



Open Source Computer Vision Library

Reference Manual

Copyright © 1999-2001 Intel Corporation

All Rights Reserved

Issued in U.S.A.

Order Number: TBD

World Wide Web: <http://developer.intel.com>

Version	Version History	Date
-001	Original Issue	December 8, 2000

This Open Source Computer Vision Library Reference Manual as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Open Source Computer Vision Library may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copyright © Intel Corporation 1999-2001.

*Third-party brands and names are the property of their respective owners.

Contents

Contents

Chapter 1 Image Functions

Overview	1-1
Reference	1-4
cvCreateImageHeader	1-4
cvCreateImage	1-5
cvReleaseImageHeader	1-5
cvReleaseImage	1-6
cvCreateImageData	1-6
cvReleaseImageData	1-7
cvSetImageData	1-7
cvSetImageCOI	1-8
cvSetImageROI	1-8
cvGetImageRawData	1-9
cvInitImageHeader	1-9
cvCopyImage	1-10
Pixel Access Macros	1-10
Overview	1-10
CV_INIT_PIXEL_POS	1-12
CV_MOVE_TO	1-13
CV_MOVE	1-13
CV_MOVE_WRAP	1-14
CV_MOVE_PARAM	1-14

CV_MOVE_PARAM_WRAP	1-15
--------------------------	------

Chapter 2

Dynamic Data Structures

Memory Storage	2-1
Overview	2-1
cvCreateMemStorage	2-3
cvCreateChildMemStorage	2-3
cvReleaseMemStorage	2-4
cvClearMemStorage	2-4
cvSaveMemStoragePos	2-5
cvRestoreMemStoragePos	2-5
Sequences	2-7
Overview	2-7
cvCreateSeq	2-10
cvSetSeqBlockSize	2-11
cvSeqPush	2-12
cvSeqPop	2-13
cvSeqPushFront	2-13
cvSeqPopFront	2-13
cvSeqPushMulti	2-14
cvSeqPopMulti	2-14
cvSeqInsert	2-15
cvSeqRemove	2-16
cvClearSeq	2-16
cvGetSeqElem	2-17
cvSeqElemIdx	2-17
cvCvtSeqToArray	2-18
cvMakeSeqHeaderForArray	2-18
Writing and Reading Sequences	2-19
Overview	2-19
Reference	2-21
cvStartAppendToSeq	2-21

cvStartWriteSeq.....	2-21
cvEndWriteSeq.....	2-22
cvFlushSeqWriter.....	2-23
cvStartReadSeq.....	2-23
cvGetSeqReaderPos.....	2-24
cvSetSeqReaderPos.....	2-25
.....	2-25
Sets.....	2-25
Overview.....	2-25
Reference.....	2-29
cvCreateSet.....	2-29
cvSetAdd.....	2-29
cvSetRemove.....	2-30
cvGetSetElem.....	2-30
cvClearSet.....	2-31
Graphs.....	2-32
Overview.....	2-32
Reference.....	2-35
cvCreateGraph.....	2-35
cvGraphAddVtx.....	2-36
cvGraphRemoveVtx.....	2-36
cvGraphRemoveVtxByPtr.....	2-37
cvGraphAddEdge.....	2-37
cvGraphAddEdgeByPtr.....	2-38
cvGraphRemoveEdge.....	2-39
cvGraphRemoveEdgeByPtr.....	2-39
cvFindGraphEdge.....	2-40
cvFindGraphEdgeByPtr.....	2-40
cvGraphVtxDegree.....	2-41
cvGraphVtxDegreeByPtr.....	2-42
cvClearGraph.....	2-42
cvGetGraphVtx.....	2-43

cvGraphVtxIdx	2-43
cvGraphEdgeIdx	2-44

Chapter 3 Contour Processing

Overview	3-1
Basic Definitions	3-1
Contour Representation	3-4
Contour Retrieving Algorithm	3-5
Polygonal Approximation	3-6
Douglas-Peucker Approximation	3-9
Contours Moments	3-10
Hierarchical Representation of Contours	3-13
Data Structures	3-19
Reference	3-20
cvFindContours	3-20
cvStartFindContours	3-21
cvFindNextContour	3-22
cvSubstituteContour	3-23
cvEndFindContours	3-23
cvApproxChains	3-24
cvStartReadChainPoints	3-25
cvReadChainPoint	3-25
cvApproxPoly	3-26
cvDrawContours	3-26
cvContoursMoments	3-27
cvContourArea	3-28
cvMatchContours	3-28
cvCreateContourTree	3-30
cvContourFromContourTree	3-30
cvMatchContourTrees	3-31

Chapter 4

Geometry

Overview	4-1
Ellipse Fitting	4-1
Line Fitting	4-2
Convexity Defects	4-3
Reference	4-4
cvFitEllipse	4-4
cvFitLine2D	4-5
cvFitLine3D	4-6
cvProject3D	4-8
cvConvexHull	4-9
cvContourConvexHull	4-9
cvConvexHullApprox	4-10
cvContourConvexHullApprox	4-11
cvCheckContourConvexity	4-12
cvConvexityDefects	4-12
cvMinAreaRect	4-13
cvCalcPGH	4-14
cvMinEnclosingCircle	4-15

**Chapter 5
Features**

Fixed Filters	5-1
Overview	5-1
Sobel Derivatives	5-1
Optimal Filter Kernels with Floating Point Coefficients	5-5
First Derivatives	5-5
Second Derivatives	5-6
Laplacian Approximation	5-6
Reference	5-7
cvLaplace	5-7
cvSobel	5-7
Feature Detection Functions	5-8

Overview	5-8
Corner Detection	5-8
Canny Edge Detector	5-9
Reference	5-11
cvCanny	5-11
cvPreCornerDetect	5-12
cvCornerEigenValsAndVecs	5-12
cvCornerMinEigenVal	5-13
cvFindCornerSubPix	5-14
cvGoodFeaturesToTrack	5-16
Hough Transform	5-17
Overview	5-17
Reference	5-18
cvHoughLines	5-18
cvHoughLinesSDiv	5-19
Discussion	5-19
cvHoughLinesP	5-19
Discussion	5-20

Chapter 6

Image Statistics

Overview	6-1
Reference	6-2
cvCountNonZero	6-2
cvSumPixels	6-2
cvMean	6-3
cvMean_StdDev	6-3
cvMinMaxLoc	6-4
cvNorm	6-4
cvMoments	6-6
cvGetSpatialMoment	6-7
cvGetCentralMoment	6-7
cvGetNormalizedCentralMoment	6-8

cvGetHuMoments	6-9
----------------------	-----

Chapter 7 Pyramids

Overview	7-1
Reference	7-6
cvPyrDown	7-6
cvPyrUp	7-6
cvPyrSegmentation	7-7

Chapter 8 Morphology

Overview	8-1
Flat Structuring Elements for Gray Scale	8-3
Reference	8-6
cvCreateStructuringElementEx	8-6
cvReleaseStructuringElement	8-7
cvErode	8-7
cvDilate	8-8
cvMorphologyEx	8-9

Chapter 9 Background Subtraction

Overview	9-1
Reference	9-2
cvAcc	9-2
cvSquareAcc	9-3
cvMultiplyAcc	9-3
cvRunningAvg	9-4

Chapter 10 Distance Transform

Overview	10-1
Reference	10-1

cvDistTransform	10-1
-----------------------	------

Chapter 11 Threshold Functions

Overview	11-1
Reference	11-2
cvAdaptiveThreshold	11-2
cvThreshold	11-3

Chapter 12 Flood Fill

Overview	12-1
Reference	12-2
cvFloodFill	12-2

Chapter 13 Camera Calibration

Overview	13-1
Camera Parameters	13-1
Homography	13-2
Pattern	13-3
Lens Distortion	13-3
Rotation Matrix and Rotation Vector	13-5
Reference	13-5
cvCalibrateCamera	13-5
cvCalibrateCamera_64d	13-6
cvFindExtrinsicCameraParams	13-7
cvFindExtrinsicCameraParams_64d	13-8
cvRodrigues	13-9
cvRodrigues_64d	13-9
cvUndistortOnce	13-10
cvUndistortInit	13-11
cvUndistort	13-12

cvFindChessBoardCornerGuesses	13-12
-------------------------------------	-------

Chapter 14

View Morphing

Overview	14-1
Algorithm	14-1
Using Functions for View Morphing Algorithm	14-4
Reference	14-5
cvFindFundamentalMatrix	14-5
cvMakeScanlines	14-6
cvPreWarpImage	14-7
cvFindRuns	14-8
cvDynamicCorrespondMulti	14-9
cvMakeAlphaScanlines	14-9
cvMorphEpilinesMulti	14-10
cvPostWarpImage	14-11
cvDeleteMoire	14-12

Chapter 15

Motion Templates

Overview	15-1
Motion Representation and Normal Optical Flow Method	15-1
Motion Representation	15-1
A) Updating MHI Images	15-2
B) Making Motion Gradient Image	15-2
C) Finding Regional Orientation or Normal Optical Flow	15-4
Motion Segmentation	15-6
Reference	15-8
cvUpdateMotionHistory	15-8
cvCalcMotionGradient	15-8
cvCalcGlobalOrientation	15-9
cvSegmentMotion	15-10

Chapter 16**CamShift**

Overview	16-1
Mass Center Calculation for 2D Probability Distribution	16-3
CamShift Algorithm	16-3
Calculation of 2D Orientation	16-6
Reference	16-7
cvCamShift	16-7
cvMeanShift	16-8

Chapter 17**Active Contours**

Overview	17-1
Reference	17-3
cvSnakeImage	17-3

Chapter 18**Optical Flow**

Overview	18-1
Lucas & Kanade Technique	18-2
Horn & Schunck Technique	18-2
Block Matching	18-3
Reference	18-4
cvCalcOpticalFlowHS	18-4
cvCalcOpticalFlowLK	18-4
cvCalcOpticalFlowBM	18-5
cvCalcOpticalFlowPyrLK	18-6

Chapter 19**Estimators**

Overview	19-1
Definitions and Motivation	19-1
Models	19-1

Estimators	19-2
Kalman Filtering	19-2
Reference	19-4
cvCreateKalman	19-4
cvReleaseKalman	19-5
cvKalmanUpdateByTime	19-5
cvKalmanUpdateByMeasurement	19-6
ConDensation Algorithm	19-6
Implementation of Nonlinear Models	19-7
Reference	19-7
cvCreateConDensation	19-7
cvReleaseConDensation	19-8
cvConDensInitSampleSet	19-8
cvConDensUpdatebyTime	19-9

Chapter 20

POSIT

Overview	20-1
Background	20-1
Camera parameters	20-1
Geometric Image Formation	20-2
Pose Approximation Method	20-3
Algorithm	20-5
Reference	20-7
cvCreatePOSITObject	20-7
cvPOSIT	20-7
cvReleasePOSITObject	20-8

Chapter 21

Histogram

Overview	21-1
Histograms and Signatures	21-2
Example Ground Distances	21-5

Lower Boundary for EMD	21-6
Reference	21-6
cvCreateHist	21-6
cvReleaseHist	21-7
cvMakeHistHeaderForArray	21-8
cvQueryHistValue_1D	21-8
cvQueryHistValue_2D	21-9
cvQueryHistValue_3D	21-9
cvQueryHistValue_nD	21-10
cvGetHistValue_1D	21-10
cvGetHistValue_2D	21-11
cvGetHistValue_3D	21-11
cvGetHistValue_nD	21-12
cvGetMinMaxHistValue	21-12
cvNormalizeHist	21-13
cvThreshHist	21-13
cvCompareHist	21-14
cvCopyHist	21-15
cvSetHistBinRanges	21-15
cvCalcHist	21-16
cvCalcBackProject	21-16
cvCalcBackProjectPatch	21-17
cvCalcEMD	21-20

Chapter 22

Gesture Recognition

Overview	22-1
Reference	22-4
cvFindHandRegion	22-4
cvFindHandRegionA	22-5
cvCreateHandMask	22-6
cvCalcImageHomography	22-6
cvCalcProbDensity	22-7

cvMaxRect	22-8
-----------------	------

Chapter 23 Matrix Operations

Overview	23-1
Reference	23-2
cvmAlloc	23-2
cvmAllocArray	23-2
cvmFree	23-3
cvmFreeArray	23-3
cvmAdd	23-3
cvmSub	23-4
cvmScale	23-4
cvmDotProduct	23-5
cvmCrossProduct	23-5
cvmMul	23-6
cvmMulTransposed	23-6
cvmTranspose	23-7
cvmInvert	23-7
cvmTrace	23-8
cvmDet	23-8
cvmCopy	23-8
cvmSetZero_32f	23-9
cvmSetIdentity	23-9
cvmMahalonobis	23-10
cvmSVD	23-10
cvmEigenVV	23-11
cvmPerspectiveProject	23-12

Chapter 24 Eigen Objects

Overview	24-1
Reference	24-2

cvCalcCovarMatrixEx.....	24-2
cvCalcEigenObjects.....	24-3
cvCalcDecompCoeff	24-4
cvEigenDecomposite	24-5
cvEigenProjection	24-6
Use of Functions	24-6

