT_101);

printf("method1 = %g, method2 = %g\n", val1, val2);
```

Note on the data types. As it was mentioned in cv::BaseFilter description, the specific filters can process data of any type, despite that `Base*Filter::operator()` only takes `uchar` pointers and no information about the actual types. To make it all work, the following rules are used:

- in case of separable filtering `FilterEngine::rowFilter` applied first. It transforms the input image data (of type `srcType`) to the intermediate results stored in the internal buffers

(of type bufType). Then these intermediate results are processed *as single-channel data* with FilterEngine::columnFilter and stored in the output image (of type dstType). Thus, the input type for rowFilter is srcType and the output type is bufType; the input type for columnFilter is CV_MAT_DEPTH(bufType) and the output type is CV_MAT_DEPTH(dstType).

- in case of non-separable filtering bufType must be the same as srcType. The source data is copied to the temporary buffer if needed and then just passed to FilterEngine::filter2D. That is, the input type for filter2D is srcType (=bufType) and the output type is dstType.

See also: cv::BaseColumnFilter, cv::BaseFilter, cv::BaseRowFilter, cv::createBoxFilter, cv::createDerivFilter, cv::createGaussianFilter, cv::createLinearFilter, cv::createMorphologyFilter, cv::createSeparableLinearFilter

---

cv::bilateralFilter (view/add comments)

Applies bilateral filter to the image

```
void bilateralFilter( const Mat& src, Mat& dst, int d,
           double sigmaColor, double sigmaSpace,
           int borderType=BORDER_DEFAULT );
```

src The source 8-bit or floating-point, 1-channel or 3-channel image

dst The destination image; will have the same size and the same type as src

d The diameter of each pixel neighborhood, that is used during filtering. If it is non-positive, it's computed from sigmaSpace

sigmaColor Filter sigma in the color space. Larger value of the parameter means that farther colors within the pixel neighborhood (see sigmaSpace) will be mixed together, resulting in larger areas of semi-equal color

sigmaSpace Filter sigma in the coordinate space. Larger value of the parameter means that farther pixels will influence each other (as long as their colors are close enough; see sigmaColor). Then d>0, it specifies the neighborhood size regardless of sigmaSpace, otherwise d is proportional to sigmaSpace

The function applies bilateral filtering to the input image, as described in http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

cv::blur (view/add comments)

Smoothes image using normalized box filter

```
void blur( const Mat& src, Mat& dst,
           Size ksize, Point anchor=Point(-1,-1),
           int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same type as `src`

ksize The smoothing kernel size

anchor The anchor point. The default value `Point(-1,-1)` means that the anchor is at the kernel center

borderType The border mode used to extrapolate pixels outside of the image

The function smoothes the image using the kernel:

$$K = \frac{1}{\texttt{ksize.width*ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \hdots & \hdots & \hdots & \hdots & \hdots & \hdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

The call `blur(src, dst, ksize, anchor, borderType)` is equivalent to `boxFilter(src, dst, src.type(), anchor, true, borderType)`.

See also: cv::boxFilter, cv::bilateralFilter, cv::GaussianBlur, cv::medianBlur.

cv::borderInterpolate (view/add comments)

Computes source location of extrapolated pixel

```
int borderInterpolate( int p, int len, int borderType );
```

p 0-based coordinate of the extrapolated pixel along one of the axes, likely ¡0 or ¿=`len`

len length of the array along the corresponding axis

borderType the border type, one of the `BORDER_*`, except for `BORDER_TRANSPARENT` and `BORDER_ISOLATED`. When `borderType==BORDER_CONSTANT` the function always returns -1, regardless of `p` and `len`

The function computes and returns the coordinate of the donor pixel, corresponding to the specified extrapolated pixel when using the specified extrapolation border mode. For example, if we use `BORDER_WRAP` mode in the horizontal direction, `BORDER_REFLECT_101` in the vertical direction and want to compute value of the "virtual" pixel `Point(-5, 100)` in a floating-point image `img`, it will be

```
float val = img.at<float>(borderInterpolate(100, img.rows, BORDER_REFLECT_101),
                          borderInterpolate(-5, img.cols, BORDER_WRAP));
```

Normally, the function is not called directly; it is used inside cv::FilterEngine and cv::copyMakeBorder to compute tables for quick extrapolation.

See also: cv::FilterEngine, cv::copyMakeBorder

---

cv::boxFilter (view/add comments)

Smoothes image using box filter

```
void boxFilter( const Mat& src, Mat& dst, int ddepth,
            Size ksize, Point anchor=Point(-1,-1),
            bool normalize=true,
            int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same type as `src`

ksize The smoothing kernel size

anchor The anchor point. The default value `Point(-1,-1)` means that the anchor is at the kernel center

normalize Indicates, whether the kernel is normalized by its area or not

borderType The border mode used to extrapolate pixels outside of the image

The function smoothes the image using the kernel:

$$\mathrm{K} = \alpha \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

where

$$\alpha = \begin{cases} \frac{1}{\texttt{ksize.width*ksize.height}} & \text{when } \texttt{normalize=true} \\ 1 & \text{otherwise} \end{cases}$$

Unnormalized box filter is useful for computing various integral characteristics over each pixel neighborhood, such as covariation matrices of image derivatives (used in dense optical flow algorithms, Harris corner detector etc.). If you need to compute pixel sums over variable-size windows, use cv::integral.

See also: cv::boxFilter, cv::bilateralFilter, cv::GaussianBlur, cv::medianBlur, cv::integral.

---

cv::buildPyramid (view/add comments)

Constructs Gaussian pyramid for an image

```
void buildPyramid( const Mat& src, vector<Mat>& dst, int maxlevel );
```

src The source image; check cv::pyrDown for the list of supported types

dst The destination vector of `maxlevel+1` images of the same type as `src`; `dst[0]` will be the same as `src`, `dst[1]` is the next pyramid layer, a smoothed and down-sized `src` etc.

maxlevel The 0-based index of the last (i.e. the smallest) pyramid layer; it must be non-negative

The function constructs a vector of images and builds the gaussian pyramid by recursively applying cv::pyrDown to the previously built pyramid layers, starting from `dst[0]==src`.

---

cv::copyMakeBorder (view/add comments)

Forms a border around the image

```
void copyMakeBorder( const Mat& src, Mat& dst,
            int top, int bottom, int left, int right,
            int borderType, const Scalar& value=Scalar() );
```

src The source image

dst The destination image; will have the same type as `src` and the size `Size(src.cols+left+right, src.rows+top+bottom)`

top, bottom, left, right Specify how much pixels in each direction from the source image rectangle one needs to extrapolate, e.g. `top=1, bottom=1, left=1, right=1` mean that 1 pixel-wide border needs to be built

borderType The border type; see cv::borderInterpolate

value The border value if `borderType==BORDER_CONSTANT`

The function copies the source image into the middle of the destination image. The areas to the left, to the right, above and below the copied source image will be filled with extrapolated pixels. This is not what cv::FilterEngine or based on it filtering functions do (they extrapolate pixels on-fly), but what other more complex functions, including your own, may do to simplify image boundary handling.

The function supports the mode when `src` is already in the middle of `dst`. In this case the function does not copy `src` itself, but simply constructs the border, e.g.:

```
// let border be the same in all directions
int border=2;
// constructs a larger image to fit both the image and the border
Mat gray_buf(rgb.rows + border*2, rgb.cols + border*2, rgb.depth());
// select the middle part of it w/o copying data
Mat gray(gray_canvas, Rect(border, border, rgb.cols, rgb.rows));
// convert image from RGB to grayscale
cvtColor(rgb, gray, CV_RGB2GRAY);
// form a border in-place
copyMakeBorder(gray, gray_buf, border, border,
               border, border, BORDER_REPLICATE);
// now do some custom filtering ...
...
```

See also: cv::borderInterpolate

cv::createBoxFilter (view/add comments)

Returns box filter engine

```
Ptr<FilterEngine> createBoxFilter( int srcType, int dstType,
           Size ksize, Point anchor=Point(-1,-1),
           bool normalize=true,
           int borderType=BORDER_DEFAULT);
Ptr<BaseRowFilter> getRowSumFilter(int srcType, int sumType,
           int ksize, int anchor=-1);
Ptr<BaseColumnFilter> getColumnSumFilter(int sumType, int dstType,
           int ksize, int anchor=-1, double scale=1);
```

srcType The source image type

sumType The intermediate horizontal sum type; must have as many channels as `srcType`

dstType The destination image type; must have as many channels as `srcType`

ksize The aperture size

anchor The anchor position with the kernel; negative values mean that the anchor is at the kernel center

normalize Whether the sums are normalized or not; see cv::boxFilter

scale Another way to specify normalization in lower-level `getColumnSumFilter`

borderType Which border type to use; see cv::borderInterpolate

The function is a convenience function that retrieves horizontal sum primitive filter with cv::getRowSumFilter, vertical sum filter with cv::getColumnSumFilter, constructs new cv::FilterEngine and passes both of the primitive filters there. The constructed filter engine can be used for image filtering with normalized or unnormalized box filter.

The function itself is used by cv::blur and cv::boxFilter.

See also: cv::FilterEngine, cv::blur, cv::boxFilter.

cv::createDerivFilter (view/add comments)

Returns engine for computing image derivatives

```
Ptr<FilterEngine> createDerivFilter( int srcType, int dstType,
            int dx, int dy, int ksize,
            int borderType=BORDER_DEFAULT );
```

srcType The source image type

dstType The destination image type; must have as many channels as `srcType`

dx The derivative order in respect with x

dy The derivative order in respect with y

ksize The aperture size; see cv::getDerivKernels

borderType Which border type to use; see cv::borderInterpolate

The function cv::createDerivFilter is a small convenience function that retrieves linear filter coefficients for computing image derivatives using cv::getDerivKernels and then creates a separable linear filter with cv::createSeparableLinearFilter. The function is used by cv::Sobel and cv::Scharr.

See also: cv::createSeparableLinearFilter, cv::getDerivKernels, cv::Scharr, cv::Sobel.

cv::createGaussianFilter (view/add comments)

Returns engine for smoothing images with a Gaussian filter

```
Ptr<FilterEngine> createGaussianFilter( int type, Size ksize,
            double sigmaX, double sigmaY=0,
            int borderType=BORDER_DEFAULT);
```

type The source and the destination image type

ksize The aperture size; see cv::getGaussianKernel

sigmaX The Gaussian sigma in the horizontal direction; see cv::getGaussianKernel

sigmaY The Gaussian sigma in the vertical direction; if 0, then `sigmaY` ← `sigmaX`

borderType Which border type to use; see cv::borderInterpolate

The function cv::createGaussianFilter computes Gaussian kernel coefficients and then returns separable linear filter for that kernel. The function is used by cv::GaussianBlur. Note that while the function takes just one data type, both for input and output, you can pass by this limitation by calling cv::getGaussianKernel and then cv::createSeparableFilter directly.

See also: cv::createSeparableLinearFilter, cv::getGaussianKernel, cv::GaussianBlur.

---

cv::createLinearFilter (view/add comments)

Creates non-separable linear filter engine

```
Ptr<FilterEngine> createLinearFilter(int srcType, int dstType,
            const Mat& kernel, Point _anchor=Point(-1,-1),
            double delta=0, int rowBorderType=BORDER_DEFAULT,
            int columnBorderType=-1, const Scalar& borderValue=Scalar());
Ptr<BaseFilter> getLinearFilter(int srcType, int dstType,
            const Mat& kernel,
            Point anchor=Point(-1,-1),
            double delta=0, int bits=0);
```

srcType The source image type

dstType The destination image type; must have as many channels as `srcType`

kernel The 2D array of filter coefficients

anchor The anchor point within the kernel; special value `Point(-1,-1)` means that the anchor is at the kernel center

delta The value added to the filtered results before storing them

bits When the kernel is an integer matrix representing fixed-point filter coefficients, the parameter specifies the number of the fractional bits

rowBorderType, columnBorderType The pixel extrapolation methods in the horizontal and the vertical directions; see cv::borderInterpolate

borderValue Used in case of constant border

The function returns pointer to 2D linear filter for the specified kernel, the source array type and the destination array type. The function is a higher-level function that calls `getLinearFilter` and passes the retrieved 2D filter to cv::FilterEngine constructor.

See also: cv::createSeparableLinearFilter, cv::FilterEngine, cv::filter2D

---

cv::createMorphologyFilter (view/add comments)

Creates engine for non-separable morphological operations

```
Ptr<FilterEngine> createMorphologyFilter(int op, int type,
           const Mat& element, Point anchor=Point(-1,-1),
           int rowBorderType=BORDER_CONSTANT,
           int columnBorderType=-1,
           const Scalar& borderValue=morphologyDefaultBorderValue());
Ptr<BaseFilter> getMorphologyFilter(int op, int type, const Mat&
element,
           Point anchor=Point(-1,-1));
Ptr<BaseRowFilter> getMorphologyRowFilter(int op, int type,
           int esize, int anchor=-1);
Ptr<BaseColumnFilter> getMorphologyColumnFilter(int op, int type,
           int esize, int anchor=-1);
static inline Scalar morphologyDefaultBorderValue()
           return Scalar::all(DBL_MAX);
```

op The morphology operation id, `MORPH_ERODE` or `MORPH_DILATE`

type The input/output image type

element The 2D 8-bit structuring element for the morphological operation. Non-zero elements indicate the pixels that belong to the element

esize The horizontal or vertical structuring element size for separable morphological operations

anchor The anchor position within the structuring element; negative values mean that the anchor is at the center

rowBorderType, columnBorderType The pixel extrapolation methods in the horizontal and the vertical directions; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value, `morphologyDefaultBorderValue`, has the special meaning. It is transformed $+\inf$ for the erosion and to $-\inf$ for the dilation, which means that the minimum (maximum) is effectively computed only over the pixels that are inside the image.

The functions construct primitive morphological filtering operations or a filter engine based on them. Normally it's enough to use cv::createMorphologyFilter or even higher-level cv::erode, cv::dilate or cv::morphologyEx, Note, that cv::createMorphologyFilter analyses the structuring element shape and builds a separable morphological filter engine when the structuring element is square.

See also: cv::erode, cv::dilate, cv::morphologyEx, cv::FilterEngine

---

cv::createSeparableLinearFilter (view/add comments)

Creates engine for separable linear filter

```
Ptr<FilterEngine> createSeparableLinearFilter(int srcType, int dstType,
           const Mat& rowKernel, const Mat& columnKernel,
           Point anchor=Point(-1,-1), double delta=0,
           int rowBorderType=BORDER_DEFAULT,
           int columnBorderType=-1,
           const Scalar& borderValue=Scalar());
Ptr<BaseColumnFilter> getLinearColumnFilter(int bufType, int dstType,
           const Mat& columnKernel, int anchor,
           int symmetryType, double delta=0,
           int bits=0);
Ptr<BaseRowFilter> getLinearRowFilter(int srcType, int bufType,
           const Mat& rowKernel, int anchor,
           int symmetryType);
```

srcType The source array type

dstType The destination image type; must have as many channels as `srcType`

bufType The inermediate buffer type; must have as many channels as `srcType`

rowKernel The coefficients for filtering each row

columnKernel The coefficients for filtering each column

anchor The anchor position within the kernel; negative values mean that anchor is positioned at the aperture center

delta The value added to the filtered results before storing them

bits When the kernel is an integer matrix representing fixed-point filter coefficients, the parameter specifies the number of the fractional bits

rowBorderType, columnBorderType The pixel extrapolation methods in the horizontal and the vertical directions; see cv::borderInterpolate

borderValue Used in case of a constant border

symmetryType The type of each of the row and column kernel; see cv::getKernelType.

The functions construct primitive separable linear filtering operations or a filter engine based on them. Normally it's enough to use cv::createSeparableLinearFilter or even higher-level cv::sepFilter2D. The function cv::createMorphologyFilter is smart enough to figure out the `symmetryType` for each of the two kernels, the intermediate `bufType`, and, if the filtering can be done in integer arithmetics, the number of `bits` to encode the filter coefficients. If it does not work for you, it's possible to call `getLinearColumnFilter`, `getLinearRowFilter` directly and then pass them to cv::FilterEngine constructor.

See also: cv::sepFilter2D, cv::createLinearFilter, cv::FilterEngine, cv::getKernelType

---

cv::dilate (view/add comments)

Dilates an image by using a specific structuring element.

```
void dilate( const Mat& src, Mat& dst, const Mat& element,
             Point anchor=Point(-1,-1), int iterations=1,
             int borderType=BORDER_CONSTANT,
             const Scalar& borderValue=morphologyDefaultBorderValue() );
```

src The source image

dst The destination image. It will have the same size and the same type as `src`

element The structuring element used for dilation. If `element=Mat()`, a 3×3 rectangular structuring element is used

anchor Position of the anchor within the element. The default value $(-1, -1)$ means that the anchor is at the element center

iterations The number of times dilation is applied

borderType The pixel extrapolation method; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value has a special meaning, see cv::createMorphologyFilter

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\texttt{dst}(x,y) = \max_{(x',y'):\, \texttt{element}(x',y') \neq 0} \texttt{src}(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. In the case of multi-channel images each channel is processed independently.

See also: cv::erode, cv::morphologyEx, cv::createMorphologyFilter

---

cv::erode (view/add comments)

Erodes an image by using a specific structuring element.

```
void erode( const Mat& src, Mat& dst, const Mat& element,
            Point anchor=Point(-1,-1), int iterations=1,
            int borderType=BORDER_CONSTANT,
            const Scalar& borderValue=morphologyDefaultBorderValue() );
```

src The source image

dst The destination image. It will have the same size and the same type as `src`

element The structuring element used for dilation. If `element=Mat()`, a 3×3 rectangular structuring element is used

anchor Position of the anchor within the element. The default value $(-1, -1)$ means that the anchor is at the element center

iterations The number of times erosion is applied

borderType The pixel extrapolation method; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value has a special meaning, see cv::createMorphoogyFilter

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\texttt{dst}(x,y) = \min_{(x',y'):\, \texttt{element}(x',y') \neq 0} \texttt{src}(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. In the case of multi-channel images each channel is processed independently.

See also: cv::dilate, cv::morphologyEx, cv::createMorphologyFilter

---

cv::filter2D (view/add comments)

Convolves an image with the kernel

```
void filter2D( const Mat& src, Mat& dst, int ddepth,
           const Mat& kernel, Point anchor=Point(-1,-1),
           double delta=0, int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image. It will have the same size and the same number of channels as `src`

ddepth The desired depth of the destination image. If it is negative, it will be the same as `src.depth()`

kernel Convolution kernel (or rather a correlation kernel), a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cv::split and process them individually

anchor The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor should lie within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center

delta The optional value added to the filtered pixels before storing them in `dst`

borderType The pixel extrapolation method; see cv::borderInterpolate

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

The function does actually computes correlation, not the convolution:

$$\texttt{dst}(x,y) = \sum_{\substack{0 \le x' < \texttt{kernel.cols}, \\ 0 \le y' < \texttt{kernel.rows}}} \texttt{kernel}(x',y') * \texttt{src}(x+x'-\texttt{anchor.x}, y+y'-\texttt{anchor.y})$$

That is, the kernel is not mirrored around the anchor point. If you need a real convolution, flip the kernel using cv::flip and set the new anchor to `(kernel.cols - anchor.x - 1, kernel.rows - anchor.y - 1)`.

The function uses DFT-based algorithm in case of sufficiently large kernels (11×11) and the direct algorithm (that uses the engine retrieved by cv::createLinearFilter) for small kernels.

See also: cv::sepFilter2D, cv::createLinearFilter, cv::dft, cv::matchTemplate

---

cv::GaussianBlur (view/add comments)

Smoothes image using a Gaussian filter

```
void GaussianBlur( const Mat& src, Mat& dst, Size ksize,
           double sigmaX, double sigmaY=0,
           int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same type as `src`

ksize The Gaussian kernel size; `ksize.width` and `ksize.height` can differ, but they both must be positive and odd. Or, they can be zero's, then they are computed from `sigma*`

sigmaX, sigmaY The Gaussian kernel standard deviations in X and Y direction. If `sigmaY` is zero, it is set to be equal to `sigmaX`. If they are both zeros, they are computed from `ksize.width` and `ksize.height`, respectively, see cv::getGaussianKernel. To fully control the result regardless of possible future modification of all this semantics, it is recommended to specify all of `ksize`, `sigmaX` and `sigmaY`

borderType The pixel extrapolation method; see cv::borderInterpolate

The function convolves the source image with the specified Gaussian kernel. In-place filtering is supported.

See also: cv::sepFilter2D, cv::filter2D, cv::blur, cv::boxFilter, cv::bilateralFilter, cv::medianBlur

---

cv::getDerivKernels (view/add comments)

Returns filter coefficients for computing spatial image derivatives

```
void getDerivKernels( Mat& kx, Mat& ky, int dx, int dy, int ksize,
            bool normalize=false, int ktype=CV_32F );
```

kx The output matrix of row filter coefficients; will have type `ktype`

ky The output matrix of column filter coefficients; will have type `ktype`

dx The derivative order in respect with x

dy The derivative order in respect with y

ksize The aperture size. It can be `CV_SCHARR`, 1, 3, 5 or 7

normalize Indicates, whether to normalize (scale down) the filter coefficients or not. In theory the coefficients should have the denominator $= 2^{ksize*2-dx-dy-2}$. If you are going to filter floating-point images, you will likely want to use the normalized kernels. But if you compute derivatives of a 8-bit image, store the results in 16-bit image and wish to preserve all the fractional bits, you may want to set `normalize=false`.

ktype The type of filter coefficients. It can be `CV_32f` or `CV_64F`

The function computes and returns the filter coefficients for spatial image derivatives. When `ksize=CV_SCHARR`, the Scharr 3×3 kernels are generated, see cv::Scharr. Otherwise, Sobel kernels are generated, see cv::Sobel. The filters are normally passed to cv::sepFilter2D or to cv::createSeparableLinearFilter.

cv::getGaussianKernel (view/add comments)

Returns Gaussian filter coefficients

```
Mat getGaussianKernel( int ksize, double sigma, int ktype=CV_64F );
```

ksize The aperture size. It should be odd ($\texttt{ksize} \mod 2 = 1$) and positive.

sigma The Gaussian standard deviation. If it is non-positive, it is computed from `ksize` as
`sigma = 0.3*(ksize/2 - 1) + 0.8`

ktype The type of filter coefficients. It can be `CV_32f` or `CV_64F`

The function computes and returns the $\texttt{ksize} \times 1$ matrix of Gaussian filter coefficients:

$$G_i = \alpha * e^{-(i-(\texttt{ksize}-1)/2)^2/(2*\texttt{sigma})^2},$$

where $i = 0..\texttt{ksize} - 1$ and α is the scale factor chosen so that $\sum_i G_i = 1$
Two of such generated kernels can be passed to cv::sepFilter2D or to cv::createSeparableLinearFilter that will automatically detect that these are smoothing kernels and handle them accordingly. Also you may use the higher-level cv::GaussianBlur.
See also: cv::sepFilter2D, cv::createSeparableLinearFilter, cv::getDerivKernels, cv::getStructuringElement, cv::GaussianBlur.

cv::getKernelType (view/add comments)

Returns the kernel type

```
int getKernelType(const Mat& kernel, Point anchor);
```

kernel 1D array of the kernel coefficients to analyze

anchor The anchor position within the kernel

The function analyzes the kernel coefficients and returns the corresponding kernel type:

KERNEL_GENERAL Generic kernel - when there is no any type of symmetry or other properties

KERNEL_SYMMETRICAL The kernel is symmetrical: $\texttt{kernel}_i == \texttt{kernel}_{ksize-i-1}$ and the anchor is at the center

KERNEL_ASYMMETRICAL The kernel is asymmetrical: $\texttt{kernel}_i == -\texttt{kernel}_{ksize-i-1}$ and the anchor is at the center

KERNEL_SMOOTH All the kernel elements are non-negative and sum to 1. E.g. the Gaussian kernel is both smooth kernel and symmetrical, so the function will return KERNEL_SMOOTH | KERNEL_SYMMETRICAL

KERNEL_INTEGER Al the kernel coefficients are integer numbers. This flag can be combined with KERNEL_SYMMETRICAL or KERNEL_ASYMMETRICAL

---

cv::getStructuringElement (view/add comments)

Returns the structuring element of the specified size and shape for morphological operations

```
Mat getStructuringElement(int shape, Size esize,
          Point anchor=Point(-1,-1));
```

shape The element shape, one of:

- MORPH_RECT - rectangular structuring element

$$E_{ij} = 1$$

- MORPH_ELLIPSE - elliptic structuring element, i.e. a filled ellipse inscribed into the rectangle Rect(0, 0, esize.width, 0.esize.height)
- MORPH_CROSS - cross-shaped structuring element:

$$E_{ij} = \begin{cases} 1 & \text{if i=anchor.y or j=anchor.x} \\ 0 & \text{otherwise} \end{cases}$$

esize Size of the structuring element

anchor The anchor position within the element. The default value $(-1, -1)$ means that the anchor is at the center. Note that only the cross-shaped element's shape depends on the anchor position; in other cases the anchor just regulates by how much the result of the morphological operation is shifted

The function constructs and returns the structuring element that can be then passed to cv::createMorphologyF cv::erode, cv::dilate or cv::morphologyEx. But also you can construct an arbitrary binary mask yourself and use it as the structuring element.

cv::medianBlur (view/add comments)

Smoothes image using median filter

```
void medianBlur( const Mat& src, Mat& dst, int ksize );
```

src The source 1-, 3- or 4-channel image. When `ksize` is 3 or 5, the image depth should be `CV_8U`, `CV_16U` or `CV_32F`. For larger aperture sizes it can only be `CV_8U`

dst The destination array; will have the same size and the same type as `src`

ksize The aperture linear size. It must be odd and more than 1, i.e. 3, 5, 7 ...

The function smoothes image using the median filter with `ksize` × `ksize` aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

See also: cv::bilateralFilter, cv::blur, cv::boxFilter, cv::GaussianBlur

cv::morphologyEx (view/add comments)

Performs advanced morphological transformations

```
void morphologyEx( const Mat& src, Mat& dst,
           int op, const Mat& element,
           Point anchor=Point(-1,-1), int iterations=1,
           int borderType=BORDER_CONSTANT,
           const Scalar& borderValue=morphologyDefaultBorderValue() );
```

src Source image

dst Destination image. It will have the same size and the same type as `src`

element Structuring element

op Type of morphological operation, one of the following:

MORPH_OPEN opening

MORPH_CLOSE closing

MORPH_GRADIENT morphological gradient

MORPH_TOPHAT "top hat"

MORPH_BLACKHAT "black hat"

iterations Number of times erosion and dilation are applied

borderType The pixel extrapolation method; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value has a special meaning, see cv::createMorphoogyFilter

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$\texttt{dst} = \mathrm{open}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\mathrm{erode}(\texttt{src}, \texttt{element}))$$

Closing:

$$\texttt{dst} = \mathrm{close}(\texttt{src}, \texttt{element}) = \mathrm{erode}(\mathrm{dilate}(\texttt{src}, \texttt{element}))$$

Morphological gradient:

$$\texttt{dst} = \mathrm{morph_grad}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\texttt{src}, \texttt{element}) - \mathrm{erode}(\texttt{src}, \texttt{element})$$

"Top hat":

$$\texttt{dst} = \mathrm{tophat}(\texttt{src}, \texttt{element}) = \texttt{src} - \mathrm{open}(\texttt{src}, \texttt{element})$$

"Black hat":

$$\texttt{dst} = \mathrm{blackhat}(\texttt{src}, \texttt{element}) = \mathrm{close}(\texttt{src}, \texttt{element}) - \texttt{src}$$

Any of the operations can be done in-place.

See also: cv::dilate, cv::erode, cv::createMorphologyFilter

---

cv::Laplacian (view/add comments)

Calculates the Laplacian of an image

```
void Laplacian( const Mat& src, Mat& dst, int ddepth,
            int ksize=1, double scale=1, double delta=0,
            int borderType=BORDER_DEFAULT );
```

src Source image

dst Destination image; will have the same size and the same number of channels as `src`

ddepth The desired depth of the destination image

ksize The aperture size used to compute the second-derivative filters, see cv::getDerivKernels. It must be positive and odd

scale The optional scale factor for the computed Laplacian values (by default, no scaling is applied, see cv::getDerivKernels)

delta The optional delta value, added to the results prior to storing them in `dst`

borderType The pixel extrapolation method, see cv::borderInterpolate

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst} = \Delta \texttt{src} = \frac{\partial^2 \texttt{src}}{\partial x^2} + \frac{\partial^2 \texttt{src}}{\partial y^2}$$

This is done when `ksize > 1`. When `ksize == 1`, the Laplacian is computed by filtering the image with the following 3×3 aperture:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

See also: cv::Sobel, cv::Scharr

---

cv::pyrDown (view/add comments)

Smoothes an image and downsamples it.

```
void pyrDown( const Mat& src, Mat& dst, const Size& dstsize=Size());
```

src The source image

dst The destination image. It will have the specified size and the same type as `src`

dstsize Size of the destination image. By default it is computed as `Size((src.cols+1)/2, (src.rows+1)/2)`. But in any case the following conditions should be satisfied:

$$|\texttt{dstsize.width} * 2 - src.cols| \leq 2$$
$$|\texttt{dstsize.height} * 2 - src.rows| \leq 2$$

The function performs the downsampling step of the Gaussian pyramid construction. First it convolves the source image with the kernel:

$$\frac{1}{16}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

and then downsamples the image by rejecting even rows and columns.

---

cv::pyrUp (view/add comments)

Upsamples an image and then smoothes it

```
void pyrUp( const Mat& src, Mat& dst, const Size& dstsize=Size());
```

src The source image

dst The destination image. It will have the specified size and the same type as `src`

dstsize Size of the destination image. By default it is computed as `Size(src.cols*2, (src.rows*2)`. But in any case the following conditions should be satisfied:

$$|\texttt{dstsize.width} - src.cols * 2| \leq (\texttt{dstsize.width} \mod 2)$$
$$|\texttt{dstsize.height} - src.rows * 2| \leq (\texttt{dstsize.height} \mod 2)$$

The function performs the upsampling step of the Gaussian pyramid construction (it can actually be used to construct the Laplacian pyramid). First it upsamples the source image by injecting even zero rows and columns and then convolves the result with the same kernel as in cv::pyrDown, multiplied by 4.

---

cv::sepFilter2D (view/add comments)

Applies separable linear filter to an image

```
void sepFilter2D( const Mat& src, Mat& dst, int ddepth,
           const Mat& rowKernel, const Mat& columnKernel,
           Point anchor=Point(-1,-1),
           double delta=0, int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same number of channels as `src`

ddepth The destination image depth

rowKernel The coefficients for filtering each row

columnKernel The coefficients for filtering each column

anchor The anchor position within the kernel; The default value $(-1, 1)$ means that the anchor is at the kernel center

delta The value added to the filtered results before storing them

borderType The pixel extrapolation method; see cv::borderInterpolate

The function applies a separable linear filter to the image. That is, first, every row of `src` is filtered with 1D kernel `rowKernel`. Then, every column of the result is filtered with 1D kernel `columnKernel` and the final result shifted by `delta` is stored in `dst`.

See also: cv::createSeparableLinearFilter, cv::filter2D, cv::Sobel, cv::GaussianBlur, cv::boxFilter, cv::blur.

cv::Sobel (view/add comments)

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator

```
void Sobel( const Mat& src, Mat& dst, int ddepth,
            int xorder, int yorder, int ksize=3,
            double scale=1, double delta=0,
            int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same number of channels as `src`

ddepth The destination image depth

xorder Order of the derivative x

yorder Order of the derivative y

ksize Size of the extended Sobel kernel, must be 1, 3, 5 or 7

scale The optional scale factor for the computed derivative values (by default, no scaling is applied, see cv::getDerivKernels)

delta The optional delta value, added to the results prior to storing them in `dst`

borderType The pixel extrapolation method, see cv::borderInterpolate

In all cases except 1, an `ksize` $\times$ `ksize` separable kernel will be used to calculate the derivative. When `ksize = 1`, a 3×1 or 1×3 kernel will be used (i.e. no Gaussian smoothing is done). `ksize = 1` can only be used for the first or the second x- or y- derivatives.

There is also the special value `ksize = CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. The Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\texttt{dst} = \frac{\partial^{xorder+yorder}\texttt{src}}{\partial x^{xorder}\partial y^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation, so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `ksize` = 3) or (`xorder` = 0, `yorder` = 1, `ksize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and the second one corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

See also: cv::Scharr, cv::Lapacian, cv::sepFilter2D, cv::filter2D, cv::GaussianBlur

---

cv::Scharr (view/add comments)

Calculates the first x- or y- image derivative using Scharr operator

```
void Scharr( const Mat& src, Mat& dst, int ddepth,
             int xorder, int yorder,
             double scale=1, double delta=0,
             int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same number of channels as `src`

ddepth The destination image depth

xorder Order of the derivative x

yorder Order of the derivative y

scale The optional scale factor for the computed derivative values (by default, no scaling is applied, see cv::getDerivKernels)

delta The optional delta value, added to the results prior to storing them in `dst`

borderType The pixel extrapolation method, see cv::borderInterpolate

The function computes the first x- or y- spatial image derivative using Scharr operator. The call

```
Scharr(src, dst, ddepth, xorder, yorder, scale, delta, borderType)
```

is equivalent to

```
Sobel(src, dst, ddepth, xorder, yorder, CV_SCHARR, scale, delta, borderType).
```

12.3 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x, y) = \texttt{src}(f_x(x, y), f_y(x, y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic cv::remap and to the simplest and the fastest cv::resize, need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x, y) one of $f_x(x, y)$ or $f_y(x, y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method `BORDER_TRANSPARENT`, which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x, y)$ and $f_y(x, y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction

etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x, y), f_y(x, y))$ and then the value of the polynomial at $(f_x(x, y), f_y(x, y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see cv::resize.

---

cv::convertMaps (view/add comments)

Converts image transformation maps from one representation to another

```
void convertMaps( const Mat& map1, const Mat& map2,
           Mat& dstmap1, Mat& dstmap2,
           int dstmap1type, bool nninterpolation=false );
```

map1 The first input map of type `CV_16SC2` or `CV_32FC1` or `CV_32FC2`

map2 The second input map of type `CV_16UC1` or `CV_32FC1` or none (empty matrix), respectively

dstmap1 The first output map; will have type `dstmap1type` and the same size as `src`

dstmap2 The second output map

dstmap1type The type of the first output map; should be `CV_16SC2`, `CV_32FC1` or `CV_32FC2`

nninterpolation Indicates whether the fixed-point maps will be used for nearest-neighbor or for more complex interpolation

The function converts a pair of maps for cv::remap from one representation to another. The following options (`(map1.type(), map2.type())` → `(dstmap1.type(), dstmap2.type())`) are supported:

1. `(CV_32FC1, CV_32FC1)` → `(CV_16SC2, CV_16UC1)`. This is the most frequently used conversion operation, in which the original floating-point maps (see cv::remap) are converted to more compact and much faster fixed-point representation. The first output array will contain the rounded coordinates and the second array (created only when `nninterpolation=false`) will contain indices in the interpolation tables.

2. `(CV_32FC2)` $\rightarrow$ `(CV_16SC2, CV_16UC1)`. The same as above, but the original maps are stored in one 2-channel matrix.

3. the reverse conversion. Obviously, the reconstructed floating-point maps will not be exactly the same as the originals.

See also: cv::remap, cv::undisort, cv::initUndistortRectifyMap

---

cv::getAffineTransform (view/add comments)

Calculates the affine transform from 3 pairs of the corresponding points

```
Mat getAffineTransform( const Point2f src[], const Point2f dst[] );
```

src Coordinates of a triangle vertices in the source image

dst Coordinates of the corresponding triangle vertices in the destination image

The function calculates the 2×3 matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{map_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

See also: cv::warpAffine, cv::transform

---

cv::getPerspectiveTransform (view/add comments)

Calculates the perspective transform from 4 pairs of the corresponding points

```
Mat getPerspectiveTransform( const Point2f src[],
           const Point2f dst[] );
```

src Coordinates of a quadrange vertices in the source image

dst Coordinates of the corresponding quadrangle vertices in the destination image

The function calculates the 3×3 matrix of a perspective transform such that:

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \texttt{map_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

See also: cv::findHomography, cv::warpPerspective, cv::perspectiveTransform

---

cv::getRectSubPix (view/add comments)

Retrieves the pixel rectangle from an image with sub-pixel accuracy

```
void getRectSubPix( const Mat& image, Size patchSize,
           Point2f center, Mat& dst, int patchType=-1 );
```

src Source image

patchSize Size of the extracted patch

center Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image

dst The extracted patch; will have the size `patchSize` and the same number of channels as `src`

patchType The depth of the extracted pixels. By default they will have the same depth as `src`

The function `getRectSubPix` extracts pixels from `src`:

$$dst(x, y) = src(x + \texttt{center.x} - (\texttt{dst.cols} - 1) * 0.5, y + \texttt{center.y} - (\texttt{dst.rows} - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode (see cv::borderInterpolate) is used to extrapolate the pixel values outside of the image.

See also: cv::warpAffine, cv::warpPerspective

---

cv::getRotationMatrix2D (view/add comments)

Calculates the affine matrix of 2d rotation.

```
Mat getRotationMatrix2D( Point2f center, double angle, double scale );
```

center Center of the rotation in the source image

angle The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)

scale Isotropic scale factor

The function calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} + (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\begin{aligned} \alpha &= \texttt{scale} \cdot \cos \texttt{angle}, \\ \beta &= \texttt{scale} \cdot \sin \texttt{angle} \end{aligned}$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

See also: cv::getAffineTransform, cv::warpAffine, cv::transform

---

cv::invertAffineTransform (view/add comments)

Inverts an affine transformation

```
void invertAffineTransform(const Mat& M, Mat& iM);
```

M The original affine transformation

iM The output reverse affine transformation

The function computes inverse affine transformation represented by 2×3 matrix M:

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

The result will also be a 2×3 matrix of the same type as M.

---

cv::remap (view/add comments)

Applies a generic geometrical transformation to an image.

```
void remap( const Mat& src, Mat& dst, const Mat& map1, const Mat& map2,
            int interpolation, int borderMode=BORDER_CONSTANT,
            const Scalar& borderValue=Scalar());
```

src Source image

dst Destination image. It will have the same size as map1 and the same type as src

map1 The first map of either (x,y) points or just x values having type CV_16SC2, CV_32FC1 or CV_32FC2. See cv::convertMaps for converting floating point representation to fixed-point for speed.

map2 The second map of y values having type CV_16UC1, CV_32FC1 or none (empty map if map1 is (x,y) points), respectively

interpolation The interpolation method, see cv::resize. The method INTER_AREA is not supported by this function

borderMode The pixel extrapolation method, see cv::borderInterpolate. When the borderMode=BORDER_TRANSPARENT, it means that the pixels in the destination image that corresponds to the ”outliers” in the source image are not modified by the function

borderValue A value used in the case of a constant border. By default it is 0

The function `remap` transforms the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(map_x(x,y), map_y(x,y))$$

Where values of pixels with non-integer coordinates are computed using one of the available interpolation methods. map_x and map_y can be encoded as separate floating-point maps in map_1 and map_2 respectively, or interleaved floating-point maps of (x,y) in map_1, or fixed-point maps made by using cv::convertMaps. The reason you might want to convert from floating to fixed-point representations of a map is that they can yield much faster (2x) remapping operations. In the converted case, map_1 contains pairs `(cvFloor(x), cvFloor(y))` and map_2 contains indices in a table of interpolation coefficients.

This function can not operate in-place.

---

cv::resize (view/add comments)

Resizes an image

```
void resize( const Mat& src, Mat& dst,
             Size dsize, double fx=0, double fy=0,
             int interpolation=INTER_LINEAR );
```

src Source image

dst Destination image. It will have size `dsize` (when it is non-zero) or the size computed from `src.size()` and `fx` and `fy`. The type of `dst` will be the same as of `src`.

dsize The destination image size. If it is zero, then it is computed as:

```
dsize = Size(round(fx*src.cols), round(fy*src.rows))
```

. Either `dsize` or both `fx` or `fy` must be non-zero.

fx The scale factor along the horizontal axis. When 0, it is computed as

```
(double)dsize.width/src.cols
```

fy The scale factor along the vertical axis. When 0, it is computed as

```
(double)dsize.height/src.rows
```

interpolation The interpolation method:

- **INTER_NEAREST** nearest-neighbor interpolation
- **INTER_LINEAR** bilinear interpolation (used by default)
- **INTER_AREA** resampling using pixel area relation. It may be the preferred method for image decimation, as it gives moire-free results. But when the image is zoomed, it is similar to the INTER_NEAREST method
- **INTER_CUBIC** bicubic interpolation over 4x4 pixel neighborhood
- **INTER_LANCZOS4** Lanczos interpolation over 8x8 pixel neighborhood

The function resize resizes an image src down to or up to the specified size. Note that the initial dst type or size are not taken into account. Instead the size and type are derived from the src, dsize, fx and fy. If you want to resize src so that it fits the pre-created dst, you may call the function as:

```
// explicitly specify dsize=dst.size(); fx and fy will be computed from that.
resize(src, dst, dst.size(), 0, 0, interpolation);
```

If you want to decimate the image by factor of 2 in each direction, you can call the function this way:

```
// specify fx and fy and let the function to compute the destination image size.
resize(src, dst, Size(), 0.5, 0.5, interpolation);
```

See also: cv::warpAffine, cv::warpPerspective, cv::remap.

---

cv::warpAffine (view/add comments)

Applies an affine transformation to an image.

```
void warpAffine( const Mat& src, Mat& dst,
                 const Mat& M, Size dsize,
                 int flags=INTER_LINEAR,
                 int borderMode=BORDER_CONSTANT,
                 const Scalar& borderValue=Scalar());
```

src Source image

dst Destination image; will have size dsize and the same type as src

M 2×3 transformation matrix

dsize Size of the destination image

flags A combination of interpolation methods, see cv::resize, and the optional flag `WARP_INVERSE_MAP` that means that `M` is the inverse transformation (`dst` $\rightarrow$ `src`)

borderMode The pixel extrapolation method, see cv::borderInterpolate. When the `borderMode=BORDER_TRANSPARENT`, it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function

borderValue A value used in case of a constant border. By default it is 0

The function `warpAffine` transforms the source image using the specified matrix:

$$\texttt{dst}(x,y) = \texttt{src}(\texttt{M}_{11}x + \texttt{M}_{12}y + \texttt{M}_{13}, \texttt{M}_{21}x + \texttt{M}_{22}y + \texttt{M}_{23})$$

when the flag `WARP_INVERSE_MAP` is set. Otherwise, the transformation is first inverted with cv::invertAffineTransform and then put in the formula above instead of `M`. The function can not operate in-place.

See also: cv::warpPerspective, cv::resize, cv::remap, cv::getRectSubPix, cv::transform

---

cv::warpPerspective (view/add comments)

Applies a perspective transformation to an image.

```
void warpPerspective( const Mat& src, Mat& dst,
           const Mat& M, Size dsize,
           int flags=INTER_LINEAR,
           int borderMode=BORDER_CONSTANT,
           const Scalar& borderValue=Scalar());
```

src Source image

dst Destination image; will have size `dsize` and the same type as `src`

M 3×3 transformation matrix

dsize Size of the destination image

flags A combination of interpolation methods, see cv::resize, and the optional flag WARP_INVERSE_MAP that means that M is the inverse transformation (dst → src)

borderMode The pixel extrapolation method, see cv::borderInterpolate. When the borderMode=BORDER_TRANSPARENT, it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function

borderValue A value used in case of a constant border. By default it is 0

The function warpPerspective transforms the source image using the specified matrix:

$$\texttt{dst}(x,y) = \texttt{src}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right)$$

when the flag WARP_INVERSE_MAP is set. Otherwise, the transformation is first inverted with cv::invert and then put in the formula above instead of M. The function can not operate in-place.

See also: cv::warpAffine, cv::resize, cv::remap, cv::getRectSubPix, cv::perspectiveTransform

12.4 Miscellaneous Image Transformations

---

cv::adaptiveThreshold (view/add comments)

Applies an adaptive threshold to an array.

```
void adaptiveThreshold( const Mat& src, Mat& dst, double maxValue,
           int adaptiveMethod, int thresholdType,
           int blockSize, double C );
```

src Source 8-bit single-channel image

dst Destination image; will have the same size and the same type as src

maxValue The non-zero value assigned to the pixels for which the condition is satisfied. See the discussion

adaptiveMethod Adaptive thresholding algorithm to use, ADAPTIVE_THRESH_MEAN_C or ADAPTIVE_THRESH_G (see the discussion)

thresholdType Thresholding type; must be one of THRESH_BINARY or THRESH_BINARY_INV

blockSize The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

C The constant subtracted from the mean or weighted mean (see the discussion); normally, it's positive, but may be zero or negative as well

The function transforms a grayscale image to a binary image according to the formulas:

THRESH_BINARY

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

1. For the method ADAPTIVE_THRESH_MEAN_C the threshold value $T(x,y)$ is the mean of a blockSize $\times$ blockSize neighborhood of (x,y), minus C.
2. For the method ADAPTIVE_THRESH_GAUSSIAN_C the threshold value $T(x,y)$ is the weighted sum (i.e. cross-correlation with a Gaussian window) of a blockSize $\times$ blockSize neighborhood of (x,y), minus C. The default sigma (standard deviation) is used for the specified blockSize, see cv::getGaussianKernel.

The function can process the image in-place.
See also: cv::threshold, cv::blur, cv::GaussianBlur

---

cv::cvtColor (view/add comments)

Converts image from one color space to another

```
void cvtColor( const Mat& src, Mat& dst, int code, int dstCn=0 );
```

src The source image, 8-bit unsigned, 16-bit unsigned (CV_16UC...) or single-precision floating-point

dst The destination image; will have the same size and the same depth as src

code The color space conversion code; see the discussion

dstCn The number of channels in the destination image; if the parameter is 0, the number of the channels will be derived automatically from `src` and the `code`

The function converts the input image from one color space to another. In the case of transformation to-from RGB color space the ordering of the channels should be specified explicitly (RGB or BGR).

The conventional ranges for R, G and B channel values are:

- 0 to 255 for `CV_8U` images
- 0 to 65535 for `CV_16U` images and
- 0 to 1 for `CV_32F` images.

Of course, in the case of linear transformations the range does not matter, but in the non-linear cases the input RGB image should be normalized to the proper value range in order to get the correct results, e.g. for RGB→L*u*v* transformation. For example, if you have a 32-bit floating-point image directly converted from 8-bit image without any scaling, then it will have 0..255 value range, instead of the assumed by the function 0..1. So, before calling `cvtColor`, you need first to scale the image down:

```
img *= 1./255;
cvtColor(img, img, CV_BGR2Luv);
```

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} \quad Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} \quad R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

  ```
  cvtColor(src, bwsrc, CV_RGB2GRAY);
  ```

  Some more advanced channel reordering can also be done with cv::mixChannels.

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ`, `CV_RGB2XYZ`, `CV_XYZ2BGR`, `CV_XYZ2RGB`):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB $\leftrightarrow$ YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb`, `CV_RGB2YCrCb`, `CV_YCrCb2BGR`, `CV_YCrCb2RGB`)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

  where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

  Y, Cr and Cb cover the whole value range.

- RGB $\leftrightarrow$ HSV (`CV_BGR2HSV`, `CV_RGB2HSV`, `CV_HSV2BGR`, `CV_HSV2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V = R \\ 120 + 60(B - R)/S & \text{if } V = G \\ 240 + 60(R - G)/S & \text{if } V = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$

On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2\text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R, G, B)$$

$$V_{min} \leftarrow min(R, G, B)$$

$$L \leftarrow \frac{V_{max} + V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max} - V_{min}}{V_{max} + V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max} - V_{min}}{2 - (V_{max} + V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V_{max} = R \\ 120 + 60(B - R)/S & \text{if } V_{max} = G \\ 240 + 60(R - G)/S & \text{if } V_{max} = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$ On output $0 \leq L \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255 \cdot V, S \leftarrow 255 \cdot S, H \leftarrow H/2 \text{ (to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535 \cdot V, S < -65535 \cdot S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \le 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t \le 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \le L \le 100$, $-127 \le a \le 127$, $-127 \le b \le 127$

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow L * 255/100, \; a \leftarrow a + 128, \; b \leftarrow b + 128$$

16-bit images currently not supported

32-bit images L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y \le 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow 255/100L, \; u \leftarrow 255/354(u + 134), \; v \leftarrow 255/256(v + 140)$$

16-bit images currently not supported

32-bit images L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Charles Poynton site http://www.poynton.com/ColorFAQ.html

- Bayer → RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerGB2RGB, CV_BayerRG2RGB, CV_BayerGR2RGB) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{matrix} R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants CV_Bayer C_1C_2 2BGR and CV_Bayer C_1C_2 2RGB indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

cv::distanceTransform (view/add comments)

Calculates the distance to the closest zero pixel for each pixel of the source image.

```
void distanceTransform( const Mat& src, Mat& dst,
           int distanceType, int maskSize );
void distanceTransform( const Mat& src, Mat& dst, Mat& labels,
           int distanceType, int maskSize );
```

src 8-bit, single-channel (binary) source image

dst Output image with calculated distances; will be 32-bit floating-point, single-channel image of the same size as `src`

distanceType Type of distance; can be `CV_DIST_L1, CV_DIST_L2` or `CV_DIST_C`

maskSize Size of the distance transform mask; can be 3, 5 or `CV_DIST_MASK_PRECISE` (the latter option is only supported by the first of the functions). In the case of `CV_DIST_L1` or `CV_DIST_C` distance type the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 or any larger aperture.

labels The optional output 2d array of labels - the discrete Voronoi diagram; will have type `CV_32SC1` and the same size as `src`. See the discussion

The functions `distanceTransform` calculate the approximate or precise distance from every binary image pixel to the nearest zero pixel. (for zero image pixels the distance will obviously be zero).

When `maskSize == CV_DIST_MASK_PRECISE` and `distanceType == CV_DIST_L2`, the function runs the algorithm described in [**?**].

In other cases the algorithm [**?**] is used, that is, for pixel the function finds the shortest path to the nearest zero pixel consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (that is denoted as `a`), all the diagonal shifts must have the same cost (denoted `b`), and all knight's moves must have the same cost (denoted `c`). For `CV_DIST_C` and `CV_DIST_L1` types the distance is calculated precisely, whereas for `CV_DIST_L2` (Euclidian distance) the distance can be calculated only with some relative error (a 5×5 mask gives more accurate results). For `a`, `b` and `c` OpenCV uses the values suggested in the original paper:

CV_DIST_C	(3×3)	a = 1, b = 1
CV_DIST_L1	(3×3)	a = 1, b = 2
CV_DIST_L2	(3×3)	a=0.955, b=1.3693
CV_DIST_L2	(5×5)	a=1, b=1.4, c=2.1969

Typically, for a fast, coarse distance estimation CV_DIST_L2, a 3×3 mask is used, and for a more accurate distance estimation CV_DIST_L2, a 5×5 mask or the precise algorithm is used. Note that both the precise and the approximate algorithms are linear on the number of pixels.

The second variant of the function does not only compute the minimum distance for each pixel (x, y), but it also identifies the nearest the nearest connected component consisting of zero pixels. Index of the component is stored in labels(x, y). The connected components of zero pixels are also found and marked by the function.

In this mode the complexity is still linear. That is, the function provides a very fast way to compute Voronoi diagram for the binary image. Currently, this second variant can only use the approximate distance transform algorithm.

---

cv::floodFill (view/add comments)

Fills a connected component with the given color.

```
int floodFill( Mat& image,
               Point seed, Scalar newVal, Rect* rect=0,
               Scalar loDiff=Scalar(), Scalar upDiff=Scalar(),
               int flags=4 );
int floodFill( Mat& image, Mat& mask,
               Point seed, Scalar newVal, Rect* rect=0,
               Scalar loDiff=Scalar(), Scalar upDiff=Scalar(),
               int flags=4 );
```

image Input/output 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the FLOODFILL_MASK_ONLY flag is set (in the second variant of the function; see below)

mask (For the second function only) Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller. The function uses and updates the mask, so the user takes responsibility of initializing the mask content. Flood-filling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. It is possible to use the same mask in multiple calls to the function to make sure the

filled area do not overlap. **Note**: because the mask is larger than the filled image, a pixel (x, y) in `image` will correspond to the pixel $(x + 1, y + 1)$ in the `mask`

seed The starting point

newVal New value of the repainted domain pixels

loDiff Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component

upDiff Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component

rect The optional output parameter that the function sets to the minimum bounding rectangle of the repainted domain

flags The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:

FLOODFILL_FIXED_RANGE if set, the difference between the current pixel and seed pixel is considered, otherwise the difference between neighbor pixels is considered (i.e. the range is floating)

FLOODFILL_MASK_ONLY (for the second variant only) if set, the function does not change the image (`newVal` is ignored), but fills the mask

The functions `floodFill` fill a connected component starting from the seed point with the specified color. The connectivity is determined by the color/brightness closeness of the neighbor pixels. The pixel at (x, y) is considered to belong to the repainted domain if:

grayscale image, floating range

$$\texttt{src}(x', y') - \texttt{loDiff} \leq \texttt{src}(x, y) \leq \texttt{src}(x', y') + \texttt{upDiff}$$

grayscale image, fixed range

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y) - \texttt{loDiff} \leq \texttt{src}(x, y) \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y) + \texttt{upDiff}$$

color image, floating range

$$\texttt{src}(x', y')_r - \texttt{loDiff}_r \leq \texttt{src}(x, y)_r \leq \texttt{src}(x', y')_r + \texttt{upDiff}_r$$

$$\texttt{src}(x', y')_g - \texttt{loDiff}_g \leq \texttt{src}(x, y)_g \leq \texttt{src}(x', y')_g + \texttt{upDiff}_g$$

$$\texttt{src}(x', y')_b - \texttt{loDiff}_b \leq \texttt{src}(x, y)_b \leq \texttt{src}(x', y')_b + \texttt{upDiff}_b$$

color image, fixed range

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_r - \texttt{loDiff}_r \leq \texttt{src}(x,y)_r \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_r + \texttt{upDiff}_r$$

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_g - \texttt{loDiff}_g \leq \texttt{src}(x,y)_g \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_g + \texttt{upDiff}_g$$

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_b - \texttt{loDiff}_b \leq \texttt{src}(x,y)_b \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_b + \texttt{upDiff}_b$$

where $src(x', y')$ is the value of one of pixel neighbors that is already known to belong to the component. That is, to be added to the connected component, a pixel's color/brightness should be close enough to the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

By using these functions you can either mark a connected component with the specified color in-place, or build a mask and then extract the contour or copy the region to another image etc. Various modes of the function are demonstrated in `floodfill.c` sample.

See also: cv::findContours

---

cv::inpaint (view/add comments)

Inpaints the selected region in the image.

```
void inpaint( const Mat& src, const Mat& inpaintMask,
              Mat& dst, double inpaintRadius, int flags );
```

src The input 8-bit 1-channel or 3-channel image.

inpaintMask The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.

dst The output image; will have the same size and the same type as `src`

inpaintRadius The radius of a circlular neighborhood of each point inpainted that is considered by the algorithm.

flags The inpainting method, one of the following:

INPAINT_NS Navier-Stokes based method.

INPAINT_TELEA The method by Alexandru Telea [**?**]

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video. See http://en.wikipedia.org/wiki/Inpainting for more details.

---

cv::integral (view/add comments)

Calculates the integral of an image.

```
void integral( const Mat& image, Mat& sum, int sdepth=-1 );
void integral( const Mat& image, Mat& sum, Mat& sqsum, int sdepth=-1 );
void integral( const Mat& image, Mat& sum,
           Mat& sqsum, Mat& tilted, int sdepth=-1 );
```

image The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

sum The integral image, $(W+1) \times (H+1)$, 32-bit integer or floating-point (32f or 64f)

sqsum The integral image for squared pixel values, $(W+1) \times (H+1)$, double precision floating-point (64f)

tilted The integral for the image rotated by 45 degrees, $(W+1) \times (H+1)$, the same data type as `sum`

sdepth The desired depth of the integral and the tilted integral images, `CV_32S`, `CV_32F` or `CV_64F`

The functions `integral` calculate one or more integral images for the source image as following:

$$\texttt{sum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)$$

$$\texttt{sqsum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)^2$$

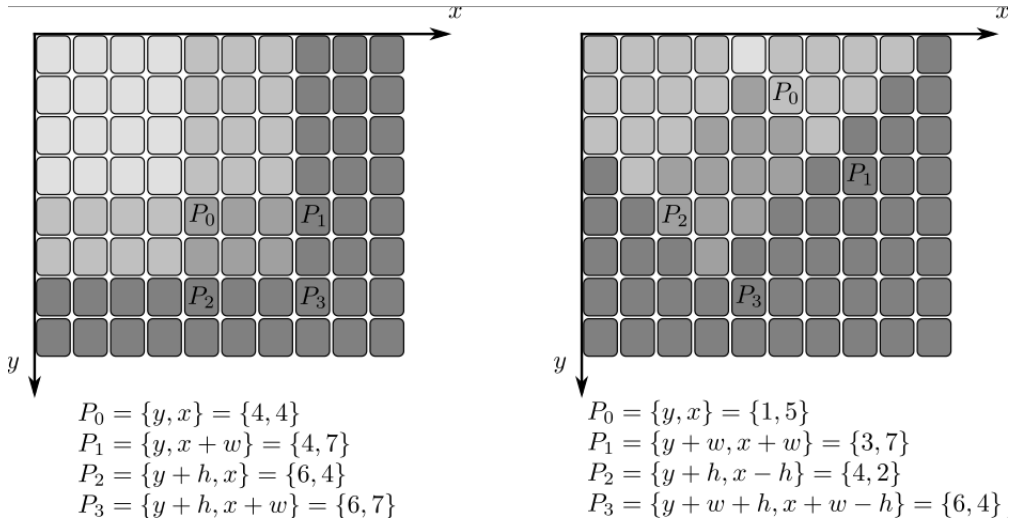
$$\texttt{tilted}(X,Y) = \sum_{y<Y, abs(x-X+1)\leq Y-y-1} \texttt{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum_{x_1\leq x<x_2,\, y_1\leq y<y_2} \texttt{image}(x,y) = \texttt{sum}(x_2,y_2) - \texttt{sum}(x_1,y_2) - \texttt{sum}(x_2,y_1) + \texttt{sum}(x_1,x_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

As a practical example, the next figure shows the calculation of the integral of a straight rectangle `Rect(3,3,3,2)` and of a tilted rectangle `Rect(5,1,2,3)`. The selected pixels in the original `image` are shown, as well as the relative pixels in the integral images `sum` and `tilted`.



---

cv::threshold (view/add comments)

Applies a fixed-level threshold to each array element

```
double threshold( const Mat& src, Mat& dst, double thresh,
          double maxVal, int thresholdType );
```

src Source array (single-channel, 8-bit of 32-bit floating point)

dst Destination array; will have the same size and the same type as `src`

thresh Threshold value

maxVal Maximum value to use with `THRESH_BINARY` and `THRESH_BINARY_INV` thresholding types

thresholdType Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (cv::compare could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by `thresholdType`:

THRESH_BINARY

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxVal} & \text{otherwise} \end{cases}$$

THRESH_TRUNC

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

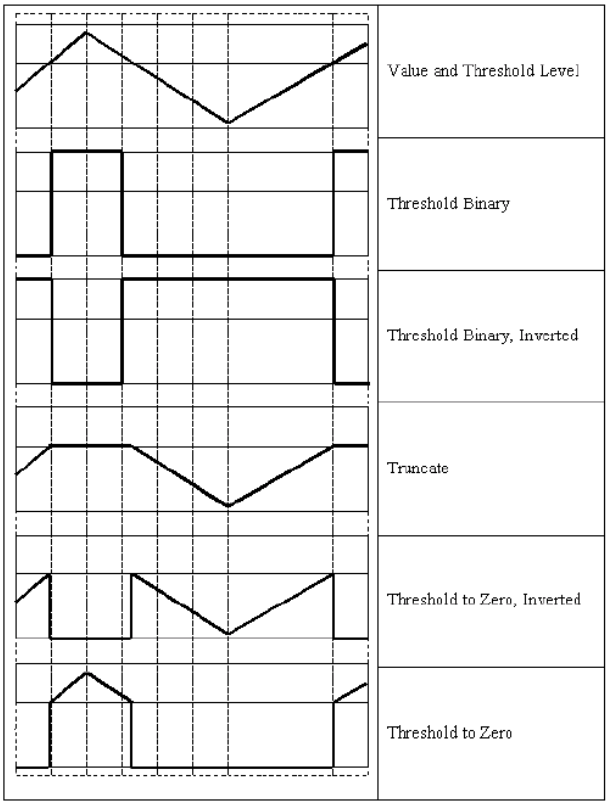
THRESH_TOZERO

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

THRESH_TOZERO_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Also, the special value `THRESH_OTSU` may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified `thresh`. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.



See also: cv::adaptiveThreshold, cv::findContours, cv::compare, cv::min, cv::max

---

cv::watershed (view/add comments)

Does marker-based image segmentation using watershed algrorithm

```
void watershed( const Mat& image, Mat& markers );
```

image The input 8-bit 3-channel image.

markers The input/output 32-bit single-channel image (map) of markers. It should have the same size as `image`

The function implements one of the variants of watershed, non-parametric marker-based segmentation algorithm, described in [**?**]. Before passing the image to the function, user has to outline

roughly the desired regions in the image `markers` with positive (> 0) indices, i.e. every region is represented as one or more connected components with the pixel values 1, 2, 3 etc (such markers can be retrieved from a binary mask using cv::findContoursand cv::drawContours, see `watershed.cpp` demo). The markers will be ”seeds” of the future image regions. All the other pixels in `markers`, which relation to the outlined regions is not known and should be defined by the algorithm, should be set to 0's. On the output of the function, each pixel in markers is set to one of values of the ”seed” components, or to -1 at boundaries between the regions.

Note, that it is not necessary that every two neighbor connected components are separated by a watershed boundary (-1's pixels), for example, in case when such tangent components exist in the initial marker image. Visual demonstration and usage example of the function can be found in OpenCV samples directory; see `watershed.cpp` demo.

See also: cv::findContours

---

cv::grabCut (view/add comments)

Runs GrabCut algorithm

```
void grabCut(const Mat& image, Mat& mask, Rect rect,
             Mat& bgdModel, Mat& fgdModel,
             int iterCount, int mode );
```

image The input 8-bit 3-channel image.

mask The input/output 8-bit single-channel mask. Its elements may have one of four values. The mask is initialize when `mode==GC_INIT_WITH_RECT`

- **GC_BGD** Certainly a background pixel
- **GC_FGD** Certainly a foreground (object) pixel
- **GC_PR_BGD** Likely a background pixel
- **GC_PR_BGD** Likely a foreground pixel

rect The ROI containing the segmented object. The pixels outside of the ROI are marked as ”certainly a background”. The parameter is only used when `mode==GC_INIT_WITH_RECT`

bgdModel, fgdModel Temporary arrays used for segmentation. Do not modify them while you are processing the same image

iterCount The number of iterations the algorithm should do before returning the result. Note that the result can be refined with further calls with the `mode==GC_INIT_WITH_MASK` or `mode==GC_EVAL`

mode The operation mode

- **GC_INIT_WITH_RECT** The function initializes the state and the mask using the provided rectangle. After that it runs `iterCount` iterations of the algorithm
- **GC_INIT_WITH_MASK** The function initializes the state using the provided mask. Note that `GC_INIT_WITH_RECT` and `GC_INIT_WITH_MASK` can be combined, then all the pixels outside of the ROI are automatically initialized with `GC_BGD` .
- **GC_EVAL** The value means that algorithm should just resume.

The function implements the GrabCut image segmentation algorithm. See the sample grabcut.cpp on how to use the function.

12.5 Structural Analysis and Shape Descriptors

---

cv::moments (view/add comments)

Calculates all of the moments up to the third order of a polygon or rasterized shape.

```
Moments moments( const Mat& array, bool binaryImage=false );
```

where the class `Moments` is defined as:

```
class Moments
{
public:
    Moments();
    Moments(double m00, double m10, double m01, double m20, double m11,
            double m02, double m30, double m21, double m12, double m03 );
    Moments( const CvMoments\& moments );
    operator CvMoments() const;

    // spatial moments
    double  m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;
    // central moments
    double  mu20, mu11, mu02, mu30, mu21, mu12, mu03;
```

```
    // central normalized moments
    double  nu20, nu11, nu02, nu30, nu21, nu12, nu03;
};
```

array A raster image (single-channel, 8-bit or floating-point 2D array) or an array ($1 \times N$ or $N \times 1$) of 2D points (`Point` or `Point2f`)

binaryImage (For images only) If it is true, then all the non-zero image pixels are treated as 1's

The function computes moments, up to the 3rd order, of a vector shape or a rasterized shape. In case of a raster image, the spatial moments `Moments::m`$_{ji}$ are computed as:

$$\texttt{m}_{ji} = \sum_{x,y} \left(\texttt{array}(x,y) \cdot x^j \cdot y^i \right),$$

the central moments `Moments::mu`$_{ji}$ are computed as:

$$\texttt{mu}_{ji} = \sum_{x,y} \left(\texttt{array}(x,y) \cdot (x - \bar{x})^j \cdot (y - \bar{y})^i \right)$$

where $(\bar{x}, \bar{y})$ is the mass center:

$$\bar{x} = \frac{\texttt{m}_{10}}{\texttt{m}_{00}}, \; \bar{y} = \frac{\texttt{m}_{01}}{\texttt{m}_{00}}$$

and the normalized central moments `Moments::nu`$_{ij}$ are computed as:

$$\texttt{nu}_{ji} = \frac{\texttt{mu}_{ji}}{\texttt{m}_{00}^{(i+j)/2+1}}.$$

Note that $\texttt{mu}_{00} = \texttt{m}_{00}$, $\texttt{nu}_{00} = 1$ $\texttt{nu}_{10} = \texttt{mu}_{10} = \texttt{mu}_{01} = \texttt{mu}_{10} = 0$, hence the values are not stored.

The moments of a contour are defined in the same way, but computed using Green's formula (see http://en.wikipedia.org/wiki/Green_theorem), therefore, because of a limited raster resolution, the moments computed for a contour will be slightly different from the moments computed for the same contour rasterized.

See also: cv::contourArea, cv::arcLength

---

cv::HuMoments (view/add comments)

Calculates the seven Hu invariants.

```
void HuMoments( const Moments& moments, double h[7] );
```

moments The input moments, computed with cv::moments

h The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment, that are defined as:

$$
\begin{aligned}
h[0] &= \eta_{20} + \eta_{02} \\
h[1] &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
h[2] &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
h[3] &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
h[4] &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
h[5] &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
h[6] &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}
$$

where η_{ji} stand for `Moments::nu`$_{ji}$.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

See also: cv::matchShapes

---

cv::findContours (view/add comments)

Finds the contours in a binary image.

```
void findContours( const Mat& image, vector<vector<Point> >& contours,
            vector<Vec4i>& hierarchy, int mode,
            int method, Point offset=Point());
void findContours( const Mat& image, vector<vector<Point> >& contours,
            int mode, int method, Point offset=Point());
```

image The source, an 8-bit single-channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary`. You can use cv::compare, cv::inRange, cv::threshold, cv::adaptiveThreshold, cv::Canny etc. to create a binary image out of a grayscale or color one. The function modifies the `image` while extracting the contours

contours The detected contours. Each contour is stored as a vector of points

hiararchy The optional output vector that will contain information about the image topology. It will have as many elements as the number of contours. For each contour `contours[i]`, the elements `hierarchy[i][0]`, `hiearchy[i][1]`, `hiearchy[i][2]`, `hiearchy[i][3]` will be set to 0-based indices in `contours` of the next and previous contours at the same hierarchical level, the first child contour and the parent contour, respectively. If for some contour `i` there is no next, previous, parent or nested contours, the corresponding elements of `hierarchy[i]` will be negative

mode The contour retrieval mode

CV_RETR_EXTERNAL retrieves only the extreme outer contours; It will set `hierarchy[i][2]=hierarchy` for all the contours

CV_RETR_LIST retrieves all of the contours without establishing any hierarchical relationships

CV_RETR_CCOMP retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes. If inside a hole of a connected component there is another contour, it will still be put on the top level

CV_RETR_TREE retrieves all of the contours and reconstructs the full hierarchy of nested contours. This full hierarchy is built and shown in OpenCV `contours.c` demo

method The contour approximation method.

CV_CHAIN_APPROX_NONE stores absolutely all the contour points. That is, every 2 points of a contour stored with this method are 8-connected neighbors of each other

CV_CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and diagonal segments and leaves only their end points. E.g. an up-right rectangular contour will be encoded with 4 points

CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS applies one of the flavors of the Teh-Chin chain approximation algorithm; see [**?**]

offset The optional offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context

The function retrieves contours from the binary image using the algorithm [**?**]. The contours are a useful tool for shape analysis and object detection and recognition. See `squares.c` in the OpenCV sample directory.

Note: the source `image` is modified by this function.

---

cv::drawContours (view/add comments)

Draws contours' outlines or filled contours.

```
void drawContours( Mat& image, const vector<vector<Point> >& contours,
           int contourIdx, const Scalar& color, int thickness=1,
           int lineType=8, const vector<Vec4i>& hierarchy=vector<Vec4i>(),
           int maxLevel=INT_MAX, Point offset=Point() );
```

`image` The destination image

`contours` All the input contours. Each contour is stored as a point vector

`contourIdx` Indicates the contour to draw. If it is negative, all the contours are drawn

`color` The contours' color

`thickness` Thickness of lines the contours are drawn with. If it is negative (e.g. `thickness=CV_FILLED`), the contour interiors are drawn.

`lineType` The line connectivity; see cv::line description

`hierarchy` The optional information about hierarchy. It is only needed if you want to draw only some of the contours (see `maxLevel`)

`maxLevel` Maximal level for drawn contours. If 0, only the specified contour is drawn. If 1, the function draws the contour(s) and all the nested contours. If 2, the function draws the contours, all the nested contours and all the nested into nested contours etc. This parameter is only taken into account when there is `hierarchy` available.

`offset` The optional contour shift parameter. Shift all the drawn contours by the specified $\texttt{offset} = (dx, dy)$

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$. Here is the example on how to retrieve connected components from the binary image and label them

```
#include "cv.h"
#include "highgui.h"

using namespace cv;

int main( int argc, char** argv )
{
    Mat src;
    // the first command line parameter must be file name of binary
    // (black-n-white) image
    if( argc != 2 || !(src=imread(argv[1], 0)).data)
        return -1;

    Mat dst = Mat::zeros(src.rows, src.cols, CV_8UC3);

    src = src > 1;
    namedWindow( "Source", 1 );
    imshow( "Source", src );

    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    findContours( src, contours, hierarchy,
        CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );

    // iterate through all the top-level contours,
    // draw each connected component with its own random color
    int idx = 0;
    for( ; idx >= 0; idx = hierarchy[idx][0] )
    {
        Scalar color( rand()&255, rand()&255, rand()&255 );
        drawContours( dst, contours, idx, color, CV_FILLED, 8, hierarchy );
    }

    namedWindow( "Components", 1 );
    imshow( "Components", dst );
    waitKey(0);
}
```

---

cv::approxPolyDP (view/add comments)

Approximates polygonal curve(s) with the specified precision.

```
void approxPolyDP( const Mat& curve,
           vector<Point>& approxCurve,
           double epsilon, bool closed );
void approxPolyDP( const Mat& curve,
           vector<Point2f>& approxCurve,
           double epsilon, bool closed );
```

curve The polygon or curve to approximate. Must be $1 \times N$ or $N \times 1$ matrix of type `CV_32SC2` or `CV_32FC2`. You can also convert `vector<Point>` or `vector<Point2f` to the matrix by calling `Mat(const vector<T>&)` constructor.

approxCurve The result of the approximation; The type should match the type of the input curve

epsilon Specifies the approximation accuracy. This is the maximum distance between the original curve and its approximation

closed If true, the approximated curve is closed (i.e. its first and last vertices are connected), otherwise it's not

The functions `approxPolyDP` approximate a curve or a polygon with another curve/polygon with less vertices, so that the distance between them is less or equal to the specified precision. It used Douglas-Peucker algorithm http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm

---

cv::arcLength (view/add comments)

Calculates a contour perimeter or a curve length.

```
double arcLength( const Mat& curve, bool closed );
```

curve The input vector of 2D points, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to a matrix with `Mat(const vector<T>&)` constructor

closed Indicates, whether the curve is closed or not

The function computes the curve length or the closed contour perimeter.

cv::boundingRect (view/add comments)

Calculates the up-right bounding rectangle of a point set.

```
Rect boundingRect( const Mat& points );
```

points The input 2D point set, represented by CV_32SC2 or CV_32FC2 matrix, or by vector<Point> or vector<Point2f> converted to the matrix using Mat(const vector<T>&) constructor.

The function calculates and returns the minimal up-right bounding rectangle for the specified point set.

cv::estimateRigidTransform (view/add comments)

Computes optimal affine transformation between two 2D point sets

```
Mat estimateRigidTransform( const Mat& srcpt, const Mat& dstpt,
            bool fullAffine );
```

srcpt The first input 2D point set

dst The second input 2D point set of the same size and the same type as A

fullAffine If true, the function finds the optimal affine transformation with no any additional resrictions (i.e. there are 6 degrees of freedom); otherwise, the class of transformations to choose from is limited to combinations of translation, rotation and uniform scaling (i.e. there are 5 degrees of freedom)

The function finds the optimal affine transform $[A|b]$ (a 2×3 floating-point matrix) that approximates best the transformation from $\texttt{srcpt}_i$ to $\texttt{dstpt}_i$:

$$[A^*|b^*] = arg \min_{[A|b]} \sum_i \|\texttt{dstpt}_i - A{\texttt{srcpt}_i}^T - b\|^2$$

where $[A|b]$ can be either arbitrary (when `fullAffine=true`) or have form

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ -a_{12} & a_{11} & b_2 \end{bmatrix}$$

when `fullAffine=false`.

See also: cv::getAffineTransform, cv::getPerspectiveTransform, cv::findHomography

---

cv::estimateAffine3D (view/add comments)

Computes optimal affine transformation between two 3D point sets

```
int estimateAffine3D(const Mat& srcpt, const Mat& dstpt, Mat& out,
           vector<uchar>& outliers,
           double ransacThreshold = 3.0,
           double confidence = 0.99);
```

srcpt The first input 3D point set

dstpt The second input 3D point set

out The output 3D affine transformation matrix 3×4

outliers The output vector indicating which points are outliers

ransacThreshold The maximum reprojection error in RANSAC algorithm to consider a point an inlier

confidence The confidence level, between 0 and 1, with which the matrix is estimated

The function estimates the optimal 3D affine transformation between two 3D point sets using RANSAC algorithm.

---

cv::contourArea (view/add comments)

Calculates the contour area

```
double contourArea( const Mat& contour );
```

contour The contour vertices, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

The function computes the contour area. Similarly to cv::moments the area is computed using the Green formula, thus the returned area and the number of non-zero pixels, if you draw the contour using cv::drawContours or cv::fillPoly, can be different. Here is a short example:

```
vector<Point> contour;
contour.push_back(Point2f(0, 0));
contour.push_back(Point2f(10, 0));
contour.push_back(Point2f(10, 10));
contour.push_back(Point2f(5, 4));

double area0 = contourArea(contour);
vector<Point> approx;
approxPolyDP(contour, approx, 5, true);
double area1 = contourArea(approx);

cout << "area0 =" << area0 << endl <<
        "area1 =" << area1 << endl <<
        "approx poly vertices" << approx.size() << endl;
```

---

cv::convexHull (view/add comments)

Finds the convex hull of a point set.

```
void convexHull( const Mat& points, vector<int>& hull,
           bool clockwise=false );
void convexHull( const Mat& points, vector<Point>& hull,
           bool clockwise=false );
void convexHull( const Mat& points, vector<Point2f>& hull,
           bool clockwise=false );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

hull The output convex hull. It is either a vector of points that form the hull (must have the same type as the input points), or a vector of 0-based point indices of the hull points in the original array (since the set of convex hull points is a subset of the original point set).

clockwise If true, the output convex hull will be oriented clockwise, otherwise it will be oriented counter-clockwise. Here, the usual screen coordinate system is assumed - the origin is at the top-left corner, x axis is oriented to the right, and y axis is oriented downwards.

The functions find the convex hull of a 2D point set using Sklansky's algorithm [**?**] that has $O(NlogN)$ or $O(N)$ complexity (where N is the number of input points), depending on how the initial sorting is implemented (currently it is $O(NlogN)$. See the OpenCV sample `convexhull.c` that demonstrates the use of the different function variants.

---

cv::fitEllipse (view/add comments)

Fits an ellipse around a set of 2D points.

```
RotatedRect fitEllipse( const Mat& points );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

The function calculates the ellipse that fits best (in least-squares sense) a set of 2D points. It returns the rotated rectangle in which the ellipse is inscribed.

---

cv::fitLine (view/add comments)

Fits a line to a 2D or 3D point set.

```
void fitLine( const Mat& points, Vec4f& line, int distType,
           double param, double reps, double aeps );
void fitLine( const Mat& points, Vec6f& line, int distType,
           double param, double reps, double aeps );
```

points The input 2D point set, represented by CV_32SC2 or CV_32FC2 matrix, or by vector<Point>, vector<Point2f>, vector<Point3i> or vector<Point3f> converted to the matrix by Mat(const vector<T>&) constructor

line The output line parameters. In the case of a 2d fitting, it is a vector of 4 floats (vx, vy, x0, y0) where (vx, vy) is a normalized vector collinear to the line and (x0, y0) is some point on the line. in the case of a 3D fitting it is vector of 6 floats (vx, vy, vz, x0, y0, z0) where (vx, vy, vz) is a normalized vector collinear to the line and (x0, y0, z0) is some point on the line

distType The distance used by the M-estimator (see the discussion)

param Numerical parameter (C) for some types of distances, if 0 then some optimal value is chosen

reps, aeps Sufficient accuracy for the radius (distance between the coordinate origin and the line) and angle, respectively; 0.01 would be a good default value for both.

The functions fitLine fit a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i^{th} point and the line and $\rho(r)$ is a distance function, one of:

distType=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

distType=CV_DIST_L1

$$\rho(r) = r$$

distType=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

distType=CV_DIST_FAIR

$$\rho\left(r\right) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right) \quad \text{where} \quad C = 1.3998$$

distType=CV_DIST_WELSCH

$$\rho\left(r\right) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

distType=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

The algorithm is based on the M-estimator (http://en.wikipedia.org/wiki/M-estimator) technique, that iteratively fits the line using weighted least-squares algorithm and after each iteration the weights w_i are adjusted to beinversely proportional to $\rho(r_i)$.

---

cv::isContourConvex (view/add comments)

Tests contour convexity.

```
bool isContourConvex( const Mat& contour );
```

contour The tested contour, a matrix of type CV_32SC2 or CV_32FC2, or vector<Point> or vector<Point2f> converted to the matrix using Mat(const vector<T>&) constructor.

The function tests whether the input contour is convex or not. The contour must be simple, i.e. without self-intersections, otherwise the function output is undefined.

---

cv::minAreaRect (view/add comments)

Finds the minimum area rotated rectangle enclosing a 2D point set.

```
RotatedRect minAreaRect( const Mat& points );
```

points The input 2D point set, represented by CV_32SC2 or CV_32FC2 matrix, or by vector<Point> or vector<Point2f> converted to the matrix using Mat(const vector<T>&) constructor.

The function calculates and returns the minimum area bounding rectangle (possibly rotated) for the specified point set. See the OpenCV sample minarea.c

cv::minEnclosingCircle (view/add comments)

Finds the minimum area circle enclosing a 2D point set.

```
void minEnclosingCircle( const Mat& points, Point2f& center, float&
radius );
```

points The input 2D point set, represented by CV_32SC2 or CV_32FC2 matrix, or by vector<Point> or vector<Point2f> converted to the matrix using Mat(const vector<T>&) constructor.

center The output center of the circle

radius The output radius of the circle

The function finds the minimal enclosing circle of a 2D point set using iterative algorithm. See the OpenCV sample minarea.c

cv::matchShapes (view/add comments)

Compares two shapes.

```
double matchShapes( const Mat& object1,
           const Mat& object2,
           int method, double parameter=0 );
```

object1 The first contour or grayscale image

object2 The second contour or grayscale image

method Comparison method: CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3 (see the discussion below)

parameter Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu invariants (see cv::HuMoments) as following (A denotes `object1`, B denotes `object2`):

method=CV_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1\ldots7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1\ldots7} \left| m_i^A - m_i^B \right|$$

method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1\ldots7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

where

$$m_i^A = \mathrm{sign}(h_i^A) \cdot \log h_i^A$$
$$m_i^B = \mathrm{sign}(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

---

cv::pointPolygonTest (view/add comments)

Performs point-in-contour test.

```
double pointPolygonTest( const Mat& contour,
           Point2f pt, bool measureDist );
```

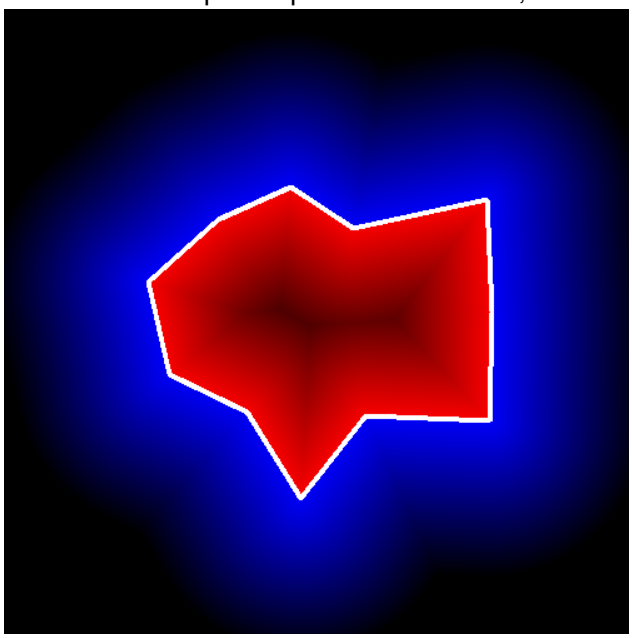
contour The input contour

pt The point tested against the contour

measureDist If true, the function estimates the signed distance from the point to the nearest contour edge; otherwise, the function only checks if the point is inside or not.

The function determines whether the point is inside a contour, outside, or lies on an edge (or coincides with a vertex). It returns positive (inside), negative (outside) or zero (on an edge) value, correspondingly. When `measureDist=false`, the return value is +1, -1 and 0, respectively. Otherwise, the return value it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



12.6 Planar Subdivisions

12.7 Motion Analysis and Object Tracking

---

cv::accumulate (view/add comments)

Adds image to the accumulator.

```
void accumulate( const Mat& src, Mat& dst, const Mat& mask=Mat() );
```

src The input image, 1- or 3-channel, 8-bit or 32-bit floating point

dst The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds `src`, or some of its elements, to `dst`:

$$\texttt{dst}(x,y) \leftarrow \texttt{dst}(x,y) + \texttt{src}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.

The functions `accumulate*` can be used, for example, to collect statistic of background of a scene, viewed by a still camera, for the further foreground-background segmentation.

See also: cv::accumulateSquare, cv::accumulateProduct, cv::accumulateWeighted

---

cv::accumulateSquare (view/add comments)

Adds the square of the source image to the accumulator.

```
void accumulateSquare( const Mat& src, Mat& dst,
           const Mat& mask=Mat() );
```

src The input image, 1- or 3-channel, 8-bit or 32-bit floating point

dst The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the input image `src` or its selected region, raised to power 2, to the accumulator `dst`:

$$\texttt{dst}(x,y) \leftarrow \texttt{dst}(x,y) + \texttt{src}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.

See also: cv::accumulateSquare, cv::accumulateProduct, cv::accumulateWeighted

cv::accumulateProduct (view/add comments)

Adds the per-element product of two input images to the accumulator.

```
void accumulateProduct( const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat() );
```

src1 The first input image, 1- or 3-channel, 8-bit or 32-bit floating point

src2 The second input image of the same type and the same size as `src1`

dst Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `dst`:

$$\texttt{dst}(x,y) \leftarrow \texttt{dst}(x,y) + \texttt{src1}(x,y) \cdot \texttt{src2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.
See also: cv::accumulate, cv::accumulateSquare, cv::accumulateWeighted

cv::accumulateWeighted (view/add comments)

Updates the running average.

```
void accumulateWeighted( const Mat& src, Mat& dst,
           double alpha, const Mat& mask=Mat() );
```

src The input image, 1- or 3-channel, 8-bit or 32-bit floating point

dst The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point

alpha Weight of the input image

mask Optional operation mask

The function calculates the weighted sum of the input image `src` and the accumulator `dst` so that `dst` becomes a running average of frame sequence:

$$\texttt{dst}(x,y) \leftarrow (1 - \texttt{alpha}) \cdot \texttt{dst}(x,y) + \texttt{alpha} \cdot \texttt{src}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

that is, `alpha` regulates the update speed (how fast the accumulator "forgets" about earlier images). The function supports multi-channel images; each channel is processed independently.

See also: cv::accumulate, cv::accumulateSquare, cv::accumulateProduct

12.8 Feature Detection

---

cv::Canny (view/add comments)

Finds edges in an image using Canny algorithm.

```
void Canny( const Mat& image, Mat& edges,
            double threshold1, double threshold2,
            int apertureSize=3, bool L2gradient=false );
```

image Single-channel 8-bit input image

edges The output edge map. It will have the same size and the same type as `image`

threshold1 The first threshold for the hysteresis procedure

threshold2 The second threshold for the hysteresis procedure

apertureSize Aperture size for the cv::Sobel operator

L2gradient Indicates, whether the more accurate L_2 norm $= \sqrt{(dI/dx)^2 + (dI/dy)^2}$ should be used to compute the image gradient magnitude (`L2gradient=true`), or a faster default L_1 norm $= |dI/dx| + |dI/dy|$ is enough (`L2gradient=false`)

The function finds edges in the input image `image` and marks them in the output map `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges, see http://en.wikipedia.org/wiki/Canny_edge_detector

cv::cornerEigenValsAndVecs (view/add comments)

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
void cornerEigenValsAndVecs( const Mat& src, Mat& dst,
           int blockSize, int apertureSize,
           int borderType=BORDER_DEFAULT );
```

src Input single-channel 8-bit or floating-point image

dst Image to store the results. It will have the same size as `src` and the type `CV_32FC(6)`

blockSize Neighborhood size (see discussion)

apertureSize Aperture parameter for the cv::Sobel operator

boderType Pixel extrapolation method; see cv::borderInterpolate

For every pixel p, the function `cornerEigenValsAndVecs` considers a `blockSize` $\times$ `blockSize` neigborhood $S(p)$. It calculates the covariation matrix of derivatives over the neighborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dxdI/dy)^2 \\ \sum_{S(p)}(dI/dxdI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

Where the derivatives are computed using cv::Sobel operator.

After that it finds eigenvectors and eigenvalues of M and stores them into destination image in the form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the eigenvalues of M; not sorted

x_1, y_1 are the eigenvectors corresponding to λ_1

x_2, y_2 are the eigenvectors corresponding to λ_2

The output of the function can be used for robust edge or corner detection.

See also: cv::cornerMinEigenVal, cv::cornerHarris, cv::preCornerDetect

cv::cornerHarris (view/add comments)

Harris edge detector.