Chapter 25

Embedded Hidden Markov Models

Overview.....	25-1
HMM Structures	25-1
Reference	25-3
cvCreate2DHMM	25-3
cvRelease2DHMM	25-3
cvCreateObsInfo	25-4
cvReleaseObsInfo	25-4
cvImgToObs_DCT.....	25-5
cvUniformImgSegm	25-6
cvInitMixSegm	25-6
cvEstimateHMMStateParams	25-7
cvEstimateTransProb	25-7
cvEstimateObsProb	25-8
cvEViterbi.....	25-8
cvMixSegmL2	25-9

Chapter 26

Drawing Primitives

Overview.....	26-1
Reference	26-2
cvLine	26-2
cvLineAA.....	26-3
cvRectangle	26-4
cvCircle	26-4

cvEllipse.....	26-5
cvEllipseAA.....	26-6
cvFillPoly.....	26-7
cvFillConvexPoly.....	26-8
cvPolyLine.....	26-8
cvPolyLineAA.....	26-9
cvInitFont.....	26-10
cvPutText.....	26-10
cvGetTextSize.....	26-11

Chapter 27 System Functions

Reference.....	27-1
cvLoadPrimitives.....	27-1
cvGetLibraryInfo.....	27-2

Chapter 28 Utility

Reference.....	28-1
cvAbsDiff.....	28-1
cvAbsDiffS.....	28-2
cvMatchTemplate.....	28-2
cvCvtPixToPlane.....	28-5
cvCvtPlaneToPix.....	28-5
cvConvertScale.....	28-6
cvInitLineIterator.....	28-7
cvSampleLine.....	28-8
cvGetRectSubPix.....	28-8
cvbFastArctan.....	28-9
cvSqrt.....	28-10
cvbSqrt.....	28-10
cvInvSqrt.....	28-11
cvbInvSqrt.....	28-11

cvbReciprocal	28-12
cvbCartToPolar	28-12
cvbFastExp	28-13
cvbFastLog	28-13
cvRandInit	28-14
cvbRand	28-14
cvFillImage	28-15
cvRandSetRange	28-15
cvKMeans	28-16

Bibliography

Index

Image Functions

1

The chapter describes basic functions for manipulating raster images.

Overview

OpenCV library represents images in the format `IplImage` that comes from Intel® Image Processing Library (IPL). IPL reference manual gives detailed information about the format, but, for completeness, it is also briefly described here.

Example 1-1 `IplImage` Structure Definition

```
typedef struct _IplImage {
    int nSize; /* size of iplImage struct */
    int ID; /* image header version */
    int nChannels;
    int alphaChannel;
    int depth; /* pixel depth in bits */
    char colorModel[4];
    char channelSeq[4];
    int dataOrder;
    int origin;
    int align; /* 4- or 8-byte align */
    int width;
    int height;
    struct _IplROI *roi; /* pointer to ROI if any */
    struct _IplImage *maskROI; /* pointer to mask ROI if any */
    void *imageId; /* use of the application */
    struct _IplTileInfo *tileInfo; /* contains information on tiling
*/
    int imageSize; /* useful size in bytes */
    char *imageData; /* pointer to aligned image */
    int widthStep; /* size of aligned line in bytes */
    int BorderMode[4]; /* the top, bottom, left,
and right border mode */
    int BorderConst[4]; /* constants for the top, bottom,
left, and right border */
    char *imageDataOrigin; /* ptr to full, nonaligned image */
}
```

Example 1-1 IplImage Structure Definition (continued)

```
} IplImage;
```

Only a few of the most important fields of the structure are described here. The fields *width* and *height* contain image width and height in pixels, respectively. The field *depth* contains information about the type of pixel values.

All possible values of the field *depth* listed in *ipl.h* header file include:

- IPL_DEPTH_8U - unsigned 8-bit integer value (unsigned *char*),
- IPL_DEPTH_8S - signed 8-bit integer value (signed *char* or simply *char*),
- IPL_DEPTH_16S - signed 16-bit integer value (short *int*),
- IPL_DEPTH_32S - signed 32-bit integer value (*int*),
- IPL_DEPTH_32F - 32-bit floating-point single-precision value (*float*).

In the above list the corresponding types in *C* are placed in parentheses. The parameter *nChannels* means the number of color planes in the image. Grayscale images contain a single channel, while color images usually include three or four channels. The parameter *origin* indicates, whether the top image row (*origin* == IPL_ORIGIN_TL) or bottom image row (*origin* == IPL_ORIGIN_BL) goes first in memory. Windows bitmaps are usually bottom-origin, while in most of other environments images are top-origin. The parameter *dataOrder* indicates, whether the color planes in the color image are interleaved (*dataOrder* == IPL_DATA_ORDER_PIXEL) or separate (*dataOrder* == IPL_DATA_ORDER_PLANE). The parameter *widthStep* contains the number of bytes between points in the same column and successive rows. The parameter *width* is not sufficient to calculate the distance, because each row may be aligned with a certain number of bytes to achieve faster processing of the image, so there can be some gaps between the end of *i*th row and the start of (*i*+1)th row. The parameter *imageData* contains pointer to the first row of image data. If there are several separate planes in the image (when *dataOrder* == IPL_DATA_ORDER_PLANE), they are placed consecutively as separate images with *height***nChannels* rows total.

It is possible to select some rectangular part of the image or a certain color plane in the image, or both, and process only this part. The selected rectangle is called "Region of Interest" or ROI. The structure `IplImage` contains the field `roi` for this purpose. If the pointer not `NULL`, it points to the structure `IplROI` that contains parameters of selected ROI, otherwise a whole image is considered selected.

Example 1-2 IplROI Structure Definition

```
typedef struct _IplROI {
    int    coi;        /* channel of interest or COI */
    int    xOffset;
    int    yOffset;
    int    width;
    int    height;
} IplROI;
```

As can be seen, `IplROI` includes ROI origin and size as well as COI ("Channel of Interest") specification. The field `coi`, equal to 0, means that all the image channels are selected, otherwise it specifies an index of the selected image plane.

Unlike IPL, OpenCV has several limitations in support of `IplImage`:

- Each function supports only a few certain depths and/or number of channels. For example, image statistics functions support only single-channel or three-channel images of the depth `IPL_DEPTH_8U`, `IPL_DEPTH_8S` or `IPL_DEPTH_32F`. The exact information about supported image formats is usually contained in the description of parameters or in the beginning of the chapter if all the functions described in the chapter are similar. It is quite different from IPL that tries to support all possible image formats in each function.
- OpenCV supports only interleaved images, not planar ones.
- The fields `colorModel`, `channelSeq`, `BorderMode`, and `BorderConst` are ignored.
- The field `align` is ignored and `widthStep` is simply used instead of recalculating it using the fields `width` and `align`.
- The fields `maskROI` and `tileInfo` must be zero.
- COI support is very limited. Now only image statistics functions accept non-zero COI values. Use the functions [cvCvtPixToPlane](#) and [cvCvtPlaneToPix](#) as a work-around.

- ROIs of all the input/output images have to match exactly one another. For example, input and output images of the function [cvErode](#) must have ROIs with equal sizes. It is unlike IPL again, where the ROIs intersection is actually affected.

Despite all the limitations, OpenCV still supports most of the commonly used image formats that can be supported by `IplImage` and, thus, can be successfully used with IPL on common subset of possible `IplImage` formats.

The functions described in this chapter are mainly short-cuts for operations of creating, destroying, and other common operations on `IplImage`, and they are often implemented as wrappers for original IPL functions.

Reference

cvCreateImageHeader

Allocates, initializes, and returns structure `IplImage`.

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

size Image width and height.

depth Image depth.

channels Number of channels.

Discussion

The function [cvCreateImageHeader](#) allocates, initializes, and returns the structure `IplImage`. This call is a shortened form of

```
iplCreateImageHeader( channels, 0, depth,  
    channels == 1 ? "GRAY" : "RGB",  
    channels == 1 ? "GRAY" : channels == 3 ? "BGR" : "BGRA",  
    IPL_DATA_ORDER_PIXEL, IPL_ORIGIN_TL, 4,
```



```
size.width, size.height,  
0,0,0,0);
```

cvCreateImage

Creates header and allocates data.

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

<i>size</i>	Image width and height.
<i>depth</i>	Image depth.
<i>channels</i>	Number of channels.

Discussion

The function [cvCreateImage](#) creates the header and allocates data. This call is a shortened form of

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

cvReleaseImageHeader

Releases header.

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

<i>image</i>	Double pointer to the deallocated header.
--------------	---

Discussion

The function [cvReleaseImageHeader](#) releases the header. This call is a shortened form of

```
if( image )
```

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

cvReleaseImage

Releases header and image data.

```
void cvReleaseImage( IplImage** image )
    image          Double pointer to the header of the deallocated image.
```

Discussion

The function [cvReleaseImage](#) releases the header and image data. This call is a shortened form of

```
if( image )
{
    iplDeallocate( *image, IPL_IMAGE_ALL );
    *image = 0;
}
```

cvCreateImageData

Allocates image data.

```
void cvCreateImageData( IplImage* image );
    image              Image header.
```

Discussion

The function [cvCreateImageData](#) allocates the image data. This call is a shortened form of

```
if( image->depth == IPL_DEPTH_32F )
{
    iplAllocateImageFP( image, 0, 0 );
}
else
{
    iplAllocateImage( image, 0, 0 );
}
```

cvReleaseImageData

Releases image data.

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

image Image header.

Discussion

The function [cvReleaseImageData](#) releases the image data. This call is a shortened form of

```
iplDeallocate( image, IPL_IMAGE_DATA );
```

cvSetImageData

Sets pointer to data and step parameters to given values.

```
void cvSetImageData( IplImage* image, void* data, int step );
```

<i>image</i>	Image header.
<i>data</i>	User data.
<i>step</i>	Distance between the raster lines in bytes.

Discussion

The function [cvSetImageData](#) sets the pointer to *data* and *step* parameters to given values.

cvSetImageCOI

Sets channel of interest to given value.

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

<i>image</i>	Image header.
<i>coi</i>	Channel of interest.

Discussion

The function [cvSetImageCOI](#) sets the channel of interest to a given value. If ROI is NULL and *coi* $\neq 0$, ROI is allocated.

cvSetImageROI

Sets image ROI to given rectangle.

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

<i>image</i>	Image header.
<i>rect</i>	ROI rectangle.

Discussion

The function [cvSetImageROI](#) sets the image ROI to a given rectangle. If ROI is `NULL` and the value of the parameter `rect` is not equal to the whole image, ROI is allocated.

cvGetImageRawData

Fills output variables with image parameters.

```
void cvGetImageRawData( const IplImage* image, uchar** data, int* step,
                        CvSize* roiSize );
```

<code>image</code>	Image header.
<code>data</code>	Pointer to the top-left corner of ROI.
<code>step</code>	Full width of the raster line, equals to <code>image->widthStep</code> .
<code>roiSize</code>	ROI width and height.

Discussion

The function [cvGetImageRawData](#) fills output variables with the image parameters. All output parameters are optional and could be set to `NULL`.

cvInitImageHeader

Initializes image header structure without memory allocation.

```
void cvInitImageHeader( IplImage* image, CvSize size, int depth, int channels,
                       int origin, int align, int clear );
```

<code>image</code>	Image header.
<code>size</code>	Image width and height.
<code>depth</code>	Image depth.

<i>channels</i>	Number of channels.
<i>origin</i>	IPL_ORIGIN_TL or IPL_ORIGIN_BL.
<i>align</i>	Alignment for the raster lines.
<i>clear</i>	If the parameter value equals 1, the header is cleared before initialization.

Discussion

The function [cvInitImageHeader](#) initializes the image header structure without memory allocation.

cvCopyImage

Copies entire image to another without considering ROI.

```
void cvCopyImage(IplImage* src, IplImage* dst);
```

<i>src</i>	Source image.
<i>dst</i>	Destination image.

Discussion

The function [cvCopyImage](#) copies the entire image to another without considering ROI. If the destination image is smaller, the destination image data is reallocated.

Pixel Access Macros

Overview

This section describes macros that are useful for fast and flexible access to image pixels. The basic ideas behind these macros are as follows:

1. Some structures of `CvPixelAccess` type are introduced. These structures contain all information about ROI and its current position. The only difference across all these structures is the data type, not the number of channels.
2. There exist fast versions for moving in a specific direction, e.g., `CV_MOVE_LEFT`, wrap and non-wrap versions. More complicated and slower macros are used for moving in an arbitrary direction that is passed as a parameter.
3. Most of the macros require the parameter `cs` that specifies the number of the image channels to enable the compiler to remove superfluous multiplications in case the image has a single channel, and substitute faster machine instructions for them in case of three and four channels.