```
void cornerHarris( const Mat& src, Mat& dst, int blockSize,
           int apertureSize, double k,
           int borderType=BORDER_DEFAULT );
```

src Input single-channel 8-bit or floating-point image

dst Image to store the Harris detector responses; will have type `CV_32FC1` and the same size as `src`

blockSize Neighborhood size (see the discussion of cv::cornerEigenValsAndVecs)

apertureSize Aperture parameter for the cv::Sobel operator

k Harris detector free parameter. See the formula below

boderType Pixel extrapolation method; see cv::borderInterpolate

The function runs the Harris edge detector on the image. Similarly to cv::cornerMinEigenVal and cv::cornerEigenValsAndVecs, for each pixel (x, y) it calculates a 2×2 gradient covariation matrix $M^{(x,y)}$ over a `blockSize` $\times$ `blockSize` neighborhood. Then, it computes the following characteristic:

$$\texttt{dst}(x,y) = \det M^{(x,y)} - k \cdot \left(\operatorname{tr} M^{(x,y)} \right)^2$$

Corners in the image can be found as the local maxima of this response map.

cv::cornerMinEigenVal (view/add comments)

Calculates the minimal eigenvalue of gradient matrices for corner detection.

```
void cornerMinEigenVal( const Mat& src, Mat& dst,
           int blockSize, int apertureSize=3,
           int borderType=BORDER_DEFAULT );
```

src Input single-channel 8-bit or floating-point image

dst Image to store the minimal eigenvalues; will have type `CV_32FC1` and the same size as `src`

blockSize Neighborhood size (see the discussion of cv::cornerEigenValsAndVecs)

apertureSize Aperture parameter for the cv::Sobel operator

boderType Pixel extrapolation method; see cv::borderInterpolate

The function is similar to cv::cornerEigenValsAndVecs but it calculates and stores only the minimal eigenvalue of the covariation matrix of derivatives, i.e. $\min(\lambda_1, \lambda_2)$ in terms of the formulae in cv::cornerEigenValsAndVecs description.

---

cv::cornerSubPix (view/add comments)

Refines the corner locations.

```
void cornerSubPix( const Mat& image, vector<Point2f>& corners,
           Size winSize, Size zeroZone,
           TermCriteria criteria );
```

image Input image

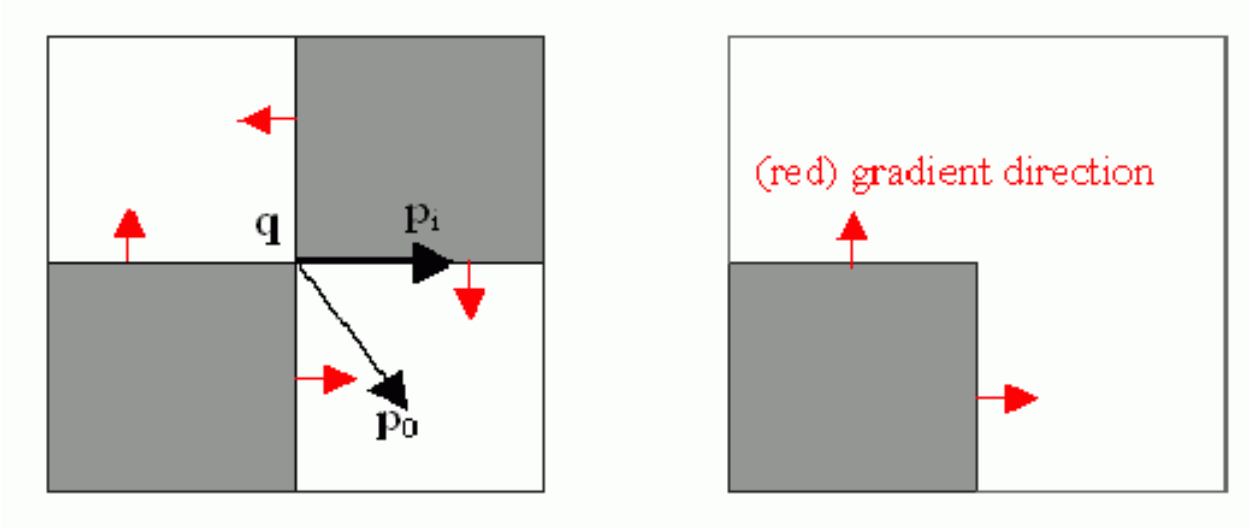
corners Initial coordinates of the input corners; refined coordinates on output

winSize Half of the side length of the search window. For example, if `winSize=Size(5,5)`, then a $5*2+1 \times 5*2+1 = 11 \times 11$ search window would be used

zeroZone Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size

criteria Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below.



Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider the expression:

$$\epsilon_i = {DI_{p_i}}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot {DI_{p_i}}^T) - \sum_i (DI_{p_i} \cdot {DI_{p_i}}^T \cdot p_i)$$

where the gradients are summed within a neighborhood ("search window") of q. Calling the first gradient term G and the second gradient term b gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center q and then iterates until the center keeps within a set threshold.

---

cv::goodFeaturesToTrack (view/add comments)

Determines strong corners on an image.

```
void goodFeaturesToTrack( const Mat& image, vector<Point2f>& corners,
           int maxCorners, double qualityLevel, double minDistance,
           const Mat& mask=Mat(), int blockSize=3,
           bool useHarrisDetector=false, double k=0.04 );
```

image The input 8-bit or floating-point 32-bit, single-channel image

corners The output vector of detected corners

maxCorners The maximum number of corners to return. If there are more corners than that will be found, the strongest of them will be returned

qualityLevel Characterizes the minimal accepted quality of image corners; the value of the parameter is multiplied by the by the best corner quality measure (which is the min eigenvalue, see cv::cornerMinEigenVal, or the Harris function response, see cv::cornerHarris). The corners, which quality measure is less than the product, will be rejected. For example, if the best corner has the quality measure = 1500, and the `qualityLevel=0.01`, then all the corners which quality measure is less than 15 will be rejected.

minDistance The minimum possible Euclidean distance between the returned corners

mask The optional region of interest. If the image is not empty (then it needs to have the type `CV_8UC1` and the same size as `image`), it will specify the region in which the corners are detected

blockSize Size of the averaging block for computing derivative covariation matrix over each pixel neighborhood, see cv::cornerEigenValsAndVecs

useHarrisDetector Indicates, whether to use Harris operator or cv::cornerMinEigenVal

k Free parameter of Harris detector

The function finds the most prominent corners in the image or in the specified image region, as described in [**?**]:

1. the function first calculates the corner quality measure at every source image pixel using the cv::cornerMinEigenVal or cv::cornerHarris
2. then it performs non-maxima suppression (the local maxima in 3×3 neighborhood are retained).

3. the next step rejects the corners with the minimal eigenvalue less than `qualityLevel` · $\max_{x,y} qualityMeasureMap(x,y)$.

4. the remaining corners are then sorted by the quality measure in the descending order.

5. finally, the function throws away each corner pt_j if there is a stronger corner pt_i ($i < j$) such that the distance between them is less than `minDistance`

The function can be used to initialize a point-based tracker of an object.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel`, and `A` ¿ B, the vector of returned corners with `qualityLevel=A` will be the prefix of the output vector with `qualityLevel=B`.

See also: cv::cornerMinEigenVal, cv::cornerHarris, cv::calcOpticalFlowPyrLK, cv::estimateRigidMotion, cv::PlanarObjectDetector, cv::OneWayDescriptor

---

cv::HoughCircles (view/add comments)

Finds circles in a grayscale image using a Hough transform.

```
void HoughCircles( Mat& image, vector<Vec3f>& circles,
           int method, double dp, double minDist,
           double param1=100, double param2=100,
           int minRadius=0, int maxRadius=0 );
```

image The 8-bit, single-channel, grayscale input image

circles The output vector of found circles. Each vector is encoded as 3-element floating-point vector $(x, y, radius)$

method Currently, the only implemented method is `CV_HOUGH_GRADIENT`, which is basically *21HT*, described in [**?**].

dp The inverse ratio of the accumulator resolution to the image resolution. For example, if `dp=1`, the accumulator will have the same resolution as the input image, if `dp=2` - accumulator will have half as big width and height, etc

minDist Minimum distance between the centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed

param1 The first method-specific parameter. in the case of CV_HOUGH_GRADIENT it is the higher threshold of the two passed to cv::Canny edge detector (the lower one will be twice smaller)

param2 The second method-specific parameter. in the case of CV_HOUGH_GRADIENT it is the accumulator threshold at the center detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first

minRadius Minimum circle radius

maxRadius Maximum circle radius

The function finds circles in a grayscale image using some modification of Hough transform. Here is a short usage example:

```
#include <cv.h>
#include <highgui.h>
#include <math.h>

using namespace cv;

int main(int argc, char** argv)
{
    Mat img, gray;
    if( argc != 2 && !(img=imread(argv[1], 1)).data)
        return -1;
    cvtColor(img, gray, CV_BGR2GRAY);
    // smooth it, otherwise a lot of false circles may be detected
    GaussianBlur( gray, gray, Size(9, 9), 2, 2 );
    vector<Vec3f> circles;
    HoughCircles(gray, circles, CV_HOUGH_GRADIENT,
                 2, gray->rows/4, 200, 100 );
    for( size_t i = 0; i < circles.size(); i++ )
    {
         Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
         int radius = cvRound(circles[i][2]);
         // draw the circle center
         circle( img, center, 3, Scalar(0,255,0), -1, 8, 0 );
         // draw the circle outline
         circle( img, center, radius, Scalar(0,0,255), 3, 8, 0 );
    }
    namedWindow( "circles", 1 );
    imshow( "circles", img );
    return 0;
}
```

Note that usually the function detects the circles' centers well, however it may fail to find the correct radii. You can assist the function by specifying the radius range (`minRadius` and `maxRadius`) if you know it, or you may ignore the returned radius, use only the center and find the correct radius using some additional procedure.

See also: cv::fitEllipse, cv::minEnclosingCircle

---

cv::HoughLines (view/add comments)

Finds lines in a binary image using standard Hough transform.

```
void HoughLines( Mat& image, vector<Vec2f>& lines,
            double rho, double theta, int threshold,
            double srn=0, double stn=0 );
```

image The 8-bit, single-channel, binary source image. The image may be modified by the function

lines The output vector of lines. Each line is represented by a two-element vector (ρ, θ). ρ is the distance from the coordinate origin $(0, 0)$ (top-left corner of the image) and θ is the line rotation angle in radians ($0 \sim$ vertical line, $\pi/2 \sim$ horizontal line)

rho Distance resolution of the accumulator in pixels

theta Angle resolution of the accumulator in radians

threshold The accumulator threshold parameter. Only those lines are returned that get enough votes ($>$ `threshold`)

srn For the multi-scale Hough transform it is the divisor for the distance resolution `rho`. The coarse accumulator distance resolution will be `rho` and the accurate accumulator resolution will be `rho/srn`. If both `srn=0` and `stn=0` then the classical Hough transform is used, otherwise both these parameters should be positive.

stn For the multi-scale Hough transform it is the divisor for the distance resolution `theta`

The function implements standard or standard multi-scale Hough transform algorithm for line detection. See cv::HoughLinesP for the code example.

cv::HoughLinesP (view/add comments)

Finds lines segments in a binary image using probabilistic Hough transform.

```
void HoughLinesP( Mat& image, vector<Vec4i>& lines,
                  double rho, double theta, int threshold,
                  double minLineLength=0, double maxLineGap=0 );
```

image The 8-bit, single-channel, binary source image. The image may be modified by the function

lines The output vector of lines. Each line is represented by a 4-element vector (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) are the ending points of each line segment detected.

rho Distance resolution of the accumulator in pixels

theta Angle resolution of the accumulator in radians

threshold The accumulator threshold parameter. Only those lines are returned that get enough votes ($>$ `threshold`)

minLineLength The minimum line length. Line segments shorter than that will be rejected

maxLineGap The maximum allowed gap between points on the same line to link them.

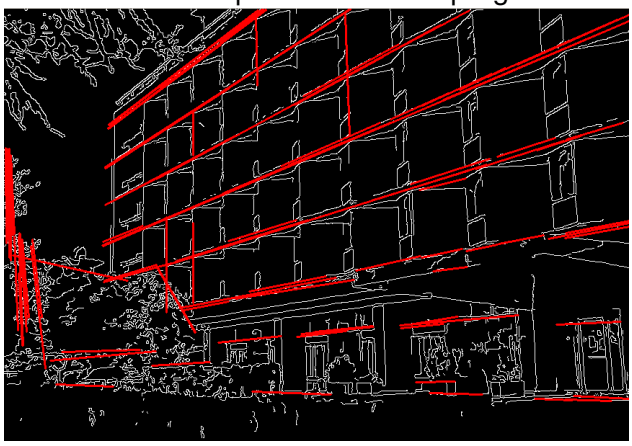
The function implements probabilistic Hough transform algorithm for line detection, described in [**?**]. Below is line detection example:

```
/* This is a standalone program. Pass an image name as a first parameter
of the program.  Switch between standard and probabilistic Hough transform
by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

using namespace cv;

int main(int argc, char** argv)
{
    Mat src, dst, color_dst;
    if( argc != 2 || !(src=imread(argv[1], 0)).data)
```

```
        return -1;

    Canny( src, dst, 50, 200, 3 );
    cvtColor( dst, color_dst, CV_GRAY2BGR );

#if 0
    vector<Vec2f> lines;
    HoughLines( dst, lines, 1, CV_PI/180, 100 );

    for( size_t i = 0; i < lines.size(); i++ )
    {
        float rho = lines[i][0];
        float theta = lines[i][1];
        double a = cos(theta), b = sin(theta);
        double x0 = a*rho, y0 = b*rho;
        Point pt1(cvRound(x0 + 1000*(-b)),
                  cvRound(y0 + 1000*(a)));
        Point pt2(cvRound(x0 - 1000*(-b)),
                  cvRound(y0 - 1000*(a)));
        line( color_dst, pt1, pt2, Scalar(0,0,255), 3, 8 );
    }
#else
    vector<Vec4i> lines;
    HoughLinesP( dst, lines, 1, CV_PI/180, 80, 30, 10 );
    for( size_t i = 0; i < lines.size(); i++ )
    {
        line( color_dst, Point(lines[i][0], lines[i][1]),
            Point(lines[i][2], lines[i][3]), Scalar(0,0,255), 3, 8 );
    }
#endif
    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "Detected Lines", 1 );
    imshow( "Detected Lines", color_dst );

    waitKey(0);
    return 0;
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in the case of probabilistic Hough transform



---

cv::preCornerDetect (view/add comments)

Calculates the feature map for corner detection

```
void preCornerDetect( const Mat& src, Mat& dst, int apertureSize,
            int borderType=BORDER_DEFAULT );
```

src The source single-channel 8-bit of floating-point image

dst The output image; will have type `CV_32F` and the same size as `src`

apertureSize Aperture size of cv::Sobel

borderType The pixel extrapolation method; see cv::borderInterpolate

The function calculates the complex spatial derivative-based function of the source image

$$\texttt{dst} = (D_x \texttt{src})^2 \cdot D_{yy} \texttt{src} + (D_y \texttt{src})^2 \cdot D_{xx} \texttt{src} - 2 D_x \texttt{src} \cdot D_y \texttt{src} \cdot D_{xy} \texttt{src}$$

where D_x, D_y are the first image derivatives, D_{xx}, D_{yy} are the second image derivatives and D_{xy} is the mixed derivative.

The corners can be found as local maximums of the functions, as shown below:

```
Mat corners, dilated_corners;
preCornerDetect(image, corners, 3);
// dilation with 3x3 rectangular structuring element
dilate(corners, dilated_corners, Mat(), 1);
Mat corner_mask = corners == dilated_corners;
```

12.9 Object Detection

---

cv::matchTemplate (view/add comments)

Compares a template against overlapped image regions.

```
void matchTemplate( const Mat& image, const Mat& templ,
           Mat& result, int method );
```

image Image where the search is running; should be 8-bit or 32-bit floating-point

templ Searched template; must be not greater than the source image and have the same data type

result A map of comparison results; will be single-channel 32-bit floating-point. If `image` is $W \times H$ and `templ` is $w \times h$ then `result` will be $(W - w + 1) \times (H - h + 1)$

method Specifies the comparison method (see below)

The function slides through `image`, compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result`. Here are the formulas for the available comparison methods (I denotes `image`, T `template`, R `result`). The summation is done over template and/or the image patch: $x' = 0...w - 1, y' = 0...h - 1$

method=CV_TM_SQDIFF

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

method=CV_TM_SQDIFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCORR

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I(x+x',y+y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

After the function finishes the comparison, the best matches can be found as global minimums (when `CV_TM_SQDIFF` was used) or maximums (when `CV_TM_CCORR` or `CV_TM_CCOEFF` was used) using the cv::minMaxLoc function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel). That is, the function can take a color template and a color image; the result will still be a single-channel image, which is easier to analyze.

Chapter 13

features2d. Feature Detection and Descriptor Extraction

13.1 Feature detection and description

---

cv::FAST (view/add comments)

Detects corners using FAST algorithm by E. Rosten (”Machine learning for high-speed corner detection”, 2006).

```
void FAST( const Mat& image, vector<KeyPoint>& keypoints, int
threshold, bool nonmaxSupression=true );
```

image The image. Keypoints (corners) will be detected on this.

keypoints Keypoints detected on the image.

threshold Threshold on difference between intensity of center pixel and pixels on circle around this pixel. See description of the algorithm.

nonmaxSupression If it is true then non-maximum supression will be applied to detected corners (keypoints).

---

cv::MSER (view/add comments)

Maximally-Stable Extremal Region Extractor

```
class MSER : public CvMSERParams
{
public:
    // default constructor
    MSER();
    // constructor that initializes all the algorithm parameters
    MSER( int _delta, int _min_area, int _max_area,
          float _max_variation, float _min_diversity,
          int _max_evolution, double _area_threshold,
          double _min_margin, int _edge_blur_size );
    // runs the extractor on the specified image; returns the MSERs,
    // each encoded as a contour (vector<Point>, see findContours)
    // the optional mask marks the area where MSERs are searched for
    void operator()( const Mat& image, vector<vector<Point> >& msers, const Mat& mask ) co
};
```

The class encapsulates all the parameters of MSER (see http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions) extraction algorithm.

---

cv::StarDetector (view/add comments)

Implements Star keypoint detector

```
class StarDetector : CvStarDetectorParams
{
public:
    // default constructor
    StarDetector();
    // the full constructor initialized all the algorithm parameters:
    // maxSize - maximum size of the features. The following
    //      values of the parameter are supported:
    //      4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
    // responseThreshold - threshold for the approximated laplacian,
    //      used to eliminate weak features. The larger it is,
    //      the less features will be retrieved
    // lineThresholdProjected - another threshold for the laplacian to
    //      eliminate edges
    // lineThresholdBinarized - another threshold for the feature
    //      size to eliminate edges.
    // The larger the 2 threshold, the more points you get.
    StarDetector(int maxSize, int responseThreshold,
                 int lineThresholdProjected,
                 int lineThresholdBinarized,
                 int suppressNonmaxSize);
```

```
    // finds keypoints in an image
    void operator()(const Mat& image, vector<KeyPoint>& keypoints) const;
};
```

The class implements a modified version of CenSurE keypoint detector described in [**?**]

---

cv::SIFT (view/add comments)

Class for extracting keypoints and computing descriptors using approach named Scale Invariant Feature Transform (SIFT).

```
class CV_EXPORTS SIFT
{
public:
    struct CommonParams
    {
        static const int DEFAULT_NOCTAVES = 4;
        static const int DEFAULT_NOCTAVE_LAYERS = 3;
        static const int DEFAULT_FIRST_OCTAVE = -1;
        enum{ FIRST_ANGLE = 0, AVERAGE_ANGLE = 1 };

        CommonParams();
        CommonParams( int _nOctaves, int _nOctaveLayers, int _firstOctave,
                                            int _angleMode );
        int nOctaves, nOctaveLayers, firstOctave;
        int angleMode;
    };

    struct DetectorParams
    {
        static double GET_DEFAULT_THRESHOLD()
          { return 0.04 / SIFT::CommonParams::DEFAULT_NOCTAVE_LAYERS / 2.0; }
        static double GET_DEFAULT_EDGE_THRESHOLD() { return 10.0; }

        DetectorParams();
        DetectorParams( double _threshold, double _edgeThreshold );
        double threshold, edgeThreshold;
    };

    struct DescriptorParams
    {
        static double GET_DEFAULT_MAGNIFICATION() { return 3.0; }
        static const bool DEFAULT_IS_NORMALIZE = true;
        static const int DESCRIPTOR_SIZE = 128;
```

```
        DescriptorParams();
        DescriptorParams( double _magnification, bool _isNormalize,
                                                  bool _recalculateAngles );
        double magnification;
        bool isNormalize;
        bool recalculateAngles;
    };

    SIFT();
    //! sift-detector constructor
    SIFT( double _threshold, double _edgeThreshold,
          int _nOctaves=CommonParams::DEFAULT_NOCTAVES,
          int _nOctaveLayers=CommonParams::DEFAULT_NOCTAVE_LAYERS,
          int _firstOctave=CommonParams::DEFAULT_FIRST_OCTAVE,
          int _angleMode=CommonParams::FIRST_ANGLE );
    //! sift-descriptor constructor
    SIFT( double _magnification, bool _isNormalize=true,
          bool _recalculateAngles = true,
          int _nOctaves=CommonParams::DEFAULT_NOCTAVES,
          int _nOctaveLayers=CommonParams::DEFAULT_NOCTAVE_LAYERS,
          int _firstOctave=CommonParams::DEFAULT_FIRST_OCTAVE,
          int _angleMode=CommonParams::FIRST_ANGLE );
    SIFT( const CommonParams& _commParams,
          const DetectorParams& _detectorParams = DetectorParams(),
          const DescriptorParams& _descriptorParams = DescriptorParams() );

    //! returns the descriptor size in floats (128)
    int descriptorSize() const { return DescriptorParams::DESCRIPTOR_SIZE; }
    //! finds the keypoints using SIFT algorithm
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints) const;
    //! finds the keypoints and computes descriptors for them using SIFT algorithm.
    //! Optionally it can compute descriptors for the user-provided keypoints
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints,
                    Mat& descriptors,
                    bool useProvidedKeypoints=false) const;

    CommonParams getCommonParams () const { return commParams; }
    DetectorParams getDetectorParams () const { return detectorParams; }
    DescriptorParams getDescriptorParams () const { return descriptorParams; }
protected:
    ...
};
```

cv::SURF (view/add comments)

Class for extracting Speeded Up Robust Features from an image.

```
class SURF : public CvSURFParams
{
public:
    // default constructor
    SURF();
    // constructor that initializes all the algorithm parameters
    SURF(double _hessianThreshold, int _nOctaves=4,
         int _nOctaveLayers=2, bool _extended=false);
    // returns the number of elements in each descriptor (64 or 128)
    int descriptorSize() const;
    // detects keypoints using fast multi-scale Hessian detector
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints) const;
    // detects keypoints and computes the SURF descriptors for them;
    // output vector "descriptors" stores elements of descriptors and has size
    // equal descriptorSize()*keypoints.size() as each descriptor is
    // descriptorSize() elements of this vector.
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints,
                    vector<float>& descriptors,
                    bool useProvidedKeypoints=false) const;
};
```

The class SURF implements Speeded Up Robust Features descriptor [**?**]. There is fast multi-scale Hessian keypoint detector that can be used to find the keypoints (which is the default option), but the descriptors can be also computed for the user-specified keypoints. The function can be used for object tracking and localization, image stitching etc. See the find_obj.cpp demo in OpenCV samples directory.

cv::RandomizedTree (view/add comments)

The class contains base structure for RTreeClassifier

```
class CV_EXPORTS RandomizedTree
{
public:
        friend class RTreeClassifier;

        RandomizedTree();
        ~RandomizedTree();
```

```
        void train(std::vector<BaseKeypoint> const& base_set,
                cv::RNG &rng, int depth, int views,
                size_t reduced_num_dim, int num_quant_bits);
        void train(std::vector<BaseKeypoint> const& base_set,
                cv::RNG &rng, PatchGenerator &make_patch, int depth,
                int views, size_t reduced_num_dim, int num_quant_bits);

        // following two funcs are EXPERIMENTAL
        //(do not use unless you know exactly what you do)
        static void quantizeVector(float *vec, int dim, int N, float bnds[2],
                int clamp_mode=0);
        static void quantizeVector(float *src, int dim, int N, float bnds[2],
                uchar *dst);

        // patch_data must be a 32x32 array (no row padding)
        float* getPosterior(uchar* patch_data);
        const float* getPosterior(uchar* patch_data) const;
        uchar* getPosterior2(uchar* patch_data);

        void read(const char* file_name, int num_quant_bits);
        void read(std::istream &is, int num_quant_bits);
        void write(const char* file_name) const;
        void write(std::ostream &os) const;

        int classes() { return classes_; }
        int depth() { return depth_; }

        void discardFloatPosteriors() { freePosteriors(1); }

        inline void applyQuantization(int num_quant_bits)
                { makePosteriors2(num_quant_bits); }

private:
        int classes_;
        int depth_;
        int num_leaves_;
        std::vector<RTreeNode> nodes_;
        float **posteriors_;        // 16-bytes aligned posteriors
        uchar **posteriors2_;     // 16-bytes aligned posteriors
        std::vector<int> leaf_counts_;

        void createNodes(int num_nodes, cv::RNG &rng);
        void allocPosteriorsAligned(int num_leaves, int num_classes);
        void freePosteriors(int which);
                // which: 1=posteriors_, 2=posteriors2_, 3=both
```

```
        void init(int classes, int depth, cv::RNG &rng);
        void addExample(int class_id, uchar* patch_data);
        void finalize(size_t reduced_num_dim, int num_quant_bits);
        int getIndex(uchar* patch_data) const;
        inline float* getPosteriorByIndex(int index);
        inline uchar* getPosteriorByIndex2(int index);
        inline const float* getPosteriorByIndex(int index) const;
        void convertPosteriorsToChar();
        void makePosteriors2(int num_quant_bits);
        void compressLeaves(size_t reduced_num_dim);
        void estimateQuantPercForPosteriors(float perc[2]);
};
```

---

cv::RandomizedTree::train (view/add comments)

Trains a randomized tree using input set of keypoints

```
void train(std::vector<BaseKeypoint> const& base_set, cv::RNG &rng,
PatchGenerator &make_patch, int depth, int views, size_t reduced_num_dim,
int num_quant_bits);
```

```
void train(std::vector<BaseKeypoint> const& base_set, cv::RNG &rng,
PatchGenerator &make_patch, int depth, int views, size_t reduced_num_dim,
int num_quant_bits);
```

base_set Vector of `BaseKeypoint` type. Contains keypoints from the image are used for training

rng Random numbers generator is used for training

make_patch Patch generator is used for training

depth Maximum tree depth

reduced_num_dim Number of dimensions are used in compressed signature

num_quant_bits Number of bits are used for quantization

cv::RandomizedTree::read (view/add comments)

Reads pre-saved randomized tree from file or stream

```
read(const char* file_name, int num_quant_bits)
```

```
read(std::istream &is, int num_quant_bits)
```

file_name Filename of file contains randomized tree data

is Input stream associated with file contains randomized tree data

num_quant_bits Number of bits are used for quantization

cv::RandomizedTree::write (view/add comments)

Writes current randomized tree to a file or stream

```
void write(const char* file_name) const;
```

```
void write(std::ostream &os) const;
```

file_name Filename of file where randomized tree data will be stored

is Output stream associated with file where randomized tree data will be stored

cv::RandomizedTree::applyQuantization (view/add comments)

Applies quantization to the current randomized tree

```
void applyQuantization(int num_quant_bits)
```

num_quant_bits Number of bits are used for quantization

cv::RTreeNode (view/add comments)

The class contains base structure for `RandomizedTree`

```
struct RTreeNode
{
        short offset1, offset2;

        RTreeNode() {}

        RTreeNode(uchar x1, uchar y1, uchar x2, uchar y2)
                : offset1(y1*PATCH_SIZE + x1),
                offset2(y2*PATCH_SIZE + x2)
        {}

        //! Left child on 0, right child on 1
        inline bool operator() (uchar* patch_data) const
        {
                return patch_data[offset1] > patch_data[offset2];
        }
};
```

cv::RTreeClassifier (view/add comments)

The class contains `RTreeClassifier`. It represents calonder descriptor which was originally introduced by Michael Calonder

```
class CV_EXPORTS RTreeClassifier
{
public:
        static const int DEFAULT_TREES = 48;
        static const size_t DEFAULT_NUM_QUANT_BITS = 4;
```

```
    RTreeClassifier();

    void train(std::vector<BaseKeypoint> const& base_set,
            cv::RNG &rng,
            int num_trees = RTreeClassifier::DEFAULT_TREES,
            int depth = DEFAULT_DEPTH,
            int views = DEFAULT_VIEWS,
            size_t reduced_num_dim = DEFAULT_REDUCED_NUM_DIM,
            int num_quant_bits = DEFAULT_NUM_QUANT_BITS,
                    bool print_status = true);
    void train(std::vector<BaseKeypoint> const& base_set,
            cv::RNG &rng,
            PatchGenerator &make_patch,
            int num_trees = RTreeClassifier::DEFAULT_TREES,
            int depth = DEFAULT_DEPTH,
            int views = DEFAULT_VIEWS,
            size_t reduced_num_dim = DEFAULT_REDUCED_NUM_DIM,
            int num_quant_bits = DEFAULT_NUM_QUANT_BITS,
             bool print_status = true);

    // sig must point to a memory block of at least
    //classes()*sizeof(float|uchar) bytes
    void getSignature(IplImage *patch, uchar *sig);
    void getSignature(IplImage *patch, float *sig);
    void getSparseSignature(IplImage *patch, float *sig,
             float thresh);

    static int countNonZeroElements(float *vec, int n, double tol=1e-10);
    static inline void safeSignatureAlloc(uchar **sig, int num_sig=1,
                    int sig_len=176);
    static inline uchar* safeSignatureAlloc(int num_sig=1,
                     int sig_len=176);

    inline int classes() { return classes_; }
    inline int original_num_classes()
             { return original_num_classes_; }

    void setQuantization(int num_quant_bits);
    void discardFloatPosteriors();

    void read(const char* file_name);
    void read(std::istream &is);
    void write(const char* file_name) const;
    void write(std::ostream &os) const;
```

```
        std::vector<RandomizedTree> trees_;

private:
        int classes_;
        int num_quant_bits_;
        uchar **posteriors_;
        ushort *ptemp_;
        int original_num_classes_;
        bool keep_floats_;
};
```

---

cv::RTreeClassifier::train (view/add comments)

Trains a randomized tree classificator using input set of keypoints

```
void train(std::vector<BaseKeypoint> const& base_set, cv::RNG
&rng, int num_trees = RTreeClassifier::DEFAULT_TREES, int depth =
DEFAULT_DEPTH, int views = DEFAULT_VIEWS, size_t reduced_num_dim =
DEFAULT_REDUCED_NUM_DIM, int num_quant_bits = DEFAULT_NUM_QUANT_BITS, bool
print_status = true);
```

```
void train(std::vector<BaseKeypoint> const& base_set,
cv::RNG &rng, PatchGenerator &make_patch, int num_trees =
RTreeClassifier::DEFAULT_TREES, int depth = DEFAULT_DEPTH, int views
= DEFAULT_VIEWS, size_t reduced_num_dim = DEFAULT_REDUCED_NUM_DIM, int
num_quant_bits = DEFAULT_NUM_QUANT_BITS, bool print_status = true);
```

base_set Vector of `BaseKeypoint` type. Contains keypoints from the image are used for training

rng Random numbers generator is used for training

make_patch Patch generator is used for training

num_trees Number of randomized trees used in RTreeClassificator

depth Maximum tree depth

reduced_num_dim Number of dimensions are used in compressed signature

num_quant_bits Number of bits are used for quantization

print_status Print current status of training on the console

---

cv::RTreeClassifier::getSignature (view/add comments)

Returns signature for image patch

```
void getSignature(IplImage *patch, uchar *sig)
```

```
void getSignature(IplImage *patch, float *sig)
```

patch Image patch to calculate signature for

sig Output signature (array dimension is `reduced_num_dim`)

---

cv::RTreeClassifier::getSparseSignature (view/add comments)

The function is simular to `getSignature` but uses the threshold for removing all signature elements less than the threshold. So that the signature is compressed

```
void getSparseSignature(IplImage *patch, float *sig, float thresh);
```

patch Image patch to calculate signature for

sig Output signature (array dimension is `reduced_num_dim`)

tresh The threshold that is used for compressing the signature

cv::RTreeClassifier::countNonZeroElements (view/add comments)

The function returns the number of non-zero elements in the input array.

```
static int countNonZeroElements(float *vec, int n, double tol=1e-10);
```

vec Input vector contains float elements

n Input vector size

tol The threshold used for elements counting. We take all elements are less than `tol` as zero elements

cv::RTreeClassifier::read (view/add comments)

Reads pre-saved RTreeClassifier from file or stream

```
read(const char* file_name)
```

```
read(std::istream &is)
```

file_name Filename of file contains randomized tree data

is Input stream associated with file contains randomized tree data

cv::RTreeClassifier::write (view/add comments)

Writes current RTreeClassifier to a file or stream

```
void write(const char* file_name) const;
```

```
void write(std::ostream &os) const;
```

file_name Filename of file where randomized tree data will be stored

is Output stream associated with file where randomized tree data will be stored

---

cv::RTreeClassifier::setQuantization (view/add comments)

Applies quantization to the current randomized tree

```
void setQuantization(int num_quant_bits)
```

num_quant_bits Number of bits are used for quantization

Below there is an example of `RTreeClassifier` usage for feature matching. There are test and train images and we extract features from both with SURF. Output is $best_corr$ and $best_corr_idx$ arrays which keep the best probabilities and corresponding features indexes for every train feature.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq *objectKeypoints = 0, *objectDescriptors = 0;
CvSeq *imageKeypoints = 0, *imageDescriptors = 0;
CvSURFParams params = cvSURFParams(500, 1);
cvExtractSURF( test_image, 0, &imageKeypoints, &imageDescriptors,
                storage, params );
cvExtractSURF( train_image, 0, &objectKeypoints, &objectDescriptors,
                storage, params );

cv::RTreeClassifier detector;
int patch_width = cv::PATCH_SIZE;
iint patch_height = cv::PATCH_SIZE;
vector<cv::BaseKeypoint> base_set;
int i=0;
CvSURFPoint* point;
for (i=0;i<(n_points > 0 ? n_points : objectKeypoints->total);i++)
{
        point=(CvSURFPoint*)cvGetSeqElem(objectKeypoints,i);
```

```
        base_set.push_back(
                cv::BaseKeypoint(point->pt.x,point->pt.y,train_image));
}

        //Detector training
 cv::RNG rng( cvGetTickCount() );
cv::PatchGenerator gen(0,255,2,false,0.7,1.3,-CV_PI/3,CV_PI/3,
                       -CV_PI/3,CV_PI/3);

printf("RTree Classifier training...\n");
detector.train(base_set,rng,gen,24,cv::DEFAULT_DEPTH,2000,
        (int)base_set.size(), detector.DEFAULT_NUM_QUANT_BITS);
printf("Done\n");

float* signature = new float[detector.original_num_classes()];
float* best_corr;
int* best_corr_idx;
if (imageKeypoints->total > 0)
{
        best_corr = new float[imageKeypoints->total];
        best_corr_idx = new int[imageKeypoints->total];
}

for(i=0; i < imageKeypoints->total; i++)
{
        point=(CvSURFPoint*)cvGetSeqElem(imageKeypoints,i);
        int part_idx = -1;
        float prob = 0.0f;

        CvRect roi = cvRect((int)(point->pt.x) - patch_width/2,
                (int)(point->pt.y) - patch_height/2,
                 patch_width, patch_height);
        cvSetImageROI(test_image, roi);
        roi = cvGetImageROI(test_image);
        if(roi.width != patch_width || roi.height != patch_height)
        {
                best_corr_idx[i] = part_idx;
                best_corr[i] = prob;
        }
        else
        {
                cvSetImageROI(test_image, roi);
                IplImage* roi_image =
                         cvCreateImage(cvSize(roi.width, roi.height),
                         test_image->depth, test_image->nChannels);
```

```
                cvCopy(test_image,roi_image);

                detector.getSignature(roi_image, signature);
                for (int j = 0; j< detector.original_num_classes();j++)
                {
                        if (prob < signature[j])
                        {
                                part_idx = j;
                                prob = signature[j];
                        }
                }

                best_corr_idx[i] = part_idx;
                best_corr[i] = prob;


                if (roi_image)
                        cvReleaseImage(&roi_image);
        }
        cvResetImageROI(test_image);
}
```

13.2 Common Interfaces of Feature Detectors

Feature detectors in OpenCV have wrappers with common interface that enables to switch easily between different algorithms solving the same problem. All objects that implement keypoint detectors inherit cv::FeatureDetector interface.

---

cv::KeyPoint (view/add comments)

Data structure for salient point detectors.

```
class KeyPoint
{
public:
    // the default constructor
    KeyPoint() : pt(0,0), size(0), angle(-1), response(0), octave(0),
                 class_id(-1) {}
    // the full constructor
    KeyPoint(Point2f _pt, float _size, float _angle=-1,
             float _response=0, int _octave=0, int _class_id=-1)
             : pt(_pt), size(_size), angle(_angle), response(_response),
               octave(_octave), class_id(_class_id) {}
```

```
    // another form of the full constructor
    KeyPoint(float x, float y, float _size, float _angle=-1,
            float _response=0, int _octave=0, int _class_id=-1)
            : pt(x, y), size(_size), angle(_angle), response(_response),
              octave(_octave), class_id(_class_id) {}
    // converts vector of keypoints to vector of points
    static void convert(const std::vector<KeyPoint>& keypoints,
                        std::vector<Point2f>& points2f,
                        const std::vector<int>& keypointIndexes=std::vector<int>());
    // converts vector of points to the vector of keypoints, where each
    // keypoint is assigned the same size and the same orientation
    static void convert(const std::vector<Point2f>& points2f,
                        std::vector<KeyPoint>& keypoints,
                        float size=1, float response=1, int octave=0,
                        int class_id=-1);

    // computes overlap for pair of keypoints;
    // overlap is a ratio between area of keypoint regions intersection and
    // area of keypoint regions union (now keypoint region is circle)
    static float overlap(const KeyPoint& kp1, const KeyPoint& kp2);

    Point2f pt; // coordinates of the keypoints
    float size; // diameter of the meaningfull keypoint neighborhood
    float angle; // computed orientation of the keypoint (-1 if not applicable)
    float response; // the response by which the most strong keypoints
                    // have been selected. Can be used for the further sorting
                    // or subsampling
    int octave; // octave (pyramid layer) from which the keypoint has been extracted
    int class_id; // object class (if the keypoints need to be clustered by
                  // an object they belong to)
};

// writes vector of keypoints to the file storage
void write(FileStorage& fs, const string& name, const vector<KeyPoint>& keypoints);
// reads vector of keypoints from the specified file storage node
void read(const FileNode& node, CV_OUT vector<KeyPoint>& keypoints);
```

---

cv::FeatureDetector (view/add comments)

Abstract base class for 2D image feature detectors.

```
class CV_EXPORTS FeatureDetector
{
public:
```

```
    virtual ~FeatureDetector();

    void detect( const Mat& image, vector<KeyPoint>& keypoints,
                 const Mat& mask=Mat() ) const;

    void detect( const vector<Mat>& images,
                 vector<vector<KeyPoint> >& keypoints,
                 const vector<Mat>& masks=vector<Mat>() ) const;

    virtual void read(const FileNode&);
    virtual void write(FileStorage&) const;

    static Ptr<FeatureDetector> create( const string& detectorType );

protected:
...
};
```

---

cv::FeatureDetector::detect (view/add comments)

Detect keypoints in an image (first variant) or image set (second variant).

```
void FeatureDetector::detect( const Mat& image,
            vector<KeyPoint>& keypoints,
            const Mat& mask=Mat() ) const;
```

image The image.

keypoints The detected keypoints.

mask Mask specifying where to look for keypoints (optional). Must be a char matrix with non-zero values in the region of interest.

```
void FeatureDetector::detect( const vector<Mat>& images,
            vector<vector<KeyPoint> >& keypoints,
            const vector<Mat>& masks=vector<Mat>() ) const;
```

images Images set.

keypoints Collection of keypoints detected in an input images. keypoints[i] is a set of keypoints detected in an images[i].

masks Masks for each input image specifying where to look for keypoints (optional). masks[i] is a mask for images[i]. Each element of `masks` vector must be a char matrix with non-zero values in the region of interest.

---

cv::FeatureDetector::read (view/add comments)

Read feature detector object from file node.

```
void FeatureDetector::read( const FileNode& fn );
```

fn File node from which detector will be read.

---

cv::FeatureDetector::write (view/add comments)

Write feature detector object to file storage.

```
void FeatureDetector::write( FileStorage& fs ) const;
```

fs File storage in which detector will be written.

---

cv::FeatureDetector::create (view/add comments)

Feature detector factory that creates cv::FeatureDetector of given type with default parameters (rather using default constructor).

```
Ptr<FeatureDetector> FeatureDetector::create( const string& detectorType );
```

detectorType Feature detector type.

Now the following detector types are supported:
"FAST" – cv::FastFeatureDetector,
"STAR" – cv::StarFeatureDetector,
"SIFT" – cv::SiftFeatureDetector,
"SURF" – cv::SurfFeatureDetector,
"MSER" – cv::MserFeatureDetector,
"GFTT" – cv::GfttFeatureDetector,
"HARRIS" – cv::HarrisFeatureDetector.
Also combined format is supported: feature detector adapter name ("Grid" – cv::GridAdaptedFeatureDetector, "Pyramid" – cv::PyramidAdaptedFeatureDetector) + feature detector name (see above), e.g. "GridFAST", "PyramidSTAR", etc.

---

cv::FastFeatureDetector (view/add comments)

Wrapping class for feature detection using cv::FAST method.

```
class FastFeatureDetector : public FeatureDetector
{
public:
    FastFeatureDetector( int threshold=1, bool nonmaxSuppression=true );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

---

cv::GoodFeaturesToTrackDetector (view/add comments)

Wrapping class for feature detection using cv::goodFeaturesToTrack function.

```
class GoodFeaturesToTrackDetector : public FeatureDetector
{
public:
    class Params
    {
    public:
        Params( int maxCorners=1000, double qualityLevel=0.01,
                double minDistance=1., int blockSize=3,
                bool useHarrisDetector=false, double k=0.04 );
        void read( const FileNode& fn );
        void write( FileStorage& fs ) const;

        int maxCorners;
```

```
        double qualityLevel;
        double minDistance;
        int blockSize;
        bool useHarrisDetector;
        double k;
    };

    GoodFeaturesToTrackDetector( const GoodFeaturesToTrackDetector::Params& params=
                                                GoodFeaturesToTrackDetector::Params() );
    GoodFeaturesToTrackDetector( int maxCorners, double qualityLevel,
                                 double minDistance, int blockSize=3,
                                 bool useHarrisDetector=false, double k=0.04 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

cv::MserFeatureDetector (view/add comments)

Wrapping class for feature detection using cv::MSER class.

```
class MserFeatureDetector : public FeatureDetector
{
public:
    MserFeatureDetector( CvMSERParams params=cvMSERParams() );
    MserFeatureDetector( int delta, int minArea, int maxArea,
                         double maxVariation, double minDiversity,
                         int maxEvolution, double areaThreshold,
                         double minMargin, int edgeBlurSize );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

cv::StarFeatureDetector (view/add comments)

Wrapping class for feature detection using cv::StarDetector class.

```
class StarFeatureDetector : public FeatureDetector
{
public:
```

```
    StarFeatureDetector( int maxSize=16, int responseThreshold=30,
                         int lineThresholdProjected = 10,
                         int lineThresholdBinarized=8, int suppressNonmaxSize=5 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

---

cv::SiftFeatureDetector (view/add comments)

Wrapping class for feature detection using cv::SIFT class.

```
class SiftFeatureDetector : public FeatureDetector
{
public:
    SiftFeatureDetector(
        const SIFT::DetectorParams& detectorParams=SIFT::DetectorParams(),
        const SIFT::CommonParams& commonParams=SIFT::CommonParams() );
    SiftFeatureDetector( double threshold, double edgeThreshold,
                         int nOctaves=SIFT::CommonParams::DEFAULT_NOCTAVES,
                         int nOctaveLayers=SIFT::CommonParams::DEFAULT_NOCTAVE_LAYERS,
                         int firstOctave=SIFT::CommonParams::DEFAULT_FIRST_OCTAVE,
                         int angleMode=SIFT::CommonParams::FIRST_ANGLE );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

---

cv::SurfFeatureDetector (view/add comments)

Wrapping class for feature detection using cv::SURF class.

```
class SurfFeatureDetector : public FeatureDetector
{
public:
    SurfFeatureDetector( double hessianThreshold = 400., int octaves = 3,
                         int octaveLayers = 4 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

cv::GridAdaptedFeatureDetector (view/add comments)

Adapts a detector to partition the source image into a grid and detect points in each cell.

```
class GridAdaptedFeatureDetector : public FeatureDetector
{
public:
    /*
     * detector            Detector that will be adapted.
     * maxTotalKeypoints   Maximum count of keypoints detected on the image.
     *                     Only the strongest keypoints will be keeped.
     * gridRows            Grid rows count.
     * gridCols            Grid column count.
     */
    GridAdaptedFeatureDetector( const Ptr<FeatureDetector>& detector,
                                int maxTotalKeypoints, int gridRows=4,
                                int gridCols=4 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

cv::PyramidAdaptedFeatureDetector (view/add comments)

Adapts a detector to detect points over multiple levels of a Gaussian pyramid. Useful for detectors that are not inherently scaled.

```
class PyramidAdaptedFeatureDetector : public FeatureDetector
{
public:
    PyramidAdaptedFeatureDetector( const Ptr<FeatureDetector>& detector,
                                   int levels=2 );
    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;
protected:
    ...
};
```

cv::DynamicAdaptedFeatureDetector (view/add comments)

An adaptively adjusting detector that iteratively detects until the desired number of features are found.

If the detector is persisted, it will ”remember” the parameters used on the last detection. In this way, the detector may be used for consistent numbers of keypoints in a sets of images that are temporally related such as video streams or panorama series.

The DynamicAdaptedFeatureDetector uses another detector such as FAST or SURF to do the dirty work, with the help of an AdjusterAdapter. After a detection, and an unsatisfactory number of features are detected, the AdjusterAdapter will adjust the detection parameters so that the next detection will result in more or less features. This is repeated until either the number of desired features are found or the parameters are maxed out.

Adapters can easily be implemented for any detector via the AdjusterAdapter interface.

Beware that this is not thread safe - as the adjustment of parameters breaks the const of the detection routine...

Here is a sample of how to create a DynamicAdaptedFeatureDetector.

```
//sample usage:
//will create a detector that attempts to find
//100 - 110 FAST Keypoints, and will at most run
//FAST feature detection 10 times until that
//number of keypoints are found
Ptr<FeatureDetector> detector(new DynamicAdaptedFeatureDetector (100, 110, 10,
                              new FastAdjuster(20,true)));
```

```
class DynamicAdaptedFeatureDetector: public FeatureDetector
{
public:
    DynamicAdaptedFeatureDetector( const Ptr<AdjusterAdapter>& adjaster,
        int min_features=400, int max_features=500, int max_iters=5 );
    ...
};
```

---

cv::DynamicAdaptedFeatureDetector::DynamicAdaptedFeatureDetector (view/add comments)

DynamicAdaptedFeatureDetector constructor.

```
DynamicAdaptedFeatureDetector::DynamicAdaptedFeatureDetector(
           const Ptr<AdjusterAdapter>& adjaster,
           int min_features,
           int max_features,
           int max_iters );
```

adjaster An cv::AdjusterAdapter that will do the detection and parameter adjustment

min_features This minimum desired number features.

max_features The maximum desired number of features.

max_iters The maximum number of times to try to adjust the feature detector parameters. For the cv::FastAdjuster this number can be high, but with Star or Surf, many iterations can get time consuming. At each iteration the detector is rerun, so keep this in mind when choosing this value.

---

cv::AdjusterAdapter (view/add comments)

A feature detector parameter adjuster interface, this is used by the cv::DynamicAdaptedFeatureDetector and is a wrapper for cv::FeatureDetector that allow them to be adjusted after a detection.

See cv::FastAdjuster, cv::StarAdjuster, cv::SurfAdjuster for concrete implementations.

```
class AdjusterAdapter: public FeatureDetector
{
public:
        virtual ~AdjusterAdapter() {}
        virtual void tooFew(int min, int n_detected) = 0;
        virtual void tooMany(int max, int n_detected) = 0;
        virtual bool good() const = 0;
};
```

---

cv::AdjusterAdapter::tooFew (view/add comments)

```
virtual void tooFew(int min, int n_detected) = 0;
```

Too few features were detected so, adjust the detector parameters accordingly - so that the next detection detects more features.

min This minimum desired number features.

n_detected The actual number detected last run.

An example implementation of this is

```
void FastAdjuster::tooFew(int min, int n_detected)
{
        thresh_--;
}
```

---

cv::AdjusterAdapter::tooMany (view/add comments)

Too many features were detected so, adjust the detector parameters accordingly - so that the next detection detects less features.

```
virtual void tooMany(int max, int n_detected) = 0;
```

max This maximum desired number features.

n_detected The actual number detected last run.

An example implementation of this is

```
void FastAdjuster::tooMany(int min, int n_detected)
{
        thresh_++;
}
```

---

cv::AdjusterAdapter::good (view/add comments)

Are params maxed out or still valid? Returns false if the parameters can't be adjusted any more.

```
virtual bool good() const = 0;
```

An example implementation of this is

```
bool FastAdjuster::good() const
{
        return (thresh_ > 1) && (thresh_ < 200);
}
```

cv::FastAdjuster (view/add comments)

An cv::AdjusterAdapter for the cv::FastFeatureDetector. This will basically decrement or increment the threshhold by 1

```
class FastAdjuster FastAdjuster: public AdjusterAdapter
{
public:
        FastAdjuster(int init_thresh = 20, bool nonmax = true);
        ...
};
```

cv::StarAdjuster (view/add comments)

An cv::AdjusterAdapter for the cv::StarFeatureDetector. This adjusts the responseThreshhold of StarFeatureDetector.

```
class StarAdjuster: public AdjusterAdapter
{
        StarAdjuster(double initial_thresh = 30.0);
        ...
};
```

cv::SurfAdjuster (view/add comments)

An cv::AdjusterAdapter for the cv::SurfFeatureDetector. This adjusts the hessianThreshold of SurfFeatureDetector.

```
class SurfAdjuster: public SurfAdjuster
{
        SurfAdjuster();
        ...
};
```

13.3 Common Interfaces of Descriptor Extractors

Extractors of keypoint descriptors in OpenCV have wrappers with common interface that enables to switch easily between different algorithms solving the same problem. This section is devoted to computing descriptors that are represented as vectors in a multidimensional space. All objects that implement ”vector” descriptor extractors inherit cv::DescriptorExtractor interface.

cv::DescriptorExtractor (view/add comments)

Abstract base class for computing descriptors for image keypoints.

```
class CV_EXPORTS DescriptorExtractor
{
public:
    virtual ~DescriptorExtractor();

    void compute( const Mat& image, vector<KeyPoint>& keypoints,
                  Mat& descriptors ) const;
    void compute( const vector<Mat>& images, vector<vector<KeyPoint> >& keypoints,
                  vector<Mat>& descriptors ) const;

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;

    virtual int descriptorSize() const = 0;
    virtual int descriptorType() const = 0;

    static Ptr<DescriptorExtractor> create( const string& descriptorExtractorType );

protected:
    ...
};
```

In this interface we assume a keypoint descriptor can be represented as a dense, fixed-dimensional vector of some basic type. Most descriptors used in practice follow this pattern, as it makes it very easy to compute distances between descriptors. Therefore we represent a collection of descriptors as a cv::Mat, where each row is one keypoint descriptor.

cv::DescriptorExtractor::compute (view/add comments)

Compute the descriptors for a set of keypoints detected in an image (first variant) or image set (second variant).

```
void DescriptorExtractor::compute( const Mat& image,
           vector<KeyPoint>& keypoints,
           Mat& descriptors ) const;
```

image The image.

keypoints The keypoints. Keypoints for which a descriptor cannot be computed are removed.

descriptors The descriptors. Row i is the descriptor for keypoint i.

```
void DescriptorExtractor::compute( const vector<Mat>& images,
           vector<vector<KeyPoint> >& keypoints,
           vector<Mat>& descriptors ) const;
```

images The image set.

keypoints Input keypoints collection. keypoints[i] is keypoints detected in images[i]. Keypoints for which a descriptor can not be computed are removed.

descriptors Descriptor collection. descriptors[i] are descriptors computed for a set keypoints[i].

---

cv::DescriptorExtractor::read (view/add comments)

Read descriptor extractor object from file node.

```
void DescriptorExtractor::read( const FileNode& fn );
```

fn File node from which detector will be read.

---

cv::DescriptorExtractor::write (view/add comments)

Write descriptor extractor object to file storage.

```
void DescriptorExtractor::write( FileStorage& fs ) const;
```

fs File storage in which detector will be written.

cv::DescriptorExtractor::create (view/add comments)

Descriptor extractor factory that creates cv::DescriptorExtractor of given type with default parameters (rather using default constructor).

```
Ptr<DescriptorExtractor>
DescriptorExtractor::create( const string& descriptorExtractorType );
```

descriptorExtractorType Descriptor extractor type.

Now the following descriptor extractor types are supported:
"SIFT" – cv::SiftFeatureDetector,
"SURF" – cv::SurfFeatureDetector,
"BRIEF" – cv::BriefFeatureDetector.
Also combined format is supported: descriptor extractor adapter name ("Opponent" – cv::OpponentColorDescri
+ descriptor extractor name (see above), e.g. "OpponentSIFT", etc.

cv::SiftDescriptorExtractor (view/add comments)

Wrapping class for descriptors computing using cv::SIFT class.

```
class SiftDescriptorExtractor : public DescriptorExtractor
{
public:
    SiftDescriptorExtractor(
        const SIFT::DescriptorParams& descriptorParams=SIFT::DescriptorParams(),
        const SIFT::CommonParams& commonParams=SIFT::CommonParams() );
    SiftDescriptorExtractor( double magnification, bool isNormalize=true,
        bool recalculateAngles=true, int nOctaves=SIFT::CommonParams::DEFAULT_NOCTAVES,
        int nOctaveLayers=SIFT::CommonParams::DEFAULT_NOCTAVE_LAYERS,
        int firstOctave=SIFT::CommonParams::DEFAULT_FIRST_OCTAVE,
        int angleMode=SIFT::CommonParams::FIRST_ANGLE );

    virtual void read (const FileNode &fn);
    virtual void write (FileStorage &fs) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

cv::SurfDescriptorExtractor (view/add comments)

Wrapping class for descriptors computing using cv::SURF class.

```
class SurfDescriptorExtractor : public DescriptorExtractor
{
public:
    SurfDescriptorExtractor( int nOctaves=4,
                             int nOctaveLayers=2, bool extended=false );

    virtual void read (const FileNode &fn);
    virtual void write (FileStorage &fs) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

cv::CalonderDescriptorExtractor (view/add comments)

Wrapping class for descriptors computing using cv::RTreeClassifier class.

```
template<typename T>
class CalonderDescriptorExtractor : public DescriptorExtractor
{
public:
    CalonderDescriptorExtractor( const string& classifierFile );

    virtual void read( const FileNode &fn );
    virtual void write( FileStorage &fs ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
}
```

cv::OpponentColorDescriptorExtractor (view/add comments)

Adapts a descriptor extractor to compute descripors in Opponent Color Space (refer to van de Sande et al., CGIV 2008 "Color Descriptors for Object Category Recognition"). Input RGB image is transformed in Opponent Color Space. Then unadapted descriptor extractor (set in constructor) computes descriptors on each of the three channel and concatenate them into a single color descriptor.

```
class OpponentColorDescriptorExtractor : public DescriptorExtractor
{
public:
    OpponentColorDescriptorExtractor( const Ptr<DescriptorExtractor>& dextractor );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
};
```

---

cv::BriefDescriptorExtractor (view/add comments)

Class for computing BRIEF descriptors described in paper of Calonder M., Lepetit V., Strecha C., Fua P.: ”BRIEF: Binary Robust Independent Elementary Features.” 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.

```
class BriefDescriptorExtractor : public DescriptorExtractor
{
public:
    static const int PATCH_SIZE = 48;
    static const int KERNEL_SIZE = 9;

    // bytes is a length of descriptor in bytes. It can be equal 16, 32 or 64 bytes.
    BriefDescriptorExtractor( int bytes = 32 );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;
    virtual int descriptorSize() const;
    virtual int descriptorType() const;
protected:
    ...
};
```

13.4 Common Interfaces of Descriptor Matchers

Matchers of keypoint descriptors in OpenCV have wrappers with common interface that enables to switch easily between different algorithms solving the same problem. This section is devoted to matching descriptors that are represented as vectors in a multidimensional space. All objects that implement ”vector” descriptor matchers inherit cv::DescriptorMatcher interface.

cv::DMatch (view/add comments)

Match between two keypoint descriptors: query descriptor index, train descriptor index, train image index and distance between descriptors.

```
struct DMatch
{
    DMatch() : queryIdx(-1), trainIdx(-1), imgIdx(-1),
               distance(std::numeric_limits<float>::max()) {}
    DMatch( int _queryIdx, int _trainIdx, float _distance ) :
            queryIdx(_queryIdx), trainIdx(_trainIdx), imgIdx(-1),
            distance(_distance) {}
    DMatch( int _queryIdx, int _trainIdx, int _imgIdx, float _distance ) :
            queryIdx(_queryIdx), trainIdx(_trainIdx), imgIdx(_imgIdx),
            distance(_distance) {}

    int queryIdx; // query descriptor index
    int trainIdx; // train descriptor index
    int imgIdx;   // train image index

    float distance;

    // less is better
    bool operator<( const DMatch &m ) const;
};
```

cv::DescriptorMatcher (view/add comments)

Abstract base class for matching keypoint descriptors. It has two groups of match methods: for matching descriptors of one image with other image or with image set.

```
class DescriptorMatcher
{
public:
    virtual ~DescriptorMatcher();

    virtual void add( const vector<Mat>& descriptors );

    const vector<Mat>& getTrainDescriptors() const;
    virtual void clear();
    bool empty() const;
    virtual bool isMaskSupported() const = 0;

    virtual void train();
```

```
    /*
     * Group of methods to match descriptors from image pair.
     */
    void match( const Mat& queryDescriptors, const Mat& trainDescriptors,
                vector<DMatch>& matches, const Mat& mask=Mat() ) const;
    void knnMatch( const Mat& queryDescriptors, const Mat& trainDescriptors,
                   vector<vector<DMatch> >& matches, int k,
                   const Mat& mask=Mat(), bool compactResult=false ) const;
    void radiusMatch( const Mat& queryDescriptors, const Mat& trainDescriptors,
                      vector<vector<DMatch> >& matches, float maxDistance,
                      const Mat& mask=Mat(), bool compactResult=false ) const;
    /*
     * Group of methods to match descriptors from one image to image set.
     */
    void match( const Mat& queryDescriptors, vector<DMatch>& matches,
                const vector<Mat>& masks=vector<Mat>() );
    void knnMatch( const Mat& queryDescriptors, vector<vector<DMatch> >& matches,
                   int k, const vector<Mat>& masks=vector<Mat>(),
                   bool compactResult=false );
    void radiusMatch( const Mat& queryDescriptors, vector<vector<DMatch> >& matches,
                      float maxDistance, const vector<Mat>& masks=vector<Mat>(),
                      bool compactResult=false );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;

    virtual Ptr<DescriptorMatcher> clone( bool emptyTrainData=false ) const = 0;

    static Ptr<DescriptorMatcher> create( const string& descriptorMatcherType );

protected:
    vector<Mat> trainDescCollection;
    ...
};
```

---

cv::DescriptorMatcher::add (view/add comments)

Add descriptors to train descriptor collection. If collection `trainDescCollection` is not empty the new descriptors are added to existing train descriptors.

```
void add( const vector<Mat>& descriptors );
```

descriptors Descriptors to add. Each `descriptors[i]` is a set of descriptors from the same (one) train image.

---

cv::DescriptorMatcher::getTrainDescriptors (view/add comments)

Returns constant link to the train descriptor collection (i.e. `trainDescCollection`).

```
const vector<Mat>& getTrainDescriptors() const;
```

---

cv::DescriptorMatcher::clear (view/add comments)

Clear train descriptor collection.

```
void DescriptorMatcher::clear();
```

---

cv::DescriptorMatcher::empty (view/add comments)

Return true if there are not train descriptors in collection.

```
bool DescriptorMatcher::empty() const;
```

---

cv::DescriptorMatcher::isMaskSupported (view/add comments)

Returns true if descriptor matcher supports masking permissible matches.

```
bool DescriptorMatcher::isMaskSupported();
```

---

cv::DescriptorMatcher::train (view/add comments)

Train descriptor matcher (e.g. train flann index). In all methods to match the method train() is run every time before matching. Some descriptor matchers (e.g. BruteForceMatcher) have empty implementation of this method, other matchers realy train their inner structures (e.g. FlannBasedMatcher trains flann::Index)

```
void DescriptorMatcher::train();
```

---

cv::DescriptorMatcher::match (view/add comments)

Find the best match for each descriptor from a query set with train descriptors. Supposed that the query descriptors are of keypoints detected on the same query image. In first variant of this method train descriptors are set as input argument and supposed that they are of keypoints detected on the same train image. In second variant of the method train descriptors collection that was set using `add` method is used. Optional mask (or masks) can be set to describe which descriptors can be matched. `queryDescriptors[i]` can be matched with `trainDescriptors[j]` only if `mask.at<uchar>(i,j)` is non-zero.

```
void DescriptorMatcher::match( const Mat& queryDescriptors,
           const Mat& trainDescriptors,
           vector<DMatch>& matches,
           const Mat& mask=Mat() ) const;
```

```
void DescriptorMatcher::match( const Mat& queryDescriptors,
           vector<DMatch>& matches,
           const vector<Mat>& masks=vector<Mat>() );
```

queryDescriptors Query set of descriptors.

trainDescriptors Train set of descriptors. This will not be added to train descriptors collection stored in class object.

matches Matches. If some query descriptor masked out in `mask` no match will be added for this descriptor. So `matches` size may be less query descriptors count.

mask Mask specifying permissible matches between input query and train matrices of descriptors.

masks The set of masks. Each `masks[i]` specifies permissible matches between input query descriptors and stored train descriptors from i-th image (i.e. `trainDescCollection[i]`).

---

cv::DescriptorMatcher::knnMatch (view/add comments)

Find the k best matches for each descriptor from a query set with train descriptors. Found k (or less if not possible) matches are returned in distance increasing order. Details about query and train descriptors see in cv::DescriptorMatcher::match.

```
void DescriptorMatcher::knnMatch( const Mat& queryDescriptors,
           const Mat& trainDescriptors,
           vector<vector<DMatch> >& matches,
           int k, const Mat& mask=Mat(),
           bool compactResult=false ) const;
```

```
void DescriptorMatcher::knnMatch( const Mat& queryDescriptors,
           vector<vector<DMatch> >& matches, int k,
           const vector<Mat>& masks=vector<Mat>(),
           bool compactResult=false );
```

queryDescriptors, trainDescriptors, mask, masks See in cv::DescriptorMatcher::match.

matches Mathes. Each `matches[i]` is k or less matches for the same query descriptor.

k Count of best matches will be found per each query descriptor (or less if it's not possible).

compactResult It's used when mask (or masks) is not empty. If `compactResult` is false `matches` vector will have the same size as `queryDescriptors` rows. If `compactResult` is true `matches` vector will not contain matches for fully masked out query descriptors.

---

cv::DescriptorMatcher::radiusMatch (view/add comments)

Find the best matches for each query descriptor which have distance less than given threshold. Found matches are returned in distance increasing order. Details about query and train descriptors see in cv::DescriptorMatcher::match.

```
void DescriptorMatcher::radiusMatch( const Mat& queryDescriptors,
           const Mat& trainDescriptors,
           vector<vector<DMatch> >& matches,
           float maxDistance, const Mat& mask=Mat(),
           bool compactResult=false ) const;
```

```
void DescriptorMatcher::radiusMatch( const Mat& queryDescriptors,
           vector<vector<DMatch> >& matches,
           float maxDistance,
           const vector<Mat>& masks=vector<Mat>(),
           bool compactResult=false );
```

queryDescriptors, trainDescriptors, mask, masks See in cv::DescriptorMatcher::match.

matches, compactResult See in cv::DescriptorMatcher::knnMatch.

maxDistance The threshold to found match distances.

---

cv::DescriptorMatcher::clone (view/add comments)

Clone the matcher.

```
Ptr<DescriptorMatcher>
DescriptorMatcher::clone( bool emptyTrainData ) const;
```

emptyTrainData If emptyTrainData is false the method create deep copy of the object, i.e. copies both parameters and train data. If emptyTrainData is true the method create object copy with current parameters but with empty train data..

---

cv::DescriptorMatcher::create (view/add comments)

Descriptor matcher factory that creates cv::DescriptorMatcher of given type with default parameters (rather using default constructor).

```
Ptr<DescriptorMatcher>
DescriptorMatcher::create( const string& descriptorMatcherType );
```

descriptorMatcherType Descriptor matcher type.

Now the following matcher types are supported: `"BruteForce"` (it uses `L2`), `"BruteForce-L1"`, `"BruteForce-Hamming"`, `"BruteForce-HammingLUT"`, `"FlannBased"`.

---

cv::BruteForceMatcher (view/add comments)

Brute-force descriptor matcher. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. This descriptor matcher supports masking permissible matches between descriptor sets.

```
template<class Distance>
class BruteForceMatcher : public DescriptorMatcher
{
public:
    BruteForceMatcher( Distance d = Distance() );
    virtual ~BruteForceMatcher();

    virtual bool isMaskSupported() const;
    virtual Ptr<DescriptorMatcher> clone( bool emptyTrainData=false ) const;
protected:
    ...
}
```

For efficiency, BruteForceMatcher is templated on the distance metric. For float descriptors, a common choice would be `L2<float>`. Class of supported distances are:

```
template<typename T>
struct Accumulator
{
    typedef T Type;
};
```

```
template<> struct Accumulator<unsigned char>  { typedef unsigned int Type; };
template<> struct Accumulator<unsigned short> { typedef unsigned int Type; };
template<> struct Accumulator<char>   { typedef int Type; };
template<> struct Accumulator<short>  { typedef int Type; };

/*
 * Squared Euclidean distance functor
 */
template<class T>
struct L2
{
    typedef T ValueType;
    typedef typename Accumulator<T>::Type ResultType;

    ResultType operator()( const T* a, const T* b, int size ) const;
};

/*
 * Manhattan distance (city block distance) functor
 */
template<class T>
struct CV_EXPORTS L1
{
    typedef T ValueType;
    typedef typename Accumulator<T>::Type ResultType;

    ResultType operator()( const T* a, const T* b, int size ) const;
    ...
};

/*
 * Hamming distance (city block distance) functor
 */
struct HammingLUT
{
    typedef unsigned char ValueType;
    typedef int ResultType;

    ResultType operator()( const unsigned char* a, const unsigned char* b,
                           int size ) const;
    ...
};

struct Hamming
```

```
{
    typedef unsigned char ValueType;
    typedef int ResultType;

    ResultType operator()( const unsigned char* a, const unsigned char* b,
                           int size ) const;
    ...
};
```

---

cv::FlannBasedMatcher (view/add comments)

Flann based descriptor matcher. This matcher trains cv::flann::Index on train descriptor collection and calls it's nearest search methods to find best matches. So this matcher may be faster in cases of matching to large train collection than brute force matcher. `FlannBasedMatcher` does not support masking permissible matches between descriptor sets, because cv::flann::Index does not support this.

```
class FlannBasedMatcher : public DescriptorMatcher
{
public:
    FlannBasedMatcher(
      const Ptr<flann::IndexParams>& indexParams=new flann::KDTreeIndexParams(),
      const Ptr<flann::SearchParams>& searchParams=new flann::SearchParams() );

    virtual void add( const vector<Mat>& descriptors );
    virtual void clear();

    virtual void train();
    virtual bool isMaskSupported() const;

    virtual Ptr<DescriptorMatcher> clone( bool emptyTrainData=false ) const;
protected:
    ...
};
```

13.5 Common Interfaces of Generic Descriptor Matchers

Matchers of keypoint descriptors in OpenCV have wrappers with common interface that enables to switch easily between different algorithms solving the same problem. This section is devoted to matching descriptors that can not be represented as vectors in a multidimensional space. `GenericDescriptorMatcher` is a more generic interface for descriptors. It does not make any

assumptions about descriptor representation. Every descriptor with cv::DescriptorExtractor interface has a wrapper with `GenericDescriptorMatcher` interface (see cv::VectorDescriptorMatcher). There are descriptors such as One way descriptor and Ferns that have `GenericDescriptorMatcher` interface implemented, but do not support cv::DescriptorExtractor.

---

cv::GenericDescriptorMatcher (view/add comments)

Abstract interface for a keypoint descriptor extracting and matching. There is cv::DescriptorExtractor and cv::DescriptorMatcher for these purposes too, but their interfaces are intended for descriptors represented as vectors in a multidimensional space. `GenericDescriptorMatcher` is a more generic interface for descriptors. As cv::DescriptorMatcher, `GenericDescriptorMatcher` has two groups of match methods: for matching keypoints of one image with other image or with image set.

```
class GenericDescriptorMatcher
{
public:
    GenericDescriptorMatcher();
    virtual ~GenericDescriptorMatcher();

    virtual void add( const vector<Mat>& images,
                      vector<vector<KeyPoint> >& keypoints );

    const vector<Mat>& getTrainImages() const;
    const vector<vector<KeyPoint> >& getTrainKeypoints() const;
    virtual void clear();

    virtual void train() = 0;

    virtual bool isMaskSupported() = 0;

    void classify( const Mat& queryImage,
                   vector<KeyPoint>& queryKeypoints,
                   const Mat& trainImage,
                   vector<KeyPoint>& trainKeypoints ) const;
    void classify( const Mat& queryImage,
                   vector<KeyPoint>& queryKeypoints );

    /*
     * Group of methods to match keypoints from image pair.
     */
    void match( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
                vector<DMatch>& matches, const Mat& mask=Mat() ) const;
```

```
    void knnMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                   const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
                   vector<vector<DMatch> >& matches, int k,
                   const Mat& mask=Mat(), bool compactResult=false ) const;
    void radiusMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                      const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
                      vector<vector<DMatch> >& matches, float maxDistance,
                      const Mat& mask=Mat(), bool compactResult=false ) const;
    /*
     * Group of methods to match keypoints from one image to image set.
     */
    void match( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                vector<DMatch>& matches, const vector<Mat>& masks=vector<Mat>() );
    void knnMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                   vector<vector<DMatch> >& matches, int k,
                   const vector<Mat>& masks=vector<Mat>(), bool compactResult=false );
    void radiusMatch( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                      vector<vector<DMatch> >& matches, float maxDistance,
                      const vector<Mat>& masks=vector<Mat>(), bool compactResult=false );

    virtual void read( const FileNode& );
    virtual void write( FileStorage& ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const = 0;

protected:
    ...
};
```

---

cv::GenericDescriptorMatcher::add (view/add comments)

Adds images and keypoints from them to the train collection (descriptors are supposed to be calculated here). If train collection is not empty new image and keypoints from them will be added to existing data.

```
void GenericDescriptorMatcher::add( const vector<Mat>& images,
           vector<vector<KeyPoint> >& keypoints );
```

images Image collection.

keypoints Point collection. Assumes that `keypoints[i]` are keypoints detected in an image `images[i]`.

---

cv::GenericDescriptorMatcher::getTrainImages (view/add comments)

Returns train image collection.

```
const vector<Mat>& GenericDescriptorMatcher::getTrainImages() const;
```

---

cv::GenericDescriptorMatcher::getTrainKeypoints (view/add comments)

Returns train keypoints collection.

```
const vector<vector<KeyPoint> >&
GenericDescriptorMatcher::getTrainKeypoints() const;
```

---

cv::GenericDescriptorMatcher::clear (view/add comments)

Clear train collection (iamges and keypoints).

```
void GenericDescriptorMatcher::clear();
```

---

cv::GenericDescriptorMatcher::train (view/add comments)

Train the object, e.g. tree-based structure to extract descriptors or to optimize descriptors matching.

```
void GenericDescriptorMatcher::train();
```

---

cv::GenericDescriptorMatcher::isMaskSupported (view/add comments)

Returns true if generic descriptor matcher supports masking permissible matches.

```
void GenericDescriptorMatcher::isMaskSupported();
```

---

cv::GenericDescriptorMatcher::classify (view/add comments)

Classifies query keypoints under keypoints of one train image qiven as input argument (first version of the method) or train image collection that set using cv::GenericDescriptorMatcher::add (second version).

```
void GenericDescriptorMatcher::classify(
           const Mat& queryImage,
           vector<KeyPoint>& queryKeypoints,
           const Mat& trainImage,
           vector<KeyPoint>& trainKeypoints ) const;
```

```
void GenericDescriptorMatcher::classify( const Mat& queryImage,
           vector<KeyPoint>& queryKeypoints );
```

queryImage The query image.

queryKeypoints Keypoints from the query image.

trainImage The train image.

trainKeypoints Keypoints from the train image.

---

cv::GenericDescriptorMatcher::match (view/add comments)

Find best match for query keypoints to the training set. In first version of method one train image and keypoints detected on it - are input arguments. In second version query keypoints are matched to training collectin that set using cv::GenericDescriptorMatcher::add. As in cv::DescriptorMatcher::match the mask can be set.

```
void GenericDescriptorMatcher::match(
           const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
           const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
           vector<DMatch>& matches, const Mat& mask=Mat() ) const;
```

```
void GenericDescriptorMatcher::match(
           const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
           vector<DMatch>& matches,
           const vector<Mat>& masks=vector<Mat>() );
```

queryImage Query image.

queryKeypoints Keypoints detected in `queryImage`.

trainImage Train image. This will not be added to train image collection stored in class object.

trainKeypoints Keypoints detected in `trainImage`. They will not be added to train points collection stored in class object.

matches Matches. If some query descriptor (keypoint) masked out in `mask` no match will be added for this descriptor. So `matches` size may be less query keypoints count.

mask Mask specifying permissible matches between input query and train keypoints.

masks The set of masks. Each `masks[i]` specifies permissible matches between input query keypoints and stored train keypointss from i-th image.

---

cv::GenericDescriptorMatcher::knnMatch (view/add comments)

Find the knn best matches for each keypoint from a query set with train keypoints. Found knn (or less if not possible) matches are returned in distance increasing order. Details see in cv::GenericDescriptorMatche and cv::DescriptorMatcher::knnMatch.

```
void GenericDescriptorMatcher::knnMatch(
           const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
           const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
           vector<vector<DMatch> >& matches, int k,
           const Mat& mask=Mat(), bool compactResult=false ) const;
```

```
void GenericDescriptorMatcher::knnMatch(
           const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
           vector<vector<DMatch> >& matches, int k,
           const vector<Mat>& masks=vector<Mat>(),
           bool compactResult=false );
```

cv::GenericDescriptorMatcher::radiusMatch (view/add comments)

Find the best matches for each query keypoint which have distance less than given threshold. Found matches are returned in distance increasing order. Details see in cv::GenericDescriptorMatcher::match and cv::DescriptorMatcher::radiusMatch.

```
void GenericDescriptorMatcher::radiusMatch(
           const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
           const Mat& trainImage, vector<KeyPoint>& trainKeypoints,
           vector<vector<DMatch> >& matches, float maxDistance,
           const Mat& mask=Mat(), bool compactResult=false ) const;
```

```
void GenericDescriptorMatcher::radiusMatch(
           const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
           vector<vector<DMatch> >& matches, float maxDistance,
           const vector<Mat>& masks=vector<Mat>(),
           bool compactResult=false );
```

cv::GenericDescriptorMatcher::read (view/add comments)

Reads matcher object from a file node.

```
void GenericDescriptorMatcher::read( const FileNode& fn );
```

cv::GenericDescriptorMatcher::write (view/add comments)

Writes match object to a file storage

```
void GenericDescriptorMatcher::write( FileStorage& fs ) const;
```

cv::GenericDescriptorMatcher::clone (view/add comments)

Clone the matcher.

```
Ptr<GenericDescriptorMatcher>
GenericDescriptorMatcher::clone( bool emptyTrainData ) const;
```

emptyTrainData If emptyTrainData is false the method create deep copy of the object, i.e. copies both parameters and train data. If emptyTrainData is true the method create object copy with current parameters but with empty train data.

cv::OneWayDescriptorMatcher (view/add comments)

Wrapping class for computing, matching and classification of descriptors using cv::OneWayDescriptorBase class.

```
class OneWayDescriptorMatcher : public GenericDescriptorMatcher
{
public:
    class Params
    {
    public:
        static const int POSE_COUNT = 500;
        static const int PATCH_WIDTH = 24;
        static const int PATCH_HEIGHT = 24;
        static float GET_MIN_SCALE() { return 0.7f; }
        static float GET_MAX_SCALE() { return 1.5f; }
        static float GET_STEP_SCALE() { return 1.2f; }

        Params( int poseCount = POSE_COUNT,
```

```
                Size patchSize = Size(PATCH_WIDTH, PATCH_HEIGHT),
                string pcaFilename = string(),
                string trainPath = string(), string trainImagesList = string(),
                float minScale = GET_MIN_SCALE(), float maxScale = GET_MAX_SCALE(),
                float stepScale = GET_STEP_SCALE() );

        int poseCount;
        Size patchSize;
        string pcaFilename;
        string trainPath;
        string trainImagesList;

        float minScale, maxScale, stepScale;
    };

    OneWayDescriptorMatcher( const Params& params=Params() );
    virtual ~OneWayDescriptorMatcher();

    void initialize( const Params& params, const Ptr<OneWayDescriptorBase>& base=Ptr<OneWay

    // Clears keypoints storing in collection and OneWayDescriptorBase
    virtual void clear();

    virtual void train();

    virtual bool isMaskSupported();

    virtual void read( const FileNode &fn );
    virtual void write( FileStorage& fs ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const;
protected:
    ...
};
```

---

cv::FernDescriptorMatcher (view/add comments)

Wrapping class for computing, matching and classification of descriptors using cv::FernClassifier class.

```
class FernDescriptorMatcher : public GenericDescriptorMatcher
{
public:
    class Params
```

```
    {
    public:
        Params( int nclasses=0,
                int patchSize=FernClassifier::PATCH_SIZE,
                int signatureSize=FernClassifier::DEFAULT_SIGNATURE_SIZE,
                int nstructs=FernClassifier::DEFAULT_STRUCTS,
                int structSize=FernClassifier::DEFAULT_STRUCT_SIZE,
                int nviews=FernClassifier::DEFAULT_VIEWS,
                int compressionMethod=FernClassifier::COMPRESSION_NONE,
                const PatchGenerator& patchGenerator=PatchGenerator() );

        Params( const string& filename );

        int nclasses;
        int patchSize;
        int signatureSize;
        int nstructs;
        int structSize;
        int nviews;
        int compressionMethod;
        PatchGenerator patchGenerator;

        string filename;
    };

    FernDescriptorMatcher( const Params& params=Params() );
    virtual ~FernDescriptorMatcher();

    virtual void clear();

    virtual void train();

    virtual bool isMaskSupported();

    virtual void read( const FileNode &fn );
    virtual void write( FileStorage& fs ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const;

protected:
        ...
};
```

cv::VectorDescriptorMatcher (view/add comments)

Class used for matching descriptors that can be described as vectors in a finite-dimensional space.

```
class CV_EXPORTS VectorDescriptorMatcher : public GenericDescriptorMatcher
{
public:
    VectorDescriptorMatcher( const Ptr<DescriptorExtractor>& extractor, const Ptr<Descriptc
    virtual ~VectorDescriptorMatcher();

    virtual void add( const vector<Mat>& imgCollection,
                      vector<vector<KeyPoint> >& pointCollection );
    virtual void clear();
    virtual void train();
    virtual bool isMaskSupported();

    virtual void read( const FileNode& fn );
    virtual void write( FileStorage& fs ) const;

    virtual Ptr<GenericDescriptorMatcher> clone( bool emptyTrainData=false ) const;

protected:
    ...
};
```

Example of creating:

```
VectorDescriptorMatcher matcher( new SurfDescriptorExtractor,
                                 new BruteForceMatcher<L2<float> > );
```

13.6 Drawing Function of Keypoints and Matches

cv::drawMatches (view/add comments)

This function draws matches of keypints from two images on output image. Match is a line connecting two keypoints (circles).

```
void drawMatches( const Mat& img1, const vector<KeyPoint>& keypoints1,
           const Mat& img2, const vector<KeyPoint>& keypoints2,
           const vector<DMatch>& matches1to2, Mat& outImg,
           const Scalar& matchColor=Scalar::all(-1),
```

```
            const Scalar& singlePointColor=Scalar::all(-1),
            const vector<char>& matchesMask=vector<char>(),
            int flags=DrawMatchesFlags::DEFAULT );
```

```
void drawMatches( const Mat& img1, const vector<KeyPoint>& keypoints1,
            const Mat& img2, const vector<KeyPoint>& keypoints2,
            const vector<vector<DMatch> >& matches1to2, Mat& outImg,
            const Scalar& matchColor=Scalar::all(-1),
            const Scalar& singlePointColor=Scalar::all(-1),
            const vector<vector<char>>& matchesMask=
            vector<vector<char> >(),
            int flags=DrawMatchesFlags::DEFAULT );
```

img1 First source image.

keypoints1 Keypoints from first source image.

img2 Second source image.

keypoints2 Keypoints from second source image.

matches Matches from first image to second one, i.e. `keypoints1[i]` has corresponding point `keypoints2[matches[i]]`.

outImg Output image. Its content depends on `flags` value what is drawn in output image. See below possible `flags` bit values.

matchColor Color of matches (lines and connected keypoints). If `matchColor==Scalar::all(-1)` color will be generated randomly.

singlePointColor Color of single keypoints (circles), i.e. keypoints not having the matches. If `singlePointColor==Scalar::all(-1)` color will be generated randomly.

matchesMask Mask determining which matches will be drawn. If mask is empty all matches will be drawn.

flags Each bit of `flags` sets some feature of drawing. Possible `flags` bit values is defined by `DrawMatchesFlags`, see below.

```
struct DrawMatchesFlags
{
    enum{ DEFAULT = 0, // Output image matrix will be created (Mat::create),
                       // i.e. existing memory of output image may be reused.
                       // Two source image, matches and single keypoints
                       // will be drawn.
                       // For each keypoint only the center point will be
                       // drawn (without the circle around keypoint with
                       // keypoint size and orientation).
          DRAW_OVER_OUTIMG = 1, // Output image matrix will not be
                       // created (Mat::create). Matches will be drawn
                       // on existing content of output image.
          NOT_DRAW_SINGLE_POINTS = 2, // Single keypoints will not be drawn.
          DRAW_RICH_KEYPOINTS = 4 // For each keypoint the circle around
                       // keypoint with keypoint size and orientation will
                       // be drawn.
        };
};
```

---

cv::drawKeypoints (view/add comments)

Draw keypoints.

```
void drawKeypoints( const Mat& image,
            const vector<KeyPoint>& keypoints,
            Mat& outImg, const Scalar& color=Scalar::all(-1),
            int flags=DrawMatchesFlags::DEFAULT );
```

image Source image.

keypoints Keypoints from source image.

outImg Output image. Its content depends on `flags` value what is drawn in output image. See possible `flags` bit values.

color Color of keypoints .

flags Each bit of `flags` sets some feature of drawing. Possible `flags` bit values is defined by `DrawMatchesFlags`, see above in cv::drawMatches.

13.7 Object Categorization

Some approaches based on local 2D features and used to object categorization are described in this section.

---

cv::BOWTrainer (view/add comments)

Abstract base class for training "bag of visual words" vocabulary from a set of descriptors. See e.g. "Visual Categorization with Bags of Keypoints" of Gabriella Csurka, Christopher R. Dance, Lixin Fan, Jutta Willamowski, Cedric Bray, 2004.

```
class BOWTrainer
{
public:
    BOWTrainer(){}
    virtual ~BOWTrainer(){}

    void add( const Mat& descriptors );
    const vector<Mat>& getDescriptors() const;
    int descripotorsCount() const;

    virtual void clear();

    virtual Mat cluster() const = 0;
    virtual Mat cluster( const Mat& descriptors ) const = 0;

protected:
    ...
};
```

---

cv::BOWTrainer::add (view/add comments)

Add descriptors to training set. The training set will be clustered using `cluster` method to construct vocabulary.

```
void BOWTrainer::add( const Mat& descriptors );
```

`descriptors` Descriptors to add to training set. Each row of `descriptors` matrix is a one descriptor.

cv::BOWTrainer::getDescriptors (view/add comments)

Returns training set of descriptors.

```
const vector<Mat>& BOWTrainer::getDescriptors() const;
```

cv::BOWTrainer::descripotorsCount (view/add comments)

Returns count of all descriptors stored in the training set.

```
const vector<Mat>& BOWTrainer::descripotorsCount() const;
```

cv::BOWTrainer::cluster (view/add comments)

Cluster train descriptors. Vocabulary consists from cluster centers. So this method returns vocabulary. In first method variant the stored in object train descriptors will be clustered, in second variant – input descriptors will be clustered.

```
Mat BOWTrainer::cluster() const;
```

```
Mat BOWTrainer::cluster( const Mat& descriptors ) const;
```

descriptors Descriptors to cluster. Each row of `descriptors` matrix is a one descriptor. Descriptors will not be added to the inner train descriptor set.

cv::BOWKMeansTrainer (view/add comments)

cv::kmeans based class to train visual vocabulary using the ”bag of visual words” approach.

```
class BOWKMeansTrainer : public BOWTrainer
{
public:
    BOWKMeansTrainer( int clusterCount, const TermCriteria& termcrit=TermCriteria(),
                      int attempts=3, int flags=KMEANS_PP_CENTERS );
    virtual ~BOWKMeansTrainer(){}

    // Returns trained vocabulary (i.e. cluster centers).
    virtual Mat cluster() const;
    virtual Mat cluster( const Mat& descriptors ) const;

protected:
    ...
};
```

To gain an understanding of constructor parameters see cv::kmeans function arguments.

---

cv::BOWImgDescriptorExtractor (view/add comments)

Class to compute image descriptor using ”bad of visual words”. In few, such computing consists from the following steps: 1. Compute descriptors for given image and it’s keypoints set,
2. Find nearest visual words from vocabulary for each keypoint descriptor,
3. Image descriptor is a normalized histogram of vocabulary words encountered in the image. I.e. `i`-bin of the histogram is a frequency of `i`-word of vocabulary in the given image.

```
class BOWImgDescriptorExtractor
{
public:
    BOWImgDescriptorExtractor( const Ptr<DescriptorExtractor>& dextractor,
                               const Ptr<DescriptorMatcher>& dmatcher );
    virtual ~BOWImgDescriptorExtractor(){}

    void setVocabulary( const Mat& vocabulary );
    const Mat& getVocabulary() const;
    void compute( const Mat& image, vector<KeyPoint>& keypoints,
                  Mat& imgDescriptor,
                  vector<vector<int> >* pointIdxsOfClusters=0,
                  Mat* descriptors=0 );
    int descriptorSize() const;
    int descriptorType() const;

protected:
    ...
};
```

cv::BOWImgDescriptorExtractor::BOWImgDescriptorExtractor (view/add comments)

Constructor.

```
BOWImgDescriptorExtractor::BOWImgDescriptorExtractor(
            const Ptr<DescriptorExtractor>& dextractor,
            const Ptr<DescriptorMatcher>& dmatcher );
```

dextractor Descriptor extractor that will be used to compute descriptors for input image and it's keypoints.

dmatcher Descriptor matcher that will be used to find nearest word of trained vocabulary to each keupoints descriptor of the image.

cv::BOWImgDescriptorExtractor::setVocabulary (view/add comments)

Method to set visual vocabulary.

```
void BOWImgDescriptorExtractor::setVocabulary( const Mat& vocabulary );
```

vocabulary Vocabulary (can be trained using inheritor of cv::BOWTrainer). Each row of vocabulary is a one visual word (cluster center).

cv::BOWImgDescriptorExtractor::getVocabulary (view/add comments)

Returns set vocabulary.

```
const Mat& BOWImgDescriptorExtractor::getVocabulary() const;
```

cv::BOWImgDescriptorExtractor::compute (view/add comments)

Compute image descriptor using set visual vocabulary.

```
void BOWImgDescriptorExtractor::compute( const Mat& image,
           vector<KeyPoint>& keypoints, Mat& imgDescriptor,
           vector<vector<int> >* pointIdxsOfClusters=0,
           Mat* descriptors=0 );
```

image The image. Image descriptor will be computed for this.

keypoints Keypoints detected in the input image.

imgDescriptor This is output, i.e. computed image descriptor.

pointIdxsOfClusters Indices of keypoints which belong to the cluster, i.e. `pointIdxsOfClusters[i]` is keypoint indices which belong to the `i`-cluster (word of vocabulary) (returned if it is not 0.)

descriptors Descriptors of the image keypoints (returned if it is not 0.)

cv::BOWImgDescriptorExtractor::descriptorSize (view/add comments)

Returns image discriptor size, if vocabulary was set, and 0 otherwise.

```
int BOWImgDescriptorExtractor::descriptorSize() const;
```

cv::BOWImgDescriptorExtractor::descriptorType (view/add comments)

Returns image descriptor type.

```
int BOWImgDescriptorExtractor::descriptorType() const;
```

Chapter 14

flann. Clustering and Search in Multi-Dimensional Spaces

14.1 Fast Approximate Nearest Neighbor Search

This section documents OpenCV's interface to the FLANN[1] library. FLANN (Fast Library for Approximate Nearest Neighbors) is a library that contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. More information about FLANN can be found in [**?**].

---

cv::flann::Index_

The FLANN nearest neighbor index class. This class is templated with the type of elements for which the index is built.

```
namespace cv
{
namespace flann
{
    template <typename T>
    class Index_
    {
    public:
            Index_(const Mat& features, const IndexParams& params);

            ~Index_();
```

[1] http://people.cs.ubc.ca/~mariusm/flann

```
            void knnSearch(const vector<T>& query,
                           vector<int>& indices,
                           vector<float>& dists,
                           int knn,
                           const SearchParams& params);
            void knnSearch(const Mat& queries,
                           Mat& indices,
                           Mat& dists,
                           int knn,
                           const SearchParams& params);

            int radiusSearch(const vector<T>& query,
                             vector<int>& indices,
                             vector<float>& dists,
                             float radius,
                             const SearchParams& params);
            int radiusSearch(const Mat& query,
                             Mat& indices,
                             Mat& dists,
                             float radius,
                             const SearchParams& params);

            void save(std::string filename);

            int veclen() const;

            int size() const;

            const IndexParams* getIndexParameters();
    };

    typedef Index_<float> Index;

} } // namespace cv::flann
```

---

cvflann::Index_$< T >$::Index_

Constructs a nearest neighbor search index for a given dataset.

```
Index_<T>::Index_(const Mat& features, const IndexParams& params);
```

features Matrix of containing the features(points) to index. The size of the matrix is num_features x feature_dimensionality and the data type of the elements in the matrix must coincide with the type of the index.

params Structure containing the index parameters. The type of index that will be constructed depends on the type of this parameter. The possible parameter types are:

LinearIndexParams When passing an object of this type, the index will perform a linear, brute-force search.

```
struct LinearIndexParams : public IndexParams
{
};
```

KDTreeIndexParams When passing an object of this type the index constructed will consist of a set of randomized kd-trees which will be searched in parallel.

```
struct KDTreeIndexParams : public IndexParams
{
    KDTreeIndexParams( int trees = 4 );
};
```

trees The number of parallel kd-trees to use. Good values are in the range [1..16]

KMeansIndexParams When passing an object of this type the index constructed will be a hierarchical k-means tree.

```
struct KMeansIndexParams : public IndexParams
{
    KMeansIndexParams(
        int branching = 32,
        int iterations = 11,
        flann_centers_init_t centers_init = CENTERS_RANDOM,
        float cb_index = 0.2 );
};
```

branching The branching factor to use for the hierarchical k-means tree

iterations The maximum number of iterations to use in the k-means clustering stage when building the k-means tree. A value of -1 used here means that the k-means clustering should be iterated until convergence

centers_init The algorithm to use for selecting the initial centers when performing a k-means clustering step. The possible values are `CENTERS_RANDOM` (picks the initial cluster centers randomly), `CENTERS_GONZALES` (picks the initial centers using Gonzales' algorithm) and `CENTERS_KMEANSPP` (picks the initial centers using the algorithm suggested in [**?**])

cb_index This parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When `cb_index` is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

CompositeIndexParams When using a parameters object of this type the index created combines the randomized kd-trees and the hierarchical k-means tree.

```
struct CompositeIndexParams : public IndexParams
{
    CompositeIndexParams(
        int trees = 4,
        int branching = 32,
        int iterations = 11,
        flann_centers_init_t centers_init = CENTERS_RANDOM,
        float cb_index = 0.2 );
};
```

AutotunedIndexParams When passing an object of this type the index created is automatically tuned to offer the best performance, by choosing the optimal index type (randomized kd-trees, hierarchical kmeans, linear) and parameters for the dataset provided.

```
struct AutotunedIndexParams : public IndexParams
{
    AutotunedIndexParams(
        float target_precision = 0.9,
        float build_weight = 0.01,
        float memory_weight = 0,
        float sample_fraction = 0.1 );
};
```

target_precision Is a number between 0 and 1 specifying the percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

build_weight Specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times.

memory_weight Is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

sample_fraction Is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm on the full dataset gives the most accurate results, but for very large datasets can take longer than desired. In such case using just a fraction of the data helps speeding up this algorithm while still giving good approximations of the optimum parameters.

SavedIndexParams This object type is used for loading a previously saved index from the disk.

```
struct SavedIndexParams : public IndexParams
{
    SavedIndexParams( std::string filename );
};
```

filename The filename in which the index was saved.

---

cv::flann::Index_$< T >$::knnSearch

Performs a K-nearest neighbor search for a given query point using the index.

```
void Index_<T>::knnSearch(const vector<T>& query,
          vector<int>& indices,
          vector<float>& dists,
          int knn,
          const SearchParams& params);
void Index_<T>::knnSearch(const Mat& queries,
          Mat& indices, Mat& dists,
          int knn, const SearchParams& params);
```

query The query point

indices Vector that will contain the indices of the K-nearest neighbors found. It must have at least knn size.

dists Vector that will contain the distances to the K-nearest neighbors found. It must have at least knn size.

knn Number of nearest neighbors to search for.

params Search parameters

```
struct SearchParams {
        SearchParams(int checks = 32);
};
```

checks The number of times the tree(s) in the index should be recursively traversed. A higher value for this parameter would give better search precision, but also take more time. If automatic configuration was used when the index was created, the number of checks required to achieve the specified precision was also computed, in which case this parameter is ignored.

---

cv::flann::Index_< T >::radiusSearch

Performs a radius nearest neighbor search for a given query point.

```
int Index_<T>::radiusSearch(const vector<T>& query,
           vector<int>& indices,
           vector<float>& dists,
           float radius,
           const SearchParams& params);
int Index_<T>::radiusSearch(const Mat& query,
           Mat& indices,
           Mat& dists,
           float radius,
           const SearchParams& params);
```

query The query point

indices Vector that will contain the indices of the points found within the search radius in decreasing order of the distance to the query point. If the number of neighbors in the search radius is bigger than the size of this vector, the ones that don't fit in the vector are ignored.

dists Vector that will contain the distances to the points found within the search radius

radius The search radius

params Search parameters

cv::flann::Index_< T >::save

Saves the index to a file.

```
void Index_<T>::save(std::string filename);
```

filename The file to save the index to

cv::flann::Index_< T >::getIndexParameters

Returns the index paramreters. This is usefull in case of autotuned indices, when the parameters computed can be retrived using this method.

```
const IndexParams* Index_<T>::getIndexParameters();
```

14.2 Clustering

cv::cv::flann::hierarchicalClustering¡ET,DT¿ (view/add comments)

Clusters the given points by constructing a hierarchical k-means tree and choosing a cut in the tree that minimizes the cluster's variance.

```
int hierarchicalClustering<ET,DT>(const Mat& features, Mat& centers,
          const KMeansIndexParams& params);
```

features The points to be clustered. The matrix must have elements of type ET.

centers The centers of the clusters obtained. The matrix must have type DT. The number of rows in this matrix represents the number of clusters desired, however, because of the way the cut in the hierarchical tree is chosen, the number of clusters computed will be the highest number of the form `(branching-1)*k+1` that's lower than the number of clusters desired, where `branching` is the tree's branching factor (see description of the KMeansIndexParams).

`params` Parameters used in the construction of the hierarchical k-means tree

The function returns the number of clusters computed.

Chapter 15

objdetect. Object Detection

15.1 Cascade Classification

cv::FeatureEvaluator (view/add comments)

Base class for computing feature values in cascade classifiers.

```
class CV_EXPORTS FeatureEvaluator
{
public:
    enum { HAAR = 0, LBP = 1 }; // supported feature types
    virtual ~FeatureEvaluator(); // destructor
    virtual bool read(const FileNode& node);
    virtual Ptr<FeatureEvaluator> clone() const;
    virtual int getFeatureType() const;

    virtual bool setImage(const Mat& img, Size origWinSize);
    virtual bool setWindow(Point p);

    virtual double calcOrd(int featureIdx) const;
    virtual int calcCat(int featureIdx) const;

    static Ptr<FeatureEvaluator> create(int type);
};
```

cv::FeatureEvaluator::read (view/add comments)

Reads parameters of the features from a FileStorage node.

```
bool FeatureEvaluator::read(const FileNode& node);
```

node File node from which the feature parameters are read.

---

cv::FeatureEvaluator::clone (view/add comments)

Returns a full copy of the feature evaluator.

```
Ptr<FeatureEvaluator> FeatureEvaluator::clone() const;
```

---

cv::FeatureEvaluator::getFeatureType (view/add comments)

Returns the feature type (HAAR or LBP for now).

```
int FeatureEvaluator::getFeatureType() const;
```

---

cv::FeatureEvaluator::setImage (view/add comments)

Sets the image in which to compute the features.

```
bool FeatureEvaluator::setImage(const Mat& img, Size origWinSize);
```

img Matrix of type `CV_8UC1` containing the image in which to compute the features.

origWinSize Size of training images.

cv::FeatureEvaluator::setWindow (view/add comments)

Sets window in the current image in which the features will be computed (called by cv::CascadeClassifier::runAt).

```
bool FeatureEvaluator::setWindow(Point p);
```

p The upper left point of window in which the features will be computed. Size of the window is equal to size of training images.

cv::FeatureEvaluator::calcOrd (view/add comments)

Computes value of an ordered (numerical) feature.

```
double FeatureEvaluator::calcOrd(int featureIdx) const;
```

featureIdx Index of feature whose value will be computed.

Returns computed value of ordered feature.

cv::FeatureEvaluator::calcCat (view/add comments)

Computes value of a categorical feature.

```
int FeatureEvaluator::calcCat(int featureIdx) const;
```

featureIdx Index of feature whose value will be computed.

Returns computed label of categorical feature, i.e. value from [0,... (number of categories - 1)].

cv::FeatureEvaluator::create (view/add comments)

Constructs feature evaluator.

```
static Ptr<FeatureEvaluator> FeatureEvaluator::create(int type);
```

type Type of features evaluated by cascade (HAAR or LBP for now).

cv::CascadeClassifier (view/add comments)

The cascade classifier class for object detection.

```
class CascadeClassifier
{
public:
        // structure for storing tree node
    struct CV_EXPORTS DTreeNode
    {
        int featureIdx; // feature index on which is a split
        float threshold; // split threshold of ordered features only
        int left; // left child index in the tree nodes array
        int right; // right child index in the tree nodes array
    };

    // structure for storing desision tree
    struct CV_EXPORTS DTree
    {
        int nodeCount; // nodes count
    };

    // structure for storing cascade stage (BOOST only for now)
    struct CV_EXPORTS Stage
    {
        int first; // first tree index in tree array
        int ntrees; // number of trees
        float threshold; // treshold of stage sum
    };

    enum { BOOST = 0 }; // supported stage types

    // mode of detection (see parameter flags in function HaarDetectObjects)
```

```
    enum { DO_CANNY_PRUNING = CV_HAAR_DO_CANNY_PRUNING,
           SCALE_IMAGE = CV_HAAR_SCALE_IMAGE,
           FIND_BIGGEST_OBJECT = CV_HAAR_FIND_BIGGEST_OBJECT,
           DO_ROUGH_SEARCH = CV_HAAR_DO_ROUGH_SEARCH };

    CascadeClassifier(); // default constructor
    CascadeClassifier(const string& filename);
    ~CascadeClassifier(); // destructor

    bool empty() const;
    bool load(const string& filename);
    bool read(const FileNode& node);

    void detectMultiScale( const Mat& image, vector<Rect>& objects,
                           double scaleFactor=1.1, int minNeighbors=3,
                                                   int flags=0, Size minSize=Size());

    bool setImage( Ptr<FeatureEvaluator>&, const Mat& );
    int runAt( Ptr<FeatureEvaluator>&, Point );

    bool is_stump_based; // true, if the trees are stumps

    int stageType; // stage type (BOOST only for now)
    int featureType; // feature type (HAAR or LBP for now)
    int ncategories; // number of categories (for categorical features only)
    Size origWinSize; // size of training images

    vector<Stage> stages; // vector of stages (BOOST for now)
    vector<DTree> classifiers; // vector of decision trees
    vector<DTreeNode> nodes; // vector of tree nodes
    vector<float> leaves; // vector of leaf values
    vector<int> subsets; // subsets of split by categorical feature

    Ptr<FeatureEvaluator> feval; // pointer to feature evaluator
    Ptr<CvHaarClassifierCascade> oldCascade; // pointer to old cascade
};
```

---

cv::CascadeClassifier::CascadeClassifier (view/add comments)

Loads the classifier from file.

```
CascadeClassifier::CascadeClassifier(const string& filename);
```

filename Name of file from which classifier will be load.

cv::CascadeClassifier::empty (view/add comments)

Checks if the classifier has been loaded or not.

```
bool CascadeClassifier::empty() const;
```

cv::CascadeClassifier::load (view/add comments)

Loads the classifier from file. The previous content is destroyed.

```
bool CascadeClassifier::load(const string& filename);
```

filename Name of file from which classifier will be load. File may contain as old haar classifier (trained by haartraining application) or new cascade classifier (trained traincascade application).

cv::CascadeClassifier::read (view/add comments)

Reads the classifier from a FileStorage node. File may contain a new cascade classifier (trained traincascade application) only.

```
bool CascadeClassifier::read(const FileNode& node);
```

cv::CascadeClassifier::detectMultiScale (view/add comments)

Detects objects of different sizes in the input image. The detected objects are returned as a list of rectangles.

```
void CascadeClassifier::detectMultiScale( const Mat& image,
vector<Rect>& objects, double scaleFactor=1.1, int minNeighbors=3, int
flags=0, Size minSize=Size());
```

image Matrix of type CV_8U containing the image in which to detect objects.

objects Vector of rectangles such that each rectangle contains the detected object.

scaleFactor Specifies how much the image size is reduced at each image scale.

minNeighbors Speficifes how many neighbors should each candiate rectangle have to retain it.

flags This parameter is not used for new cascade and have the same meaning for old cascade as in function cvHaarDetectObjects.

minSize The minimum possible object size. Objects smaller than that are ignored.

---

cv::CascadeClassifier::setImage (view/add comments)

Sets the image for detection (called by detectMultiScale at each image level).

```
bool CascadeClassifier::setImage( Ptr<FeatureEvaluator>& feval, const
Mat& image );
```

feval Pointer to feature evaluator which is used for computing features.

image Matrix of type CV_8UC1 containing the image in which to compute the features.

---

cv::CascadeClassifier::runAt (view/add comments)

Runs the detector at the specified point (the image that the detector is working with should be set by setImage).

```
int CascadeClassifier::runAt( Ptr<FeatureEvaluator>& feval, Point pt );
```

feval Feature evaluator which is used for computing features.

pt The upper left point of window in which the features will be computed. Size of the window is equal to size of training images.

Returns: 1 - if cascade classifier detects object in the given location. -si - otherwise. si is an index of stage which first predicted that given window is a background image.

---

cv::groupRectangles (view/add comments)

Groups the object candidate rectangles

```
void groupRectangles(vector<Rect>& rectList,
           int groupThreshold, double eps=0.2);
```

rectList The input/output vector of rectangles. On output there will be retained and grouped rectangles

groupThreshold The minimum possible number of rectangles, minus 1, in a group of rectangles to retain it.

eps The relative difference between sides of the rectangles to merge them into a group

The function is a wrapper for a generic function cv::partition. It clusters all the input rectangles using the rectangle equivalence criteria, that combines rectangles that have similar sizes and similar locations (the similarity is defined by `eps`). When `eps=0`, no clustering is done at all. If `eps` $\rightarrow +\inf$, all the rectangles will be put in one cluster. Then, the small clusters, containing less than or equal to `groupThreshold` rectangles, will be rejected. In each other cluster the average rectangle will be computed and put into the output rectangle list.

Chapter 16

video. Video Analysis

16.1 Motion Analysis and Object Tracking

cv::calcOpticalFlowPyrLK (view/add comments)

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids

```
void calcOpticalFlowPyrLK( const Mat& prevImg, const Mat& nextImg,
           const vector<Point2f>& prevPts, vector<Point2f>& nextPts,
           vector<uchar>& status, vector<float>& err,
           Size winSize=Size(15,15), int maxLevel=3,
           TermCriteria criteria=TermCriteria(
           TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01),
           double derivLambda=0.5, int flags=0 );
```

prevImg The first 8-bit single-channel or 3-channel input image

nextImg The second input image of the same size and the same type as `prevImg`

prevPts Vector of points for which the flow needs to be found

nextPts The output vector of points containing the calculated new positions of the input features in the second image

status The output status vector. Each element of the vector is set to 1 if the flow for the corresponding features has been found, 0 otherwise

err The output vector that will contain the difference between patches around the original and moved points

winSize Size of the search window at each pyramid level

maxLevel 0-based maximal pyramid level number. If 0, pyramids are not used (single level), if 1, two levels are used etc.

criteria Specifies the termination criteria of the iterative search algorithm (after the specified maximum number of iterations `criteria.maxCount` or when the search window moves by less than `criteria.epsilon`

derivLambda The relative weight of the spatial image derivatives impact to the optical flow estimation. If `derivLambda=0`, only the image intensity is used, if `derivLambda=1`, only derivatives are used. Any other values between 0 and 1 means that both derivatives and the image intensity are used (in the corresponding proportions).

flags The operation flags:

OPTFLOW_USE_INITIAL_FLOW use initial estimations stored in `nextPts`. If the flag is not set, then initially `nextPts` ← `prevPts`

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids, see [**?**].

---

cv::calcOpticalFlowFarneback (view/add comments)

Computes dense optical flow using Gunnar Farneback's algorithm

```
void calcOpticalFlowFarneback( const Mat& prevImg, const Mat& nextImg,
           Mat& flow, double pyrScale, int levels, int winsize,
           int iterations, int polyN, double polySigma, int flags );
```

prevImg The first 8-bit single-channel input image

nextImg The second input image of the same size and the same type as `prevImg`

flow The computed flow image; will have the same size as `prevImg` and type `CV_32FC2`

pyrScale Specifies the image scale (¡1) to build the pyramids for each image. `pyrScale=0.5` means the classical pyramid, where each next layer is twice smaller than the previous

levels The number of pyramid layers, including the initial image. `levels=1` means that no extra layers are created and only the original images are used

winsize The averaging window size; The larger values increase the algorithm robustness to image noise and give more chances for fast motion detection, but yield more blurred motion field

iterations The number of iterations the algorithm does at each pyramid level

polyN Size of the pixel neighborhood used to find polynomial expansion in each pixel. The larger values mean that the image will be approximated with smoother surfaces, yielding more robust algorithm and more blurred motion field. Typically, `polyN`=5 or 7

polySigma Standard deviation of the Gaussian that is used to smooth derivatives that are used as a basis for the polynomial expansion. For `polyN=5` you can set `polySigma=1.1`, for `polyN=7` a good value would be `polySigma=1.5`

flags The operation flags; can be a combination of the following:

OPTFLOW_USE_INITIAL_FLOW Use the input `flow` as the initial flow approximation

OPTFLOW_FARNEBACK_GAUSSIAN Use a Gaussian `winsize` $\times$ `winsize` filter instead of box filter of the same size for optical flow estimation. Usually, this option gives more accurate flow than with a box filter, at the cost of lower speed (and normally `winsize` for a Gaussian window should be set to a larger value to achieve the same level of robustness)

The function finds optical flow for each `prevImg` pixel using the alorithm so that

$$\texttt{prevImg}(x,y) \sim \texttt{nextImg}(\texttt{flow}(x,y)[0], \texttt{flow}(x,y)[1])$$

---

cv::updateMotionHistory (view/add comments)

Updates the motion history image by a moving silhouette.

```
void updateMotionHistory( const Mat& silhouette, Mat& mhi,
            double timestamp, double duration );
```

silhouette Silhouette mask that has non-zero pixels where the motion occurs

mhi Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp Current time in milliseconds or other units

duration Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current `timestamp`, while the pixels where motion happened last time a long time ago are cleared.

The function, together with cv::calcMotionGradient and cv::calcGlobalOrientation, implements the motion templates technique, described in [**?**] and [**?**]. See also the OpenCV sample `motempl.c` that demonstrates the use of all the motion template functions.

---

cv::calcMotionGradient (view/add comments)

Calculates the gradient orientation of a motion history image.

```
void calcMotionGradient( const Mat& mhi, Mat& mask,
            Mat& orientation,
            double delta1, double delta2,
            int apertureSize=3 );
```

mhi Motion history single-channel floating-point image

mask The output mask image; will have the type `CV_8UC1` and the same size as `mhi`. Its non-zero elements will mark pixels where the motion gradient data is correct

orientation The output motion gradient orientation image; will have the same type and the same size as `mhi`. Each pixel of it will the motion orientation in degrees, from 0 to 360.

delta1, delta2 The minimal and maximal allowed difference between `mhi` values within a pixel neighorhood. That is, the function finds the minimum ($m(x,y)$) and maximum ($M(x,y)$) `mhi` values over 3×3 neighborhood of each pixel and marks the motion orientation at (x,y) as valid only if

$$\min(\texttt{delta1}, \texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1}, \texttt{delta2}).$$

apertureSize The aperture size of cv::Sobel operator

The function calculates the gradient orientation at each pixel (x, y) as:

$$\texttt{orientation}(x,y) = \arctan \frac{d\texttt{mhi}/dy}{d\texttt{mhi}/dx}$$

(in fact, cv::fastArctan and cv::phase are used, so that the computed angle is measured in degrees and covers the full range 0..360). Also, the `mask` is filled to indicate pixels where the computed angle is valid.

---

cv::calcGlobalOrientation (view/add comments)

Calculates the global motion orientation in some selected region.

```
double calcGlobalOrientation( const Mat& orientation, const Mat& mask,
            const Mat& mhi, double timestamp,
            double duration );
```

orientation Motion gradient orientation image, calculated by the function cv::calcMotionGradient

mask Mask image. It may be a conjunction of a valid gradient mask, also calculated by cv::calcMotionGradient, and the mask of the region, whose direction needs to be calculated

mhi The motion history image, calculated by cv::updateMotionHistory

timestamp The timestamp passed to cv::updateMotionHistory

duration Maximal duration of motion track in milliseconds, passed to cv::updateMotionHistory

The function calculates the average motion direction in the selected region and returns the angle between 0 degrees and 360 degrees. The average direction is computed from the weighted orientation histogram, where a recent motion has larger weight and the motion occurred in the past has smaller weight, as recorded in `mhi`.

---

cv::CamShift (view/add comments)

Finds the object center, size, and orientation

```
RotatedRect CamShift( const Mat& probImage, Rect& window,
            TermCriteria criteria );
```

probImage Back projection of the object histogram; see cv::calcBackProject

window Initial search window

criteria Stop criteria for the underlying cv::meanShift

The function implements the CAMSHIFT object tracking algrorithm [**?**]. First, it finds an object center using cv::meanShift and then adjust the window size and finds the optimal rotation. The function returns the rotated rectangle structure that includes the object position, size and the orientation. The next position of the search window can be obtained with `RotatedRect::boundingRect()`.

See the OpenCV sample `camshiftdemo.c` that tracks colored objects.

---

cv::meanShift (view/add comments)

Finds the object on a back projection image.

```
int meanShift( const Mat& probImage, Rect& window,
            TermCriteria criteria );
```

probImage Back projection of the object histogram; see cv::calcBackProject

window Initial search window

criteria The stop criteria for the iterative search algorithm

The function implements iterative object search algorithm. It takes the object back projection on input and the initial position. The mass center in `window` of the back projection image is computed and the search window center shifts to the mass center. The procedure is repeated until the specified number of iterations `criteria.maxCount` is done or until the window center shifts by less than `criteria.epsilon`. The algorithm is used inside cv::CamShift and, unlike cv::CamShift, the search window size or orientation do not change during the search. You can simply pass the output of cv::calcBackProject to this function, but better results can be obtained if you pre-filter the back projection and remove the noise (e.g. by retrieving connected components with cv::findContours, throwing away contours with small area (cv::contourArea) and rendering the remaining contours with cv::drawContours)

cv::KalmanFilter (view/add comments)

Kalman filter class

```
class KalmanFilter
{
public:
    KalmanFilter();
    KalmanFilter(int dynamParams, int measureParams, int controlParams=0);
    void init(int dynamParams, int measureParams, int controlParams=0);
    // predicts statePre from statePost
    const Mat& predict(const Mat& control=Mat());
    // corrects statePre based on the input measurement vector
    // and stores the result to statePost.
    const Mat& correct(const Mat& measurement);

    Mat statePre;           // predicted state (x'(k)):
                            //    x(k)=A*x(k-1)+B*u(k)
    Mat statePost;          // corrected state (x(k)):
                            //    x(k)=x'(k)+K(k)*(z(k)-H*x'(k))
    Mat transitionMatrix;   // state transition matrix (A)
    Mat controlMatrix;      // control matrix (B)
                            //   (it is not used if there is no control)
    Mat measurementMatrix;  // measurement matrix (H)
    Mat processNoiseCov;    // process noise covariance matrix (Q)
    Mat measurementNoiseCov;// measurement noise covariance matrix (R)
    Mat errorCovPre;        // priori error estimate covariance matrix (P'(k)):
                            //    P'(k)=A*P(k-1)*At + Q)*/
    Mat gain;               // Kalman gain matrix (K(k)):
                            //    K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)
    Mat errorCovPost;       // posteriori error estimate covariance matrix (P(k)):
                            //    P(k)=(I-K(k)*H)*P'(k)
    ...
};
```

The class implements standard Kalman filter http://en.wikipedia.org/wiki/Kalman_filter. However, you can modify `transitionMatrix`, `controlMatrix` and `measurementMatrix` to get the extended Kalman filter functionality. See the OpenCV sample `kalman.c`

Chapter 17

highgui. High-level GUI and Media I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and ”remember” their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

17.1 User Interface

---

cv::createTrackbar (view/add comments)

Creates a trackbar and attaches it to the specified window

```
int createTrackbar( const string& trackbarname,
           const string& winname,
           int* value, int count,
           TrackbarCallback onChange CV_DEFAULT(0),
           void* userdata CV_DEFAULT(0));
```

trackbarname Name of the created trackbar.

winname Name of the window which will be used as a parent of the created trackbar.

value The optional pointer to an integer variable, whose value will reflect the position of the slider. Upon creation, the slider position is defined by this variable.

count The maximal position of the slider. The minimal position is always 0.

onChange Pointer to the function to be called every time the slider changes position. This function should be prototyped as `void Foo(int,void*);`, where the first parameter is the trackbar position and the second parameter is the user data (see the next parameter). If the callback is NULL pointer, then no callbacks is called, but only `value` is updated

userdata The user data that is passed as-is to the callback; it can be used to handle trackbar events without using global variables

The function `createTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable `value` to be syncronized with trackbar position and specifies a callback function `onChange` to be called on the trackbar position change. The created trackbar is displayed on the top of the given window.

[Qt Backend Only] qt-specific details:

winname Name of the window which will be used as a parent for created trackbar. Can be NULL if the trackbar should be attached to the control panel.

The created trackbar is displayed at the bottom of the given window if *winname* is correctly provided, or displayed on the control panel if *winname* is NULL.

By clicking on the label of each trackbar, it is possible to edit the trackbar's value manually for a more accurate control of it.

---

cv::getTrackbarPos (view/add comments)

Returns the trackbar position.

```
int getTrackbarPos( const string& trackbarname,
           const string& winname );
```

trackbarname Name of the trackbar.

winname Name of the window which is the parent of the trackbar.

The function returns the current position of the specified trackbar.

[Qt Backend Only] qt-specific details:

winname Name of the window which is the parent of the trackbar. Can be NULL if the trackbar is attached to the control panel.

---

cv::imshow (view/add comments)

Displays the image in the specified window

```
void imshow( const string& winname,
             const Mat& image );
```

winname Name of the window.

image Image to be shown.

The function `imshow` displays the image in the specified window. If the window was created with the `CV_WINDOW_AUTOSIZE` flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

---

cv::namedWindow (view/add comments)

Creates a window.

```
void namedWindow( const string& winname,
                  int flags );
```

name Name of the window in the window caption that may be used as a window identifier.

flags Flags of the window. Currently the only supported flag is CV_WINDOW_AUTOSIZE. If this is set, the window size is automatically adjusted to fit the displayed image (see imshow), and the user can not change the window size manually.

The function namedWindow creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing.

[Qt Backend Only] qt-specific details:

flags Flags of the window. Currently the supported flags are:

CV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE: CV_WINDOW_NORMAL let the user resize the window, whereas CV_WINDOW_AUTOSIZE adjusts automatically the window's size to fit the displayed image (see ShowImage), and the user can not change the window size manually.

CV_WINDOW_FREERATIO or CV_WINDOW_KEEPRATIO: CV_WINDOW_FREERATIO adjust the image without respect the its ration, whereas CV_WINDOW_KEEPRATIO keep the image's ratio.

CV_GUI_NORMAL or CV_GUI_EXPANDED: CV_GUI_NORMAL is the old way to draw the window without statusbar and toolbar, whereas CV_GUI_EXPANDED is the new enhance GUI.

This parameter is optional. The default flags set for a new window are CV_WINDOW_AUTOSIZE, CV_WINDOW_KEEPRATIO, and CV_GUI_EXPANDED.

However, if you want to modify the flags, you can combine them using OR operator, ie:

```
V_WINDOW_NORMALV_GUI_NORMAL
```

cv::setTrackbarPos (view/add comments)

Sets the trackbar position.

```
void setTrackbarPos( const string& trackbarname,
           const string& winname, int pos );
```

trackbarname Name of the trackbar.

winname Name of the window which is the parent of trackbar.

pos The new position.

The function sets the position of the specified trackbar in the specified window.

[Qt Backend Only] qt-specific details:

winname Name of the window which is the parent of trackbar. Can be NULL if the trackbar is attached to the control panel.

cv::waitKey (view/add comments)

Waits for a pressed key.

```
int waitKey(int delay=0);
```

delay Delay in milliseconds. 0 is the special value that means ”forever”

The function `waitKey` waits for key event infinitely (when `delay` ≤ 0) or for `delay` milliseconds, when it's positive. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

Note 2: The function only works if there is at least one HighGUI window created and the window is active. If there are several HighGUI windows, any of them can be active.

17.2 Reading and Writing Images and Video

cv::imdecode (view/add comments)

Reads an image from a buffer in memory.

```
Mat imdecode( const Mat& buf,
              int flags );
```

buf The input array of vector of bytes

flags The same flags as in imread

The function reads image from the specified buffer in memory. If the buffer is too short or contains invalid data, the empty matrix will be returned.

See imread for the list of supported formats and the flags description.

cv::imencode (view/add comments)

Encode an image into a memory buffer.

```
bool imencode( const string& ext,
               const Mat& img,
               vector<uchar>& buf,
               const vector<int>& params=vector<int>());
```

ext The file extension that defines the output format

img The image to be written

buf The output buffer; resized to fit the compressed image

params The format-specific parameters; see imwrite

The function compresses the image and stores it in the memory buffer, which is resized to fit the result. See imwrite for the list of supported formats and the flags description.

cv::imread (view/add comments)

Loads an image from a file.

```
Mat imread( const string& filename,
            int flags=1 );
```

filename Name of file to be loaded.

flags Specifies color type of the loaded image:

- **>0** the loaded image is forced to be a 3-channel color image
- **=0** the loaded image is forced to be grayscale
- **<0** the loaded image will be loaded as-is (note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB if $flags \geq 0$).

The function `imread` loads an image from the specified file and returns it. If the image can not be read (because of missing file, improper permissions, unsupported or invalid format), the function returns empty matrix (`Mat::data==NULL`).Currently, the following file formats are supported:

- Windows bitmaps - `*.bmp, *.dib` (always supported)
- JPEG files - `*.jpeg, *.jpg, *.jpe` (see **Note2**)
- JPEG 2000 files - `*.jp2` (see **Note2**)
- Portable Network Graphics - `*.png` (see **Note2**)
- Portable image format - `*.pbm, *.pgm, *.ppm` (always supported)
- Sun rasters - `*.sr, *.ras` (always supported)
- TIFF files - `*.tiff, *.tif` (see **Note2**)

Note1: The function determines type of the image by the content, not by the file extension.
Note2: On Windows and MacOSX the shipped with OpenCV image codecs (libjpeg, libpng, libtiff and libjasper) are used by default; so OpenCV can always read JPEGs, PNGs and TIFFs. On MacOSX there is also the option to use native MacOSX image readers. But beware that

currently these native image loaders give images with somewhat different pixel values, because of the embedded into MacOSX color management.

On Linux, BSD flavors and other Unix-like open-source operating systems OpenCV looks for the supplied with OS image codecs. Please, install the relevant packages (do not forget the development files, e.g. "libjpeg-dev" etc. in Debian and Ubuntu) in order to get the codec support, or turn on `OPENCV_BUILD_3RDPARTY_LIBS` flag in CMake.

---

cv::imwrite (view/add comments)

Saves an image to a specified file.

```
bool imwrite( const string& filename,
              const Mat& img,
              const vector<int>& params=vector<int>());
```

`filename` Name of the file.

`img` The image to be saved.

`params` The format-specific save parameters, encoded as pairs `paramId_1, paramValue_1, paramId_2, paramValue_2, ...`. The following parameters are currently supported:

- In the case of JPEG it can be a quality (`CV_IMWRITE_JPEG_QUALITY`), from 0 to 100 (the higher is the better), 95 by default.
- In the case of PNG it can be the compression level (`CV_IMWRITE_PNG_COMPRESSION`), from 0 to 9 (the higher value means smaller size and longer compression time), 3 by default.
- In the case of PPM, PGM or PBM it can a binary format flag (`CV_IMWRITE_PXM_BINARY`), 0 or 1, 1 by default.

The function `imwrite` saves the image to the specified file. The image format is chosen based on the `filename` extension, see imread for the list of extensions. Only 8-bit (or 16-bit in the case of PNG, JPEG 2000 and TIFF) single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use Mat::convertTo , and cvtColor to convert it before saving, or use the universal XML I/O functions to save the image to XML or YAML format.

cv::VideoCapture (view/add comments)

Class for video capturing from video files or cameras

```
class VideoCapture
{
public:
    // the default constructor
    VideoCapture();
    // the constructor that opens video file
    VideoCapture(const string& filename);
    // the constructor that starts streaming from the camera
    VideoCapture(int device);

    // the destructor
    virtual ~VideoCapture();

    // opens the specified video file
    virtual bool open(const string& filename);

    // starts streaming from the specified camera by its id
    virtual bool open(int device);

    // returns true if the file was open successfully or if the camera
    // has been initialized succesfully
    virtual bool isOpened() const;

    // closes the camera stream or the video file
    // (automatically called by the destructor)
    virtual void release();

    // grab the next frame or a set of frames from a multi-head camera;
    // returns false if there are no more frames
    virtual bool grab();
    // reads the frame from the specified video stream
    // (non-zero channel is only valid for multi-head camera live streams)
    virtual bool retrieve(Mat& image, int channel=0);
    // equivalent to grab() + retrieve(image, 0);
    virtual VideoCapture& operator >> (Mat& image);

    // sets the specified property propId to the specified value
    virtual bool set(int propId, double value);
    // retrieves value of the specified property
    virtual double get(int propId);
```

```
protected:
    ...
};
```

The class provides C++ video capturing API. Here is how the class can be used:

```
#include "cv.h"
#include "highgui.h"

using namespace cv;

int main(int, char**)
{
    VideoCapture cap(0); // open the default camera
    if(!cap.isOpened())  // check if we succeeded
        return -1;

    Mat edges;
    namedWindow("edges",1);
    for(;;)
    {
        Mat frame;
        cap >> frame; // get a new frame from camera
        cvtColor(frame, edges, CV_BGR2GRAY);
        GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
        Canny(edges, edges, 0, 30, 3);
        imshow("edges", edges);
        if(waitKey(30) >= 0) break;
    }
    // the camera will be deinitialized automatically in VideoCapture destructor
    return 0;
}
```

---

cv::VideoCapture::VideoCapture (view/add comments)

```
VideoCapture::VideoCapture();
VideoCapture::VideoCapture(const string& filename);
VideoCapture::VideoCapture(int device);
```

filename TOWRITE

device TOWRITE

VideoCapture constructors.

---

cv::VideoCapture::get (view/add comments)

```
double VideoCapture::get(int property_id);
```

property_id Property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
- **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
- **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
- **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
- **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
- **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
- **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB
- **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
- **CV_CAP_PROP_RECTIFICATION** TOWRITE (note: only supported by DC1394 v 2.x backend currently)

Note that when querying a property which is unsupported by the backend used by the VideoCapture class, the value 0 is returned.

cv::VideoCapture::set (view/add comments)

```
bool VideoCapture::set(int property_id, double value);
```

property_id Property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
- **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
- **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
- **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
- **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
- **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
- **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB
- **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
- **CV_CAP_PROP_RECTIFICATION** TOWRITE (note: only supported by DC1394 v 2.x backend currently)

value value of the property.

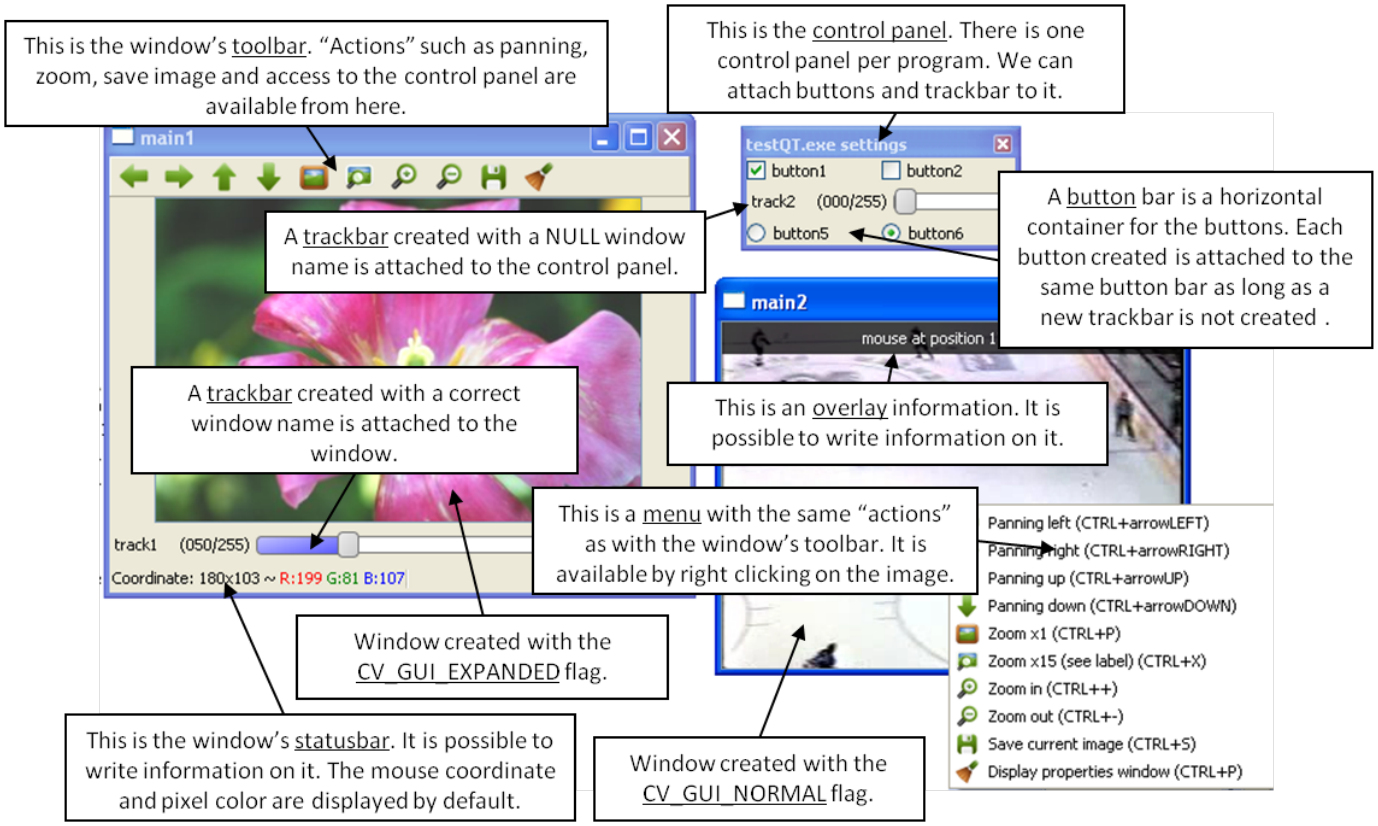
Sets a property in the VideoCapture backend.

cv::VideoWriter (view/add comments)

Video writer class

```
class VideoWriter
{
public:
    // default constructor
    VideoWriter();
    // constructor that calls open
    VideoWriter(const string& filename, int fourcc,
                double fps, Size frameSize, bool isColor=true);

    // the destructor
    virtual ~VideoWriter();

    // opens the file and initializes the video writer.
    // filename - the output file name.
    // fourcc - the codec
    // fps - the number of frames per second
    // frameSize - the video frame size
    // isColor - specifies whether the video stream is color or grayscale
    virtual bool open(const string& filename, int fourcc,
                      double fps, Size frameSize, bool isColor=true);

    // returns true if the writer has been initialized successfully
    virtual bool isOpened() const;

    // writes the next video frame to the stream
    virtual VideoWriter& operator << (const Mat& image);

protected:
    ...
};
```

17.3 Qt new functions



This figure explains the new functionalities implemented with Qt GUI. As we can see, the new GUI provides a statusbar, a toolbar, and a control panel. The control panel can have trackbars and buttonbars attached to it.

- To attach a trackbar, the window_name parameter must be NULL.
- To attach a buttonbar, a button must be created. If the last bar attached to the control panel is a buttonbar, the new button is added on the right of the last button. If the last bar attached to the control panel is a trackbar, or the control panel is empty, a new buttonbar is created. Then a new button is attached to it.

The following code is an example used to generate the figure.

```
int main(int argc, char *argv[])
    int value = 50;
    int value2 = 0;
```

```
    cvNamedWindow("main1",CV_WINDOW_NORMAL);
    cvNamedWindow("main2",CV_WINDOW_AUTOSIZE | CV_GUI_NORMAL);

    cvCreateTrackbar( "track1", "main1", &value, 255,  NULL);//OK tested
    char* nameb1 = "button1";
    char* nameb2 = "button2";
    cvCreateButton(nameb1,callbackButton,nameb1,CV_CHECKBOX,1);

    cvCreateButton(nameb2,callbackButton,nameb2,CV_CHECKBOX,0);
    cvCreateTrackbar( "track2", NULL, &value2, 255, NULL);
    cvCreateButton("button5",callbackButton1,NULL,CV_RADIOBOX,0);
    cvCreateButton("button6",callbackButton2,NULL,CV_RADIOBOX,1);

    cvSetMouseCallback( "main2",on_mouse,NULL );

    IplImage* img1 = cvLoadImage("files/flower.jpg");
    IplImage* img2 = cvCreateImage(cvGetSize(img1),8,3);
    CvCapture* video = cvCaptureFromFile("files/hockey.avi");
    IplImage* img3 = cvCreateImage(cvGetSize(cvQueryFrame(video)),8,3);

    while(cvWaitKey(33) != 27)
    {
        cvAddS(img1,cvScalarAll(value),img2);
        cvAddS(cvQueryFrame(video),cvScalarAll(value2),img3);
        cvShowImage("main1",img2);
        cvShowImage("main2",img3);
    }

    cvDestroyAllWindows();
    cvReleaseImage(&img1);
    cvReleaseImage(&img2);
    cvReleaseImage(&img3);
    cvReleaseCapture(&video);
    return 0;
}
```

---

cv::setWindowProperty (view/add comments)

Change the parameters of the window dynamically.

```
void setWindowProperty(const string& name, int prop_id, double
prop_value);
```

name Name of the window.

prop_id Window's property to edit. The operation flags:

- **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (CV_WINDOW_NORMAL or CV_WINDOW_FULLSCREEN).
- **CV_WND_PROP_AUTOSIZE** Change if the user can resize the window (texttttCV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE).
- **CV_WND_PROP_ASPECTRATIO** Change if the image's aspect ratio is preserved (texttttCV_WINDOW_FREERA or CV_WINDOW_KEEPRATIO).

prop_value New value of the Window's property. The operation flags:

- **CV_WINDOW_NORMAL** Change the window in normal size, or allows the user to resize the window.
- **CV_WINDOW_AUTOSIZE** The user cannot resize the window, the size is constrainted by the image displayed.
- **CV_WINDOW_FULLSCREEN** Change the window to fullscreen.
- **CV_WINDOW_FREERATIO** The image expends as much as it can (no ratio constraint)
- **CV_WINDOW_KEEPRATIO** The ration image is respected.

The function setWindowProperty allows to change the window's properties.

---

cv::getWindowProperty (view/add comments)

Get the parameters of the window.

```
void getWindowProperty(const char* name, int prop_id);
```

name Name of the window.

prop_id Window's property to retrive. The operation flags:

- **CV_WND_PROP_FULLSCREEN** Change if the window is fullscreen (CV_WINDOW_NORMAL or CV_WINDOW_FULLSCREEN).

CV_WND_PROP_AUTOSIZE Change if the user can resize the window (texttttCV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE).

CV_WND_PROP_ASPECTRATIO Change if the image's aspect ratio is preserved (texttttCV_WINDOW_FREERA or CV_WINDOW_KEEPRATIO).

See setWindowProperty to know the meaning of the returned values.
The function getWindowProperty return window's properties.

---

cv::fontQt (view/add comments)

Create the font to be used to draw text on an image.

```
CvFont fontQt(const string& nameFont, int pointSize = -1, Scalar
color = Scalar::all(0), int weight = CV_FONT_NORMAL, int style =
CV_STYLE_NORMAL, int spacing = 0);
```

nameFont Name of the font. The name should match the name of a system font (such as "Times"). If the font is not found, a default one will be used.

pointSize Size of the font. If not specified, equal zero or negative, the point size of the font is set to a system-dependent default value. Generally, this is 12 points.

color Color of the font in BGRA – A = 255 is fully transparent. Use the macro CV_RGB for simplicity.

weight The operation flags:

CV_FONT_LIGHT Weight of 25

CV_FONT_NORMAL Weight of 50

CV_FONT_DEMIBOLD Weight of 63

CV_FONT_BOLD Weight of 75

CV_FONT_BLACK Weight of 87 You can also specify a positive integer for more control.

style The operation flags:

CV_STYLE_NORMAL Font is normal

CV_STYLE_ITALIC Font is in italic

CV_STYLE_OBLIQUE Font is oblique

spacing Spacing between characters. Can be negative or positive

The function `fontQt` creates a CvFont object. This CvFont is not compatible with putText. A basic usage of this function is:

```
CvFont font = fontQt(''Times'');
addText( img1, ``Hello World !'', Point(50,50), font);
```

---

cv::addText (view/add comments)

Create the font to be used to draw text on an image

```
void addText(const Mat& img, const string& text, Point location, CvFont
*font);
```

img Image where the text should be drawn

text Text to write on the image

location Point(x,y) where the text should start on the image

font Font to use to draw the text

The function `addText` draw *text* on the image *img* using a specific font *font* (see example fontQt)

---

cv::displayOverlay (view/add comments)

Display text on the window's image as an overlay for delay milliseconds. This is not editing the image's data. The text is display on the top of the image.

```
void displayOverlay(const string& name, const string& text, int delay);
```

name Name of the window

text Overlay text to write on the window's image

delay Delay to display the overlay text. If this function is called before the previous overlay text time out, the timer is restarted and the text updated. . If this value is zero, the text never disapers.

The function `displayOverlay` aims at displaying useful information/tips on the window for a certain amount of time *delay*. This information is display on the top of the window.

---

cv::displayStatusBar (view/add comments)

Display text on the window's statusbar as for delay milliseconds.

```
void displayStatusBar(const string& name, const string& text, int
delayms);
```

name Name of the window

text Text to write on the window's statusbar

delay Delay to display the text. If this function is called before the previous text time out, the timer is restarted and the text updated. If this value is zero, the text never disapers.

The function `displayOverlay` aims at displaying useful information/tips on the window for a certain amount of time *delay*. This information is displayed on the window's statubar (the window must be created with `CV_GUI_EXPANDED` flags).

---

cv::createOpenGLCallback (view/add comments)

Create a callback function called to draw OpenGL on top the the image display by *window_name*.

```
void createOpenGLCallback( const string& window_name, OpenGLCallback
callbackOpenGL, void* userdata CV_DEFAULT(NULL), double angle
CV_DEFAULT(-1), double zmin CV_DEFAULT(-1), double zmax CV_DEFAULT(-1);
```

window_name Name of the window

callbackOpenGL Pointer to the function to be called every frame. This function should be prototyped as `void Foo(*void);`.

userdata pointer passed to the callback function. *(Optional)*

angle Specifies the field of view angle, in degrees, in the y direction.. *(Optional - Default 45 degree)*

zmin Specifies the distance from the viewer to the near clipping plane (always positive). *(Optional - Default 0.01)*

zmax Specifies the distance from the viewer to the far clipping plane (always positive). *(Optional - Default 1000)*

The function `createOpenGLCallback` can be used to draw 3D data on the window. An example of callback could be:

```
void on_opengl(void* param)
{
    glLoadIdentity();

    glTranslated(0.0, 0.0, -1.0);

    glRotatef( 55, 1, 0, 0 );
    glRotatef( 45, 0, 1, 0 );
    glRotatef( 0, 0, 0, 1 );

    static const int coords[6][4][3] = {
        { { +1, -1, -1 }, { -1, -1, -1 }, { -1, +1, -1 }, { +1, +1, -1 } },
        { { +1, +1, -1 }, { -1, +1, -1 }, { -1, +1, +1 }, { +1, +1, +1 } },
        { { +1, -1, +1 }, { +1, -1, -1 }, { +1, +1, -1 }, { +1, +1, +1 } },
        { { -1, -1, -1 }, { -1, -1, +1 }, { -1, +1, +1 }, { -1, +1, -1 } },
        { { +1, -1, +1 }, { -1, -1, +1 }, { -1, -1, -1 }, { +1, -1, -1 } },
        { { -1, -1, +1 }, { +1, -1, +1 }, { +1, +1, +1 }, { -1, +1, +1 } }
    };

    for (int i = 0; i < 6; ++i) {
                glColor3ub( i*20, 100+i*10, i*42 );
                glBegin(GL_QUADS);
                for (int j = 0; j < 4; ++j) {
                        glVertex3d(0.2 * coords[i][j][0], 0.2 * coords[i][j][1], 0.2 * coo
                }
                glEnd();
    }
}
```

cv::saveWindowParameters (view/add comments)

Save parameters of the window *window_name*.

```
void saveWindowParameters(const string& name);
```

name Name of the window

The function `saveWindowParameters` saves size, location, flags, trackbars' value, zoom and panning location of the window *window_name*

cv::loadWindowParameters (view/add comments)

Load parameters of the window *window_name*.

```
void loadWindowParameters(const string& name);
```

name Name of the window

The function `loadWindowParameters` load size, location, flags, trackbars' value, zoom and panning location of the window *window_name*

cv::createButton (view/add comments)

Create a callback function called to draw OpenGL on top the the image display by *window_name*.

```
createButton( const string& button_name CV_DEFAULT(NULL),ButtonCallback
on_change CV_DEFAULT(NULL), void* userdata CV_DEFAULT(NULL) , int
button_type CV_DEFAULT(CV_PUSH_BUTTON), int initial_button_state
CV_DEFAULT(0);
```

button_name Name of the button *(if NULL, the name will be "button ¡number of boutton¿")*

on_change Pointer to the function to be called every time the button changed its state. This function should be prototyped as `void Foo(int state,*void);`. *state* is the current state of the button. It could be -1 for a push button, 0 or 1 for a check/radio box button.

userdata pointer passed to the callback function. *(Optional)*

The `button_type` parameter can be : *(Optional – Will be a push button by default.)*

CV_PUSH_BUTTON *The button will be a push button.*

CV_CHECKBOX *The button will be a checkbox button.*

CV_RADIOBOX *The button will be a radiobox button. The radiobox on the same buttonbar (same line) are exclusive; one on can be select at the time.*

initial_button_state Default state of the button. Use for checkbox and radiobox, its value could be 0 or 1. *(Optional)*

The function `createButton` attach a button to the control panel. Each button is added to a buttonbar on the right of the last button. A new buttonbar is create if nothing was attached to the control panel before, or if the last element attached to the control panel was a trackbar.

Here are various example of `createButton` function call:

```
createButton(NULL,callbackButton);//create a push button "button 0", that will call callbac
createButton("button2",callbackButton,NULL,CV\_CHECKBOX,0);
createButton("button3",callbackButton,&value);
createButton("button5",callbackButton1,NULL,CV\_RADIOBOX);
createButton("button6",callbackButton2,NULL,CV\_PUSH\_BUTTON,1);
```

Chapter 18

calib3d. Camera Calibration, Pose Estimation and Stereo

18.1 Camera Calibration and 3d Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re-used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distortion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$
where $r^2 = x'^2 + y'^2$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

k_1, k_2, k_3, k_4, k_5, k_6 are radial distortion coefficients, p_1, p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters. *And they remain the same regardless of the captured image resolution.* That is, if, for example, a camera has been calibrated on images of 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x, f_y, c_x and c_y need to be scaled appropriately).

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspondences).

- Estimate the relative position and orientation of the stereo camera "heads" and compute the *rectification* transformation that makes the camera optical axes parallel.

---

cv::calibrateCamera (view/add comments)

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
double calibrateCamera( const vector<vector<Point3f> >& objectPoints,
            const vector<vector<Point2f> >& imagePoints,
            Size imageSize,
            Mat& cameraMatrix, Mat& distCoeffs,
            vector<Mat>& rvecs, vector<Mat>& tvecs,
            int flags=0 );
```

objectPoints The vector of vectors of points on the calibration pattern in its coordinate system, one vector per view. If the same calibration pattern is shown in each view and it's fully visible then all the vectors will be the same, although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0

imagePoints The vector of vectors of the object point projections on the calibration pattern views, one vector per a view. The projections must be in the same order as the corresponding object points.

imageSize Size of the image, used only to initialize the intrinsic camera matrix

cameraMatrix The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized before calling the function

distCoeffs The output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements

rvecs The output vector of rotation vectors (see cv::Rodrigues), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector

(see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0..*M*-1)

tvecs The output vector of translation vectors, estimated for each pattern view.

flags Different flags, may be 0 or combination of the following values:

CV_CALIB_USE_INTRINSIC_GUESS `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`imageSize` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate the extrinsic parameters. Use cv::solvePnP instead.

CV_CALIB_FIX_PRINCIPAL_POINT The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.

CV_CALIB_FIX_ASPECT_RATIO The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix`.
When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.

CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6 Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.

CV_CALIB_RATIONAL_MODEL Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute and return only 5 distortion coefficients.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see cv::findChessboardCorners). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).

2. The initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using cv::solvePnP

3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points `imagePoints` and the projected (using the current estimates for camera parameters and the poses) object points `objectPoints`; see cv::projectPoints.

The function returns the final re-projection error.

Note: if you're using a non-square (=non-NxN) grid and cv::findChessboardCorners for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2 - 0.5, h/2 - 0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probably used `patternSize=cvSize(rows,cols)`, but should use `patternSize=cvSize(cols,rows)` in cv::findChessboardCorners.

See also: cv::findChessboardCorners, cv::solvePnP, cv::initCameraMatrix2D, cv::stereoCalibrate, cv::undistort

---

cv::calibrationMatrixValues (view/add comments)

Computes some useful camera characteristics from the camera matrix

```
void calibrationMatrixValues( const Mat& cameraMatrix,
           Size imageSize,
           double apertureWidth,
           double apertureHeight,
           double& fovx,
           double& fovy,
           double& focalLength,
           Point2d& principalPoint,
           double& aspectRatio );
```

cameraMatrix The input camera matrix that can be estimated by cv::calibrateCamera or cv::stereoCalibrate

imageSize The input image size in pixels

apertureWidth Physical width of the sensor

apertureHeight Physical height of the sensor

fovx The output field of view in degrees along the horizontal sensor axis

fovy The output field of view in degrees along the vertical sensor axis

focalLength The focal length of the lens in mm

principalPoint The principal point in pixels

aspectRatio f_y/f_x

The function computes various useful camera characteristics from the previously estimated camera matrix.

---

cv::composeRT (view/add comments)

Combines two rotation-and-shift transformations

```
void composeRT( const Mat& rvec1, const Mat& tvec1,
           const Mat& rvec2, const Mat& tvec2,
           Mat& rvec3, Mat& tvec3 );
void composeRT( const Mat& rvec1, const Mat& tvec1,
           const Mat& rvec2, const Mat& tvec2,
           Mat& rvec3, Mat& tvec3,
           Mat& dr3dr1, Mat& dr3dt1,
           Mat& dr3dr2, Mat& dr3dt2,
           Mat& dt3dr1, Mat& dt3dt1,
           Mat& dt3dr2, Mat& dt3dt2 );
```

rvec1 The first rotation vector

tvec1 The first translation vector

rvec2 The second rotation vector

tvec2 The second translation vector

rvec3 The output rotation vector of the superposition

tvec3 The output translation vector of the superposition

d??d?? The optional output derivatives of `rvec3` or `tvec3` w.r.t. `rvec?` or `tvec?`

The functions compute:

$$\begin{array}{l} \texttt{rvec3} = \mathrm{rodrigues}^{-1}\left(\mathrm{rodrigues}(\texttt{rvec2}) \cdot \mathrm{rodrigues}(\texttt{rvec1})\right) \\ \texttt{tvec3} = \mathrm{rodrigues}(\texttt{rvec2}) \cdot \texttt{tvec1} + \texttt{tvec2} \end{array} ,$$

where $\mathrm{rodrigues}$ denotes a rotation vector to rotation matrix transformation, and $\mathrm{rodrigues}^{-1}$ denotes the inverse transformation, see cv::Rodrigues.

Also, the functions can compute the derivatives of the output vectors w.r.t the input vectors (see cv::matMulDeriv). The functions are used inside cv::stereoCalibrate but can also be used in your own code where Levenberg-Marquardt or another gradient-based solver is used to optimize a function that contains matrix multiplication.

---

cv::computeCorrespondEpilines (view/add comments)

For points in one image of a stereo pair, computes the corresponding epilines in the other image.

```
void computeCorrespondEpilines( const Mat& points,
           int whichImage, const Mat& F,
           vector<Vec3f>& lines );
```

points The input points. $N \times 1$ or $1 \times N$ matrix of type `CV_32FC2` or `vector<Point2f>`

whichImage Index of the image (1 or 2) that contains the `points`

F The fundamental matrix that can be estimated using cv::findFundamentalMat or cv::stereoRectify.

lines The output vector of the corresponding to the points epipolar lines in the other image. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see cv::findFundamentalMat), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

---

cv::convertPointsHomogeneous (view/add comments)

Convert points to/from homogeneous coordinates.

```
void convertPointsHomogeneous( const Mat& src, vector<Point3f>& dst );
void convertPointsHomogeneous( const Mat& src, vector<Point2f>& dst );
```

src The input array or vector of 2D, 3D or 4D points

dst The output vector of 2D or 2D points

The functions convert 2D or 3D points from/to homogeneous coordinates, or simply copy or transpose the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$x' = x/w$
$y' = y/w$
$z' = z/w$ (if output is 3D)

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

---

cv::decomposeProjectionMatrix (view/add comments)

Decomposes the projection matrix into a rotation matrix and a camera matrix.

```
void decomposeProjectionMatrix( const Mat& projMatrix,
           Mat& cameraMatrix,
           Mat& rotMatrix, Mat& transVect );
void decomposeProjectionMatrix( const Mat& projMatrix,
           Mat& cameraMatrix,
           Mat& rotMatrix, Mat& transVect,
           Mat& rotMatrixX, Mat& rotMatrixY,
           Mat& rotMatrixZ, Vec3d& eulerAngles );
```

projMatrix The 3x4 input projection matrix P

cameraMatrix The output 3x3 camera matrix K

rotMatrix The output 3x3 external rotation matrix R

transVect The output 4x1 translation vector T

rotMatrX Optional 3x3 rotation matrix around x-axis

rotMatrY Optional 3x3 rotation matrix around y-axis

rotMatrZ Optional 3x3 rotation matrix around z-axis

eulerAngles Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on cv::RQDecomp3x3.

---

cv::drawChessboardCorners (view/add comments)

Renders the detected chessboard corners.

```
void drawChessboardCorners( Mat& image, Size patternSize,
           const Mat& corners,
           bool patternWasFound );
```

image The destination image; it must be an 8-bit color image

patternSize The number of inner corners per chessboard row and column. (patternSize = cv::Size(points_per_row,points_per_column) = cv::Size(rows,columns))

corners The array of corners detected, this should be the output from findChessboardCorners wrapped in a cv::Mat().

patternWasFound Indicates whether the complete board was found or not . One may just pass the return value cv::findChessboardCorners here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

---

cv::findChessboardCorners (view/add comments)

Finds the positions of the internal corners of the chessboard.

```
bool findChessboardCorners( const Mat& image, Size patternSize,
           vector<Point2f>& corners,
           int flags=CV_CALIB_CB_ADAPTIVE_THRESH+
           CV_CALIB_CB_NORMALIZE_IMAGE );
```

image Source chessboard view; it must be an 8-bit grayscale or color image

patternSize The number of inner corners per chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The output array of corners detected

flags Various operation flags, can be 0 or a combination of the following values:

- **CV_CALIB_CB_ADAPTIVE_THRESH** use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
- **CV_CALIB_CB_NORMALIZE_IMAGE** normalize the image gamma with cv::equalizeHist before applying fixed or adaptive thresholding.
- **CV_CALIB_CB_FILTER_QUADS** use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

CALIB_CB_FAST_CHECK Runs a fast check on the image that looks for chessboard corners, and shortcuts the call if none are found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cv::cornerSubPix.

Sample usage of detecting and drawing chessboard corners:

```
Size patternsize(8,6); //interior number of corners
Mat gray = ....; //source image
vector<Point2f> corners; //this will be filled by the detected corners

//CALIB_CB_FAST_CHECK saves a lot of time on images
//that don't contain any chessboard corners
bool patternfound = findChessboardCorners(gray, patternsize, corners,
        CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE
        + CALIB_CB_FAST_CHECK);

if(patternfound)
  cornerSubPix(gray, corners, Size(11, 11), Size(-1, -1),
    TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

drawChessboardCorners(img, patternsize, Mat(corners), patternfound);
```

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

---

cv::findCirclesGrid (view/add comments)

Finds the centers of the cirlces' grid.

```
bool findCirclesGrid( const Mat& image, Size patternSize,
           vector<Point2f>& centers,
           int flags=0 );
```

image Source circles' grid view; it must be an 8-bit grayscale or color image

patternSize The number of circles per grid row and column (patternSize = Size(points_per_row, points_per_colum) = Size(columns, rows))

centers The output array of centers detected

flags Various operation flags, no flags are supported currently

The function attempts to determine whether the input image is a view of the circles' grid pattern and locate the circles' centers. The function returns a non-zero value if all of the centers have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0.

Sample usage of detecting and drawing circles' centers:

```
Size patternsize(7,7); //number of centers
Mat gray = ....; //source image
vector<Point2f> centers; //this will be filled by the detected centers

bool patternfound = findCirclesGrid(gray, patternsize, centers);

drawChessboardCorners(img, patternsize, Mat(centers), patternfound);
```

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment.

---

cv::solvePnP (view/add comments)

Finds the object pose from the 3D-2D point correspondences

```
void solvePnP( const Mat& objectPoints,
           const Mat& imagePoints,
           const Mat& cameraMatrix,
           const Mat& distCoeffs,
           Mat& rvec, Mat& tvec,
           bool useExtrinsicGuess=false );
```

objectPoints The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points. Can also pass `vector<Point3f>` here.

imagePoints The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points. Can also pass `vector<Point2f>` here.

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

rvec The output rotation vector (see cv::Rodrigues) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system

tvec The output translation vector

useExtrinsicGuess If true (1), the function will use the provided `rvec` and `tvec` as the initial approximations of the rotation and translation vectors, respectively, and will further optimize them.

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using cv::projectPoints) `objectPoints`.

---

cv::findFundamentalMat (view/add comments)

Calculates the fundamental matrix from the corresponding points in two images.

```
Mat findFundamentalMat( const Mat& points1, const Mat& points2,
           vector<uchar>& status, int method=FM_RANSAC,
           double param1=3., double param2=0.99 );
Mat findFundamentalMat( const Mat& points1, const Mat& points2,
           int method=FM_RANSAC,
           double param1=3., double param2=0.99 );
```

points1 Array of `N` points from the first image. . The point coordinates should be floating-point (single or double precision)

points2 Array of the second image points of the same size and format as `points1`

method Method for computing the fundamental matrix

- **CV_FM_7POINT** for a 7-point algorithm. $N = 7$
- **CV_FM_8POINT** for an 8-point algorithm. $N \geq 8$
- **CV_FM_RANSAC** for the RANSAC algorithm. $N \geq 8$
- **CV_FM_LMEDS** for the LMedS algorithm. $N \geq 8$

param1 The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise

param2 The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct

status The output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1's

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the found fundamental matrix . Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to cv::computeCorrespondEpilines that finds the epipolar lines corresponding to the specified points. It can also be passed to cv::stereoRectifyUncalibrated to compute the rectification transformation.

```
// Example. Estimation of fundamental matrix using RANSAC algorithm
int point_count = 100;
vector<Point2f> points1(point_count);
vector<Point2f> points2(point_count);

// initialize the points here ... */
for( int i = 0; i < point_count; i++ )
{
    points1[i] = ...;
    points2[i] = ...;
```

```
}

Mat fundamental_matrix =
 findFundamentalMat(points1, points2, FM_RANSAC, 3, 0.99);
```

cv::findHomography (view/add comments)

Finds the perspective transformation between two planes.

```
Mat findHomography( const Mat& srcPoints, const Mat& dstPoints,
           Mat& status, int method=0,
           double ransacReprojThreshold=3 );
Mat findHomography( const Mat& srcPoints, const Mat& dstPoints,
           vector<uchar>& status, int method=0,
           double ransacReprojThreshold=3 );
Mat findHomography( const Mat& srcPoints, const Mat& dstPoints,
           int method=0, double ransacReprojThreshold=3 );
```

srcPoints Coordinates of the points in the original plane, a matrix of type `CV_32FC2` or a `vector<Point2f>`.

dstPoints Coordinates of the points in the target plane, a matrix of type `CV_32FC2` or a `vector<Point2f>`.

method The method used to computed homography matrix; one of the following:

- **0** a regular method using all the points
- **CV_RANSAC** RANSAC-based robust method
- **CV_LMEDS** Least-Median robust method

ransacReprojThreshold The maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.

status The optional output mask set by a robust method (CV_RANSAC or CV_LMEDS). *Note that the input mask values are ignored.*

The functions find and return the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter method is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, RANSAC and LMeDS, try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection error for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method RANSAC can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method LMeDS does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features, where can be only some small noise present, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized so that $h_{33} = 1$.

See also: cv::getAffineTransform, cv::getPerspectiveTransform, cv::estimateRigidMotion, cv::warpPerspective, cv::perspectiveTransform

---

cv::getDefaultNewCameraMatrix (view/add comments)

Returns the default new camera matrix

```
Mat getDefaultNewCameraMatrix(
            const Mat& cameraMatrix,
            Size imgSize=Size(),
            bool centerPrincipalPoint=false );
```

cameraMatrix The input camera matrix

imageSize The camera view image size in pixels

centerPrincipalPoint Indicates whether in the new camera matrix the principal point should be at the image center or not

The function returns the camera matrix that is either an exact copy of the input `cameraMatrix` (when `centerPrinicipalPoint=false`), or the modified one (when `centerPrincipalPoint`=true).

In the latter case the new camera matrix will be:

$$\begin{bmatrix} f_x & 0 & (\texttt{imgSize.width} - 1) * 0.5 \\ 0 & f_y & (\texttt{imgSize.height} - 1) * 0.5 \\ 0 & 0 & 1 \end{bmatrix},$$

where f_x and f_y are $(0,0)$ and $(1,1)$ elements of `cameraMatrix`, respectively.

By default, the undistortion functions in OpenCV (see `initUndistortRectifyMap`, `undistort`) do not move the principal point. However, when you work with stereo, it's important to move the principal points in both views to the same y-coordinate (which is required by most of stereo correspondence algorithms), and maybe to the same x-coordinate too. So you can form the new camera matrix for each view, where the principal points will be at the center.

---

cv::getOptimalNewCameraMatrix (view/add comments)

Returns the new camera matrix based on the free scaling parameter

```
Mat getOptimalNewCameraMatrix(
            const Mat& cameraMatrix, const Mat& distCoeffs,
            Size imageSize, double alpha, Size newImageSize=Size(),
            Rect* validPixROI=0);
```

cameraMatrix The input camera matrix

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imageSize The original image size

alpha The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see cv::stereoRectify

newCameraMatrix The output new camera matrix.

newImageSize The image size after rectification. By default it will be set to `imageSize`.

validPixROI The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in cv::stereoRectify

The function computes and returns the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0`, keep all the original image pixels if there is valuable information in the corners `alpha=1`, or get something in between. When `alpha>0`, the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to cv::initUndistortRectifyMap to produce the maps for cv::remap.

---

cv::initCameraMatrix2D (view/add comments)

Finds the initial camera matrix from the 3D-2D point correspondences

```
Mat initCameraMatrix2D( const vector<vector<Point3f> >& objectPoints,
            const vector<vector<Point2f> >& imagePoints,
            Size imageSize, double aspectRatio=1.);
```

objectPoints The vector of vectors of the object points. See cv::calibrateCamera

imagePoints The vector of vectors of the corresponding image points. See cv::calibrateCamera

imageSize The image size in pixels; used to initialize the principal point

aspectRatio If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

---

cv::initUndistortRectifyMap (view/add comments)

Computes the undistortion and rectification transformation map.

```
void initUndistortRectifyMap( const Mat& cameraMatrix,
            const Mat& distCoeffs, const Mat& R,
            const Mat& newCameraMatrix,
            Size size, int m1type,
            Mat& map1, Mat& map2 );
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R The optional rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv::stereoRectify can be passed here. If the matrix is empty , the identity transformation is assumed

newCameraMatrix The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$

size The undistorted image size

m1type The type of the first output map, can be `CV_32FC1` or `CV_16SC2`. See cv::convertMaps

map1 The first output map

map2 The second output map

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for cv::remap. The undistorted image will look like the original, as if it was captured with a camera with camera matrix =newCameraMatrix and zero distortion. In the case of monocular camera newCameraMatrix is usually equal to cameraMatrix, or it can be computed by cv::getOptimalNewCameraMatrix for a better control over scaling. In the case of stereo camera newCameraMatrix is normally set to P1 or P2 computed by cv::stereoRectify.

Also, this new camera will be oriented differently in the coordinate space, according to R. That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images become horizontal and have the same y- coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by cv::remap. That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$
\begin{array}{l}
x \leftarrow (u - {c'}_x)/{f'}_x \\
y \leftarrow (v - {c'}_y)/{f'}_y \\
[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\
x' \leftarrow X/W \\
y' \leftarrow Y/W \\
x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
map_x(u, v) \leftarrow x" f_x + c_x \\
map_y(u, v) \leftarrow y" f_y + c_y
\end{array}
$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after cv::stereoRectify, which in its turn is called after cv::stereoCalibrate. But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using cv::stereoRectifyUncalibrated. For each camera the function computes homography H as the rectification transformation in pixel domain, not a rotation matrix R in 3D space. The R can be computed from H as

$$\texttt{R} = \texttt{cameraMatrix}^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the cameraMatrix can be chosen arbitrarily.

---

cv::matMulDeriv (view/add comments)

Computes partial derivatives of the matrix product w.r.t each multiplied matrix

```
void matMulDeriv( const Mat& A, const Mat& B, Mat& dABdA, Mat& dABdB );
```

A The first multiplied matrix

B The second multiplied matrix

dABdA The first output derivative matrix `d(A*B)/dA` of size `A.rows*B.cols` $\times A.rows * A.cols$

dABdA The second output derivative matrix `d(A*B)/dB` of size `A.rows*B.cols` $\times B.rows * B.cols$

The function computes the partial derivatives of the elements of the matrix product $A * B$ w.r.t. the elements of each of the two input matrices. The function is used to compute Jacobian matrices in cv::stereoCalibrate, but can also be used in any other similar optimization function.

---

cv::projectPoints (view/add comments)

Project 3D points on to an image plane.

```
void projectPoints( const Mat& objectPoints,
           const Mat& rvec, const Mat& tvec,
           const Mat& cameraMatrix,
           const Mat& distCoeffs,
           vector<Point2f>& imagePoints );
void projectPoints( const Mat& objectPoints,
           const Mat& rvec, const Mat& tvec,
           const Mat& cameraMatrix,
           const Mat& distCoeffs,
           vector<Point2f>& imagePoints,
           Mat& dpdrot, Mat& dpdt, Mat& dpdf,
           Mat& dpdc, Mat& dpddist,
           double aspectRatio=0 );
```

objectPoints The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel (or `vector<Point3f>`) , where N is the number of points in the view

rvec The rotation vector, see cv::Rodrigues

tvec The translation vector

cameraMatrix The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imagePoints The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel (or `vector<Point2f>`)

dpdrot Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector

dpdt Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector

dpdf Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y

dpdc Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y

dpddist Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in cv::calibrateCamera, cv::solvePnP and cv::stereoCalibrate. The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

---

cv::reprojectImageTo3D (view/add comments)

Reprojects disparity image to 3D space.

```
void reprojectImageTo3D( const Mat& disparity,
           Mat& _3dImage, const Mat& Q,
           bool handleMissingValues=false );
```

disparity The input single-channel 16-bit signed or 32-bit floating-point disparity image

_3dImage The output 3-channel floating-point image of the same size as `disparity`. Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)`, computed from the disparity map.

Q The 4×4 perspective transformation matrix that can be obtained with cv::stereoRectify

handleMissingValues If true, when the pixels with the minimal disparity (that corresponds to the outliers; see cv::StereoBM) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \mathtt{Q} * [x\ y\ \mathtt{disparity}(x,y)\ 1]^T$$
$$\mathtt{_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by cv::stereoRectify. To reproject a sparse set of points (x,y,d),... to 3D space, use cv::perspectiveTransform.

---

cv::RQDecomp3x3 (view/add comments)

Computes the 'RQ' decomposition of 3x3 matrices.

```
void RQDecomp3x3( const Mat& M, Mat& R, Mat& Q );
Vec3d RQDecomp3x3( const Mat& M, Mat& R, Mat& Q,
           Mat& Qx, Mat& Qy, Mat& Qz );
```

M The 3x3 input matrix

R The output 3x3 upper-triangular matrix

Q The output 3x3 orthogonal matrix

Qx Optional 3x3 rotation matrix around x-axis

Qy Optional 3x3 rotation matrix around y-axis

Qz Optional 3x3 rotation matrix around z-axis

The function computes a RQ decomposition using the given rotations. This function is used in cv::decomposeProjectionMatrix to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles (as the return value) that could be used in OpenGL.

---

cv::Rodrigues (view/add comments)

Converts a rotation matrix to a rotation vector or vice versa.

```
void Rodrigues(const Mat& src, Mat& dst);
void Rodrigues(const Mat& src, Mat& dst, Mat& jacobian);
```

src The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)

dst The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively

jacobian Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like cv::calibrateCamera, cv::stereoCalibrate or cv::solvePnP.

---

cv::StereoBM (view/add comments)

The class for computing stereo correspondence using block matching algorithm.

```
// Block matching stereo correspondence algorithm\par
class StereoBM
{
    enum { NORMALIZED_RESPONSE = CV_STEREO_BM_NORMALIZED_RESPONSE,
        BASIC_PRESET=CV_STEREO_BM_BASIC,
        FISH_EYE_PRESET=CV_STEREO_BM_FISH_EYE,
        NARROW_PRESET=CV_STEREO_BM_NARROW };

    StereoBM();
    // the preset is one of ..._PRESET above.
    // ndisparities is the size of disparity range,
    // in which the optimal disparity at each pixel is searched for.
    // SADWindowSize is the size of averaging window used to match pixel blocks
    //    (larger values mean better robustness to noise, but yield blurry disparity maps)
    StereoBM(int preset, int ndisparities=0, int SADWindowSize=21);
    // separate initialization function
    void init(int preset, int ndisparities=0, int SADWindowSize=21);
    // computes the disparity for the two rectified 8-bit single-channel images.
    // the disparity will be 16-bit signed (fixed-point) or 32-bit floating-point image of
    void operator()( const Mat& left, const Mat& right, Mat& disparity, int disptype=CV_16S

    Ptr<CvStereoBMState> state;
};
```

The class is a C++ wrapper for cvStereoBMState and the associated functions. In particular, `StereoBM::operator ()` is the wrapper for cv::. See the respective descriptions.

---

cv::StereoSGBM (view/add comments)

The class for computing stereo correspondence using semi-global block matching algorithm.

```
class StereoSGBM
{
    StereoSGBM();
    StereoSGBM(int minDisparity, int numDisparities, int SADWindowSize,
               int P1=0, int P2=0, int disp12MaxDiff=0,
               int preFilterCap=0, int uniquenessRatio=0,
```

```
                int speckleWindowSize=0, int speckleRange=0,
                bool fullDP=false);
    virtual ~StereoSGBM();

    virtual void operator()(const Mat& left, const Mat& right, Mat& disp);

    int minDisparity;
    int numberOfDisparities;
    int SADWindowSize;
    int preFilterCap;
    int uniquenessRatio;
    int P1, P2;
    int speckleWindowSize;
    int speckleRange;
    int disp12MaxDiff;
    bool fullDP;

    ...
};
```

The class implements modified H. Hirschmuller algorithm [**?**]. The main differences between the implemented algorithm and the original one are:

- by default the algorithm is single-pass, i.e. instead of 8 directions we only consider 5. Set `fullDP=true` to run the full variant of the algorithm (which could consume *a lot* of memory)
- the algorithm matches blocks, not individual pixels (though, by setting `SADWindowSize=1` the blocks are reduced to single pixels)
- mutual information cost function is not implemented. Instead, we use a simpler Birchfield-Tomasi sub-pixel metric from [**?**], though the color images are supported as well.
- we include some pre- and post- processing steps from K. Konolige algorithm cv::, such as pre-filtering (`CV_STEREO_BM_XSOBEL` type) and post-filtering (uniqueness check, quadratic interpolation and speckle filtering)

---

cv::StereoSGBM::StereoSGBM (view/add comments)

StereoSGBM constructors

```
StereoSGBM::StereoSGBM();
StereoSGBM::StereoSGBM(
```

```
int minDisparity, int numDisparities, int SADWindowSize,
int P1=0, int P2=0, int disp12MaxDiff=0,
int preFilterCap=0, int uniquenessRatio=0,
int speckleWindowSize=0, int speckleRange=0,
bool fullDP=false);
```

minDisparity The minimum possible disparity value. Normally it is 0, but sometimes rectification algorithms can shift images, so this parameter needs to be adjusted accordingly

numDisparities This is maximum disparity minus minimum disparity. Always greater than 0. In the current implementation this parameter must be divisible by 16.

SADWindowSize The matched block size. Must be an odd number `>=1`. Normally, it should be somewhere in `3..11` range .

P1, P2 Parameters that control disparity smoothness. The larger the values, the smoother the disparity. `P1` is the penalty on the disparity change by plus or minus 1 between neighbor pixels. `P2` is the penalty on the disparity change by more than 1 between neighbor pixels. The algorithm requires `P2 > P1`. See `stereo_match.cpp` sample where some reasonably good `P1` and `P2` values are shown (like `8*number_of_image_channels*SADWindowSize*SADWindowS` and `32*number_of_image_channels*SADWindowSize*SADWindowSize`, respectively).

disp12MaxDiff Maximum allowed difference (in integer pixel units) in the left-right disparity check. Set it to non-positive value to disable the check.

preFilterCap Truncation value for the prefiltered image pixels. The algorithm first computes x-derivative at each pixel and clips its value by `[-preFilterCap, preFilterCap]` interval. The result values are passed to the Birchfield-Tomasi pixel cost function.

uniquenessRatio The margin in percents by which the best (minimum) computed cost function value should ”win” the second best value to consider the found match correct. Normally, some value within 5-15 range is good enough

speckleWindowSize Maximum size of smooth disparity regions to consider them noise speckles and invdalidate. Set it to 0 to disable speckle filtering. Otherwise, set it somewhere in 50-200 range.

speckleRange Maximum disparity variation within each connected component. If you do speckle filtering, set it to some positive value, multiple of 16. Normally, 16 or 32 is good enough.

fullDP Set it to `true` to run full-scale 2-pass dynamic programming algorithm. It will consume O(W*H*numDisparities) bytes, which is large for 640x480 stereo and huge for HD-size pictures. By default this is `false`

The first constructor initializes `StereoSGBM` with all the default parameters (so actually one will only have to set `StereoSGBM::numberOfDisparities` at minimum). The second constructor allows you to set each parameter to a custom value.

---

cv::StereoSGBM::operator () (view/add comments)

Computes disparity using SGBM algorithm for a rectified stereo pair

```
void SGBM::operator()(const Mat& left, const Mat& right, Mat& disp);
```

left The left image, 8-bit single-channel or 3-channel.

right The right image of the same size and the same type as the left one.

disp The output disparity map. It will be 16-bit signed single-channel image of the same size as the input images. It will contain scaled by 16 disparity values, so that to get the floating-point disparity map, you will need to divide each `disp` element by 16.

The method executes SGBM algorithm on a rectified stereo pair. See `stereo_match.cpp` OpenCV sample on how to prepare the images and call the method. Note that the method is not constant, thus you should not use the same `StereoSGBM` instance from within different threads simultaneously.

---

cv::stereoCalibrate (view/add comments)

Calibrates stereo camera.

```
double stereoCalibrate( const vector<vector<Point3f> >& objectPoints,
           const vector<vector<Point2f> >& imagePoints1,
           const vector<vector<Point2f> >& imagePoints2,
           Mat& cameraMatrix1, Mat& distCoeffs1,
           Mat& cameraMatrix2, Mat& distCoeffs2,
```

```
            Size imageSize, Mat& R, Mat& T,
            Mat& E, Mat& F,
            TermCriteria term_crit = TermCriteria(TermCriteria::COUNT+
            TermCriteria::EPS, 30, 1e-6),
            int flags=CALIB_FIX_INTRINSIC );
```

objectPoints The vector of vectors of points on the calibration pattern in its coordinate system, one vector per view. If the same calibration pattern is shown in each view and it's fully visible then all the vectors will be the same, although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0

imagePoints1 The vector of vectors of the object point projections on the calibration pattern views from the 1st camera, one vector per a view. The projections must be in the same order as the corresponding object points.

imagePoints2 The vector of vectors of the object point projections on the calibration pattern views from the 2nd camera, one vector per a view. The projections must be in the same order as the corresponding object points.

cameraMatrix1 The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of CV_CALIB_USE_INTRINSIC_GUESS, CV_CALIB_FIX_ASPECT_RATIO, CV_CALIB_FIX_INTRINSIC or CV_CALIB_FIX_FOCAL_LENGTH are specified, some or all of the matrices' components must be initialized; see the flags description

distCoeffs The input/output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. On output vector length depends on the flags.

cameraMatrix2 The input/output second camera matrix, as cameraMatrix1.

distCoeffs2 The input/output lens distortion coefficients for the second camera, as distCoeffs1.

imageSize Size of the image, used only to initialize intrinsic camera matrix.

R The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The output translation vector between the cameras' coordinate systems.

E The output essential matrix.

F The output fundamental matrix.

term_crit The termination criteria for the iterative optimization algorithm.

flags Different flags, may be 0 or combination of the following values:

- **CV_CALIB_FIX_INTRINSIC** If it is set, `cameraMatrix?`, as well as `distCoeffs?` are fixed, so that only `R, T, E` and `F` are estimated.
- **CV_CALIB_USE_INTRINSIC_GUESS** The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.
- **CV_CALIB_FIX_PRINCIPAL_POINT** The principal points are fixed during the optimization.
- **CV_CALIB_FIX_FOCAL_LENGTH** $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.
- **CV_CALIB_FIX_ASPECT_RATIO** $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.
- **CV_CALIB_SAME_FOCAL_LENGTH** Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$
- **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients for each camera are set to zeros and fixed there.
- **CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6** Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.
- **CV_CALIB_RATIONAL_MODEL** Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute and return only 5 distortion coefficients.