Example 1-3 `CvPixelPosition` Structures Definition

```
typedef struct _CvPixelPosition8u
{
    unsigned char*    currline;
                        /* pointer to the start of the current
                        pixel line */
    unsigned char*    topline;
                        /* pointer to the start of the top pixel
                        line */
    unsigned char*    bottomline;
                        /* pointer to the start of the first
                        line which is below the image */
    int    x;    /* current x coordinate ( in pixels ) */
    int    width; /* width of the image ( in pixels ) */
    int    height; /* height of the image ( in pixels ) */
    int    step; /* distance between lines ( in
                  elements of single plane ) */
    int    step_arr[3]; /* array: ( 0, -step, step ).
                           It is used for vertical
                           moving */
} CvPixelPosition8u;

/*this structure differs from the above only in data type*/
typedef struct _CvPixelPosition8s
{
    char*    currline;
    char*    topline;
    char*    bottomline;
    int    x;
    int    width;
    int    height;
    int    step;
}
```

Example 1-3 CvPixelPosition Structures Definition (continued)

```

        int      step_arr[3];
    } CvPixelPosition8s;

    /* this structure differs from the CvPixelPosition8u only in data type
    */
    typedef struct _CvPixelPosition32f
    {
        float*   currline;
        float*   topline;
        float*   bottomline;
        int      x;
        int      width;
        int      height;
        int      step;
        int      step_arr[3];
    } CvPixelPosition32f;

```

CV_INIT_PIXEL_POS*Initializes one of CvPixelPosition structures.*

```
#define CV_INIT_PIXEL_POS( pos, origin, step, roi, x, y, orientation )
```

<i>pos</i>	Initialization of structure.
<i>origin</i>	Pointer to the left-top corner of ROI.
<i>step</i>	Width of the whole image in bytes.
<i>roi</i>	Width and height of ROI.
<i>x, y</i>	Initial position.
<i>orientation</i>	Image orientation; could be either CV_ORIGIN_TL - top/left orientation, or CV_ORIGIN_BL - bottom/left orientation.

CV_MOVE_TO

Moves to specified absolute position.

```
#define CV_MOVE_TO( pos, x, y, cs )  
    pos          Position structure.  
    x, y         Coordinates of the new position.  
    cs           Number of the image channels.
```

CV_MOVE

Moves by one pixel relative to current position.

```
#define CV_MOVE_LEFT( pos, cs )  
#define CV_MOVE_RIGHT( pos, cs )  
#define CV_MOVE_UP( pos, cs )  
#define CV_MOVE_DOWN( pos, cs )  
#define CV_MOVE_LU( pos, cs )  
#define CV_MOVE_RU( pos, cs )  
#define CV_MOVE_LD( pos, cs )  
#define CV_MOVE_RD( pos, cs )  
    pos          Position structure.  
    cs           Number of the image channels.
```

CV_MOVE_WRAP

Moves by one pixel relative to current position and wraps when position reaches image boundary.

```
#define CV_MOVE_LEFT_WRAP( pos, cs )  
#define CV_MOVE_RIGHT_WRAP( pos, cs )  
#define CV_MOVE_UP_WRAP( pos, cs )  
#define CV_MOVE_DOWN_WRAP( pos, cs )  
#define CV_MOVE_LU_WRAP( pos, cs )  
#define CV_MOVE_RU_WRAP( pos, cs )  
#define CV_MOVE_LD_WRAP( pos, cs )  
#define CV_MOVE_RD_WRAP( pos, cs )
```

<i>pos</i>	Position structure.
<i>cs</i>	Number of the image channels.

CV_MOVE_PARAM

Moves by one pixel in specified direction.

```
#define CV_MOVE_PARAM( pos, shift, cs )  
  
    pos           Position structure.  
  
    cs           Number of the image channels.  
  
    shift        Direction; could be any of the following:  
                  CV_SHIFT_NONE,  
                  CV_SHIFT_LEFT,  
                  CV_SHIFT_RIGHT,  
                  CV_SHIFT_UP,
```

```
CV_SHIFT_DOWN,  
CV_SHIFT_UL,  
CV_SHIFT_UR,  
CV_SHIFT_DL.
```

CV_MOVE_PARAM_WRAP

Moves by one pixel in specified direction with wrapping.

```
#define CV_MOVE_PARAM_WRAP( pos, shift, cs )  
    pos          Position structure.  
    cs           Number of the image channels.  
    shift        Direction; could be any of the following:  
                CV_SHIFT_NONE,  
                CV_SHIFT_LEFT,  
                CV_SHIFT_RIGHT,  
                CV_SHIFT_UP,  
                CV_SHIFT_DOWN,  
                CV_SHIFT_UL,  
                CV_SHIFT_UR,  
                CV_SHIFT_DL.
```


Dynamic Data Structures

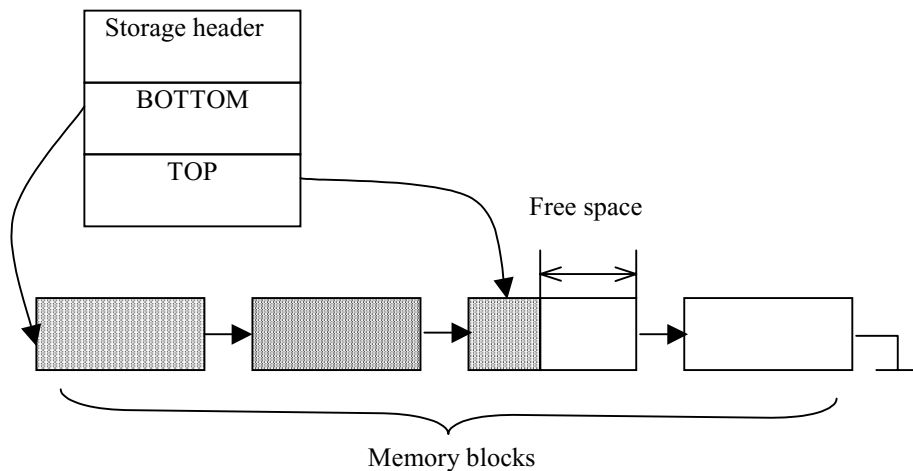
2

This chapter describes several resizable data structures and basic functions that are designed to operate on these structures.

Memory Storage

Overview

Memory storages provide the space for storing all the dynamic data structures described in this chapter. A storage consists of a header and a double-linked list of memory blocks. This list is treated as a stack, that is, the storage header contains a pointer to the block that is not occupied entirely and an integer value, the number of free bytes in this block. When the free space in the block has run out, the pointer is moved to the next block, if any, otherwise, a new block is allocated and then added to the list of blocks. All the blocks are of the same size and, therefore, this technique ensures an accurate memory allocation and helps avoid memory fragmentation if the blocks are large enough (see [Figure 2-1](#)).

Figure 2-1 Memory Storage Organization**Example 2-1 CvMemStorage Structure Definition**

```
typedef struct CvMemStorage
{
    CvMemBlock* bottom; /* first allocated block */
    CvMemBlock* top; /* 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 current block */
} CvMemStorage;
```

Example 2-2 CvMemBlock Structure Definition

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

Actual data of 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, the occasions on which the need for direct access to the memory blocks arises are quite rare. The structure described below stores the position of the stack top

that can be saved/restored:

Example 2-3 CvMemStoragePos Structure Definition

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
    int free_space;
}
CvMemStoragePos;
```

cvCreateMemStorage

Creates memory storage.

```
CvMemStorage* cvCreateMemStorage( int blockSize=0 );
```

blockSize Size of the memory blocks in the storage; bytes.

Discussion

The function [cvCreateMemStorage](#) creates a memory storage and returns the pointer to it. Initially the storage is empty. All fields of the header are set to 0. The parameter *blockSize* must be positive or zero; if the parameter equals 0, the block size is set to the default value, currently 64K.

cvCreateChildMemStorage

Creates child memory storage.

```
CvMemStorage* cvCreateChildMemStorage( CvMemStorage* parent );
```

parent Parent memory storage.

Discussion

The function [cvCreateChildMemStorage](#) creates a child memory storage similar to the simple memory storage except for the differences in the memory allocation/de-allocation 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. Note again, that in other aspects, the child storage is the same as the simple storage.

cvReleaseMemStorage

Releases memory storage.

```
void cvCreateChildMemStorage( CvMemStorage** storage );
```

storage Pointer to the released storage.

Discussion

The function [cvReleaseMemStorage](#) de-allocates all storage memory blocks or returns them to the parent, if any. Then it de-allocates the storage header and clears the pointer to the storage. All children of the storage must be released before the parent is released.

cvClearMemStorage

Clears memory storage

```
void cvClearMemStorage( CvMemStorage* storage );
```

storage Memory storage.

Discussion

The function [cvClearMemStorage](#) resets the top (free space boundary) of the storage to the very beginning. This function does not de-allocate any memory. If the storage has a parent, the function returns all blocks to the parent.

cvSaveMemStoragePos

Saves memory storage position.

```
void cvSaveMemStoragePos( CvMemStorage* storage, CvMemStoragePos* pos );
```

storage Memory storage.

pos Currently retrieved position of the in-memory storage top.

Discussion

The function [cvSaveMemStoragePos](#) saves the current position of the storage top to the parameter *pos*. This position can be retrieved further by the function [cvRestoreMemStoragePos](#).

cvRestoreMemStoragePos

Restores memory storage position.

```
void cvRestoreMemStoragePos( CvMemStorage* storage, CvMemStoragePos* pos );
```

storage Memory storage.

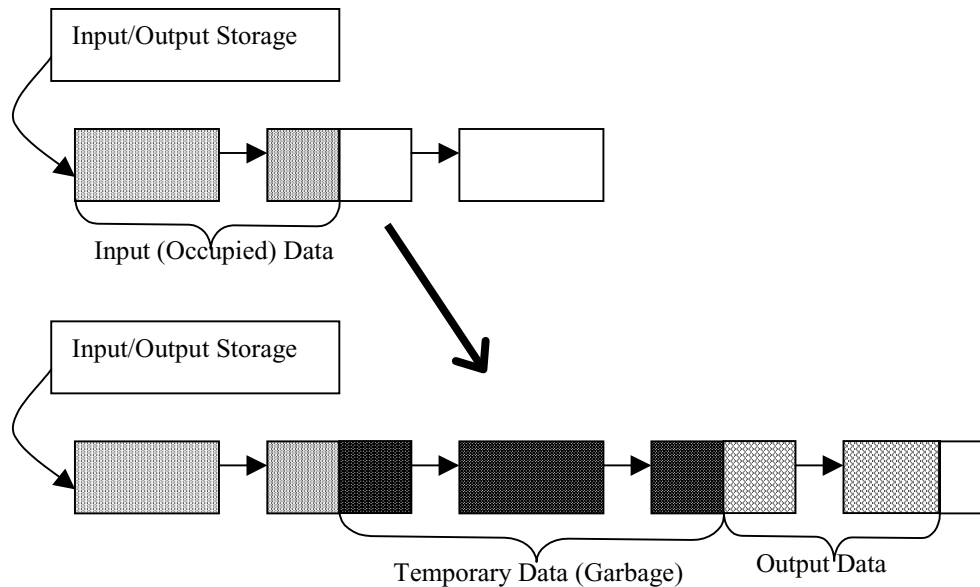
pos New storage top position.

Discussion

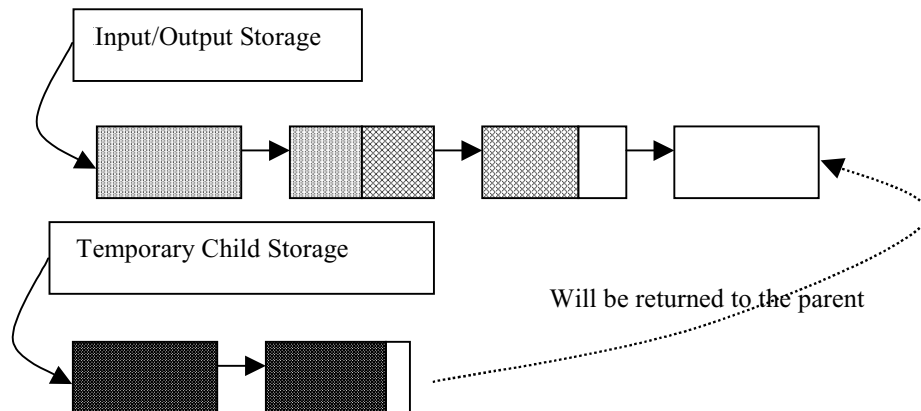
The function [cvRestoreMemStoragePos](#) restores the position of the storage top from the parameter *pos*. This function and the function [cvClearMemStorage](#) are the only methods to release memory occupied in memory blocks.

In other words, the occupied space and free space in the storage are continuous. If the user needs to process data and put the result to the storage, there arises a need for the storage space to be allocated for temporary results. In this case the user may simply write all the temporary data to that single storage. However, as a result garbage appears in the middle of the occupied part. See [Figure 2-2](#).

Figure 2-2 Storage Allocation for Temporary Results



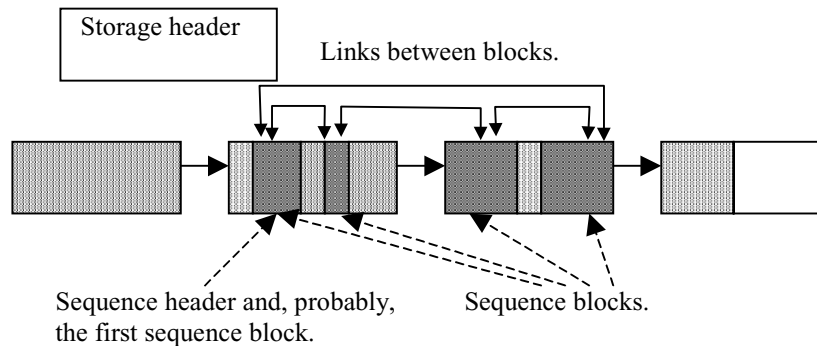
Saving/Restoring does not work in this case. Creating a child memory storage, however, can resolve this problem. The algorithm writes to both storages simultaneously, and, once done, releases the temporary storage. See [Figure 2-3](#).

Figure 2-3 Release of Temporary Storage

Sequences

Overview

A sequence is a resizable array of arbitrary type elements located in the memory storage. The sequence is discontinuous. Sequence data may be partitioned into several continuous blocks, called sequence blocks, that can be located in different memory blocks. Sequence blocks are connected into a circular double-linked list to store large sequences in several memory blocks or keep several small sequences in a single memory block. For example, such organization is suitable for storing contours. The sequence implementation provides fast functions for adding/removing elements to/from the head and tail of the sequence, so that the sequence implements a deque. The functions for inserting/removing elements in the middle of a sequence are also available but they are slower. The sequence is the basic type for many other dynamic data structures in the library, e.g., sets, graphs, and contours; just like all these types, the sequence never returns the occupied memory to the storage. However, the sequence keeps track of the memory released after removing elements from the sequence; this memory is used repeatedly. To return the memory to the storage, the user may clear a whole storage, or use save/restoring position functions, or keep temporary data in child storages.

Figure 2-4 Sequence Structure**Example 2-4 CvSequence Structure Definition**