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with cv::solvePnP), obviously, those poses will relate to each other, i.e. given (R_1, T_1) it should be possible to compute (R_2, T_2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector T: $T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform full calibration of each of the 2 cameras. However, because of the high dimensionality of the parameter space and noise in the input data the function can diverge from the correct solution. Thus, if intrinsic parameters can be estimated with high accuracy for each of the cameras individually (e.g. using cv::calibrateCamera), it is recommended to do so and then pass `CV_CALIB_FIX_INTRINSIC` flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are estimated at once, it makes sense to restrict some parameters, e.g. pass `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which are usually reasonable assumptions.

Similarly to cv::calibrateCamera, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

---

cv::stereoRectify (view/add comments)

Computes rectification transforms for each head of a calibrated stereo camera.

```
void stereoRectify( const Mat& cameraMatrix1, const Mat& distCoeffs1,
            const Mat& cameraMatrix2, const Mat& distCoeffs2,
            Size imageSize, const Mat& R, const Mat& T,
            Mat& R1, Mat& R2, Mat& P1, Mat& P2, Mat& Q,
            int flags=CALIB_ZERO_DISPARITY );
void stereoRectify( const Mat& cameraMatrix1, const Mat& distCoeffs1,
            const Mat& cameraMatrix2, const Mat& distCoeffs2,
            Size imageSize, const Mat& R, const Mat& T,
            Mat& R1, Mat& R2, Mat& P1, Mat& P2, Mat& Q,
            double alpha, Size newImageSize=Size(),
            Rect* roi1=0, Rect* roi2=0,
            int flags=CALIB_ZERO_DISPARITY );
```

cameraMatrix1, cameraMatrix2 The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs1, distCoeffs2

distCoeffs The input vectors of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements each. If the vectors are NULL/empty, the zero distortion coefficients are assumed.

imageSize Size of the image used for stereo calibration.

R The rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The translation vector between the cameras' coordinate systems.

R1, R2 The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.

P1, P2 The output 3×4 projection matrices in the new (rectified) coordinate systems.

Q The output 4×4 disparity-to-depth mapping matrix, see cv::reprojectImageTo3D.

flags The operation flags; may be 0 or `CV_CALIB_ZERO_DISPARITY`. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.

alpha The free scaling parameter. If it is -1 or absent , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.

newImageSize The new image resolution after rectification. The same size should be passed to cv::initUndistortRectifyMap, see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.

roi1, roi2 The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0`, the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by cv::stereoCalibrate and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

1. Horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. Vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

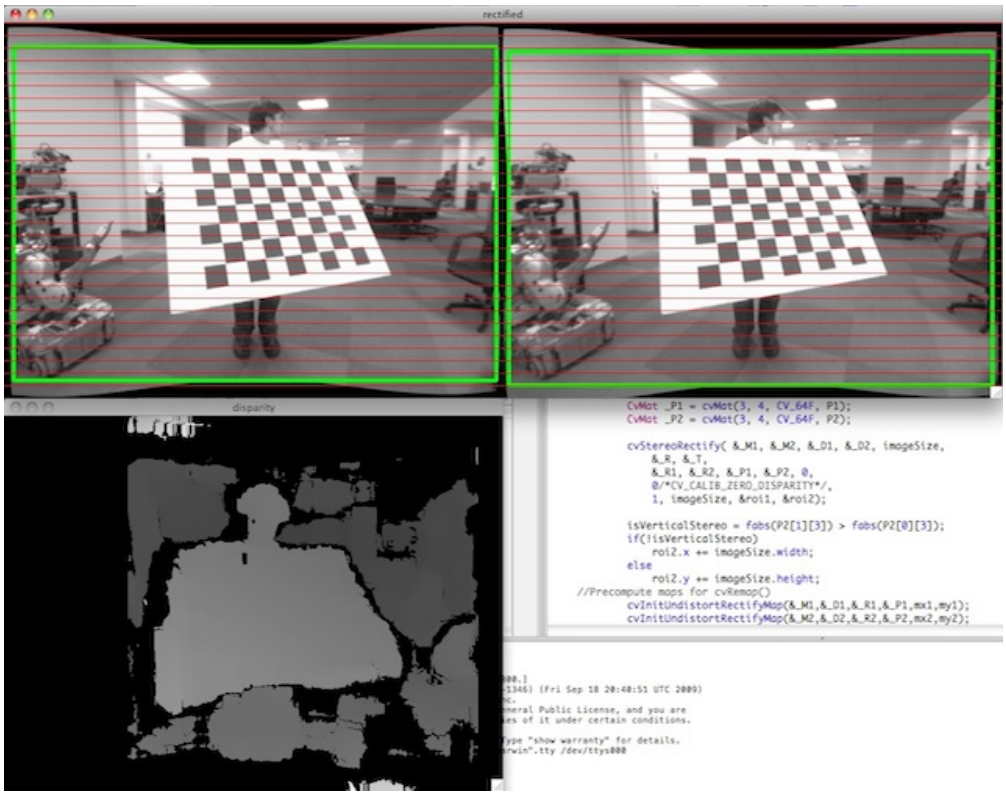
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of `P1` and `P2` will effectively be the new "rectified" camera matrices. The matrices, together with `R1` and `R2`, can then be passed to cv::initUndistortRectifyMap to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



---

cv::stereoRectifyUncalibrated (view/add comments)

Computes rectification transform for uncalibrated stereo camera.

```
bool stereoRectifyUncalibrated( const Mat& points1,
            const Mat& points2,
            const Mat& F, Size imgSize,
            Mat& H1, Mat& H2,
            double threshold=5 );
```

points1, points2 The 2 arrays of corresponding 2D points. The same formats as in cv::findFundamentalMat are supported

F The input fundamental matrix. It can be computed from the same set of point pairs using cv::findFundamentalMat.

imageSize Size of the image.

H1, H2 The output rectification homography matrices for the first and for the second images.

threshold The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from cv::stereoRectify is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the algorithm [**?**].

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using cv::calibrateCamera and then the images can be corrected using cv::undistort, or just the point coordinates can be corrected with cv::undistortPoints.

---

cv::undistort (view/add comments)

Transforms an image to compensate for lens distortion.

```
void undistort( const Mat& src, Mat& dst, const Mat& cameraMatrix,
           const Mat& distCoeffs, const Mat& newCameraMatrix=Mat() );
```

src The input (distorted) image

dst The output (corrected) image; will have the same size and the same type as `src`

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

newCameraMatrix Camera matrix of the distorted image. By default it is the same as `cameraMatrix`, but you may additionally scale and shift the result by using some different matrix

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of cv::initUndistortRectifyMap (with unity `R`) and cv::remap (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix`. You can use cv::getOptimalNewCameraMatrix to compute the appropriate `newCameraMatrix`, depending on your requirements.

The camera matrix and the distortion parameters can be determined using cv::calibrateCamera. If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

---

cv::undistortPoints (view/add comments)

Computes the ideal point coordinates from the observed point coordinates.

```
void undistortPoints( const Mat& src, vector<Point2f>& dst,
           const Mat& cameraMatrix, const Mat& distCoeffs,
           const Mat& R=Mat(), const Mat& P=Mat());
void undistortPoints( const Mat& src, Mat& dst,
           const Mat& cameraMatrix, const Mat& distCoeffs,
           const Mat& R=Mat(), const Mat& P=Mat());
```

src The observed point coordinates, 1xN or Nx1 2-channel (CV_32FC2 or CV_64FC2).

dst The output ideal point coordinates, after undistortion and reverse perspective transformation .

cameraMatrix The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv::StereoRectify can be passed here. If the matrix is empty, the identity transformation is used

P The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2`, computed by cv::StereoRectify can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to cv::undistort and cv::initUndistortRectifyMap, but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to cv::projectPoints (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used both for a stereo camera head or for monocular camera (when R is empty).

Chapter 19

ml. Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different seta of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class 'CvStatModel' that all the other ML classes are derived from.

19.1 Statistical Models

---

cv::CvStatModel (view/add comments)

Base class for the statistical models in ML.

```
class CvStatModel
{
public:
    /* CvStatModel(); */
    /* CvStatModel( const CvMat* train_data ... ); */

    virtual ~CvStatModel();

    virtual void clear()=0;

    /* virtual bool train( const CvMat* train_data, [int tflag,] ..., const
        CvMat* responses, ...,
     [const CvMat* var_idx,] ..., [const CvMat* sample_idx,] ...
     [const CvMat* var_type,] ..., [const CvMat* missing_mask,]
```

```
        <misc_training_alg_params> ... )=0;
      */

    /* virtual float predict( const CvMat* sample ... ) const=0; */

    virtual void save( const char* filename, const char* name=0 )=0;
    virtual void load( const char* filename, const char* name=0 )=0;

    virtual void write( CvFileStorage* storage, const char* name )=0;
    virtual void read( CvFileStorage* storage, CvFileNode* node )=0;
};
```

In this declaration some methods are commented off. Actually, these are methods for which there is no unified API (with the exception of the default constructor), however, there are many similarities in the syntax and semantics that are briefly described below in this section, as if they are a part of the base class.

CvStatModel::CvStatModel (view/add comments)

Default constructor.

```
CvStatModel::CvStatModel();
```

Each statistical model class in ML has a default constructor without parameters. This constructor is useful for 2-stage model construction, when the default constructor is followed by `train()` or `load()`.

CvStatModel::CvStatModel(...) (view/add comments)

Training constructor.

```
CvStatModel::CvStatModel( const CvMat* train_data ... );
```

Most ML classes provide single-step construct and train constructors. This constructor is equivalent to the default constructor, followed by the `train()` method with the parameters that are passed to the constructor.

CvStatModel:: CvStatModel (view/add comments)

Virtual destructor.

```
CvStatModel:: CvStatModel();
```

The destructor of the base class is declared as virtual, so it is safe to write the following code:

```
CvStatModel* model;
if( use\_svm )
    model = new CvSVM(... /* SVM params */);
else
    model = new CvDTree(... /* Decision tree params */);
...
delete model;
```

Normally, the destructor of each derived class does nothing, but in this instance it calls the overridden method `clear()` that deallocates all the memory.

CvStatModel::clear (view/add comments)

Deallocates memory and resets the model state.

```
void CvStatModel::clear();
```

The method `clear` does the same job as the destructor; it deallocates all the memory occupied by the class members. But the object itself is not destructed, and can be reused further. This method is called from the destructor, from the `train` methods of the derived classes, from the methods `load()`, `read()` or even explicitly by the user.

CvStatModel::save (view/add comments)

Saves the model to a file.

```
void CvStatModel::save( const char* filename, const char* name=0 );
```

The method `save` stores the complete model state to the specified XML or YAML file with the specified name or default name (that depends on the particular class). `Data persistence` functionality from CxCore is used.

CvStatModel::load (view/add comments)

Loads the model from a file.

```
void CvStatModel::load( const char* filename, const char* name=0 );
```

The method `load` loads the complete model state with the specified name (or default model-dependent name) from the specified XML or YAML file. The previous model state is cleared by `clear()`.

Note that the method is virtual, so any model can be loaded using this virtual method. However, unlike the C types of OpenCV that can be loaded using the generic crosscvLoad, here the model type must be known, because an empty model must be constructed beforehand. This limitation will be removed in the later ML versions.

CvStatModel::write (view/add comments)

Writes the model to file storage.

```
void CvStatModel::write( CvFileStorage* storage, const char* name );
```

The method `write` stores the complete model state to the file storage with the specified name or default name (that depends on the particular class). The method is called by `save()`.

CvStatModel::read (view/add comments)

Reads the model from file storage.

```
void CvStatMode::read( CvFileStorage* storage, CvFileNode* node );
```

The method `read` restores the complete model state from the specified node of the file storage. The node must be located by the user using the function GetFileNodeByName .

The previous model state is cleared by `clear()`.

CvStatModel::train (view/add comments)

Trains the model.

```
bool CvStatMode::train( const CvMat* train_data, [int tflag,] ...,
const CvMat* responses, ...,
           [const CvMat* var_idx,] ..., [const CvMat* sample_idx,] ...
           [const CvMat* var_type,] ..., [const CvMat* missing_mask,]
<misc_training_alg_params> ... );
```

The method trains the statistical model using a set of input feature vectors and the corresponding output values (responses). Both input and output vectors/values are passed as matrices. By default the input feature vectors are stored as `train_data` rows, i.e. all the components (features) of a training vector are stored continuously. However, some algorithms can handle the transposed representation, when all values of each particular feature (component/input variable) over the whole input set are stored continuously. If both layouts are supported, the method includes `tflag` parameter that specifies the orientation:

- `tflag=CV_ROW_SAMPLE` means that the feature vectors are stored as rows,
- `tflag=CV_COL_SAMPLE` means that the feature vectors are stored as columns.

The `train_data` must have a `CV_32FC1` (32-bit floating-point, single-channel) format. Responses are usually stored in the 1d vector (a row or a column) of `CV_32SC1` (only in the classification problem) or `CV_32FC1` format, one value per input vector (although some algorithms, like various flavors of neural nets, take vector responses).

For classification problems the responses are discrete class labels; for regression problems the responses are values of the function to be approximated. Some algorithms can deal only with classification problems, some - only with regression problems, and some can deal with both problems. In the latter case the type of output variable is either passed as separate parameter, or as a last element of `var_type` vector:

- `CV_VAR_CATEGORICAL` means that the output values are discrete class labels,
- `CV_VAR_ORDERED(=CV_VAR_NUMERICAL)` means that the output values are ordered, i.e. 2 different values can be compared as numbers, and this is a regression problem

The types of input variables can be also specified using var_type. Most algorithms can handle only ordered input variables.

Many models in the ML may be trained on a selected feature subset, and/or on a selected sample subset of the training set. To make it easier for the user, the method train usually includes var_idx and sample_idx parameters. The former identifies variables (features) of interest, and the latter identifies samples of interest. Both vectors are either integer (CV_32SC1) vectors, i.e. lists of 0-based indices, or 8-bit (CV_8UC1) masks of active variables/samples. The user may pass NULL pointers instead of either of the arguments, meaning that all of the variables/samples are used for training.

Additionally some algorithms can handle missing measurements, that is when certain features of certain training samples have unknown values (for example, they forgot to measure a temperature of patient A on Monday). The parameter missing_mask, an 8-bit matrix the same size as train_data, is used to mark the missed values (non-zero elements of the mask).

Usually, the previous model state is cleared by clear() before running the training procedure. However, some algorithms may optionally update the model state with the new training data, instead of resetting it.

CvStatModel::predict (view/add comments)

Predicts the response for the sample.

```
float CvStatMode::predict( const CvMat* sample[, <prediction_params>] )
const;
```

The method is used to predict the response for a new sample. In the case of classification the method returns the class label, in the case of regression - the output function value. The input sample must have as many components as the train_data passed to train contains. If the var_idx parameter is passed to train, it is remembered and then is used to extract only the necessary components from the input sample in the method predict.

The suffix ”const” means that prediction does not affect the internal model state, so the method can be safely called from within different threads.

19.2 Normal Bayes Classifier

This is a simple classification model assuming that feature vectors from each class are normally distributed (though, not necessarily independently distributed), so the whole data distribution function is assumed to be a Gaussian mixture, one component per class. Using the training data the

algorithm estimates mean vectors and covariance matrices for every class, and then it uses them for prediction.

[Fukunaga90] K. Fukunaga. Introduction to Statistical Pattern Recognition. second ed., New York: Academic Press, 1990.

---

cv::CvNormalBayesClassifier (view/add comments)

Bayes classifier for normally distributed data.

```
class CvNormalBayesClassifier : public CvStatModel
{
public:
    CvNormalBayesClassifier();
    virtual ~CvNormalBayesClassifier();

    CvNormalBayesClassifier( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx=0, const CvMat* _sample_idx=0 );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx = 0, const CvMat* _sample_idx=0, bool update=false );

    virtual float predict( const CvMat* _samples, CvMat* results=0 ) const;
    virtual void clear();

    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );
protected:
    ...
};
```

CvNormalBayesClassifier::train (view/add comments)

Trains the model.

```
bool CvNormalBayesClassifier::train(
           const CvMat* _train_data,
           const CvMat* _responses,
```

```
            const CvMat* _var_idx =0,
            const CvMat* _sample_idx=0,
            bool update=false );
```

The method trains the Normal Bayes classifier. It follows the conventions of the generic `train` "method" with the following limitations: only CV_ROW_SAMPLE data layout is supported; the input variables are all ordered; the output variable is categorical (i.e. elements of `_responses` must be integer numbers, though the vector may have `CV_32FC1` type), and missing measurements are not supported.

In addition, there is an `update` flag that identifies whether the model should be trained from scratch (`update=false`) or should be updated using the new training data (`update=true`).

CvNormalBayesClassifier::predict (view/add comments)

Predicts the response for sample(s)

```
float CvNormalBayesClassifier::predict(
            const CvMat* samples,
            CvMat* results=0 ) const;
```

The method `predict` estimates the most probable classes for the input vectors. The input vectors (one or more) are stored as rows of the matrix `samples`. In the case of multiple input vectors, there should be one output vector `results`. The predicted class for a single input vector is returned by the method.

19.3 K Nearest Neighbors

The algorithm caches all of the training samples, and predicts the response for a new sample by analyzing a certain number (**K**) of the nearest neighbors of the sample (using voting, calculating weighted sum etc.) The method is sometimes referred to as "learning by example", because for prediction it looks for the feature vector with a known response that is closest to the given vector.

---

cv::CvKNearest (view/add comments)

K Nearest Neighbors model.

```
class CvKNearest : public CvStatModel
{
public:

    CvKNearest();
    virtual ~CvKNearest();

    CvKNearest( const CvMat* _train_data, const CvMat* _responses,
                const CvMat* _sample_idx=0, bool _is_regression=false, int max_k=32 );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
                        const CvMat* _sample_idx=0, bool is_regression=false,
                        int _max_k=32, bool _update_base=false );

    virtual float find_nearest( const CvMat* _samples, int k, CvMat* results,
        const float** neighbors=0, CvMat* neighbor_responses=0, CvMat* dist=0 ) const;

    virtual void clear();
    int get_max_k() const;
    int get_var_count() const;
    int get_sample_count() const;
    bool is_regression() const;

protected:
    ...
};
```

CvKNearest::train (view/add comments)

Trains the model.

```
bool CvKNearest::train(
            const CvMat* _train_data,
            const CvMat* _responses,
            const CvMat* _sample_idx=0,
            bool is_regression=false,
            int _max_k=32,
            bool _update_base=false );
```

The method trains the K-Nearest model. It follows the conventions of generic `train` "method"

with the following limitations: only CV_ROW_SAMPLE data layout is supported, the input variables are all ordered, the output variables can be either categorical (`is_regression=false`) or ordered (`is_regression=true`), variable subsets (`var_idx`) and missing measurements are not supported.

The parameter `_max_k` specifies the number of maximum neighbors that may be passed to the method `find_nearest`.

The parameter `_update_base` specifies whether the model is trained from scratch (`_update_base=false`), or it is updated using the new training data (`_update_base=true`). In the latter case the parameter `_max_k` must not be larger than the original value.

CvKNearest::find_nearest (view/add comments)

Finds the neighbors for the input vectors.

```
float CvKNearest::find_nearest(
            const CvMat* _samples,
            int k, CvMat* results=0,
            const float** neighbors=0,
            CvMat* neighbor_responses=0,
            CvMat* dist=0 ) const;
```

For each input vector (which are the rows of the matrix `_samples`) the method finds the $k \leq$ `get_max_k()` nearest neighbor. In the case of regression, the predicted result will be a mean value of the particular vector's neighbor responses. In the case of classification the class is determined by voting.

For custom classification/regression prediction, the method can optionally return pointers to the neighbor vectors themselves (`neighbors`, an array of `k*_samples->rows` pointers), their corresponding output values (`neighbor_responses`, a vector of `k*_samples->rows` elements) and the distances from the input vectors to the neighbors (`dist`, also a vector of `k*_samples->rows` elements).

For each input vector the neighbors are sorted by their distances to the vector.

If only a single input vector is passed, all output matrices are optional and the predicted value is returned by the method.

```
#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
```

```
{
    const int K = 10;
    int i, j, k, accuracy;
    float response;
    int train_sample_count = 100;
    CvRNG rng_state = cvRNG(-1);
    CvMat* trainData = cvCreateMat( train_sample_count, 2, CV_32FC1 );
    CvMat* trainClasses = cvCreateMat( train_sample_count, 1, CV_32FC1 );
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    float _sample[2];
    CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
    cvZero( img );

    CvMat trainData1, trainData2, trainClasses1, trainClasses2;

    // form the training samples
    cvGetRows( trainData, &trainData1, 0, train_sample_count/2 );
    cvRandArr( &rng_state, &trainData1, CV_RAND_NORMAL, cvScalar(200,200), cvScalar(50,50)

    cvGetRows( trainData, &trainData2, train_sample_count/2, train_sample_count );
    cvRandArr( &rng_state, &trainData2, CV_RAND_NORMAL, cvScalar(300,300), cvScalar(50,50)

    cvGetRows( trainClasses, &trainClasses1, 0, train_sample_count/2 );
    cvSet( &trainClasses1, cvScalar(1) );

    cvGetRows( trainClasses, &trainClasses2, train_sample_count/2, train_sample_count );
    cvSet( &trainClasses2, cvScalar(2) );

    // learn classifier
    CvKNearest knn( trainData, trainClasses, 0, false, K );
    CvMat* nearests = cvCreateMat( 1, K, CV_32FC1);

    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;

            // estimates the response and get the neighbors' labels
            response = knn.find_nearest(&sample,K,0,0,nearests,0);

            // compute the number of neighbors representing the majority
            for( k = 0, accuracy = 0; k < K; k++ )
            {
```

```
                if( nearests->data.fl[k] == response)
                    accuracy++;
            }
            // highlight the pixel depending on the accuracy (or confidence)
            cvSet2D( img, i, j, response == 1 ?
                (accuracy > 5 ? CV_RGB(180,0,0) : CV_RGB(180,120,0)) :
                (accuracy > 5 ? CV_RGB(0,180,0) : CV_RGB(120,120,0)) );
        }
    }

    // display the original training samples
    for( i = 0; i < train_sample_count/2; i++ )
    {
        CvPoint pt;
        pt.x = cvRound(trainData1.data.fl[i*2]);
        pt.y = cvRound(trainData1.data.fl[i*2+1]);
        cvCircle( img, pt, 2, CV_RGB(255,0,0), CV_FILLED );
        pt.x = cvRound(trainData2.data.fl[i*2]);
        pt.y = cvRound(trainData2.data.fl[i*2+1]);
        cvCircle( img, pt, 2, CV_RGB(0,255,0), CV_FILLED );
    }

    cvNamedWindow( "classifier result", 1 );
    cvShowImage( "classifier result", img );
    cvWaitKey(0);

    cvReleaseMat( &trainClasses );
    cvReleaseMat( &trainData );
    return 0;
}
```

19.4 Support Vector Machines

Originally, support vector machines (SVM) was a technique for building an optimal (in some sense) binary (2-class) classifier. Then the technique has been extended to regression and clustering problems. SVM is a partial case of kernel-based methods, it maps feature vectors into higher-dimensional space using some kernel function, and then it builds an optimal linear discriminating function in this space (or an optimal hyper-plane that fits into the training data, ...). in the case of SVM the kernel is not defined explicitly. Instead, a distance between any 2 points in the hyper-space needs to be defined.

The solution is optimal in a sense that the margin between the separating hyper-plane and the nearest feature vectors from the both classes (in the case of 2-class classifier) is maximal. The

feature vectors that are the closest to the hyper-plane are called ”support vectors”, meaning that the position of other vectors does not affect the hyper-plane (the decision function).

There are a lot of good references on SVM. Here are only a few ones to start with.

- **[Burges98] C. Burges. ”A tutorial on support vector machines for pattern recognition”, Knowledge Discovery and Data Mining 2(2), 1998.** (available online at http://citeseer.ist.psu.edu/burges98tutorial.html).
- **LIBSVM - A Library for Support Vector Machines. By Chih-Chung Chang and Chih-Jen Lin** (http://www.csie.ntu.edu.tw/~cjlin/libsvm/)

---

cv::CvSVM (view/add comments)

Support Vector Machines.

```
class CvSVM : public CvStatModel
{
public:
    // SVM type
    enum { C_SVC=100, NU_SVC=101, ONE_CLASS=102, EPS_SVR=103, NU_SVR=104 };

    // SVM kernel type
    enum { LINEAR=0, POLY=1, RBF=2, SIGMOID=3 };

    // SVM params type
    enum { C=0, GAMMA=1, P=2, NU=3, COEF=4, DEGREE=5 };

    CvSVM();
    virtual ~CvSVM();

    CvSVM( const CvMat* _train_data, const CvMat* _responses,
           const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
           CvSVMParams _params=CvSVMParams() );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
                        const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
                        CvSVMParams _params=CvSVMParams() );

    virtual bool train_auto( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx, const CvMat* _sample_idx, CvSVMParams _params,
        int k_fold = 10,
        CvParamGrid C_grid      = get_default_grid(CvSVM::C),
        CvParamGrid gamma_grid  = get_default_grid(CvSVM::GAMMA),
        CvParamGrid p_grid      = get_default_grid(CvSVM::P),
```

```
        CvParamGrid nu_grid     = get_default_grid(CvSVM::NU),
        CvParamGrid coef_grid   = get_default_grid(CvSVM::COEF),
        CvParamGrid degree_grid = get_default_grid(CvSVM::DEGREE) );

    virtual float predict( const CvMat* _sample ) const;
    virtual int get_support_vector_count() const;
    virtual const float* get_support_vector(int i) const;
    virtual CvSVMParams get_params() const { return params; };
    virtual void clear();

    static CvParamGrid get_default_grid( int param_id );

    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );
    int get_var_count() const { return var_idx ? var_idx->cols : var_all; }

protected:
    ...
};
```

---

cv::CvSVMParams (view/add comments)

SVM training parameters.

```
struct CvSVMParams
{
    CvSVMParams();
    CvSVMParams( int _svm_type, int _kernel_type,
                 double _degree, double _gamma, double _coef0,
                 double _C, double _nu, double _p,
                 CvMat* _class_weights, CvTermCriteria _term_crit );

    int         svm_type;
    int         kernel_type;
    double      degree; // for poly
    double      gamma;  // for poly/rbf/sigmoid
    double      coef0;  // for poly/sigmoid

    double      C;  // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
    double      nu; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
    double      p; // for CV_SVM_EPS_SVR
```

```
    CvMat*      class_weights; // for CV_SVM_C_SVC
    CvTermCriteria term_crit; // termination criteria
};
```

The structure must be initialized and passed to the training method of CvSVM .

CvSVM::train (view/add comments)

Trains SVM.

```
bool CvSVM::train(
            const CvMat* _train_data,
            const CvMat* _responses,
            const CvMat* _var_idx=0,
            const CvMat* _sample_idx=0,
            CvSVMParams _params=CvSVMParams() );
```

The method trains the SVM model. It follows the conventions of the generic `train` "method" with the following limitations: only the CV_ROW_SAMPLE data layout is supported, the input variables are all ordered, the output variables can be either categorical (`_params.svm_type=CvSVM::C_SVC` or `_params.svm_type=CvSVM::NU_SVC`), or ordered (`_params.svm_type=CvSVM::EPS_SVR` or `_params.svm_type=CvSVM::NU_SVR`), or not required at all (`_params.svm_type=CvSVM::ONE_CLASS`), missing measurements are not supported.

All the other parameters are gathered in CvSVMParams structure.

CvSVM::train_auto (view/add comments)

Trains SVM with optimal parameters.

```
train_auto(
            const CvMat* _train_data,
            const CvMat* _responses,
            const CvMat* _var_idx,
            const CvMat* _sample_idx,
            CvSVMParams params,
            int k_fold = 10,
```

```
CvParamGrid C_grid = get_default_grid(CvSVM::C),
CvParamGrid gamma_grid = get_default_grid(CvSVM::GAMMA),
CvParamGrid p_grid = get_default_grid(CvSVM::P),
CvParamGrid nu_grid = get_default_grid(CvSVM::NU),
CvParamGrid coef_grid = get_default_grid(CvSVM::COEF),
CvParamGrid degree_grid = get_default_grid(CvSVM::DEGREE) );
```

k_fold Cross-validation parameter. The training set is divided into k_fold subsets, one subset being used to train the model, the others forming the test set. So, the SVM algorithm is executed k_fold times.

The method trains the SVM model automatically by choosing the optimal parameters C, gamma, p, nu, coef0, degree from CvSVMParams . By optimal one means that the cross-validation estimate of the test set error is minimal. The parameters are iterated by a logarithmic grid, for example, the parameter gamma takes the values in the set (min, $min * step$, $min * step^2$, ... $min * step^n$) where min is gamma_grid.min_val, $step$ is gamma_grid.step, and n is the maximal index such, that

$$\texttt{gamma_grid.min_val} * \texttt{gamma_grid.step}^n < \texttt{gamma_grid.max_val}$$

So step must always be greater than 1.

If there is no need in optimization in some parameter, the according grid step should be set to any value less or equal to 1. For example, to avoid optimization in gamma one should set gamma_grid.step = 0, gamma_grid.min_val, gamma_grid.max_val being arbitrary numbers. In this case, the value params.gamma will be taken for gamma.

And, finally, if the optimization in some parameter is required, but there is no idea of the corresponding grid, one may call the function CvSVM::get_default_grid. In order to generate a grid, say, for gamma, call CvSVM::get_default_grid(CvSVM::GAMMA).

This function works for the case of classification (params.svm_type=CvSVM::C_SVC or params.svm_type= as well as for the regression (params.svm_type=CvSVM::EPS_SVR or params.svm_type=CvSVM::NU_SVR). If params.svm_type=CvSVM::ONE_CLASS, no optimization is made and the usual SVM with specified in params parameters is executed.

CvSVM::get_default_grid (view/add comments)

Generates a grid for the SVM parameters.

```
CvParamGrid CvSVM::get_default_grid( int param_id );
```

param_id Must be one of the following:

CvSVM::C

CvSVM::GAMMA

CvSVM::P

CvSVM::NU

CvSVM::COEF

CvSVM::DEGREE .

The grid will be generated for the parameter with this ID.

The function generates a grid for the specified parameter of the SVM algorithm. The grid may be passed to the function `CvSVM::train_auto`.

CvSVM::get_params (view/add comments)

Returns the current SVM parameters.

```
CvSVMParams CvSVM::get_params() const;
```

This function may be used to get the optimal parameters that were obtained while automatically training `CvSVM::train_auto`.

CvSVM::get_support_vector* (view/add comments)

Retrieves the number of support vectors and the particular vector.

```
int CvSVM::get_support_vector_count() const;
const float* CvSVM::get_support_vector(int i) const;
```

The methods can be used to retrieve the set of support vectors.

19.5 Decision Trees

The ML classes discussed in this section implement Classification And Regression Tree algorithms, which are described in [Breiman84].

The class CvDTree represents a single decision tree that may be used alone, or as a base class in tree ensembles (see Boosting and Random Trees).

A decision tree is a binary tree (i.e. tree where each non-leaf node has exactly 2 child nodes). It can be used either for classification, when each tree leaf is marked with some class label (multiple leafs may have the same label), or for regression, when each tree leaf is also assigned a constant (so the approximation function is piecewise constant).

Predicting with Decision Trees

To reach a leaf node, and to obtain a response for the input feature vector, the prediction procedure starts with the root node. From each non-leaf node the procedure goes to the left (i.e. selects the left child node as the next observed node), or to the right based on the value of a certain variable, whose index is stored in the observed node. The variable can be either ordered or categorical. In the first case, the variable value is compared with the certain threshold (which is also stored in the node); if the value is less than the threshold, the procedure goes to the left, otherwise, to the right (for example, if the weight is less than 1 kilogram, the procedure goes to the left, else to the right). And in the second case the discrete variable value is tested to see if it belongs to a certain subset of values (also stored in the node) from a limited set of values the variable could take; if yes, the procedure goes to the left, else - to the right (for example, if the color is green or red, go to the left, else to the right). That is, in each node, a pair of entities (variable_index, decision_rule (threshold/subset)) is used. This pair is called a split (split on the variable variable_index). Once a leaf node is reached, the value assigned to this node is used as the output of prediction procedure.

Sometimes, certain features of the input vector are missed (for example, in the darkness it is difficult to determine the object color), and the prediction procedure may get stuck in the certain node (in the mentioned example if the node is split by color). To avoid such situations, decision trees use so-called surrogate splits. That is, in addition to the best "primary" split, every tree node may also be split on one or more other variables with nearly the same results.

Training Decision Trees

The tree is built recursively, starting from the root node. All of the training data (feature vectors and the responses) is used to split the root node. In each node the optimum decision rule (i.e. the best "primary" split) is found based on some criteria (in ML `gini` "purity" criteria is used for classification, and sum of squared errors is used for regression). Then, if necessary, the surrogate splits are found that resemble the results of the primary split on the training data; all of the data is

divided using the primary and the surrogate splits (just like it is done in the prediction procedure) between the left and the right child node. Then the procedure recursively splits both left and right nodes. At each node the recursive procedure may stop (i.e. stop splitting the node further) in one of the following cases:

- depth of the tree branch being constructed has reached the specified maximum value.
- number of training samples in the node is less than the specified threshold, when it is not statistically representative to split the node further.
- all the samples in the node belong to the same class (or, in the case of regression, the variation is too small).
- the best split found does not give any noticeable improvement compared to a random choice.

When the tree is built, it may be pruned using a cross-validation procedure, if necessary. That is, some branches of the tree that may lead to the model overfitting are cut off. Normally this procedure is only applied to standalone decision trees, while tree ensembles usually build small enough trees and use their own protection schemes against overfitting.

Variable importance

Besides the obvious use of decision trees - prediction, the tree can be also used for various data analysis. One of the key properties of the constructed decision tree algorithms is that it is possible to compute importance (relative decisive power) of each variable. For example, in a spam filter that uses a set of words occurred in the message as a feature vector, the variable importance rating can be used to determine the most ”spam-indicating” words and thus help to keep the dictionary size reasonable.

Importance of each variable is computed over all the splits on this variable in the tree, primary and surrogate ones. Thus, to compute variable importance correctly, the surrogate splits must be enabled in the training parameters, even if there is no missing data.

[Breiman84] Breiman, L., Friedman, J. Olshen, R. and Stone, C. (1984), ”Classification and Regression Trees”, Wadsworth.

---

cv::CvDTreeSplit (view/add comments)

Decision tree node split.

```
struct CvDTreeSplit
{
    int var_idx;
    int inversed;
```

```
    float quality;
    CvDTreeSplit* next;
    union
    {
        int subset[2];
        struct
        {
            float c;
            int split_point;
        }
        ord;
    };
};
```

---

cv::CvDTreeNode (view/add comments)

Decision tree node.

```
struct CvDTreeNode
{
    int class_idx;
    int Tn;
    double value;

    CvDTreeNode* parent;
    CvDTreeNode* left;
    CvDTreeNode* right;

    CvDTreeSplit* split;

    int sample_count;
    int depth;
    ...
};
```

Other numerous fields of `CvDTreeNode` are used internally at the training stage.

---

cv::CvDTreeParams (view/add comments)

Decision tree training parameters.

```
struct CvDTreeParams
{
    int max_categories;
```

```
    int max_depth;
    int min_sample_count;
    int cv_folds;
    bool use_surrogates;
    bool use_1se_rule;
    bool truncate_pruned_tree;
    float regression_accuracy;
    const float* priors;

    CvDTreeParams() : max_categories(10), max_depth(INT_MAX), min_sample_count(10),
        cv_folds(10), use_surrogates(true), use_1se_rule(true),
        truncate_pruned_tree(true), regression_accuracy(0.01f), priors(0)
    {}

    CvDTreeParams( int _max_depth, int _min_sample_count,
                   float _regression_accuracy, bool _use_surrogates,
                   int _max_categories, int _cv_folds,
                   bool _use_1se_rule, bool _truncate_pruned_tree,
                   const float* _priors );
};
```

The structure contains all the decision tree training parameters. There is a default constructor that initializes all the parameters with the default values tuned for standalone classification tree. Any of the parameters can be overridden then, or the structure may be fully initialized using the advanced variant of the constructor.

---

cv::CvDTreeTrainData (view/add comments)

Decision tree training data and shared data for tree ensembles.

```
struct CvDTreeTrainData
{
    CvDTreeTrainData();
    CvDTreeTrainData( const CvMat* _train_data, int _tflag,
                      const CvMat* _responses, const CvMat* _var_idx=0,
                      const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                      const CvMat* _missing_mask=0,
                      const CvDTreeParams& _params=CvDTreeParams(),
                      bool _shared=false, bool _add_labels=false );
    virtual ~CvDTreeTrainData();

    virtual void set_data( const CvMat* _train_data, int _tflag,
                           const CvMat* _responses, const CvMat* _var_idx=0,
                           const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                           const CvMat* _missing_mask=0,
```

```
                        const CvDTreeParams& _params=CvDTreeParams(),
                        bool _shared=false, bool _add_labels=false,
                        bool _update_data=false );

    virtual void get_vectors( const CvMat* _subsample_idx,
         float* values, uchar* missing, float* responses,
         bool get_class_idx=false );

    virtual CvDTreeNode* subsample_data( const CvMat* _subsample_idx );

    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    // release all the data
    virtual void clear();

    int get_num_classes() const;
    int get_var_type(int vi) const;
    int get_work_var_count() const;

    virtual int* get_class_labels( CvDTreeNode* n );
    virtual float* get_ord_responses( CvDTreeNode* n );
    virtual int* get_labels( CvDTreeNode* n );
    virtual int* get_cat_var_data( CvDTreeNode* n, int vi );
    virtual CvPair32s32f* get_ord_var_data( CvDTreeNode* n, int vi );
    virtual int get_child_buf_idx( CvDTreeNode* n );

    ////////////////////////////////////

    virtual bool set_params( const CvDTreeParams& params );
    virtual CvDTreeNode* new_node( CvDTreeNode* parent, int count,
                                   int storage_idx, int offset );

    virtual CvDTreeSplit* new_split_ord( int vi, float cmp_val,
                int split_point, int inversed, float quality );
    virtual CvDTreeSplit* new_split_cat( int vi, float quality );
    virtual void free_node_data( CvDTreeNode* node );
    virtual void free_train_data();
    virtual void free_node( CvDTreeNode* node );

    int sample_count, var_all, var_count, max_c_count;
    int ord_var_count, cat_var_count;
    bool have_labels, have_priors;
    bool is_classifier;
```

```
    int buf_count, buf_size;
    bool shared;

    CvMat* cat_count;
    CvMat* cat_ofs;
    CvMat* cat_map;

    CvMat* counts;
    CvMat* buf;
    CvMat* direction;
    CvMat* split_buf;

    CvMat* var_idx;
    CvMat* var_type; // i-th element =
                     //   k<0  - ordered
                     //   k>=0 - categorical, see k-th element of cat_* arrays
    CvMat* priors;

    CvDTreeParams params;

    CvMemStorage* tree_storage;
    CvMemStorage* temp_storage;

    CvDTreeNode* data_root;

    CvSet* node_heap;
    CvSet* split_heap;
    CvSet* cv_heap;
    CvSet* nv_heap;

    CvRNG rng;
};
```

This structure is mostly used internally for storing both standalone trees and tree ensembles efficiently. Basically, it contains 3 types of information:

1. The training parameters, an instance of CvDTreeParams .

2. The training data, preprocessed in order to find the best splits more efficiently. For tree ensembles this preprocessed data is reused by all the trees. Additionally, the training data characteristics that are shared by all trees in the ensemble are stored here: variable types, the number of classes, class label compression map etc.

3. Buffers, memory storages for tree nodes, splits and other elements of the trees constructed.

There are 2 ways of using this structure. In simple cases (e.g. a standalone tree, or the ready-to-use "black box" tree ensemble from ML, like Random Trees or Boosting) there is no need to care or even to know about the structure - just construct the needed statistical model, train it and use it. The `CvDTreeTrainData` structure will be constructed and used internally. However, for custom tree algorithms, or another sophisticated cases, the structure may be constructed and used explicitly. The scheme is the following:

- The structure is initialized using the default constructor, followed by `set_data` (or it is built using the full form of constructor). The parameter `_shared` must be set to `true`.
- One or more trees are trained using this data, see the special form of the method `CvDTree::train`.
- Finally, the structure can be released only after all the trees using it are released.

---

cv::CvDTree (view/add comments)

Decision tree.

```
class CvDTree : public CvStatModel
{
public:
    CvDTree();
    virtual ~CvDTree();

    virtual bool train( const CvMat* _train_data, int _tflag,
                        const CvMat* _responses, const CvMat* _var_idx=0,
                        const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                        const CvMat* _missing_mask=0,
                        CvDTreeParams params=CvDTreeParams() );

    virtual bool train( CvDTreeTrainData* _train_data,
                        const CvMat* _subsample_idx );

    virtual CvDTreeNode* predict( const CvMat* _sample,
                                  const CvMat* _missing_data_mask=0,
                                  bool raw_mode=false ) const;
    virtual const CvMat* get_var_importance();
    virtual void clear();

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name );

    // special read & write methods for trees in the tree ensembles
    virtual void read( CvFileStorage* fs, CvFileNode* node,
```

```
                        CvDTreeTrainData* data );
    virtual void write( CvFileStorage* fs );

    const CvDTreeNode* get_root() const;
    int get_pruned_tree_idx() const;
    CvDTreeTrainData* get_data();

protected:

    virtual bool do_train( const CvMat* _subsample_idx );

    virtual void try_split_node( CvDTreeNode* n );
    virtual void split_node_data( CvDTreeNode* n );
    virtual CvDTreeSplit* find_best_split( CvDTreeNode* n );
    virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_surrogate_split_ord( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_surrogate_split_cat( CvDTreeNode* n, int vi );
    virtual double calc_node_dir( CvDTreeNode* node );
    virtual void complete_node_dir( CvDTreeNode* node );
    virtual void cluster_categories( const int* vectors, int vector_count,
        int var_count, int* sums, int k, int* cluster_labels );

    virtual void calc_node_value( CvDTreeNode* node );

    virtual void prune_cv();
    virtual double update_tree_rnc( int T, int fold );
    virtual int cut_tree( int T, int fold, double min_alpha );
    virtual void free_prune_data(bool cut_tree);
    virtual void free_tree();

    virtual void write_node( CvFileStorage* fs, CvDTreeNode* node );
    virtual void write_split( CvFileStorage* fs, CvDTreeSplit* split );
    virtual CvDTreeNode* read_node( CvFileStorage* fs,
                                    CvFileNode* node,
                                    CvDTreeNode* parent );
    virtual CvDTreeSplit* read_split( CvFileStorage* fs, CvFileNode* node );
    virtual void write_tree_nodes( CvFileStorage* fs );
    virtual void read_tree_nodes( CvFileStorage* fs, CvFileNode* node );

    CvDTreeNode* root;

    int pruned_tree_idx;
```

```
    CvMat* var_importance;

    CvDTreeTrainData* data;
};
```

CvDTree::train (view/add comments)

Trains a decision tree.

```
bool CvDTree::train(
            const CvMat* _train_data,
            int _tflag,
            const CvMat* _responses,
            const CvMat* _var_idx=0,
            const CvMat* _sample_idx=0,
            const CvMat* _var_type=0,
            const CvMat* _missing_mask=0,
            CvDTreeParams params=CvDTreeParams() );

bool CvDTree::train( CvDTreeTrainData* _train_data, const CvMat*
_subsample_idx );
```

There are 2 `train` methods in `CvDTree`.

The first method follows the generic `CvStatModel::train` conventions, it is the most complete form. Both data layouts (`_tflag=CV_ROW_SAMPLE` and `_tflag=CV_COL_SAMPLE`) are supported, as well as sample and variable subsets, missing measurements, arbitrary combinations of input and output variable types etc. The last parameter contains all of the necessary training parameters, see the CvDTreeParams description.

The second method `train` is mostly used for building tree ensembles. It takes the pre-constructed CvDTreeTrainData instance and the optional subset of training set. The indices in `_subsample_idx` are counted relatively to the `_sample_idx`, passed to `CvDTreeTrainData` constructor. For example, if `_sample_idx=[1, 5, 7, 100]`, then `_subsample_idx=[0,3]` means that the samples `[1, 100]` of the original training set are used.

CvDTree::predict (view/add comments)

Returns the leaf node of the decision tree corresponding to the input vector.

```
CvDTreeNode* CvDTree::predict(
            const CvMat* _sample,
            const CvMat* _missing_data_mask=0,
            bool raw_mode=false ) const;
```

The method takes the feature vector and the optional missing measurement mask on input, traverses the decision tree and returns the reached leaf node on output. The prediction result, either the class label or the estimated function value, may be retrieved as the `value` field of the CvDTreeNode structure, for example: dtree->predict(sample,mask)->value.

The last parameter is normally set to `false`, implying a regular input. If it is `true`, the method assumes that all the values of the discrete input variables have been already normalized to 0 to $num_of_categories_i - 1$ ranges. (as the decision tree uses such normalized representation internally). It is useful for faster prediction with tree ensembles. For ordered input variables the flag is not used.

Example: Building A Tree for Classifying Mushrooms. See the `mushroom.cpp` sample that demonstrates how to build and use the decision tree.

19.6 Boosting

A common machine learning task is supervised learning. In supervised learning, the goal is to learn the functional relationship $F : y = F(x)$ between the input x and the output y. Predicting the qualitative output is called classification, while predicting the quantitative output is called regression.

Boosting is a powerful learning concept, which provide a solution to the supervised classification learning task. It combines the performance of many "weak" classifiers to produce a powerful 'committee' HTF01 . A weak classifier is only required to be better than chance, and thus can be very simple and computationally inexpensive. Many of them smartly combined, however, results in a strong classifier, which often outperforms most 'monolithic' strong classifiers such as SVMs and Neural Networks.

Decision trees are the most popular weak classifiers used in boosting schemes. Often the simplest decision trees with only a single split node per tree (called stumps) are sufficient.

The boosted model is based on N training examples $(x_i, y_i)1N$ with $x_i \in R^K$ and $y_i \in -1, +1$. x_i is a K-component vector. Each component encodes a feature relevant for the learning task at hand. The desired two-class output is encoded as -1 and +1.

Different variants of boosting are known such as Discrete Adaboost, Real AdaBoost, LogitBoost, and Gentle AdaBoost FHT98 . All of them are very similar in their overall structure.

Therefore, we will look only at the standard two-class Discrete AdaBoost algorithm as shown in the box below. Each sample is initially assigned the same weight (step 2). Next a weak classifier $f_{m(x)}$ is trained on the weighted training data (step 3a). Its weighted training error and scaling factor c_m is computed (step 3b). The weights are increased for training samples, which have been misclassified (step 3c). All weights are then normalized, and the process of finding the next weak classifier continues for another M-1 times. The final classifier $F(x)$ is the sign of the weighted sum over the individual weak classifiers (step 4).

- Given N examples $(x_i, y_i)1N$ with $x_i \in R^K, y_i \in -1, +1$.
- Start with weights $w_i = 1/N, i = 1, ..., N$.
- Repeat for $m = 1, 2, ..., M$:
  - Fit the classifier $f_m(x) \in -1, 1$, using weights w_i on the training data.
  - Compute $err_m = E_w[1_{(y \neq f_m(x))}], c_m = log((1 - err_m)/err_m)$.
  - Set $w_i \Leftarrow w_i exp[c_m 1_{(y_i \neq f_m(x_i))}], i = 1, 2, ..., N$, and renormalize so that $\Sigma i w_i = 1$.
  - Output the classifier sign$[\Sigma m = 1 M c_m f_m(x)]$.

Two-class Discrete AdaBoost Algorithm: Training (steps 1 to 3) and Evaluation (step 4)

NOTE: As well as the classical boosting methods, the current implementation supports 2-class classifiers only. For M>2 classes there is the **AdaBoost.MH** algorithm, described in FHT98 , that reduces the problem to the 2-class problem, yet with a much larger training set.

In order to reduce computation time for boosted models without substantially losing accuracy, the influence trimming technique may be employed. As the training algorithm proceeds and the number of trees in the ensemble is increased, a larger number of the training samples are classified correctly and with increasing confidence, thereby those samples receive smaller weights on the subsequent iterations. Examples with very low relative weight have small impact on training of the weak classifier. Thus such examples may be excluded during the weak classifier training without having much effect on the induced classifier. This process is controlled with the weight_trim_rate parameter. Only examples with the summary fraction weight_trim_rate of the total weight mass are used in the weak classifier training. Note that the weights for **all** training examples are recomputed at each training iteration. Examples deleted at a particular iteration may be used again for learning some of the weak classifiers further FHT98 .

[HTF01] Hastie, T., Tibshirani, R., Friedman, J. H. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. 2001.

[FHT98] Friedman, J. H., Hastie, T. and Tibshirani, R. Additive Logistic Regression: a Statistical View of Boosting. Technical Report, Dept. of Statistics, Stanford University, 1998.

cv::CvBoostParams (view/add comments)

Boosting training parameters.

```
struct CvBoostParams : public CvDTreeParams
{
    int boost_type;
    int weak_count;
    int split_criteria;
    double weight_trim_rate;

    CvBoostParams();
    CvBoostParams( int boost_type, int weak_count, double weight_trim_rate,
                   int max_depth, bool use_surrogates, const float* priors );
};
```

The structure is derived from CvDTreeParams , but not all of the decision tree parameters are supported. In particular, cross-validation is not supported.

cv::CvBoostTree (view/add comments)

Weak tree classifier.

```
class CvBoostTree: public CvDTree
{
public:
    CvBoostTree();
    virtual ~CvBoostTree();

    virtual bool train( CvDTreeTrainData* _train_data,
                        const CvMat* subsample_idx, CvBoost* ensemble );
    virtual void scale( double s );
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvBoost* ensemble, CvDTreeTrainData* _data );
    virtual void clear();

protected:
    ...
    CvBoost* ensemble;
};
```

The weak classifier, a component of the boosted tree classifier CvBoost , is a derivative of CvDTree . Normally, there is no need to use the weak classifiers directly, however they can be accessed as elements of the sequence `CvBoost::weak`, retrieved by `CvBoost::get_weak_predictors`.

Note, that in the case of LogitBoost and Gentle AdaBoost each weak predictor is a regression tree, rather than a classification tree. Even in the case of Discrete AdaBoost and Real AdaBoost the `CvBoostTree::predict` return value (`CvDTreeNode::value`) is not the output class label; a negative value "votes" for class #0, a positive - for class #1. And the votes are weighted. The weight of each individual tree may be increased or decreased using the method `CvBoostTree::scale`.

---

cv::CvBoost (view/add comments)

Boosted tree classifier.

```
class CvBoost : public CvStatModel
{
public:
    // Boosting type
    enum { DISCRETE=0, REAL=1, LOGIT=2, GENTLE=3 };

    // Splitting criteria
    enum { DEFAULT=0, GINI=1, MISCLASS=3, SQERR=4 };

    CvBoost();
    virtual ~CvBoost();

    CvBoost( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams() );

    virtual bool train( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams(),
             bool update=false );

    virtual float predict( const CvMat* _sample, const CvMat* _missing=0,
                           CvMat* weak_responses=0, CvSlice slice=CV_WHOLE_SEQ,
                           bool raw_mode=false ) const;

    virtual void prune( CvSlice slice );

    virtual void clear();
```

```
    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );

    CvSeq* get_weak_predictors();
    const CvBoostParams& get_params() const;
    ...

protected:
    virtual bool set_params( const CvBoostParams& _params );
    virtual void update_weights( CvBoostTree* tree );
    virtual void trim_weights();
    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvDTreeTrainData* data;
    CvBoostParams params;
    CvSeq* weak;
    ...
};
```

CvBoost::train (view/add comments)

Trains a boosted tree classifier.

```
bool CvBoost::train(
            const CvMat* _train_data,
            int _tflag,
            const CvMat* _responses,
            const CvMat* _var_idx=0,
            const CvMat* _sample_idx=0,
            const CvMat* _var_type=0,
            const CvMat* _missing_mask=0,
            CvBoostParams params=CvBoostParams(),
            bool update=false );
```

The train method follows the common template; the last parameter `update` specifies whether the classifier needs to be updated (i.e. the new weak tree classifiers added to the existing ensemble), or the classifier needs to be rebuilt from scratch. The responses must be categorical, i.e. boosted trees can not be built for regression, and there should be 2 classes.

CvBoost::predict (view/add comments)

Predicts a response for the input sample.

```
float CvBoost::predict(
            const CvMat* sample,
            const CvMat* missing=0,
            CvMat* weak_responses=0,
            CvSlice slice=CV_WHOLE_SEQ,
            bool raw_mode=false ) const;
```

The method `CvBoost::predict` runs the sample through the trees in the ensemble and returns the output class label based on the weighted voting.

CvBoost::prune (view/add comments)

Removes the specified weak classifiers.

```
void CvBoost::prune( CvSlice slice );
```

The method removes the specified weak classifiers from the sequence. Note that this method should not be confused with the pruning of individual decision trees, which is currently not supported.

CvBoost::get_weak_predictors (view/add comments)

Returns the sequence of weak tree classifiers.

```
CvSeq* CvBoost::get_weak_predictors();
```

The method returns the sequence of weak classifiers. Each element of the sequence is a pointer to a `CvBoostTree` class (or, probably, to some of its derivatives).

19.7 Random Trees

Random trees have been introduced by Leo Breiman and Adele Cutler: http://www.stat.berkeley.edu/users/breiman/RandomForests/. The algorithm can deal with both classification and regression problems. Random trees is a collection (ensemble) of tree predictors that is called **forest** further in this section (the term has been also introduced by L. Breiman). The classification works as follows: the random trees classifier takes the input feature vector, classifies it with every tree in the forest, and outputs the class label that recieved the majority of "votes". In the case of regression the classifier response is the average of the responses over all the trees in the forest.

All the trees are trained with the same parameters, but on the different training sets, which are generated from the original training set using the bootstrap procedure: for each training set we randomly select the same number of vectors as in the original set (=N). The vectors are chosen with replacement. That is, some vectors will occur more than once and some will be absent. At each node of each tree trained not all the variables are used to find the best split, rather than a random subset of them. With each node a new subset is generated, however its size is fixed for all the nodes and all the trees. It is a training parameter, set to $\sqrt{number_of_variables}$ by default. None of the trees that are built are pruned.

In random trees there is no need for any accuracy estimation procedures, such as cross-validation or bootstrap, or a separate test set to get an estimate of the training error. The error is estimated internally during the training. When the training set for the current tree is drawn by sampling with replacement, some vectors are left out (so-called *oob (out-of-bag) data*). The size of oob data is about N/3. The classification error is estimated by using this oob-data as following:

- Get a prediction for each vector, which is oob relatively to the i-th tree, using the very i-th tree.
- After all the trees have been trained, for each vector that has ever been oob, find the class-"winner" for it (i.e. the class that has got the majority of votes in the trees, where the vector was oob) and compare it to the ground-truth response.
- Then the classification error estimate is computed as ratio of number of misclassified oob vectors to all the vectors in the original data. In the case of regression the oob-error is computed as the squared error for oob vectors difference divided by the total number of vectors.

References:

- Machine Learning, Wald I, July 2002. http://stat-www.berkeley.edu/users/breiman/wald2002-1.pdf
- Looking Inside the Black Box, Wald II, July 2002. http://stat-www.berkeley.edu/users/breiman/wald2002-2.pdf

- Software for the Masses, Wald III, July 2002. http://stat-www.berkeley.edu/users/breiman/wald2002-3.pdf
- And other articles from the web site http://www.stat.berkeley.edu/users/breiman/RandomForests/cc_home.htm.

---

cv::CvRTParams (view/add comments)

Training Parameters of Random Trees.

```
struct CvRTParams : public CvDTreeParams
{
    bool calc_var_importance;
    int nactive_vars;
    CvTermCriteria term_crit;

    CvRTParams() : CvDTreeParams( 5, 10, 0, false, 10, 0, false, false, 0 ),
        calc_var_importance(false), nactive_vars(0)
    {
        term_crit = cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 50, 0.1 );
    }

    CvRTParams( int _max_depth, int _min_sample_count,
                float _regression_accuracy, bool _use_surrogates,
                int _max_categories, const float* _priors,
                bool _calc_var_importance,
                int _nactive_vars, int max_tree_count,
                float forest_accuracy, int termcrit_type );
};
```

The set of training parameters for the forest is the superset of the training parameters for a single tree. However, Random trees do not need all the functionality/features of decision trees, most noticeably, the trees are not pruned, so the cross-validation parameters are not used.

---

cv::CvRTrees (view/add comments)

Random Trees.

```
class CvRTrees : public CvStatModel
{
public:
    CvRTrees();
    virtual ~CvRTrees();
    virtual bool train( const CvMat* _train_data, int _tflag,
                        const CvMat* _responses, const CvMat* _var_idx=0,
```

```
                        const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                        const CvMat* _missing_mask=0,
                        CvRTParams params=CvRTParams() );
    virtual float predict( const CvMat* sample, const CvMat* missing = 0 )
                                                                    const;
    virtual void clear();

    virtual const CvMat* get_var_importance();
    virtual float get_proximity( const CvMat* sample_1, const CvMat* sample_2 )
                                                                        const;

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name );

    CvMat* get_active_var_mask();
    CvRNG* get_rng();

    int get_tree_count() const;
    CvForestTree* get_tree(int i) const;

protected:

    bool grow_forest( const CvTermCriteria term_crit );

    // array of the trees of the forest
    CvForestTree** trees;
    CvDTreeTrainData* data;
    int ntrees;
    int nclasses;
    ...
};
```

CvRTrees::train (view/add comments)

Trains the Random Trees model.

```
bool CvRTrees::train(
           const CvMat* train_data,
           int tflag,
           const CvMat* responses,
           const CvMat* comp_idx=0,
```

```
            const CvMat* sample_idx=0,
            const CvMat* var_type=0,
            const CvMat* missing_mask=0,
            CvRTParams params=CvRTParams() );
```

The method `CvRTrees::train` is very similar to the first form of `CvDTree::train`() and follows the generic method `CvStatModel::train` conventions. All of the specific to the algorithm training parameters are passed as a CvRTParams instance. The estimate of the training error (`oob-error`) is stored in the protected class member `oob_error`.

CvRTrees::predict (view/add comments)

Predicts the output for the input sample.

```
double CvRTrees::predict(
            const CvMat* sample,
            const CvMat* missing=0 ) const;
```

The input parameters of the prediction method are the same as in `CvDTree::predict`, but the return value type is different. This method returns the cumulative result from all the trees in the forest (the class that receives the majority of voices, or the mean of the regression function estimates).

CvRTrees::get_var_importance (view/add comments)

Retrieves the variable importance array.

```
const CvMat* CvRTrees::get_var_importance() const;
```

The method returns the variable importance vector, computed at the training stage when CvRTParams `::calc_var_importance` is set. If the training flag is not set, then the `NULL` pointer is returned. This is unlike decision trees, where variable importance can be computed anytime after the training.

CvRTrees::get_proximity (view/add comments)

Retrieves the proximity measure between two training samples.

```
float CvRTrees::get_proximity(
           const CvMat* sample_1,
           const CvMat* sample_2 ) const;
```

The method returns proximity measure between any two samples (the ratio of the those trees in the ensemble, in which the samples fall into the same leaf node, to the total number of the trees).

Example: Prediction of mushroom goodness using random trees classifier

```
#include <float.h>
#include <stdio.h>
#include <ctype.h>
#include "ml.h"

int main( void )
{
    CvStatModel*    cls = NULL;
    CvFileStorage*  storage = cvOpenFileStorage( "Mushroom.xml",
                                                 NULL,CV_STORAGE_READ );
    CvMat*          data = (CvMat*)cvReadByName(storage, NULL, "sample", 0 );
    CvMat           train_data, test_data;
    CvMat           response;
    CvMat*          missed = NULL;
    CvMat*          comp_idx = NULL;
    CvMat*          sample_idx = NULL;
    CvMat*          type_mask = NULL;
    int             resp_col = 0;
    int             i,j;
    CvRTreesParams  params;
    CvTreeClassifierTrainParams cart_params;
    const int       ntrain_samples = 1000;
    const int       ntest_samples  = 1000;
    const int       nvars = 23;

    if(data == NULL || data->cols != nvars)
    {
        puts("Error in source data");
```

```
    return -1;
}

cvGetSubRect( data, &train_data, cvRect(0, 0, nvars, ntrain_samples) );
cvGetSubRect( data, &test_data, cvRect(0, ntrain_samples, nvars,
    ntrain_samples + ntest_samples) );

resp_col = 0;
cvGetCol( &train_data, &response, resp_col);

/* create missed variable matrix */
missed = cvCreateMat(train_data.rows, train_data.cols, CV_8UC1);
for( i = 0; i < train_data.rows; i++ )
    for( j = 0; j < train_data.cols; j++ )
        CV_MAT_ELEM(*missed,uchar,i,j)
      = (uchar)(CV_MAT_ELEM(train_data,float,i,j) < 0);

/* create comp_idx vector */
comp_idx = cvCreateMat(1, train_data.cols-1, CV_32SC1);
for( i = 0; i < train_data.cols; i++ )
{
    if(i<resp_col)CV_MAT_ELEM(*comp_idx,int,0,i) = i;
    if(i>resp_col)CV_MAT_ELEM(*comp_idx,int,0,i-1) = i;
}

/* create sample_idx vector */
sample_idx = cvCreateMat(1, train_data.rows, CV_32SC1);
for( j = i = 0; i < train_data.rows; i++ )
{
    if(CV_MAT_ELEM(response,float,i,0) < 0) continue;
    CV_MAT_ELEM(*sample_idx,int,0,j) = i;
    j++;
}
sample_idx->cols = j;

/* create type mask */
type_mask = cvCreateMat(1, train_data.cols+1, CV_8UC1);
cvSet( type_mask, cvRealScalar(CV_VAR_CATEGORICAL), 0);

// initialize training parameters
cvSetDefaultParamTreeClassifier((CvStatModelParams*)&cart_params);
cart_params.wrong_feature_as_unknown = 1;
params.tree_params = &cart_params;
params.term_crit.max_iter = 50;
params.term_crit.epsilon = 0.1;
```

```
    params.term_crit.type = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

    puts("Random forest results");
    cls = cvCreateRTreesClassifier( &train_data,
                                    CV_ROW_SAMPLE,
                                    &response,
                                    (CvStatModelParams*)&
                                    params,
                                    comp_idx,
                                    sample_idx,
                                    type_mask,
                                    missed );
    if( cls )
    {
        CvMat sample = cvMat( 1, nvars, CV_32FC1, test_data.data.fl );
        CvMat test_resp;
        int wrong = 0, total = 0;
        cvGetCol( &test_data, &test_resp, resp_col);
        for( i = 0; i < ntest_samples; i++, sample.data.fl += nvars )
        {
            if( CV_MAT_ELEM(test_resp,float,i,0) >= 0 )
            {
                float resp = cls->predict( cls, &sample, NULL );
                wrong += (fabs(resp-response.data.fl[i]) > 1e-3 ) ? 1 : 0;
                total++;
            }
        }
        printf( "Test set error = %.2f\n", wrong*100.f/(float)total );
    }
    else
       puts("Error forest creation");

    cvReleaseMat(&missed);
    cvReleaseMat(&sample_idx);
    cvReleaseMat(&comp_idx);
    cvReleaseMat(&type_mask);
    cvReleaseMat(&data);
    cvReleaseStatModel(&cls);
    cvReleaseFileStorage(&storage);
    return 0;
}
```

19.8 Expectation-Maximization

The EM (Expectation-Maximization) algorithm estimates the parameters of the multivariate probability density function in the form of a Gaussian mixture distribution with a specified number of mixtures.

Consider the set of the feature vectors $x_1, x_2, ..., x_N$: N vectors from a d-dimensional Euclidean space drawn from a Gaussian mixture:

$$p(x; a_k, S_k, \pi_k) = \sum_{k=1}^{m} \pi_k p_k(x), \quad \pi_k \geq 0, \quad \sum_{k=1}^{m} \pi_k = 1,$$

$$p_k(x) = \varphi(x; a_k, S_k) = \frac{1}{(2\pi)^{d/2} \mid S_k \mid^{1/2}} exp\left\{-\frac{1}{2}(x - a_k)^T S_k^{-1}(x - a_k)\right\},$$

where m is the number of mixtures, p_k is the normal distribution density with the mean a_k and covariance matrix S_k, π_k is the weight of the k-th mixture. Given the number of mixtures M and the samples x_i, $i = 1..N$ the algorithm finds the maximum-likelihood estimates (MLE) of the all the mixture parameters, i.e. a_k, S_k and π_k :

$$L(x, \theta) = logp(x, \theta) = \sum_{i=1}^{N} log\left(\sum_{k=1}^{m} \pi_k p_k(x)\right) \rightarrow \max_{\theta \in \Theta},$$

$$\Theta = \left\{(a_k, S_k, \pi_k) : a_k \in \mathbb{R}^d, S_k = S_k^T > 0, S_k \in \mathbb{R}^{d \times d}, \pi_k \geq 0, \sum_{k=1}^{m} \pi_k = 1\right\}.$$

EM algorithm is an iterative procedure. Each iteration of it includes two steps. At the first step (Expectation-step, or E-step), we find a probability $p_{i,k}$ (denoted $\alpha_{i,k}$ in the formula below) of sample `i` to belong to mixture `k` using the currently available mixture parameter estimates:

$$\alpha_{ki} = \frac{\pi_k \varphi(x; a_k, S_k)}{\sum\limits_{j=1}^{m} \pi_j \varphi(x; a_j, S_j)}.$$

At the second step (Maximization-step, or M-step) the mixture parameter estimates are refined using the computed probabilities:

$$\pi_k = \frac{1}{N} \sum_{i=1}^{N} \alpha_{ki}, \quad a_k = \frac{\sum\limits_{i=1}^{N} \alpha_{ki} x_i}{\sum\limits_{i=1}^{N} \alpha_{ki}}, \quad S_k = \frac{\sum\limits_{i=1}^{N} \alpha_{ki}(x_i - a_k)(x_i - a_k)^T}{\sum\limits_{i=1}^{N} \alpha_{ki}},$$

Alternatively, the algorithm may start with the M-step when the initial values for $p_{i,k}$ can be provided. Another alternative when $p_{i,k}$ are unknown, is to use a simpler clustering algorithm to pre-cluster the input samples and thus obtain initial $p_{i,k}$. Often (and in ML) the KMeans2 algorithm is used for that purpose.

One of the main that EM algorithm should deal with is the large number of parameters to estimate. The majority of the parameters sits in covariance matrices, which are $d \times d$ elements each (where d is the feature space dimensionality). However, in many practical problems the covariance matrices are close to diagonal, or even to $\mu_k * I$, where I is identity matrix and μ_k is mixture-dependent "scale" parameter. So a robust computation scheme could be to start with the harder constraints on the covariance matrices and then use the estimated parameters as an input for a less constrained optimization problem (often a diagonal covariance matrix is already a good enough approximation).

References:

- Bilmes98 J. A. Bilmes. A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models. Technical Report TR-97-021, International Computer Science Institute and Computer Science Division, University of California at Berkeley, April 1998.

---

cv::CvEMParams (view/add comments)

Parameters of the EM algorithm.

```
struct CvEMParams
{
    CvEMParams() : nclusters(10), cov_mat_type(CvEM::COV_MAT_DIAGONAL),
        start_step(CvEM::START_AUTO_STEP), probs(0), weights(0), means(0),
                                                   covs(0)
    {
        term_crit=cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
                                              100, FLT_EPSILON );
    }

    CvEMParams( int _nclusters, int _cov_mat_type=1/*CvEM::COV_MAT_DIAGONAL*/,
                int _start_step=0/*CvEM::START_AUTO_STEP*/,
                CvTermCriteria _term_crit=cvTermCriteria(
                                  CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
                                  100, FLT_EPSILON),
                CvMat* _probs=0, CvMat* _weights=0,
                CvMat* _means=0, CvMat** _covs=0 ) :
                nclusters(_nclusters), cov_mat_type(_cov_mat_type),
                start_step(_start_step),
                probs(_probs), weights(_weights), means(_means), covs(_covs),
```

```
                term_crit(_term_crit)
    {}

    int nclusters;
    int cov_mat_type;
    int start_step;
    const CvMat* probs;
    const CvMat* weights;
    const CvMat* means;
    const CvMat** covs;
    CvTermCriteria term_crit;
};
```

The structure has 2 constructors, the default one represents a rough rule-of-thumb, with another one it is possible to override a variety of parameters, from a single number of mixtures (the only essential problem-dependent parameter), to the initial values for the mixture parameters.

---

cv::CvEM (view/add comments)

EM model.

```
class CV_EXPORTS CvEM : public CvStatModel
{
public:
    // Type of covariance matrices
    enum { COV_MAT_SPHERICAL=0, COV_MAT_DIAGONAL=1, COV_MAT_GENERIC=2 };

    // The initial step
    enum { START_E_STEP=1, START_M_STEP=2, START_AUTO_STEP=0 };

    CvEM();
    CvEM( const CvMat* samples, const CvMat* sample_idx=0,
          CvEMParams params=CvEMParams(), CvMat* labels=0 );
    virtual ~CvEM();

    virtual bool train( const CvMat* samples, const CvMat* sample_idx=0,
                        CvEMParams params=CvEMParams(), CvMat* labels=0 );

    virtual float predict( const CvMat* sample, CvMat* probs ) const;
    virtual void clear();

    int get_nclusters() const { return params.nclusters; }
    const CvMat* get_means() const { return means; }
    const CvMat** get_covs() const { return covs; }
    const CvMat* get_weights() const { return weights; }
```

```
    const CvMat* get_probs() const { return probs; }

protected:

    virtual void set_params( const CvEMParams& params,
                             const CvVectors& train_data );
    virtual void init_em( const CvVectors& train_data );
    virtual double run_em( const CvVectors& train_data );
    virtual void init_auto( const CvVectors& samples );
    virtual void kmeans( const CvVectors& train_data, int nclusters,
                         CvMat* labels, CvTermCriteria criteria,
                         const CvMat* means );
    CvEMParams params;
    double log_likelihood;

    CvMat* means;
    CvMat** covs;
    CvMat* weights;
    CvMat* probs;

    CvMat* log_weight_div_det;
    CvMat* inv_eigen_values;
    CvMat** cov_rotate_mats;
};
```

CvEM::train (view/add comments)

Estimates the Gaussian mixture parameters from the sample set.

```
void CvEM::train(
            const CvMat* samples,
            const CvMat* sample_idx=0,
            CvEMParams params=CvEMParams(),
            CvMat* labels=0 );
```

Unlike many of the ML models, EM is an unsupervised learning algorithm and it does not take responses (class labels or the function values) on input. Instead, it computes the MLE of the Gaussian mixture parameters from the input sample set, stores all the parameters inside the structure: $p_{i,k}$ in `probs`, a_k in `means` S_k in `covs[k]`, π_k in `weights` and optionally computes

the output "class label" for each sample: $\texttt{labels}_i = \texttt{arg max}_k(p_{i,k}), i = 1..N$ (i.e. indices of the most-probable mixture for each sample).

The trained model can be used further for prediction, just like any other classifier. The model trained is similar to the Bayes classifier .

Example: Clustering random samples of multi-Gaussian distribution using EM

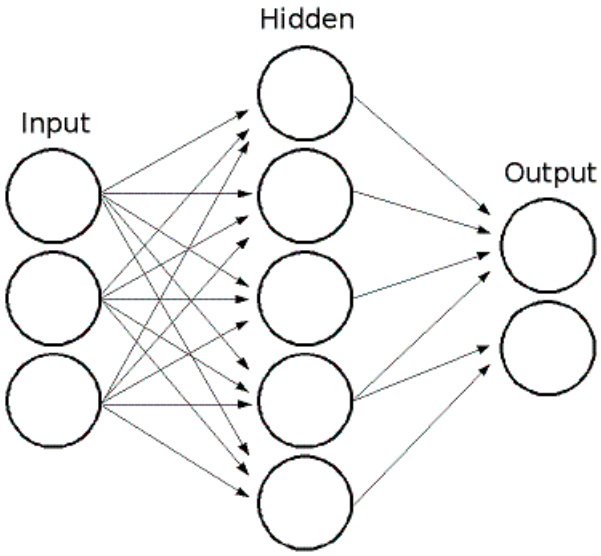
```
#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    const int N = 4;
    const int N1 = (int)sqrt((double)N);
    const CvScalar colors[] = {{0,0,255}},{{0,255,0}},
                                {{0,255,255}},{{255,255,0}
                                ;
    int i, j;
    int nsamples = 100;
    CvRNG rng_state = cvRNG(-1);
    CvMat* samples = cvCreateMat( nsamples, 2, CV_32FC1 );
    CvMat* labels = cvCreateMat( nsamples, 1, CV_32SC1 );
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    float _sample[2];
    CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
    CvEM em_model;
    CvEMParams params;
    CvMat samples_part;

    cvReshape( samples, samples, 2, 0 );
    for( i = 0; i < N; i++ )
    {
        CvScalar mean, sigma;

        // form the training samples
        cvGetRows( samples, &samples_part, i*nsamples/N,
                                            (i+1)*nsamples/N );
        mean = cvScalar(((i%N1)+1.)*img->width/(N1+1),
                        ((i/N1)+1.)*img->height/(N1+1));
        sigma = cvScalar(30,30);
        cvRandArr( &rng_state, &samples_part, CV_RAND_NORMAL,
                                                    mean, sigma );
    }
    cvReshape( samples, samples, 1, 0 );

    // initialize model's parameters
```

```
    params.covs      = NULL;
    params.means     = NULL;
    params.weights   = NULL;
    params.probs     = NULL;
    params.nclusters = N;
    params.cov_mat_type       = CvEM::COV_MAT_SPHERICAL;
    params.start_step         = CvEM::START_AUTO_STEP;
    params.term_crit.max_iter = 10;
    params.term_crit.epsilon  = 0.1;
    params.term_crit.type     = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

    // cluster the data
    em_model.train( samples, 0, params, labels );

#if 0
    // the piece of code shows how to repeatedly optimize the model
    // with less-constrained parameters
    //(COV_MAT_DIAGONAL instead of COV_MAT_SPHERICAL)
    // when the output of the first stage is used as input for the second.
    CvEM em_model2;
    params.cov_mat_type = CvEM::COV_MAT_DIAGONAL;
    params.start_step = CvEM::START_E_STEP;
    params.means = em_model.get_means();
    params.covs = (const CvMat**)em_model.get_covs();
    params.weights = em_model.get_weights();

    em_model2.train( samples, 0, params, labels );
    // to use em_model2, replace em_model.predict()
    // with em_model2.predict() below
#endif
    // classify every image pixel
    cvZero( img );
    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            CvPoint pt = cvPoint(j, i);
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;
            int response = cvRound(em_model.predict( &sample, NULL ));
            CvScalar c = colors[response];

            cvCircle( img, pt, 1, cvScalar(c.val[0]*0.75,
                c.val[1]*0.75,c.val[2]*0.75), CV_FILLED );
        }
```

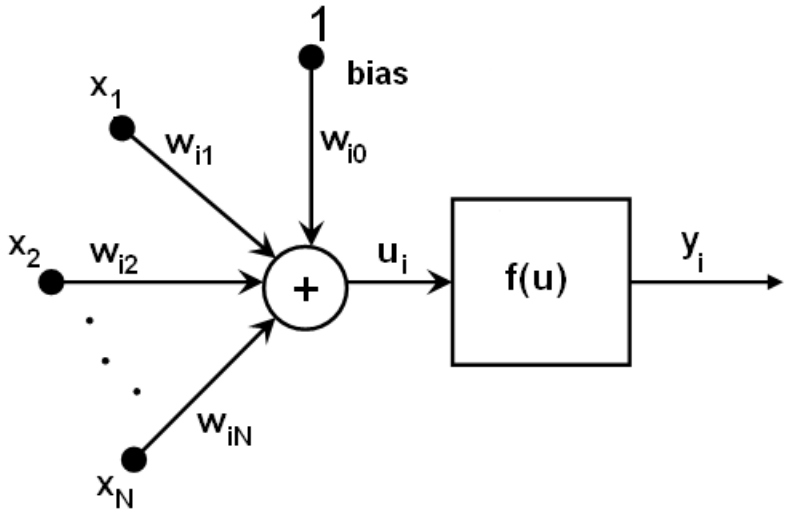
```
    }

    //draw the clustered samples
    for( i = 0; i < nsamples; i++ )
    {
        CvPoint pt;
        pt.x = cvRound(samples->data.fl[i*2]);
        pt.y = cvRound(samples->data.fl[i*2+1]);
        cvCircle( img, pt, 1, colors[labels->data.i[i]], CV_FILLED );
    }

    cvNamedWindow( "EM-clustering result", 1 );
    cvShowImage( "EM-clustering result", img );
    cvWaitKey(0);

    cvReleaseMat( &samples );
    cvReleaseMat( &labels );
    return 0;
}
```

19.9 Neural Networks

ML implements feed-forward artificial neural networks, more particularly, multi-layer perceptrons (MLP), the most commonly used type of neural networks. MLP consists of the input layer, output layer and one or more hidden layers. Each layer of MLP includes one or more neurons that are directionally linked with the neurons from the previous and the next layer. Here is an example of a 3-layer perceptron with 3 inputs, 2 outputs and the hidden layer including 5 neurons:



All the neurons in MLP are similar. Each of them has several input links (i.e. it takes the output values from several neurons in the previous layer on input) and several output links (i.e. it passes the response to several neurons in the next layer). The values retrieved from the previous layer are summed with certain weights, individual for each neuron, plus the bias term, and the sum is transformed using the activation function f that may be also different for different neurons. Here is the picture:



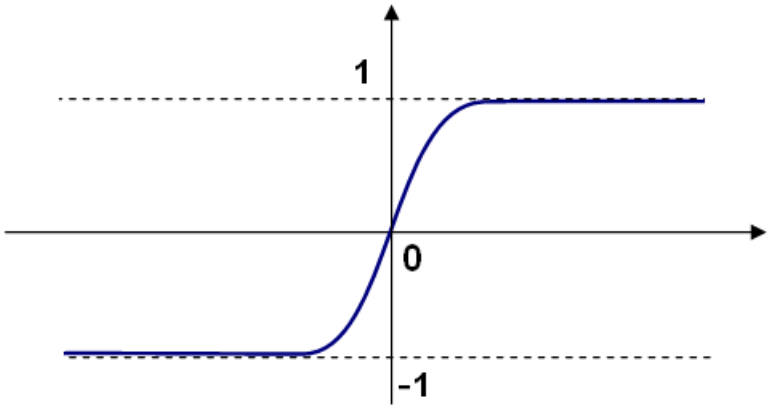
In other words, given the outputs x_j of the layer n, the outputs y_i of the layer $n+1$ are computed as:

$$u_i = \sum_j (w_{i,j}^{n+1} * x_j) + w_{i,bias}^{n+1}$$

$$y_i = f(u_i)$$

Different activation functions may be used, ML implements 3 standard ones:

- Identity function (`CvANN_MLP::IDENTITY`): $f(x) = x$
- Symmetrical sigmoid (`CvANN_MLP::SIGMOID_SYM`): $f(x) = \beta * (1 - e^{-\alpha x})/(1 + e^{-\alpha x})$, the default choice for MLP; the standard sigmoid with $\beta = 1, \alpha = 1$ is shown below:



- Gaussian function (`CvANN_MLP::GAUSSIAN`): $f(x) = \beta e^{-\alpha x * x}$, not completely supported by the moment.

In ML all the neurons have the same activation functions, with the same free parameters (α, β) that are specified by user and are not altered by the training algorithms.

So the whole trained network works as follows: It takes the feature vector on input, the vector size is equal to the size of the input layer, when the values are passed as input to the first hidden layer, the outputs of the hidden layer are computed using the weights and the activation functions and passed further downstream, until we compute the output layer.

So, in order to compute the network one needs to know all the weights $w_{i,j}^{n+1)}$. The weights are computed by the training algorithm. The algorithm takes a training set: multiple input vectors with the corresponding output vectors, and iteratively adjusts the weights to try to make the network give the desired response on the provided input vectors.

The larger the network size (the number of hidden layers and their sizes), the more is the potential network flexibility, and the error on the training set could be made arbitrarily small. But at the same time the learned network will also ”learn” the noise present in the training set, so the

error on the test set usually starts increasing after the network size reaches some limit. Besides, the larger networks are train much longer than the smaller ones, so it is reasonable to preprocess the data (using CalcPCA or similar technique) and train a smaller network on only the essential features.

Another feature of the MLP's is their inability to handle categorical data as is, however there is a workaround. If a certain feature in the input or output (i.e. in the case of `n`-class classifier for $n > 2$) layer is categorical and can take $M > 2$ different values, it makes sense to represent it as binary tuple of `M` elements, where `i`-th element is 1 if and only if the feature is equal to the `i`-th value out of `M` possible. It will increase the size of the input/output layer, but will speedup the training algorithm convergence and at the same time enable "fuzzy" values of such variables, i.e. a tuple of probabilities instead of a fixed value.

ML implements 2 algorithms for training MLP's. The first is the classical random sequential back-propagation algorithm and the second (default one) is batch RPROP algorithm.

References:

- http://en.wikipedia.org/wiki/Backpropagation. Wikipedia article about the back-propagation algorithm.
- Y. LeCun, L. Bottou, G.B. Orr and K.-R. Muller, "Efficient backprop", in Neural Networks—Tricks of the Trade, Springer Lecture Notes in Computer Sciences 1524, pp.5-50, 1998.
- M. Riedmiller and H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", Proc. ICNN, San Francisco (1993).

---

cv::CvANN_MLP_TrainParams (view/add comments)

Parameters of the MLP training algorithm.

```
struct CvANN_MLP_TrainParams
{
    CvANN_MLP_TrainParams();
    CvANN_MLP_TrainParams( CvTermCriteria term_crit, int train_method,
                           double param1, double param2=0 );
    ~CvANN_MLP_TrainParams();

    enum { BACKPROP=0, RPROP=1 };

    CvTermCriteria term_crit;
    int train_method;

    // backpropagation parameters
    double bp_dw_scale, bp_moment_scale;
```

```
    // rprop parameters
    double rp_dw0, rp_dw_plus, rp_dw_minus, rp_dw_min, rp_dw_max;
};
```

The structure has default constructor that initializes parameters for RPROP algorithm. There is also more advanced constructor to customize the parameters and/or choose backpropagation algorithm. Finally, the individual parameters can be adjusted after the structure is created.

---

cv::CvANN_MLP (view/add comments)

MLP model.

```
class CvANN_MLP : public CvStatModel
{
public:
    CvANN_MLP();
    CvANN_MLP( const CvMat* _layer_sizes,
               int _activ_func=SIGMOID_SYM,
               double _f_param1=0, double _f_param2=0 );

    virtual ~CvANN_MLP();

    virtual void create( const CvMat* _layer_sizes,
                         int _activ_func=SIGMOID_SYM,
                         double _f_param1=0, double _f_param2=0 );

    virtual int train( const CvMat* _inputs, const CvMat* _outputs,
                       const CvMat* _sample_weights,
                       const CvMat* _sample_idx=0,
                       CvANN_MLP_TrainParams _params = CvANN_MLP_TrainParams(),
                       int flags=0 );
    virtual float predict( const CvMat* _inputs,
                           CvMat* _outputs ) const;

    virtual void clear();

    // possible activation functions
    enum { IDENTITY = 0, SIGMOID_SYM = 1, GAUSSIAN = 2 };

    // available training flags
    enum { UPDATE_WEIGHTS = 1, NO_INPUT_SCALE = 2, NO_OUTPUT_SCALE = 4 };

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* storage, const char* name );
```

```
    int get_layer_count() { return layer_sizes ? layer_sizes->cols : 0; }
    const CvMat* get_layer_sizes() { return layer_sizes; }

protected:

    virtual bool prepare_to_train( const CvMat* _inputs, const CvMat* _outputs,
            const CvMat* _sample_weights, const CvMat* _sample_idx,
            CvANN_MLP_TrainParams _params,
            CvVectors* _ivecs, CvVectors* _ovecs, double** _sw, int _flags );

    // sequential random backpropagation
    virtual int train_backprop( CvVectors _ivecs, CvVectors _ovecs,
                                            const double* _sw );

    // RPROP algorithm
    virtual int train_rprop( CvVectors _ivecs, CvVectors _ovecs,
                                          const double* _sw );

    virtual void calc_activ_func( CvMat* xf, const double* bias ) const;
    virtual void calc_activ_func_deriv( CvMat* xf, CvMat* deriv,
                                          const double* bias ) const;
    virtual void set_activ_func( int _activ_func=SIGMOID_SYM,
                                 double _f_param1=0, double _f_param2=0 );
    virtual void init_weights();
    virtual void scale_input( const CvMat* _src, CvMat* _dst ) const;
    virtual void scale_output( const CvMat* _src, CvMat* _dst ) const;
    virtual void calc_input_scale( const CvVectors* vecs, int flags );
    virtual void calc_output_scale( const CvVectors* vecs, int flags );

    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvMat* layer_sizes;
    CvMat* wbuf;
    CvMat* sample_weights;
    double** weights;
    double f_param1, f_param2;
    double min_val, max_val, min_val1, max_val1;
    int activ_func;
    int max_count, max_buf_sz;
    CvANN_MLP_TrainParams params;
    CvRNG rng;
};
```

Unlike many other models in ML that are constructed and trained at once, in the MLP model these steps are separated. First, a network with the specified topology is created using the non-default constructor or the method `create`. All the weights are set to zeros. Then the network is trained using the set of input and output vectors. The training procedure can be repeated more than once, i.e. the weights can be adjusted based on the new training data.

CvANN_MLP::create (view/add comments)

Constructs the MLP with the specified topology

```
void CvANN_MLP::create(
            const CvMat* _layer_sizes,
            int _activ_func=SIGMOID_SYM,
            double _f_param1=0,
            double _f_param2=0 );
```

_layer_sizes The integer vector specifies the number of neurons in each layer including the input and output layers.

_activ_func Specifies the activation function for each neuron; one of `CvANN_MLP::IDENTITY`, `CvANN_MLP::SIGMOID_SYM` and `CvANN_MLP::GAUSSIAN`.

_f_param1, _f_param2 Free parameters of the activation function, α and β, respectively. See the formulas in the introduction section.

The method creates a MLP network with the specified topology and assigns the same activation function to all the neurons.

CvANN_MLP::train (view/add comments)

Trains/updates MLP.

```
int CvANN_MLP::train(
            const CvMat* _inputs,
            const CvMat* _outputs,
            const CvMat* _sample_weights,
```

```
            const CvMat* _sample_idx=0,
            CvANN_MLP_TrainParams _params = CvANN_MLP_TrainParams(),
            int flags=0 );
```

_inputs A floating-point matrix of input vectors, one vector per row.

_outputs A floating-point matrix of the corresponding output vectors, one vector per row.

_sample_weights (RPROP only) The optional floating-point vector of weights for each sample. Some samples may be more important than others for training, and the user may want to raise the weight of certain classes to find the right balance between hit-rate and false-alarm rate etc.

_sample_idx The optional integer vector indicating the samples (i.e. rows of `_inputs` and `_outputs`) that are taken into account.

_params The training params. See `CvANN_MLP_TrainParams` description.

_flags The various parameters to control the training algorithm. May be a combination of the following:

UPDATE_WEIGHTS = 1 algorithm updates the network weights, rather than computes them from scratch (in the latter case the weights are initialized using *Nguyen-Widrow* algorithm).

NO_INPUT_SCALE algorithm does not normalize the input vectors. If this flag is not set, the training algorithm normalizes each input feature independently, shifting its mean value to 0 and making the standard deviation =1. If the network is assumed to be updated frequently, the new training data could be much different from original one. In this case user should take care of proper normalization.

NO_OUTPUT_SCALE algorithm does not normalize the output vectors. If the flag is not set, the training algorithm normalizes each output features independently, by transforming it to the certain range depending on the activation function used.

This method applies the specified training algorithm to compute/adjust the network weights. It returns the number of done iterations.

Chapter 20

gpu. GPU-based Functionality

20.1 Initalization and Info

---

cv::gpu::hasNativeDoubleSupport (view/add comments)

Returns true if current GPU has native double support, false otherwise.

```
bool hasNativeDoubleSupport(int device);
```

device GPU identity.

---

cv::gpu::hasAtomicsSupport (view/add comments)

Returns true if current GPU has atomics support, false otherwise.

```
bool hasAtomicsSupport(int device);
```

device GPU identity.

20.2 Data Structures

cv::gpu::createContinuous (view/add comments)

Creates continuous matrix in GPU memory.

```
void createContinuous(int rows, int cols, int type, GpuMat& m);
```

rows Row count.

cols Column count.

type Type of the matrix.

m Destionation matrix. Will do only reshape if m has proper type and area ($rows \times cols$).

20.3 Operations on Matrices

cv::gpu::merge (view/add comments)

Makes multi-channel matrix out of several single-channel matrices.

```
void merge(const GpuMat* src, size_t n, GpuMat& dst);
void merge(const vector<GpuMat>& src, GpuMat& dst);
void merge(const GpuMat* src, size_t n, GpuMat& dst,
           const Stream& stream);
void merge(const vector<GpuMat>& src, GpuMat& dst,
           const Stream& stream);
```

src Vector or pointer to array of the source matrixes.

n Number of source matrices.

dst Destination matrix.

stream Stream for the asynchronous versions.

cv::gpu::split (view/add comments)

Copies each plane of a multi-channel matrix into an array.

```
void split(const GpuMat& src, GpuMat* dst);
void split(const GpuMat& src, vector<GpuMat>& dst);
void split(const GpuMat& src, GpuMat* dst, const Stream& stream);
void split(const GpuMat& src, vector<GpuMat>& dst,
           const Stream& stream);
```

src Source matrix.

dst Destination vector or pointer to array of single-channel matrixes.

stream Stream for the asynchronous versions.

cv::gpu::bitwise_not (view/add comments)

Performs per-element bitwise inversion.

```
void bitwise_not(const GpuMat& src, GpuMat& dst,
           const GpuMat& mask=GpuMat());
void bitwise_not(const GpuMat& src, GpuMat& dst,
           const GpuMat& mask, const Stream& stream);
```

src Source matrix.

dst Destination matrix. Will have the same size and type as `src`.

mask Optional operation mask. 8-bit single channel image.

stream Stream for the asynchronous version.

See also: cv::bitwise_not.

cv::gpu::bitwise_or (view/add comments)

Performs per-element bitwise disjunction of two matrixes.

```
void bitwise_or(const GpuMat& src1, const GpuMat& src2, GpuMat& dst,
           const GpuMat& mask=GpuMat());
void bitwise_or(const GpuMat& src1, const GpuMat& src2, GpuMat& dst,
           const GpuMat& mask, const Stream& stream);
```

src1 First source matrix.

src2 Second source matrix. It must have the same size and type as `src1`.

dst Destination matrix. Will have the same size and type as `src1`.

mask Optional operation mask. 8-bit single channel image.

stream Stream for the asynchronous version.

See also: cv::bitwise_or.

cv::gpu::bitwise_and (view/add comments)

Performs per-element bitwise conjunction of two matrixes.

```
void bitwise_and(const GpuMat& src1, const GpuMat& src2, GpuMat& dst,
           const GpuMat& mask=GpuMat());
void bitwise_and(const GpuMat& src1, const GpuMat& src2, GpuMat& dst,
           const GpuMat& mask, const Stream& stream);
```

src1 First source matrix.

src2 Second source matrix. It must have the same size and type as `src1`.

dst Destination matrix. Will have the same size and type as `src1`.

mask Optional operation mask. 8-bit single channel image.

stream Stream for the asynchronous version.

See also: cv::bitwise_and.

cv::gpu::bitwise_xor (view/add comments)

Performs per-element bitwise "exclusive or" of two matrixes.

```
void bitwise_xor(const GpuMat& src1, const GpuMat& src2, GpuMat& dst,
           const GpuMat& mask=GpuMat());
void bitwise_xor(const GpuMat& src1, const GpuMat& src2, GpuMat& dst,
           const GpuMat& mask, const Stream& stream);
```

src1 First source matrix.

src2 Second source matrix. It must have the same size and type as `src1`.

dst Destination matrix. Will have the same size and type as `src1`.

mask Optional operation mask. 8-bit single channel image.

stream Stream for the asynchronous version.

See also: cv::bitwise_xor.

20.4 Image Processing

cv::gpu::meanShiftFiltering (view/add comments)

Performs mean-shift filtering.

```
void meanShiftFiltering(const GpuMat& src, GpuMat& dst,
           int sp, int sr,
           TermCriteria criteria = TermCriteria(TermCriteria::MAX_ITER
           + TermCriteria::EPS, 5, 1));
```

src Source image. Only 8UC4 images are supported for now.

dst Destination image. Will have the same size and type as `src`. Each pixel `(x,y)` of the destination image will contain color of converged point started from `(x,y)` pixel of the source image.

sp Spatial window radius.

sr Color window radius.

criteria Termination criteria. See cv::TermCriteria.

---

cv::gpu::meanShiftProc (view/add comments)

Performs mean-shift procedure and stores information about converged points in two images..

```
void meanShiftProc(const GpuMat& src, GpuMat& dstr, GpuMat& dstsp,
           int sp, int sr,
           TermCriteria criteria = TermCriteria(TermCriteria::MAX_ITER
           + TermCriteria::EPS, 5, 1));
```

src Source image. Only 8UC4 images are supported for now.

dstr Destination image. Will have the same size and type as `src`. Each pixel `(x,y)` of the destination image will contain color of converged point started from `(x,y)` pixel of the source image.

dstsp 16SC2 matrix, which will contain coordinates of converged points and have the same size as `src`.

sp Spatial window radius.

sr Color window radius.

criteria Termination criteria. See cv::TermCriteria.

---

cv::gpu::meanShiftSegmentation (view/add comments)

Performs mean-shift segmentation of the source image and eleminates small segments.

```
void meanShiftSegmentation(const GpuMat& src, Mat& dst,
           int sp, int sr, int minsize,
           TermCriteria criteria = TermCriteria(TermCriteria::MAX_ITER
           + TermCriteria::EPS, 5, 1));
```

src Source image. Only 8UC4 images are supported for now.

dst Segmented image. Will have the same size and type as `src`.

sp Spatial window radius.

sr Color window radius.

minsize Minimum segment size. Smaller segements will be merged.

criteria Termination criteria. See cv::TermCriteria.

---

cv::gpu::integral (view/add comments)

Computes the integral image and squared integral image.

```
void integral(const GpuMat& src, GpuMat& sum);
void integral(const GpuMat& src, GpuMat& sum, GpuMat& sqsum);
```

src Source image. Only 8UC1 images are supported for now.

sum Integral image. Will contain 32-bit unsigned integer values packed into 32SC1.

sqsum Squared integral image. Will have 32FC1 type.

See also: cv::integral.

---

cv::gpu::sqrIntegral (view/add comments)

Computes squared integral image.

```
void sqrIntegral(const GpuMat& src, GpuMat& sqsum);
```

src Source image. Only 8UC1 images are supported for now.

sqsum Squared integral image. Will contain 64-bit floating point values packed into 64U.

cv::gpu::columnSum (view/add comments)

Computes vertical (column) sum.

```
void columnSum(const GpuMat& src, GpuMat& sum);
```

src Source image. Only 32FC1 images are supported for now.

sum Destination image. Will have 32FC1 type.

cv::gpu::cornerHarris (view/add comments)

Computes Harris cornerness criteria at each image pixel.

```
void cornerHarris(const GpuMat& src, GpuMat& dst,
           int blockSize, int ksize, double k,
           int borderType=BORDER_REFLECT101);
```

src Source image. Only 8UC1 and 32FC1 images are supported for now.

dst Destination image. Will have the same size and 32FC1 type and contain cornerness values.

blockSize Neighborhood size.

ksize Aperture parameter for the Sobel operator.

k Harris detector free parameter.

borderType Pixel extrapolation method. Only `BORDER_REFLECT101` and `BORDER_REPLICATE` are supported for now.

See also: cv::cornerHarris.

cv::gpu::cornerMinEigenVal (view/add comments)

Computes minimum eigen value of 2x2 derivative covariation matrix at each pixel - the cornerness criteria.

```
void cornerMinEigenVal(const GpuMat& src, GpuMat& dst,
           int blockSize, int ksize,
           int borderType=BORDER_REFLECT101);
```

src Source image. Only 8UC1 and 32FC1 images are supported for now.

dst Destination image. Will have the same size and 32FC1 type and contain cornerness values.

blockSize Neighborhood size.

ksize Aperture parameter for the Sobel operator.

k Harris detector free parameter.

borderType Pixel extrapolation method. Only `BORDER_REFLECT101` and `BORDER_REPLICATE` are supported for now.

See also: cv::cornerMinEigenValue.

cv::gpu::mulSpectrums (view/add comments)

Performs per-element multiplication of two Fourier spectrums.

```
void mulSpectrums(const GpuMat& a, const GpuMat& b,
           GpuMat& c, int flags, bool conjB=false);
```

a First spectrum.

b Second spectrum. Must have the same size and type as `a`.

c Destination spectrum.

flags Mock paramter is kept for CPU/GPU interfaces similarity.

conjB Optional flag indicates if the second spectrum must be conjugated before the multiplcation.

Only full (i.e. not packed) 32FC2 complex spectrums in the interleaved format are supported for now.

See also: cv::mulSpectrums.

---

cv::gpu::mulAndScaleSpectrums (view/add comments)

Performs per-element multiplication of two Fourier spectrums and scales the result.

```
void mulAndScaleSpectrums(const GpuMat& a, const GpuMat& b,
           GpuMat& c, int flags, float scale, bool conjB=false);
```

a First spectrum.

b Second spectrum. Must have the same size and type as `a`.

c Destination spectrum.

flags Mock paramter is kept for CPU/GPU interfaces similarity.

scale Scale constant.

conjB Optional flag indicates if the second spectrum must be conjugated before the multiplcation.

Only full (i.e. not packed) 32FC2 complex spectrums in the interleaved format are supported for now.

See also: cv::mulSpectrums.

---

cv::gpu::dft (view/add comments)

Performs a forward or inverse discrete Fourier transform (1D or 2D) of floating point matrix.

```
void dft(const GpuMat& src, GpuMat& dst, Size dft_size, int flags=0);
```

src Real of complex source matrix.

dst Real or complex destination matrix.

dft_size Size of discrete Fourier transform.

flags Optional flags:

DFT_ROWS Transform each individual row of the source matrix.

DFT_SCALE Scale the result: divide it by the number of elements in the transform (it's obtained from `dft_size`).

DFT_INVERSE Inverse DFT must be perfromed for complex-complex case (real-complex and complex-real cases are respectively forward and inverse always).

DFT_REAL_OUTPUT The source matrix is the result of real-complex transform and the destination matrix must be real.

The source matrix should be continuous, otherwise reallocation and data copying will be performed. Function chooses the operation mode depending on the flags, size and channel count of the source matrix:

- If the source matrix is complex and the output isn't specified as real then the destination matrix will be complex, will have `dft_size` size and 32FC2 type. It will contain full result of the DFT (forward or inverse).
- If the source matrix is complex and the output is specified as real then function assumes that its input is the result of the forward transform (see next item). The destionation matrix will have `dft_size` size and 32FC1 type. It will contain result of the inverse DFT.
- If the source matrix is real (i.e. its type is 32FC1) then forward DFT will be performed. The result of the DFT will be packed into complex (32FC2) matrix so its width will be `dft_size.width / 2 + 1`, but if the source is a single column then height will be reduced.

See also: cv::dft.

---

cv::gpu::convolve (view/add comments)

Computes convolution (or cross-correlation) of two images.

```
void convolve(const GpuMat& image, const GpuMat& templ, GpuMat& result,
           bool ccorr=false);
void convolve(const GpuMat& image, const GpuMat& templ, GpuMat& result,
           bool ccorr, ConvolveBuf& buf);
```

image Source image. Only 32FC1 images are supported for now.

templ Template image. Must have size not greater then `image` size and be the same type as `image`.

result Result image. Will have the same size and type as `image`.

ccorr Indicates that cross-correlation must be evaluated instead of convolution.

buf Optional buffer to decrease memory reallocation count (for many calls with the same sizes).

---

cv::gpu::ConvolveBuf (view/add comments)

Memory buffer for the cv::gpu::convolve function.

```
struct CV_EXPORTS ConvolveBuf
{
    ConvolveBuf() {}
    ConvolveBuf(Size image_size, Size templ_size)
        { create(image_size, templ_size); }
    void create(Size image_size, Size templ_size);

private:
    // Hidden
};
```

---

cv::gpu::ConvolveBuf::ConvolveBuf (view/add comments)

```
ConvolveBuf();
```

Construct empty buffer which will be properly resized after first call of the convolve function.

```
ConvolveBuf(Size image_size, Size templ_size);
```

Construct buffer for the convolve function with respectively arguments.

---

cv::gpu::matchTemplate (view/add comments)

Computes the proximity map for the raster template and the image where the template is searched for.

```
void matchTemplate(const GpuMat& image, const GpuMat& templ,
           GpuMat& result, int method);
```

image Source image. 32F and 8U images (1..4 channels) are supported for now.

templ Template image. Must have the same size and type as `image`.

result A map of comparison results (32FC1). If `image` is $W \times H$ and `templ` is $w \times h$ then `result` must be $(W - w + 1) \times (H - h + 1)$.

method Specifies the way the template must be compared with the image.

Following methods are supported for 8U images for now:

- CV_TM_SQDIFF
- CV_TM_SQDIFF_NORMED
- CV_TM_CCORR
- CV_TM_CCORR_NORMED
- CV_TM_CCOEFF
- CV_TM_CCOEFF_NORMED

Following methods are supported for 32F images for now:

- CV_TM_SQDIFF
- CV_TM_CCORR

See also: cv::matchTemplate.

20.5 Matrix Reductions

cv::gpu::sum (view/add comments)

Returns sum of array elements.

```
Scalar sum(const GpuMat& src);
Scalar sum(const GpuMat& src, GpuMat& buf);
```

src Source image of any depth excepting 64F, single-channel.

buf Optional buffer. It's resized automatically.

See also: cv::sum.

cv::gpu::sqrSum (view/add comments)

Returns squared sum of array elements.

```
Scalar sqrSum(const GpuMat& src);
Scalar sqrSum(const GpuMat& src, GpuMat& buf);
```

src Source image of any depth excepting 64F, single-channel.

buf Optional buffer. It's resized automatically.

cv::gpu::minMax (view/add comments)

Finds global minimum and maximum array elements and returns their values.

```
void minMax(const GpuMat& src, double* minVal,
            double* maxVal=0, const GpuMat& mask=GpuMat());
void minMax(const GpuMat& src, double* minVal, double* maxVal,
            const GpuMat& mask, GpuMat& buf);
```

src Single-channel source image.

minVal Pointer to returned minimum value. NULL if not required.

maxVal Pointer to returned maximum value. NULL if not required.

mask Optional mask to select a sub-array.

buf Optional buffer. It's resized automatically.

Function doesn't work with 64F images on GPU with compute capability $<$ 1.3.
See also: cv::minMaxLoc.

---

cv::gpu::minMaxLoc (view/add comments)

Finds global minimum and maximum array elements and returns their values with locations.

```
void minMaxLoc(const GpuMat& src, double minVal, double* maxVal=0,
           Point* minLoc=0, Point* maxLoc=0,
           const GpuMat& mask=GpuMat());
void minMaxLoc(const GpuMat& src, double* minVal, double* maxVal,
           Point* minLoc, Point* maxLoc, const GpuMat& mask,
           GpuMat& valbuf, GpuMat& locbuf);
```

src Single-channel source image.

minVal Pointer to returned minimum value. NULL if not required.

maxVal Pointer to returned maximum value. NULL if not required.

minValLoc Pointer to returned minimum location. NULL if not required.

maxValLoc Pointer to returned maximum location. NULL if not required.

mask Optional mask to select a sub-array.

valbuf Optional values buffer. It's resized automatically.

locbuf Optional location buffer. It's resized automatically.

Function doesn't work with 64F images on GPU with compute capability $<$ 1.3.
See also: cv::minMaxLoc.

cv::gpu::countNonZero (view/add comments)

Counts non-zero array elements.

```
int countNonZero(const GpuMat& src);
int countNonZero(const GpuMat& src, GpuMat& buf);
```

src Single-channel source image.

buf Optional buffer. It's resized automatically.

Function doesn't work with 64F images on GPU with compute capability $<$ 1.3.
See also: cv::countNonZero.

20.6 Object Detection

cv::gpu::HOGDescriptor (view/add comments)

Histogram of Oriented Gradients descriptor and detector.

```
struct CV_EXPORTS HOGDescriptor
{
    enum { DEFAULT_WIN_SIGMA = -1 };
    enum { DEFAULT_NLEVELS = 64 };
    enum { DESCR_FORMAT_ROW_BY_ROW, DESCR_FORMAT_COL_BY_COL };

    HOGDescriptor(Size win_size=Size(64, 128), Size block_size=Size(16, 16),
                  Size block_stride=Size(8, 8), Size cell_size=Size(8, 8),
                  int nbins=9, double win_sigma=DEFAULT_WIN_SIGMA,
                  double threshold_L2hys=0.2, bool gamma_correction=true,
                  int nlevels=DEFAULT_NLEVELS);

    size_t getDescriptorSize() const;
    size_t getBlockHistogramSize() const;

    void setSVMDetector(const vector<float>& detector);

    static vector<float> getDefaultPeopleDetector();
    static vector<float> getPeopleDetector48x96();
    static vector<float> getPeopleDetector64x128();
```

```
    void detect(const GpuMat& img, vector<Point>& found_locations,
                double hit_threshold=0, Size win_stride=Size(),
                Size padding=Size());

    void detectMultiScale(const GpuMat& img, vector<Rect>& found_locations,
                          double hit_threshold=0, Size win_stride=Size(),
                          Size padding=Size(), double scale0=1.05,
                          int group_threshold=2);

    void getDescriptors(const GpuMat& img, Size win_stride,
                        GpuMat& descriptors,
                        int descr_format=DESCR_FORMAT_COL_BY_COL);

    Size win_size;
    Size block_size;
    Size block_stride;
    Size cell_size;
    int nbins;
    double win_sigma;
    double threshold_L2hys;
    bool gamma_correction;
    int nlevels;

private:
    // Hidden
}
```

Interfaces of all methods are kept similar to CPU HOG descriptor and detector's analogues as much as possible.

---

cv::gpu::HOGDescriptor::HOGDescriptor (view/add comments)

Creates HOG descriptor and detector.

```
HOGDescriptor(Size win_size=Size(64, 128), Size block_size=Size(16, 16),
              Size block_stride=Size(8, 8), Size cell_size=Size(8, 8),
              int nbins=9, double win_sigma=DEFAULT_WIN_SIGMA,
              double threshold_L2hys=0.2, bool gamma_correction=true,
              int nlevels=DEFAULT_NLEVELS);
```

win_size Detection window size. Must be aligned to block size and block stride.

block_size Block size in cells. Only (2,2) is supported for now.

block_stride Block stride. Must be a multiple of cell size.

cell_size Cell size. Only (8, 8) is supported for now.

nbins Number of bins. Only 9 bins per cell is supported for now.

win_sigma Gaussian smoothing window parameter.

threshold_L2Hys L2-Hys normalization method shrinkage.

gamma_correction Do gamma correction preprocessing or not.

nlevels Maximum number of detection window increases.

---

cv::gpu::HOGDescriptor::getDescriptorSize (view/add comments)

Returns number of coefficients required for the classification.

```
size_t getDescriptorSize() const;
```

---

cv::gpu::HOGDescriptor::getBlockHistogramSize (view/add comments)

Returns block histogram size.

```
size_t getBlockHistogramSize() const;
```

---

cv::gpu::HOGDescriptor::setSVMDetector (view/add comments)

Sets coefficients for the linear SVM classifier.

```
void setSVMDetector(const vector<float>& detector);
```

cv::gpu::HOGDescriptor::getDefaultPeopleDetector (view/add comments)

Returns coefficients of the classifier trained for people detection (for default window size).

```
static vector<float> getDefaultPeopleDetector();
```

cv::gpu::HOGDescriptor::getPeopleDetector48x96 (view/add comments)

Returns coefficients of the classifier trained for people detection (for 48x96 windows).

```
static vector<float> getPeopleDetector48x96();
```

cv::gpu::HOGDescriptor::getPeopleDetector64x128 (view/add comments)

Returns coefficients of the classifier trained for people detection (for 64x128 windows).

```
static vector<float> getPeopleDetector64x128();
```

cv::gpu::HOGDescriptor::detect (view/add comments)

Perfroms object detection without increasing detection window.

```
void detect(const GpuMat& img, vector<Point>& found_locations,
            double hit_threshold=0, Size win_stride=Size(),
            Size padding=Size());
```

img Source image. 8UC1 and 8UC4 types are supported for now.

found_locations Will contain left-top corner points of detected objects boundaries.

hit_threshold Threshold for the distance between features and classifying plane. Usually it's 0, and should be specfied in the detector coefficients (as the last free coefficient), but if the free coefficient is missed (it's allowed) you can specify it manually here.

win_stride Window stride. Must be a multiple of block stride.

padding Mock parameter to keep CPU interface compatibility. Must be (0,0).

---

cv::gpu::HOGDescriptor::detectMultiScale (view/add comments)

Perfroms object detection with increasing detection window.

```
void detectMultiScale(const GpuMat& img, vector<Rect>& found_locations,
           double hit_threshold=0, Size win_stride=Size(),
           Size padding=Size(), double scale0=1.05,
           int group_threshold=2);
```

img Source image. See cv::gpu::HOGDescriptor::detect for type limitations.

found_locations Will contain detected objects boundaries.

hit_threshold The threshold for the distance between features and classifying plane. See cv::gpu::HOGDescriptor::detect for details.

win_stride Window stride. Must be a multiple of block stride.

padding Mock parameter to keep CPU interface compatibility. Must be (0,0).

scale0 Coefficient of the detection window increase.

group_threshold After detection some objects could be covered by many rectangles. This coefficient regulates similarity threshold. 0 means don't perform grouping.
See cv::groupRectangles.

---

cv::gpu::HOGDescriptor::getDescriptors (view/add comments)

Returns block descriptors computed for the whole image.

```
void getDescriptors(const GpuMat& img, Size win_stride,
           GpuMat& descriptors,
           int descr_format=DESCR_FORMAT_COL_BY_COL);
```

img Source image. See cv::gpu::HOGDescriptor::detect for type limitations.

win_stride Window stride. Must be a multiple of block stride.

descriptors 2D array of descriptors.

descr_format Descriptor storage format:

DESCR_FORMAT_ROW_BY_ROW Row-major order.

DESCR_FORMAT_COL_BY_COL Column-major order.

20.7 Feature detection and description

---

cv::gpu::SURFParams_GPU (view/add comments)

Various SURF algorithm parameters.

```
struct SURFParams_GPU
{
    SURFParams_GPU() :
        threshold(0.1f),
        nOctaves(4),
        nIntervals(4),
        initialScale(2.f),

        l1(3.f/1.5f),
        l2(5.f/1.5f),
        l3(3.f/1.5f),
        l4(1.f/1.5f),
        edgeScale(0.81f),
        initialStep(1),

        extended(true),

        featuresRatio(0.01f)
    {
```

```
    }

    //! The interest operator threshold
    float threshold;
    //! The number of octaves to process
    int nOctaves;
    //! The number of intervals in each octave
    int nIntervals;
    //! The scale associated with the first interval of the first octave
    float initialScale;

    //! mask parameter l_1
    float l1;
    //! mask parameter l_2
    float l2;
    //! mask parameter l_3
    float l3;
    //! mask parameter l_4
    float l4;
    //! The amount to scale the edge rejection mask
    float edgeScale;
    //! The initial sampling step in pixels.
    int initialStep;

    //! True, if generate 128-len descriptors, false - 64-len descriptors
    bool extended;

    //! max features = featuresRatio * img.size().srea()
    float featuresRatio;
};
```

---

cv::gpu::SURF_GPU (view/add comments)

Class for extracting Speeded Up Robust Features from an image.

```
class SURF_GPU : public SURFParams_GPU
{
public:
    //! returns the descriptor size in float's (64 or 128)
    int descriptorSize() const;

    //! upload host keypoints to device memory
    static void uploadKeypoints(const vector<KeyPoint>& keypoints,
        GpuMat& keypointsGPU);
```

```
    //! download keypoints from device to host memory
    static void downloadKeypoints(const GpuMat& keypointsGPU,
        vector<KeyPoint>& keypoints);

    //! download descriptors from device to host memory
    static void downloadDescriptors(const GpuMat& descriptorsGPU,
        vector<float>& descriptors);

    void operator()(const GpuMat& img, const GpuMat& mask,
        GpuMat& keypoints);

    void operator()(const GpuMat& img, const GpuMat& mask,
        GpuMat& keypoints, GpuMat& descriptors,
        bool useProvidedKeypoints = false,
        bool calcOrientation = true);

    void operator()(const GpuMat& img, const GpuMat& mask,
        std::vector<KeyPoint>& keypoints);

    void operator()(const GpuMat& img, const GpuMat& mask,
        std::vector<KeyPoint>& keypoints, GpuMat& descriptors,
        bool useProvidedKeypoints = false,
        bool calcOrientation = true);

    void operator()(const GpuMat& img, const GpuMat& mask,
        std::vector<KeyPoint>& keypoints,
        std::vector<float>& descriptors,
        bool useProvidedKeypoints = false,
        bool calcOrientation = true);

    GpuMat sum;
    GpuMat sumf;

    GpuMat mask1;
    GpuMat maskSum;

    GpuMat hessianBuffer;
    GpuMat maxPosBuffer;
    GpuMat featuresBuffer;
};
```

The class `SURF_GPU` implements Speeded Up Robust Features descriptor. There is fast multi-scale Hessian keypoint detector that can be used to find the keypoints (which is the default option), but the descriptors can be also computed for the user-specified keypoints. Supports only 8 bit grayscale images.

The class SURF_GPU can store results to GPU and CPU memory and provides static functions to convert results between CPU and GPU version (uploadKeypoints, downloadKeypoints, downloadDescriptors). CPU results has the same format as cv::SURF results. GPU results is stored to GpuMat. keypoints matrix is one row matrix with CV_32FC6 type. It contains 6 float values per feature: x, y, size, response, angle, octave. descriptors matrix is nFeatures x descriptorSize matrix with CV_32FC1 type.

The class SURF_GPU uses some buffers and provides access to it. All buffers can be safely released between function calls.

---

cv::gpu::BruteForceMatcher_GPU (view/add comments)

Brute-force descriptor matcher. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one. This descriptor matcher supports masking permissible matches between descriptor sets.

```
template<class Distance>
class BruteForceMatcher_GPU
{
public:
    // Add descriptors to train descriptor collection.
    void add(const std::vector<GpuMat>& descCollection);

    // Get train descriptors collection.
    const std::vector<GpuMat>& getTrainDescriptors() const;

    // Clear train descriptors collection.
    void clear();

    // Return true if there are not train descriptors in collection.
    bool empty() const;

    // Return true if the matcher supports mask in match methods.
    bool isMaskSupported() const;

    void matchSingle(const GpuMat& queryDescs, const GpuMat& trainDescs,
        GpuMat& trainIdx, GpuMat& distance,
        const GpuMat& mask = GpuMat());

    static void matchDownload(const GpuMat& trainIdx,
        const GpuMat& distance, std::vector<DMatch>& matches);

    void match(const GpuMat& queryDescs, const GpuMat& trainDescs,
        std::vector<DMatch>& matches, const GpuMat& mask = GpuMat());
```

```
    void makeGpuCollection(GpuMat& trainCollection, GpuMat& maskCollection,
        const vector<GpuMat>& masks = std::vector<GpuMat>());

    void matchCollection(const GpuMat& queryDescs,
        const GpuMat& trainCollection,
        GpuMat& trainIdx, GpuMat& imgIdx, GpuMat& distance,
        const GpuMat& maskCollection);

    static void matchDownload(const GpuMat& trainIdx, GpuMat& imgIdx,
        const GpuMat& distance, std::vector<DMatch>& matches);

    void match(const GpuMat& queryDescs, std::vector<DMatch>& matches,
        const std::vector<GpuMat>& masks = std::vector<GpuMat>());

    void knnMatch(const GpuMat& queryDescs, const GpuMat& trainDescs,
        GpuMat& trainIdx, GpuMat& distance, GpuMat& allDist, int k,
        const GpuMat& mask = GpuMat());

    static void knnMatchDownload(const GpuMat& trainIdx,
        const GpuMat& distance, std::vector< std::vector<DMatch> >& matches,
        bool compactResult = false);

    void knnMatch(const GpuMat& queryDescs, const GpuMat& trainDescs,
        std::vector< std::vector<DMatch> >& matches, int k,
        const GpuMat& mask = GpuMat(), bool compactResult = false);

    void knnMatch(const GpuMat& queryDescs,
        std::vector< std::vector<DMatch> >& matches, int knn,
        const std::vector<GpuMat>& masks = std::vector<GpuMat>(),
        bool compactResult = false );

    void radiusMatch(const GpuMat& queryDescs, const GpuMat& trainDescs,
        GpuMat& trainIdx, GpuMat& nMatches, GpuMat& distance,
        float maxDistance, const GpuMat& mask = GpuMat());

    static void radiusMatchDownload(const GpuMat& trainIdx,
        const GpuMat& nMatches, const GpuMat& distance,
        std::vector< std::vector<DMatch> >& matches,
        bool compactResult = false);

    void radiusMatch(const GpuMat& queryDescs, const GpuMat& trainDescs,
        std::vector< std::vector<DMatch> >& matches, float maxDistance,
        const GpuMat& mask = GpuMat(), bool compactResult = false);

    void radiusMatch(const GpuMat& queryDescs,
```

```
        std::vector< std::vector<DMatch> >& matches, float maxDistance,
        const std::vector<GpuMat>& masks = std::vector<GpuMat>(),
        bool compactResult = false);

private:
    std::vector<GpuMat> trainDescCollection;
};
```

The class `BruteForceMatcher_GPU` has the similar interface with class cv::DescriptorMatcher. It has two groups of match methods: for matching descriptors of one image with other image or with image set. Also all functions have alternative: save results to GPU memory or to CPU memory. `BruteForceMatcher_GPU` is templated on the distance metric as cv::BruteForceMatcher, but supports only `L1` and `L2` distance types.

---

gpu::BruteForceMatcher_GPU::match (view/add comments)

Find the best match for each descriptor from a query set with train descriptors. This function is equivalent of cv::DescriptorMatcher::match.

```
void match(const GpuMat& queryDescs,
           const GpuMat& trainDescs,
           std::vector<DMatch>& matches,
           const GpuMat& mask = GpuMat());
```

```
void match(const GpuMat& queryDescs,
           std::vector<DMatch>& matches,
           const std::vector<GpuMat>& masks = std::vector<GpuMat>());
```

---

gpu::BruteForceMatcher_GPU::matchSingle (view/add comments)

Find one best match for each query descriptor. Results stored to GPU memory.

```
void matchSingle(const GpuMat& queryDescs,
```

```
            const GpuMat& trainDescs,
            GpuMat& trainIdx,
            GpuMat& distance,
            const GpuMat& mask = GpuMat());
```

queryDescs Query set of descriptors.

trainDescs Train set of descriptors. This will not be added to train descriptors collection stored in class object.

trainIdx One row `CV_32SC1` matrix. Will contain best train index for each query. If some query descriptor masked out in `mask` it will contain -1.

distance One row `CV_32FC1` matrix. Will contain best distance for each query. If some query descriptor masked out in `mask` it will contain `FLT_MAX`.

mask Mask specifying permissible matches between input query and train matrices of descriptors.

---

gpu::BruteForceMatcher_GPU::matchCollection (view/add comments)

Find one best match for each query descriptor from train collection. Results stored to GPU memory.

```
void matchCollection(const GpuMat& queryDescs,
            const GpuMat& trainCollection,
            GpuMat& trainIdx,
            GpuMat& imgIdx,
            GpuMat& distance,
            const GpuMat& maskCollection);
```

queryDescs Query set of descriptors.

trainCollection `GpuMat` with train collection. It can be obtained from train descriptors collection that was set using `add` method by makeGpuCollection. Or it can contain user defined collection. It must be one row matrix, each element is a `DevMem2D` that points to one train descriptors matrix (matrix must have `CV_32FC1` type).

trainIdx One row `CV_32SC1` matrix. Will contain best train index for each query. If some query descriptor masked out in `mask` it will contain -1.

imgIdx One row `CV_32SC1` matrix. Will contain image train index for each query. If some query descriptor masked out in `mask` it will contain -1.

distance One row `CV_32FC1` matrix. Will contain best distance for each query. If some query descriptor masked out in `mask` it will contain `FLT_MAX`.

maskCollection `GpuMat` with set of masks. It can be obtained from `std::vector<GpuMat>` by makeGpuCollection. Or it can contain user defined mask set. It must be empty matrix or one row matrix, each element is a `PtrStep` that points to one mask (must have `CV_8UC1` type).

---

gpu::BruteForceMatcher_GPU::makeGpuCollection (view/add comments)

Make gpu collection of train descriptors and masks in suitable format for matchCollection function.

```
void makeGpuCollection(GpuMat& trainCollection,
           GpuMat& maskCollection,
           const vector<GpuMat>& masks = std::vector<GpuMat>());
```

---

gpu::BruteForceMatcher_GPU::matchDownload (view/add comments)

Download `trainIdx`, `imgIdx` and `distance` matrices obtained by matchSingle or matchCollection to CPU vector with cv::DMatch.

```
static void matchDownload(const GpuMat& trainIdx,
           const GpuMat& distance,
           std::vector<DMatch>& matches);
```

```
static void matchDownload(const GpuMat& trainIdx,
           GpuMat& imgIdx,
           const GpuMat& distance,
           std::vector<DMatch>& matches);
```

---

gpu::BruteForceMatcher_GPU::knnMatch (view/add comments)

Find the k best matches for each descriptor from a query set with train descriptors. Found k (or less if not possible) matches are returned in distance increasing order. This function is equivalent of cv::DescriptorMatcher::knnMatch.

```
void knnMatch(const GpuMat& queryDescs,
           const GpuMat& trainDescs,
           std::vector< std::vector<DMatch> >& matches,
           int k,
           const GpuMat& mask = GpuMat(),
           bool compactResult = false);
```

```
void knnMatch(const GpuMat& queryDescs,
           std::vector< std::vector<DMatch> >& matches,
           int k,
           const std::vector<GpuMat>& masks = std::vector<GpuMat>(),
           bool compactResult = false );
```

---

gpu::BruteForceMatcher_GPU::knnMatch (view/add comments)

Find the k best matches for each descriptor from a query set with train descriptors. Found k (or less if not possible) matches are returned in distance increasing order. Results stored to GPU memory.

```
void knnMatch(const GpuMat& queryDescs,
           const GpuMat& trainDescs,
           GpuMat& trainIdx,
           GpuMat& distance,
           GpuMat& allDist,
           int k,
           const GpuMat& mask = GpuMat());
```

queryDescs Query set of descriptors.

trainDescs Train set of descriptors. This will not be added to train descriptors collection stored in class object.

trainIdx Matrix `nQueries` x `k` with type `CV_32SC1`. `trainIdx.at<int>(queryIdx, i)` will contain index of i'th best trains. If some query descriptor masked out in `mask` it will contain -1.

distance Matrix `nQuery` x `k` with type `CV_32FC1`. Will contain distance for each query and i'th best trains. If some query descriptor masked out in `mask` it will contain `FLT_MAX`.

allDist Buffer to store all distances between query descriptors and train descriptors. It have size `nQuery` x `nTrain` and `CV_32F` type. `allDist.at<float>(queryIdx, trainIdx)` will contain `FLT_MAX`, if `trainIdx` is one from k best, otherwise it will contain distance between `queryIdx` and `trainIdx` descriptors.

k Count of best matches will be found per each query descriptor (or less if it's not possible).

mask Mask specifying permissible matches between input query and train matrices of descriptors.

---

gpu::BruteForceMatcher_GPU::knnMatchDownload (view/add comments)

Download `trainIdx` and `distance` matrices obtained by knnMatch to CPU vector with cv::DMatch. If `compactResult` is true `matches` vector will not contain matches for fully masked out query descriptors.

```
static void knnMatchDownload(const GpuMat& trainIdx,
           const GpuMat& distance,
           std::vector< std::vector<DMatch> >& matches,
           bool compactResult = false);
```

---

gpu::BruteForceMatcher_GPU::radiusMatch (view/add comments)

Find the best matches for each query descriptor which have distance less than given threshold. Found matches are returned in distance increasing order. This function is equivalent of cv::DescriptorMatcher::radiusMatch. Works only on device with Compute Capability >= 1.1.

```
void radiusMatch(const GpuMat& queryDescs,
           const GpuMat& trainDescs,
           std::vector< std::vector<DMatch> >& matches,
           float maxDistance,
           const GpuMat& mask = GpuMat(),
           bool compactResult = false);
```

```
void radiusMatch(const GpuMat& queryDescs,
           std::vector< std::vector<DMatch> >& matches,
           float maxDistance,
           const std::vector<GpuMat>& masks = std::vector<GpuMat>(),
           bool compactResult = false);
```

---

gpu::BruteForceMatcher_GPU::radiusMatch (view/add comments)

Find the best matches for each query descriptor which have distance less than given threshold. Results stored to GPU memory. Results are not sorted by distance increasing order. Works only on device with Compute Capability >= 1.1.

```
void radiusMatch(const GpuMat& queryDescs,
           const GpuMat& trainDescs,
           GpuMat& trainIdx,
           GpuMat& nMatches,
           GpuMat& distance,
           float maxDistance,
           const GpuMat& mask = GpuMat());
```

queryDescs Query set of descriptors.

trainDescs Train set of descriptors. This will not be added to train descriptors collection stored in class object.

trainIdx `trainIdx.at<int>(queryIdx, i)` will contain i'th train index `(i < min(nMatches.at<unsigned int>(0, queryIdx), trainIdx.cols)`. If `trainIdx` is empty, it will be created with size `nQuery` x `nTrain`. Or it can be allocated by user (it must have `nQuery` rows and `CV_32SC1` type). Cols can be less than `nTrain`, but it can be that matcher won't find all matches, because it haven't enough memory to store results.

nMatches `nMatches.at<unsigned int>(0, queryIdx)` will contain matches count for `queryIdx`. Carefully, `nMatches` can be greater than `trainIdx.cols` - it means that matcher didn't find all matches, because it didn't have enough memory.

distance `distance.at<int>(queryIdx, i)` will contain i'th distance `(i < min(nMatches.at<unsigned int>(0, queryIdx), trainIdx.cols)`. If `trainIdx` is empty, it will be created with size `nQuery` x `nTrain`. Otherwise it must be also allocated by user (it must have the same size as `trainIdx` and `CV_32FC1` type).

maxDistance The threshold to found match distances.

mask Mask specifying permissible matches between input query and train matrices of descriptors.

---

gpu::BruteForceMatcher_GPU::radiusMatchDownload (view/add comments)

Download `trainIdx`, `nMatches` and `distance` matrices obtained by radiusMatch to CPU vector with cv::DMatch. If `compactResult` is true `matches` vector will not contain matches for fully masked out query descriptors.

```
static void radiusMatchDownload(const GpuMat& trainIdx,
           const GpuMat& nMatches,
           const GpuMat& distance,
           std::vector< std::vector<DMatch> >& matches,
           bool compactResult = false);
```

20.8 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images.

See also: Image Filtering

---

cv::gpu::BaseRowFilter_GPU (view/add comments)

The base class for linear or non-linear filters that process rows of 2D arrays. Such filters are used for the "horizontal" filtering parts in separable filters.

```
class BaseRowFilter_GPU
{
public:
    BaseRowFilter_GPU(int ksize_, int anchor_);
    virtual ~BaseRowFilter_GPU() {}
    virtual void operator()(const GpuMat& src, GpuMat& dst) = 0;
    int ksize, anchor;
};
```

---

cv::gpu::BaseColumnFilter_GPU (view/add comments)

The base class for linear or non-linear filters that process columns of 2D arrays. Such filters are used for the "vertical" filtering parts in separable filters.

```
class BaseColumnFilter_GPU
{
public:
    BaseColumnFilter_GPU(int ksize_, int anchor_);
    virtual ~BaseColumnFilter_GPU() {}
    virtual void operator()(const GpuMat& src, GpuMat& dst) = 0;
    int ksize, anchor;
};
```

cv::gpu::BaseFilter_GPU (view/add comments)

The base class for non-separable 2D filters.

```
class CV_EXPORTS BaseFilter_GPU
{
public:
    BaseFilter_GPU(const Size& ksize_, const Point& anchor_);
    virtual ~BaseFilter_GPU() {}
    virtual void operator()(const GpuMat& src, GpuMat& dst) = 0;
    Size ksize;
    Point anchor;
};
```

cv::gpu::FilterEngine_GPU (view/add comments)

The base class for Filter Engine.

```
class CV_EXPORTS FilterEngine_GPU
{
public:
    virtual ~FilterEngine_GPU() {}

    virtual void apply(const GpuMat& src, GpuMat& dst,
                       Rect roi = Rect(0,0,-1,-1)) = 0;
};
```

The class can be used to apply an arbitrary filtering operation to an image. It contains all the necessary intermediate buffers. Pointers to the initialized `FilterEngine_GPU` instances are returned by various `create*Filter_GPU` functions, see below, and they are used inside high-level functions such as cv::gpu::filter2D, cv::gpu::erode, cv::gpu::Sobel etc.

By using `FilterEngine_GPU` instead functions you can avoid unnessesary memory allocation for intermediate buffers and get much better performance:

```
while (...)
{
    cv::gpu::GpuMat src = getImg();
    cv::gpu::GpuMat dst;
    // Allocate and release buffers at each iterations
    cv::gpu::GaussianBlur(src, dst, ksize, sigma1);
}

// Allocate buffers only once
cv::Ptr<cv::gpu::FilterEngine_GPU> filter =
    cv::gpu::createGaussianFilter_GPU(CV_8UC4, ksize, sigma1);
```

```
while (...)
{
    cv::gpu::GpuMat src = getImg();
    cv::gpu::GpuMat dst;
    filter->apply(src, dst, cv::Rect(0, 0, src.cols, src.rows));
}
// Release buffers only once
filter.release();
```

FilterEngine_GPU can process a rectangular sub-region of an image. By default, if roi == Rect(0,0,-1,-1), FilterEngine_GPU process inner region of image (Rect(anchor.x, anchor.y, src_size.width - ksize.width, src_size.height - ksize.height)), because some filters doesn't check indexies outside the image for better perfomace. Which filters supports processing the whole image and which not and image type limitations see below.

The GPU filters doesn't support the in-place mode.

See also: BaseRowFilter_GPU, BaseColumnFilter_GPU, BaseFilter_GPU, createFilter2D_GPU, createSeparableFilter_GPU, createBoxFilter_GPU, createMorphologyFilter_GPU, createLinearFilter_GPU, createSeparableLinearFilter_GPU, createDerivFilter_GPU, createGaussianFilter_GPU

---

gpu::createFilter2D_GPU (view/add comments)

Create the non-separable filter engine with the specified filter.

```
Ptr<FilterEngine_GPU> createFilter2D_GPU(
            const Ptr<BaseFilter_GPU>& filter2D,
            int srcType, int dstType);
```

Usually this function is used inside high-level functions, like createLinearFilter_GPU, createBoxFilter_GPU.

---

gpu::createSeparableFilter_GPU (view/add comments)

Create the separable filter engine with the specified filters.

```
Ptr<FilterEngine_GPU> createSeparableFilter_GPU(
            const Ptr<BaseRowFilter_GPU>& rowFilter,
            const Ptr<BaseColumnFilter_GPU>& columnFilter,
```

```
            int srcType, int bufType, int dstType);
```

Usually this function is used inside high-level functions, like createSeparableLinearFilter_GPU.

---

gpu::getRowSumFilter_GPU (view/add comments)

Create horizontal 1D box filter. Supports only `CV_8UC1` source type and `CV_32FC1` sum type.

```
Ptr<BaseRowFilter_GPU> getRowSumFilter_GPU(int srcType, int sumType,
            int ksize, int anchor = -1);
```

---

gpu::getColumnSumFilter_GPU (view/add comments)

Create vertical 1D box filter. Supports only `CV_8UC1` sum type and `CV_32FC1` dst type.

```
Ptr<BaseColumnFilter_GPU> getColumnSumFilter_GPU(int sumType,
            int dstType, int ksize, int anchor = -1);
```

---

gpu::createBoxFilter_GPU (view/add comments)

Create normalized 2D box filter. Supports `CV_8UC1` and `CV_8UC4` source type, dst type must be the same as source type. This filter doesn't check indexies outside the image.

```
Ptr<FilterEngine_GPU> createBoxFilter_GPU(int srcType, int dstType,
            const Size& ksize,
            const Point& anchor = Point(-1,-1));
```

```
Ptr<BaseFilter_GPU> getBoxFilter_GPU(int srcType, int dstType,
            const Size& ksize,
            Point anchor = Point(-1, -1));
```

---

cv::gpu::boxFilter (view/add comments)

Smooths the image using the normalized box filter. Supports `CV_8UC1` and `CV_8UC4` source type, dst type must be the same as source type.

```
void boxFilter(const GpuMat& src, GpuMat& dst, int ddepth, Size ksize,
            Point anchor = Point(-1,-1));
```

See cv::boxFilter, createBoxFilter_GPU.

---

cv::gpu::blur (view/add comments)

A synonym for normalized box filter.

```
void blur(const GpuMat& src, GpuMat& dst, Size ksize,
            Point anchor = Point(-1,-1));
```

See cv::blur, cv::gpu::boxFilter.

---

gpu::createMorphologyFilter_GPU (view/add comments)

Create 2D morphological filter. Supports `CV_8UC1` and `CV_8UC4` source type. This filter doesn't check indexies outside the image.

```
Ptr<FilterEngine_GPU> createMorphologyFilter_GPU(int op, int type,
           const Mat& kernel,
           const Point& anchor = Point(-1,-1),
           int iterations = 1);
```

```
Ptr<BaseFilter_GPU> getMorphologyFilter_GPU(int op, int type,
           const Mat& kernel, const Size& ksize,
           Point anchor=Point(-1,-1));
```

op The morphology operation id. Only `MORPH_ERODE` and `MORPH_DILATE` are supported.

type The input/output image type. Only `CV_8UC1` and `CV_8UC4` are supported.

kernel The 2D 8-bit structuring element for the morphological operation. It must be continuous matrix.

size The horizontal or vertical structuring element size for separable morphological operations

anchor The anchor position within the structuring element; negative values mean that the anchor is at the center

See cv::createMorphologyFilter.

---

cv::gpu::erode (view/add comments)

Erodes an image by using a specific structuring element. Supports `CV_8UC1` and `CV_8UC4` source type.

```
void erode(const GpuMat& src, GpuMat& dst, const Mat& kernel,
           Point anchor = Point(-1, -1),
           int iterations = 1);
```

See cv::erode, createMorphologyFilter_GPU.

cv::gpu::dilate (view/add comments)

Dilates an image by using a specific structuring element. Supports `CV_8UC1` and `CV_8UC4` source type.

```
void dilate(const GpuMat& src, GpuMat& dst, const Mat& kernel,
            Point anchor = Point(-1, -1),
            int iterations = 1);
```

See cv::dilate, createMorphologyFilter_GPU.

cv::gpu::morphologyEx (view/add comments)

Applies an advanced morphological operation to the image. Supports `CV_8UC1` and `CV_8UC4` source type.

```
void morphologyEx(const GpuMat& src, GpuMat& dst, int op,
            const Mat& kernel,
            Point anchor = Point(-1, -1),
            int iterations = 1);
```

See cv::morphologyEx.

gpu::createLinearFilter_GPU (view/add comments)

Create the non-separable linear filter. Supports `CV_8UC1` and `CV_8UC4` source type. This filter doesn't check indexies outside the image.

```
Ptr<FilterEngine_GPU> createLinearFilter_GPU(int srcType, int dstType,
            const Mat& kernel,
            const Point& anchor = Point(-1,-1));
```

```
Ptr<BaseFilter_GPU> getLinearFilter_GPU(int srcType, int dstType,
            const Mat& kernel, const Size& ksize,
            Point anchor = Point(-1, -1));
```

cv::gpu::filter2D (view/add comments)

Applies non-separable 2D linear filter to the image. Supports `CV_8UC1` and `CV_8UC4` source type.

```
void filter2D(const GpuMat& src, GpuMat& dst, int ddepth,
            const Mat& kernel,
            Point anchor=Point(-1,-1));
```

See cv::filter2D, createLinearFilter_GPU.

cv::gpu::Laplacian (view/add comments)

Applies Laplacian operator to the image. Supports `CV_8UC1` and `CV_8UC4` source type. Supports only `ksize` = 1 and `ksize` = 3.

```
void Laplacian(const GpuMat& src, GpuMat& dst, int ddepth,
            int ksize = 1, double scale = 1);
```

See cv::Laplacian, cv::gpu::filter2D.

gpu::getLinearRowFilter_GPU (view/add comments)

Create the primitive row filter with the specified kernel.

```
Ptr<BaseRowFilter_GPU> getLinearRowFilter_GPU(int srcType,
            int bufType, const Mat& rowKernel, int anchor = -1,
            int borderType = BORDER_CONSTANT);
```

Supports only CV_8UC1, CV_8UC4, CV_16SC1, CV_16SC2, CV_32SC1, CV_32FC1 source type. There are two version of algorithm: NPP and OpenCV. NPP calls when srcType == CV_8UC1 or srcType == CV_8UC4 and bufType == srcType, otherwise calls OpenCV version. NPP supports only BORDER_CONSTANT border type and doesn't check indexies outside image. OpenCV version supports only CV_32F as buffer depth and BORDER_REFLECT101, BORDER_REPLICATE and BORDER_CONSTANT border types and checks indexies outside image.

See also: getLinearColumnFilter_GPU, cv::createSeparableLinearFilter.

---

gpu::getLinearColumnFilter_GPU (view/add comments)

Create the primitive column filter with the specified kernel.

```
Ptr<BaseColumnFilter_GPU> getLinearColumnFilter_GPU(int bufType,
            int dstType, const Mat& columnKernel, int anchor = -1,
            int borderType = BORDER_CONSTANT);
```

Supports only CV_8UC1, CV_8UC4, CV_16SC1, CV_16SC2, CV_32SC1, CV_32FC1 dst type. There are two version of algorithm: NPP and OpenCV. NPP calls when dstType == CV_8UC1 or dstType == CV_8UC4 and bufType == dstType, otherwise calls OpenCV version. NPP supports only BORDER_CONSTANT border type and doesn't check indexies outside image. OpenCV version supports only CV_32F as buffer depth and BORDER_REFLECT101, BORDER_REPLICATE and BORDER_CONSTANT border types and checks indexies outside image.

See also: getLinearRowFilter_GPU, cv::createSeparableLinearFilter.

---

gpu::createSeparableLinearFilter_GPU (view/add comments)

Create the separable linear filter engine.

```
Ptr<FilterEngine_GPU> createSeparableLinearFilter_GPU(int srcType,
            int dstType, const Mat& rowKernel, const Mat& columnKernel,
            const Point& anchor = Point(-1,-1),
            int rowBorderType = BORDER_DEFAULT,
            int columnBorderType = -1);
```

See getLinearRowFilter_GPU, getLinearColumnFilter_GPU, cv::createSeparableLinearFilter.

---

cv::gpu::sepFilter2D (view/add comments)

Applies separable 2D linear filter to the image.

```
void sepFilter2D(const GpuMat& src, GpuMat& dst, int ddepth,
            const Mat& kernelX, const Mat& kernelY,
            Point anchor = Point(-1,-1),
            int rowBorderType = BORDER_DEFAULT,
            int columnBorderType = -1);
```

See createSeparableLinearFilter_GPU, cv::sepFilter2D.

---

gpu::createDerivFilter_GPU (view/add comments)

Create filter engine for the generalized Sobel operator.

```
Ptr<FilterEngine_GPU> createDerivFilter_GPU(int srcType, int dstType,
            int dx, int dy, int ksize,
            int rowBorderType = BORDER_DEFAULT,
            int columnBorderType = -1);
```

See createSeparableLinearFilter_GPU, cv::createDerivFilter.

cv::gpu::Sobel (view/add comments)

Applies generalized Sobel operator to the image.

```
void Sobel(const GpuMat& src, GpuMat& dst, int ddepth, int dx, int dy,
           int ksize = 3, double scale = 1,
           int rowBorderType = BORDER_DEFAULT,
           int columnBorderType = -1);
```

See createSeparableLinearFilter_GPU, cv::Sobel.

cv::gpu::Scharr (view/add comments)

Calculates the first x- or y- image derivative using Scharr operator.

```
void Scharr(const GpuMat& src, GpuMat& dst, int ddepth,
           int dx, int dy, double scale = 1,
           int rowBorderType = BORDER_DEFAULT,
           int columnBorderType = -1);
```

See createSeparableLinearFilter_GPU, cv::Scharr.

gpu::createGaussianFilter_GPU (view/add comments)

Create the Gaussian filter engine.

```
Ptr<FilterEngine_GPU> createGaussianFilter_GPU(int type, Size ksize,
           double sigma1, double sigma2 = 0,
           int rowBorderType = BORDER_DEFAULT,
           int columnBorderType = -1);
```

See createSeparableLinearFilter_GPU, cv::createGaussianFilter.

cv::gpu::GaussianBlur (view/add comments)

Smooths the image using Gaussian filter.

```
void GaussianBlur(const GpuMat& src, GpuMat& dst, Size ksize,
           double sigma1, double sigma2 = 0,
           int rowBorderType = BORDER_DEFAULT,
           int columnBorderType = -1);
```

See createGaussianFilter_GPU, cv::GaussianBlur.

gpu::getMaxFilter_GPU (view/add comments)

Create maximum filter. Supports only `CV_8UC1` and `CV_8UC4` source and dst type.

```
Ptr<BaseFilter_GPU> getMaxFilter_GPU(int srcType, int dstType,
           const Size& ksize, Point anchor = Point(-1,-1));
```

gpu::getMinFilter_GPU (view/add comments)

Create minimum filter. Supports only `CV_8UC1` and `CV_8UC4` source and dst type.

```
Ptr<BaseFilter_GPU> getMinFilter_GPU(int srcType, int dstType,
           const Size& ksize, Point anchor = Point(-1,-1));
```

Part III

Python API Reference

Chapter 21

Introduction

Starting with release 2.0, OpenCV has a new Python interface. This replaces the previous SWIG-based Python interface.

Some highlights of the new bindings:

- single import of all of OpenCV using `import cv`
- OpenCV functions no longer have the ”cv” prefix
- simple types like CvRect and CvScalar use Python tuples
- sharing of Image storage, so image transport between OpenCV and other systems (e.g. numpy and ROS) is very efficient
- complete documentation for the Python functions

This cookbook section contains a few illustrative examples of OpenCV Python code.

21.1 Cookbook

Here is a collection of code fragments demonstrating some features of the OpenCV Python bindings.

Convert an image

```
>>> import cv
>>> im = cv.LoadImageM("building.jpg")
>>> print type(im)
<type 'cv.cvmat'>
>>> cv.SaveImage("foo.png", im)
```

Resize an image

To resize an image in OpenCV, create a destination image of the appropriate size, then call Resize .

```
>>> import cv
>>> original = cv.LoadImageM("building.jpg")
>>> thumbnail = cv.CreateMat(original.rows / 10, original.cols / 10, cv.CV_8UC3)
>>> cv.Resize(original, thumbnail)
```

Compute the Laplacian

```
>>> import cv
>>> im = cv.LoadImageM("building.jpg", 1)
>>> dst = cv.CreateImage(cv.GetSize(im), cv.IPL_DEPTH_16S, 3)
>>> laplace = cv.Laplace(im, dst)
>>> cv.SaveImage("foo-laplace.png", dst)
```

Using GoodFeaturesToTrack

To find the 10 strongest corner features in an image, use GoodFeaturesToTrack like this:

```
>>> import cv
>>> img = cv.LoadImageM("building.jpg", cv.CV_LOAD_IMAGE_GRAYSCALE)
>>> eig_image = cv.CreateMat(img.rows, img.cols, cv.CV_32FC1)
>>> temp_image = cv.CreateMat(img.rows, img.cols, cv.CV_32FC1)
>>> for (x,y) in cv.GoodFeaturesToTrack(img, eig_image, temp_image, 10, 0.04, 1.0, useHarri
...     print "good feature at", x,y
good feature at 198.0 514.0
good feature at 791.0 260.0
good feature at 370.0 467.0
good feature at 374.0 469.0
good feature at 490.0 520.0
good feature at 262.0 278.0
good feature at 781.0 134.0
good feature at 3.0 247.0
good feature at 667.0 321.0
good feature at 764.0 304.0
```

Using GetSubRect

GetSubRect returns a rectangular part of another image. It does this without copying any data.

```
>>> import cv
>>> img = cv.LoadImageM("building.jpg")
>>> sub = cv.GetSubRect(img, (60, 70, 32, 32))  # sub is 32x32 patch within img
>>> cv.SetZero(sub)                             # clear sub to zero, which also clears 32x
```

Using CreateMat, and accessing an element

```
>>> import cv
>>> mat = cv.CreateMat(5, 5, cv.CV_32FC1)
>>> cv.Set(mat, 1.0)
>>> mat[3,1] += 0.375
>>> print mat[3,1]
1.375
>>> print [mat[3,i] for i in range(5)]
[1.0, 1.375, 1.0, 1.0, 1.0]
```

ROS image message to OpenCV

See this tutorial: Using CvBridge to convert between ROS images And OpenCV images.

PIL Image to OpenCV

(For details on PIL see the PIL handbook.)

```
>>> import Image, cv
>>> pi = Image.open('building.jpg')        # PIL image
>>> cv_im = cv.CreateImageHeader(pi.size, cv.IPL_DEPTH_8U, 3)
>>> cv.SetData(cv_im, pi.tostring())
>>> print pi.size, cv.GetSize(cv_im)
(868, 600) (868, 600)
>>> print pi.tostring() == cv_im.tostring()
True
```

OpenCV to PIL Image

```
>>> import Image, cv
>>> cv_im = cv.CreateImage((320,200), cv.IPL_DEPTH_8U, 1)
>>> pi = Image.fromstring("L", cv.GetSize(cv_im), cv_im.tostring())
>>> print pi.size
(320, 200)
```

NumPy and OpenCV

Using the array interface, to use an OpenCV CvMat in NumPy:

```
>>> import cv, numpy
>>> mat = cv.CreateMat(3, 5, cv.CV_32FC1)
>>> cv.Set(mat, 7)
>>> a = numpy.asarray(mat)
>>> print a
[[ 7.  7.  7.  7.  7.]
 [ 7.  7.  7.  7.  7.]
 [ 7.  7.  7.  7.  7.]]
```

and to use a NumPy array in OpenCV:

```
>>> import cv, numpy
>>> a = numpy.ones((480, 640))
>>> mat = cv.fromarray(a)
>>> print mat.rows
480
>>> print mat.cols
640
```

also, most OpenCV functions can work on NumPy arrays directly, for example:

```
>>> picture = numpy.ones((640, 480))
>>> cv.Smooth(picture, picture, cv.CV_GAUSSIAN, 15, 15)
```

Given a 2D array, the fromarray function (or the implicit version shown above) returns a single-channel CvMat of the same size. For a 3D array of size $j \times k \times l$, it returns a CvMat sized $j \times k$ with l channels.

Alternatively, use fromarray with the `allowND` option to always return a cvMatND .

OpenCV to pygame

To convert an OpenCV image to a pygame surface:

```
>>> import pygame.image, cv
>>> src = cv.LoadImage("lena.jpg")
>>> src_rgb = cv.CreateMat(src.height, src.width, cv.CV_8UC3)
>>> cv.CvtColor(src, src_rgb, cv.CV_BGR2RGB)
>>> pg_img = pygame.image.frombuffer(src_rgb.tostring(), cv.GetSize(src_rgb), "RGB")
>>> print pg_img
<Surface(512x512x24 SW)>
```

OpenCV and OpenEXR

Using OpenEXR's Python bindings you can make a simple image viewer:

```
import OpenEXR, Imath, cv
filename = "GoldenGate.exr"
exrimage = OpenEXR.InputFile(filename)

dw = exrimage.header()['dataWindow']
(width, height) = (dw.max.x - dw.min.x + 1, dw.max.y - dw.min.y + 1)

def fromstr(s):
    mat = cv.CreateMat(height, width, cv.CV_32FC1)
    cv.SetData(mat, s)
    return mat

pt = Imath.PixelType(Imath.PixelType.FLOAT)
(r, g, b) = [fromstr(s) for s in exrimage.channels("RGB", pt)]

bgr = cv.CreateMat(height, width, cv.CV_32FC3)
cv.Merge(b, g, r, None, bgr)

cv.ShowImage(filename, bgr)
cv.WaitKey()
```

Chapter 22

core. The Core Functionality

22.1 Basic Structures

CvPoint (view/add comments)

2D point with integer coordinates (usually zero-based).
2D point, represented as a tuple `(x, y)`, where x and y are integers.

CvPoint2D32f (view/add comments)

2D point with floating-point coordinates
2D point, represented as a tuple `(x, y)`, where x and y are floats.

CvPoint3D32f (view/add comments)

3D point with floating-point coordinates
3D point, represented as a tuple `(x, y, z)`, where x, y and z are floats.

CvPoint2D64f (view/add comments)

2D point with double precision floating-point coordinates
2D point, represented as a tuple `(x, y)`, where x and y are floats.

CvPoint3D64f (view/add comments)

3D point with double precision floating-point coordinates
3D point, represented as a tuple `(x, y, z)`, where x, y and z are floats.

CvSize (view/add comments)

Pixel-accurate size of a rectangle.

Size of a rectangle, represented as a tuple `(width, height)`, where width and height are integers.

CvSize2D32f (view/add comments)

Sub-pixel accurate size of a rectangle.

Size of a rectangle, represented as a tuple `(width, height)`, where width and height are floats.

CvRect (view/add comments)

Offset (usually the top-left corner) and size of a rectangle.

Rectangle, represented as a tuple `(x, y, width, height)`, where all are integers.

CvScalar (view/add comments)

A container for 1-,2-,3- or 4-tuples of doubles.

CvScalar is always represented as a 4-tuple.

```
>>> import cv
>>> cv.Scalar(1, 2, 3, 4)
(1.0, 2.0, 3.0, 4.0)
>>> cv.ScalarAll(7)
(7.0, 7.0, 7.0, 7.0)
>>> cv.RealScalar(7)
(7.0, 0.0, 0.0, 0.0)
>>> cv.RGB(17, 110, 255)
(255.0, 110.0, 17.0, 0.0)
```

CvTermCriteria (view/add comments)

Termination criteria for iterative algorithms.

Represented by a tuple `(type, max_iter, epsilon)`.

type `CV_TERMCRIT_ITER`, `CV_TERMCRIT_EPS` or `CV_TERMCRIT_ITER | CV_TERMCRIT_EPS`

max_iter Maximum number of iterations

epsilon Required accuracy

```
(cv.CV_TERMCRIT_ITER, 10, 0)                          # terminate after 10 iterations
(cv.CV_TERMCRIT_EPS, 0, 0.01)                         # terminate when epsilon reaches 0.01
(cv.CV_TERMCRIT_ITER | cv.CV_TERMCRIT_EPS, 10, 0.01) # terminate as soon as either conditi
```

CvMat (view/add comments)

A multi-channel 2D matrix. Created by CreateMat , LoadImageM , CreateMatHeader , fromarray .

type A CvMat signature containing the type of elements and flags, int

step Full row length in bytes, int

rows Number of rows, int

cols Number of columns, int

tostring() -> str Returns the contents of the CvMat as a single string.

CvMatND (view/add comments)

Multi-dimensional dense multi-channel array.

type A CvMatND signature combining the type of elements and flags, int

tostring() -> str Returns the contents of the CvMatND as a single string.

IplImage (view/add comments)

The IplImage object was inherited from the Intel Image Processing Library, in which the format is native. OpenCV only supports a subset of possible IplImage formats.

nChannels Number of channels, int.

width Image width in pixels

height Image height in pixels

depth Pixel depth in bits. The supported depths are:

IPL_DEPTH_8U Unsigned 8-bit integer

IPL_DEPTH_8S Signed 8-bit integer
IPL_DEPTH_16U Unsigned 16-bit integer
IPL_DEPTH_16S Signed 16-bit integer
IPL_DEPTH_32S Signed 32-bit integer
IPL_DEPTH_32F Single-precision floating point
IPL_DEPTH_64F Double-precision floating point

origin 0 - top-left origin, 1 - bottom-left origin (Windows bitmap style)

tostring() -> str Returns the contents of the CvMatND as a single string.

CvArr (view/add comments)

Arbitrary array

`CvArr` is used *only* as a function parameter to specify that the parameter can be:

- an IplImage
- a CvMat
- any other type that exports the array interface

22.2 Operations on Arrays

---

cv.AbsDiff (view/add comments)

Calculates absolute difference between two arrays.

```
AbsDiff(src1,src2,dst)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates absolute difference between two arrays.

$$\texttt{dst}(i)_c = |\texttt{src1}(I)_c - \texttt{src2}(I)_c|$$

All the arrays must have the same data type and the same size (or ROI size).

cv.AbsDiffS (view/add comments)

Calculates absolute difference between an array and a scalar.

```
AbsDiffS(src,value,dst)-> None
```

src The source array

dst The destination array

value The scalar

The function calculates absolute difference between an array and a scalar.

$$\texttt{dst}(i)_c = |\texttt{src}(I)_c - \texttt{value}_c|$$

All the arrays must have the same data type and the same size (or ROI size).

cv.Add (view/add comments)

Computes the per-element sum of two arrays.

```
Add(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds one array to another:

```
dst(I)=src1(I)+src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cv.AddS (view/add comments)

Computes the sum of an array and a scalar.

```
AddS(src,value,dst,mask=NULL)-> None
```

src The source array

value Added scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds a scalar `value` to every element in the source array `src1` and stores the result in `dst`. For types that have limited range this operation is saturating.

```
dst(I)=src(I)+value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size).

cv.AddWeighted (view/add comments)

Computes the weighted sum of two arrays.

```
AddWeighted(src1,alpha,src2,beta,gamma,dst)-> None
```

src1 The first source array

alpha Weight for the first array elements

src2 The second source array

beta Weight for the second array elements

dst The destination array

gamma Scalar, added to each sum

The function calculates the weighted sum of two arrays as follows:

```
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
```

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cv.And (view/add comments)

Calculates per-element bit-wise conjunction of two arrays.

```
And(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)&src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.AndS (view/add comments)

Calculates per-element bit-wise conjunction of an array and a scalar.

```
AndS(src,value,dst,mask=NULL)-> None
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)&value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.Avg (view/add comments)

Calculates average (mean) of array elements.

```
Avg(arr,mask=NULL)-> CvScalar
```

arr The array

mask The optional operation mask

The function calculates the average value `M` of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$M_c = \frac{\sum_{I,\,\texttt{mask}(I)\neq 0} \texttt{arr}(I)_c}{N}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average to the first scalar component S_0 .

---

cv.AvgSdv (view/add comments)

Calculates average (mean) of array elements.

```
AvgSdv(arr,mask=NULL)-> (mean, stdDev)
```

arr The array

mask The optional operation mask

mean Mean value, a CvScalar

stdDev Standard deviation, a CvScalar

The function calculates the average value and standard deviation of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$mean_c = \frac{1}{N} \sum_{I, \texttt{mask}(I) \neq 0} \texttt{arr}(I)_c$$
$$stdDev_c = \sqrt{\frac{1}{N} \sum_{I, \texttt{mask}(I) \neq 0} (\texttt{arr}(I)_c - mean_c)^2}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average and standard deviation to the first components of the output scalars ($mean_0$ and $stdDev_0$).

---

cv.CalcCovarMatrix (view/add comments)

Calculates covariance matrix of a set of vectors.

```
CalcCovarMatrix(vects,covMat,avg,flags)-> None
```

vects The input vectors, all of which must have the same type and the same size. The vectors do not have to be 1D, they can be 2D (e.g., images) and so forth

covMat The output covariance matrix that should be floating-point and square

avg The input or output (depending on the flags) array - the mean (average) vector of the input vectors

flags The operation flags, a combination of the following values

CV_COVAR_SCRAMBLED The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]$$

, that is, the covariance matrix is $\texttt{count} \times \texttt{count}$. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces

technique for face recognition). Eigenvalues of this ”scrambled” matrix will match the eigenvalues of the true covariance matrix and the ”true” eigenvectors can be easily calculated from the eigenvectors of the ”scrambled” covariance matrix.

CV_COVAR_NORMAL The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...] \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T$$

, that is, `covMat` will be a covariance matrix with the same linear size as the total number of elements in each input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

CV_COVAR_USE_AVG If the flag is specified, the function does not calculate `avg` from the input vectors, but, instead, uses the passed `avg` vector. This is useful if `avg` has been already calculated somehow, or if the covariance matrix is calculated by parts - in this case, `avg` is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.

CV_COVAR_SCALE If the flag is specified, the covariance matrix is scaled. In the ”normal” mode `scale` is ’1./count’; in the ”scrambled” mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled (’scale=1’).

CV_COVAR_ROWS Means that all the input vectors are stored as rows of a single matrix, `vects[0].count` is ignored in this case, and `avg` should be a single-row vector of an appropriate size.

CV_COVAR_COLS Means that all the input vectors are stored as columns of a single matrix, `vects[0].count` is ignored in this case, and `avg` should be a single-column vector of an appropriate size.

The function calculates the covariance matrix and, optionally, the mean vector of the set of input vectors. The function can be used for PCA, for comparing vectors using Mahalanobis distance and so forth.

---

cv.CartToPolar (view/add comments)

Calculates the magnitude and/or angle of 2d vectors.

```
CartToPolar(x,y,magnitude,angle=NULL,angleInDegrees=0)-> None
```

x The array of x-coordinates

y The array of y-coordinates

magnitude The destination array of magnitudes, may be set to NULL if it is not needed

angle The destination array of angles, may be set to NULL if it is not needed. The angles are measured in radians (0 to 2π) or in degrees (0 to 360 degrees).

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

```
magnitude(I)=sqrt(x(I)^2^+y(I)^2^ ),
angle(I)=atan(y(I)/x(I) )
```

The angles are calculated with 0.1 degree accuracy. For the (0,0) point, the angle is set to 0.

---

cv.Cbrt (view/add comments)

Calculates the cubic root

```
Cbrt(value)-> float
```

value The input floating-point value

The function calculates the cubic root of the argument, and normally it is faster than `pow(value,1./3)`. In addition, negative arguments are handled properly. Special values ($\pm\infty$, NaN) are not handled.

---

cv.ClearND (view/add comments)

Clears a specific array element.

```
ClearND(arr,idx)-> None
```

arr Input array

idx Array of the element indices

The function cv.ClearND clears (sets to zero) a specific element of a dense array or deletes the element of a sparse array. If the sparse array element does not exists, the function does nothing.

cv.CloneImage (view/add comments)

Makes a full copy of an image, including the header, data, and ROI.

```
CloneImage(image)-> copy
```

image The original image

The returned `IplImage*` points to the image copy.

cv.CloneMat (view/add comments)

Creates a full matrix copy.

```
CloneMat(mat)-> copy
```

mat Matrix to be copied

Creates a full copy of a matrix and returns a pointer to the copy.

cv.CloneMatND (view/add comments)

Creates full copy of a multi-dimensional array and returns a pointer to the copy.

```
CloneMatND(mat)-> copy
```

mat Input array

cv.Cmp (view/add comments)

Performs per-element comparison of two arrays.

```
Cmp(src1,src2,dst,cmpOp)-> None
```

src1 The first source array

src2 The second source array. Both source arrays must have a single channel.

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

- **CV_CMP_EQ** src1(I) ”equal to” value
- **CV_CMP_GT** src1(I) ”greater than” value
- **CV_CMP_GE** src1(I) ”greater or equal” value
- **CV_CMP_LT** src1(I) ”less than” value
- **CV_CMP_LE** src1(I) ”less or equal” value
- **CV_CMP_NE** src1(I) ”not equal” value

The function compares the corresponding elements of two arrays and fills the destination mask array:

```
dst(I)=src1(I) op src2(I),
```

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

---

cv.CmpS (view/add comments)

Performs per-element comparison of an array and a scalar.

```
CmpS(src,value,dst,cmpOp)-> None
```

src The source array, must have a single channel

value The scalar value to compare each array element with

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

CV_CMP_EQ src1(I) "equal to" value

CV_CMP_GT src1(I) "greater than" value

CV_CMP_GE src1(I) "greater or equal" value

CV_CMP_LT src1(I) "less than" value

CV_CMP_LE src1(I) "less or equal" value

CV_CMP_NE src1(I) "not equal" value

The function compares the corresponding elements of an array and a scalar and fills the destination mask array:

```
dst(I)=src(I) op scalar
```

where `op` is $=, >, \geq, <, \leq$ *or* $\neq$.

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same size (or ROI size).

---

cv.Convert (view/add comments)

Converts one array to another.

```
Convert(src,dst)-> None
```

src Source array

dst Destination array

The type of conversion is done with rounding and saturation, that is if the result of scaling + conversion can not be represented exactly by a value of the destination array element type, it is set to the nearest representable value on the real axis.

All the channels of multi-channel arrays are processed independently.

cv.ConvertScale (view/add comments)

Converts one array to another with optional linear transformation.

```
ConvertScale(src,dst,scale=1.0,shift=0.0)-> None
```

src Source array

dst Destination array

scale Scale factor

shift Value added to the scaled source array elements

The function has several different purposes, and thus has several different names. It copies one array to another with optional scaling, which is performed first, and/or optional type conversion, performed after:

$$\texttt{dst}(I) = \texttt{scale}\texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)$$

All the channels of multi-channel arrays are processed independently.

The type of conversion is done with rounding and saturation, that is if the result of scaling + conversion can not be represented exactly by a value of the destination array element type, it is set to the nearest representable value on the real axis.

In the case of `scale=1, shift=0` no prescaling is done. This is a specially optimized case and it has the appropriate cv.Convert name. If source and destination array types have equal types, this is also a special case that can be used to scale and shift a matrix or an image and that is caled cv.Scale.

cv.ConvertScaleAbs (view/add comments)

Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

```
ConvertScaleAbs(src,dst,scale=1.0,shift=0.0)-> None
```

src Source array

dst Destination array (should have 8u depth)

scale ScaleAbs factor

shift Value added to the scaled source array elements

The function is similar to cv.ConvertScale, but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scalesrc}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of cv.ConvertScale and cv.Abs functions.

---

cv.CvtScaleAbs (view/add comments)

Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

```
CvtScaleAbs(src,dst,scale=1.0,shift=0.0)-> None
```

src Source array

dst Destination array (should have 8u depth)

scale ScaleAbs factor

shift Value added to the scaled source array elements

The function is similar to cv.ConvertScale, but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scalesrc}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of cv.ConvertScale and cv.Abs functions.

---

cv.Copy (view/add comments)

Copies one array to another.

```
Copy(src,dst,mask=NULL)-> None
```

src The source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies selected elements from an input array to an output array:

$$\texttt{dst}(I) = \texttt{src}(I) \quad \text{if} \quad \texttt{mask}(I) \neq 0.$$

If any of the passed arrays is of `IplImage` type, then its ROI and COI fields are used. Both arrays must have the same type, the same number of dimensions, and the same size. The function can also copy sparse arrays (mask is not supported in this case).

---

cv.CountNonZero (view/add comments)

Counts non-zero array elements.

```
CountNonZero(arr)-> int
```

arr The array must be a single-channel array or a multi-channel image with COI set

The function returns the number of non-zero elements in arr:

$$\sum_I (\texttt{arr}(I) \neq 0)$$

In the case of `IplImage` both ROI and COI are supported.

---

cv.CreateData (view/add comments)

Allocates array data

```
CreateData(arr) -> None
```

arr Array header

The function allocates image, matrix or multi-dimensional array data. Note that in the case of matrix types OpenCV allocation functions are used and in the case of IplImage they are used unless `CV_TURN_ON_IPL_COMPATIBILITY` was called. In the latter case IPL functions are used to allocate the data.

---

cv.CreateImage (view/add comments)

Creates an image header and allocates the image data.

```
CreateImage(size, depth, channels)->image
```

size Image width and height

depth Bit depth of image elements. See IplImage for valid depths.

channels Number of channels per pixel. See IplImage for details. This function only creates images with interleaved channels.

---

cv.CreateImageHeader (view/add comments)

Creates an image header but does not allocate the image data.

```
CreateImageHeader(size, depth, channels) -> image
```

size Image width and height

depth Image depth (see cv.CreateImage)

channels Number of channels (see cv.CreateImage)

cv.CreateMat (view/add comments)

Creates a matrix header and allocates the matrix data.

```
CreateMat(rows, cols, type) -> mat
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type The type of the matrix elements in the form `CV_<bit depth><S|U|F>C<number of channels>`, where S=signed, U=unsigned, F=float. For example, CV_8UC1 means the elements are 8-bit unsigned and the there is 1 channel, and CV_32SC2 means the elements are 32-bit signed and there are 2 channels.

cv.CreateMatHeader (view/add comments)

Creates a matrix header but does not allocate the matrix data.

```
CreateMatHeader(rows, cols, type) -> mat
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements, see cv.CreateMat

The function allocates a new matrix header and returns a pointer to it. The matrix data can then be allocated using cv.CreateData or set explicitly to user-allocated data via cv.SetData.

cv.CreateMatND (view/add comments)

Creates the header and allocates the data for a multi-dimensional dense array.

```
CreateMatND(dims, type) -> None
```

dims List or tuple of array dimensions, up to 32 in length.

type Type of array elements, see cv.CreateMat.

This is a short form for:

---

cv.CreateMatNDHeader (view/add comments)

Creates a new matrix header but does not allocate the matrix data.

```
CreateMatNDHeader(dims, type) -> None
```

dims List or tuple of array dimensions, up to 32 in length.

type Type of array elements, see cv.CreateMat

The function allocates a header for a multi-dimensional dense array. The array data can further be allocated using cv.CreateData or set explicitly to user-allocated data via cv.SetData.

---

cv.CrossProduct (view/add comments)

Calculates the cross product of two 3D vectors.

```
CrossProduct(src1,src2,dst)-> None
```

src1 The first source vector

src2 The second source vector

dst The destination vector

The function calculates the cross product of two 3D vectors:

$$\texttt{dst} = \texttt{src1} \times \texttt{src2}$$

or:

$$\begin{aligned}\texttt{dst}_1 &= \texttt{src1}_2\texttt{src2}_3 - \texttt{src1}_3\texttt{src2}_2 \\ \texttt{dst}_2 &= \texttt{src1}_3\texttt{src2}_1 - \texttt{src1}_1\texttt{src2}_3 \\ \texttt{dst}_3 &= \texttt{src1}_1\texttt{src2}_2 - \texttt{src1}_2\texttt{src2}_1\end{aligned}$$

CvtPixToPlane

Synonym for Split .

---

cv.DCT (view/add comments)

Performs a forward or inverse Discrete Cosine transform of a 1D or 2D floating-point array.

```
DCT(src,dst,flags)-> None
```

src Source array, real 1D or 2D array

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

CV_DXT_FORWARD do a forward 1D or 2D transform.

CV_DXT_INVERSE do an inverse 1D or 2D transform.

CV_DXT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

The function performs a forward or inverse transform of a 1D or 2D floating-point array: Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C_{jk}^{(N)} = \sqrt{\alpha_j/N} \cos\left(\frac{\pi(2k+1)j}{2N}\right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.

Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)}\right)^{-1} \cdot Y = \left(C^{(N)}\right)^{T} \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)}\right)^{T} = I$)

Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)}\right)^{T}$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)}\right)^{T} \cdot X \cdot C^{(N)}$$

---

cv.DFT (view/add comments)

Performs a forward or inverse Discrete Fourier transform of a 1D or 2D floating-point array.

```
DFT(src,dst,flags,nonzeroRows=0)-> None
```

src Source array, real or complex

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

- **CV_DXT_FORWARD** do a forward 1D or 2D transform. The result is not scaled.
- **CV_DXT_INVERSE** do an inverse 1D or 2D transform. The result is not scaled. CV_DXT_FORWARD and CV_DXT_INVERSE are mutually exclusive, of course.
- **CV_DXT_SCALE** scale the result: divide it by the number of array elements. Usually, it is combined with CV_DXT_INVERSE, and one may use a shortcut CV_DXT_INV_SCALE.
- **CV_DXT_ROWS** do a forward or inverse transform of every individual row of the input matrix. This flag allows the user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.
- **CV_DXT_INVERSE_SCALE** same as CV_DXT_INVERSE + CV_DXT_SCALE

nonzeroRows Number of nonzero rows in the source array (in the case of a forward 2d transform), or a number of rows of interest in the destination array (in the case of an inverse 2d transform). If the value is negative, zero, or greater than the total number of rows, it is ignored. The parameter can be used to speed up 2d convolution/correlation when computing via DFT. See the example below.

The function performs a forward or inverse transform of a 1D or 2D floating-point array:
Forward Fourier transform of 1D vector of N elements:

$$y = F^{(N)} \cdot x, where F^{(N)}_{jk} = exp(-i \cdot 2\pi \cdot j \cdot k/N)$$

,

$$i = sqrt(-1)$$

Inverse Fourier transform of 1D vector of N elements:

$$x' = (F^{(N)})^{-1} \cdot y = conj(F^(N)) \cdot yx = (1/N) \cdot x$$

Forward Fourier transform of 2D vector of M $\times$ N elements:

$$Y = F^{(M)} \cdot X \cdot F^{(N)}$$

Inverse Fourier transform of 2D vector of M $\times$ N elements:

$$X' = conj(F^{(M)}) \cdot Y \cdot conj(F^{(N)})X = (1/(M \cdot N)) \cdot X'$$

In the case of real (single-channel) data, the packed format, borrowed from IPL, is used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix} ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\ ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\ ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\ \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots \\ ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \hdots & \hdots & \hdots & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\ ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \hdots & \hdots & \hdots & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\ ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \hdots & \hdots & \hdots & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2} \end{bmatrix}$$

Note: the last column is present if `N` is even, the last row is present if `M` is even. In the case of 1D real transform the result looks like the first row of the above matrix.

Here is the example of how to compute 2D convolution using DFT.

cv.Det (view/add comments)

Returns the determinant of a matrix.

```
Det(mat)-> double
```

mat The source matrix

The function returns the determinant of the square matrix `mat`. The direct method is used for small matrices and Gaussian elimination is used for larger matrices. For symmetric positive-determined matrices, it is also possible to run cv.SVD with $U = V = 0$ and then calculate the determinant as a product of the diagonal elements of W.

cv.Div (view/add comments)

Performs per-element division of two arrays.

```
Div(src1,src2,dst,scale)-> None
```

src1 The first source array. If the pointer is NULL, the array is assumed to be all 1's.

src2 The second source array

dst The destination array

scale Optional scale factor

The function divides one array by another:

$$\texttt{dst}(I) = \begin{cases} \texttt{scale} \cdot \texttt{src1}(I)/\texttt{src2}(I) & \text{if } \texttt{src1} \text{ is not } \texttt{NULL} \\ \texttt{scale}/\texttt{src2}(I) & \text{otherwise} \end{cases}$$

All the arrays must have the same type and the same size (or ROI size).

cv.DotProduct (view/add comments)

Calculates the dot product of two arrays in Euclidian metrics.

```
DotProduct(src1,src2)-> double
```

src1 The first source array

src2 The second source array

The function calculates and returns the Euclidean dot product of two arrays.

$$src1 \bullet src2 = \sum_I (\texttt{src1}(I)\texttt{src2}(I))$$

In the case of multiple channel arrays, the results for all channels are accumulated. In particular, `cvDotProduct(a,a)` where `a` is a complex vector, will return $||\texttt{a}||^2$. The function can process multi-dimensional arrays, row by row, layer by layer, and so on.

cv.EigenVV (view/add comments)

Computes eigenvalues and eigenvectors of a symmetric matrix.

```
EigenVV(mat,evects,evals,eps,lowindex,highindex)-> None
```

mat The input symmetric square matrix, modified during the processing

evects The output matrix of eigenvectors, stored as subsequent rows

evals The output vector of eigenvalues, stored in the descending order (order of eigenvalues and eigenvectors is syncronized, of course)

eps Accuracy of diagonalization. Typically, `DBL_EPSILON` (about 10^{-15}) works well. THIS PARAMETER IS CURRENTLY IGNORED.

lowindex Optional index of largest eigenvalue/-vector to calculate. (See below.)

highindex Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The function computes the eigenvalues and eigenvectors of matrix A:

```
mat*evects(i,:)' = evals(i)*evects(i,:)' (in MATLAB notation)
```

If either low- or highindex is supplied the other is required, too. Indexing is 0-based. Example: To calculate the largest eigenvector/-value set lowindex=highindex=0. To calculate all the eigenvalues, leave lowindex=highindex=-1. For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

The contents of matrix A is destroyed by the function.

Currently the function is slower than cv.SVD yet less accurate, so if A is known to be positively-defined (for example, it is a covariance matrix)it is recommended to use cv.SVD to find eigenvalues and eigenvectors of A, especially if eigenvectors are not required.

---

cv.Exp (view/add comments)

Calculates the exponent of every array element.

```
Exp(src,dst)-> None
```

src The source array

dst The destination array, it should have double type or the same type as the source

The function calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}(I)}$$

The maximum relative error is about 7×10^{-6}. Currently, the function converts denormalized values to zeros on output.

---

cv.FastArctan (view/add comments)

Calculates the angle of a 2D vector.

```
FastArctan(y,x)-> float
```

x x-coordinate of 2D vector

y y-coordinate of 2D vector

The function calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0 degrees to 360 degrees. The accuracy is about 0.1 degrees.