```

#define CV_SEQUENCE_FIELDS()
    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      flags; /* miscellaneous flags */
    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 seq
grows */
    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;

```

Such an unusual definition 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()*. The field *header_size* contains the actual size of the sequence header and must be more 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 sequence in the vertical direction, that is, 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 `flags` contain miscellaneous information on the type of the sequence and should be discussed in greater detail. By convention, the lowest `CV_SEQ_ELTYPE_BITS` bits contain the ID of the element type. The current version has `CV_SEQ_ELTYPE_BITS` equal to 5, that is, it supports up to 32 non-overlapping element types now. The file `CVTypes.h` declares the predefined types.

Example 2-5 Standard Types of Sequence Elements

```

#define CV_SEQ_ELTYPE_POINT          1 /* (x,y) */
#define CV_SEQ_ELTYPE_CODE          2 /* freeman code: 0..7 */
#define CV_SEQ_ELTYPE_PPOINT        3 /* &(x,y) */
#define CV_SEQ_ELTYPE_INDEX         4 /* #(x,y) */
#define CV_SEQ_ELTYPE_GRAPH_EDGE    5 /* &next_o,&next_d,&vtx_o,
&vtx_d */
#define CV_SEQ_ELTYPE_GRAPH_VERTEX  6 /* first_edge, &(x,y) */
#define CV_SEQ_ELTYPE_TRIAN_ATR     7 /* vertex of the binary tree
*/
#define CV_SEQ_ELTYPE_CONNECTED_COMP 8 /* connected component */
#define CV_SEQ_ELTYPE_POINT3D       9 /* (x,y,z) */

```

The next `CV_SEQ_KIND_BITS` bits, also 5 in number, specify the kind of the sequence. Again, predefined kinds of sequences are declared in the file `CVTypes.h`.

Example 2-6 Standard Kinds of Sequences

```

#define CV_SEQ_KIND_SET              (0 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_CURVE           (1 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_BIN_TREE        (2 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_GRAPH           (3 << 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, 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.

Below follows the definition of the building block of sequences.

Example 2-7 **CvSeqBlock Structure Definition**

```
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 *start_index* 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 *start_index* = 2, then pairs $\langle \textit{start_index}, \textit{count} \rangle$ for the sequence blocks are $\langle 2, 3 \rangle$, $\langle 5, 5 \rangle$, and $\langle 10, 2 \rangle$ correspondingly. The parameter *start_index* of the first block is usually 0 unless some elements have been inserted at the beginning of the sequence.

cvCreateSeq

Creates sequence.

```
CvSeq* cvCreateSeq(int seqFlags, int headerSize, int elemSize, CvMemStorage*
storage);
```

<i>seqFlags</i>	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.
<i>headerSize</i>	Size of the sequence header; must be more than or equal to <code>sizeof(CvSeq)</code> . If a specific type or its extension is indicated, this type must fit the base type header.
<i>elemSize</i>	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 <code>CV_SEQ_ELTYPE_POINT</code> should be specified and the parameter <i>elemSize</i> must be equal to <code>sizeof(CvPoint)</code> .
<i>storage</i>	Sequence location.

Discussion

The function [cvCreateSeq](#) creates a sequence and returns the pointer to it. The function allocates the sequence header in the storage block as one continuous chunk and fills the parameter *elemSize*, flags *headerSize*, and *storage* with passed values, sets the parameter *deltaElems* (see the function [cvSetSeqBlockSize](#)) to the default value, and clears other fields, including the space behind `sizeof(CvSeq)`.



NOTE. All headers in the memory storage, including sequence headers and sequence block headers, are aligned with the 4-byte boundary.

cvSetSeqBlockSize

Sets up sequence block size.

```
void cvSetSeqBlockSize( CvSeq* seq, int blockSize );
```

seq Sequence.
blockSize Desirable block size.

Discussion

The function [cvSetSeqBlockSize](#) affects the memory allocation granularity. When the free space in the internal sequence buffers has run out, the function allocates *blockSize* bytes in the storage. 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, the lower the sequence fragmentation probability, but the more space in the storage is wasted. When the sequence is created, the parameter *blockSize* is set to the default value ~1K. The function can be called any time after the sequence is created and affects future allocations. The final block size can be different from the one desired, e.g., if it is larger than the storage block size, or smaller than the sequence header size plus the sequence element size.

The next four functions [cvSeqPush](#), [cvSeqPop](#), [cvSeqPushFront](#), [cvSeqPopFront](#) add or remove elements to/from one of the sequence ends. Their time complexity is $O(1)$, that is, all these operations do not shift existing sequence elements.

cvSeqPush

Adds element to sequence end.

```
void cvSeqPush( CvSeq* seq, void* element );
```

seq Sequence.
element Added element.

Discussion

The function [cvSeqPush](#) adds an element to the end of the sequence. Although this function can be used to create a sequence element by element, there is a faster method (refer to [Writing and Reading Sequences](#)).

cvSeqPop

Removes element from sequence end.

```
void cvSeqPop( CvSeq* seq, void* element );
```

seq Sequence.

element Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

Discussion

The function [cvSeqPop](#) removes an element from the sequence. The function reports an error if the sequence is already empty.

cvSeqPushFront

Adds element to sequence beginning.

```
void cvSeqPushFront( CvSeq* seq, void* element );
```

seq Sequence.

element Added element.

Discussion

The function [cvSeqPushFront](#) adds an element to the beginning of the sequence.

cvSeqPopFront

Removes element from sequence beginning.

```
void cvSeqPopFront( CvSeq* seq, void* element );
```

<i>seq</i>	Sequence.
<i>element</i>	Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

Discussion

The function [cvSeqPopFront](#) removes an element from the beginning of the sequence. The function reports an error if the sequence is already empty.

Next two functions [cvSeqPushMulti](#), [cvSeqPopMulti](#) are batch versions of the PUSH/POP operations.

cvSeqPushMulti

Pushes several elements to sequence end.

```
void cvSeqPushMulti(CvSeq* seq, void* elements, int count );
```

<i>seq</i>	Sequence.
<i>elements</i>	Added elements.
<i>count</i>	Number of elements to push.

Discussion

The function [cvSeqPushMulti](#) adds several elements to the end of the 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.

cvSeqPopMulti

Removes several elements from sequence end.

```
void cvSeqPopMulti( CvSeq* seq, void* elements, int count );
```

<i>seq</i>	Sequence.
<i>elements</i>	Removed elements.
<i>count</i>	Number of elements to pop.

Discussion

The function [cvSeqPopMulti](#) removes several elements from the 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.

cvSeqInsert

Inserts element in sequence middle.

```
void cvSeqInsert( CvSeq* seq, int beforeIndex, void* element );
```

<i>seq</i>	Sequence.
<i>beforeIndex</i>	Index before which the element is inserted. Inserting before 0 is equal to <i>cvSeqPushFront</i> and inserting before <i>seq->total</i> is equal to <i>cvSeqPush</i> . The index values in these two examples are boundaries for allowed parameter values.
<i>element</i>	Inserted element.

Discussion

The function [cvSeqInsert](#) shifts the sequence elements from the inserted position to the nearest end of the sequence before it copies an element there, therefore, the algorithm time complexity is $O(n/2)$.

cvSeqRemove

Removes element from sequence middle.

```
void cvSeqRemove( CvSeq* seq, int index );
```

seq Sequence.
index Index of removed element.

Discussion

The function [cvSeqRemove](#) removes elements with the given index. If the index is negative or greater than the total number of elements less 1, the function reports an error. An attempt to remove an element from an empty sequence is a specific case of this situation. The function removes an element by shifting the sequence elements from the nearest end of the sequence *index*.

cvClearSeq

Clears sequence.

```
void cvClearSeq( CvSeq* seq );
```

seq Sequence.

Discussion

The function [cvClearSeq](#) empties the sequence. The function does not return the memory to the storage, but this memory is used again when new elements are added to the sequence. This function time complexity is $O(1)$.

cvGetSeqElem

Returns n-th element of sequence.

```
char* cvGetSeqElem( CvSeq* seq, int index, CvSeqBlock** block=0 );
```

<i>seq</i>	Sequence.
<i>index</i>	Index of element.
<i>block</i>	Optional argument. If the pointer is not NULL, the address of the sequence block that contains the element is stored in this location.

Discussion

The function [cvGetSeqElem](#) finds the element with the given index in the sequence and returns the pointer to it. In addition, the function can return the pointer to the sequence block that contains the element. 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 there is a big chance that the sequence consists of a single sequence block or desired element is 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, if so, returns the element, otherwise the macro calls the main function [cvGetSeqElem](#). Negative indices always cause the *cvGetSeqElem* call.

cvSeqElemIdx

Returns index of concrete sequence element.

```
int cvSeqElemIdx( CvSeq* seq, void* element, CvSeqBlock** block=0 );
```

<i>seq</i>	Sequence.
<i>element</i>	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.

Discussion

The function [cvSeqElemIdx](#) returns the index of a sequence element or a negative number if the element is not found.

cvCvtSeqToArray

Copies sequence to one continuous block of memory.

```
void* cvCvtToArray( CvSeq* seq, void* array, CvSlice slice=CV_WHOLE_SEQ(seq) );
```

<i>seq</i>	Sequence.
<i>array</i>	Pointer to the destination array that must fit all the sequence elements.
<i>slice</i>	Start and end indices within the sequence so that the corresponding subsequence is copied.

Discussion

The function [cvCvtSeqToArray](#) copies the entire sequence or subsequence to the specified buffer and returns the pointer to the buffer.

cvMakeSeqHeaderForArray

Constructs sequence from array.

```
void cvMakeSeqHeaderForArray( int seqType, int headerSize, int elemSize, void* array, int total, CvSeq* sequence, CvSeqBlock* block );
```

<i>seqType</i>	Type of the created sequence.
<i>headerSize</i>	Size of the header of the sequence. Parameter <i>sequence</i> must point to the structure of that size or greater size.
<i>elemSize</i>	Size of the sequence element.
<i>array</i>	Pointer to the array that makes up the sequence.
<i>total</i>	Total number of elements in the sequence. The number of array elements must be equal to the value of this parameter.
<i>sequence</i>	Pointer to the local variable that is used as the sequence header.
<i>block</i>	Pointer to the local variable that is the header of the single sequence block.

Discussion

The function [cvMakeSeqHeaderForArray](#), the exact opposite of the function [cvCvtSeqToArray](#), builds a sequence from an array. The sequence always consists of a single sequence block, and the total number of elements may not be greater than the value of the parameter *total*, though the user may remove elements from the sequence, then add other elements to it with the above restriction.

Writing and Reading Sequences

Overview

Although the functions and macros described below are irrelevant in theory because functions like [cvSeqPush](#) and [cvGetSeqElem](#) enable the user to write to sequences and read from them, the writing/reading functions and macros are very useful in practice because of their speed.

The following problem could provide an illustrative example. If the task is to create a function that forms a sequence from *N* random values, the PUSH version runs as follows:

```
CvSeq* create_seq1( CvStorage* storage, int N ) {  
    CvSeq* seq = cvCreateSeq( 0, sizeof(*seq), sizeof(int), storage);  
    for( int i = 0; i < N; i++ ) {
```

```
int a = rand();
cvSeqPush( seq, &a );
}
return seq;
}
```

The second version makes use of the fast writing scheme, that includes the following steps: initialization of the writing process (creating writer), writing, closing the writer (flush).

```
CvSeq* create_seq1( CvStorage* storage, int N ) {
CvSeqWriter writer;
cvStartWriteSeq( 0, sizeof(*seq), sizeof(int),
storage, &writer );
for( int i = 0; i < N; i++ ) {
int a = rand();
CV_WRITE_SEQ_ELEM( a, writer );
}
return cvEndWriteSeq( &writer );
}
```

If $N = 100000$ and 500MHz Pentium® III processor is used, the first version takes 230 milliseconds and the second one takes 111 milliseconds to finish. These characteristics assume that the storage already contains a sufficient number of blocks so that no new blocks are allocated. A comparison with the simple loop that does not use sequences gives an idea as to how effective and efficient this approach is.

```
int* create_seq3( int* buffer, int N ) {
for( i = 0; i < N; i++ ) {
buffer[i] = rand();
}
return buffer;
}
```

This function takes 104 milliseconds to finish using the same machine.

Generally, the sequences do not make a great impact on the performance and the difference is very insignificant (less than 7% in the above example). However, the advantage of sequences is that the user can operate the input or output data even without knowing their amount in advance. These structures enable him/her to allocate memory iteratively. Another problem solution would be to use lists, yet the sequences are much faster and require less memory.

Reference

cvStartAppendToSeq

Initializes process of writing to sequence.

```
void cvStartAppendToSeq( CvSeq* seq, CvSeqWriter* writer );
```

<i>seq</i>	Pointer to the sequence.
<i>writer</i>	Pointer to the working structure that contains the current status of the writing process.

Discussion

The function [cvStartAppendToSeq](#) initializes the writer to write to the sequence. Written elements are added to the end of the sequence. Note that during the writing process other operations on the sequence may yield incorrect result or even corrupt the sequence (see Discussion of the function [cvFlushSeqWriter](#)).

cvStartWriteSeq

Creates new sequence and initializes writer for it.

```
void cvStartWriteSeq(int seqFlags, int headerSize, int elemSize, CvMemStorage*  
    storage, CvSeqWriter* writer);
```

<i>seqFlags</i>	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.
<i>headerSize</i>	Size of the sequence header. The parameter value may not be less than <code>sizeof(CvSeq)</code> . If a certain type or extension is specified, it must fit the base type header.
<i>elemSize</i>	Size of the sequence elements in bytes; must be consistent with the sequence type. For example, if the sequence of points is created (element type <code>CV_SEQ_ELTYPE_POINT</code>), then the parameter <i>elemSize</i> must be equal to <code>sizeof(CvPoint)</code> .
<i>storage</i>	Sequence location.
<i>writer</i>	Pointer to the writer status.

Discussion

The function [cvStartWriteSeq](#) is the exact sum of the functions [cvCreateSeq](#) and [cvStartAppendToSeq](#).

cvEndWriteSeq

Finishes process of writing.

```
CvSeq* cvEndWriteSeq( CvSeqWriter* writer);
```

writer Pointer to the writer status.

Discussion

The function [cvEndWriteSeq](#) finishes the writing process and returns the pointer to the resulting sequence. The function also truncates the last sequence block to return the whole of unfilled space to the memory storage. After that the user may read freely from the sequence and modify it.

cvFlushSeqWriter

Updates sequence headers using writer state.

```
void cvFlushSeqWriter( CvSeqWriter* writer );
```

writer Pointer to the writer status.

Discussion

The function [cvFlushSeqWriter](#) 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 any time. Frequent flushes are not recommended, the function [cvSeqPush](#) is preferred.

cvStartReadSeq

Initializes process of sequential reading from sequence.

```
void cvStartReadSeq( CvSeq* seq, CvSeqReader* reader, int reverse=0 );
```

seq Sequence.

reader Pointer to the reader status.

reverse Whenever the parameter value equals 0, the reading process is going in the forward direction, that is, from the beginning to the end, otherwise the reading process direction is reverse, from the end to the beginning.

Discussion

The function [cvStartReadSeq](#) initializes the reader structure. After that all the sequence elements from the first down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM(elem, reader)` that is similar to `CV_WRITE_SEQ_ELEM`. The function puts the reading pointer to the last sequence element if the parameter *reverse* does not equal zero. After that the macro `CV_REV_READ_SEQ_ELEM(elem, reader)` can be used to get sequence elements from the last to the first. Both macros put the sequence element to *elem* and move the reading pointer forward (`CV_READ_SEQ_ELEM`) or backward (`CV_REV_READ_SEQ_ELEM`). 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`. Neither function ends reading since the reading process does not modify the sequence, nor requires any temporary buffers. The reader field *ptr* points to the current element of the sequence that is to be read first.

cvGetSeqReaderPos

Returns index of element to read position.

```
int cvGetSeqReaderPos( CvSeqReader* reader );
```

reader Pointer to the reader status.

Discussion

The function [cvGetSeqReaderPos](#) returns the index of the element in which the reader is currently located.

cvSetSeqReaderPos

Moves read position to specified index.

```
void cvGetSeqReaderPos( CvSeqReader* reader, int index, int isRelative=0 );
```

<i>reader</i>	Pointer to the reader status.
<i>index</i>	Position where the reader must be moved.
<i>isRelative</i>	If the parameter value is not equal to zero, the index means an offset relative to the current position.

Discussion

The function [cvSetSeqReaderPos](#) moves the read position to the absolute or relative position. This function allows for cycle character of the sequence.

Sets

Overview

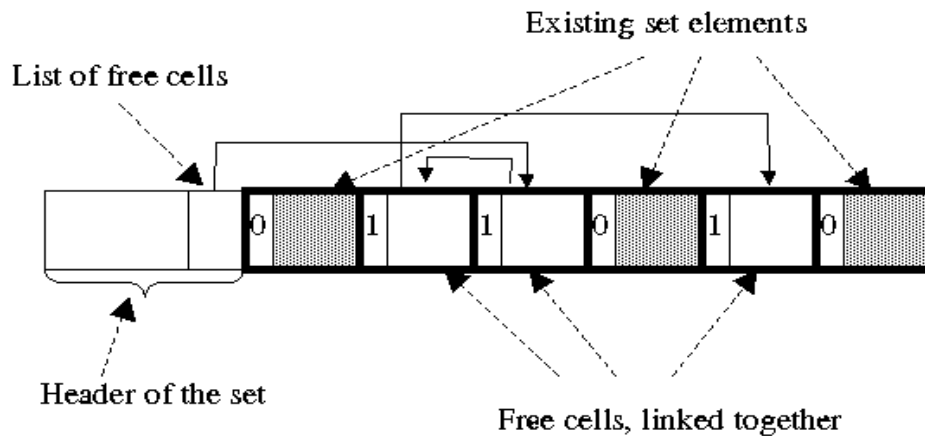
The set structure is mostly based on sequences but has a totally different purpose. For example, the user is unable to use sequences for location of the dynamic structure elements that have links between one another because if some elements have been removed from the middle of the sequence, other sequence elements are moved to another location and their addresses and indices change. In this case all links have to be fixed anew. Another aspect of this problem is that removing elements from the middle of the sequence is slow, with time complexity of $O(n)$, where n is the number of elements in the sequence.

The problem solution lies in making the structure sparse and unordered, that is, whenever a structure element is removed, other elements must stay where they have been, while the cell previously occupied by the element is added to the pool of free cells; when a new element is inserted into the structure, the vacant cell is used to store this new element. The set (See [Example 2-8](#)) operates in this very way.

The set looks like a list yet keeps no links between the structure elements. However, the user is free to make and keep such lists, if needed. The set is implemented as a sequence subclass; the set uses sequence elements as cells and organizes a list of free cells.

See [Figure 2-5](#) for an example of a set. For simplicity, the figure does not show division of the sequence/set into memory blocks and sequence blocks.

Figure 2-5 Set Structure



The set elements, both existing and free cells, are all sequence elements. A special bit indicates whether the set element exists or not: in the above diagram the bits marked by 1 are free cells and the ones marked by 0 are occupied cells. The macro

`CV_IS_SET_ELEM_EXISTS(set_elem_ptr)` uses this special bit to return a non-zero value if the set element specified by the parameter `set_elem_ptr` belongs to the set, and 0 otherwise. Below follows the definition of the structure `CvSet`:

Example 2-8 CvSet Structure Definition

```
#define CV_SET_FIELDS()      \
    CV_SEQUENCE_FIELDS()    \
    CvMemBlock* free_elems;

typedef struct CvSet
{
    CV_SET_FIELDS()
}
CvSet;
```

In other words, a set is a sequence plus a list of free cells.

There are two modes of working with sets. The first mode uses indices for referencing the set elements within a sequence while the second mode uses pointers for the same purpose. Whereas at times the first mode is a better option, the pointer mode is faster because it does not need to find the set elements by their indices, which is done in the same way as in simple sequences. The decision on which method should be used in each particular case depends on the type of operations to be performed on the set and the way these operations should be performed.

The ways in which a new set is created and new elements are added to the existing set are the same in either mode, the only difference between the two being the way the elements are removed from the set. The user may even use both methods of access simultaneously, provided he or she has enough memory available to store both the index and the pointer to each element.

Like in sequences, the user may create a set with elements of arbitrary type and specify any size of the header, which, however, may not be less than `sizeof(CvSet)`. At the same time the size of the set elements is restricted to be not less than 8 bytes and divisible by 4. The reason behind this restriction is the internal set organization: if the set has a free cell available, the first 4-byte field of this set element is used as a pointer to the next free cell, which enables the user to keep track of all free cells. The second 4-byte field of the cell contains the cell to be returned when the cell becomes occupied.

When the user removes a set element while operating in the index mode, the index of the removed element is passed and stored in the released cell again. The bit indicating whether the element belongs to the set is the least significant bit of the first 4-byte field. This is the reason why all the elements must have their size divisible by 4. In this case they are all aligned with the 4-byte boundary, so that the least significant bits of their addresses are always 0.

In free cells the corresponding bit is set to 1 and, in order to get the real address of the next free cell, the functions mask this bit off. On the other hand, if the cell is occupied, the corresponding bit must be equal to 0, which is the second and last restriction: the least significant bit of the first 4-byte field of the set element must be 0, otherwise the corresponding cell is considered free. If the set elements comply with this restriction, e.g., if the first field of the set element is a pointer to another set element or to some aligned structure outside the set, then the only restriction left is a non-zero number of 4- or 8-byte fields after the pointer. If the set elements do not comply with this restriction, e.g., if the user wants to store integers in the set, the user may derive his or her own structure from the structure *CvSetElem* or include it into his or her structure as the first field.

Example 2-9 CvSetElem Structure Definition

```
#define CV_SET_ELEM_FIELDS()    \
    int* aligned_ptr;
typedef struct _CvSetElem
{
    CV_SET_ELEM_FIELDS()
}
CvSetElem;
```

The first field is a dummy field and is not used in the occupied cells, except the least significant bit, which is 0. With this structure the integer element could be defined as follows:

```
typedef struct _IntSetElem
{
    CV_SET_ELEM_FIELDS()
    int value;
}
IntSetElem;
```


Reference

cvCreateSet

Creates empty set.

```
CvSet* cvCreateSet( int setFlags, int headerSize, int elemSize, CvMemStorage*
                    storage );
```

<i>setFlags</i>	Type of the created set.
<i>headerSize</i>	Set header size; may not be less than <code>sizeof(CvSeq)</code> .
<i>elemSize</i>	Set element size; may not be less than 8 bytes, must be divisible by 4.
<i>storage</i>	Future set location.

Discussion

The function [cvCreateSet](#) creates an empty set with the specified header size and returns the pointer to the set. The function simply redirects the call to the function [cvCreateSeq](#).

cvSetAdd

Adds element to set.

```
int cvSetAdd( CvSet* set, CvSet* elem, CvSet** insertedElem=0 );
```

<i>set</i>	Set.
<i>elem</i>	Optional input argument, inserted element. If not <code>NULL</code> , the function copies the data to the allocated cell omitting the first 4-byte field.
<i>insertedElem</i>	Optional output argument; points to the allocated cell.

Discussion

The function [cvSetAdd](#) allocates the new cell, optionally copies input element data to it, and returns the pointer and the index to the cell. The index value is taken from the second 4-byte field of the cell. In case the cell was previously deleted and a wrong index was specified, the function returns this wrong index. However, if the user works in the pointer mode, no problem occurs and the pointer stored at the parameter *insertedElem* may be used to get access to the added set element.

cvSetRemove

Removes element from set.