---

cv.Flip (view/add comments)

Flip a 2D array around vertical, horizontal or both axes.

```
Flip(src,dst=NULL,flipMode=0)-> None
```

src Source array

dst Destination array. If $\texttt{dst} = \texttt{NULL}$ the flipping is done in place.

flipMode Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas:

The function flips the array in one of three different ways (row and column indices are 0-based):

$$dst(i,j) = \begin{cases} \texttt{src}(rows(\texttt{src}) - i - 1, j) & \text{if } \texttt{flipMode} = 0 \\ \texttt{src}(i, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} > 0 \\ \texttt{src}(rows(\texttt{src}) - i - 1, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (flipMode = 0) to switch between top-left and bottom-left image origin, which is a typical operation in video processing under Win32 systems.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (flipMode $>$ 0)

- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flipMode < 0)
- reversing the order of 1d point arrays (flipMode ¿ 0)

---

cv.fromarray (view/add comments)

Create a CvMat from an object that supports the array interface.

```
fromarray(object, allowND = False) -> CvMat
```

object Any object that supports the array interface

allowND If true, will return a CvMatND

If the object supports the array interface, return a CvMat (`allowND = False`) or CvMatND (`allowND = True`).

If `allowND = False`, then the object's array must be either 2D or 3D. If it is 2D, then the returned CvMat has a single channel. If it is 3D, then the returned CvMat will have N channels, where N is the last dimension of the array. In this case, N cannot be greater than OpenCV's channel limit, `CV_CN_MAX`.

If `allowND = True`, then `fromarray` returns a single-channel CvMatND with the same shape as the original array.

For example, NumPy arrays support the array interface, so can be converted to OpenCV objects:

```
>>> import cv, numpy
>>> a = numpy.ones((480, 640))
>>> mat = cv.fromarray(a)
>>> print cv.GetDims(mat), cv.CV_MAT_CN(cv.GetElemType(mat))
(480, 640) 1
>>> a = numpy.ones((480, 640, 3))
>>> mat = cv.fromarray(a)
>>> print cv.GetDims(mat), cv.CV_MAT_CN(cv.GetElemType(mat))
(480, 640) 3
>>> a = numpy.ones((480, 640, 3))
>>> mat = cv.fromarray(a, allowND = True)
>>> print cv.GetDims(mat), cv.CV_MAT_CN(cv.GetElemType(mat))
(480, 640, 3) 1
```

cv.GEMM (view/add comments)

Performs generalized matrix multiplication.

```
GEMM(src1,src2,alphs,src3,beta,dst,tABC=0)-> None
```

src1 The first source array

src2 The second source array

src3 The third source array (shift). Can be NULL, if there is no shift.

dst The destination array

tABC The operation flags that can be 0 or a combination of the following values

CV_GEMM_A_T transpose src1

CV_GEMM_B_T transpose src2

CV_GEMM_C_T transpose src3

For example, CV_GEMM_A_T+CV_GEMM_C_T corresponds to

$$\texttt{alpha}\,\texttt{src1}^T\,\texttt{src2} + \texttt{beta}\,\texttt{src3}^T$$

The function performs generalized matrix multiplication:

$$\texttt{dst} = \texttt{alpha}\,op(\texttt{src1})\,op(\texttt{src2}) + \texttt{beta}\,op(\texttt{src3}) \quad \text{where } op(X) \text{ is } X \text{ or } X^T$$

All the matrices should have the same data type and coordinated sizes. Real or complex floating-point matrices are supported.

cv.Get1D (view/add comments)

Return a specific array element.

```
Get1D(arr, idx) -> scalar
```

arr Input array

idx Zero-based element index

Return a specific array element. Array must have dimension 3.

---

cv.Get2D (view/add comments)

Return a specific array element.

```
Get2D(arr, idx0, idx1) -> scalar
```

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

Return a specific array element. Array must have dimension 2.

---

cv.Get3D (view/add comments)

Return a specific array element.

```
Get3D(arr, idx0, idx1, idx2) -> scalar
```

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

Return a specific array element. Array must have dimension 3.

cv.GetND (view/add comments)

Return a specific array element.

```
GetND(arr, indices) -> scalar
```

arr Input array

indices List of zero-based element indices

Return a specific array element. The length of array indices must be the same as the dimension of the array.

cv.GetCol (view/add comments)

Returns array column.

```
GetCol(arr,col)-> submat
```

arr Input array

col Zero-based index of the selected column

submat resulting single-column array

The function `GetCol` returns a single column from the input array.

cv.GetCols (view/add comments)

Returns array column span.

```
GetCols(arr,startCol,endCol)-> submat
```

arr Input array

startCol Zero-based index of the starting column (inclusive) of the span

endCol Zero-based index of the ending column (exclusive) of the span

submat resulting multi-column array

The function `GetCols` returns a column span from the input array.

---

cv.GetDiag (view/add comments)

Returns one of array diagonals.

```
GetDiag(arr,diag=0)-> submat
```

arr Input array

submat Pointer to the resulting sub-array header

diag Array diagonal. Zero corresponds to the main diagonal, -1 corresponds to the diagonal above the main , 1 corresponds to the diagonal below the main, and so forth.

The function returns the header, corresponding to a specified diagonal of the input array.

---

cv.GetDims (view/add comments)

Returns list of array dimensions

```
GetDims(arr)-> list
```

arr Input array

The function returns a list of array dimensions. In the case of `IplImage` or CvMat it always returns a list of length 2.

cv.GetElemType (view/add comments)

Returns type of array elements.

```
GetElemType(arr)-> int
```

arr Input array

The function returns type of the array elements as described in cv.CreateMat discussion: `CV_8UC1` ... `CV_64FC4`.

cv.GetImage (view/add comments)

Returns image header for arbitrary array.

```
GetImage(arr) -> iplimage
```

arr Input array

The function returns the image header for the input array that can be a matrix - CvMat , or an image - `IplImage*`. In the case of an image the function simply returns the input pointer. In the case of CvMat it initializes an `imageHeader` structure with the parameters of the input matrix. Note that if we transform `IplImage` to CvMat and then transform CvMat back to IplImage, we can get different headers if the ROI is set, and thus some IPL functions that calculate image stride from its width and align may fail on the resultant image.

cv.GetImageCOI (view/add comments)

Returns the index of the channel of interest.

```
GetImageCOI(image)-> channel
```

image A pointer to the image header

Returns the channel of interest of in an IplImage. Returned values correspond to the `coi` in cv.SetImageCOI.

---

cv.GetImageROI (view/add comments)

Returns the image ROI.

```
GetImageROI(image)-> CvRect
```

image A pointer to the image header

If there is no ROI set, `cvRect(0,0,image->width,image->height)` is returned.

---

cv.GetMat (view/add comments)

Returns matrix header for arbitrary array.

```
GetMat(arr, allowND=0) -> cvmat
```

arr Input array

allowND If non-zero, the function accepts multi-dimensional dense arrays (CvMatND*) and returns 2D (if CvMatND has two dimensions) or 1D matrix (when CvMatND has 1 dimension or more than 2 dimensions). The array must be continuous.

The function returns a matrix header for the input array that can be a matrix - CvMat , an image - `IplImage` or a multi-dimensional dense array - CvMatND (latter case is allowed only if `allowND != 0`) . In the case of matrix the function simply returns the input pointer. In the case of `IplImage*` or CvMatND it initializes the `header` structure with parameters of the current image ROI and returns the pointer to this temporary structure. Because COI is not supported by CvMat , it is returned separately.

The function provides an easy way to handle both types of arrays - `IplImage` and CvMat - using the same code. Reverse transform from CvMat to `IplImage` can be done using the cv.GetImage function.

Input array must have underlying data allocated or attached, otherwise the function fails.

If the input array is `IplImage` with planar data layout and COI set, the function returns the pointer to the selected plane and COI = 0. It enables per-plane processing of multi-channel images with planar data layout using OpenCV functions.

---

cv.GetOptimalDFTSize (view/add comments)

Returns optimal DFT size for a given vector size.

```
GetOptimalDFTSize(size0)-> int
```

size0 Vector size

The function returns the minimum number `N` that is greater than or equal to `size0`, such that the DFT of a vector of size `N` can be computed fast. In the current implementation $N = 2^p \times 3^q \times 5^r$, for some p, q, r.

The function returns a negative number if `size0` is too large (very close to `INT_MAX`)

---

cv.GetReal1D (view/add comments)

Return a specific element of single-channel 1D array.

```
GetReal1D(arr, idx0)->float
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cv.Get function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

cv.GetReal2D (view/add comments)

Return a specific element of single-channel 2D array.

```
GetReal2D(arr, idx0, idx1)->float
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cv.Get function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

cv.GetReal3D (view/add comments)

Return a specific element of single-channel array.

```
GetReal3D(arr, idx0, idx1, idx2)->float
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cv.Get function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

cv.GetRealND (view/add comments)

Return a specific element of single-channel array.

```
GetRealND(arr, idx)->float
```

arr Input array. Must have a single channel.

idx Array of the element indices

Returns a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cv.Get function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

cv.GetRow (view/add comments)

Returns array row.

```
GetRow(arr,row)-> submat
```

arr Input array

row Zero-based index of the selected row

submat resulting single-row array

The function `GetRow` returns a single row from the input array.

cv.GetRows (view/add comments)

Returns array row span.

```
GetRows(arr,startRow,endRow,deltaRow=1)-> submat
```

arr Input array

startRow Zero-based index of the starting row (inclusive) of the span

endRow Zero-based index of the ending row (exclusive) of the span

deltaRow Index step in the row span.

submat resulting multi-row array

The function `GetRows` returns a row span from the input array.

---

cv.GetSize (view/add comments)

Returns size of matrix or image ROI.

```
GetSize(arr)-> CvSize
```

arr array header

The function returns number of rows (CvSize::height) and number of columns (CvSize::width) of the input matrix or image. In the case of image the size of ROI is returned.

---

cv.GetSubRect (view/add comments)

Returns matrix header corresponding to the rectangular sub-array of input image or matrix.

```
GetSubRect(arr, rect) -> cvmat
```

arr Input array

rect Zero-based coordinates of the rectangle of interest

The function returns header, corresponding to a specified rectangle of the input array. In other words, it allows the user to treat a rectangular part of input array as a stand-alone array. ROI is taken into account by the function so the sub-array of ROI is actually extracted.

---

cv.InRange (view/add comments)

Checks that array elements lie between the elements of two other arrays.

```
InRange(src,lower,upper,dst)-> None
```

src The first source array

lower The inclusive lower boundary array

upper The exclusive upper boundary array

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0 \wedge \texttt{lower}(I)_1 <= \texttt{src}(I)_1 < \texttt{upper}(I)_1$$

For two-channel arrays and so forth,
dst(I) is set to 0xff (all $\texttt{1}$-bits) if src(I) is within the range and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size).

---

cv.InRangeS (view/add comments)

Checks that array elements lie between two scalars.

```
InRangeS(src,lower,upper,dst)-> None
```

src The first source array

lower The inclusive lower boundary

upper The exclusive upper boundary

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0 \wedge \texttt{lower}_1 <= \texttt{src}(I)_1 < \texttt{upper}_1$$

For two-channel arrays nd so forth,
'dst(I)' is set to 0xff (all `1`-bits) if 'src(I)' is within the range and 0 otherwise. All the arrays must have the same size (or ROI size).

---

cv.InvSqrt (view/add comments)

Calculates the inverse square root.

```
InvSqrt(value)-> float
```

value The input floating-point value

The function calculates the inverse square root of the argument, and normally it is faster than `1./sqrt(value)`. If the argument is zero or negative, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

---

cv.Inv (view/add comments)

Synonym for Invert

cv.Invert (view/add comments)

Finds the inverse or pseudo-inverse of a matrix.

```
Invert(src,dst,method=CV_LU)-> double
```

src The source matrix

dst The destination matrix

method Inversion method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix

The function inverts matrix `src1` and stores the result in `src2`.

In the case of `LU` method, the function returns the `src1` determinant (src1 must be square). If it is 0, the matrix is not inverted and `src2` is filled with zeros.

In the case of `SVD` methods, the function returns the inversed condition of `src1` (ratio of the smallest singular value to the largest singular value) and 0 if `src1` is all zeros. The SVD methods calculate a pseudo-inverse matrix if `src1` is singular.

cv.IsInf (view/add comments)

Determines if the argument is Infinity.

```
IsInf(value)-> int
```

value The input floating-point value

The function returns 1 if the argument is $\pm\infty$ (as defined by IEEE754 standard), 0 otherwise.

cv.IsNaN (view/add comments)

Determines if the argument is Not A Number.

```
IsNaN(value)-> int
```

value The input floating-point value

The function returns 1 if the argument is Not A Number (as defined by IEEE754 standard), 0 otherwise.

cv.LUT (view/add comments)

Performs a look-up table transform of an array.

```
LUT(src,dst,lut)-> None
```

src Source array of 8-bit elements

dst Destination array of a given depth and of the same number of channels as the source array

lut Look-up table of 256 elements; should have the same depth as the destination array. In the case of multi-channel source and destination arrays, the table should either have a single-channel (in this case the same table is used for all channels) or the same number of channels as the source/destination array.

The function fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of `src` as follows:

$$\texttt{dst}_i \leftarrow \texttt{lut}_{\texttt{src}_i + d}$$

where

$$d = \begin{cases} 0 & \text{if } \texttt{src} \text{ has depth } \texttt{CV_8U} \\ 128 & \text{if } \texttt{src} \text{ has depth } \texttt{CV_8S} \end{cases}$$

cv.Log (view/add comments)

Calculates the natural logarithm of every array element's absolute value.

```
Log(src,dst)-> None
```

src The source array

dst The destination array, it should have `double` type or the same type as the source

The function calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}[I] = \begin{cases} \log|\texttt{src}(I)| & \text{if } \texttt{src}[I] \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where `C` is a large negative number (about -700 in the current implementation).

cv.Mahalanobis (view/add comments)

Calculates the Mahalanobis distance between two vectors.

```
Mahalonobis(vec1,vec2,mat)-> None
```

vec1 The first 1D source vector

vec2 The second 1D source vector

mat The inverse covariance matrix

The function calculates and returns the weighted distance between two vectors:

$$d(\texttt{vec1},\texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the cv.CalcCovarMatrix function and further inverted using the cv.Invert function (CV_SVD method is the prefered one because the matrix might be singular).

cv.Max (view/add comments)

Finds per-element maximum of two arrays.

```
Max(src1,src2,dst)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

cv.MaxS (view/add comments)

Finds per-element maximum of array and scalar.

```
MaxS(src,value,dst)-> None
```

src The first source array

value The scalar value

dst The destination array

The function calculates per-element maximum of array and scalar:

$$\texttt{dst}(I) = \max(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

cv.Merge (view/add comments)

Composes a multi-channel array from several single-channel arrays or inserts a single channel into the array.

```
Merge(src0,src1,src2,src3,dst)-> None
```

src0 Input channel 0

src1 Input channel 1

src2 Input channel 2

src3 Input channel 3

dst Destination array

The function is the opposite to cv.Split. If the destination array has N channels then if the first N input channels are not NULL, they all are copied to the destination array; if only a single source channel of the first N is not NULL, this particular channel is copied into the destination array; otherwise an error is raised. The rest of the source channels (beyond the first N) must always be NULL. For IplImage cv.Copy with COI set can be also used to insert a single channel into the image.

cv.Min (view/add comments)

Finds per-element minimum of two arrays.

```
Min(src1,src2,dst)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cv.MinMaxLoc (view/add comments)

Finds global minimum and maximum in array or subarray.

```
MinMaxLoc(arr,mask=NULL)-> (minVal,maxVal,minLoc,maxLoc)
```

arr The source array, single-channel or multi-channel with COI set

minVal Pointer to returned minimum value

maxVal Pointer to returned maximum value

minLoc Pointer to returned minimum location

maxLoc Pointer to returned maximum location

mask The optional mask used to select a subarray

The function finds minimum and maximum element values and their positions. The extremums are searched across the whole array, selected `ROI` (in the case of `IplImage`) or, if `mask` is not `NULL`, in the specified array region. If the array has more than one channel, it must be `IplImage` with `COI` set. In the case of multi-dimensional arrays, `minLoc->x` and `maxLoc->x` will contain raw (linear) positions of the extremums.

---

cv.MinS (view/add comments)

Finds per-element minimum of an array and a scalar.

```
MinS(src,value,dst)-> None
```

src The first source array

value The scalar value

dst The destination array

The function calculates minimum of an array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

Mirror

Synonym for Flip .

---

cv.MixChannels (view/add comments)

Copies several channels from input arrays to certain channels of output arrays

```
MixChannels(src, dst, fromTo) -> None
```

src Input arrays

dst Destination arrays

fromTo The array of pairs of indices of the planes copied. Each pair `fromTo[k]=(i,j)` means that i-th plane from `src` is copied to the j-th plane in `dst`, where continuous plane numbering is used both in the input array list and the output array list. As a special case, when the `fromTo[k][0]` is negative, the corresponding output plane `j` is filled with zero.

The function is a generalized form of cv.cvSplit and cv.Merge and some forms of CvtColor . It can be used to change the order of the planes, add/remove alpha channel, extract or insert a single plane or multiple planes etc.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B swapped) and separate alpha channel image:

```
rgba = cv.CreateMat(100, 100, cv.CV_8UC4)
bgr =  cv.CreateMat(100, 100, cv.CV_8UC3)
alpha = cv.CreateMat(100, 100, cv.CV_8UC1)
cv.Set(rgba, (1,2,3,4))
cv.MixChannels([rgba], [bgr, alpha], [
```

```
            (0, 2),     # rgba[0] -> bgr[2]
            (1, 1),     # rgba[1] -> bgr[1]
            (2, 0),     # rgba[2] -> bgr[0]
            (3, 3)      # rgba[3] -> alpha[0]
        ])
```

MulAddS

Synonym for ScaleAdd .

---

cv.Mul (view/add comments)

Calculates the per-element product of two arrays.

```
Mul(src1,src2,dst,scale)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

scale Optional scale factor

The function calculates the per-element product of two arrays:

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I)$$

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cv.MulSpectrums (view/add comments)

Performs per-element multiplication of two Fourier spectrums.

```
MulSpectrums(src1,src2,dst,flags)-> None
```

src1 The first source array

src2 The second source array

dst The destination array of the same type and the same size as the source arrays

flags A combination of the following values;

CV_DXT_ROWS treats each row of the arrays as a separate spectrum (see cv.DFT parameters description).

CV_DXT_MUL_CONJ conjugate the second source array before the multiplication.

The function performs per-element multiplication of the two CCS-packed or complex matrices that are results of a real or complex Fourier transform.

The function, together with cv.DFT, may be used to calculate convolution of two arrays rapidly.

---

cv.MulTransposed (view/add comments)

Calculates the product of an array and a transposed array.

```
MulTransposed(src,dst,order,delta=NULL,scale)-> None
```

src The source matrix

dst The destination matrix. Must be `CV_32F` or `CV_64F`.

order Order of multipliers

delta An optional array, subtracted from `src` before multiplication

scale An optional scaling

The function calculates the product of src and its transposition:

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})(\texttt{src} - \texttt{delta})^T$$

if $\texttt{order} = 0$, and

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})^T(\texttt{src} - \texttt{delta})$$

otherwise.

cv.Norm (view/add comments)

Calculates absolute array norm, absolute difference norm, or relative difference norm.

```
Norm(arr1,arr2,normType=CV_L2,mask=NULL)-> double
```

arr1 The first source image

arr2 The second source image. If it is NULL, the absolute norm of `arr1` is calculated, otherwise the absolute or relative norm of `arr1-arr2` is calculated.

normType Type of norm, see the discussion

mask The optional operation mask

The function calculates the absolute norm of `arr1` if `arr2` is NULL:

$$norm = \begin{cases} ||\texttt{arr1}||_C = \max_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV_C} \\ ||\texttt{arr1}||_{L1} = \sum_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV_L1} \\ ||\texttt{arr1}||_{L2} = \sqrt{\sum_I \texttt{arr1}(I)^2} & \text{if } \texttt{normType} = \texttt{CV_L2} \end{cases}$$

or the absolute difference norm if `arr2` is not NULL:

$$norm = \begin{cases} ||\texttt{arr1} - \texttt{arr2}||_C = \max_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV_C} \\ ||\texttt{arr1} - \texttt{arr2}||_{L1} = \sum_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV_L1} \\ ||\texttt{arr1} - \texttt{arr2}||_{L2} = \sqrt{\sum_I (\texttt{arr1}(I) - \texttt{arr2}(I))^2} & \text{if } \texttt{normType} = \texttt{CV_L2} \end{cases}$$

or the relative difference norm if `arr2` is not NULL and `(normType & CV_RELATIVE) != 0`:

$$norm = \begin{cases} \frac{||\texttt{arr1}-\texttt{arr2}||_C}{||\texttt{arr2}||_C} & \text{if } \texttt{normType} = \texttt{CV_RELATIVE_C} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L1}}{||\texttt{arr2}||_{L1}} & \text{if } \texttt{normType} = \texttt{CV_RELATIVE_L1} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L2}}{||\texttt{arr2}||_{L2}} & \text{if } \texttt{normType} = \texttt{CV_RELATIVE_L2} \end{cases}$$

The function returns the calculated norm. A multiple-channel array is treated as a single-channel, that is, the results for all channels are combined.

cv.Not (view/add comments)

Performs per-element bit-wise inversion of array elements.

```
Not(src,dst)-> None
```

src The source array

dst The destination array

The function Not inverses every bit of every array element:

```
dst(I)=~src(I)
```

cv.Or (view/add comments)

Calculates per-element bit-wise disjunction of two arrays.

```
Or(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise disjunction of two arrays:

```
dst(I)=src1(I)|src2(I)
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

cv.OrS (view/add comments)

Calculates a per-element bit-wise disjunction of an array and a scalar.

```
OrS(src,value,dst,mask=NULL)-> None
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function OrS calculates per-element bit-wise disjunction of an array and a scalar:

```
dst(I)=src(I)|value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

cv.PerspectiveTransform (view/add comments)

Performs perspective matrix transformation of a vector array.

```
PerspectiveTransform(src,dst,mat)-> None
```

src The source three-channel floating-point array

dst The destination three-channel floating-point array

mat 3×3 or 4×4 transformation matrix

The function transforms every element of `src` (by treating it as 2D or 3D vector) in the following way:

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

---

cv.PolarToCart (view/add comments)

Calculates Cartesian coordinates of 2d vectors represented in polar form.

```
PolarToCart(magnitude,angle,x,y,angleInDegrees=0)-> None
```

magnitude The array of magnitudes. If it is NULL, the magnitudes are assumed to be all 1's.

angle The array of angles, whether in radians or degrees

x The destination array of x-coordinates, may be set to NULL if it is not needed

y The destination array of y-coordinates, mau be set to NULL if it is not needed

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the x-coodinate, y-coordinate or both of every vector `magnitude(I)*exp(ang j=sqrt(-1)`:

```
x(I)=magnitude(I)*cos(angle(I)),
y(I)=magnitude(I)*sin(angle(I))
```

cv.Pow (view/add comments)

Raises every array element to a power.

```
Pow(src,dst,power)-> None
```

src The source array

dst The destination array, should be the same type as the source

power The exponent of power

The function raises every element of the input array to p:

$$\texttt{dst}[I] = \begin{cases} \texttt{src}(I)^p & \text{if p is integer} \\ |\texttt{src}(I)^p| & \text{otherwise} \end{cases}$$

That is, for a non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following example, computing the cube root of array elements, shows:

```
>>> import cv
>>> src = cv.CreateMat(1, 10, cv.CV_32FC1)
>>> mask = cv.CreateMat(src.rows, src.cols, cv.CV_8UC1)
>>> dst = cv.CreateMat(src.rows, src.cols, cv.CV_32FC1)
>>> cv.CmpS(src, 0, mask, cv.CV_CMP_LT)          # find negative elements
>>> cv.Pow(src, dst, 1. / 3)
>>> cv.SubRS(dst, cv.ScalarAll(0), dst, mask)    # negate the results of negative inputs
```

For some values of power, such as integer values, 0.5, and -0.5, specialized faster algorithms are used.

cv.RNG (view/add comments)

Initializes a random number generator state.

```
RNG(seed=-1LL)-> CvRNG
```

seed 64-bit value used to initiate a random sequence

The function initializes a random number generator and returns the state. The pointer to the state can be then passed to the cv.RandInt, cv.RandReal and cv.RandArr functions. In the current implementation a multiply-with-carry generator is used.

---

cv.RandArr (view/add comments)

Fills an array with random numbers and updates the RNG state.

```
RandArr(rng,arr,distType,param1,param2)-> None
```

rng RNG state initialized by cv.RNG

arr The destination array

distType Distribution type

CV_RAND_UNI uniform distribution

CV_RAND_NORMAL normal or Gaussian distribution

param1 The first parameter of the distribution. In the case of a uniform distribution it is the inclusive lower boundary of the random numbers range. In the case of a normal distribution it is the mean value of the random numbers.

param2 The second parameter of the distribution. In the case of a uniform distribution it is the exclusive upper boundary of the random numbers range. In the case of a normal distribution it is the standard deviation of the random numbers.

The function fills the destination array with uniformly or normally distributed random numbers.

---

cv.RandInt (view/add comments)

Returns a 32-bit unsigned integer and updates RNG.

```
RandInt(rng)-> unsigned
```

rng RNG state initialized by `RandInit` and, optionally, customized by `RandSetRange` (though, the latter function does not affect the discussed function outcome)

The function returns a uniformly-distributed random 32-bit unsigned integer and updates the RNG state. It is similar to the rand() function from the C runtime library, but it always generates a 32-bit number whereas rand() returns a number in between 0 and `RAND_MAX` which is 2^{16} or 2^{32}, depending on the platform.

The function is useful for generating scalar random numbers, such as points, patch sizes, table indices, etc., where integer numbers of a certain range can be generated using a modulo operation and floating-point numbers can be generated by scaling from 0 to 1 or any other specific range.

---

cv.RandReal (view/add comments)

Returns a floating-point random number and updates RNG.

```
RandReal(rng)-> double
```

rng RNG state initialized by cv.RNG

The function returns a uniformly-distributed random floating-point number between 0 and 1 (1 is not included).

---

cv.Reduce (view/add comments)

Reduces a matrix to a vector.

```
Reduce(src,dst,dim=-1,op=CV_REDUCE_SUM)-> None
```

src The input matrix.

dst The output single-row/single-column vector that accumulates somehow all the matrix rows/-columns.

dim The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row, 1 means that the matrix is reduced to a single column and -1 means that the dimension is chosen automatically by analysing the dst size.

op The reduction operation. It can take of the following values:

- **CV_REDUCE_SUM** The output is the sum of all of the matrix's rows/columns.
- **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
- **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/-columns.
- **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/-columns.

The function reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In the case of CV_REDUCE_SUM and CV_REDUCE_AVG the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

---

cv.Repeat (view/add comments)

Fill the destination array with repeated copies of the source array.

```
Repeat(src,dst)-> None
```

src Source array, image or matrix

dst Destination array, image or matrix

The function fills the destination array with repeated copies of the source array:

```
dst(i,j)=src(i mod rows(src), j mod cols(src))
```

So the destination array may be as larger as well as smaller than the source array.

---

cv.ResetImageROI (view/add comments)

Resets the image ROI to include the entire image and releases the ROI structure.

```
ResetImageROI(image)-> None
```

image A pointer to the image header

This produces a similar result to the following

```
cv.SetImageROI(image, (0, 0, image.width, image.height))
cv.SetImageCOI(image, 0)
```

---

cv.Reshape (view/add comments)

Changes shape of matrix/image without copying data.

```
Reshape(arr, newCn, newRows=0) -> cvmat
```

arr Input array

newCn New number of channels. 'newCn = 0' means that the number of channels remains unchanged.

newRows New number of rows. 'newRows = 0' means that the number of rows remains unchanged unless it needs to be changed according to `newCn` value.

The function initializes the CvMat header so that it points to the same data as the original array but has a different shape - different number of channels, different number of rows, or both.

---

cv.ReshapeMatND (view/add comments)

Changes the shape of a multi-dimensional array without copying the data.

```
ReshapeMatND(arr, newCn, newDims) -> cvmat
```

arr Input array

newCn New number of channels. `newCn` $= 0$ means that the number of channels remains unchanged.

newDims List of new dimensions.

Returns a new CvMatND that shares the same data as `arr` but has different dimensions or number of channels. The only requirement is that the total length of the data is unchanged.

```
>>> import cv
>>> mat = cv.CreateMatND([24], cv.CV_32FC1)
>>> print cv.GetDims(cv.ReshapeMatND(mat, 0, [8, 3]))
(8, 3)
>>> m2 = cv.ReshapeMatND(mat, 4, [3, 2])
>>> print cv.GetDims(m2)
(3, 2)
>>> print m2.channels
4
```

---

cv.Round (view/add comments)

Converts a floating-point number to the nearest integer value.

```
Round(value) -> int
```

value The input floating-point value

On some architectures this function is much faster than the standard cast operations. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

---

cv.Floor (view/add comments)

Converts a floating-point number to the nearest integer value that is not larger than the argument.

```
Floor(value) -> int
```

value The input floating-point value

On some architectures this function is much faster than the standard cast operations. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

cv.Ceil (view/add comments)

Converts a floating-point number to the nearest integer value that is not smaller than the argument.

```
Ceil(value) -> int
```

value The input floating-point value

On some architectures this function is much faster than the standard cast operations. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

cv.ScaleAdd (view/add comments)

Calculates the sum of a scaled array and another array.

```
ScaleAdd(src1,scale,src2,dst)-> None
```

src1 The first source array

scale Scale factor for the first array

src2 The second source array

dst The destination array

The function calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale}\,\texttt{src1}(I) + \texttt{src2}(I)$$

All array parameters should have the same type and the same size.

cv.Set (view/add comments)

Sets every element of an array to a given value.

```
Set(arr,value,mask=NULL)-> None
```

arr The destination array

value Fill value

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies the scalar `value` to every selected element of the destination array:

$$\texttt{arr}(I) = \texttt{value} \quad \text{if} \quad \texttt{mask}(I) \neq 0$$

If array `arr` is of `IplImage` type, then is ROI used, but COI must not be set.

---

cv.Set1D (view/add comments)

Set a specific array element.

```
Set1D(arr, idx, value) -> None
```

arr Input array

idx Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 1.

---

cv.Set2D (view/add comments)

Set a specific array element.

```
Set2D(arr, idx0, idx1, value) -> None
```

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

value The value to assign to the element

Sets a specific array element. Array must have dimension 2.

---

cv.Set3D (view/add comments)

Set a specific array element.

```
Set3D(arr, idx0, idx1, idx2, value) -> None
```

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 3.

---

cv.SetND (view/add comments)

Set a specific array element.

```
SetND(arr, indices, value) -> None
```

arr Input array

indices List of zero-based element indices

value The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

---

cv.SetData (view/add comments)

Assigns user data to the array header.

```
SetData(arr, data, step)-> None
```

arr Array header

data User data

step Full row length in bytes

The function assigns user data to the array header. Header should be initialized before using `cvCreate*Header`, `cvInit*Header` or cv.Mat (in the case of matrix) function.

---

cv.SetIdentity (view/add comments)

Initializes a scaled identity matrix.

```
SetIdentity(mat,value=1)-> None
```

mat The matrix to initialize (not necesserily square)

value The value to assign to the diagonal elements

The function initializes a scaled identity matrix:

$$\texttt{arr}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

cv.SetImageCOI (view/add comments)

Sets the channel of interest in an IplImage.

```
SetImageCOI(image, coi)-> None
```

image A pointer to the image header

coi The channel of interest. 0 - all channels are selected, 1 - first channel is selected, etc. Note that the channel indices become 1-based.

If the ROI is set to NULL and the coi is *not* 0, the ROI is allocated. Most OpenCV functions do *not* support the COI setting, so to process an individual image/matrix channel one may copy (via cv.Copy or cv.Split) the channel to a separate image/matrix, process it and then copy the result back (via cv.Copy or cv.Merge) if needed.

cv.SetImageROI (view/add comments)

Sets an image Region Of Interest (ROI) for a given rectangle.

```
SetImageROI(image, rect)-> None
```

image A pointer to the image header

rect The ROI rectangle

If the original image ROI was NULL and the rect is not the whole image, the ROI structure is allocated.

Most OpenCV functions support the use of ROI and treat the image rectangle as a separate image. For example, all of the pixel coordinates are counted from the top-left (or bottom-left) corner of the ROI, not the original image.

cv.SetReal1D (view/add comments)

Set a specific array element.

```
SetReal1D(arr, idx, value) -> None
```

arr Input array

idx Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 1.

cv.SetReal2D (view/add comments)

Set a specific array element.

```
SetReal2D(arr, idx0, idx1, value) -> None
```

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

value The value to assign to the element

Sets a specific array element. Array must have dimension 2.

cv.SetReal3D (view/add comments)

Set a specific array element.

```
SetReal3D(arr, idx0, idx1, idx2, value) -> None
```

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 3.

---

cv.SetRealND (view/add comments)

Set a specific array element.

```
SetRealND(arr, indices, value) -> None
```

arr Input array

indices List of zero-based element indices

value The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

---

cv.SetZero (view/add comments)

Clears the array.

```
SetZero(arr)-> None
```

arr Array to be cleared

The function clears the array. In the case of dense arrays (CvMat, CvMatND or IplImage), cvZero(array) is equivalent to cvSet(array,cvScalarAll(0),0). In the case of sparse arrays all the elements are removed.

---

cv.Solve (view/add comments)

Solves a linear system or least-squares problem.

```
Solve(A,B,X,method=CV_LU)-> None
```

A The source matrix

B The right-hand part of the linear system

X The output solution

method The solution (matrix inversion) method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix.

The function solves a linear system or least-squares problem (the latter is possible with SVD methods):

$$\texttt{dst} = argmin_X ||\texttt{src1}\,\texttt{X} - \texttt{src2}||$$

If CV_LU method is used, the function returns 1 if src1 is non-singular and 0 otherwise; in the latter case dst is not valid.

---

cv.SolveCubic (view/add comments)

Finds the real roots of a cubic equation.

```
SolveCubic(coeffs,roots)-> None
```

coeffs The equation coefficients, an array of 3 or 4 elements

roots The output array of real roots which should have 3 elements

The function finds the real roots of a cubic equation:
If coeffs is a 4-element vector:

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or if coeffs is 3-element vector:

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

The function returns the number of real roots found. The roots are stored to `root` array, which is padded with zeros if there is only one root.

---

cv.Split (view/add comments)

Divides multi-channel array into several single-channel arrays or extracts a single channel from the array.

```
Split(src,dst0,dst1,dst2,dst3)-> None
```

src Source array

dst0 Destination channel 0

dst1 Destination channel 1

dst2 Destination channel 2

dst3 Destination channel 3

The function divides a multi-channel array into separate single-channel arrays. Two modes are available for the operation. If the source array has N channels then if the first N destination channels are not NULL, they all are extracted from the source array; if only a single destination channel of the first N is not NULL, this particular channel is extracted; otherwise an error is raised. The rest of the destination channels (beyond the first N) must always be NULL. For IplImage cv.Copy with COI set can be also used to extract a single channel from the image.

cv.Sqrt (view/add comments)

Calculates the square root.

```
Sqrt(value)-> float
```

value The input floating-point value

The function calculates the square root of the argument. If the argument is negative, the result is not determined.

cv.Sub (view/add comments)

Computes the per-element difference between two arrays.

```
Sub(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts one array from another one:

```
dst(I)=src1(I)-src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cv.SubRS (view/add comments)

Computes the difference between a scalar and an array.

```
SubRS(src,value,dst,mask=NULL)-> None
```

src The first source array

value Scalar to subtract from

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts every element of source array from a scalar:

```
dst(I)=value-src(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cv.SubS (view/add comments)

Computes the difference between an array and a scalar.

```
SubS(src,value,dst,mask=NULL)-> None
```

src The source array

value Subtracted scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts a scalar from every element of the source array:

```
dst(I)=src(I)-value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cv.Sum (view/add comments)

Adds up array elements.

```
Sum(arr)-> CvScalar
```

arr The array

The function calculates the sum `S` of array elements, independently for each channel:

$$\sum_I \texttt{arr}(I)_c$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the sum to the first scalar component.

cv.SVBkSb (view/add comments)

Performs singular value back substitution.

```
SVBkSb(W,U,V,B,X,flags)-> None
```

W Matrix or vector of singular values

U Left orthogonal matrix (tranposed, perhaps)

V Right orthogonal matrix (tranposed, perhaps)

B The matrix to multiply the pseudo-inverse of the original matrix `A` by. This is an optional parameter. If it is omitted then it is assumed to be an identity matrix of an appropriate size (so that `X` will be the reconstructed pseudo-inverse of `A`).

X The destination matrix: result of back substitution

flags Operation flags, should match exactly to the `flags` passed to cv.SVD

The function calculates back substitution for decomposed matrix A (see cv.SVD description) and matrix B:

$$X = VW^{-1}U^{T}B$$

where

$$W^{-1}_{(i,i)} = \begin{cases} 1/W_{(i,i)} & \text{if } W_{(i,i)} > \epsilon \sum_i W_{(i,i)} \\ 0 & \text{otherwise} \end{cases}$$

and ϵ is a small number that depends on the matrix data type.

This function together with cv.SVD is used inside cv.Invert and cv.Solve, and the possible reason to use these (svd and bksb) "low-level" function, is to avoid allocation of temporary matrices inside the high-level counterparts (inv and solve).

---

cv.SVD (view/add comments)

Performs singular value decomposition of a real floating-point matrix.

```
SVD(A,W, U = None, V = None, flags=0)-> None
```

A Source $M \times N$ matrix

W Resulting singular value diagonal matrix ($M \times N$ or $\min(M,N) \times \min(M,N)$) or $\min(M,N) \times 1$ vector of the singular values

U Optional left orthogonal matrix, $M \times \min(M,N)$ (when CV_SVD_U_T is not set), or $\min(M,N) \times M$ (when CV_SVD_U_T is set), or $M \times M$ (regardless of CV_SVD_U_T flag).

V Optional right orthogonal matrix, $N \times \min(M,N)$ (when CV_SVD_V_T is not set), or $\min(M,N) \times N$ (when CV_SVD_V_T is set), or $N \times N$ (regardless of CV_SVD_V_T flag).

flags Operation flags; can be 0 or a combination of the following values:

- **CV_SVD_MODIFY_A** enables modification of matrix A during the operation. It speeds up the processing.
- **CV_SVD_U_T** means that the transposed matrix U is returned. Specifying the flag speeds up the processing.
- **CV_SVD_V_T** means that the transposed matrix V is returned. Specifying the flag speeds up the processing.

The function decomposes matrix A into the product of a diagonal matrix and two orthogonal matrices:

$$A = U\,W\,V^T$$

where W is a diagonal matrix of singular values that can be coded as a 1D vector of singular values and U and V. All the singular values are non-negative and sorted (together with U and V columns) in descending order.

An SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix A is a square, symmetric, and positively defined matrix, for example, when it is a covariance matrix. W in this case will be a vector/-matrix of the eigenvalues, and $U = V$ will be a matrix of the eigenvectors.
- accurate solution of a poor-conditioned linear system.
- least-squares solution of an overdetermined linear system. This and the preceeding is done by using the cv.Solve function with the CV_SVD method.
- accurate calculation of different matrix characteristics such as the matrix rank (the number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), and determinant (absolute value of the determinant is equal to the product of singular values).

---

cv.Trace (view/add comments)

Returns the trace of a matrix.

```
Trace(mat)-> CvScalar
```

mat The source matrix

The function returns the sum of the diagonal elements of the matrix src1.

$$tr(\texttt{mat}) = \sum_i \texttt{mat}(i,i)$$

cv.Transform (view/add comments)

Performs matrix transformation of every array element.

```
Transform(src,dst,transmat,shiftvec=NULL)-> None
```

src The first source array

dst The destination array

transmat Transformation matrix

shiftvec Optional shift vector

The function performs matrix transformation of every element of array `src` and stores the results in `dst`:

$$dst(I) = transmat \cdot src(I) + shiftvec$$

That is, every element of an `N`-channel array `src` is considered as an `N`-element vector which is transformed using a `M` × `N` matrix `transmat` and shift vector `shiftvec` into an element of `M`-channel array `dst`. There is an option to embedd `shiftvec` into `transmat`. In this case `transmat` should be a `M` × $(N + 1)$ matrix and the rightmost column is treated as the shift vector.

Both source and destination arrays should have the same depth and the same size or selected ROI size. `transmat` and `shiftvec` should be real floating-point matrices.

The function may be used for geometrical transformation of n dimensional point set, arbitrary linear color space transformation, shuffling the channels and so forth.

cv.Transpose (view/add comments)

Transposes a matrix.

```
Transpose(src,dst)-> None
```

src The source matrix

dst The destination matrix

The function transposes matrix `src1`:

$$\texttt{dst}(i,j) = \texttt{src}(j,i)$$

Note that no complex conjugation is done in the case of a complex matrix. Conjugation should be done separately: look at the sample code in cv.XorS for an example.

---

cv.Xor (view/add comments)

Performs per-element bit-wise "exclusive or" operation on two arrays.

```
Xor(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)^src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.XorS (view/add comments)

Performs per-element bit-wise "exclusive or" operation on an array and a scalar.

```
XorS(src,value,dst,mask=NULL)-> None
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function XorS calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)^value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

---

cv.mGet (view/add comments)

Returns the particular element of single-channel floating-point matrix.

```
mGet(mat,row,col)-> double
```

mat Input matrix

row The zero-based index of row

col The zero-based index of column

The function is a fast replacement for cv.GetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

---

cv.mSet (view/add comments)

Returns a specific element of a single-channel floating-point matrix.

```
mSet(mat,row,col,value)-> None
```

mat The matrix

row The zero-based index of row

col The zero-based index of column

value The new value of the matrix element

The function is a fast replacement for cv.SetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

22.3 Dynamic Structures

CvMemStorage (view/add comments)

Growing memory storage.

Many OpenCV functions use a given storage area for their results and working storage. These storage areas can be created using CreateMemStorage . OpenCV Python tracks the objects occupying a CvMemStorage, and automatically releases the CvMemStorage when there are no objects referring to it. For this reason, there is explicit function to release a CvMemStorage.

```
>>> import cv
>>> image = cv.LoadImageM("building.jpg", cv.CV_LOAD_IMAGE_GRAYSCALE)
>>> seq = cv.FindContours(image, cv.CreateMemStorage(), cv.CV_RETR_TREE, cv.CV_CHAIN_APPROX
>>> del seq   # associated storage is also released
```

CvSeq (view/add comments)

Growable sequence of elements.

Many OpenCV functions return a CvSeq object. The CvSeq obect is a sequence, so these are all legal:

```
seq = cv.FindContours(scribble, storage, cv.CV_RETR_CCOMP, cv.CV_CHAIN_APPROX_SIMPLE)
# seq is a sequence of point pairs
print len(seq)
# FindContours returns a sequence of (x,y) points, so to print them out:
for (x,y) in seq:
   print (x,y)
print seq[10]             # tenth entry in the seqeuence
print seq[::-1]           # reversed sequence
print sorted(list(seq))   # sorted sequence
```

Also, a CvSeq object has methods `h_next()`, `h_prev()`, `v_next()` and `v_prev()`. Some OpenCV functions (for example cv.FindContours) can return multiple CvSeq objects, connected by these relations. In this case the methods return the other sequences. If no relation between sequences exists, then the methods return `None`.

CvSet (view/add comments)

Collection of nodes.

Some OpenCV functions return a CvSet object. The CvSet obect is iterable, for example:

```
for i in s:
  print i
print set(s)
print list(s)
```

---

cv.CloneSeq (view/add comments)

Creates a copy of a sequence.

```
CloneSeq(seq,storage)-> None
```

seq Sequence

storage The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

The function makes a complete copy of the input sequence and returns it.

---

cv.CreateMemStorage (view/add comments)

Creates memory storage.

```
CreateMemStorage(blockSize = 0) -> memstorage
```

blockSize Size of the storage blocks in bytes. If it is 0, the block size is set to a default value - currently it is about 64K.

The function creates an empty memory storage. See CvMemStorage description.

cv.SeqInvert (view/add comments)

Reverses the order of sequence elements.

```
SeqInvert(seq)-> None
```

seq Sequence

The function reverses the sequence in-place - makes the first element go last, the last element go first and so forth.

cv.SeqRemove (view/add comments)

Removes an element from the middle of a sequence.

```
SeqRemove(seq,index)-> None
```

seq Sequence

index Index of removed element

The function removes elements with the given index. If the index is out of range the function reports an error. An attempt to remove an element from an empty sequence is a special case of this situation. The function removes an element by shifting the sequence elements between the nearest end of the sequence and the `index`-th position, not counting the latter.

cv.SeqRemoveSlice (view/add comments)

Removes a sequence slice.

```
SeqRemoveSlice(seq,slice)-> None
```

seq Sequence

slice The part of the sequence to remove

The function removes a slice from the sequence.

22.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with `CV_RGB`) for color images and brightness for grayscale images. For color images the order channel is normally *Blue, Green, Red*, this is what cv., cv. and cv. expect

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using cv..

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}(x*2^{-shift}, y*2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

---

cv.Circle (view/add comments)

Draws a circle.

```
Circle(img,center,radius,color,thickness=1,lineType=8,shift=0)-> None
```

img Image where the circle is drawn

center Center of the circle

radius Radius of the circle

color Circle color

thickness Thickness of the circle outline if positive, otherwise this indicates that a filled circle is to be drawn

lineType Type of the circle boundary, see Line description

shift Number of fractional bits in the center coordinates and radius value

The function draws a simple or filled circle with a given center and radius.

---

cv.ClipLine (view/add comments)

Clips the line against the image rectangle.

```
ClipLine(imgSize, pt1, pt2) -> (clipped_pt1, clipped_pt2)
```

imgSize Size of the image

pt1 First ending point of the line segment.

pt2 Second ending point of the line segment.

The function calculates a part of the line segment which is entirely within the image. If the line segment is outside the image, it returns None. If the line segment is inside the image it returns a new pair of points.

---

cv.DrawContours (view/add comments)

Draws contour outlines or interiors in an image.

```
DrawContours(img,contour,external_color,hole_color,max_level,thickness=1,lineType=8,
None
```

img Image where the contours are to be drawn. As with any other drawing function, the contours are clipped with the ROI.

contour Pointer to the first contour

external_color Color of the external contours

hole_color Color of internal contours (holes)

max_level Maximal level for drawn contours. If 0, only `contour` is drawn. If 1, the contour and all contours following it on the same level are drawn. If 2, all contours following and all contours one level below the contours are drawn, and so forth. If the value is negative, the function does not draw the contours following after `contour` but draws the child contours of `contour` up to the $|\texttt{max_level}| - 1$ level.

thickness Thickness of lines the contours are drawn with. If it is negative (For example, =CV_FILLED), the contour interiors are drawn.

lineType Type of the contour segments, see Line description

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$.

---

cv.Ellipse (view/add comments)

Draws a simple or thick elliptic arc or an fills ellipse sector.

```
Ellipse(img,center,axes,angle,start_angle,end_angle,color,thickness=1,lineType=8,shi
None
```

img The image

center Center of the ellipse

axes Length of the ellipse axes

angle Rotation angle

start_angle Starting angle of the elliptic arc

end_angle Ending angle of the elliptic arc.

color Ellipse color

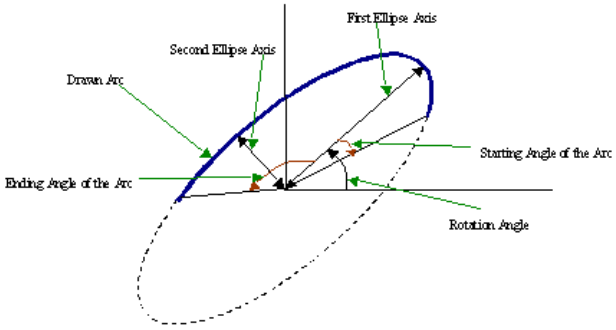
thickness Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the center coordinates and axes' values

The function draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by the ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are given in degrees. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



---

cv.EllipseBox (view/add comments)

Draws a simple or thick elliptic arc or fills an ellipse sector.

```
EllipseBox(img,box,color,thickness=1,lineType=8,shift=0)-> None
```

img Image

box The enclosing box of the ellipse drawn

thickness Thickness of the ellipse boundary

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the box vertex coordinates

The function draws a simple or thick ellipse outline, or fills an ellipse. The functions provides a convenient way to draw an ellipse approximating some shape; that is what CamShift and FitEllipse do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

---

cv.FillConvexPoly (view/add comments)

Fills a convex polygon.

```
FillConvexPoly(img,pn,color,lineType=8,shift=0)-> None
```

img Image

pn List of coordinate pairs

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills a convex polygon's interior. This function is much faster than the function `cvFillPoly` and can fill not only convex polygons but any monotonic polygon, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most.

---

cv.FillPoly (view/add comments)

Fills a polygon's interior.

```
FillPoly(img,polys,color,lineType=8,shift=0)-> None
```

img Image

polys List of lists of (x,y) pairs. Each list of points is a polygon.

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, and so forth.

---

cv.GetTextSize (view/add comments)

Retrieves the width and height of a text string.

```
GetTextSize(textString,font)-> (textSize,baseline)
```

font Pointer to the font structure

textString Input string

textSize Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.

baseline y-coordinate of the baseline relative to the bottom-most text point

The function calculates the dimensions of a rectangle to enclose a text string when a specified font is used.

---

cv.InitFont (view/add comments)

Initializes font structure.

```
InitFont(fontFace,hscale,vscale,shear=0,thickness=1,lineType=8)-> font
```

font Pointer to the font structure initialized by the function

fontFace Font name identifier. Only a subset of Hershey fonts http://sources.isc.org/utils/misc/hershey-font.txt are supported now:

CV_FONT_HERSHEY_SIMPLEX normal size sans-serif font

CV_FONT_HERSHEY_PLAIN small size sans-serif font

CV_FONT_HERSHEY_DUPLEX normal size sans-serif font (more complex than CV_FONT_HERSHEY_SIMPLEX)

CV_FONT_HERSHEY_COMPLEX normal size serif font

CV_FONT_HERSHEY_TRIPLEX normal size serif font (more complex than CV_FONT_HERSHEY_COMPLEX)

CV_FONT_HERSHEY_COMPLEX_SMALL smaller version of CV_FONT_HERSHEY_COMPLEX

CV_FONT_HERSHEY_SCRIPT_SIMPLEX hand-writing style font

CV_FONT_HERSHEY_SCRIPT_COMPLEX more complex variant of CV_FONT_HERSHEY_SCRIPT_SIMPLEX

The parameter can be composited from one of the values above and an optional CV_FONT_ITALIC flag, which indicates italic or oblique font.

hscale Horizontal scale. If equal to 1.0f, the characters have the original width depending on the font type. If equal to 0.5f, the characters are of half the original width.

vscale Vertical scale. If equal to 1.0f, the characters have the original height depending on the font type. If equal to 0.5f, the characters are of half the original height.

shear Approximate tangent of the character slope relative to the vertical line. A zero value means a non-italic font, 1.0f means about a 45 degree slope, etc.

thickness Thickness of the text strokes

lineType Type of the strokes, see Line description

The function initializes the font structure that can be passed to text rendering functions.

---

cv.InitLineIterator (view/add comments)

Initializes the line iterator.

```
InitLineIterator(image, pt1, pt2, connectivity=8, left_to_right=0) ->
line_iterator
```

image Image to sample the line from

pt1 First ending point of the line segment

pt2 Second ending point of the line segment

connectivity The scanned line connectivity, 4 or 8.

left_to_right If ($\texttt{left_to_right} = 0$) then the line is scanned in the specified order, from $\texttt{pt1}$ to $\texttt{pt2}$. If ($\texttt{left_to_right} \neq 0$) the line is scanned from left-most point to right-most.

line_iterator Iterator over the pixels of the line

The function returns an iterator over the pixels connecting the two points.

The points on the line are calculated one by one using a 4-connected or 8-connected Bresenham algorithm.

Example: Using line iterator to calculate the sum of pixel values along a color line

```
>>> import cv
>>> img = cv.LoadImageM("building.jpg", cv.CV_LOAD_IMAGE_COLOR)
>>> li = cv.InitLineIterator(img, (100, 100), (125, 150))
>>> red_sum = 0
>>> green_sum = 0
>>> blue_sum = 0
>>> for (r, g, b) in li:
...     red_sum += r
...     green_sum += g
...     blue_sum += b
>>> print red_sum, green_sum, blue_sum
10935.0 9496.0 7946.0
```

or more concisely using zip:

```
>>> import cv
>>> img = cv.LoadImageM("building.jpg", cv.CV_LOAD_IMAGE_COLOR)
>>> li = cv.InitLineIterator(img, (100, 100), (125, 150))
>>> print [sum(c) for c in zip(*li)]
[10935.0, 9496.0, 7946.0]
```

---

cv.Line (view/add comments)

Draws a line segment connecting two points.

```
Line(img,pt1,pt2,color,thickness=1,lineType=8,shift=0)-> None
```

img The image

pt1 First point of the line segment

pt2 Second point of the line segment

color Line color

thickness Line thickness

lineType Type of the line:

8 (or omitted) 8-connected line.

4 4-connected line.

CV_AA antialiased line.

shift Number of fractional bits in the point coordinates

The function draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB( r, g, b )`.

---

cv.PolyLine (view/add comments)

Draws simple or thick polygons.

```
PolyLine(img,polys,is_closed,color,thickness=1,lineType=8,shift=0)->
None
```

polys List of lists of (x,y) pairs. Each list of points is a polygon.

img Image

is_closed Indicates whether the polylines must be drawn closed. If closed, the function draws the line from the last vertex of every contour to the first vertex.

color Polyline color

thickness Thickness of the polyline edges

lineType Type of the line segments, see Line description

shift Number of fractional bits in the vertex coordinates

The function draws single or multiple polygonal curves.

---

cv.PutText (view/add comments)

Draws a text string.

```
PutText(img,text,org,font,color)-> None
```

img Input image

text String to print

org Coordinates of the bottom-left corner of the first letter

font Pointer to the font structure

color Text color

The function renders the text in the image with the specified font and color. The printed text is clipped by the ROI rectangle. Symbols that do not belong to the specified font are replaced with the symbol for a rectangle.

---

cv.Rectangle (view/add comments)

Draws a simple, thick, or filled rectangle.

```
Rectangle(img,pt1,pt2,color,thickness=1,lineType=8,shift=0)-> None
```

img Image

pt1 One of the rectangle's vertices

pt2 Opposite rectangle vertex

color Line color (RGB) or brightness (grayscale image)

thickness Thickness of lines that make up the rectangle. Negative values, e.g., CV_FILLED, cause the function to draw a filled rectangle.

lineType Type of the line, see Line description

shift Number of fractional bits in the point coordinates

The function draws a rectangle with two opposite corners `pt1` and `pt2`.

CV_RGB (view/add comments)

Constructs a color value.

```
CV_RGB(red,grn,blu)->CvScalar
```

red Red component

grn Green component

blu Blue component

22.5 XML/YAML Persistence

---

cv.Load (view/add comments)

Loads an object from a file.

```
Load(filename,storage=NULL,name=NULL)-> generic
```

filename File name

storage Memory storage for dynamic structures, such as CvSeq or CvGraph . It is not used for matrices or images.

name Optional object name. If it is NULL, the first top-level object in the storage will be loaded.

The function loads an object from a file. It provides a simple interface to cv.Read. After the object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as a sequence, contour, or graph, one should pass a valid memory storage destination to the function.

---

cv.Save (view/add comments)

Saves an object to a file.

```
Save(filename,structPtr,name=NULL,comment=NULL)-> None
```

filename File name

structPtr Object to save

name Optional object name. If it is NULL, the name will be formed from `filename`.

comment Optional comment to put in the beginning of the file

The function saves an object to a file. It provides a simple interface to Write .

22.6 Clustering

---

cv.KMeans2 (view/add comments)

Splits set of vectors by a given number of clusters.

```
KMeans2(samples,nclusters,labels,termcrit)-> None
```

samples Floating-point matrix of input samples, one row per sample

nclusters Number of clusters to split the set by

labels Output integer vector storing cluster indices for every sample

termcrit Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

The function `cvKMeans2` implements a k-means algorithm that finds the centers of `nclusters` clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a cluster index for samples stored in the i-th row of the `samples` matrix.

22.7 Utility and System Functions and Macros

Error Handling

Errors in argument type cause a `TypeError` exception. OpenCV errors cause an `cv.error` exception.

For example a function argument that is the wrong type produces a `TypeError`:

```
>>> import cv
>>> cv.LoadImage(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument 1 must be string, not int
```

A function with the

```
>>> cv.CreateMat(-1, -1, cv.CV_8UC1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
error: Non-positive width or height
```

cv.GetTickCount (view/add comments)

Returns the number of ticks.

```
GetTickCount() -> long
```

The function returns number of the ticks starting from some platform-dependent event (number of CPU ticks from the startup, number of milliseconds from 1970th year, etc.). The function is useful for accurate measurement of a function/user-code execution time. To convert the number of ticks to time units, use cv.GetTickFrequency.

cv.GetTickFrequency (view/add comments)

Returns the number of ticks per microsecond.

```
GetTickFrequency() -> long
```

The function returns the number of ticks per microsecond. Thus, the quotient of cv.GetTickCount and cv.GetTickFrequency will give the number of microseconds starting from the platform-dependent event.

Chapter 23

imgproc. Image Processing

23.1 Histograms

CvHistogram (view/add comments)

Multi-dimensional histogram.

A CvHistogram is a multi-dimensional histogram, created by function CreateHist . It has an attribute `bins` a CvMatND containing the histogram counts.

cv.CalcBackProject (view/add comments)

Calculates the back projection.

```
CalcBackProject(image,back_project,hist)-> None
```

image Source images (though you may pass CvMat** as well)

back_project Destination back projection image of the same type as the source images

hist Histogram

The function calculates the back project of the histogram. For each tuple of pixels at the same position of all input single-channel images the function puts the value of the histogram bin, corresponding to the tuple in the destination image. In terms of statistics, the value of each output image pixel is the probability of the observed tuple given the distribution (histogram). For example, to find a red object in the picture, one may do the following:

1. Calculate a hue histogram for the red object assuming the image contains only this object. The histogram is likely to have a strong maximum, corresponding to red color.
2. Calculate back projection of a hue plane of input image where the object is searched, using the histogram. Threshold the image.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

That is the approximate algorithm of Camshift color object tracker, except for the 3rd step, instead of which CAMSHIFT algorithm is used to locate the object on the back projection given the previous object position.

---

cv.CalcBackProjectPatch (view/add comments)

Locates a template within an image by using a histogram comparison.

```
CalcBackProjectPatch(images,dst,patch_size,hist,method,factor)-> None
```

images Source images (though, you may pass CvMat** as well)

dst Destination image

patch_size Size of the patch slid though the source image

hist Histogram

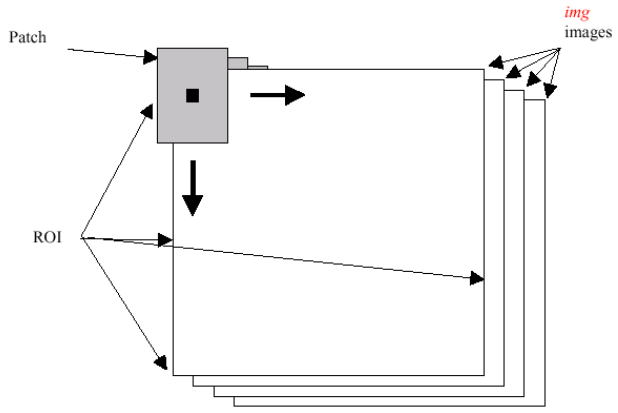
method Comparison method, passed to cv.CompareHist (see description of that function)

factor Normalization factor for histograms, will affect the normalization scale of the destination image, pass 1 if unsure

The function calculates the back projection by comparing histograms of the source image patches with the given histogram. Taking measurement results from some image at each location over ROI creates an array `image`. These results might be one or more of hue, `x` derivative, `y` derivative, Laplacian filter, oriented Gabor filter, etc. Each measurement output is collected into its own separate image. The `image` image array is a collection of these measurement images. A multi-dimensional histogram `hist` is constructed by sampling from the `image` image array. The final histogram is normalized. The `hist` histogram has as many dimensions as the number of elements in `image` array.

Each new image is measured and then converted into an `image` image array over a chosen ROI. Histograms are taken from this `image` image in an area covered by a "patch" with an anchor at center as shown in the picture below. The histogram is normalized using the parameter `norm_factor` so that it may be compared with `hist`. The calculated histogram is compared to the model histogram; `hist` uses The function `cvCompareHist` with the comparison method=`method`). The resulting output is placed at the location corresponding to the patch anchor in the probability image `dst`. This process is repeated as the patch is slid over the ROI. Iterative histogram update by subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can save a lot of operations, though it is not implemented yet.

Back Project Calculation by Patches



---

cv.CalcHist (view/add comments)

Calculates the histogram of image(s).

```
CalcHist(image,hist,accumulate=0,mask=NULL)-> None
```

image Source images (though you may pass CvMat** as well)

hist Pointer to the histogram

accumulate Accumulation flag. If it is set, the histogram is not cleared in the beginning. This feature allows user to compute a single histogram from several images, or to update the histogram online

mask The operation mask, determines what pixels of the source images are counted

The function calculates the histogram of one or more single-channel images. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input images.

```
# Calculating and displaying 2D Hue-Saturation histogram of a color image

import sys
import cv

def hs_histogram(src):
    # Convert to HSV
    hsv = cv.CreateImage(cv.GetSize(src), 8, 3)
    cv.CvtColor(src, hsv, cv.CV_BGR2HSV)

    # Extract the H and S planes
    h_plane = cv.CreateMat(src.rows, src.cols, cv.CV_8UC1)
    s_plane = cv.CreateMat(src.rows, src.cols, cv.CV_8UC1)
    cv.Split(hsv, h_plane, s_plane, None, None)
    planes = [h_plane, s_plane]

    h_bins = 30
    s_bins = 32
    hist_size = [h_bins, s_bins]
    # hue varies from 0 (~0 deg red) to 180 (~360 deg red again */
    h_ranges = [0, 180]
    # saturation varies from 0 (black-gray-white) to
    # 255 (pure spectrum color)
    s_ranges = [0, 255]
    ranges = [h_ranges, s_ranges]
    scale = 10
    hist = cv.CreateHist([h_bins, s_bins], cv.CV_HIST_ARRAY, ranges, 1)
    cv.CalcHist([cv.GetImage(i) for i in planes], hist)
    (_, max_value, _, _) = cv.GetMinMaxHistValue(hist)

    hist_img = cv.CreateImage((h_bins*scale, s_bins*scale), 8, 3)

    for h in range(h_bins):
        for s in range(s_bins):
            bin_val = cv.QueryHistValue_2D(hist, h, s)
            intensity = cv.Round(bin_val * 255 / max_value)
            cv.Rectangle(hist_img,
                         (h*scale, s*scale),
                         ((h+1)*scale - 1, (s+1)*scale - 1),
                         cv.RGB(intensity, intensity, intensity),
                         cv.CV_FILLED)
```

```
    return hist_img

if __name__ == '__main__':
    src = cv.LoadImageM(sys.argv[1])
    cv.NamedWindow("Source", 1)
    cv.ShowImage("Source", src)

    cv.NamedWindow("H-S Histogram", 1)
    cv.ShowImage("H-S Histogram", hs_histogram(src))

    cv.WaitKey(0)
```

cv.CalcProbDensity (view/add comments)

Divides one histogram by another.

```
CalcProbDensity(hist1,hist2,dst_hist,scale=255)-> None
```

hist1 first histogram (the divisor)

hist2 second histogram

dst_hist destination histogram

scale scale factor for the destination histogram

The function calculates the object probability density from the two histograms as:

$$\texttt{dist_hist}(I) = \begin{cases} 0 & \text{if } \texttt{hist1}(I) = 0 \\ \texttt{scale} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) > \texttt{hist1}(I) \\ \frac{\texttt{hist2}(I) \cdot \texttt{scale}}{\texttt{hist1}(I)} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) \leq \texttt{hist1}(I) \end{cases}$$

So the destination histogram bins are within less than `scale`.

cv.ClearHist (view/add comments)

Clears the histogram.

```
ClearHist(hist)-> None
```

hist Histogram

The function sets all of the histogram bins to 0 in the case of a dense histogram and removes all histogram bins in the case of a sparse array.

---

cv.CompareHist (view/add comments)

Compares two dense histograms.

```
CompareHist(hist1,hist2,method)->float
```

hist1 The first dense histogram

hist2 The second dense histogram

method Comparison method, one of the following:

CV_COMP_CORREL Correlation

CV_COMP_CHISQR Chi-Square

CV_COMP_INTERSECT Intersection

CV_COMP_BHATTACHARYYA Bhattacharyya distance

The function compares two dense histograms using the specified method (H_1 denotes the first histogram, H_2 the second):

Correlation (method=CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H'_1(I) \cdot H'_2(I))}{\sqrt{\sum_I (H'_1(I)^2) \cdot \sum_I (H'_2(I)^2)}}$$

where

$$H'_k(I) = \frac{H_k(I) - 1}{N \cdot \sum_J H_k(J)}$$

where N is the number of histogram bins.

Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \sum_I \frac{\sqrt{H_1(I) \cdot H_2(I)}}{\sqrt{\sum_I H_1(I) \cdot \sum_I H_2(I)}}}$$

The function returns $d(H_1, H_2)$.

Note: the method `CV_COMP_BHATTACHARYYA` only works with normalized histograms.

To compare a sparse histogram or more general sparse configurations of weighted points, consider using the cv.CalcEMD2 function.

---

cv.CreateHist (view/add comments)

Creates a histogram.

```
CreateHist(dims, type, ranges, uniform = 1) -> hist
```

dims for an N-dimensional histogram, list of length N giving the size of each dimension

type Histogram representation format: `CV_HIST_ARRAY` means that the histogram data is represented as a multi-dimensional dense array CvMatND; `CV_HIST_SPARSE` means that histogram data is represented as a multi-dimensional sparse array CvSparseMat

ranges Array of ranges for the histogram bins. Its meaning depends on the `uniform` parameter value. The ranges are used for when the histogram is calculated or backprojected to determine which histogram bin corresponds to which value/tuple of values from the input image(s)

uniform Uniformity flag; if not 0, the histogram has evenly spaced bins and for every $0 <= i < cDims$ `ranges[i]` is an array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is then split into `dims[i]` equal parts to determine the `i-th` input tuple value ranges for every histogram bin. And if `uniform=0`, then `i-th` element of `ranges` array contains `dims[i]+1` elements: $\texttt{lower}_0, \texttt{upper}_0, \texttt{lower}_1, \texttt{upper}_1 = \texttt{lower}_2, ...\texttt{upper}_{dims[i]-1}$ where $\texttt{lower}_j$ and $\texttt{upper}_j$ are lower and upper boundaries of `i-th` input tuple value for `j-th` bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin are not counted by cv.CalcHist and filled with 0 by cv.CalcBackProject

The function creates a histogram of the specified size and returns a pointer to the created histogram. If the array `ranges` is 0, the histogram bin ranges must be specified later via the function cv.SetHistBinRanges. Though cv.CalcHist and cv.CalcBackProject may process 8-bit images without setting bin ranges, they assume thy are equally spaced in 0 to 255 bins.

---

cv.GetMinMaxHistValue (view/add comments)

Finds the minimum and maximum histogram bins.

```
GetMinMaxHistValue(hist)-> (min_value,max_value,min_idx,max_idx)
```

hist Histogram

min_value Minimum value of the histogram

max_value Maximum value of the histogram

min_idx Coordinates of the minimum

max_idx Coordinates of the maximum

The function finds the minimum and maximum histogram bins and their positions. All of output arguments are optional. Among several extremas with the same value the ones with the minimum index (in lexicographical order) are returned. In the case of several maximums or minimums, the earliest in lexicographical order (extrema locations) is returned.

cv.NormalizeHist (view/add comments)

Normalizes the histogram.

```
NormalizeHist(hist,factor)-> None
```

hist Pointer to the histogram

factor Normalization factor

The function normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor`.

QueryHistValue_1D (view/add comments)

Returns the value from a 1D histogram bin.

```
QueryHistValue_1D(hist, idx0) -> float
```

hist Histogram

idx0 bin index 0

QueryHistValue_2D (view/add comments)

Returns the value from a 2D histogram bin.

```
QueryHistValue_2D(hist, idx0, idx1) -> float
```

hist Histogram

idx0 bin index 0

idx1 bin index 1

QueryHistValue_3D (view/add comments)

Returns the value from a 3D histogram bin.

```
QueryHistValue_3D(hist, idx0, idx1, idx2) -> float
```

hist Histogram

idx0 bin index 0

idx1 bin index 1

idx2 bin index 2

QueryHistValue_nD (view/add comments)

Returns the value from a 1D histogram bin.

```
QueryHistValue_nD(hist, idx) -> float
```

hist Histogram

idx list of indices, of same length as the dimension of the histogram's bin.

cv.ThreshHist (view/add comments)

Thresholds the histogram.

```
ThreshHist(hist,threshold)-> None
```

hist Pointer to the histogram

threshold Threshold level

The function clears histogram bins that are below the specified threshold.

23.2 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as cv.'s), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y). It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc.

IplConvKernel (view/add comments)

An IplConvKernel is a rectangular convolution kernel, created by function CreateStructuringElementEx .

---

cv.CopyMakeBorder (view/add comments)

Copies an image and makes a border around it.

```
CopyMakeBorder(src,dst,offset,bordertype,value=(0,0,0,0))-> None
```

src The source image

dst The destination image

offset Coordinates of the top-left corner (or bottom-left in the case of images with bottom-left origin) of the destination image rectangle where the source image (or its ROI) is copied. Size of the rectanlge matches the source image size/ROI size

bordertype Type of the border to create around the copied source image rectangle; types include:

IPL_BORDER_CONSTANT border is filled with the fixed value, passed as last parameter of the function.

IPL_BORDER_REPLICATE the pixels from the top and bottom rows, the left-most and right-most columns are replicated to fill the border.

(The other two border types from IPL, `IPL_BORDER_REFLECT` and `IPL_BORDER_WRAP`, are currently unsupported)

value Value of the border pixels if `bordertype` is `IPL_BORDER_CONSTANT`

The function copies the source 2D array into the interior of the destination array and makes a border of the specified type around the copied area. The function is useful when one needs to emulate border type that is different from the one embedded into a specific algorithm implementation. For example, morphological functions, as well as most of other filtering functions in OpenCV, internally use replication border type, while the user may need a zero border or a border, filled with 1's or 255's.

---

cv.CreateStructuringElementEx (view/add comments)

Creates a structuring element.

```
CreateStructuringElementEx(cols,rows,anchorX,anchorY,shape,values=None)->
kernel
```

cols Number of columns in the structuring element

rows Number of rows in the structuring element

anchorX Relative horizontal offset of the anchor point

anchorY Relative vertical offset of the anchor point

shape Shape of the structuring element; may have the following values:

CV_SHAPE_RECT a rectangular element

CV_SHAPE_CROSS a cross-shaped element

CV_SHAPE_ELLIPSE an elliptic element

CV_SHAPE_CUSTOM a user-defined element. In this case the parameter `values` specifies the mask, that is, which neighbors of the pixel must be considered

values Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non-zero values indicate points that belong to the element. If the pointer is `NULL`, then all values are considered non-zero, that is, the element is of a rectangular shape. This parameter is considered only if the shape is `CV_SHAPE_CUSTOM`

The function CreateStructuringElementEx allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the morphological operations.

---

cv.Dilate (view/add comments)

Dilates an image by using a specific structuring element.

```
Dilate(src,dst,element=None,iterations=1)-> None
```

src Source image

dst Destination image

element Structuring element used for dilation. If it is `None`, a 3×3 rectangular structuring element is used

iterations Number of times dilation is applied

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\max_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. For color images, each channel is processed independently.

cv.Erode (view/add comments)

Erodes an image by using a specific structuring element.

```
Erode(src,dst,element=None,iterations=1)-> None
```

src Source image

dst Destination image

element Structuring element used for erosion. If it is `None`, a 3×3 rectangular structuring element is used

iterations Number of times erosion is applied

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\min_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. For color images, each channel is processed independently.

cv.Filter2D (view/add comments)

Convolves an image with the kernel.

```
Filter2D(src,dst,kernel,anchor=(-1,-1))-> None
```

src The source image

dst The destination image

kernel Convolution kernel, a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cv.Split and process them individually

anchor The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor shoud lie within the kernel. The special default value (-1,-1) means that it is at the kernel center

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values from the nearest pixels that are inside the image.

---

cv.Laplace (view/add comments)

Calculates the Laplacian of an image.

```
Laplace(src,dst,apertureSize=3)-> None
```

src Source image

dst Destination image

apertureSize Aperture size (it has the same meaning as cv.Sobel)

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst}(x,y) = \frac{d^2\texttt{src}}{dx^2} + \frac{d^2\texttt{src}}{dy^2}$$

Setting `apertureSize` = 1 gives the fastest variant that is equal to convolving the image with the following kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Similar to the cv.Sobel function, no scaling is done and the same combinations of input and output formats are supported.

cv.MorphologyEx (view/add comments)

Performs advanced morphological transformations.

```
MorphologyEx(src,dst,temp,element,operation,iterations=1)-> None
```

src Source image

dst Destination image

temp Temporary image, required in some cases

element Structuring element

operation Type of morphological operation, one of the following:

- **CV_MOP_OPEN** opening
- **CV_MOP_CLOSE** closing
- **CV_MOP_GRADIENT** morphological gradient
- **CV_MOP_TOPHAT** "top hat"
- **CV_MOP_BLACKHAT** "black hat"

iterations Number of times erosion and dilation are applied

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$dst = open(src, element) = dilate(erode(src, element), element)$$

Closing:

$$dst = close(src, element) = erode(dilate(src, element), element)$$

Morphological gradient:

$$dst = morph_grad(src, element) = dilate(src, element) - erode(src, element)$$

"Top hat":

$$dst = tophat(src, element) = src - open(src, element)$$

"Black hat":

$$dst = blackhat(src, element) = close(src, element) - src$$

The temporary image `temp` is required for a morphological gradient and, in the case of in-place operation, for "top hat" and "black hat".

---

cv.PyrDown (view/add comments)

Downsamples an image.

```
PyrDown(src,dst,filter=CV_GAUSSIAN_5X5)-> None
```

src The source image

dst The destination image, should have a half as large width and height than the source

filter Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported

The function performs the downsampling step of the Gaussian pyramid decomposition. First it convolves the source image with the specified filter and then downsamples the image by rejecting even rows and columns.

---

cv.Smooth (view/add comments)

Smooths the image in one of several ways.

```
Smooth(src,dst,smoothtype=CV_GAUSSIAN,param1=3,param2=0,param3=0,param4=0)->
None
```

src The source image

dst The destination image

smoothtype Type of the smoothing:

- **CV_BLUR_NO_SCALE** linear convolution with param1 × param2 box kernel (all 1's). If you want to smooth different pixels with different-size box kernels, you can use the integral image that is computed using cv.Integral
- **CV_BLUR** linear convolution with param1 × param2 box kernel (all 1's) with subsequent scaling by $1/(\texttt{param1} \cdot \texttt{param2})$
- **CV_GAUSSIAN** linear convolution with a param1 × param2 Gaussian kernel
- **CV_MEDIAN** median filter with a param1 × param1 square aperture
- **CV_BILATERAL** bilateral filter with a param1×param1 square aperture, color sigma=param3 and spatial sigma=param4. If param1=0, the aperture square side is set to cvRound(param4*1.5)* Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1 The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)

param2 The second parameter of the smoothing operation, the aperture height. Ignored by CV_MEDIAN and CV_BILATERAL methods. In the case of simple scaled/non-scaled and Gaussian blur if param2 is zero, it is set to param1. Otherwise it must be a positive odd number.

param3 In the case of a Gaussian parameter this parameter may specify Gaussian σ (standard deviation). If it is zero, it is calculated from the kernel size:

$$\sigma = 0.3(n/2 - 1) + 0.8 \quad \text{where} \quad n = \begin{array}{l} \texttt{param1} \text{ for horizontal kernel} \\ \texttt{param2} \text{ for vertical kernel} \end{array}$$

Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If param3 is not zero, while param1 and param2 are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function smooths an image using one of several methods. Every of the methods has some features and restrictions listed below

Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to cv.Sobel and cv.Laplace) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

cv.Sobel (view/add comments)

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator.

```
Sobel(src,dst,xorder,yorder,apertureSize = 3)-> None
```

src Source image of type CvArr*

dst Destination image

xorder Order of the derivative x

yorder Order of the derivative y

apertureSize Size of the extended Sobel kernel, must be 1, 3, 5 or 7

In all cases except 1, an `apertureSize` × `apertureSize` separable kernel will be used to calculate the derivative. For `apertureSize` $= 1$ a 3×1 or 1×3 a kernel is used (Gaussian smoothing is not done). There is also the special value `CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\texttt{dst}(x,y) = \frac{d^{xorder+yorder}\texttt{src}}{dx^{xorder} \cdot dy^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `apertureSize` = 3) or (`xorder` = 0, `yorder` = 1, `apertureSize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and the second one corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

or a kernel of:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix}$$

depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger numbers (in absolute values) than the source image does. To avoid overflow, the function requires a 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using the cv.ConvertScale or the cv.ConvertScaleAbs function. Besides 8-bit images the function can process 32-bit floating-point images. Both the source and the destination must be single-channel images of equal size or equal ROI size.

23.3 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x, y) = \texttt{src}(f_x(x, y), f_y(x, y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic cv.Remap and to the simplest and the fastest cv.Resize, need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x, y) one of $f_x(x, y)$ or $f_y(x, y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method `BORDER_TRANSPARENT`, which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x,y)$ and $f_y(x,y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x,y), f_y(x,y))$ and then the value of the polynomial at $(f_x(x,y), f_y(x,y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see cv.Resize.

---

cv.GetRotationMatrix2D (view/add comments)

Calculates the affine matrix of 2d rotation.

```
GetRotationMatrix2D(center,angle,scale,mapMatrix)-> None
```

center Center of the rotation in the source image

angle The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)

scale Isotropic scale factor

mapMatrix Pointer to the destination 2×3 matrix

The function `cv2DRotationMatrix` calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} - (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot cos(\texttt{angle}), \beta = \texttt{scale} \cdot sin(\texttt{angle})$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

cv.GetAffineTransform (view/add comments)

Calculates the affine transform from 3 corresponding points.

```
GetAffineTransform(src,dst,mapMatrix)-> None
```

src Coordinates of 3 triangle vertices in the source image

dst Coordinates of the 3 corresponding triangle vertices in the destination image

mapMatrix Pointer to the destination 2×3 matrix

The function cvGetAffineTransform calculates the matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

cv.GetPerspectiveTransform (view/add comments)

Calculates the perspective transform from 4 corresponding points.

```
GetPerspectiveTransform(src,dst,mapMatrix)-> None
```

src Coordinates of 4 quadrangle vertices in the source image

dst Coordinates of the 4 corresponding quadrangle vertices in the destination image

mapMatrix Pointer to the destination 3×3 matrix

The function `cvGetPerspectiveTransform` calculates a matrix of perspective transforms such that:

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x_i', y_i'), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

---

cv.GetQuadrangleSubPix (view/add comments)

Retrieves the pixel quadrangle from an image with sub-pixel accuracy.

```
GetQuadrangleSubPix(src,dst,mapMatrix)-> None
```

src Source image

dst Extracted quadrangle

mapMatrix The transformation 2×3 matrix $[A|b]$ (see the discussion)

The function `cvGetQuadrangleSubPix` extracts pixels from `src` at sub-pixel accuracy and stores them to `dst` as follows:

$$dst(x, y) = src(A_{11}x' + A_{12}y' + b_1, A_{21}x' + A_{22}y' + b_2)$$

where

$$x' = x - \frac{(width(dst) - 1)}{2}, y' = y - \frac{(height(dst) - 1)}{2}$$

and

$$\texttt{mapMatrix} = \begin{bmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{bmatrix}$$

The values of pixels at non-integer coordinates are retrieved using bilinear interpolation. When the function needs pixels outside of the image, it uses replication border mode to reconstruct the values. Every channel of multiple-channel images is processed independently.

cv.GetRectSubPix (view/add comments)

Retrieves the pixel rectangle from an image with sub-pixel accuracy.

```
GetRectSubPix(src,dst,center)-> None
```

src Source image

dst Extracted rectangle

center Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image

The function `cvGetRectSubPix` extracts pixels from `src`:

$$dst(x,y) = src(x + \texttt{center.x} - (width(\texttt{dst}) - 1) * 0.5, y + \texttt{center.y} - (height(\texttt{dst}) - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

cv.LogPolar (view/add comments)

Remaps an image to log-polar space.

```
LogPolar(src,dst,center,M,flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS)->
None
```

src Source image

dst Destination image

center The transformation center; where the output precision is maximal

M Magnitude scale parameter. See below

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero

CV_WARP_INVERSE_MAP See below

The function `cvLogPolar` transforms the source image using the following transformation: Forward transformation (`CV_WARP_INVERSE_MAP` is not set):

$$dst(\phi, \rho) = src(x, y)$$

Inverse transformation (`CV_WARP_INVERSE_MAP` is set):

$$dst(x, y) = src(\phi, \rho)$$

where

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

---

cv.Remap (view/add comments)

Applies a generic geometrical transformation to the image.

```
Remap(src,dst,mapx,mapy,flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS,fillval=(0,0,0,0))
None
```

src Source image

dst Destination image

mapx The map of x-coordinates (CV_32FC1 image)

mapy The map of y-coordinates (CV_32FC1 image)

flags A combination of interpolation method and the following optional flag(s):

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`

fillval A value used to fill outliers

The function cvRemap transforms the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(\texttt{mapx}(x,y),\texttt{mapy}(x,y))$$

Similar to other geometrical transformations, some interpolation method (specified by user) is used to extract pixels with non-integer coordinates. Note that the function can not operate in-place.

---

cv.Resize (view/add comments)

Resizes an image.

```
Resize(src,dst,interpolation=CV_INTER_LINEAR)-> None
```

src Source image

dst Destination image

interpolation Interpolation method:

- **CV_INTER_NN** nearest-neigbor interpolation
- **CV_INTER_LINEAR** bilinear interpolation (used by default)
- **CV_INTER_AREA** resampling using pixel area relation. It is the preferred method for image decimation that gives moire-free results. In terms of zooming it is similar to the CV_INTER_NN method
- **CV_INTER_CUBIC** bicubic interpolation

The function cvResize resizes an image src so that it fits exactly into dst. If ROI is set, the function considers the ROI as supported.

---

cv.WarpAffine (view/add comments)

Applies an affine transformation to an image.

```
WarpAffine(src,dst,mapMatrix,flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,fillval=(0,0
None
```

src Source image

dst Destination image

mapMatrix 2×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`

CV_WARP_INVERSE_MAP indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpAffine` transforms the source image using the specified matrix:

$$dst(x', y') = src(x, y)$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV_WARP_INVERSE_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

The function is similar to cv.GetQuadrangleSubPix but they are not exactly the same. cv.WarpAffine requires input and output image have the same data type, has larger overhead (so it is not quite suitable for small images) and can leave part of destination image unchanged. While cv.GetQuadrangleSubPix may extract quadrangles from 8-bit images into floating-point buffer, has smaller overhead and always changes the whole destination image content. Note that the function can not operate in-place.

To transform a sparse set of points, use the cv.Transform function from cxcore.

---

cv.WarpPerspective (view/add comments)

Applies a perspective transformation to an image.

```
WarpPerspective(src,dst,mapMatrix,flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS,fillval
None
```

src Source image

dst Destination image

mapMatrix 3×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`

CV_WARP_INVERSE_MAP indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpPerspective` transforms the source image using the specified matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV_WARP_INVERSE_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

Note that the function can not operate in-place. For a sparse set of points use the cv.PerspectiveTransform function from CxCore.

23.4 Miscellaneous Image Transformations

---

cv.AdaptiveThreshold (view/add comments)

Applies an adaptive threshold to an array.

```
AdaptiveThreshold(src,dst,maxValue, adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C,
thresholdType=CV_THRESH_BINARY,blockSize=3,param1=5)-> None
```

src Source image

dst Destination image

maxValue Maximum value that is used with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV`

adaptive_method Adaptive thresholding algorithm to use: `CV_ADAPTIVE_THRESH_MEAN_C` or `CV_ADAPTIVE_THRESH_GAUSSIAN_C` (see the discussion)

thresholdType Thresholding type; must be one of

CV_THRESH_BINARY xxx

CV_THRESH_BINARY_INV xxx

blockSize The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

param1 The method-dependent parameter. For the methods `CV_ADAPTIVE_THRESH_MEAN_C` and `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a constant subtracted from the mean or weighted mean (see the discussion), though it may be negative

The function transforms a grayscale image to a binary image according to the formulas:

CV_THRESH_BINARY

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

For the method `CV_ADAPTIVE_THRESH_MEAN_C` it is the mean of a `blockSize` $\times$ `blockSize` pixel neighborhood, minus `param1`.

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is the weighted sum (gaussian) of a `blockSize` $\times$ `blockSize` pixel neighborhood, minus `param1`.

cv.CvtColor (view/add comments)

Converts an image from one color space to another.

```
CvtColor(src,dst,code)-> None
```

src The source 8-bit (8u), 16-bit (16u) or single-precision floating-point (32f) image

dst The destination image of the same data type as the source. The number of channels may be different

code Color conversion operation that can be specifed using `CV_` *src_color_space* `2` *dst_color_space* constants (see below)

The function converts the input image from one color space to another. The function ignores the `colorModel` and `channelSeq` fields of the `IplImage` header, so the source image color space should be specified correctly (including order of the channels in the case of RGB space. For example, BGR means 24-bit format with $B_0, G_0, R_0, B_1, G_1, R_1, ...$ layout whereas RGB means 24-format with $R_0, G_0, B_0, R_1, G_1, B_1, ...$ layout).

The conventional range for R,G,B channel values is:

- 0 to 255 for 8-bit images
- 0 to 65535 for 16-bit images and
- 0 to 1 for floating-point images.

Of course, in the case of linear transformations the range can be specific, but in order to get correct results in the case of non-linear transformations, the input image should be scaled.

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

```
cvCvtColor(src ,bwsrc, CV_RGB2GRAY)
```

- RGB ↔ CIE XYZ.Rec 709 with D65 white point (CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB ↔ YCrCb JPEG (a.k.a. YCC) (CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

  where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

  Y, Cr and Cb cover the whole value range.

- RGB ↔ HSV (CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V=R \\ 120+60(B-R)/S & \text{if } V=G \\ 240+60(R-G)/S & \text{if } V=B \end{cases}$$

if $H<0$ then $H \leftarrow H+360$

On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R,G,B)$$

$$V_{min} \leftarrow min(R,G,B)$$

$$L \leftarrow \frac{V_{max}+V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max}-V_{min}}{V_{max}+V_{min}} & \text{if } L<0.5 \\ \frac{V_{max}-V_{min}}{2-(V_{max}+V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V_{max}=R \\ 120+60(B-R)/S & \text{if } V_{max}=G \\ 240+60(R-G)/S & \text{if } V_{max}=B \end{cases}$$

if $H<0$ then $H \leftarrow H+360$ On output $0 \leq L \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \le 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t <= 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \le L \le 100$, $-127 \le a \le 127$, $-127 \le b \le 127$

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow L * 255/100, a \leftarrow a + 128, b \leftarrow b + 128$$

16-bit images currently not supported

32-bit images L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y <= 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$
$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$
$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow 255/100L, u \leftarrow 255/354(u + 134), v \leftarrow 255/256(v + 140)$$

16-bit images currently not supported

32-bit images L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Ford98 at the Charles Poynton site.