```
void cvSetRemove( CvSet* set, int index );
```

<i>set</i>	Set.
<i>index</i>	Index of the removed element.

Discussion

The function [cvSetRemove](#) removes an element with specified index from the set. The function is typically used when set elements are accessed by their indices. If pointers are used, the macro `CV_REMOVE_SET_ELEM(set, index, elem)`, where *elem* is a pointer to the removed element and *index* is any non-negative value, may be used to remove the element. Alternative way to remove an element by its pointer is to calculate index of the element via the function [cvSeqElemIdx](#) after which the function [cvSetRemove](#) may be called, but this method is much slower than the macro.

cvGetSetElem

Finds set element by index.

```
CvSetElem* cvGetSetElem( CvSet* set, int index );
```

<i>set</i>	Set.
<i>index</i>	Index of the set element within a sequence.

Discussion

The function [cvGetSetElem](#) finds the set element by index. The function returns the pointer to it or 0 if the index is invalid or the corresponding cell is free. The function supports negative indices through calling the function [cvGetSeqElem](#).



NOTE. *The user can check whether the element belongs to the set with the help of the macro `CV_IS_SET_ELEM_EXISTS(elem)` once the pointer is set to a set element.*

cvClearSet

Clears set.

```
void cvClearSet( CvSet* set );
```

<i>set</i>	Cleared set.
------------	--------------

Discussion

The function [cvClearSet](#) empties the set by calling the function [cvClearSeq](#) and setting the pointer to the list of free cells. The function takes $O(1)$ time.

Graphs

Overview

The structure set described above helps to build graphs because a graph consists of two sets, namely, vertices and edges, that refer to each other.

Example 2-10 CvGraph Structure Definition

```
#define CV_GRAPH_FIELDS() \
    CV_SET_FIELDS() \
    CvSet* edges;
typedef struct _CvGraph
{
    CV_GRAPH_FIELDS()
}
CvGraph;
```

In OOP terms, the graph structure is derived from the set of vertices and includes a set of edges. Besides, special data types exist for graph vertices and graph edges.

Example 2-11 Definitions of CvGraphEdge and CvGraphVtx Structures

```
#define CV_GRAPH_EDGE_FIELDS() \
    struct _CvGraphEdge* next[2]; \
    struct _CvGraphVertex* vtx[2];

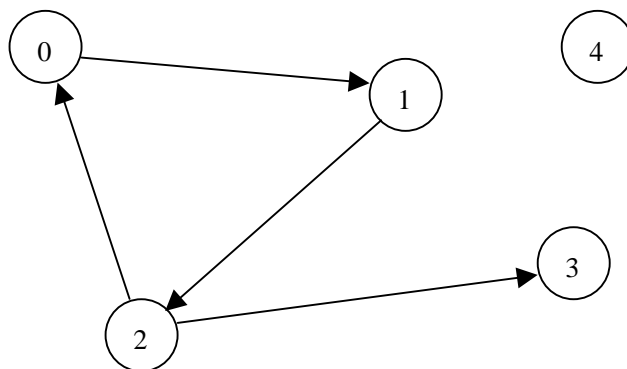
#define CV_GRAPH_VERTEX_FIELDS() \
    struct _CvGraphEdge* first;

typedef struct _CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS()
}
CvGraphEdge;

typedef struct _CvGraphVertex
{
    CV_GRAPH_VERTEX_FIELDS()
}
CvGraphVtx;
```

The graph vertex has a single predefined field that assumes the value of 1 when pointing to the first edge incident to the vertex, or 0 if the vertex is isolated. The edges incident to a vertex make up the single linked non-cycle list. The edge structure is more complex: `vtx[0]` and `vtx[1]` are the starting and ending vertices of the edge, `next[0]` and `next[1]` are the next edges in the incident lists for `vtx[0]` and `vtx[1]` respectively. In other words, each edge is included in two incident lists since any edge is incident to both the starting and the ending vertices. For example, consider the following oriented graph (see below for more information on non-oriented graphs).

Figure 2-6 Sample Graph



The structure can be created with the following code:

```

CvGraph* graph = cvCreateGraph( CV_SEQ_KIND_GRAPH |
CV_GRAPH_FLAG_ORIENTED,
sizeof(CvGraph),
sizeof(CvGraphVtx)+4,
sizeof(CvGraphEdge),
storage);
for( i = 0; i < 5; i++ )
{
cvGraphAddVtx( graph, 0, 0 );/* arguments like in

```

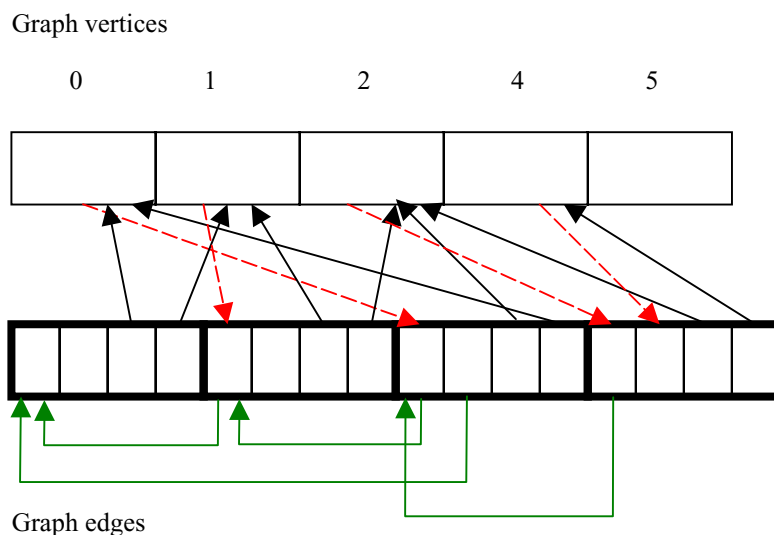
```

cvSetAdd*/
}
cvGraphAddEdge( graph, 0, 1, 0, 0 ); /* connect vertices 0
and 1, other two arguments like in cvSetAdd */
cvGraphAddEdge( graph, 1, 2, 0, 0 );
cvGraphAddEdge( graph, 2, 0, 0, 0 );
cvGraphAddEdge( graph, 2, 3, 0, 0 );

```

The internal structure comes to be as follows:

Figure 2-7 Internal Structure for Sample Graph Shown in Figure 2-6



Undirected graphs can also be represented by the structure `CvGraph`. If the non-oriented edges are substituted for the oriented ones, the internal structure remains the same. However, the function used to find edges succeeds only when it finds the edge from 3 to 2, as the function looks not only for edges from 3 to 2 but also from 2 to 3, and such an edge is present as well. As follows from the code, the type of the graph is specified when the graph is created, and the user can change the behavior of the edge searching function by specifying or omitting the flag `CV_GRAPH_FLAG_ORIENTED`. Two

edges connecting the same vertices in undirected graphs may never be created because the existence of the edge between two vertices is checked before a new edge is inserted between them. However, internally the edge can be coded from the first vertex to the second or vice versa. Like in sets, the user may work with either indices or pointers. The graph implementation uses only pointers to refer to edges, but the user can choose indices or pointers for referencing vertices.

Reference

cvCreateGraph

Creates empty graph.

```
CvGraph* cvCreateGraph( int graphFlags, int headerSize, int vertexSize, int
                        edgeSize, CvStorage* storage );
```

<i>graphFlags</i>	Type of the created graph. The kind of the sequence must be graph (CV_SEQ_KIND_GRAPH) and flag CV_GRAPH_FLAG_ORIENTED allows the oriented graph to be created. User may choose other flags, as well as types of graph vertices and edges.
<i>headerSize</i>	Graph header size; may not be less than <code>sizeof(CvGraph)</code> .
<i>vertexSize</i>	Graph vertex size; must be greater than <code>sizeof(CvGraphVertex)</code> and meet all restrictions on the set element.
<i>edgeSize</i>	Graph edge size; may not be less than <code>sizeof(CvGraphEdge)</code> and must be divisible by 4.
<i>storage</i>	Future location of the graph.

Discussion

The function [cvCreateGraph](#) creates an empty graph, that is, two empty sets, a set of vertices and a set of edges, and returns it.

cvGraphAddVtx

Adds vertex to graph.

```
int cvGraphAddVtx( CvGraph* graph, CvGraphVtx* vtx, CvGraphVtx** insertedVtx=0
);
```

<i>graph</i>	Graph.
<i>vtx</i>	Optional input argument. Similar to the parameter <i>elem</i> of the function cvSetAdd , the parameter <i>vtx</i> could be used to initialize new vertices with concrete values. If <i>vtx</i> is not <code>NULL</code> , the function copies it to a new vertex, except the first 4-byte field.
<i>insertedVtx</i>	Optional output argument. If not <code>NULL</code> , the address of the new vertex is written there.

Discussion

The function [cvGraphAddVtx](#) adds a vertex to the graph and returns the vertex index.

cvGraphRemoveVtx

Removes vertex from graph.

```
void cvGraphRemoveAddVtx( CvGraph* graph, int vtxIdx );
```

<i>graph</i>	Graph.
<i>vtxIdx</i>	Index of the removed vertex.
<i>vtx</i>	Pointer to the removed vertex.

Discussion

The function [cvGraphRemoveVtx](#) removes a vertex from the graph together with all the edges incident to it.

cvGraphRemoveVtxByPtr

Removes vertex from graph.

```
void cvGraphRemoveVtxByPtr( CvGraph* graph, CvGraphVtx* vtx );
```

<i>graph</i>	Graph.
<i>vtx</i>	Pointer to the removed vertex.

Discussion

The function [cvGraphRemoveVtxByPtr](#) removes a vertex from the graph together with all the edges incident to it.

cvGraphAddEdge

Adds edge to graph.

```
int cvGraphAddEdge( CvGraph* graph, int startIdx, int endIdx, CvGraphEdge*  
edge, CvGraphEdge** insertedEdge=0 );
```

<i>graph</i>	Graph.
<i>startIdx</i>	Index of the starting vertex of the edge.
<i>endIdx</i>	Index of the ending vertex of the edge.
<i>edge</i>	Optional input parameter, initialization data for the edge. If not <code>NULL</code> , the parameter is copied starting from the 5 th 4-byte field.
<i>insertedEdge</i>	Optional output parameter to contain the address of the inserted edge within the edge set.

Discussion

The function [cvGraphAddEdge](#) adds the edge to the graph given the starting and the ending vertices. The function returns the index of the inserted edge, which is the value of the second 4-byte field of the free cell.

The function reports an error if

- the edge that connects the vertices already exists; in this case graph orientation is taken into account;
- a pointer is `NULL` or indices are invalid;
- some of vertices do not exist, that is, not checked when the pointers are passed to vertices; or
- the starting vertex is equal to the ending vertex, that is, it is impossible to create loops from a single vertex.

cvGraphAddEdgeByPtr

Adds edge to graph.

```
int cvGraphAddEdgeByPtr( CvGraph* graph, CvGraphVtx* startVtx, CvGraphVtx*
    endVtx, CvGraphEdge* edge, CvGraphEdge** insertedEdge=0 );
```

<i>graph</i>	Graph.
<i>startVtx</i>	Pointer to the starting vertex of the edge.
<i>endVtx</i>	Pointer to the ending vertex of the edge.
<i>edge</i>	Optional input parameter, initialization data for the edge. If not <code>NULL</code> , the parameter is copied starting from the 5 th 4-byte field.
<i>insertedEdge</i>	Optional output parameter to contain the address of the inserted edge within the edge set.

Discussion

The function [cvGraphAddEdgeByPtr](#) adds the edge to the graph given the starting and the ending vertices. The function returns the index of the inserted edge, which is the value of the second 4-byte field of the free cell.

The function reports an error if

- the edge that connects the vertices already exists; in this case graph orientation is taken into account;

- a pointer is `NULL` or indices are invalid;
- some of vertices do not exist, that is, not checked when the pointers are passed to vertices; or
- the starting vertex is equal to the ending vertex, that is, it is impossible to create loops from a single vertex.

cvGraphRemoveEdge

Removes edge from graph.

```
void cvGraphRemoveEdge( CvGraph* graph, int startIdx, int endIdx );
```

<i>graph</i>	Graph.
<i>startIdx</i>	Index of the starting vertex of the edge.
<i>endIdx</i>	Index of the ending vertex of the edge.

Discussion

The function [cvGraphRemoveEdge](#) removes the edge from the graph that connects given vertices. If the graph is oriented, the vertices must be passed in the appropriate order. The function reports an error if any of the vertices or edges between them do not exist.

cvGraphRemoveEdgeByPtr

Removes edge from graph.

```
void cvGraphRemoveEdgeByPtr( CvGraph* graph, CvGraphVtx* startVtx, CvGraphVtx*  
    endVtx );
```

<i>graph</i>	Graph.
<i>startVtx</i>	Pointer to the starting vertex of the edge.

endVtx Pointer to the ending vertex of the edge.

Discussion

The function [cvGraphRemoveEdgeByPtr](#) removes the edge from the graph that connects given vertices. If the graph is oriented, the vertices must be passed in the appropriate order. The function reports an error if any of the vertices or edges between them do not exist.

cvFindGraphEdge

Finds edge in graph.

```
CvGraphEdge* cvFindGraphEdge( CvGraph* graph, int startIdx, int endIdx );
```

graph Graph.
startIdx Index of the starting vertex of the edge.
endIdx Index of the ending vertex of the edge.

Discussion

The function [cvFindGraphEdge](#) finds the graph edge that connects given vertices. If the graph is oriented, the vertices must be passed in the appropriate order. Function returns NULL if any of the vertices or edges between them do not exist.

cvFindGraphEdgeByPtr

Finds edge in graph.

```
CvGraphEdge* cvGraphRemoveEdgeByPtr( CvGraph* graph, CvGraphVtx* startVtx,  
CvGraphVtx* endVtx );
```

graph Graph.
startVtx Pointer to the starting vertex of the edge.

endVtx Pointer to the ending vertex of the edge.

Discussion

The function [cvFindGraphEdgeByPtr](#) finds the graph edge that connects given vertices. If the graph is oriented, the vertices must be passed in the appropriate order. Function returns `NULL` if any of the vertices or edges between them do not exist.

cvGraphVtxDegree

Finds edge in graph.

```
int  cvGraphVtxDegree( CvGraph* graph, int vtxIdx );  
    graph              Graph.  
    vtx                Pointer to the graph vertex.
```

Discussion

The function [cvGraphVtxDegree](#) counts the edges incident to the graph vertex, both incoming and outgoing, and returns the result. 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 next edge after the edge incident to the vertex.

cvGraphVtxDegreeByPtr

Finds edge in graph.

```
int cvGraphVtxDegreeByPtr( CvGraph* graph, CvGraphVtx* vtx );
```

graph Graph.
vtx Pointer to the graph vertex.

Discussion

The function [cvGraphVtxDegreeByPtr](#) counts the edges incident to the graph vertex, both incoming and outgoing, and returns the result. 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 next edge after the edge incident to the vertex.

cvClearGraph

Clears graph.

```
void cvClearGraph( CvGraph* graph );
```

graph Graph.

Discussion

The function [cvClearGraph](#) removes all the vertices and edges from the graph. Similar to the function [cvClearSet](#), this function takes $O(1)$ time.

cvGetGraphVtx

Finds graph vertex by index.

```
CvGraphVtx* cvGetGraphVtx( CvGraph* graph, int vtxIdx );
```

<i>graph</i>	Graph.
<i>vtxIdx</i>	Index of the vertex.

Discussion

The function [cvGetGraphVtx](#) finds the graph vertex by index and returns the pointer to it or, if not found, to a free cell at this index. Negative indices are supported.

cvGraphVtxIdx

Returns index of graph vertex.

```
int cvGraphVtxIdx( CvGraph* graph, CvGraphVtx* vtx );
```

<i>graph</i>	Graph.
<i>vtx</i>	Pointer to the graph vertex.

Discussion

The function [cvGraphVtxIdx](#) returns the index of the graph vertex by setting pointers to it.

cvGraphEdgeIdx

Returns index of graph edge.