- Bayer $\rightarrow$ RGB (`CV_BayerBG2BGR`, `CV_BayerGB2BGR`, `CV_BayerRG2BGR`, `CV_BayerGR2BGR`, `CV_BayerBG2RGB`, `CV_BayerGB2RGB`, `CV_BayerRG2RGB`, `CV_BayerGR2RGB`) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{matrix} R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants `CV_Bayer` C_1C_2 `2BGR` and `CV_Bayer` C_1C_2 `2RGB` indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

---

cv.DistTransform (view/add comments)

Calculates the distance to the closest zero pixel for all non-zero pixels of the source image.

```
DistTransform(src,dst,distance_type=CV_DIST_L2,mask_size=3,mask=None,labels=NULL)->
None
```

src 8-bit, single-channel (binary) source image

dst Output image with calculated distances (32-bit floating-point, single-channel)

distance_type Type of distance; can be CV_DIST_L1, CV_DIST_L2, CV_DIST_C or CV_DIST_USER

mask_size Size of the distance transform mask; can be 3 or 5. in the case of CV_DIST_L1 or CV_DIST_C the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 yet it is faster

mask User-defined mask in the case of a user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in the case ofa 3×3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight's move cost) in the case of a 5×5 mask

labels The optional output 2d array of integer type labels, the same size as src and dst

The function calculates the approximated distance from every binary image pixel to the nearest zero pixel. For zero pixels the function sets the zero distance, for others it finds the shortest path consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (that is denoted as a), all the diagonal shifts must have the same cost (denoted b), and all knight's moves must have the same cost (denoted c). For CV_DIST_C and CV_DIST_L1 types the distance is calculated precisely, whereas for CV_DIST_L2 (Euclidian distance) the distance can be calculated only with some relative error (a 5×5 mask gives more accurate results), OpenCV uses the values suggested in [**?**]:

CV_DIST_C	(3×3)	a = 1, b = 1
CV_DIST_L1	(3×3)	a = 1, b = 2
CV_DIST_L2	(3×3)	a=0.955, b=1.3693
CV_DIST_L2	(5×5)	a=1, b=1.4, c=2.1969

And below are samples of the distance field (black (0) pixel is in the middle of white square) in the case of a user-defined distance:

User-defined 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1		1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

User-defined 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1		1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

Typically, for a fast, coarse distance estimation `CV_DIST_L2`, a 3×3 mask is used, and for a more accurate distance estimation `CV_DIST_L2`, a 5×5 mask is used.

When the output parameter `labels` is not `NULL`, for every non-zero pixel the function also finds the nearest connected component consisting of zero pixels. The connected components themselves are found as contours in the beginning of the function.

In this mode the processing time is still O(N), where N is the number of pixels. Thus, the function provides a very fast way to compute approximate Voronoi diagram for the binary image.

CvConnectedComp (view/add comments)

Connected component, represented as a tuple (area, value, rect), where area is the area of the component as a float, value is the average color as a CvScalar , and rect is the ROI of the component, as a CvRect .

---

cv.FloodFill (view/add comments)

Fills a connected component with the given color.

```
FloodFill(image,seed_point,new_val,lo_diff=(0,0,0,0),up_diff=(0,0,0,0),flags=4,mask=
comp
```

image Input 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the `CV_FLOODFILL_MASK_ONLY` flag is set (see below)

seed_point The starting point

new_val New value of the repainted domain pixels

lo_diff Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

up_diff Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

comp Returned connected component for the repainted domain. Note that the function does not fill `comp->contour` field. The boundary of the filled component can be retrieved from the output mask image using cv.FindContours

flags The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:

CV_FLOODFILL_FIXED_RANGE if set, the difference between the current pixel and seed pixel is considered, otherwise the difference between neighbor pixels is considered (the range is floating)

CV_FLOODFILL_MASK_ONLY if set, the function does not fill the image (`new_val` is ignored), but fills the mask (that must be non-NULL in this case)

mask Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller than `image`. If not NULL, the function uses and updates the mask, so the user takes responsibility of initializing the `mask` content. Floodfilling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. It is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap. **Note**: because the mask is larger than the filled image, a pixel in `mask` that corresponds to (x, y) pixel in `image` will have coordinates $(x + 1, y + 1)$

The function fills a connected component starting from the seed point with the specified color. The connectivity is determined by the closeness of pixel values. The pixel at (x, y) is considered to belong to the repainted domain if:

grayscale image, floating range

$$src(x', y') - \texttt{lo_diff} <= src(x, y) <= src(x', y') + \texttt{up_diff}$$

grayscale image, fixed range

$$src(seed.x, seed.y) - \texttt{lo_diff} <= src(x, y) <= src(seed.x, seed.y) + \texttt{up_diff}$$

color image, floating range

$$src(x', y')_r - \texttt{lo_diff}_r <= src(x, y)_r <= src(x', y')_r + \texttt{up_diff}_r$$

$$src(x', y')_g - \texttt{lo_diff}_g <= src(x, y)_g <= src(x', y')_g + \texttt{up_diff}_g$$

$$src(x', y')_b - \texttt{lo_diff}_b <= src(x, y)_b <= src(x', y')_b + \texttt{up_diff}_b$$

color image, fixed range

$$src(seed.x, seed.y)_r - \texttt{lo_diff}_r <= src(x, y)_r <= src(seed.x, seed.y)_r + \texttt{up_diff}_r$$

$$src(seed.x, seed.y)_g - \texttt{lo_diff}_g <= src(x, y)_g <= src(seed.x, seed.y)_g + \texttt{up_diff}_g$$

$$src(seed.x, seed.y)_b - \texttt{lo_diff}_b <= src(x, y)_b <= src(seed.x, seed.y)_b + \texttt{up_diff}_b$$

where $src(x', y')$ is the value of one of pixel neighbors. That is, to be added to the connected component, a pixel's color/brightness should be close enough to the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

---

cv.Inpaint (view/add comments)

Inpaints the selected region in the image.

```
Inpaint(src,mask,dst,inpaintRadius,flags) -> None
```

src The input 8-bit 1-channel or 3-channel image.

mask The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.

dst The output image of the same format and the same size as input.

inpaintRadius The radius of circlular neighborhood of each point inpainted that is considered by the algorithm.

flags The inpainting method, one of the following:

CV_INPAINT_NS Navier-Stokes based method.

CV_INPAINT_TELEA The method by Alexandru Telea [**?**]

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video.

---

cv.Integral (view/add comments)

Calculates the integral of an image.

```
Integral(image,sum,sqsum=NULL,tiltedSum=NULL)-> None
```

image The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

sum The integral image, $(W+1) \times (H+1)$, 32-bit integer or double precision floating-point (64f)

sqsum The integral image for squared pixel values, $(W+1) \times (H+1)$, double precision floating-point (64f)

tiltedSum The integral for the image rotated by 45 degrees, $(W+1) \times (H+1)$, the same data type as `sum`

The function calculates one or more integral images for the source image as following:

$$\texttt{sum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)$$

$$\texttt{sqsum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)^2$$

$$\texttt{tiltedSum}(X,Y) = \sum_{y<Y,abs(x-X+1)\leq Y-y-1} \texttt{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum_{x_1<=x<x_2,\, y_1<=y<y_2} = \mathrm{sum}(x_2,y_2) - \mathrm{sum}(x_1,y_2) - \mathrm{sum}(x_2,y_1) + \mathrm{sum}(x_1,x_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

---

cv.PyrMeanShiftFiltering (view/add comments)

Does meanshift image segmentation

```
PyrMeanShiftFiltering(src,dst,sp,sr,max_level=1,termcrit=(CV_TERMCRIT_ITER+CV_TERMCR
None
```

src The source 8-bit, 3-channel image.

dst The destination image of the same format and the same size as the source.

sp The spatial window radius.

sr The color window radius.

max_level Maximum level of the pyramid for the segmentation.

termcrit Termination criteria: when to stop meanshift iterations.

The function implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X,Y) neighborhood in the joint space-color hyperspace is considered:

$$(x,y) : X - \mathrm{sp} \le x \le X + \mathrm{sp}, Y - \mathrm{sp} \le y \le Y + \mathrm{sp}, ||(R,G,B) - (r,g,b)|| \le \mathrm{sr}$$

where `(R,G,B)` and `(r,g,b)` are the vectors of color components at `(X,Y)` and `(x,y)`, respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value `(X',Y')` and

average color vector `(R',G',B')` are found and they act as the neighborhood center on the next iteration:

(X,Y) (X',Y'), (R,G,B) (R',G',B').

After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration):

$I(X,Y) <- (R*,G*,B*)$

Then `max_level` > 0 , the gaussian pyramid of `max_level` $+ 1$ levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much ($>$ `sr`) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when `max_level` $== 0$).

---

cv.PyrSegmentation (view/add comments)

Implements image segmentation by pyramids.

```
PyrSegmentation(src,dst,storage,level,threshold1,threshold2)-> comp
```

src The source image

dst The destination image

storage Storage; stores the resulting sequence of connected components

comp Pointer to the output sequence of the segmented components

level Maximum level of the pyramid for the segmentation

threshold1 Error threshold for establishing the links

threshold2 Error threshold for the segments clustering

The function implements image segmentation by pyramids. The pyramid builds up to the level `level`. The links between any pixel `a` on level `i` and its candidate father pixel `b` on the adjacent level are established if $p(c(a),c(b)) < threshold1$. After the connected components are defined, they are joined into several clusters. Any two segments A and B belong to the same cluster, if

$p(c(A), c(B)) < threshold2$. If the input image has only one channel, then $p(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then

$$p(c^1, c^2) = 0.30(c^1_r - c^2_r) + 0.59(c^1_g - c^2_g) + 0.11(c^1_b - c^2_b).$$

There may be more than one connected component per a cluster. The images `src` and `dst` should be 8-bit single-channel or 3-channel images or equal size.

---

cv.Threshold (view/add comments)

Applies a fixed-level threshold to array elements.

```
Threshold(src,dst,threshold,maxValue,thresholdType)-> None
```

src Source array (single-channel, 8-bit or 32-bit floating point)

dst Destination array; must be either the same type as `src` or 8-bit

threshold Threshold value

maxValue Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types

thresholdType Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (cv.CmpS could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by `thresholdType`:

CV_THRESH_BINARY

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxValue} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

CV_THRESH_TRUNC

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

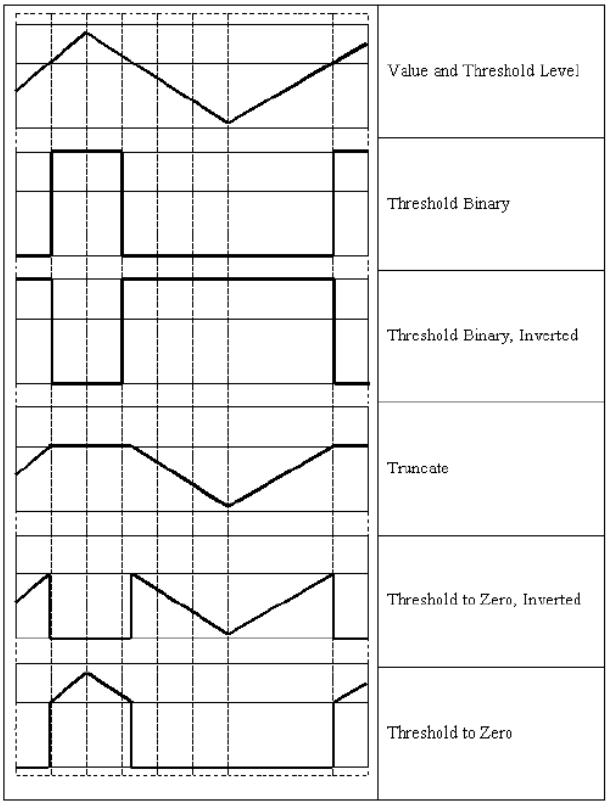
CV_THRESH_TOZERO

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_TOZERO_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Also, the special value `CV_THRESH_OTSU` may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified `thresh`. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.



23.5 Structural Analysis and Shape Descriptors

cv.ApproxChains (view/add comments)

Approximates Freeman chain(s) with a polygonal curve.

```
ApproxChains(src_seq,storage,method=CV_CHAIN_APPROX_SIMPLE,parameter=0,minimal_perime
chains
```

src_seq Pointer to the chain that can refer to other chains

storage Storage location for the resulting polylines

method Approximation method (see the description of the function cv.FindContours)

parameter Method parameter (not used now)

minimal_perimeter Approximates only those contours whose perimeters are not less than `minimal_perimeter`. Other chains are removed from the resulting structure

recursive If not 0, the function approximates all chains that access can be obtained to from `src_seq` by using the `h_next` or `v_next links`. If 0, the single chain is approximated

This is a stand-alone approximation routine. The function `cvApproxChains` works exactly in the same way as cv.FindContours with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via the `v_next` or `h_next` fields of the returned structure.

cv.ApproxPoly (view/add comments)

Approximates polygonal curve(s) with the specified precision.

```
ApproxPoly(src_seq, storage, method, parameter=0, parameter2=0) ->
sequence
```

src_seq Sequence of an array of points

storage Container for the approximated contours. If it is NULL, the input sequences' storage is used

method Approximation method; only CV_POLY_APPROX_DP is supported, that corresponds to the Douglas-Peucker algorithm

parameter Method-specific parameter; in the case of CV_POLY_APPROX_DP it is a desired approximation accuracy

parameter2 If case if src_seq is a sequence, the parameter determines whether the single sequence should be approximated or all sequences on the same level or below src_seq (see cv.FindContours for description of hierarchical contour structures). If src_seq is an array CvMat* of points, the parameter specifies whether the curve is closed (parameter2!=0) or not (parameter2 =0)

The function approximates one or more curves and returns the approximation result[s]. In the case of multiple curves, the resultant tree will have the same structure as the input one (1:1 correspondence).

---

cv.ArcLength (view/add comments)

Calculates the contour perimeter or the curve length.

```
ArcLength(curve,slice=CV_WHOLE_SEQ,isClosed=-1)-> double
```

curve Sequence or array of the curve points

slice Starting and ending points of the curve, by default, the whole curve length is calculated

isClosed Indicates whether the curve is closed or not. There are 3 cases:

- isClosed $= 0$ the curve is assumed to be unclosed.
- isClosed > 0 the curve is assumed to be closed.
- isClosed < 0 if curve is sequence, the flag CV_SEQ_FLAG_CLOSED of ((CvSeq*)curve)->flags is checked to determine if the curve is closed or not, otherwise (curve is represented by array (CvMat*) of points) it is assumed to be unclosed.

The function calculates the length or curve as the sum of lengths of segments between subsequent points

cv.BoundingRect (view/add comments)

Calculates the up-right bounding rectangle of a point set.

```
BoundingRect(points,update=0)-> CvRect
```

points 2D point set, either a sequence or vector (`CvMat`) of points

update The update flag. See below.

The function returns the up-right bounding rectangle for a 2d point set. Here is the list of possible combination of the flag values and type of `points`:

update	points	action
0	`CvContour`	the bounding rectangle is not calculated, but it is taken from `rect` field of the contour header.
1	`CvContour`	the bounding rectangle is calculated and written to `rect` field of the contour header.
0	`CvSeq` or `CvMat`	the bounding rectangle is calculated and returned.
1	`CvSeq` or `CvMat`	runtime error is raised.

cv.BoxPoints (view/add comments)

Finds the box vertices.

```
BoxPoints(box)-> points
```

box Box

points Array of vertices

The function calculates the vertices of the input 2d box.

cv.CalcPGH (view/add comments)

Calculates a pair-wise geometrical histogram for a contour.

```
CalcPGH(contour,hist)-> None
```

contour Input contour. Currently, only integer point coordinates are allowed

hist Calculated histogram; must be two-dimensional

The function calculates a 2D pair-wise geometrical histogram (PGH), described in cv.livarinen97 for the contour. The algorithm considers every pair of contour edges. The angle between the edges and the minimum/maximum distances are determined for every pair. To do this each of the edges in turn is taken as the base, while the function loops through all the other edges. When the base edge and any other edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented (that is, the histogram is transposed relatively to the cv.livarninen97 definition). The histogram can be used for contour matching.

cv.CalcEMD2 (view/add comments)

Computes the ”minimal work” distance between two weighted point configurations.

```
CalcEMD2(signature1, signature2, distance_type, distance_func = None,
cost_matrix=None, flow=None, lower_bound=None, userdata = None) -> float
```

signature1 First signature, a $\texttt{size1} \times \texttt{dims} + 1$ floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used

signature2 Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different, in this case an extra ”dummy” point is added to either `signature1` or `signature2`

distance_type Metrics used; CV_DIST_L1, CV_DIST_L2, and CV_DIST_C stand for one of the standard metrics; CV_DIST_USER means that a user-defined function distance_func or pre-calculated cost_matrix is used

distance_func The user-supplied distance function. It takes coordinates of two points pt0 and pt1, and returns the distance between the points, with sigature func(pt0, pt1, userdata) -> float

cost_matrix The user-defined size1 × size2 cost matrix. At least one of cost_matrix and distance_func must be NULL. Also, if a cost matrix is used, lower boundary (see below) can not be calculated, because it needs a metric function

flow The resultant size1 × size2 flow matrix: $\texttt{flow}_{i,j}$ is a flow from i th point of signature1 to j th point of signature2

lower_bound Optional input/output parameter: lower boundary of distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or if the signatures consist of weights only (i.e. the signature matrices have a single column). The user **must** initialize *lower_bound. If the calculated distance between mass centers is greater or equal to *lower_bound (it means that the signatures are far enough) the function does not calculate EMD. In any case *lower_bound is set to the calculated distance between mass centers on return. Thus, if user wants to calculate both distance between mass centers and EMD, *lower_bound should be set to 0

userdata Pointer to optional data that is passed into the user-defined distance function

The function computes the earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the applications described in cv.RubnerSept98 is multi-dimensional histogram comparison for image retrieval. EMD is a a transportation problem that is solved using some modification of a simplex algorithm, thus the complexity is exponential in the worst case, though, on average it is much faster. In the case of a real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

---

cv.CheckContourConvexity (view/add comments)

Tests contour convexity.

```
CheckContourConvexity(contour)-> int
```

contour Tested contour (sequence or array of points)

The function tests whether the input contour is convex or not. The contour must be simple, without self-intersections.

CvConvexityDefect (view/add comments)

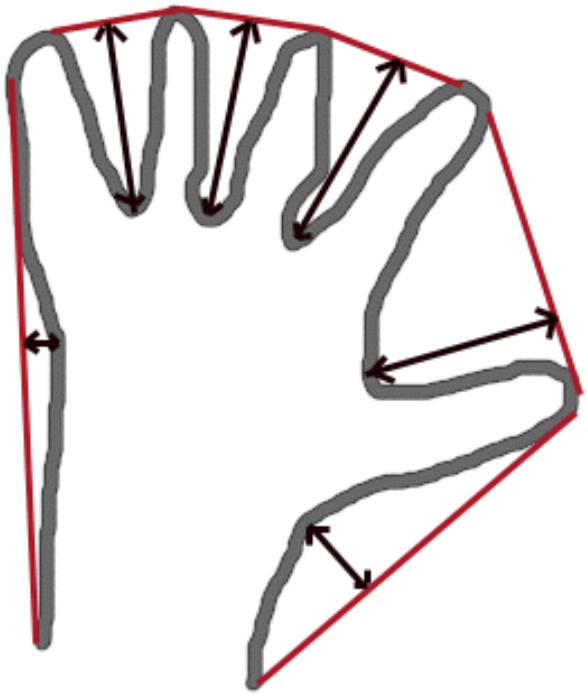
A single contour convexity defect, represented by a tuple `(start, end, depthpoint, depth)`.

start (x, y) point of the contour where the defect begins

end (x, y) point of the contour where the defect ends

depthpoint (x, y) point farthest from the convex hull point within the defect

depth distance between the farthest point and the convex hull



---

cv.ContourArea (view/add comments)

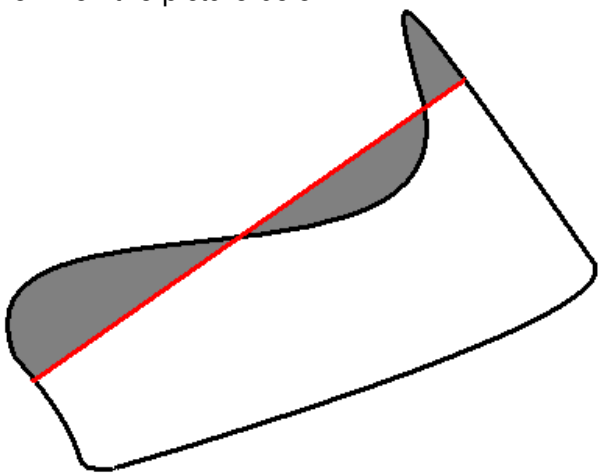
Calculates the area of a whole contour or a contour section.

```
ContourArea(contour,slice=CV_WHOLE_SEQ)-> double
```

contour Contour (sequence or array of vertices)

slice Starting and ending points of the contour section of interest, by default, the area of the whole contour is calculated

The function calculates the area of a whole contour or a contour section. In the latter case the total area bounded by the contour arc and the chord connecting the 2 selected points is calculated as shown on the picture below:



Orientation of the contour affects the area sign, thus the function may return a *negative* result. Use the `fabs()` function from C runtime to get the absolute value of the area.

---

cv.ContourFromContourTree (view/add comments)

Restores a contour from the tree.

```
ContourFromContourTree(tree,storage,criteria)-> contour
```

tree Contour tree

storage Container for the reconstructed contour

criteria Criteria, where to stop reconstruction

The function restores the contour from its binary tree representation. The parameter `criteria` determines the accuracy and/or the number of tree levels used for reconstruction, so it is possible to build an approximated contour. The function returns the reconstructed contour.

---

cv.ConvexHull2 (view/add comments)

Finds the convex hull of a point set.

```
ConvexHull2(points,storage,orientation=CV_CLOCKWISE,return_points=0)->
convex_hull
```

points Sequence or array of 2D points with 32-bit integer or floating-point coordinates

storage The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is an array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified as to truncate the array down to the hull size. If `storage` is NULL then the convex hull will be stored in the same storage as the input sequence

orientation Desired orientation of convex hull: `CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`

return_points If non-zero, the points themselves will be stored in the hull instead of indices if `storage` is an array, or pointers if `storage` is memory storage

The function finds the convex hull of a 2D point set using Sklansky's algorithm. If `storage` is memory storage, the function creates a sequence containing the hull points or pointers to them, depending on `return_points` value and returns the sequence on output. If `storage` is a CvMat, the function returns NULL.

---

cv.ConvexityDefects (view/add comments)

Finds the convexity defects of a contour.

```
ConvexityDefects(contour,convexhull,storage)-> convexity_defects
```

contour Input contour

convexhull Convex hull obtained using cv.ConvexHull2 that should contain pointers or indices to the contour points, not the hull points themselves (the `return_points` parameter in cv.ConvexHull2 should be 0)

storage Container for the output sequence of convexity defects. If it is NULL, the contour or hull (in that order) storage is used

The function finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect structures.

---

cv.CreateContourTree (view/add comments)

Creates a hierarchical representation of a contour.

```
CreateContourTree(contour,storage,threshold)-> contour_tree
```

contour Input contour

storage Container for output tree

threshold Approximation accuracy

The function creates a binary tree representation for the input `contour` and returns the pointer to its root. If the parameter `threshold` is less than or equal to 0, the function creates a full binary tree representation. If the threshold is greater than 0, the function creates a representation with the precision `threshold`: if the vertices with the interceptive area of its base line are less than `threshold`, the tree should not be built any further. The function returns the created tree.

---

cv.FindContours (view/add comments)

Finds the contours in a binary image.

```
FindContours(image, storage, mode=CV_RETR_LIST, method=CV_CHAIN_APPROX_SIMPLE,
offset=(0,0)) -> cvseq
```

image The source, an 8-bit single channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary`. To get such a binary image from grayscale, one may use cv.Threshold, cv.AdaptiveThreshold or cv.Canny. The function modifies the source image's content

storage Container of the retrieved contours

mode Retrieval mode

CV_RETR_EXTERNAL retrives only the extreme outer contours

CV_RETR_LIST retrieves all of the contours and puts them in the list

CV_RETR_CCOMP retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes

CV_RETR_TREE retrieves all of the contours and reconstructs the full hierarchy of nested contours

method Approximation method (for all the modes, except `CV_LINK_RUNS`, which uses built-in approximation)

CV_CHAIN_CODE outputs contours in the Freeman chain code. All other methods output polygons (sequences of vertices)

CV_CHAIN_APPROX_NONE translates all of the points from the chain code into points

CV_CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and diagonal segments and leaves only their end points

CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS applies one of the flavors of the Teh-Chin chain approximation algorithm.

CV_LINK_RUNS uses a completely different contour retrieval algorithm by linking horizontal segments of 1's. Only the `CV_RETR_LIST` retrieval mode can be used with this method.

offset Offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context

The function retrieves contours from the binary image using the algorithm [**?**]. The contours are a useful tool for shape analysis and object detection and recognition.

The function retrieves contours from the binary image and returns the number of retrieved contours. The pointer `first_contour` is filled by the function. It will contain a pointer to the first outermost contour or `NULL` if no contours are detected (if the image is completely black). Other contours may be reached from `first_contour` using the `h_next` and `v_next` links. The sample in the cv.DrawContours discussion shows how to use contours for connected component

detection. Contours can be also used for shape analysis and object recognition - see `squares.py` in the OpenCV sample directory.

Note: the source `image` is modified by this function.

---

cv.FitEllipse2 (view/add comments)

Fits an ellipse around a set of 2D points.

```
FitEllipse2(points)-> Box2D
```

points Sequence or array of points

The function calculates the ellipse that fits best (in least-squares sense) around a set of 2D points. The meaning of the returned structure fields is similar to those in cv.Ellipse except that `size` stores the full lengths of the ellipse axises, not half-lengths.

---

cv.FitLine (view/add comments)

Fits a line to a 2D or 3D point set.

```
FitLine(points, dist_type, param, reps, aeps) -> line
```

points Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates

dist_type The distance used for fitting (see the discussion)

param Numerical parameter (`C`) for some types of distances, if 0 then some optimal value is chosen

reps Sufficient accuracy for the radius (distance between the coordinate origin and the line). 0.01 is a good default value.

aeps Sufficient accuracy for the angle. 0.01 is a good default value.

line The output line parameters. In the case of a 2d fitting, it is a tuple of 4 floats `(vx, vy, x0, y0)` where `(vx, vy)` is a normalized vector collinear to the line and `(x0, y0)` is some point on the line. in the case of a 3D fitting it is a tuple of 6 floats `(vx, vy, vz, x0, y0, z0)` where `(vx, vy, vz)` is a normalized vector collinear to the line and `(x0, y0, z0)` is some point on the line

The function fits a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i th point and the line and $\rho(r)$ is a distance function, one of:

dist_type=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

dist_type=CV_DIST_L1

$$\rho(r) = r$$

dist_type=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

dist_type=CV_DIST_FAIR

$$\rho\left(r\right) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right) \quad \text{where} \quad C = 1.3998$$

dist_type=CV_DIST_WELSCH

$$\rho\left(r\right) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

dist_type=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

---

cv.GetCentralMoment (view/add comments)

Retrieves the central moment from the moment state structure.

```
GetCentralMoment(moments, x_order, y_order) -> double
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, x_order $>= 0$

y_order y order of the retrieved moment, y_order $>= 0$ and x_order + y_order $<= 3$

The function retrieves the central moment, which in the case of image moments is defined as:

$$\mu_{x_order, y_order} = \sum_{x,y}(I(x,y) \cdot (x - x_c)^{x_order} \cdot (y - y_c)^{y_order})$$

where x_c, y_c are the coordinates of the gravity center:

$$x_c = \frac{M_{10}}{M_{00}}, y_c = \frac{M_{01}}{M_{00}}$$

---

cv.GetHuMoments (view/add comments)

Calculates the seven Hu invariants.

```
GetHuMoments(moments) -> hu
```

moments The input moments, computed with cv.Moments

hu The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment, that are defined as:

$$\begin{aligned}
hu_1 &= \eta_{20} + \eta_{02} \\
hu_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
hu_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
hu_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
hu_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
hu_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
hu_7 &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}$$

where η_{ji} denote the normalized central moments.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the

assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

```
>>> import cv
>>> original = cv.LoadImageM("building.jpg", cv.CV_LOAD_IMAGE_GRAYSCALE)
>>> print cv.GetHuMoments(cv.Moments(original))
(0.0010620951868446141, 1.7962726159653835e-07, 1.4932744974469421e-11, 4.4832441315737963e
>>> flipped = cv.CloneMat(original)
>>> cv.Flip(original, flipped)
>>> print cv.GetHuMoments(cv.Moments(flipped))
(0.0010620951868446141, 1.796272615965384e-07, 1.4932744974469935e-11, 4.4832441315740249e-
```

---

cv.GetNormalizedCentralMoment (view/add comments)

Retrieves the normalized central moment from the moment state structure.

```
GetNormalizedCentralMoment(moments, x_order, y_order) -> double
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, $\texttt{x_order} >= 0$

y_order y order of the retrieved moment, $\texttt{y_order} >= 0$ and $\texttt{x_order} + \texttt{y_order} <= 3$

The function retrieves the normalized central moment:

$$\eta_{x_order, y_order} = \frac{\mu_{x_order, y_order}}{M_{00}^{(y_order+x_order)/2+1}}$$

---

cv.GetSpatialMoment (view/add comments)

Retrieves the spatial moment from the moment state structure.

```
GetSpatialMoment(moments, x_order, y_order) -> double
```

moments The moment state, calculated by cv.Moments

x_order x order of the retrieved moment, x_order $>= 0$

y_order y order of the retrieved moment, y_order $>= 0$ and x_order + y_order $<= 3$

The function retrieves the spatial moment, which in the case of image moments is defined as:

$$M_{x_order,\, y_order} = \sum_{x,y} (I(x,y) \cdot x^{x_order} \cdot y^{y_order})$$

where $I(x,y)$ is the intensity of the pixel (x,y).

---

cv.MatchContourTrees (view/add comments)

Compares two contours using their tree representations.

```
MatchContourTrees(tree1,tree2,method,threshold)-> double
```

tree1 First contour tree

tree2 Second contour tree

method Similarity measure, only CV_CONTOUR_TREES_MATCH_I1 is supported

threshold Similarity threshold

The function calculates the value of the matching measure for two contour trees. The similarity measure is calculated level by level from the binary tree roots. If at a certain level the difference between contours becomes less than threshold, the reconstruction process is interrupted and the current difference is returned.

---

cv.MatchShapes (view/add comments)

Compares two shapes.

```
MatchShapes(object1,object2,method,parameter=0)-> None
```

object1 First contour or grayscale image

object2 Second contour or grayscale image

method Comparison method; CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3

parameter Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu moments (see cv.GetHuMoments) (A is object1, B is object2):

method=CV_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1...7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1...7} \left| m_i^A - m_i^B \right|$$

method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1...7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

where

$$m_i^A = sign(h_i^A) \cdot \log h_i^A m_i^B = sign(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

---

cv.MinAreaRect2 (view/add comments)

Finds the circumscribed rectangle of minimal area for a given 2D point set.

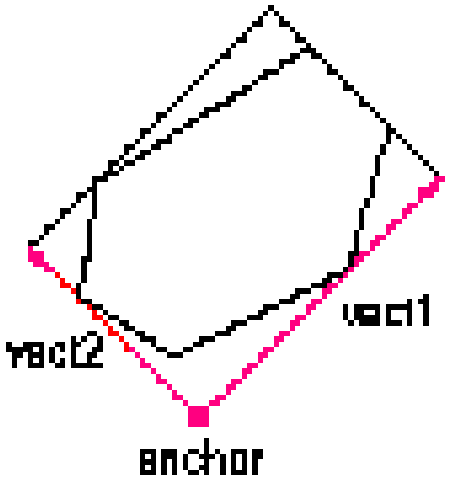
```
MinAreaRect2(points,storage=NULL)-> CvBox2D
```

points Sequence or array of points

storage Optional temporary memory storage

The function finds a circumscribed rectangle of the minimal area for a 2D point set by building a convex hull for the set and applying the rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour



---

cv.MinEnclosingCircle (view/add comments)

Finds the circumscribed circle of minimal area for a given 2D point set.

```
MinEnclosingCircle(points)-> (int,center,radius)
```

points Sequence or array of 2D points

center Output parameter; the center of the enclosing circle

radius Output parameter; the radius of the enclosing circle

The function finds the minimal circumscribed circle for a 2D point set using an iterative algorithm. It returns nonzero if the resultant circle contains all the input points and zero otherwise (i.e. the algorithm failed).

cv.Moments (view/add comments)

Calculates all of the moments up to the third order of a polygon or rasterized shape.

```
Moments(arr, binary = 0) -> moments
```

arr Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points or a vector of points)

moments Pointer to returned moment's state structure

binary (For images only) If the flag is non-zero, all of the zero pixel values are treated as zeroes, and all of the others are treated as 1's

The function calculates spatial and central moments up to the third order and writes them to `moments`. The moments may then be used then to calculate the gravity center of the shape, its area, main axises and various shape characeteristics including 7 Hu invariants.

cv.PointPolygonTest (view/add comments)

Point in contour test.

```
PointPolygonTest(contour,pt,measure_dist)-> double
```

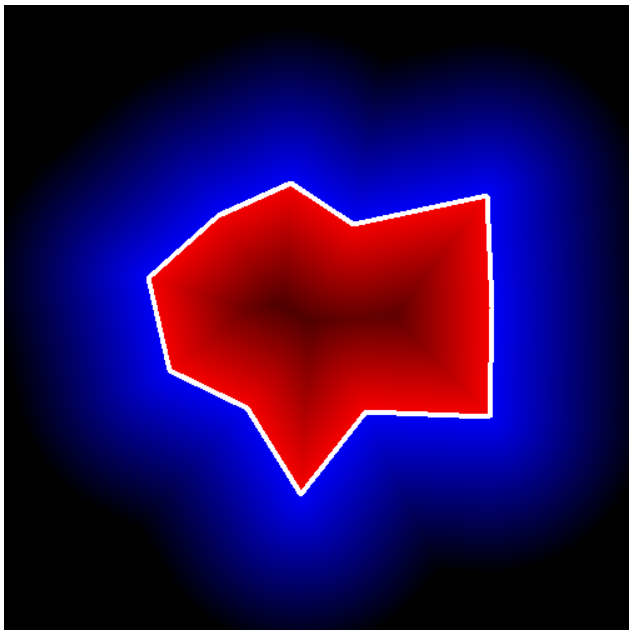
contour Input contour

pt The point tested against the contour

measure_dist If it is non-zero, the function estimates the distance from the point to the nearest contour edge

The function determines whether the point is inside a contour, outside, or lies on an edge (or coinsides with a vertex). It returns positive, negative or zero value, correspondingly. When $\texttt{measure_dist} = 0$, the return value is +1, -1 and 0, respectively. When $\texttt{measure_dist} \neq 0$, it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



23.6 Planar Subdivisions

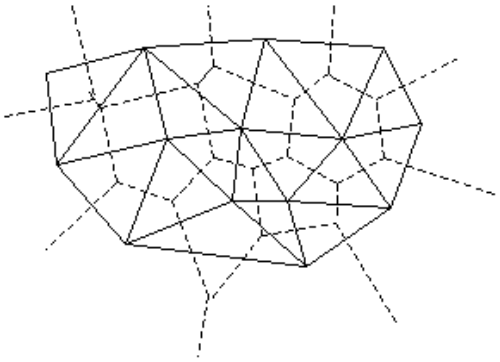
CvSubdiv2D (view/add comments)

Planar subdivision.

edges A CvSet of CvSubdiv2DEdge

Planar subdivision is the subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on a 2d point set, where the points are linked together and form a planar graph, which, together with a few edges connecting the exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists a dual subdivision in which facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called a virtual point below) of the dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dotted lines.



OpenCV subdivides a plane into triangles using Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case the dual subdivision is a Voronoi diagram of the input 2d point set. The subdivisions can be used for the 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) and so forth.

CvSubdiv2DPoint (view/add comments)

Point of original or dual subdivision.

first A connected CvSubdiv2DEdge

pt Position, as a CvPoint2D32f

---

cv.CalcSubdivVoronoi2D (view/add comments)

Calculates the coordinates of Voronoi diagram cells.

```
CalcSubdivVoronoi2D(subdiv)-> None
```

subdiv Delaunay subdivision, in which all the points are already added

The function calculates the coordinates of virtual points. All virtual points corresponding to some vertex of the original subdivision form (when connected together) a boundary of the Voronoi cell at that point.

cv.ClearSubdivVoronoi2D (view/add comments)

Removes all virtual points.

```
ClearSubdivVoronoi2D(subdiv)-> None
```

subdiv Delaunay subdivision

The function removes all of the virtual points. It is called internally in cv.CalcSubdivVoronoi2D if the subdivision was modified after previous call to the function.

cv.CreateSubdivDelaunay2D (view/add comments)

Creates an empty Delaunay triangulation.

```
CreateSubdivDelaunay2D(rect,storage)-> delaunay_triangulation
```

rect Rectangle that includes all of the 2d points that are to be added to the subdivision

storage Container for subdivision

The function creates an empty Delaunay subdivision, where 2d points can be added using the function cv.SubdivDelaunay2DInsert. All of the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

Note that the triangulation is a single large triangle that covers the given rectangle. Hence the three vertices of this triangle are outside the rectangle `rect`.

cv.FindNearestPoint2D (view/add comments)

Finds the closest subdivision vertex to the given point.

```
FindNearestPoint2D(subdiv,pt)-> point
```

subdiv Delaunay or another subdivision

pt Input point

The function is another function that locates the input point within the subdivision. It finds the subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using cv.Subdiv2DLocate) is used as a starting point. The function returns a pointer to the found subdivision vertex.

---

cv.Subdiv2DEdgeDst (view/add comments)

Returns the edge destination.

```
Subdiv2DEdgeDst(edge)-> point
```

edge Subdivision edge (not a quad-edge)

The function returns the edge destination. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using the function cv.CalcSubdivVoronoi2D.

---

cv.Subdiv2DGetEdge (view/add comments)

Returns one of the edges related to the given edge.

```
Subdiv2DGetEdge(edge,type)-> CvSubdiv2DEdge
```

edge Subdivision edge (not a quad-edge)

type Specifies which of the related edges to return, one of the following:

- **CV_NEXT_AROUND_ORG** next around the edge origin (`eOnext` on the picture below if `e` is the input edge)
- **CV_NEXT_AROUND_DST** next around the edge vertex (`eDnext`)
- **CV_PREV_AROUND_ORG** previous around the edge origin (reversed `eRnext`)

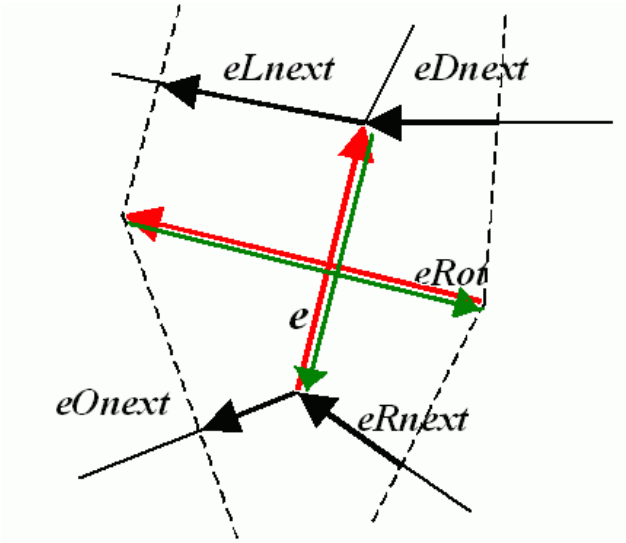
CV_PREV_AROUND_DST previous around the edge destination (reversed `eLnext`)

CV_NEXT_AROUND_LEFT next around the left facet (`eLnext`)

CV_NEXT_AROUND_RIGHT next around the right facet (`eRnext`)

CV_PREV_AROUND_LEFT previous around the left facet (reversed `eOnext`)

CV_PREV_AROUND_RIGHT previous around the right facet (reversed `eDnext`)



The function returns one of the edges related to the input edge.

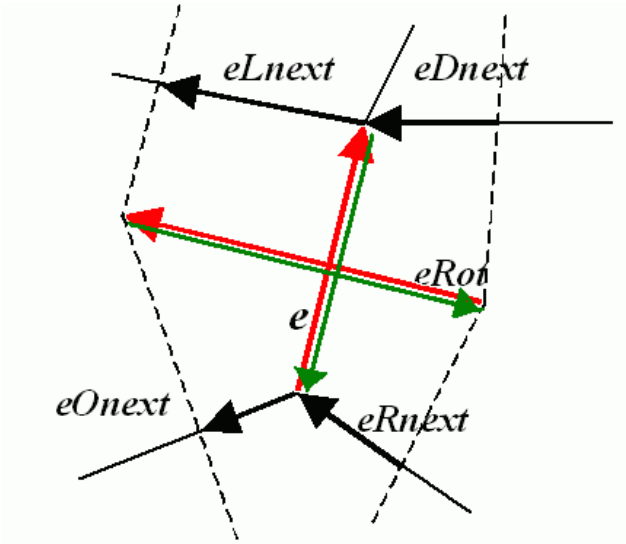
---

cv.Subdiv2DNextEdge (view/add comments)

Returns next edge around the edge origin

```
Subdiv2DNextEdge(edge)-> CvSubdiv2DEdge
```

edge Subdivision edge (not a quad-edge)



The function returns the next edge around the edge origin: `eOnext` on the picture above if `e` is the input edge)

---

cv.Subdiv2DLocate (view/add comments)

Returns the location of a point within a Delaunay triangulation.

```
Subdiv2DLocate(subdiv, pt) -> (loc, where)
```

subdiv Delaunay or another subdivision

pt The point to locate

loc The location of the point within the triangulation

where The edge or vertex. See below.

The function locates the input point within the subdivision. There are 5 cases:

- The point falls into some facet. `loc` is `CV_PTLOC_INSIDE` and `where` is one of edges of the facet.
- The point falls onto the edge. `loc` is `CV_PTLOC_ON_EDGE` and `where` is the edge.

- The point coincides with one of the subdivision vertices. `loc` is `CV_PTLOC_VERTEX` and `where` is the vertex.
- The point is outside the subdivsion reference rectangle. `loc` is `CV_PTLOC_OUTSIDE_RECT` and `where` is None.
- One of input arguments is invalid. The function raises an exception.

---

cv.Subdiv2DRotateEdge (view/add comments)

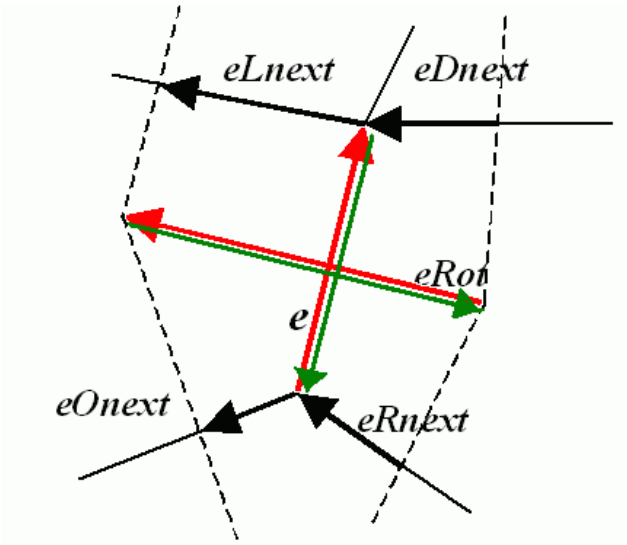
Returns another edge of the same quad-edge.

```
Subdiv2DRotateEdge(edge,rotate)-> CvSubdiv2DEdge
```

edge Subdivision edge (not a quad-edge)

rotate Specifies which of the edges of the same quad-edge as the input one to return, one of the following:

- **0** the input edge (`e` on the picture below if `e` is the input edge)
- **1** the rotated edge (`eRot`)
- **2** the reversed edge (reversed `e` (in green))
- **3** the reversed rotated edge (reversed `eRot` (in green))



The function returns one of the edges of the same quad-edge as the input edge.

---

cv.SubdivDelaunay2DInsert (view/add comments)

Inserts a single point into a Delaunay triangulation.

```
SubdivDelaunay2DInsert(subdiv,pt)-> point
```

subdiv Delaunay subdivision created by the function cv.CreateSubdivDelaunay2D

pt Inserted point

The function inserts a single point into a subdivision and modifies the subdivision topology appropriately. If a point with the same coordinates exists already, no new point is added. The function returns a pointer to the allocated point. No virtual point coordinates are calculated at this stage.

23.7 Motion Analysis and Object Tracking

---

cv.Acc (view/add comments)

Adds a frame to an accumulator.

```
Acc(image,sum,mask=NULL)-> None
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently)

sum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the whole image `image` or its selected region to the accumulator `sum`:

$$\texttt{sum}(x,y) \leftarrow \texttt{sum}(x,y) + \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

cv.MultiplyAcc (view/add comments)

Adds the product of two input images to the accumulator.

```
MultiplyAcc(image1,image2,acc,mask=NULL)-> None
```

image1 First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

image2 Second input image, the same format as the first one

acc Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `acc`:

$$\texttt{acc}(x,y) \leftarrow \texttt{acc}(x,y) + \texttt{image1}(x,y) \cdot \texttt{image2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

cv.RunningAvg (view/add comments)

Updates the running average.

```
RunningAvg(image,acc,alpha,mask=NULL)-> None
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

acc Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

alpha Weight of input image

mask Optional operation mask

The function calculates the weighted sum of the input image `image` and the accumulator `acc` so that `acc` becomes a running average of frame sequence:

$$\texttt{acc}(x,y) \leftarrow (1-\alpha) \cdot \texttt{acc}(x,y) + \alpha \cdot \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

where α regulates the update speed (how fast the accumulator forgets about previous frames).

cv.SquareAcc (view/add comments)

Adds the square of the source image to the accumulator.

```
SquareAcc(image,sqsum,mask=NULL)-> None
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

sqsum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum`:

$$\texttt{sqsum}(x,y) \leftarrow \texttt{sqsum}(x,y) + \texttt{image}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

23.8 Feature Detection

cv.Canny (view/add comments)

Implements the Canny algorithm for edge detection.

```
Canny(image,edges,threshold1,threshold2,aperture_size=3)-> None
```

image Single-channel input image

edges Single-channel image to store the edges found by the function

threshold1 The first threshold

threshold2 The second threshold

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel)

The function finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges.

---

cv.CornerEigenValsAndVecs (view/add comments)

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
CornerEigenValsAndVecs(image,eigenvv,blockSize,aperture_size=3)-> None
```

image Input image

eigenvv Image to store the results. It must be 6 times wider than the input image

blockSize Neighborhood size (see discussion)

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel)

For every pixel, the function `cvCornerEigenValsAndVecs` considers a `blockSize×blockSize` neigborhood S(p). It calcualtes the covariation matrix of derivatives over the neigborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dx \cdot dI/dy)^2 \\ \sum_{S(p)}(dI/dx \cdot dI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the eigenvalues of M; not sorted

x_1, y_1 are the eigenvectors corresponding to λ_1

x_2, y_2 are the eigenvectors corresponding to λ_2

---

cv.CornerHarris (view/add comments)

Harris edge detector.

```
CornerHarris(image,harris_dst,blockSize,aperture_size=3,k=0.04)-> None
```

image Input image

harris_dst Image to store the Harris detector responses. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cv.CornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel).

k Harris detector free parameter. See the formula below

The function runs the Harris edge detector on the image. Similarly to cv.CornerMinEigenVal and cv.CornerEigenValsAndVecs, for each pixel it calculates a 2×2 gradient covariation matrix M over a `blockSize` $\times$ `blockSize` neighborhood. Then, it stores

$$det(M) - k\, trace(M)^2$$

to the destination image. Corners in the image can be found as the local maxima of the destination image.

---

cv.CornerMinEigenVal (view/add comments)

Calculates the minimal eigenvalue of gradient matrices for corner detection.

```
CornerMinEigenVal(image,eigenval,blockSize,aperture_size=3)-> None
```

image Input image

eigenval Image to store the minimal eigenvalues. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cv.CornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel).

The function is similar to cv.CornerEigenValsAndVecs but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. $min(\lambda_1, \lambda_2)$ in terms of the previous function.

cv.FindCornerSubPix (view/add comments)

Refines the corner locations.

```
FindCornerSubPix(image,corners,win,zero_zone,criteria)-> corners
```

image Input image

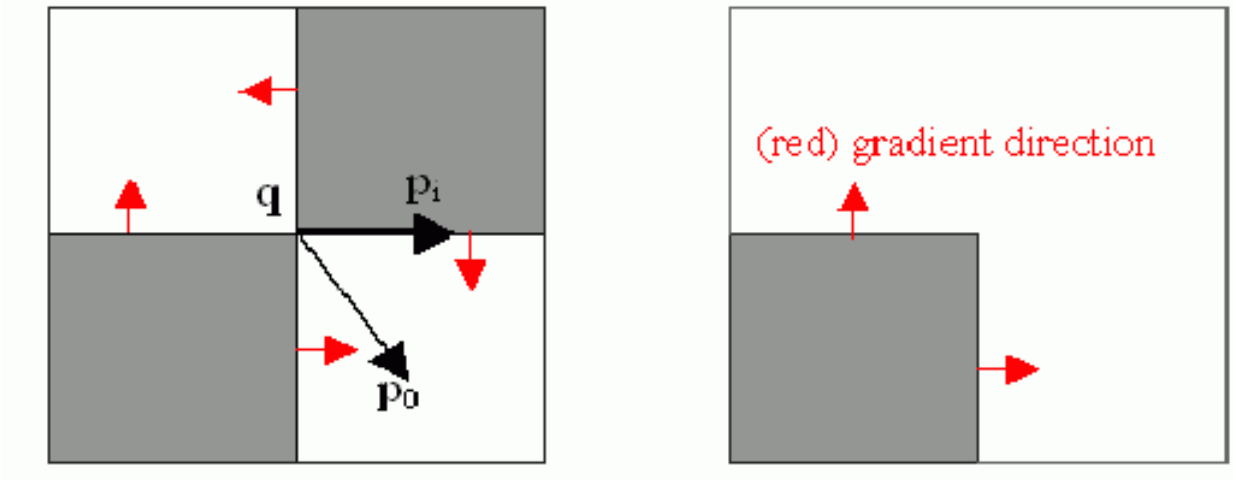
corners Initial coordinates of the input corners as a list of (x, y) pairs

win Half of the side length of the search window. For example, if win=(5,5), then a $5*2+1 \times 5*2+1 = 11 \times 11$ search window would be used

zero_zone Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size

criteria Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The criteria may specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below. It returns the refined coordinates as a list of (x, y) pairs.



Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider the expression:

$$\epsilon_i = {DI_{p_i}}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot {DI_{p_i}}^T)q = \sum_i (DI_{p_i} \cdot {DI_{p_i}}^T \cdot p_i)$$

where the gradients are summed within a neighborhood ("search window") of q. Calling the first gradient term G and the second gradient term b gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center q and then iterates until the center keeps within a set threshold.

---

cv.GoodFeaturesToTrack (view/add comments)

Determines strong corners on an image.

```
GoodFeaturesToTrack(image,eigImage,tempImage,cornerCount,qualityLevel,minDistance,m
corners
```

image The source 8-bit or floating-point 32-bit, single-channel image

eigImage Temporary floating-point 32-bit image, the same size as `image`

tempImage Another temporary image, the same size and format as `eigImage`

cornerCount number of corners to detect

qualityLevel Multiplier for the max/min eigenvalue; specifies the minimal accepted quality of image corners

minDistance Limit, specifying the minimum possible distance between the returned corners; Euclidian distance is used

mask Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL

blockSize Size of the averaging block, passed to the underlying cv.CornerMinEigenVal or cv.CornerHarris used by the function

useHarris If nonzero, Harris operator (cv.CornerHarris) is used instead of default cv.CornerMinEigenVal

k Free parameter of Harris detector; used only if ($\texttt{useHarris} \neq 0$)

The function finds the corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using the cv.CornerMinEigenVal function and stores them in `eigImage`. Then it performs non-maxima suppression (only the local maxima in 3×3 neighborhood are retained). The next step rejects the corners with the minimal eigenvalue less than $\texttt{qualityLevel} \cdot max(\texttt{eigImage}(x, y))$. Finally, the function ensures that the distance between any two corners is not smaller than `minDistance`. The weaker corners (with a smaller min eigenvalue) that are too close to the stronger corners are rejected.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel`, and `A` ¿ B, the array of returned corners with `qualityLevel=A` will be the prefix of the output corners array with `qualityLevel=B`.

---

cv.HoughLines2 (view/add comments)

Finds lines in a binary image using a Hough transform.

```
HoughLines2(image,storage,method,rho,theta,threshold,param1=0,param2=0)->
lines
```

image The 8-bit, single-channel, binary source image. In the case of a probabilistic method, the image is modified by the function

storage The storage for the lines that are detected. It can be a memory storage (in this case a sequence of lines is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of a particular type (see below) to which the lines' parameters are written. The matrix header is modified by the function so its `cols` or `rows` will contain the number of lines detected. If `storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (in the case of standard hough transform the lines are sorted by the accumulator value)

method The Hough transform variant, one of the following:

CV_HOUGH_STANDARD classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ), where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of `CV_32FC2` type

CV_HOUGH_PROBABILISTIC probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole line. Each segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of `CV_32SC4` type

CV_HOUGH_MULTI_SCALE multi-scale variant of the classical Hough transform. The lines are encoded the same way as `CV_HOUGH_STANDARD`

rho Distance resolution in pixel-related units

theta Angle resolution measured in radians

threshold Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`

param1 The first method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the minimum line length.
- For the multi-scale Hough transform it is the divisor for the distance resolution ρ. (The coarse distance resolution will be ρ and the accurate resolution will be $(\rho/\texttt{param1})$).

param2 The second method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the maximum gap between line segments lying on the same line to treat them as a single line segment (i.e. to join them).
- For the multi-scale Hough transform it is the divisor for the angle resolution θ. (The coarse angle resolution will be θ and the accurate resolution will be $(\theta/\texttt{param2})$).

The function implements a few variants of the Hough transform for line detection.

---

cv.PreCornerDetect (view/add comments)

Calculates the feature map for corner detection.

```
PreCornerDetect(image,corners,apertureSize=3)-> None
```

image Input image

corners Image to store the corner candidates

apertureSize Aperture parameter for the Sobel operator (see cv.Sobel)

The function calculates the function

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}$$

where $D_?$ denotes one of the first image derivatives and $D_{??}$ denotes a second image derivative.

The corners can be found as local maximums of the function below:

```
import cv

def precornerdetect(image):
    # assume that the image is floating-point
    corners = cv.CloneMat(image)
    cv.PreCornerDetect(image, corners, 3)

    dilated_corners = cv.CloneMat(image)
    cv.Dilate(corners, dilated_corners, None, 1)

    corner_mask = cv.CreateMat(image.rows, image.cols, cv.CV_8UC1)
    cv.Sub(corners, dilated_corners, corners)
    cv.CmpS(corners, 0, corner_mask, cv.CV_CMP_GE)
    return (corners, corner_mask)
```

23.9 Object Detection

---

cv.MatchTemplate (view/add comments)

Compares a template against overlapped image regions.

```
MatchTemplate(image,templ,result,method)-> None
```

image Image where the search is running; should be 8-bit or 32-bit floating-point

templ Searched template; must be not greater than the source image and the same data type as the image

result A map of comparison results; single-channel 32-bit floating-point. If `image` is $W \times H$ and `templ` is $w \times h$ then `result` must be $(W - w + 1) \times (H - h + 1)$

method Specifies the way the template must be compared with the image regions (see below)

The function is similar to cv.CalcBackProjectPatch. It slides through `image`, compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result`. Here are the formulas for the different comparison methods one may use (I denotes `image`, T `template`, R `result`). The summation is done over template and/or the image patch: $x' = 0...w - 1, y' = 0...h - 1$

method=CV_TM_SQDIFF

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x + x', y + y'))^2$$

method=CV_TM_SQDIFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x + x', y + y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}}$$

method=CV_TM_CCORR

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x + x', y + y'))$$

method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I'(x + x', y + y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}}$$

method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I(x + x', y + y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x + x', y + y') = I(x + x', y + y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x + x'', y + y'')$$

method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'}(T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

After the function finishes the comparison, the best matches can be found as global minimums (`CV_TM_SQDIFF`) or maximums (`CV_TM_CCORR` and `CV_TM_CCOEFF`) using the cv.MinMaxLoc function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel).

Chapter 24

features2d. Feature Detection and Descriptor Extraction

24.1 Feature detection and description

image The image. Keypoints (corners) will be detected on this.

keypoints Keypoints detected on the image.

threshold Threshold on difference between intensity of center pixel and pixels on circle around this pixel. See description of the algorithm.

nonmaxSupression If it is true then non-maximum supression will be applied to detected corners (keypoints).

CvSURFPoint (view/add comments)

A SURF keypoint, represented as a tuple `((x, y), laplacian, size, dir, hessian)`.

x x-coordinate of the feature within the image

y y-coordinate of the feature within the image

laplacian -1, 0 or +1. sign of the laplacian at the point. Can be used to speedup feature comparison since features with laplacians of different signs can not match

size size of the feature

dir orientation of the feature: 0..360 degrees

hessian value of the hessian (can be used to approximately estimate the feature strengths; see also params.hessianThreshold)

---

cv.ExtractSURF (view/add comments)

Extracts Speeded Up Robust Features from an image.

```
ExtractSURF(image,mask,storage,params)-> (keypoints,descriptors)
```

image The input 8-bit grayscale image

mask The optional input 8-bit mask. The features are only found in the areas that contain more than 50% of non-zero mask pixels

keypoints sequence of keypoints.

descriptors sequence of descriptors. Each SURF descriptor is a list of floats, of length 64 or 128.

storage Memory storage where keypoints and descriptors will be stored

params Various algorithm parameters in a tuple `(extended, hessianThreshold, nOctaves, nOctaveLayers)`:

- **extended** 0 means basic descriptors (64 elements each), 1 means extended descriptors (128 elements each)
- **hessianThreshold** only features with hessian larger than that are extracted. good default value is 300-500 (can depend on the average local contrast and sharpness of the image). user can further filter out some features based on their hessian values and other characteristics.
- **nOctaves** the number of octaves to be used for extraction. With each next octave the feature size is doubled (3 by default)
- **nOctaveLayers** The number of layers within each octave (4 by default)

The function cvExtractSURF finds robust features in the image, as described in [**?**]. For each feature it returns its location, size, orientation and optionally the descriptor, basic or extended. The function can be used for object tracking and localization, image stitching etc.

To extract strong SURF features from an image

```
>>> import cv
>>> im = cv.LoadImageM("building.jpg", cv.CV_LOAD_IMAGE_GRAYSCALE)
>>> (keypoints, descriptors) = cv.ExtractSURF(im, None, cv.CreateMemStorage(), (0, 30000, 
>>> print len(keypoints), len(descriptors)
6 6
>>> for ((x, y), laplacian, size, dir, hessian) in keypoints:
...     print "x=\%d y=\%d laplacian=\%d size=\%d dir=\%f hessian=\%f" \% (x, y, laplacian,
x=30 y=27 laplacian=-1 size=31 dir=69.778503 hessian=36979.789062
x=296 y=197 laplacian=1 size=33 dir=111.081039 hessian=31514.349609
x=296 y=266 laplacian=1 size=32 dir=107.092300 hessian=31477.908203
x=254 y=284 laplacian=1 size=31 dir=279.137360 hessian=34169.800781
x=498 y=525 laplacian=-1 size=33 dir=278.006592 hessian=31002.759766
x=777 y=281 laplacian=1 size=70 dir=167.940964 hessian=35538.363281
```

cv.GetStarKeypoints (view/add comments)

Retrieves keypoints using the StarDetector algorithm.

```
GetStarKeypoints(image,storage,params)-> keypoints
```

image The input 8-bit grayscale image

storage Memory storage where the keypoints will be stored

params Various algorithm parameters in a tuple `(maxSize, responseThreshold, lineThresholdProje lineThresholdBinarized, suppressNonmaxSize)`:

- **maxSize** maximal size of the features detected. The following values of the parameter are supported: 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
- **responseThreshold** threshold for the approximatd laplacian, used to eliminate weak features
- **lineThresholdProjected** another threshold for laplacian to eliminate edges
- **lineThresholdBinarized** another threshold for the feature scale to eliminate edges
- **suppressNonmaxSize** linear size of a pixel neighborhood for non-maxima suppression

The function GetStarKeypoints extracts keypoints that are local scale-space extremas. The scale-space is constructed by computing approximate values of laplacians with different sigma's at each pixel. Instead of using pyramids, a popular approach to save computing time, all of the

laplacians are computed at each pixel of the original high-resolution image. But each approximate laplacian value is computed in O(1) time regardless of the sigma, thanks to the use of integral images. The algorithm is based on the paper Agrawal08 , but instead of a square, hexagon or octagon it uses an 8-end star shape, hence the name, consisting of overlapping upright and tilted squares.

Each keypoint is represented by a tuple `((x, y), size, response)`:

`x, y` Screen coordinates of the keypoint

`size` feature size, up to `maxSize`

`response` approximated laplacian value for the keypoint

Chapter 25

flann. Clustering and Search in Multi-Dimensional Spaces

25.1 Fast Approximate Nearest Neighbor Search

Chapter 26

objdetect. Object Detection

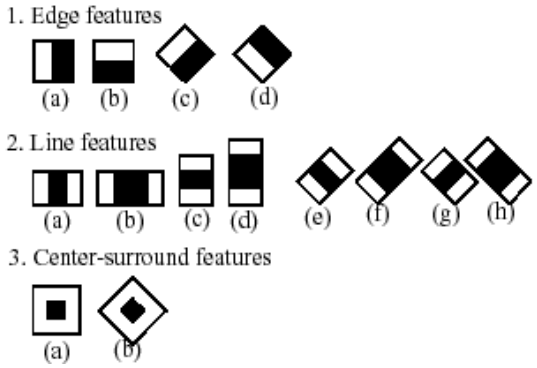
26.1 Cascade Classification

Haar Feature-based Cascade Classifier for Object Detection

The object detector described below has been initially proposed by Paul Viola cv.Viola01 and improved by Rainer Lienhart cv.Lienhart02 . First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundred sample views of a particular object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.

After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a ”1” if the region is likely to show the object (i.e., face/car), and ”0” otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily ”resized” in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word ”cascade” in the classifier name means that the resultant classifier consists of several simpler classifiers (*stages*) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word ”boosted” means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different `boosting` techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifers, and are calculated as described below. The current algorithm uses the following Haar-like features:



The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in the case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and the cv.Integral description).

A simple demonstration of face detection, which draws a rectangle around each detected face:

```
hc = cv.Load("haarcascade_frontalface_default.xml")
img = cv.LoadImage("faces.jpg", 0)
faces = cv.HaarDetectObjects(img, hc, cv.CreateMemStorage())
for (x,y,w,h),n in faces:
    cv.Rectangle(img, (x,y), (x+w,y+h), 255)
cv.SaveImage("faces_detected.jpg", img)
```

---

cv.HaarDetectObjects (view/add comments)

Detects objects in the image.

```
HaarDetectObjects(image,cascade,storage,scaleFactor=1.1,minNeighbors=3,flags=0,minS
detected_objects
```

image Image to detect objects in

cascade Haar classifier cascade in internal representation

storage Memory storage to store the resultant sequence of the object candidate rectangles

scaleFactor The factor by which the search window is scaled between the subsequent scans, 1.1 means increasing window by 10%

minNeighbors Minimum number (minus 1) of neighbor rectangles that makes up an object. All the groups of a smaller number of rectangles than `min_neighbors`-1 are rejected. If `minNeighbors` is 0, the function does not any grouping at all and returns all the detected candidate rectangles, which may be useful if the user wants to apply a customized grouping procedure

flags Mode of operation. Currently the only flag that may be specified is `CV_HAAR_DO_CANNY_PRUNING`. If it is set, the function uses Canny edge detector to reject some image regions that contain too few or too much edges and thus can not contain the searched object. The particular threshold values are tuned for face detection and in this case the pruning speeds up the processing

minSize Minimum window size. By default, it is set to the size of samples the classifier has been trained on ($\sim 20 \times 20$ for face detection)

maxSize Maximum window size to use. By default, it is set to the size of the image.

The function finds rectangular regions in the given image that are likely to contain objects the cascade has been trained for and returns those regions as a sequence of rectangles. The function scans the image several times at different scales (see cv.SetImagesForHaarClassifierCascade). Each time it considers overlapping regions in the image and applies the classifiers to the regions using cv.RunHaarClassifierCascade. It may also apply some heuristics to reduce number of analyzed regions, such as Canny prunning. After it has proceeded and collected the candidate rectangles (regions that passed the classifier cascade), it groups them and returns a sequence of average rectangles for each large enough group. The default parameters (`scale_factor` =1.1, `min_neighbors` =3, `flags` =0) are tuned for accurate yet slow object detection. For a faster operation on real video images the settings are: `scale_factor` =1.2, `min_neighbors` =2, `flags` =`CV_HAAR_DO_CANNY_PRUNING`, `min_size` =*minimum possible face size* (for example, $\sim$ 1/4 to 1/16 of the image area in the case of video conferencing).

The function returns a list of tuples, `(rect, neighbors)`, where rect is a CvRect specifying the object's extents and neighbors is a number of neighbors.

```
>>> import cv
>>> image = cv.LoadImageM("lena.jpg", cv.CV_LOAD_IMAGE_GRAYSCALE)
>>> cascade = cv.Load("../../data/haarcascades/haarcascade_frontalface_alt.xml")
>>> print cv.HaarDetectObjects(image, cascade, cv.CreateMemStorage(0), 1.2, 2, 0, (20, 20))
[((217, 203, 169, 169), 24)]
```

Chapter 27

video. Video Analysis

27.1 Motion Analysis and Object Tracking

---

cv.CalcGlobalOrientation (view/add comments)

Calculates the global motion orientation of some selected region.

```
CalcGlobalOrientation(orientation,mask,mhi,timestamp,duration)-> float
```

orientation Motion gradient orientation image; calculated by the function cv.CalcMotionGradient

mask Mask image. It may be a conjunction of a valid gradient mask, obtained with cv.CalcMotionGradient and the mask of the region, whose direction needs to be calculated

mhi Motion history image

timestamp Current time in milliseconds or other units, it is better to store time passed to cv.UpdateMotionHistory before and reuse it here, because running cv.UpdateMotionHistory and cv.CalcMotionGradient on large images may take some time

duration Maximal duration of motion track in milliseconds, the same as cv.UpdateMotionHistory

The function calculates the general motion direction in the selected region and returns the angle between 0 degrees and 360 degrees . At first the function builds the orientation histogram and finds the basic orientation as a coordinate of the histogram maximum. After that the function calculates the shift relative to the basic orientation as a weighted sum of all of the orientation

vectors: the more recent the motion, the greater the weight. The resultant angle is a circular sum of the basic orientation and the shift.

---

cv.CalcMotionGradient (view/add comments)

Calculates the gradient orientation of a motion history image.

```
CalcMotionGradient(mhi,mask,orientation,delta1,delta2,apertureSize=3)->
None
```

mhi Motion history image

mask Mask image; marks pixels where the motion gradient data is correct; output parameter

orientation Motion gradient orientation image; contains angles from 0 to 360 degrees

delta1 See below

delta2 See below

apertureSize Aperture size of derivative operators used by the function: CV_SCHARR, 1, 3, 5 or 7 (see cv.Sobel)

The function calculates the derivatives Dx and Dy of `mhi` and then calculates gradient orientation as:

$$\texttt{orientation}(x,y) = \arctan \frac{Dy(x,y)}{Dx(x,y)}$$

where both $Dx(x,y)$ and $Dy(x,y)$ signs are taken into account (as in the cv.CartToPolar function). After that `mask` is filled to indicate where the orientation is valid (see the `delta1` and `delta2` description).

The function finds the minimum ($m(x,y)$) and maximum ($M(x,y)$) mhi values over each pixel (x,y) neighborhood and assumes the gradient is valid only if

$$\min(\texttt{delta1},\texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1},\texttt{delta2}).$$

cv.CalcOpticalFlowBM (view/add comments)

Calculates the optical flow for two images by using the block matching method.

```
CalcOpticalFlowBM(prev,curr,blockSize,shiftSize,max_range,usePrevious,velx,vely)->
None
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

blockSize Size of basic blocks that are compared

shiftSize Block coordinate increments

max_range Size of the scanned neighborhood in pixels around the block

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of

$$\left\lfloor \frac{\texttt{prev->width} - \texttt{blockSize.width}}{\texttt{shiftSize.width}} \right\rfloor \times \left\lfloor \frac{\texttt{prev->height} - \texttt{blockSize.height}}{\texttt{shiftSize.height}} \right\rfloor$$

size, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size `velx`, 32-bit floating-point, single-channel

The function calculates the optical flow for overlapped blocks `blockSize.width×blockSize.height` pixels each, thus the velocity fields are smaller than the original images. For every block in `prev` the functions tries to find a similar block in `curr` in some neighborhood of the original block or shifted by (velx(x0,y0),vely(x0,y0)) block as has been calculated by previous function call (if `usePrevious=1`)

cv.CalcOpticalFlowHS (view/add comments)

Calculates the optical flow for two images.

```
CalcOpticalFlowHS(prev,curr,usePrevious,velx,vely,lambda,criteria)->
None
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

lambda Lagrangian multiplier

criteria Criteria of termination of velocity computing

The function computes the flow for every pixel of the first input image using the Horn and Schunck algorithm [**?**].

---

cv.CalcOpticalFlowLK (view/add comments)

Calculates the optical flow for two images.

```
CalcOpticalFlowLK(prev,curr,winSize,velx,vely)-> None
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

winSize Size of the averaging window used for grouping pixels

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

The function computes the flow for every pixel of the first input image using the Lucas and Kanade algorithm [**?**].

---

cv.CalcOpticalFlowPyrLK (view/add comments)

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

```
CalcOpticalFlowPyrLK( prev, curr, prevPyr, currPyr, prevFeatures,
winSize, level, criteria, flags, guesses = None) -> (currFeatures,
status, track_error)
```

prev First frame, at time `t`

curr Second frame, at time `t + dt`

prevPyr Buffer for the pyramid for the first frame. If the pointer is not `NULL` , the buffer must have a sufficient size to store the pyramid from level `1` to level `level` ; the total size of `(image_width+8)*image_height/3` bytes is sufficient

currPyr Similar to `prevPyr`, used for the second frame

prevFeatures Array of points for which the flow needs to be found

currFeatures Array of 2D points containing the calculated new positions of the input features in the second image

winSize Size of the search window of each pyramid level

level Maximal pyramid level number. If `0` , pyramids are not used (single level), if `1` , two levels are used, etc

status Array. Every element of the array is set to `1` if the flow for the corresponding feature has been found, `0` otherwise

track_error Array of double numbers containing the difference between patches around the original and moved points. Optional parameter; can be `NULL`

criteria Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped

flags Miscellaneous flags:

CV_LKFLOWPyr_A_READY pyramid for the first frame is precalculated before the call

CV_LKFLOWPyr_B_READY pyramid for the second frame is precalculated before the call

guesses optional array of estimated coordinates of features in second frame, with same length as `prevFeatures`

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids [**?**] . It calculates the coordinates of the feature points on the current video frame given their coordinates on the previous frame. The function finds the coordinates with sub-pixel accuracy.

Both parameters `prevPyr` and `currPyr` comply with the following rules: if the image pointer is 0, the function allocates the buffer internally, calculates the pyramid, and releases the buffer after processing. Otherwise, the function calculates the pyramid and stores it in the buffer unless the flag `CV_LKFLOWPyr_A[B]_READY` is set. The image should be large enough to fit the Gaussian pyramid data. After the function call both pyramids are calculated and the readiness flag for the corresponding image can be set in the next call (i.e., typically, for all the image pairs except the very first one `CV_LKFLOWPyr_A_READY` is set).

---

cv.CamShift (view/add comments)

Finds the object center, size, and orientation.

```
CamShift(prob_image,window,criteria)-> (int, comp, box)
```

prob_image Back projection of object histogram (see cv.CalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

box Circumscribed box for the object.

The function implements the CAMSHIFT object tracking algrorithm [**?**]. First, it finds an object center using cv.MeanShift and, after that, calculates the object size and orientation. The function returns number of iterations made within cv.MeanShift.

The `CamShiftTracker` class declared in cv.hpp implements the color object tracker that uses the function.

CvKalman (view/add comments)

Kalman filter state.

MP number of measurement vector dimensions

DP number of state vector dimensions

CP number of control vector dimensions

state_pre predicted state (x'(k)): x(k)=A*x(k-1)+B*u(k)

state_post corrected state (x(k)): x(k)=x'(k)+K(k)*(z(k)-H*x'(k))

transition_matrix state transition matrix (A)

control_matrix control matrix (B) (it is not used if there is no control)

measurement_matrix measurement matrix (H)

process_noise_cov process noise covariance matrix (Q)

measurement_noise_cov measurement noise covariance matrix (R)

error_cov_pre priori error estimate covariance matrix (P'(k)): P'(k)=A*P(k-1)*At + Q

gain Kalman gain matrix (K(k)): K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)

error_cov_post posteriori error estimate covariance matrix (P(k)): P(k)=(I-K(k)*H)*P'(k)

The structure `CvKalman` is used to keep the Kalman filter state. It is created by the cv.CreateKalman function, updated by the cv.KalmanPredict and cv.KalmanCorrect functions . Normally, the structure is used for the standard Kalman filter (notation and the formulas below are borrowed from the excellent Kalman tutorial [**?**])

$$x_k = A \cdot x_{k-1} + B \cdot u_k + w_k$$
$$z_k = H \cdot x_k + v_k$$

where:

x_k (x_{k-1})	state of the system at the moment *k* (*k-1*)
z_k	measurement of the system state at the moment *k*
u_k	external control applied at the moment *k*

w_k and v_k are normally-distributed process and measurement noise, respectively:

$$p(w) \sim N(0, Q)$$
$$p(v) \sim N(0, R)$$

that is,
Q process noise covariance matrix, constant or variable,
R measurement noise covariance matrix, constant or variable
In the case of the standard Kalman filter, all of the matrices: A, B, H, Q and R are initialized once after the cv.CvKalman structure is allocated via cv.CreateKalman. However, the same structure and the same functions may be used to simulate the extended Kalman filter by linearizing the extended Kalman filter equation in the current system state neighborhood, in this case A, B, H (and, probably, Q and R) should be updated on every step.

---

cv.CreateKalman (view/add comments)

Allocates the Kalman filter structure.

```
CreateKalman(dynam_params, measure_params, control_params=0) -> CvKalman
```

dynam_params dimensionality of the state vector

measure_params dimensionality of the measurement vector

control_params dimensionality of the control vector

The function allocates cv.CvKalman and all its matrices and initializes them somehow.

---

cv.KalmanCorrect (view/add comments)

Adjusts the model state.

```
KalmanCorrect(kalman, measurement) -> cvmat
```

kalman Kalman filter object returned by cv.CreateKalman

measurement CvMat containing the measurement vector

The function adjusts the stochastic model state on the basis of the given measurement of the model state:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

where

z_k	given measurement (`mesurement` parameter)
K_k	Kalman "gain" matrix.

The function stores the adjusted state at `kalman->state_post` and returns it on output.

cv.KalmanPredict (view/add comments)

Estimates the subsequent model state.

```
KalmanPredict(kalman, control=None) -> cvmat
```

kalman Kalman filter object returned by cv.CreateKalman

control Control vector u_k, should be NULL iff there is no external control (`control_params` =0)

The function estimates the subsequent stochastic model state by its current state and stores it at `kalman->state_pre`:

$$x'_k = Ax_{k-1} + Bu_k$$
$$P'_k = AP_{k-1}A^T + Q$$

where

x'_k	is predicted state `kalman->state_pre`,
x_{k-1}	is corrected state on the previous step `kalman->state_post` (should be initialized somehow in the beginning, zero vector by default),
u_k	is external control (`control` parameter),
P'_k	is priori error covariance matrix `kalman->error_cov_pre`
P_{k-1}	is posteriori error covariance matrix on the previous step `kalman->error_cov_post` (should be initialized somehow in the beginning, identity matrix by default),

The function returns the estimated state.

KalmanUpdateByMeasurement

Synonym for KalmanCorrect

KalmanUpdateByTime

Synonym for KalmanPredict

---

cv.MeanShift (view/add comments)

Finds the object center on back projection.

```
MeanShift(prob_image,window,criteria)-> comp
```

prob_image Back projection of the object histogram (see cv.CalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

The function iterates to find the object center given its back projection and initial position of search window. The iterations are made until the search window center moves by less than the given value and/or until the function has done the maximum number of iterations. The function returns the number of iterations made.

---

cv.SegmentMotion (view/add comments)

Segments a whole motion into separate moving parts.

```
SegmentMotion(mhi,seg_mask,storage,timestamp,seg_thresh)-> None
```

mhi Motion history image

seg_mask Image where the mask found should be stored, single-channel, 32-bit floating-point

storage Memory storage that will contain a sequence of motion connected components

timestamp Current time in milliseconds or other units

seg_thresh Segmentation threshold; recommended to be equal to the interval between motion history "steps" or greater

The function finds all of the motion segments and marks them in `seg_mask` with individual values (1,2,...). It also returns a sequence of cv.CvConnectedComp structures, one for each motion component. After that the motion direction for every component can be calculated with cv.CalcGlobalOrientation using the extracted mask of the particular component cv.Cmp.

---

cv.SnakeImage (view/add comments)

Changes the contour position to minimize its energy.

```
SnakeImage(image,points,alpha,beta,gamma,win,criteria,calc_gradient=1)->
new_points
```

image The source image or external energy field

points Contour points (snake)

alpha Weight[s] of continuity energy, single float or a list of floats, one for each contour point

beta Weight[s] of curvature energy, similar to `alpha`

gamma Weight[s] of image energy, similar to `alpha`

win Size of neighborhood of every point used to search the minimum, both `win.width` and `win.height` must be odd

criteria Termination criteria

calc_gradient Gradient flag; if not 0, the function calculates the gradient magnitude for every image pixel and consideres it as the energy field, otherwise the input image itself is considered

The function updates the snake in order to minimize its total energy that is a sum of internal energy that depends on the contour shape (the smoother contour is, the smaller internal energy is) and external energy that depends on the energy field and reaches minimum at the local energy extremums that correspond to the image edges in the case of using an image gradient.

The parameter `criteria.epsilon` is used to define the minimal number of points that must be moved during any iteration to keep the iteration process running.

If at some iteration the number of moved points is less than `criteria.epsilon` or the function performed `criteria.max_iter` iterations, the function terminates.

The function returns the updated list of points.

---

cv.UpdateMotionHistory (view/add comments)

Updates the motion history image by a moving silhouette.

```
UpdateMotionHistory(silhouette,mhi,timestamp,duration)-> None
```

silhouette Silhouette mask that has non-zero pixels where the motion occurs

mhi Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp Current time in milliseconds or other units

duration Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

Chapter 28

highgui. High-level GUI and Media I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and ”remember” their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

28.1 User Interface

---

cv.CreateTrackbar (view/add comments)

Creates a trackbar and attaches it to the specified window

```
CreateTrackbar(trackbarName, windowName, value, count, onChange) ->
None
```

trackbarName Name of the created trackbar.

windowName Name of the window which will be used as a parent for created trackbar.

value Initial value for the slider position, between 0 and `count`.

count Maximal position of the slider. Minimal position is always 0.

onChange OpenCV calls `onChange` every time the slider changes position. OpenCV will call it as `func(x)` where `x` is the new position of the slider.

The function `cvCreateTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable to be syncronized with trackbar position and specifies a callback function to be called on trackbar position change. The created trackbar is displayed on the top of the given window.

[Qt Backend Only] qt-specific details:

windowName Name of the window which will be used as a parent for created trackbar. Can be NULL if the trackbar should be attached to the control panel.

The created trackbar is displayed at the bottom of the given window if *windowName* is correctly provided, or displayed on the control panel if *windowName* is NULL.

By clicking on the label of each trackbar, it is possible to edit the trackbar's value manually for a more accurate control of it.

---

cv.DestroyAllWindows (view/add comments)

Destroys all of the HighGUI windows.

```
DestroyAllWindows()-> None
```

The function `cvDestroyAllWindows` destroys all of the opened HighGUI windows.

---

cv.DestroyWindow (view/add comments)

Destroys a window.

```
DestroyWindow(name)-> None
```

name Name of the window to be destroyed.

The function `cvDestroyWindow` destroys the window with the given name.

---

cv.GetTrackbarPos (view/add comments)

Returns the trackbar position.

```
GetTrackbarPos(trackbarName,windowName)-> None
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of the trackbar.

The function `cvGetTrackbarPos` returns the current position of the specified trackbar.

[Qt Backend Only] qt-specific details:

windowName Name of the window which is the parent of the trackbar. Can be NULL if the trackbar is attached to the control panel.

---

cv.MoveWindow (view/add comments)

Sets the position of the window.

```
MoveWindow(name,x,y)-> None
```

name Name of the window to be moved.

x New x coordinate of the top-left corner

y New y coordinate of the top-left corner

The function `cvMoveWindow` changes the position of the window.

---

cv.NamedWindow (view/add comments)

Creates a window.

```
NamedWindow(name,flags=CV_WINDOW_AUTOSIZE)-> None
```

name Name of the window in the window caption that may be used as a window identifier.

flags Flags of the window. Currently the only supported flag is `CV_WINDOW_AUTOSIZE`. If this is set, window size is automatically adjusted to fit the displayed image (see ShowImage), and the user can not change the window size manually.

The function `cvNamedWindow` creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing.

[Qt Backend Only] qt-specific details:

flags Flags of the window. Currently the supported flags are:

CV_WINDOW_NORMAL or CV_WINDOW_AUTOSIZE: `CV_WINDOW_NORMAL` let the user resize the window, whereas `CV_WINDOW_AUTOSIZE` adjusts automatically the window's size to fit the displayed image (see ShowImage), and the user can not change the window size manually.

CV_WINDOW_FREERATIO or CV_WINDOW_KEEPRATIO: `CV_WINDOW_FREERATIO` adjust the image without respect the its ration, whereas `CV_WINDOW_KEEPRATIO` keep the image's ratio.

CV_GUI_NORMAL or CV_GUI_EXPANDED: `CV_GUI_NORMAL` is the old way to draw the window without statusbar and toolbar, whereas `CV_GUI_EXPANDED` is the new enhance GUI.

This parameter is optional. The default flags set for a new window are `CV_WINDOW_AUTOSIZE`, `CV_WINDOW_KEEPRATIO`, and `CV_GUI_EXPANDED`.

However, if you want to modify the flags, you can combine them using OR operator, ie:

```
V_WINDOW_NORMALV_GUI_NORMAL
```

cv.ResizeWindow (view/add comments)

Sets the window size.

```
ResizeWindow(name,width,height)-> None
```

name Name of the window to be resized.

width New width

height New height

The function `cvResizeWindow` changes the size of the window.

cv.SetMouseCallback (view/add comments)

Assigns callback for mouse events.

```
SetMouseCallback(windowName, onMouse, param) -> None
```

windowName Name of the window.

onMouse Callable to be called every time a mouse event occurs in the specified window. This callable should have signature `Foo(event, x, y, flags, param)-> None` where `event` is one of `CV_EVENT_*`, `x` and `y` are the coordinates of the mouse pointer in image coordinates (not window coordinates), `flags` is a combination of `CV_EVENT_FLAG_*`, and `param` is a user-defined parameter passed to the `cvSetMouseCallback` function call.

param User-defined parameter to be passed to the callback function.

The function `cvSetMouseCallback` sets the callback function for mouse events occuring within the specified window.

The `event` parameter is one of:

CV_EVENT_MOUSEMOVE Mouse movement

CV_EVENT_LBUTTONDOWN Left button down

CV_EVENT_RBUTTONDOWN Right button down

CV_EVENT_MBUTTONDOWN Middle button down

CV_EVENT_LBUTTONUP Left button up

CV_EVENT_RBUTTONUP Right button up

CV_EVENT_MBUTTONUP Middle button up

CV_EVENT_LBUTTONDBLCLK Left button double click

CV_EVENT_RBUTTONDBLCLK Right button double click

CV_EVENT_MBUTTONDBLCLK Middle button double click

The `flags` parameter is a combination of :

CV_EVENT_FLAG_LBUTTON Left button pressed

CV_EVENT_FLAG_RBUTTON Right button pressed

CV_EVENT_FLAG_MBUTTON Middle button pressed

CV_EVENT_FLAG_CTRLKEY Control key pressed

CV_EVENT_FLAG_SHIFTKEY Shift key pressed

CV_EVENT_FLAG_ALTKEY Alt key pressed

---

cv.SetTrackbarPos (view/add comments)

Sets the trackbar position.

```
SetTrackbarPos(trackbarName,windowName,pos)-> None
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of trackbar.

pos New position.

The function `cvSetTrackbarPos` sets the position of the specified trackbar.

[Qt Backend Only] qt-specific details:

windowName Name of the window which is the parent of trackbar. Can be NULL if the trackbar is attached to the control panel.

---

cv.ShowImage (view/add comments)

Displays the image in the specified window

```
ShowImage(name,image)-> None
```

name Name of the window.

image Image to be shown.

The function `cvShowImage` displays the image in the specified window. If the window was created with the `CV_WINDOW_AUTOSIZE` flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

---

cv.WaitKey (view/add comments)

Waits for a pressed key.

```
WaitKey(delay=0)-> int
```

delay Delay in milliseconds.

The function cvWaitKey waits for key event infinitely ($delay <= 0$) or for delay milliseconds. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

[Qt Backend Only] qt-specific details: With this current Qt implementation, this is the only way to process event such as repaint for the windows, and so on ...

28.2 Reading and Writing Images and Video

---

cv.LoadImage (view/add comments)

Loads an image from a file as an IplImage.

```
LoadImage(filename, iscolor=CV_LOAD_IMAGE_COLOR)->None
```

filename Name of file to be loaded.

iscolor Specific color type of the loaded image:

- **CV_LOAD_IMAGE_COLOR** the loaded image is forced to be a 3-channel color image
- **CV_LOAD_IMAGE_GRAYSCALE** the loaded image is forced to be grayscale
- **CV_LOAD_IMAGE_UNCHANGED** the loaded image will be loaded as is.

The function cvLoadImage loads an image from the specified file and returns the pointer to the loaded image. Currently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

Note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB.

---

cv.LoadImageM (view/add comments)

Loads an image from a file as a CvMat.

```
LoadImageM(filename, iscolor=CV_LOAD_IMAGE_COLOR)->None
```

filename Name of file to be loaded.

iscolor Specific color type of the loaded image:

CV_LOAD_IMAGE_COLOR the loaded image is forced to be a 3-channel color image

CV_LOAD_IMAGE_GRAYSCALE the loaded image is forced to be grayscale

CV_LOAD_IMAGE_UNCHANGED the loaded image will be loaded as is.

The function `cvLoadImageM` loads an image from the specified file and returns the pointer to the loaded image. urrently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

Note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB.

cv.SaveImage (view/add comments)

Saves an image to a specified file.

```
SaveImage(filename,image)-> None
```

filename Name of the file.

image Image to be saved.

The function `cvSaveImage` saves the image to the specified file. The image format is chosen based on the `filename` extension, see LoadImage . Only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use `cvCvtScale` and `cvCvtColor` to convert it before saving, or use universal `cvSave` to save the image to XML or YAML format.

CvCapture (view/add comments)

Video capturing structure.

The structure `CvCapture` does not have a public interface and is used only as a parameter for video capturing functions.

cv.CaptureFromCAM (view/add comments)

Initializes capturing a video from a camera.

```
CaptureFromCAM(index) -> CvCapture
```

index Index of the camera to be used. If there is only one camera or it does not matter what camera is used -1 may be passed.

The function `cvCaptureFromCAM` allocates and initializes the CvCapture structure for reading a video stream from the camera. Currently two camera interfaces can be used on Windows: Video for Windows (VFW) and Matrox Imaging Library (MIL); and two on Linux: V4L and FireWire (IEEE1394).

To release the structure, use ReleaseCapture .

cv.CaptureFromFile (view/add comments)

Initializes capturing a video from a file.

```
CaptureFromFile(filename) -> CvCapture
```

filename Name of the video file.

The function `cvCaptureFromFile` allocates and initializes the CvCapture structure for reading the video stream from the specified file. Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect and how to prepare your video files.

After the allocated structure is not used any more it should be released by the ReleaseCapture function.

cv.GetCaptureProperty (view/add comments)

Gets video capturing properties.

```
GetCaptureProperty(capture, property_id)->double
```

capture video capturing structure.

property_id Property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec

CV_CAP_PROP_FRAME_COUNT Number of frames in the video file

CV_CAP_PROP_FORMAT The format of the Mat objects returned by retrieve()

CV_CAP_PROP_MODE A backend-specific value indicating the current capture mode

CV_CAP_PROP_BRIGHTNESS Brightness of the image (only for cameras)

CV_CAP_PROP_CONTRAST Contrast of the image (only for cameras)

CV_CAP_PROP_SATURATION Saturation of the image (only for cameras)

CV_CAP_PROP_HUE Hue of the image (only for cameras)

CV_CAP_PROP_GAIN Gain of the image (only for cameras)

CV_CAP_PROP_EXPOSURE Exposure (only for cameras)

CV_CAP_PROP_CONVERT_RGB Boolean flags indicating whether images should be converted to RGB

CV_CAP_PROP_WHITE_BALANCE Currently unsupported

CV_CAP_PROP_RECTIFICATION TOWRITE (note: only supported by DC1394 v 2.x backend currently)

The function `cvGetCaptureProperty` retrieves the specified property of the camera or video file.

---

cv.GrabFrame (view/add comments)

Grabs the frame from a camera or file.

```
GrabFrame(capture) -> int
```

capture video capturing structure.

The function `cvGrabFrame` grabs the frame from a camera or file. The grabbed frame is stored internally. The purpose of this function is to grab the frame *quickly* so that syncronization can occur if it has to read from several cameras simultaneously. The grabbed frames are not exposed because they may be stored in a compressed format (as defined by the camera/driver). To retrieve the grabbed frame, RetrieveFrame should be used.

cv.QueryFrame (view/add comments)

Grabs and returns a frame from a camera or file.

```
QueryFrame(capture) -> iplimage
```

capture video capturing structure.

The function `cvQueryFrame` grabs a frame from a camera or video file, decompresses it and returns it. This function is just a combination of GrabFrame and RetrieveFrame , but in one call. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cv.RetrieveFrame (view/add comments)

Gets the image grabbed with cvGrabFrame.

```
RetrieveFrame(capture) -> iplimage
```

capture video capturing structure.

The function `cvRetrieveFrame` returns the pointer to the image grabbed with the GrabFrame function. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cv.SetCaptureProperty (view/add comments)

Sets video capturing properties.

```
SetCaptureProperty(capture, property_id,value)->None
```

capture video capturing structure.

property_id property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file
- **CV_CAP_PROP_FORMAT** The format of the Mat objects returned by retrieve()
- **CV_CAP_PROP_MODE** A backend-specific value indicating the current capture mode
- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
- **CV_CAP_PROP_HUE** Hue of the image (only for cameras)
- **CV_CAP_PROP_GAIN** Gain of the image (only for cameras)
- **CV_CAP_PROP_EXPOSURE** Exposure (only for cameras)
- **CV_CAP_PROP_CONVERT_RGB** Boolean flags indicating whether images should be converted to RGB
- **CV_CAP_PROP_WHITE_BALANCE** Currently unsupported
- **CV_CAP_PROP_RECTIFICATION** TOWRITE (note: only supported by DC1394 v 2.x backend currently)

value value of the property.

The function `cvSetCaptureProperty` sets the specified property of video capturing. Currently the function supports only video files: `CV_CAP_PROP_POS_MSEC`, `CV_CAP_PROP_POS_FRAMES`, `CV_CAP_PROP_POS_AVI_RATIO`.

NB This function currently does nothing when using the latest CVS download on linux with FFMPEG (the function contents are hidden if 0 is used and returned).

cv.CreateVideoWriter (view/add comments)

Creates the video file writer.

```
CreateVideoWriter(filename, fourcc, fps, frame_size, is_color) ->
CvVideoWriter
```

filename Name of the output video file.

fourcc 4-character code of codec used to compress the frames. For example, `CV_FOURCC('P','I','M','1')` is a MPEG-1 codec, `CV_FOURCC('M','J','P','G')` is a motion-jpeg codec etc. Under Win32 it is possible to pass -1 in order to choose compression method and additional compression parameters from dialog. Under Win32 if 0 is passed while using an avi filename it will create a video writer that creates an uncompressed avi file.

fps Framerate of the created video stream.

frame_size Size of the video frames.

is_color If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only).

The function `cvCreateVideoWriter` creates the video writer structure.

Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect.

cv.WriteFrame (view/add comments)

Writes a frame to a video file.

```
WriteFrame(writer, image)->int
```

writer Video writer structure

image The written frame

The function `cvWriteFrame` writes/appends one frame to a video file.

Chapter 29

calib3d. Camera Calibration, Pose Estimation and Stereo

29.1 Camera Calibration and 3d Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re-used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distortion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$
where $r^2 = x'^2 + y'^2$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

k_1, k_2, k_3, k_4, k_5, k_6 are radial distortion coefficients, p_1, p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters. *And they remain the same regardless of the captured image resolution.* That is, if, for example, a camera has been calibrated on images of 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x, f_y, c_x and c_y need to be scaled appropriately).

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspondences).

- Estimate the relative position and orientation of the stereo camera "heads" and compute the *rectification* transformation that makes the camera optical axes parallel.

---

cv.CalibrateCamera2 (view/add comments)

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
CalibrateCamera2(objectPoints,imagePoints,pointCounts,imageSize,cameraMatrix,distCo
None
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints The joint matrix of object points projections in the camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints` (=N).

imageSize Size of the image, used only to initialize the intrinsic camera matrix

cameraMatrix The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized before calling the function

distCoeffs The output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements

rvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of rotation vectors (see cv.Rodrigues2), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0..*M*-1)

tvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of translation vectors, estimated for each pattern view.

flags Different flags, may be 0 or combination of the following values:

CV_CALIB_USE_INTRINSIC_GUESS `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`imageSize` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate the extrinsic parameters. Use cv.FindExtrinsicCameraParams2 instead.

CV_CALIB_FIX_PRINCIPAL_POINT The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.

CV_CALIB_FIX_ASPECT_RATIO The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix`.
When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.

CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6 Do not change the corresponding radial distortion coefficient during the optimization. If `CV_CALIB_USE_INTRINSIC_GUESS` is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.

CV_CALIB_RATIONAL_MODEL Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute only 5 distortion coefficients.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see cv.FindChessboardCorners). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).

2. The initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using cv.FindExtrinsicCameraParams2

3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points `imagePoints` and the projected (using the current estimates for camera parameters and the poses) object points `objectPoints`; see cv.ProjectPoints2.

Note: if you're using a non-square (=non-NxN) grid and cv. for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2-0.5, h/2-0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probably used `patternSize=cvSize(rows,cols)`, but should use `patternSize=cvSize(cols,rows)` in cv.FindChessboardCorners.

See also: cv.FindChessboardCorners, cv.FindExtrinsicCameraParams2, cv., cv.StereoCalibrate, cv.Undistort2

---

cv.ComputeCorrespondEpilines (view/add comments)

For points in one image of a stereo pair, computes the corresponding epilines in the other image.

```
ComputeCorrespondEpilines(points, whichImage, F, lines) -> None
```

points The input points. `2xN, Nx2, 3xN` or `Nx3` array (where `N` number of points). Multi-channel `1xN` or `Nx1` array is also acceptable

whichImage Index of the image (1 or 2) that contains the `points`

F The fundamental matrix that can be estimated using cv.FindFundamentalMat or cv.StereoRectify.

lines The output epilines, a `3xN` or `Nx3` array. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see cv.FindFundamentalMat), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

---

cv.ConvertPointsHomogeneous (view/add comments)

Convert points to/from homogeneous coordinates.

```
ConvertPointsHomogeneous( src, dst ) -> None
```

src The input array or vector of 2D, 3D or 4D points

dst The output vector of 2D or 2D points

The 2D or 3D points from/to homogeneous coordinates, or simply the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$$x' = x/w$$
$$y' = y/w$$
$$z' = z/w \quad \text{(if output is 3D)}$$

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

---

cv.CreatePOSITObject (view/add comments)

Initializes a structure containing object information.

```
CreatePOSITObject(points)-> POSITObject
```

points List of 3D points

The function allocates memory for the object structure and computes the object inverse matrix.

The preprocessed object data is stored in the structure cv.CvPOSITObject, internal for OpenCV, which means that the user cannot directly access the structure data. The user may only create this structure and pass its pointer to the function.

An object is defined as a set of points given in a coordinate system. The function cv.POSIT computes a vector that begins at a camera-related coordinate system center and ends at the `points[0]` of the object.

Once the work with a given object is finished, the function cv.ReleasePOSITObject must be called to free memory.

---

cv.CreateStereoBMState (view/add comments)

Creates block matching stereo correspondence structure.

```
CreateStereoBMState(preset=CV_STEREO_BM_BASIC,numberOfDisparities=0)->
StereoBMState
```

preset ID of one of the pre-defined parameter sets. Any of the parameters can be overridden after creating the structure. Values are

- **CV_STEREO_BM_BASIC** Parameters suitable for general cameras
- **CV_STEREO_BM_FISH_EYE** Parameters suitable for wide-angle cameras
- **CV_STEREO_BM_NARROW** Parameters suitable for narrow-angle cameras

numberOfDisparities The number of disparities. If the parameter is 0, it is taken from the preset, otherwise the supplied value overrides the one from preset.

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cv.FindStereoCorrespondenceBM.

---

cv.CreateStereoGCState (view/add comments)

Creates the state of graph cut-based stereo correspondence algorithm.

```
CreateStereoGCState(numberOfDisparities,maxIters)-> StereoGCState
```

numberOfDisparities The number of disparities. The disparity search range will be `state->minDisparity` $\leq disparity <$ `state->minDisparity` + `state->numberOfDisparities`

maxIters Maximum number of iterations. On each iteration all possible (or reasonable) alpha-expansions are tried. The algorithm may terminate earlier if it could not find an alpha-expansion that decreases the overall cost function value. See [**?**] for details.

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cv.FindStereoCorrespondenceGC.

CvStereoBMState (view/add comments)

The structure for block matching stereo correspondence algorithm.

preFilterType type of the prefilter, `CV_STEREO_BM_NORMALIZED_RESPONSE` or the default and the recommended `CV_STEREO_BM_XSOBEL`, int

preFilterSize 5x5..21x21, int

preFilterCap up to 31, int

SADWindowSize Could be 5x5..21x21 or higher, but with 21x21 or smaller windows the processing speed is much higher, int

minDisparity minimum disparity (=0), int

numberOfDisparities maximum disparity - minimum disparity, int

textureThreshold the textureness threshold. That is, if the sum of absolute values of x-derivatives computed over `SADWindowSize` by `SADWindowSize` pixel neighborhood is smaller than the parameter, no disparity is computed at the pixel, int

uniquenessRatio the minimum margin in percents between the best (minimum) cost function value and the second best value to accept the computed disparity, int

speckleWindowSize the maximum area of speckles to remove (set to 0 to disable speckle filtering), int

speckleRange acceptable range of disparity variation in each connected component, int

trySmallerWindows not used currently (0), int

roi1, roi2 These are the clipping ROIs for the left and the right images. The function cv.StereoRectify returns the largest rectangles in the left and right images where after the rectification all the pixels are valid. If you copy those rectangles to the `CvStereoBMState` structure, the stereo correspondence function will automatically clear out the pixels outside of the "valid" disparity rectangle computed by cv.GetValidDisparityROI. Thus you will get more "invalid disparity" pixels than usual, but the remaining pixels are more probable to be valid.

disp12MaxDiff The maximum allowed difference between the explicitly computed left-to-right disparity map and the implicitly (by cv.ValidateDisparity) computed right-to-left disparity. If for some pixel the difference is larger than the specified threshold, the disparity at the pixel is invalidated. By default this parameter is set to (-1), which means that the left-right check is not performed.

The block matching stereo correspondence algorithm, by Kurt Konolige, is very fast single-pass stereo matching algorithm that uses sliding sums of absolute differences between pixels in the left image and the pixels in the right image, shifted by some varying amount of pixels (from `minDisparity` to `minDisparity+numberOfDisparities`). On a pair of images WxH the algorithm computes disparity in `O(W*H*numberOfDisparities)` time. In order to improve quality and readability of the disparity map, the algorithm includes pre-filtering and post-filtering procedures.

Note that the algorithm searches for the corresponding blocks in x direction only. It means that the supplied stereo pair should be rectified. Vertical stereo layout is not directly supported, but in such a case the images could be transposed by user.

CvStereoGCState (view/add comments)

The structure for graph cuts-based stereo correspondence algorithm

Ithreshold threshold for piece-wise linear data cost function (5 by default)

interactionRadius radius for smoothness cost function (1 by default; means Potts model)

K, lambda, lambda1, lambda2 parameters for the cost function (usually computed adaptively from the input data)

occlusionCost 10000 by default

minDisparity 0 by default; see CvStereoBMState

numberOfDisparities defined by user; see CvStereoBMState

maxIters number of iterations; defined by user.

The graph cuts stereo correspondence algorithm, described in [**?**] (as **KZ1**), is non-realtime stereo correspondence algorithm that usually gives very accurate depth map with well-defined object boundaries. The algorithm represents stereo problem as a sequence of binary optimization problems, each of those is solved using maximum graph flow algorithm. The state structure above should not be allocated and initialized manually; instead, use cv.CreateStereoGCState and then override necessary parameters if needed.

---

cv.DecomposeProjectionMatrix (view/add comments)

Decomposes the projection matrix into a rotation matrix and a camera matrix.

```
DecomposeProjectionMatrix(projMatrix, cameraMatrix, rotMatrix,
transVect, rotMatrX = None, rotMatrY = None, rotMatrZ = None) ->
eulerAngles
```

projMatrix The 3x4 input projection matrix P

cameraMatrix The output 3x3 camera matrix K

rotMatrix The output 3x3 external rotation matrix R

transVect The output 4x1 translation vector T

rotMatrX Optional 3x3 rotation matrix around x-axis

rotMatrY Optional 3x3 rotation matrix around y-axis

rotMatrZ Optional 3x3 rotation matrix around z-axis

eulerAngles Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on cv.RQDecomp3x3.

cv.DrawChessboardCorners (view/add comments)

Renders the detected chessboard corners.

```
DrawChessboardCorners(image,patternSize,corners,patternWasFound)-> None
```

image The destination image; it must be an 8-bit color image

patternSize The number of inner corners per chessboard row and column. (patternSize = cv::Size(points_per_row,points_per_column) = cv::Size(rows,columns))

corners The array of corners detected, this should be the output from findChessboardCorners wrapped in a cv::Mat().

patternWasFound Indicates whether the complete board was found $(\neq 0)$ or not $(= 0)$. One may just pass the return value cv.FindChessboardCornersfindChessboardCorners here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

cv.FindChessboardCorners (view/add comments)

Finds the positions of the internal corners of the chessboard.

```
FindChessboardCorners(image, patternSize, flags=CV_CALIB_CB_ADAPTIVE_THRESH)
-> corners
```

image Source chessboard view; it must be an 8-bit grayscale or color image

patternSize The number of inner corners per chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The output array of corners detected

flags Various operation flags, can be 0 or a combination of the following values:

CV_CALIB_CB_ADAPTIVE_THRESH use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).

CV_CALIB_CB_NORMALIZE_IMAGE normalize the image gamma with cv.EqualizeHist before applying fixed or adaptive thresholding.

CV_CALIB_CB_FILTER_QUADS use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

CALIB_CB_FAST_CHECK Runs a fast check on the image that looks for chessboard corners, and shortcuts the call if none are found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cv.FindCornerSubPix.

Sample usage of detecting and drawing chessboard corners:

```
Size patternsize(8,6); //interior number of corners
Mat gray = ....; //source image
vector<Point2f> corners; //this will be filled by the detected corners

//CALIB_CB_FAST_CHECK saves a lot of time on images
//that don't contain any chessboard corners
bool patternfound = findChessboardCorners(gray, patternsize, corners,
        CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE
        + CALIB_CB_FAST_CHECK);

if(patternfound)
  cornerSubPix(gray, corners, Size(11, 11), Size(-1, -1),
    TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

drawChessboardCorners(img, patternsize, Mat(corners), patternfound);
```

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

cv.FindExtrinsicCameraParams2 (view/add comments)

Finds the object pose from the 3D-2D point correspondences

```
FindExtrinsicCameraParams2(objectPoints,imagePoints,cameraMatrix,distCoeffs,rvec,tv
None
```

objectPoints The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points.

imagePoints The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points.

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

rvec The output rotation vector (see cv.Rodrigues2) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system

tvec The output translation vector

useExtrinsicGuess If true (1), the function will use the provided `rvec` and `tvec` as the initial approximations of the rotation and translation vectors, respectively, and will further optimize them.

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using cv.ProjectPoints2) `objectPoints`.

The function's counterpart in the C++ API is cv::solvePnP

cv.FindFundamentalMat (view/add comments)

Calculates the fundamental matrix from the corresponding points in two images.

```
FindFundamentalMat(points1, points2, fundamentalMatrix,
method=CV_FM_RANSAC, param1=1., param2=0.99, status = None) -> None
```

points1 Array of `N` points from the first image. It can be `2xN, Nx2, 3xN` or `Nx3` 1-channel array or `1xN` or `Nx1` 2- or 3-channel array . The point coordinates should be floating-point (single or double precision)

points2 Array of the second image points of the same size and format as `points1`

fundamentalMatrix The output fundamental matrix or matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices)

method Method for computing the fundamental matrix

CV_FM_7POINT for a 7-point algorithm. $N = 7$

CV_FM_8POINT for an 8-point algorithm. $N \geq 8$

CV_FM_RANSAC for the RANSAC algorithm. $N \geq 8$

CV_FM_LMEDS for the LMedS algorithm. $N \geq 8$

param1 The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise

param2 The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct

status The optional output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1's

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the number of fundamental matrices found (1 or 3) and 0, if no matrix is found . Normally

just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to cv.ComputeCorrespondEpilines that finds the epipolar lines corresponding to the specified points. It can also be passed to cv.StereoRectifyUncalibrated to compute the rectification transformation.

---

cv.FindHomography (view/add comments)

Finds the perspective transformation between two planes.

```
FindHomography(srcPoints,dstPoints,H,method,ransacReprojThreshold=3.0,
status=None)-> None
```

srcPoints Coordinates of the points in the original plane, 2xN, Nx2, 3xN or Nx3 1-channel array (the latter two are for representation in homogeneous coordinates), where N is the number of points. 1xN or Nx1 2- or 3-channel array can also be passed.

dstPoints Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 1-channel, or 1xN or Nx1 2- or 3-channel array.

H The output 3x3 homography matrix

method The method used to computed homography matrix; one of the following:

- **0** a regular method using all the points
- **CV_RANSAC** RANSAC-based robust method
- **CV_LMEDS** Least-Median robust method

ransacReprojThreshold The maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.

status The optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). *Note that the input mask values are ignored.*

The function finds the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter `method` is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, `RANSAC` and `LMeDS`, try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection error for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method `RANSAC` can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method `LMeDS` does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features, where can be only some small noise present, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized so that $h_{33} = 1$.

See also: cv.GetAffineTransform, cv.GetPerspectiveTransform, cv.EstimateRigidMotion, cv.WarpPerspective, cv.PerspectiveTransform

---

cv.FindStereoCorrespondenceBM (view/add comments)

Computes the disparity map using block matching algorithm.

```
FindStereoCorrespondenceBM(left,right,disparity,state)-> None
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

disparity The output single-channel 16-bit signed, or 32-bit floating-point disparity map of the same size as input images. In the first case the computed disparities are represented as fixed-point numbers with 4 fractional bits (i.e. the computed disparity values are multiplied by 16 and rounded to integers).

state Stereo correspondence structure.

The function cvFindStereoCorrespondenceBM computes disparity map for the input rectified stereo pair. Invalid pixels (for which disparity can not be computed) are set to `state->minDisparity - 1` (or to `(state->minDisparity-1)*16` in the case of 16-bit fixed-point disparity map)

---

cv.FindStereoCorrespondenceGC (view/add comments)

Computes the disparity map using graph cut-based algorithm.

```
FindStereoCorrespondenceGC( left, right, dispLeft, dispRight, state,
useDisparityGuess=(0))-> None
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

dispLeft The optional output single-channel 16-bit signed left disparity map of the same size as input images.

dispRight The optional output single-channel 16-bit signed right disparity map of the same size as input images.

state Stereo correspondence structure.

useDisparityGuess If the parameter is not zero, the algorithm will start with pre-defined disparity maps. Both dispLeft and dispRight should be valid disparity maps. Otherwise, the function starts with blank disparity maps (all pixels are marked as occlusions).

The function computes disparity maps for the input rectified stereo pair. Note that the left disparity image will contain values in the following range:

$$-\texttt{state->numberOfDisparities}-\texttt{state->minDisparity} < dispLeft(x,y) \leq -\texttt{state->minDisparity}$$

or

$$dispLeft(x,y) == \texttt{CV_STEREO_GC_OCCLUSION}$$

and for the right disparity image the following will be true:

$$\texttt{state->minDisparity} \leq dispRight(x,y) < \texttt{state->minDisparity}+\texttt{state->numberOfDisparities}$$

or

$$dispRight(x,y) == \texttt{CV_STEREO_GC_OCCLUSION}$$

that is, the range for the left disparity image will be inversed, and the pixels for which no good match has been found, will be marked as occlusions.

Here is how the function can be used:

```
import sys
import cv

def findstereocorrespondence(image_left, image_right):
    # image_left and image_right are the input 8-bit single-channel images
    # from the left and the right cameras, respectively
    (r, c) = (image_left.rows, image_left.cols)
    disparity_left = cv.CreateMat(r, c, cv.CV_16S)
    disparity_right = cv.CreateMat(r, c, cv.CV_16S)
    state = cv.CreateStereoGCState(16, 2)
    cv.FindStereoCorrespondenceGC(image_left, image_right, disparity_left, disparity_right,
    return (disparity_left, disparity_right)


if __name__ == '__main__':

    (l, r) = [cv.LoadImageM(f, cv.CV_LOAD_IMAGE_GRAYSCALE) for f in sys.argv[1:]]

    (disparity_left, disparity_right) = findstereocorrespondence(l, r)
```

```
disparity_left_visual = cv.CreateMat(l.rows, l.cols, cv.CV_8U)
cv.ConvertScale(disparity_left, disparity_left_visual, -16)
cv.SaveImage("disparity.pgm", disparity_left_visual)
```

and this is the output left disparity image computed from the well-known Tsukuba stereo pair and multiplied by -16 (because the values in the left disparity images are usually negative):

---

cv.GetOptimalNewCameraMatrix (view/add comments)

Returns the new camera matrix based on the free scaling parameter

```
GetOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, alpha,
newCameraMatrix, newImageSize=(0,0), validPixROI=0) -> None
```

cameraMatrix The input camera matrix

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imageSize The original image size

alpha The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see cv.StereoRectify

newCameraMatrix The output new camera matrix.

newImageSize The image size after rectification. By default it will be set to `imageSize`.

validPixROI The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in cv.StereoRectify

The function computes the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0`, keep all the original image pixels if there is valuable information in the corners `alpha=1`, or get something in between. When `alpha>0`, the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to cv.InitUndistortRectifyMap to produce the maps for cv.Remap.

---

cv.InitIntrinsicParams2D (view/add comments)

Finds the initial camera matrix from the 3D-2D point correspondences

```
InitIntrinsicParams2D(objectPoints, imagePoints, npoints, imageSize,
cameraMatrix, aspectRatio=1.) -> None
```

objectPoints The joint array of object points; see cv.CalibrateCamera2

imagePoints The joint array of object point projections; see cv.CalibrateCamera2

npoints The array of point counts; see cv.CalibrateCamera2

imageSize The image size in pixels; used to initialize the principal point

cameraMatrix The output camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

aspectRatio If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

---

cv.InitUndistortMap (view/add comments)

Computes an undistortion map.

```
InitUndistortMap(cameraMatrix,distCoeffs,map1,map2)-> None
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function is a simplified variant of cv.InitUndistortRectifyMap where the rectification transformation `R` is identity matrix and `newCameraMatrix=cameraMatrix`.

---

cv.InitUndistortRectifyMap (view/add comments)

Computes the undistortion and rectification transformation map.

```
InitUndistortRectifyMap(cameraMatrix,distCoeffs,R,newCameraMatrix,map1,map2)->
None
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R The optional rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv.StereoRectify can be passed here. If the matrix is NULL , the identity transformation is assumed

newCameraMatrix The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for cv.Remap. The undistorted image will look like the original, as if it was captured with a camera with camera matrix `=newCameraMatrix` and zero distortion. In the case of monocular camera `newCameraMatrix` is usually equal to `cameraMatrix`, or it can be computed by cv.GetOptimalNewCameraMatrix for a better control over scaling. In the case of stereo camera `newCameraMatrix` is normally set to `P1` or `P2` computed by cv.StereoRectify.

Also, this new camera will be oriented differently in the coordinate space, according to `R`. That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images become horizontal and have the same y- coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by cv.Remap. That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$
\begin{array}{l}
x \leftarrow (u - {c'}_x)/{f'}_x \\
y \leftarrow (v - {c'}_y)/{f'}_y \\
[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\
x' \leftarrow X/W \\
y' \leftarrow Y/W \\
x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
map_x(u,v) \leftarrow x" f_x + c_x \\
map_y(u,v) \leftarrow y" f_y + c_y
\end{array}
$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after cv.StereoRectify, which in its turn is called after cv.StereoCalibrate. But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using cv.StereoRectifyUncalibrated. For each camera the function computes homography `H` as the rectification transformation in pixel domain, not a rotation matrix `R` in 3D space. The `R` can be computed from `H` as

$$\texttt{R} = \texttt{cameraMatrix}^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

---

cv.POSIT (view/add comments)

Implements the POSIT algorithm.

```
POSIT(posit_object,imagePoints,focal_length,criteria)->
(rotationMatrix,translation_vector)
```

posit_object Pointer to the object structure

imagePoints Pointer to the object points projections on the 2D image plane

focal_length Focal length of the camera used

criteria Termination criteria of the iterative POSIT algorithm

rotationMatrix Matrix of rotations

translation_vector Translation vector

The function implements the POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using the camera calibration functions. At every iteration of the algorithm a new perspective projection of the estimated pose is computed.

Difference norm between two projections is the maximal distance between corresponding points. The parameter `criteria.epsilon` serves to stop the algorithm if the difference is small.

---

cv.ProjectPoints2 (view/add comments)

Project 3D points on to an image plane.

```
ProjectPoints2(objectPoints,rvec,tvec,cameraMatrix,distCoeffs,
imagePoints,dpdrot=NULL,dpdt=NULL,dpdf=NULL,dpdc=NULL,dpddist=NULL)->
None
```

objectPoints The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel , where N is the number of points in the view

rvec The rotation vector, see cv.Rodrigues2

tvec The translation vector

cameraMatrix The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

imagePoints The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel

dpdrot Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector

dpdt Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector

dpdf Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y

dpdc Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y

dpddist Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in cv.CalibrateCamera2, cv.FindExtrinsicCameraParams2 and cv.StereoCalibrate. The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

---

cv.ReprojectImageTo3D (view/add comments)

Reprojects disparity image to 3D space.

```
ReprojectImageTo3D(disparity, _3dImage, Q, handleMissingValues=0) ->
None
```

disparity The input single-channel 16-bit signed or 32-bit floating-point disparity image

_3dImage The output 3-channel floating-point image of the same size as `disparity`. Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)`, computed from the disparity map.

Q The 4×4 perspective transformation matrix that can be obtained with cv.StereoRectify

handleMissingValues If true, when the pixels with the minimal disparity (that corresponds to the outliers; see cv.FindStereoCorrespondenceBM) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \texttt{Q} * [x\ y\ \texttt{disparity}(x,y)\ 1]^T$$
$$\texttt{_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by cv.StereoRectify. To reproject a sparse set of points (x,y,d),... to 3D space, use cv.PerspectiveTransform.

---

cv.RQDecomp3x3 (view/add comments)

Computes the 'RQ' decomposition of 3x3 matrices.

```
RQDecomp3x3(M, R, Q, Qx = None, Qy = None, Qz = None) -> eulerAngles
```

M The 3x3 input matrix

R The output 3x3 upper-triangular matrix

Q The output 3x3 orthogonal matrix

Qx Optional 3x3 rotation matrix around x-axis

Qy Optional 3x3 rotation matrix around y-axis

Qz Optional 3x3 rotation matrix around z-axis

eulerAngles Optional three Euler angles of rotation

The function computes a RQ decomposition using the given rotations. This function is used in cv.DecomposeProjectionMatrix to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

---

cv.Rodrigues2 (view/add comments)

Converts a rotation matrix to a rotation vector or vice versa.

```
Rodrigues2(src,dst,jacobian=0)-> None
```

src The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)

dst The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively

jacobian Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like cv.CalibrateCamera2, cv.StereoCalibrate or cv.FindExtrinsicCameraParams

---

cv.StereoCalibrate (view/add comments)

Calibrates stereo camera.

```
StereoCalibrate( objectPoints, imagePoints1, imagePoints2, pointCounts,
cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R,
T, E=NULL, F=NULL, term_crit=(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,30,1e-6),
flags=CV_CALIB_FIX_INTRINSIC)-> None
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints1 The joint matrix of object points projections in the first camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

imagePoints2 The joint matrix of object points projections in the second camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints*` (=N).

cameraMatrix1 The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of `CV_CALIB_USE_INTRINSIC_GUESS`, `CV_CALIB_FIX_ASPECT_RATIO`, `CV_CALIB_FIX_INTRINSIC` or `CV_CALIB_FIX_FOCAL_LENGTH` are specified, some or all of the matrices' components must be initialized; see the flags description

distCoeffs The input/output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements.

cameraMatrix2 The input/output second camera matrix, as cameraMatrix1.

distCoeffs2 The input/output lens distortion coefficients for the second camera, as `distCoeffs1`.

imageSize Size of the image, used only to initialize intrinsic camera matrix.

R The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The output translation vector between the cameras' coordinate systems.

E The optional output essential matrix.

F The optional output fundamental matrix.

term_crit The termination criteria for the iterative optimization algorithm.

flags Different flags, may be 0 or combination of the following values:

- **CV_CALIB_FIX_INTRINSIC** If it is set, `cameraMatrix?`, as well as `distCoeffs?` are fixed, so that only `R, T, E` and `F` are estimated.
- **CV_CALIB_USE_INTRINSIC_GUESS** The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.
- **CV_CALIB_FIX_PRINCIPAL_POINT** The principal points are fixed during the optimization.

CV_CALIB_FIX_FOCAL_LENGTH $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.

CV_CALIB_FIX_ASPECT_RATIO $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.

CV_CALIB_SAME_FOCAL_LENGTH Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients for each camera are set to zeros and fixed there.

CV_CALIB_FIX_K1,...,CV_CALIB_FIX_K6 Do not change the corresponding radial distortion coefficient during the optimization. If CV_CALIB_USE_INTRINSIC_GUESS is set, the coefficient from the supplied `distCoeffs` matrix is used, otherwise it is set to 0.

CV_CALIB_RATIONAL_MODEL Enable coefficients k4, k5 and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function will compute only 5 distortion coefficients.

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with cv.FindExtrinsicCameraParams2), obviously, those poses will relate to each other, i.e. given (R_1, T_1) it should be possible to compute (R_2, T_2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector T: $T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform full calibration of each of the 2 cameras. However, because of the high dimensionality of the parameter space and noise in the input data the function can diverge from the correct solution. Thus, if intrinsic parameters can be estimated with high accuracy for each of the cameras individually (e.g. using cv.CalibrateCamera2), it is recommended to do so and then pass CV_CALIB_FIX_INTRINSIC flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are

estimated at once, it makes sense to restrict some parameters, e.g. pass CV_CALIB_SAME_FOCAL_LENGTH and CV_CALIB_ZERO_TANGENT_DIST flags, which are usually reasonable assumptions.

Similarly to cv.CalibrateCamera2, the function minimizes the total re-projection error for all the points in all the available views from both cameras.

---

cv.StereoRectify (view/add comments)

Computes rectification transforms for each head of a calibrated stereo camera.

```
StereoRectify( cameraMatrix1, cameraMatrix2, distCoeffs1, distCoeffs2,
imageSize, R, T, R1, R2, P1, P2, Q=NULL, flags=CV_CALIB_ZERO_DISPARITY,
alpha=-1, newImageSize=(0,0))-> (roi1, roi2)
```

cameraMatrix1, cameraMatrix2 The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs1, distCoeffs2

distCoeffs The input vectors of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements each. If the vectors are NULL/empty, the zero distortion coefficients are assumed.

imageSize Size of the image used for stereo calibration.

R The rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The translation vector between the cameras' coordinate systems.

R1, R2 The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.

P1, P2 The output 3×4 projection matrices in the new (rectified) coordinate systems.

Q The output 4×4 disparity-to-depth mapping matrix, see cv..

flags The operation flags; may be 0 or CV_CALIB_ZERO_DISPARITY. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.

alpha The free scaling parameter. If it is -1 , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.

newImageSize The new image resolution after rectification. The same size should be passed to cv.InitUndistortRectifyMap, see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.

roi1, roi2 The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0`, the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by cv. and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

1. Horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. Vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines

in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

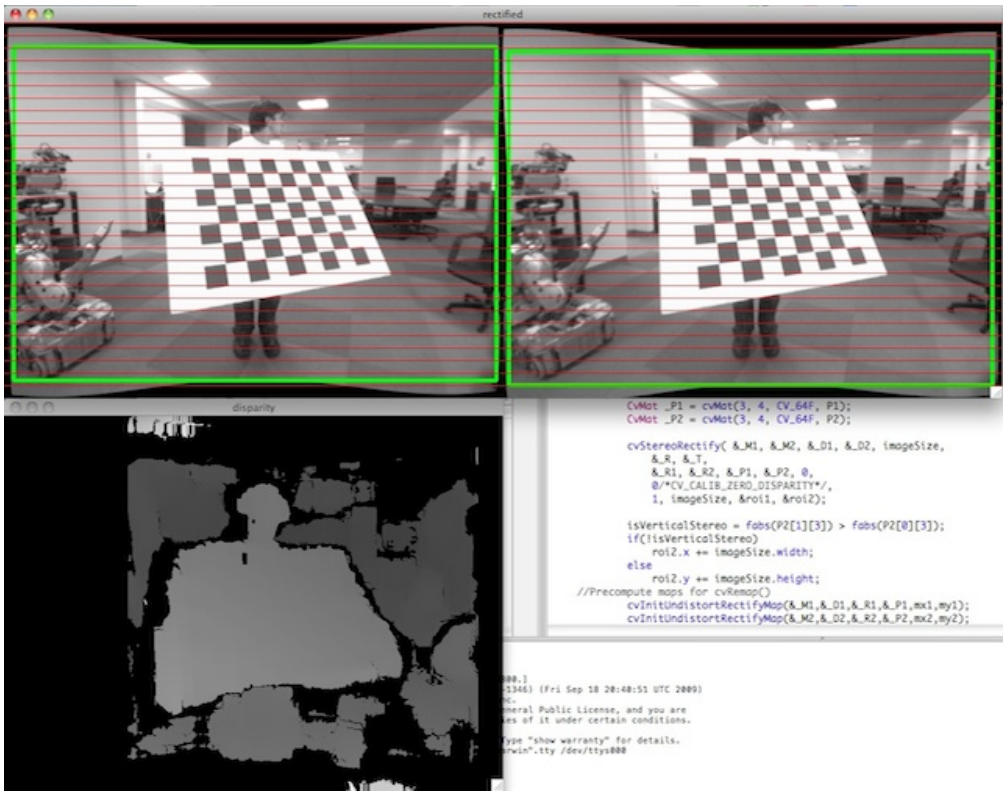
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of `P1` and `P2` will effectively be the new "rectified" camera matrices. The matrices, together with `R1` and `R2`, can then be passed to cv.InitUndistortRectifyMap to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



---

cv.StereoRectifyUncalibrated (view/add comments)

Computes rectification transform for uncalibrated stereo camera.

```
StereoRectifyUncalibrated(points1,points2,F,imageSize,H1,H2,threshold=5)->
None
```

points1, points2 The 2 arrays of corresponding 2D points. The same formats as in cv.FindFundamentalMat are supported

F The input fundamental matrix. It can be computed from the same set of point pairs using cv.FindFundamentalMat.

imageSize Size of the image.

H1, H2 The output rectification homography matrices for the first and for the second images.

threshold The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from cv.StereoRectify is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the algorithm [**?**].

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using cv.CalibrateCamera2 and then the images can be corrected using cv.Undistort2, or just the point coordinates can be corrected with cv.UndistortPoints.

---

cv.Undistort2 (view/add comments)

Transforms an image to compensate for lens distortion.

```
Undistort2(src,dst,cameraMatrix,distCoeffs)-> None
```

src The input (distorted) image

dst The output (corrected) image; will have the same size and the same type as `src`

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of cv.InitUndistortRectifyMap (with unity `R`) and cv.Remap (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix`. You can use cv.GetOptimalNewCameraMatrix to compute the appropriate `newCameraMatrix`, depending on your requirements.

The camera matrix and the distortion parameters can be determined using cv.CalibrateCamera2. If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

---

cv.UndistortPoints (view/add comments)

Computes the ideal point coordinates from the observed point coordinates.

```
UndistortPoints(src,dst,cameraMatrix,distCoeffs,R=NULL,P=NULL)-> None
```

src The observed point coordinates, 1xN or Nx1 2-channel (CV_32FC2 or CV_64FC2).

dst The output ideal point coordinates, after undistortion and reverse perspective transformation , same format as `src` .

cameraMatrix The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs

distCoeffs The input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv. can be passed here. If the matrix is empty, the identity transformation is used

P The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2`, computed by cv. can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to cv.Undistort2 and cv.InitUndistortRectifyMap, but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to cv.ProjectPoints2 (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used both for a stereo camera head or for monocular camera (when R is None).

Chapter 30

ml. Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different seta of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class ‘CvStatModel‘ that all the other ML classes are derived from.

Index

cv...
AdjusterAdapter, 753
BaseColumnFilter, 640
BaseFilter, 640
BaseRowFilter, 641
bitwise_and, 541
bitwise_not, 542
bitwise_or, 543
bitwise_xor, 544
BOWImgDescriptorExtractor, 784
BOWKMeansTrainer, 783
BOWTrainer, 782
BriefDescriptorExtractor, 760
BruteForceMatcher, 767
CalonderDescriptorExtractor, 759
CascadeClassifier, 798
cv::gpu::bitwise_and, 928
cv::gpu::bitwise_not, 927
cv::gpu::bitwise_or, 928
cv::gpu::bitwise_xor, 929
CV_Assert, 628
CV_RGB, 207, 1066
CV_TREE_NODE_FIELDS, 154
CvANN_MLP, 920
CvANN_MLP::create, 922
CvANN_MLP::train, 922
CvANN_MLP_TrainParams, 919
CvArr, 60, 980
CvAttrList, 211
CvBoost, 900
CvBoost::get_weak_predictors, 902
CvBoost::predict, 902
CvBoost::prune, 902
CvBoost::train, 901
CvBoostParams, 899
CvBoostTree, 899
CvCapture, 406, 1182
CvConnectedComp, 295, 1106
CvConvexityDefect, 309, 1119
CvDTree, 894
CvDTree::predict, 896
CvDTree::train, 896
CvDTreeNode, 890
CvDTreeParams, 890
CvDTreeSplit, 889
CvDTreeTrainData, 891
CvEM, 912
CvEM::train, 913
CvEMParams, 911
CvFileNode, 209
CvFileStorage, 208
CvGraph, 152
CvGraphScanner, 153
CvHaarFeature, CvHaarClassifier, CvHaarStage-
Classifier, CvHaarClassifierCascade, 366
CvHistogram, 253, 1071
CvKalman, 383, 1167
CvKNearest, 878
CvKNearest::find_nearest, 880
CvKNearest::train, 879
CvMat, 54, 979
CvMatND, 55, 979
CvMemBlock, 147
CvMemStorage, 146, 1053

CvMemStoragePos, 147
CvNormalBayesClassifier, 877
CvNormalBayesClassifier::predict, 878
CvNormalBayesClassifier::train, 877
cvNulDevReport cvStdErrReport cvGuiBoxReport, 246
CvPoint, 49, 977
CvPoint2D32f, 49, 977
CvPoint2D64f, 51, 977
CvPoint3D32f, 50, 977
CvPoint3D64f, 51, 977
CvQuadEdge2D, 331
CvRect, 52, 978
cvRound, cvFloor, cvCeil, 130
CvRTParams, 904
CvRTrees, 904
CvRTrees::get_proximity, 907
CvRTrees::get_var_importance, 906
CvRTrees::predict, 906
CvRTrees::train, 905
CvScalar, 53, 978
CvSeq, 147, 1053
CvSeqBlock, 150
CvSet, 151, 1054
CvSize, 52, 978
CvSize2D32f, 52, 978
CvSlice, 150
CvSparseMat, 56
CvStatModel, 871
CvStatModel:: CvStatModel, 873
CvStatModel::clear, 873
CvStatModel::CvStatModel, 872
CvStatModel::CvStatModel(...), 872
CvStatModel::load, 874
CvStatModel::predict, 876
CvStatModel::read, 874
CvStatModel::save, 873
CvStatModel::train, 875
CvStatModel::write, 874
CvStereoBMState, 431, 1196
CvStereoGCState, 433, 1197
CvSubdiv2D, 330, 1132
CvSubdiv2DPoint, 332, 1133
CvSURFPoint, 1151
CvSVM, 883
CvSVM::get_default_grid, 886
CvSVM::get_params, 887
CvSVM::get_support_vector*, 887
CvSVM::train, 885
CvSVM::train_auto, 885
CvSVMParams, 884
CvTermCriteria, 54, 978
CvTreeNodeIterator, 154
CvTypeInfo, 211
DescriptorExtractor, 756
DescriptorMatcher, 761
DMatch, 761
DynamicAdaptedFeatureDetector, 751
Exception, 629
FastAdjuster, 755
FastFeatureDetector, 748
FeatureDetector, 745
FeatureEvaluator, 795
FernDescriptorMatcher, 777
FileNode, 623
FileNodeIterator, 624
FileStorage, 621
FilterEngine, 642
FlannBasedMatcher, 769
GenericDescriptorMatcher, 770
GoodFeaturesToTrackDetector, 748
gpu::BaseColumnFilter_GPU, 957
gpu::BaseFilter_GPU, 958
gpu::BaseRowFilter_GPU, 957
gpu::BruteForceMatcher_GPU, 948
gpu::BruteForceMatcher_GPU::knnMatch, 953
gpu::BruteForceMatcher_GPU::knnMatchDownload, 954
gpu::BruteForceMatcher_GPU::makeGpuCollection, 952

gpu::BruteForceMatcher_GPU::match, 950
gpu::BruteForceMatcher_GPU::matchCollection, 951
gpu::BruteForceMatcher_GPU::matchDownload, 952
gpu::BruteForceMatcher_GPU::matchSingle, 950
gpu::BruteForceMatcher_GPU::radiusMatch, 955
gpu::BruteForceMatcher_GPU::radiusMatchDownload, 956
gpu::ConvolveBuf, 936
gpu::createBoxFilter_GPU, 960
gpu::createDerivFilter_GPU, 966
gpu::createFilter2D_GPU, 959
gpu::createGaussianFilter_GPU, 967
gpu::createLinearFilter_GPU, 963
gpu::createMorphologyFilter_GPU, 961
gpu::createSeparableFilter_GPU, 959
gpu::createSeparableLinearFilter_GPU, 965
gpu::FilterEngine_GPU, 958
gpu::getColumnSumFilter_GPU, 960
gpu::getLinearColumnFilter_GPU, 965
gpu::getLinearRowFilter_GPU, 964
gpu::getMaxFilter_GPU, 968
gpu::getMinFilter_GPU, 968
gpu::getRowSumFilter_GPU, 960
gpu::HOGDescriptor, 940
gpu::SURF_GPU, 946
gpu::SURFParams_GPU, 945
GridAdaptedFeatureDetector, 751
IplConvKernel, 266, 1081
IplImage, 57, 979
KalmanFilter, 809
KeyPoint, 744
LineIterator, 618
Mat::pop_back, 517
Mat::push_back, 517
MSER, 729
MserFeatureDetector, 749
OneWayDescriptorMatcher, 776
OpponentColorDescriptorExtractor, 759
PCA, 582
PyramidAdaptedFeatureDetector, 751
QueryHistValue_1D, 1079
QueryHistValue_2D, 1079
QueryHistValue_3D, 1080
QueryHistValue_nD, 1080
RandomizedTree, 733
RTreeClassifier, 737
RTreeNode, 737
SIFT, 731
SiftDescriptorExtractor, 758
SiftFeatureDetector, 750
StarAdjuster, 755
StarDetector, 730
StarFeatureDetector, 749
StereoBM, 857
StereoSGBM, 857
SURF, 733
SurfAdjuster, 755
SurfDescriptorExtractor, 759
SurfFeatureDetector, 750
SVD, 605
VectorDescriptorMatcher, 779
VideoCapture, 819
VideoWriter, 823
cv.AbsDiff, 980
cv.AbsDiffS, 981
cv.Acc, 1139
cv.AdaptiveThreshold, 1098
cv.Add, 981
cv.AddS, 982
cv.AddWeighted, 982
cv.And, 983
cv.AndS, 983
cv.ApproxChains, 1114
cv.ApproxPoly, 1114
cv.ArcLength, 1115
cv.Avg, 984

cv.AvgSdv, 984
cv.BoundingRect, 1116
cv.BoxPoints, 1116
cv.CalcBackProject, 1071
cv.CalcBackProjectPatch, 1072
cv.CalcCovarMatrix, 985
cv.CalcEMD2, 1117
cv.CalcGlobalOrientation, 1161
cv.CalcHist, 1073
cv.CalcMotionGradient, 1162
cv.CalcOpticalFlowBM, 1163
cv.CalcOpticalFlowHS, 1163
cv.CalcOpticalFlowLK, 1164
cv.CalcOpticalFlowPyrLK, 1165
cv.CalcPGH, 1117
cv.CalcProbDensity, 1075
cv.CalcSubdivVoronoi2D, 1133
cv.CalibrateCamera2, 1191
cv.CamShift, 1166
cv.Canny, 1141
cv.CaptureFromCAM, 1182
cv.CaptureFromFile, 1183
cv.CartToPolar, 986
cv.Cbrt, 987
cv.Ceil, 1036
cv.CheckContourConvexity, 1118
cv.Circle, 1056
cv.ClearHist, 1075
cv.ClearND, 987
cv.ClearSubdivVoronoi2D, 1134
cv.ClipLine, 1057
cv.CloneImage, 988
cv.CloneMat, 988
cv.CloneMatND, 988
cv.CloneSeq, 1054
cv.Cmp, 988
cv.CmpS, 989
cv.CompareHist, 1076
cv.ComputeCorrespondEpilines, 1193
cv.ContourArea, 1119
cv.ContourFromContourTree, 1120
cv.Convert, 990
cv.ConvertPointsHomogeneous, 1194
cv.ConvertScale, 991
cv.ConvertScaleAbs, 991
cv.ConvexHull2, 1121
cv.ConvexityDefects, 1121
cv.Copy, 992
cv.CopyMakeBorder, 1081
cv.CornerEigenValsAndVecs, 1142
cv.CornerHarris, 1142
cv.CornerMinEigenVal, 1143
cv.CountNonZero, 993
cv.CreateContourTree, 1122
cv.CreateData, 993
cv.CreateHist, 1077
cv.CreateImage, 994
cv.CreateImageHeader, 994
cv.CreateKalman, 1168
cv.CreateMat, 995
cv.CreateMatHeader, 995
cv.CreateMatND, 995
cv.CreateMatNDHeader, 996
cv.CreateMemStorage, 1054
cv.CreatePOSITObject, 1194
cv.CreateStereoBMState, 1195
cv.CreateStereoGCState, 1195
cv.CreateStructuringElementEx, 1082
cv.CreateSubdivDelaunay2D, 1134
cv.CreateTrackbar, 1173
cv.CreateVideoWriter, 1187
cv.CrossProduct, 996
cv.CvtColor, 1100
cv.CvtScaleAbs, 992
cv.DCT, 997
cv.DecomposeProjectionMatrix, 1198
cv.DestroyAllWindows, 1174
cv.DestroyWindow, 1174
cv.Det, 1000
cv.DFT, 998

cv.Dilate, 1083
cv.DistTransform, 1104
cv.Div, 1000
cv.DotProduct, 1001
cv.DrawChessboardCorners, 1199
cv.DrawContours, 1057
cv.EigenVV, 1001
cv.Ellipse, 1058
cv.EllipseBox, 1059
cv.Erode, 1084
cv.Exp, 1002
cv.ExtractSURF, 1152
cv.FastArctan, 1002
cv.FillConvexPoly, 1060
cv.FillPoly, 1060
cv.Filter2D, 1084
cv.FindChessboardCorners, 1199
cv.FindContours, 1122
cv.FindCornerSubPix, 1144
cv.FindExtrinsicCameraParams2, 1201
cv.FindFundamentalMat, 1201
cv.FindHomography, 1203
cv.FindNearestPoint2D, 1134
cv.FindStereoCorrespondenceBM, 1204
cv.FindStereoCorrespondenceGC, 1205
cv.FitEllipse2, 1124
cv.FitLine, 1124
cv.Flip, 1003
cv.FloodFill, 1106
cv.Floor, 1035
cv.fromarray, 1004
cv.GEMM, 1005
cv.Get1D, 1005
cv.Get2D, 1006
cv.Get3D, 1006
cv.GetAffineTransform, 1092
cv.GetCaptureProperty, 1183
cv.GetCentralMoment, 1125
cv.GetCol, 1007
cv.GetCols, 1007
cv.GetDiag, 1008
cv.GetDims, 1008
cv.GetElemType, 1009
cv.GetHuMoments, 1126
cv.GetImage, 1009
cv.GetImageCOI, 1009
cv.GetImageROI, 1010
cv.GetMat, 1010
cv.GetMinMaxHistValue, 1078
cv.GetND, 1007
cv.GetNormalizedCentralMoment, 1127
cv.GetOptimalDFTSize, 1011
cv.GetOptimalNewCameraMatrix, 1207
cv.GetPerspectiveTransform, 1092
cv.GetQuadrangleSubPix, 1093
cv.GetReal1D, 1011
cv.GetReal2D, 1012
cv.GetReal3D, 1012
cv.GetRealND, 1013
cv.GetRectSubPix, 1094
cv.GetRotationMatrix2D, 1091
cv.GetRow, 1013
cv.GetRows, 1013
cv.GetSize, 1014
cv.GetSpatialMoment, 1127
cv.GetStarKeypoints, 1153
cv.GetSubRect, 1014
cv.GetTextSize, 1061
cv.GetTickCount, 1068
cv.GetTickFrequency, 1069
cv.GetTrackbarPos, 1175
cv.GoodFeaturesToTrack, 1145
cv.GrabFrame, 1184
cv.HaarDetectObjects, 1158
cv.HoughLines2, 1146
cv.InitFont, 1061
cv.InitIntrinsicParams2D, 1208
cv.InitLineIterator, 1062
cv.InitUndistortMap, 1209
cv.InitUndistortRectifyMap, 1209

cv.Inpaint, 1108
cv.InRange, 1015
cv.InRangeS, 1015
cv.Integral, 1109
cv.Inv, 1016
cv.Invert, 1017
cv.InvSqrt, 1016
cv.IsInf, 1017
cv.IsNaN, 1018
cv.KalmanCorrect, 1168
cv.KalmanPredict, 1169
cv.KMeans2, 1067
cv.Laplace, 1085
cv.Line, 1063
cv.Load, 1066
cv.LoadImage, 1180
cv.LoadImageM, 1181
cv.Log, 1019
cv.LogPolar, 1094
cv.LUT, 1018
cv.Mahalanobis, 1019
cv.MatchContourTrees, 1128
cv.MatchShapes, 1128
cv.MatchTemplate, 1148
cv.Max, 1020
cv.MaxS, 1020
cv.MeanShift, 1170
cv.Merge, 1021
cv.mGet, 1052
cv.Min, 1021
cv.MinAreaRect2, 1129
cv.MinEnclosingCircle, 1130
cv.MinMaxLoc, 1022
cv.MinS, 1022
cv.MixChannels, 1023
cv.Moments, 1131
cv.MorphologyEx, 1086
cv.MoveWindow, 1175
cv.mSet, 1052
cv.Mul, 1024
cv.MulSpectrums, 1024
cv.MultiplyAcc, 1140
cv.MulTransposed, 1025
cv.NamedWindow, 1176
cv.Norm, 1026
cv.NormalizeHist, 1079
cv.Not, 1027
cv.Or, 1027
cv.OrS, 1028
cv.PerspectiveTransform, 1028
cv.PointPolygonTest, 1131
cv.PolarToCart, 1029
cv.PolyLine, 1064
cv.POSIT, 1211
cv.Pow, 1030
cv.PreCornerDetect, 1147
cv.ProjectPoints2, 1212
cv.PutText, 1065
cv.PyrDown, 1087
cv.PyrMeanShiftFiltering, 1110
cv.PyrSegmentation, 1111
cv.QueryFrame, 1185
cv.RandArr, 1031
cv.RandInt, 1031
cv.RandReal, 1032
cv.Rectangle, 1065
cv.Reduce, 1032
cv.Remap, 1095
cv.Repeat, 1033
cv.ReprojectImageTo3D, 1213
cv.ResetImageROI, 1033
cv.Reshape, 1034
cv.ReshapeMatND, 1034
cv.Resize, 1096
cv.ResizeWindow, 1177
cv.RetrieveFrame, 1185
cv.RNG, 1030
cv.Rodrigues2, 1214
cv.Round, 1035
cv.RQDecomp3x3, 1214

cv.RunningAvg, 1140
cv.Save, 1067
cv.SaveImage, 1182
cv.ScaleAdd, 1036
cv.SegmentMotion, 1170
cv.SeqInvert, 1055
cv.SeqRemove, 1055
cv.SeqRemoveSlice, 1055
cv.Set, 1036
cv.Set1D, 1037
cv.Set2D, 1037
cv.Set3D, 1038
cv.SetCaptureProperty, 1185
cv.SetData, 1039
cv.SetIdentity, 1039
cv.SetImageCOI, 1040
cv.SetImageROI, 1040
cv.SetMouseCallback, 1177
cv.SetND, 1038
cv.SetReal1D, 1041
cv.SetReal2D, 1041
cv.SetReal3D, 1041
cv.SetRealND, 1042
cv.SetTrackbarPos, 1178
cv.SetZero, 1042
cv.ShowImage, 1179
cv.Smooth, 1087
cv.SnakeImage, 1171
cv.Sobel, 1089
cv.Solve, 1043
cv.SolveCubic, 1043
cv.Split, 1044
cv.Sqrt, 1045
cv.SquareAcc, 1141
cv.StereoCalibrate, 1215
cv.StereoRectify, 1218
cv.StereoRectifyUncalibrated, 1221
cv.Sub, 1045
cv.Subdiv2DEdgeDst, 1135
cv.Subdiv2DGetEdge, 1135
cv.Subdiv2DLocate, 1137
cv.Subdiv2DNextEdge, 1136
cv.Subdiv2DRotateEdge, 1138
cv.SubdivDelaunay2DInsert, 1139
cv.SubRS, 1045
cv.SubS, 1046
cv.Sum, 1047
cv.SVBkSb, 1047
cv.SVD, 1048
cv.ThreshHist, 1080
cv.Threshold, 1112
cv.Trace, 1049
cv.Transform, 1050
cv.Transpose, 1050
cv.Undistort2, 1222
cv.UndistortPoints, 1223
cv.UpdateMotionHistory, 1172
cv.WaitKey, 1179
cv.WarpAffine, 1096
cv.WarpPerspective, 1097
cv.WriteFrame, 1187
cv.Xor, 1051
cv.XorS, 1051
cv::abs, 538
cv::absdiff, 538
cv::accumulate, 711
cv::accumulateProduct, 713
cv::accumulateSquare, 712
cv::accumulateWeighted, 713
cv::adaptiveThreshold, 680
cv::add, 539
cv::addText, 828
cv::addWeighted, 540
cv::AdjusterAdapter::good, 754
cv::AdjusterAdapter::tooFew, 753
cv::AdjusterAdapter::tooMany, 754
cv::alignPtr, 626
cv::alignSize, 627
cv::allocate, 627
cv::approxPolyDP, 701

cv::arcLength, 702
cv::bilateralFilter, 647
cv::blur, 648
cv::borderInterpolate, 648
cv::boundingRect, 703
cv::BOWImgDescriptorExtractor::BOWImgDescriptorExtractor, 785
cv::BOWImgDescriptorExtractor::compute, 786
cv::BOWImgDescriptorExtractor::descriptorSize, 786
cv::BOWImgDescriptorExtractor::descriptorType, 786
cv::BOWImgDescriptorExtractor::getVocabulary, 785
cv::BOWImgDescriptorExtractor::setVocabulary, 785
cv::BOWTrainer::add, 782
cv::BOWTrainer::cluster, 783
cv::BOWTrainer::descripotorsCount, 783
cv::BOWTrainer::getDescriptors, 783
cv::boxFilter, 649
cv::buildPyramid, 650
cv::calcBackProject, 636
cv::calcCovarMatrix, 545
cv::calcGlobalOrientation, 807
cv::calcHist, 633
cv::calcMotionGradient, 806
cv::calcOpticalFlowFarneback, 804
cv::calcOpticalFlowPyrLK, 803
cv::calibrateCamera, 835
cv::calibrationMatrixValues, 837
cv::CamShift, 807
cv::Canny, 714
cv::cartToPolar, 546
cv::CascadeClassifier::CascadeClassifier, 799
cv::CascadeClassifier::detectMultiScale, 800
cv::CascadeClassifier::empty, 800
cv::CascadeClassifier::load, 800
cv::CascadeClassifier::read, 800
cv::CascadeClassifier::runAt, 801
cv::CascadeClassifier::setImage, 801
cv::checkRange, 547
cv::circle, 612
cv::clipLine, 612
cv::compare, 548
cv::compareHist, 637
cv::completeSymm, 549
cv::composeRT, 838
cv::computeCorrespondEpilines, 839
cv::contourArea, 704
cv::convertMaps, 672
cv::convertPointsHomogeneous, 840
cv::convertScaleAbs, 549
cv::convexHull, 705
cv::copyMakeBorder, 650
cv::cornerEigenValsAndVecs, 715
cv::cornerHarris, 716
cv::cornerMinEigenVal, 716
cv::cornerSubPix, 717
cv::countNonZero, 550
cv::createBoxFilter, 652
cv::createButton, 831
cv::createDerivFilter, 653
cv::createGaussianFilter, 653
cv::createLinearFilter, 654
cv::createMorphologyFilter, 655
cv::createOpenGLCallback, 829
cv::createSeparableLinearFilter, 656
cv::createTrackbar, 811
cv::cubeRoot, 551
cv::cv::flann::hierarchicalClustering¡ET,DT¿, 793
cv::cvarrToMat, 551
cv::cvtColor, 681
cv::dct, 553
cv::deallocate, 627
cv::decomposeProjectionMatrix, 840
cv::DescriptorExtractor::compute, 756
cv::DescriptorExtractor::create, 758
cv::DescriptorExtractor::read, 757
cv::DescriptorExtractor::write, 757

cv::DescriptorMatcher::add, 762
cv::DescriptorMatcher::clear, 763
cv::DescriptorMatcher::clone, 766
cv::DescriptorMatcher::create, 767
cv::DescriptorMatcher::empty, 763
cv::DescriptorMatcher::getTrainDescriptors, 763
cv::DescriptorMatcher::isMaskSupported, 763
cv::DescriptorMatcher::knnMatch, 765
cv::DescriptorMatcher::match, 764
cv::DescriptorMatcher::radiusMatch, 766
cv::DescriptorMatcher::train, 764
cv::determinant, 559
cv::dft, 554
cv::dilate, 657
cv::displayOverlay, 828
cv::displayStatusBar, 829
cv::distanceTransform, 687
cv::divide, 558
cv::drawChessboardCorners, 841
cv::drawContours, 700
cv::drawKeypoints, 781
cv::drawMatches, 779
cv::DynamicAdaptedFeatureDetector::DynamicAdaptedFeatureDetector, 752
cv::eigen, 560
cv::ellipse, 613
cv::ellipse2Poly, 614
cv::equalizeHist, 638
cv::erode, 658
cv::error, 628
cv::estimateAffine3D, 704
cv::estimateRigidTransform, 703
cv::exp, 560
cv::extractImageCOI, 561
cv::FAST, 729
cv::fastAtan2, 562
cv::fastFree, 630
cv::fastMalloc, 629
cv::FeatureDetector::create, 747
cv::FeatureDetector::detect, 746
cv::FeatureDetector::read, 747
cv::FeatureDetector::write, 747
cv::FeatureEvaluator::calcCat, 797
cv::FeatureEvaluator::calcOrd, 797
cv::FeatureEvaluator::clone, 796
cv::FeatureEvaluator::create, 798
cv::FeatureEvaluator::getFeatureType, 796
cv::FeatureEvaluator::read, 795
cv::FeatureEvaluator::setImage, 796
cv::FeatureEvaluator::setWindow, 797
cv::fillConvexPoly, 615
cv::fillPoly, 616
cv::filter2D, 659
cv::findChessboardCorners, 842
cv::findCirclesGrid, 843
cv::findContours, 698
cv::findFundamentalMat, 845
cv::findHomography, 847
cv::fitEllipse, 706
cv::fitLine, 706
cv::flip, 562
cv::floodFill, 688
cv::format, 630
cv::GaussianBlur, 660
cv::gemm, 563
cv::GenericDescriptorMatcher::add, 771
cv::GenericDescriptorMatcher::classify, 772
cv::GenericDescriptorMatcher::clear, 772
cv::GenericDescriptorMatcher::clone, 776
cv::GenericDescriptorMatcher::getTrainImages, 772
cv::GenericDescriptorMatcher::getTrainKeypoints, 772
cv::GenericDescriptorMatcher::isMaskSupported, 772
cv::GenericDescriptorMatcher::knnMatch, 774
cv::GenericDescriptorMatcher::match, 773
cv::GenericDescriptorMatcher::radiusMatch, 775
cv::GenericDescriptorMatcher::read, 775
cv::GenericDescriptorMatcher::train, 772

cv::GenericDescriptorMatcher::write, 776
cv::getAffineTransform, 673
cv::getConvertElem, 564
cv::getDefaultNewCameraMatrix, 848
cv::getDerivKernels, 661
cv::getGaussianKernel, 662
cv::getKernelType, 662
cv::getNumThreads, 631
cv::getOptimalDFTSize, 565
cv::getOptimalNewCameraMatrix, 849
cv::getPerspectiveTransform, 673
cv::getRectSubPix, 674
cv::getRotationMatrix2D, 675
cv::getStructuringElement, 663
cv::getTextSize, 616
cv::getThreadNum, 631
cv::getTickCount, 631
cv::getTickFrequency, 632
cv::getTrackbarPos, 812
cv::getWindowProperty, 826
cv::goodFeaturesToTrack, 718
cv::gpu::blur, 961
cv::gpu::boxFilter, 961
cv::gpu::columnSum, 932
cv::gpu::convolve, 935
cv::gpu::ConvolveBuf::ConvolveBuf, 936
cv::gpu::cornerHarris, 932
cv::gpu::cornerMinEigenVal, 933
cv::gpu::countNonZero, 940
cv::gpu::createContinuous, 926
cv::gpu::dft, 934
cv::gpu::dilate, 963
cv::gpu::erode, 962
cv::gpu::filter2D, 964
cv::gpu::GaussianBlur, 968
cv::gpu::hasAtomicsSupport, 925
cv::gpu::hasNativeDoubleSupport, 925
cv::gpu::HOGDescriptor::detect, 943
cv::gpu::HOGDescriptor::detectMultiScale, 944
cv::gpu::HOGDescriptor::getBlockHistogramSize, 942
cv::gpu::HOGDescriptor::getDefaultPeopleDetector, 943
cv::gpu::HOGDescriptor::getDescriptors, 944
cv::gpu::HOGDescriptor::getDescriptorSize, 942
cv::gpu::HOGDescriptor::getPeopleDetector48x96, 943
cv::gpu::HOGDescriptor::getPeopleDetector64x128, 943
cv::gpu::HOGDescriptor::HOGDescriptor, 941
cv::gpu::HOGDescriptor::setSVMDetector, 942
cv::gpu::integral, 931
cv::gpu::Laplacian, 964
cv::gpu::matchTemplate, 937
cv::gpu::meanShiftFiltering, 929
cv::gpu::meanShiftProc, 930
cv::gpu::meanShiftSegmentation, 930
cv::gpu::merge, 926
cv::gpu::minMax, 938
cv::gpu::minMaxLoc, 939
cv::gpu::morphologyEx, 963
cv::gpu::mulAndScaleSpectrums, 934
cv::gpu::mulSpectrums, 933
cv::gpu::Scharr, 967
cv::gpu::sepFilter2D, 966
cv::gpu::Sobel, 967
cv::gpu::split, 927
cv::gpu::sqrIntegral, 931
cv::gpu::sqrSum, 938
cv::gpu::sum, 938
cv::grabCut, 695
cv::groupRectangles, 802
cv::HoughCircles, 720
cv::HoughLines, 722
cv::HoughLinesP, 723
cv::HuMoments, 697
cv::idct, 565
cv::idft, 566
cv::imdecode, 816

cv::imencode, 816
cv::imread, 817
cv::imshow, 813
cv::imwrite, 818
cv::initCameraMatrix2D, 850
cv::initUndistortRectifyMap, 851
cv::inpaint, 690
cv::inRange, 566
cv::integral, 691
cv::invert, 567
cv::invertAffineTransform, 675
cv::isContourConvex, 708
cv::kmeans, 624
cv::Laplacian, 665
cv::line, 617
cv::loadWindowParameters, 831
cv::log, 568
cv::LUT, 568
cv::magnitude, 569
cv::Mahalanobis, 570
cv::Mat::addref, 515
cv::Mat::adjustROI, 518
cv::Mat::assignTo, 509
cv::Mat::at, 525
cv::Mat::begin, 526
cv::Mat::channels, 524
cv::Mat::clone, 507
cv::Mat::col, 505
cv::Mat::colRange, 506
cv::Mat::convertTo, 508
cv::Mat::copyTo, 507
cv::Mat::create, 514
cv::Mat::cross, 512
cv::Mat::depth, 523
cv::Mat::destructor, 503
cv::Mat::diag, 506
cv::Mat::dot, 512
cv::Mat::elemSize, 522
cv::Mat::elemSize1, 522
cv::Mat::empty, 525
cv::Mat::end, 527
cv::Mat::eye, 514
cv::Mat::inv, 510
cv::Mat::isContinuous, 520
cv::Mat::locateROI, 517
cv::Mat::Mat, 501, 503
cv::Mat::mul, 511
cv::Mat::ones, 513
cv::Mat::operator =, 503
cv::Mat::operator CvMat, 519
cv::Mat::operator IplImage, 520
cv::Mat::operator MatExpr, 504
cv::Mat::operator MatExpr_, 504
cv::Mat::operator(), 519
cv::Mat::ptr, 525
cv::Mat::release, 516
cv::Mat::reshape, 509
cv::Mat::resize, 516
cv::Mat::row, 504
cv::Mat::rowRange, 506
cv::Mat::setTo, 509
cv::Mat::size, 524
cv::Mat::step1, 524
cv::Mat::t, 510
cv::Mat::total, 520
cv::Mat::type, 523
cv::Mat::zeros, 512
cv::matchShapes, 709
cv::matchTemplate, 726
cv::matMulDeriv, 852
cv::max, 570
cv::mean, 571
cv::meanShift, 808
cv::meanStdDev, 572
cv::medianBlur, 664
cv::merge, 573
cv::min, 573
cv::minAreaRect, 708
cv::minEnclosingCircle, 709
cv::minMaxLoc, 574

cv::mixChannels, 575
cv::moments, 696
cv::morphologyEx, 664
cv::mulSpectrums, 577
cv::multiply, 578
cv::mulTransposed, 578
cv::namedWindow, 813
cv::norm, 579
cv::normalize, 580
cv::partition, 626
cv::PCA::backProject, 585
cv::PCA::operator (), 584
cv::PCA::PCA, 583
cv::PCA::project, 585
cv::perspectiveTransform, 586
cv::phase, 587
cv::pointPolygonTest, 710
cv::polarToCart, 588
cv::polylines, 620
cv::pow, 588
cv::preCornerDetect, 725
cv::projectPoints, 853
cv::putText, 621
cv::pyrDown, 666
cv::pyrUp, 667
cv::randn, 594
cv::RandomizedTree::applyQuantization, 737
cv::RandomizedTree::read, 736
cv::RandomizedTree::train, 735
cv::RandomizedTree::write, 736
cv::randShuffle, 595
cv::randu, 594
cv::rectangle, 619
cv::reduce, 596
cv::remap, 676
cv::repeat, 596
cv::reprojectImageTo3D, 854
cv::resize, 677
cv::RNG::fill, 593
cv::RNG::gaussian, 593
cv::RNG::next, 591
cv::RNG::operator (), 591
cv::RNG::operator T, 591
cv::RNG::RNG, 590
cv::RNG::uniform, 592
cv::Rodrigues, 856
cv::RQDecomp3x3, 855
cv::RTreeClassifier::countNonZeroElements, 741
cv::RTreeClassifier::getSignature, 740
cv::RTreeClassifier::getSparseSignature, 740
cv::RTreeClassifier::read, 741
cv::RTreeClassifier::setQuantization, 742
cv::RTreeClassifier::train, 739
cv::RTreeClassifier::write, 741
cv::saveWindowParameters, 831
cv::scaleAdd, 598
cv::Scharr, 670
cv::sepFilter2D, 668
cv::setIdentity, 599
cv::setNumThreads, 632
cv::setTrackbarPos, 815
cv::setWindowProperty, 825
cv::Sobel, 669
cv::solve, 599
cv::solveCubic, 600
cv::solvePnP, 844
cv::solvePoly, 601
cv::sort, 601
cv::sortIdx, 602
cv::split, 603
cv::sqrt, 603
cv::stereoCalibrate, 860
cv::stereoRectify, 863
cv::stereoRectifyUncalibrated, 866
cv::StereoSGBM::operator (), 860
cv::StereoSGBM::StereoSGBM, 858
cv::subtract, 604
cv::sum, 608
cv::SVD::backSubst, 608
cv::SVD::operator (), 607

cv::SVD::solveZ, 607
cv::SVD::SVD, 606
cv::theRNG, 609
cv::threshold, 692
cv::trace, 609
cv::transform, 610
cv::transpose, 611
cv::undistort, 867
cv::undistortPoints, 868
cv::updateMotionHistory, 805
cv::VideoCapture::get, 821
cv::VideoCapture::set, 822
cv::VideoCapture::VideoCapture, 820
cv::waitKey, 815
cv::warpAffine, 678
cv::warpPerspective, 679
cv::watershed, 694
cvAbsDiff, 60
cvAbsDiffS, 60
cvAcc, 340
cvAdaptiveThreshold, 287
cvAdd, 61
cvAddS, 61
cvAddText, 418
cvAddWeighted, 62
cvAlloc, 247
cvAnd, 63
cvAndS, 63
cvApproxChains, 303
cvApproxPoly, 304
cvArcLength, 305
cvAvg, 64
cvAvgSdv, 65
cvBoundingRect, 306
cvBoxPoints, 306
cvCalcBackProject, 253
cvCalcBackProjectPatch, 254
cvCalcCovarMatrix, 65
cvCalcEMD2, 308
cvCalcGlobalOrientation, 375
cvCalcHist, 256
cvCalcImageHomography, 425
cvCalcMotionGradient, 376
cvCalcOpticalFlowBM, 377
cvCalcOpticalFlowHS, 378
cvCalcOpticalFlowLK, 379
cvCalcOpticalFlowPyrLK, 379
cvCalcPGH, 307
cvCalcProbDensity, 257
cvCalcSubdivVoronoi2D, 333
cvCalibrateCamera2, 425
cvCamShift, 381
cvCanny, 342
cvCaptureFromCAM, 407
cvCaptureFromFile, 407
cvCartToPolar, 67
cvCbrt, 68
cvCheckContourConvexity, 309
cvCircle, 194
cvClearGraph, 155
cvClearHist, 258
cvClearMemStorage, 155
cvClearND, 68
cvClearSeq, 156
cvClearSet, 156
cvClearSubdivVoronoi2D, 333
cvClipLine, 195
cvClone, 212
cvCloneGraph, 156
cvCloneImage, 68
cvCloneMat, 69
cvCloneMatND, 69
cvCloneSeq, 157
cvCloneSparseMat, 69
cvCmp, 69
cvCmpS, 70
cvCompareHist, 259
cvComputeCorrespondEpilines, 428
cvConDensInitSampleSet, 383
cvContourArea, 310

cvContourFromContourTree, 311
cvConvertImage, 395
cvConvertPointsHomogeneous, 429
cvConvertScale, 71
cvConvertScaleAbs, 72
cvConvexHull2, 312
cvConvexityDefects, 314
cvCopy, 73
cvCopyHist, 260
cvCopyMakeBorder, 266
cvCornerEigenValsAndVecs, 343
cvCornerHarris, 344
cvCornerMinEigenVal, 345
cvCountNonZero, 74
cvCreateButton, 421
cvCreateChildMemStorage, 157
cvCreateConDensation, 382
cvCreateContourTree, 315
cvCreateData, 74
cvCreateGraph, 159
cvCreateGraphScanner, 159
cvCreateHist, 260
cvCreateImage, 75
cvCreateImageHeader, 75
cvCreateKalman, 385
cvCreateMat, 76
cvCreateMatHeader, 76
cvCreateMatND, 77
cvCreateMatNDHeader, 77
cvCreateMemStorage, 160
cvCreateOpenGLCallback, 419
cvCreatePOSITObject, 430
cvCreateSeq, 161
cvCreateSet, 162
cvCreateSparseMat, 78
cvCreateStereoBMState, 430
cvCreateStereoGCState, 431
cvCreateStructuringElementEx, 267
cvCreateSubdivDelaunay2D, 333
cvCreateTrackbar, 396
cvCreateVideoWriter, 412
cvCrossProduct, 78
cvCvtColor, 288
cvCvtScaleAbs, 73
cvCvtSeqToArray, 162
cvDCT, 79
cvDecomposeProjectionMatrix, 434
cvDecRefData, 83
cvDestroyAllWindows, 397
cvDestroyWindow, 397
cvDet, 83
cvDFT, 80
cvDilate, 268
cvDisplayOverlay, 418
cvDisplayStatusBar, 419
cvDistTransform, 293
cvDiv, 83
cvDotProduct, 84
cvDrawChessboardCorners, 435
cvDrawContours, 195
cvEigenVV, 85
cvEllipse, 197
cvEllipseBox, 198
cvEndFindContours, 316
cvEndWriteSeq, 163
cvEndWriteStruct, 213
cvErode, 269
cvError, 244
cvErrorStr, 245
cvExp, 86
cvExtractSURF, 357
cvFastArctan, 86
cvFillConvexPoly, 199
cvFillPoly, 200
cvFilter2D, 270
cvFindChessboardCorners, 436
cvFindContours, 316
cvFindCornerSubPix, 345
cvFindExtrinsicCameraParams2, 437
cvFindFundamentalMat, 438

cvFindGraphEdge, 163
cvFindGraphEdgeByPtr, 164
cvFindHomography, 440
cvFindNearestPoint2D, 334
cvFindNextContour, 318
cvFindStereoCorrespondenceBM, 442
cvFindStereoCorrespondenceGC, 443
cvFindType, 213
cvFirstType, 213
cvFitEllipse2, 318
cvFitLine, 318
cvFlip, 86
cvFloodFill, 295
cvFlushSeqWriter, 164
cvFontQt, 417
cvFree, 247
cvGEMM, 87
cvGet?D, 88
cvGetAffineTransform, 278
cvGetCaptureProperty, 408
cvGetCentralMoment, 320
cvGetCol(s), 89
cvGetDiag, 90
cvGetElemType, 91
cvGetErrMode, 243
cvGetErrStatus, 243
cvGetFileNode, 214
cvGetFileNodeByName, 214
cvGetFileNodeName, 215
cvGetGraphVtx, 165
cvGetHashedKey, 215
cvGetHistValue*D, 261
cvGetHuMoments, 320
cvGetImage, 91
cvGetImageCOI, 92
cvGetImageROI, 92
cvGetMat, 93
cvGetMinMaxHistValue, 262
cvGetModuleInfo, 249
cvGetNextSparseNode, 93
cvGetNormalizedCentralMoment, 321
cvGetOptimalDFTSize, 94
cvGetOptimalNewCameraMatrix, 445
cvGetPerspectiveTransform, 279
cvGetQuadrangleSubPix, 280
cvGetRawData, 95
cvGetReal1D, 95
cvGetReal2D, 96
cvGetReal3D, 96
cvGetRealND, 97
cvGetRectSubPix, 280
cvGetRootFileNode, 217
cvGetRotationMatrix2D, 278
cvGetRow(s), 98
cvGetSeqElem, 165
cvGetSeqReaderPos, 166
cvGetSetElem, 166
cvGetSize, 98
cvGetSpatialMoment, 322
cvGetStarKeypoints, 359
cvGetSubRect, 99
cvGetTextSize, 201
cvGetTickCount, 248
cvGetTickFrequency, 248
cvGetTrackbarPos, 398
cvGetWindowHandle, 398
cvGetWindowName, 399
cvGetWindowProperty, 416
cvGoodFeaturesToTrack, 347
cvGrabFrame, 409
cvGraphAddEdge, 166
cvGraphAddEdgeByPtr, 167
cvGraphAddVtx, 168
cvGraphEdgeIdx, 168
cvGraphRemoveEdge, 169
cvGraphRemoveEdgeByPtr, 169
cvGraphRemoveVtx, 170
cvGraphRemoveVtxByPtr, 170
cvGraphVtxDegree, 171
cvGraphVtxDegreeByPtr, 171

cvGraphVtxIdx, 172
cvHaarDetectObjects, 369
cvHoughLines2, 348
cvIncRefData, 100
cvInitFont, 201
cvInitImageHeader, 101
cvInitIntrinsicParams2D, 446
cvInitLineIterator, 202
cvInitMatHeader, 102
cvInitMatNDHeader, 103
cvInitSparseMatIterator, 103
cvInitSystem, 399
cvInitTreeNodeIterator, 172
cvInitUndistortMap, 447
cvInitUndistortRectifyMap, 448
cvInpaint, 297
cvInRange, 99
cvInRangeS, 100
cvInsertNodeIntoTree, 173
cvIntegral, 298
cvInv, 104
cvInvert, 104
cvInvSqrt, 104
cvIsInf, 105
cvIsNaN, 105
cvKalmanCorrect, 385
cvKalmanPredict, 388
cvKMeans2, 234
cvLaplace, 271
cvLine, 204
cvLoad, 218
cvLoadHaarClassifierCascade, 369
cvLoadImage, 404
cvLoadImageM, 405
cvLoadWindowParameters, 421
cvLog, 106
cvLogPolar, 281
cvLUT, 105
cvMahalanobis, 107
cvMakeHistHeaderForArray, 263
cvMakeSeqHeaderForArray, 173
cvMat, 107
cvMatchContourTrees, 322
cvMatchShapes, 323
cvMatchTemplate, 354
cvMax, 108
cvMaxS, 108
cvMeanShift, 389
cvMemStorageAlloc, 174
cvMemStorageAllocString, 174
cvMerge, 109
cvmGet, 145
cvMin, 110
cvMinAreaRect2, 324
cvMinEnclosingCircle, 325
cvMinMaxLoc, 110
cvMinS, 111
cvMixChannels, 111
cvMoments, 326
cvMorphologyEx, 271
cvMoveWindow, 400
cvmSet, 145
cvMul, 112
cvMulSpectrums, 113
cvMultiplyAcc, 340
cvMulTransposed, 114
cvNamedWindow, 400
cvNextGraphItem, 175
cvNextTreeNode, 176
cvNorm, 114
cvNormalizeHist, 264
cvNot, 115
cvOpenFileStorage, 218
cvOr, 116
cvOrS, 116
cvPerspectiveTransform, 117
cvPointPolygonTest, 326
cvPointSeqFromMat, 327
cvPolarToCart, 117
cvPolyLine, 205

cvPOSIT, 449
cvPow, 118
cvPreCornerDetect, 352
cvPrevTreeNode, 176
cvProjectPoints2, 450
cvPtr?D, 119
cvPutText, 206
cvPyrDown, 273
cvPyrMeanShiftFiltering, 299
cvPyrSegmentation, 300
cvQueryFrame, 409
cvQueryHistValue*D, 264
cvRandArr, 120
cvRandInt, 122
cvRandReal, 123
cvRead, 219
cvReadByName, 220
cvReadChainPoint, 328
cvReadInt, 220
cvReadIntByName, 221
cvReadRawData, 221
cvReadRawDataSlice, 222
cvReadReal, 222
cvReadRealByName, 223
cvReadString, 223
cvReadStringByName, 224
cvRectangle, 206
cvRedirectError, 245
cvReduce, 124
cvRegisterModule, 248
cvRegisterType, 224
cvRelease, 225
cvReleaseCapture, 410
cvReleaseConDensation, 390
cvReleaseData, 124
cvReleaseFileStorage, 225
cvReleaseGraphScanner, 176
cvReleaseHaarClassifierCascade, 373
cvReleaseHist, 265
cvReleaseImage, 125
cvReleaseImageHeader, 125
cvReleaseKalman, 390
cvReleaseMat, 126
cvReleaseMatND, 126
cvReleaseMemStorage, 177
cvReleasePOSITObject, 453
cvReleaseSparseMat, 127
cvReleaseStereoBMState, 453
cvReleaseStereoGCState, 454
cvReleaseStructuringElement, 273
cvReleaseVideoWriter, 412
cvRemap, 283
cvRepeat, 127
cvReprojectImageTo3D, 451
cvResetImageROI, 127
cvReshape, 128
cvReshapeMatND, 129
cvResize, 284
cvResizeWindow, 401
cvRestoreMemStoragePos, 177
cvRetrieveFrame, 410
cvRNG, 120
cvRodrigues2, 454
cvRQDecomp3x3, 452
cvRunHaarClassifierCascade, 373
cvRunningAvg, 341
cvSampleLine, 353
cvSave, 225
cvSaveImage, 406
cvSaveMemStoragePos, 177
cvSaveWindowParameters, 421
cvScaleAdd, 130
cvSegmentMotion, 391
cvSeqElemIdx, 178
cvSeqInsert, 178
cvSeqInsertSlice, 179
cvSeqInvert, 180
cvSeqPartition, 237
cvSeqPop, 180
cvSeqPopFront, 180

cvSeqPopMulti, 181
cvSeqPush, 182
cvSeqPushFront, 182
cvSeqPushMulti, 183
cvSeqRemove, 184
cvSeqRemoveSlice, 184
cvSeqSearch, 184
cvSeqSlice, 185
cvSeqSort, 186
cvSet, 131
cvSet?D, 131
cvSetAdd, 187
cvSetCaptureProperty, 410
cvSetData, 132
cvSetErrMode, 244
cvSetErrStatus, 243
cvSetHistBinRanges, 265
cvSetIdentity, 132
cvSetImageCOI, 133
cvSetImageROI, 133
cvSetImagesForHaarClassifierCascade, 372
cvSetIPLAllocators, 251
cvSetMemoryManager, 250
cvSetMouseCallback, 401
cvSetNew, 188
cvSetReal?D, 134
cvSetRemove, 188
cvSetRemoveByPtr, 189
cvSetSeqBlockSize, 189
cvSetSeqReaderPos, 190
cvSetTrackbarPos, 403
cvSetWindowProperty, 415
cvSetZero, 134
cvShowImage, 403
cvSmooth, 274
cvSnakeImage, 392
cvSobel, 275
cvSolve, 135
cvSolveCubic, 136
cvSplit, 136
cvSqrt, 137
cvSquareAcc, 342
cvStartAppendToSeq, 190
cvStartFindContours, 329
cvStartNextStream, 226
cvStartReadChainPoints, 329
cvStartReadRawData, 226
cvStartReadSeq, 191
cvStartWriteSeq, 192
cvStartWriteStruct, 227
cvStereoCalibrate, 455
cvStereoRectify, 458
cvStereoRectifyUncalibrated, 461
cvSub, 137
cvSubdiv2DEdgeDst, 334
cvSubdiv2DGetEdge, 335
cvSubdiv2DLocate, 337
cvSubdiv2DNextEdge, 336
cvSubdiv2DRotateEdge, 338
cvSubdivDelaunay2DInsert, 339
cvSubRS, 138
cvSubS, 138
cvSubstituteContour, 330
cvSum, 139
cvSVBkSb, 139
cvSVD, 140
cvThreshHist, 265
cvThreshold, 301
cvTrace, 142
cvTransform, 142
cvTranspose, 143
cvTreeToNodeSeq, 193
cvTypeOf, 228
cvUndistort2, 462
cvUndistortPoints, 463
cvUnregisterType, 228
cvUpdateMotionHistory, 393
cvUseOptimized, 250
cvWaitKey, 404
cvWarpAffine, 285

cvWarpPerspective, 286
cvWrite, 228
cvWriteComment, 230
cvWriteFileNode, 230
cvWriteFrame, 413
cvWriteInt, 231
cvWriteRawData, 231
cvWriteReal, 233
cvWriteString, 233
cvXor, 143
cvXorS, 144