```
int cvGraphEdgeIdx( CvGraph* graph, CvGraphEdge* edge );
```

graph Graph.

edge Pointer to the graph edge.

Discussion

The function [cvGraphEdgeIdx](#) returns the index of the graph edge by setting pointers to it.

Contour Processing

3

This chapter describes contour processing functions.

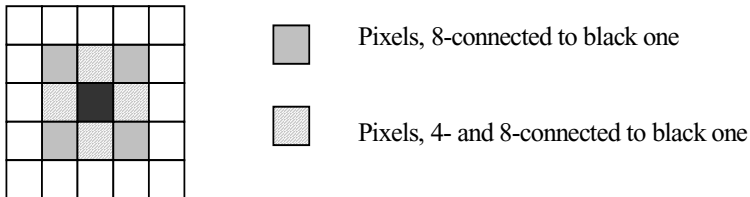
Overview

Below follow descriptions of:

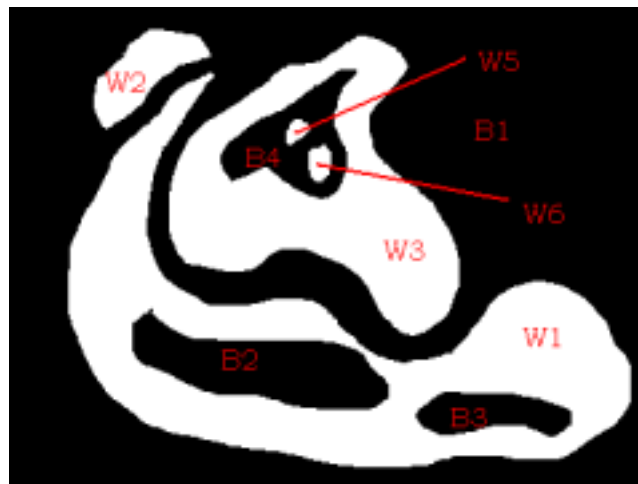
- several basic functions that retrieve contours from the binary image and store them in the chain format;
- functions for polygonal approximation of the chains.

Basic Definitions

Most of the existing *vectoring* algorithms, that is, algorithms that find contours on the raster images, deal with binary images. A binary image contains only *0-pixels*, that is, pixels with the value 0, and *1-pixels*, that is, pixels with the value 1. The set of *connected* 0- or 1-pixels makes the *0-(1-) component*. There are two common sorts of connectivity, the *4-connectivity* and *8-connectivity*. Two pixels with coordinates (x', y') and (x'', y'') are called *4-connected* if, and only if, $|x' - x''| + |y' - y''| = 1$ and *8-connected* if, and only if, $\max(|x' - x''|, |y' - y''|) = 1$. [Figure 3-1](#) shows these relations:

Figure 3-1 Pixels Connectivity Patterns

Using this relationship, the image is broken into several non-overlapped 1-(0-) 4-connected (8-connected) components. Each set consists of pixels with equal values, that is, all pixels are either equal to 1 or 0, and any pair of pixels from the set can be linked by a sequence of 4- or 8-connected pixels. In other words, a 4-(8-) path exists between any two points of the set. The components shown in [Figure 3-2](#) may have interrelations.

Figure 3-2 Hierarchical Connected Components

1-components $W1$, $W2$, and $W3$ are inside the *frame* (0-component $B1$), that is, *directly* surrounded by $B1$.

0-components $B2$ and $B3$ are inside $W1$.

1-components $W5$ and $W6$ are inside $B4$, that is inside $W3$, so these 1-components are inside $W3$ *indirectly*. However, neither $W5$ nor $W6$ enclose one another, which means they are on the same level.

In order to avoid a topological contradiction 0-pixels must be regarded as 8-(4-) connected pixels in case 1-pixels are dealt with as 4-(8-) connected. Throughout this document 8-connectivity is assumed to be used with 1-pixels and 4-connectivity with 0-pixels.

Since 0-components are complementary to 1-components, and separate 1-components are either nested to each other or their internals do not intersect, the library considers 1-components only and only their topological structure is studied, 0-pixels making up the background. A 0-component directly surrounded by a 1-component is called the *hole* of the 1-component. The *border point* of a 1-component could be any pixel that belongs to the component and has a 4-connected 0-pixel. A connected set of border points is called the *border*.

Each 1-component has a single *outer border* that separates it from the surrounding 0-component and zero or more *hole borders* that separate the 1-component from the 0-components it surrounds. It is obvious that the outer border and hole borders give a full description of the component. Therefore all the borders, also referred to as *contours*, of all components stored with information about the hierarchy make up a compressed representation of the source binary image. See Reference for description of the functions [cvFindContours](#), [cvStartFindContours](#), and [cvFindNextContour](#) that build such a contour representation of binary images.

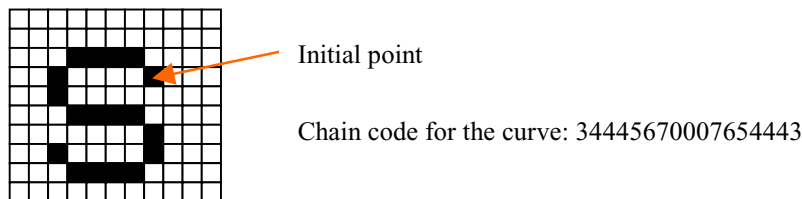
Contour Representation

The library uses two methods to represent contours. The first method is called the Freeman method or the chain code. For any pixel all its neighbors with numbers from 0 to 7 can be enumerated:

Figure 3-3 Contour Representation in Freeman Method

3	2	1
4		0
5	6	7

The 0-neighbor denotes the pixel on the right side, etc. As a sequence of 8-connected points, the border can be stored as the coordinates of the initial point, followed by codes (from 0 to 7) that specify the location of the next point relative to the current one (see [Figure 3-4](#)).

Figure 3-4 Freeman Coding of Connected Components

The chain code is a compact representation of digital curves and an output format of the contour retrieving algorithms described below.

Polygonal representation is a different option in which the curve is coded as a sequence of points, vertices of a polyline. This alternative is often a better choice for manipulating and analyzing contours over the chain codes; however, this representation is rather hard to get directly without much redundancy. Instead, algorithms that approximate the chain codes with polylines could be used.

Contour Retrieving Algorithm

Four variations of algorithms described in [Suzuki85] are used in the library to retrieve borders. The first algorithm finds only the extreme outer contours in the image and returns them linked to the list. [Figure 3-2](#) shows these external boundaries of $W1$, $W2$, and $W3$ domains. The second algorithm returns all contours linked to the list. [Figure 3-2](#) shows the total of 8 such contours. The third algorithm finds all connected components by building a two-level hierarchical structure: on the top are the external boundaries of 1-domains and every external boundary contains a link to the list of holes of the corresponding component. The third algorithm returns all the connected components as a two-level hierarchical structure: on the top are the external boundaries of 1-domains and every external boundary contour header contains a link to the list of holes in the corresponding component. The list can be accessed via `v_next` field of the external contour header.

[Figure 3-2](#) shows that $W2$, $W5$, and $W6$ domains have no holes; consequently, their boundary contour headers refer to empty lists of hole contours. $W1$ domain has two holes - the external boundary contour of $W1$ refers to a list of two hole contours. Finally, $W3$ external boundary contour refers to a list of the single hole contour.

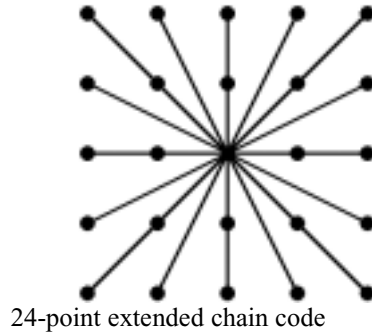
The fourth algorithm returns the complete hierarchical tree where all the contours contain a list of contours surrounded by the contour directly, that is, the hole contour of $W3$ domain has two children: external boundary contours of $W5$ and $W6$ domains.

All algorithms make a single pass through the image; there are, however, rare instances when some contours need to be scanned more than once. The algorithms do line-by-line scanning.

Whenever an algorithm finds a point that belongs to a new border the border following procedure is applied to retrieve and store the border in the chain format. During the border following procedure the algorithms mark the visited pixels with special positive or negative values. If the right neighbor of the considered border point is a 0-pixel and, at the same time, the 0-pixel is located in the right hand part of the border, the border point is marked with a negative value. Otherwise, the point is marked with the same magnitude but of positive value, if the point has not been visited yet. This can be easily determined since the border can cross itself or tangent other borders. The first and second algorithms mark all the contours with the same value and the third and fourth algorithms try to use a unique ID for each contour, which can be used to detect the parent of any newly met border.

Polygonal Approximation

As soon as all the borders have been retrieved from the image, the shape representation can be further compressed. Several algorithms are available for the purpose, including RLE coding of chain codes, higher order codes (see [Figure 3-5](#)), polygonal approximation, etc.

Figure 3-5 Higher Order Freeman Codes

Polygonal approximation is the best method in terms of the output data simplicity for further processing. Below follow descriptions of two polygonal approximation algorithms. The main idea behind them is to find and keep only the dominant points, that is, points where the local maximums of curvature absolute value are located on the digital curve, stored in the chain code or in another direct representation format. The first step here is the introduction of a discrete analog of curvature. In the continuous case curvature is determined as the speed of the tangent angle changing:

$$k = \frac{x'y'' - x''y'}{(x'^2 + y'^2)^{3/2}}.$$

In the discrete case different approximations are used. The simplest one, called L1 curvature, is the difference between successive chain codes:

$$c_i^{(1)} = ((f_i - f_{i-1} + 4) \bmod 8) - 4. \quad (3.1)$$

This method covers the changes from 0, that corresponds to the straight line, to 4, that corresponds to the sharpest angle, when the direction is changed to reverse.

The following algorithm is used for getting a more complex approximation. First, for the given point (x_i, y_i) the radius m_i of the neighborhood to be considered is selected. For some algorithms m_i is a method parameter and has a constant value for all points; for others it is calculated automatically for each point. The following value is calculated for all pairs (x_{i-k}, y_{i-k}) and (x_{i+k}, y_{i+k}) ($k=1 \dots m$):

$$c_{ik} = \frac{(a_{ik} \cdot b_{ik})}{|a_{ik}| |b_{ik}|} = \cos(a_{ik}, b_{ik}),$$

where $a_{ik} = (x_{i-k} - x_i, y_{i-k} - y_i)$, $b_{ik} = (x_{i+k} - x_i, y_{i+k} - y_i)$.

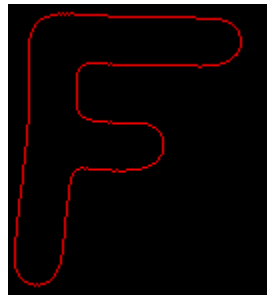
The next step is finding the index h_i such that $c_{im} < c_{im-1} < \dots < c_{ih_i} \geq c_{ih_i-1}$. The value c_{ih_i} is regarded as the curvature value of the i^{th} point. The point value changes from -1 (straight line) to 1 (sharpest angle). This approximation is called the k -cosine curvature.

Rosenfeld-Johnston algorithm [[Rosenfeld73](#)] is one of the earliest algorithms for determining the dominant points on the digital curves. The algorithm requires the parameter m , the neighborhood radius that is often equal to $1/10$ or $1/15$ of the number of points in the input curve. Rosenfeld-Johnston algorithm is used to calculate curvature values for all points and remove points that satisfy the condition

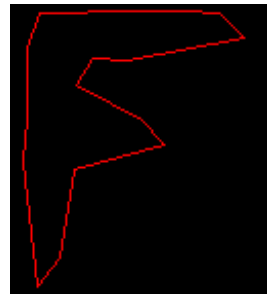
$$\exists j, |i-j| \leq h_i/2 \ ; \ c_{ih_i} < c_{jh_j} \ .$$

The remaining points are treated as dominant points. [Figure 3-6](#) shows an example of applying the algorithm.

Figure 3-6 Rosenfeld-Johnston Output for F-Letter Contour



Source Image



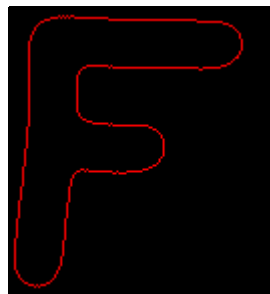
Rosenfeld-Johnston Algorithm Output

The disadvantage of the algorithm is the necessity to choose the parameter m and parameter identity for all the points, which results in either excessively rough, or excessively precise contour approximation.

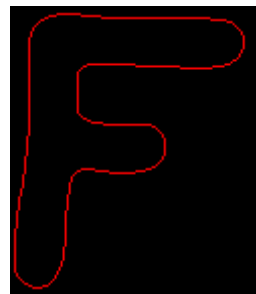
The next algorithm proposed by Teh and Chin [[Teh89](#)] includes a method for the automatic selection of the parameter m for each point. The algorithm makes several passes through the curve and deletes some points at each pass. At first all points with zero $c_i^{(1)}$ curvatures are deleted (see [Equation 3.1](#)). For other points the parameter m_i

and the curvature value are determined. After that the algorithm performs a non-maxima suppression, same as in Rosenfeld-Johnston algorithm, deleting points whose curvature satisfies the previous condition where for $c_i^{(1)}$ the metric h_i is set to m_i . Finally, the algorithm replaces groups of two successive remaining points with a single point and groups of three or more successive points with a pair of the first and the last points. This algorithm does not require any parameters except for the curvature to use. [Figure 3-7](#) shows the algorithm results.

Figure 3-7 Teh-Chin Output for F-Letter Contour



Source Picture



Teh-Chin Algorithm Output

Douglas-Peucker Approximation

Instead of applying a rather sophisticated Teh-Chin algorithm to the chain code, the user may try another way to get a smooth contour on a little number of vertices. The idea is to apply some very simple approximation techniques to the chain code with polylines, such as substituting ending points for horizontal, vertical, and diagonal segments, and then use the approximation algorithm on polylines. This preprocessing reduces the amount of data without any accuracy loss. Teh-Chin algorithm also involves this step, but uses removed points for calculating curvatures of the remaining points.

The algorithm to consider is a pure geometrical algorithm by Douglas-Peucker for approximating a polyline with another polyline with required accuracy:

1. Two points on the given polyline are selected, thus the polyline is approximated by the line connecting these two points. The algorithm iteratively adds new points to this initial approximation polyline until the required accuracy is achieved. If the polyline is not closed, two ending points are selected. Otherwise, some initial algorithm should be applied to find two initial points. The more extreme the points are, the better.
2. The algorithm iterates through all polyline vertices between the two initial vertices and finds the farthest point from the line connecting two initial vertices. If this maximum distance is less than the required error, then the approximation has been found and the next segment, if any, is taken for approximation. Otherwise, the new point is added to the approximation polyline and the approximated segment is split at this point. Then the two parts are approximated in the same way, since the algorithm is recursive. For a closed polygon there are two polygonal segments to process.

Contours Moments

The moment of order $(p; q)$ of an arbitrary region R is given by

$$v_{pq} = \iint_R x^p \cdot y^q dx dy \quad . \quad (3.2)$$

If $p = q = 0$, we obtain the area a of R . The moments are usually normalized by the area a of R . These moments are called normalized moments:

$$\alpha_{pq} = (1/a) \iint_R x^p \cdot y^q dx dy \quad . \quad (3.3)$$

Thus $\alpha_{00} = 1$. For $p + q \geq 2$ normalized central moments of R are usually the ones of interest:

$$\mu_{pq} = 1/a \iint_R (x - a_{10})^p \cdot (y - a_{01})^q dx dy \quad (3.4)$$

It is an explicit method for calculation of moments of arbitrary closed polygons. Contrary to most implementations that obtain moments from the discrete pixel data, this approach calculates moments by using only the border of a region. Since no

explicit region needs to be constructed, and because the border of a region usually consists of significantly fewer points than the entire region, the approach is very efficient. The well-known Green's formula is used to calculate moments:

$$\iint_R (\partial Q / \partial x - \partial P / \partial y) dx dy = \int_b (P dx + Q dy) ,$$

where b is the border of the region R .

It follows from the formula (3.2) that:

$$\partial Q / \partial x = x^p \cdot y^q, \partial P / \partial y = 0 ,$$

hence

$$P(x, y) = 0, Q(x, y) = 1/(p+1) \cdot x^{p+1} y^q .$$

Therefore, the moments from (3.2) can be calculated as follows:

$$v_{pq} = \int_b (1/(p+1) x^{p+1} \cdot y^q) dy . \quad (3.5)$$

If the border b consists of n points $p_i = (x_i, y_i)$, $0 \leq i \leq n$, $p_0 = p_n$, it follows that:

$$b(t) = \bigcup_{i=1}^n b_i(t) ,$$

where $b_i(t)$, $t \in [0,1]$ is defined as

$$b_i(t) = tp + (1-t)p_{i-1} .$$

Therefore, (3.5) can be calculated in the following manner:

$$v_{pq} = \sum_{i=1}^n \int_{b_i} (1/(p+1) x^{p+1} \cdot y^q) dy \quad (3.6)$$

After unnormalized moments have been transformed, (3.6) could be written as:

$$v_{pA} = \frac{1}{(p+q+2)(p+q+1) \binom{p+q}{p}} \times \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1}) \sum_{k=0}^p \sum_{t=0}^q \binom{k+t}{t} \binom{p+q-k-t}{q-t} x_i^k x_{i-1}^{p-k} y_i^t y_{i-1}^{q-t}$$

Central unnormalized and normalized moments up to order 3 look like

$$\begin{aligned} a &= 1/2 \sum_{i=1}^n x_{i-1}y_i - x_iy_{i-1} \quad , \\ a_{10} &= 1/(6a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(x_{i-1} + x_i) \quad , \\ a_{01} &= 1/(6a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(y_{i-1} + y_i) \quad , \\ a_{20} &= 1/(12a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(x_{i-1}^2 + x_{i-1}x_i + x_i^2) \quad , \\ a_{11} &= 1/(24a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(2x_{i-1} + x_{i-1}y_i + x_iy_{i-1} + 2x_iy_i) \quad , \\ a_{02} &= 1/(12a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(y_{i-1}^2 + y_{i-1}y_i + y_i^2) \quad , \\ a_{30} &= 1/(20a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(x_{i-1}^3 + x_{i-1}^2x_i + x_i^2x_{i-1} + x_i^3) \quad , \\ a_{21} &= 1/(60a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(x_{i-1}^2(3y_{i-1} + y_i) + 2x_{i-1}x_i(y_{i-1} + y_i) \\ &\quad + x_i^2(y_{i-1} + 3y_i)), \end{aligned}$$

$$a_{12} = 1/(60a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(y_{i-1}^2(3x_{i-1} + x_i) + 2y_{i-1}y_i(x_{i-1} + x_i) + y_i^2(x_{i-1} + 3x_i)),$$

$$a_{03} = 1/(20a) \sum_{i=1}^n (x_{i-1}y_i - x_iy_{i-1})(y_{i-1}^3 + y_{i-1}^2y_i + y_i^2y_{i-1} + y_i^3),$$

$$\mu_{20} = \alpha_{20} - \alpha_{10}^2,$$

$$\mu_{11} = \alpha_{11} - \alpha_{10}\alpha_{01},$$

$$\mu_{02} = \alpha_{02} - \alpha_{01}^2,$$

$$\mu_{30} = \alpha_{30} + 2\alpha_{10}^3 - 3\alpha_{10}\alpha_{20},$$

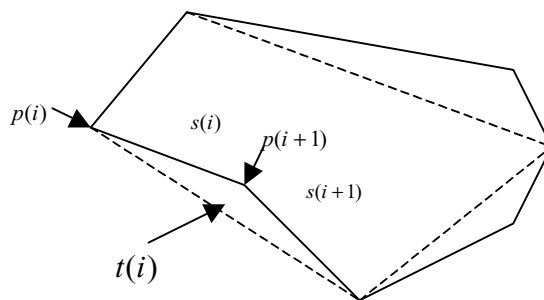
$$\mu_{21} = \alpha_{21} + 2\alpha_{10}^3\alpha_{01} - 2\alpha_{10}\alpha_{11} - \alpha_{20}\alpha_{01},$$

$$\mu_{12} = \alpha_{12} + 2\alpha_{01}^3\alpha_{10} - 2\alpha_{01}\alpha_{11} - \alpha_{02}\alpha_{10},$$

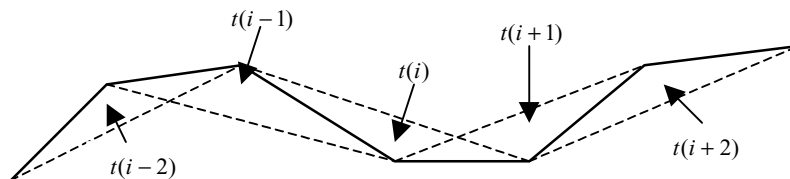
$$\mu_{03} = \alpha_{03} + 2\alpha_{01}^3 - 3\alpha_{01}\alpha_{02}.$$

Hierarchical Representation of Contours

Let T be the simple closed boundary of a shape with n points $T: \{p(1), p(2), \dots, p(n)\}$ and n runs: $\{s(1), s(2), \dots, s(n)\}$. Every run $s(i)$ is formed by the two points $(p(i), p(i+1))$. For every pair of the neighboring runs $s(i)$ and $s(i+1)$ a triangle is defined by the two runs and the line connecting the two far ends of the two runs ([Figure 3-8](#)).

Figure 3-8 Triangles Numbering

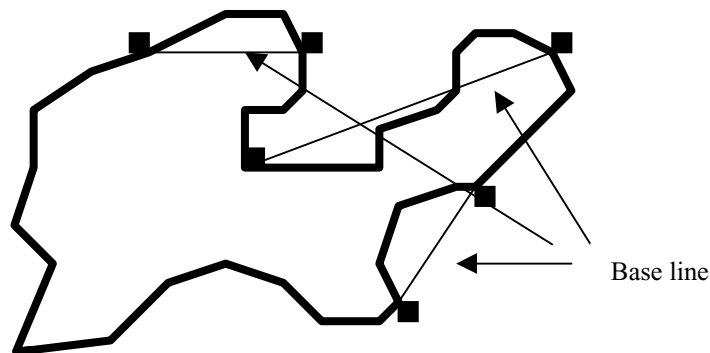
We call triangles $t(i-2)$, $t(i-1)$, $t(i+1)$, $t(i+2)$ the neighboring triangles of $t(i)$ ([Figure 3-9](#)).

Figure 3-9 Location of Neighboring Triangles

For every straight line that connects any two different vertices of a shape, the line either cuts off a region from the original shape or fills in a region of the original shape, or does both. The size of the region is called the interceptive area of that line ([Figure 3-10](#)). This line is called the base line of the triangle.

A triangle made of two boundary runs is the *locally minimum interceptive area triangle* (LMIAT) if the interceptive area of its base line is smaller than both its neighboring triangles areas.

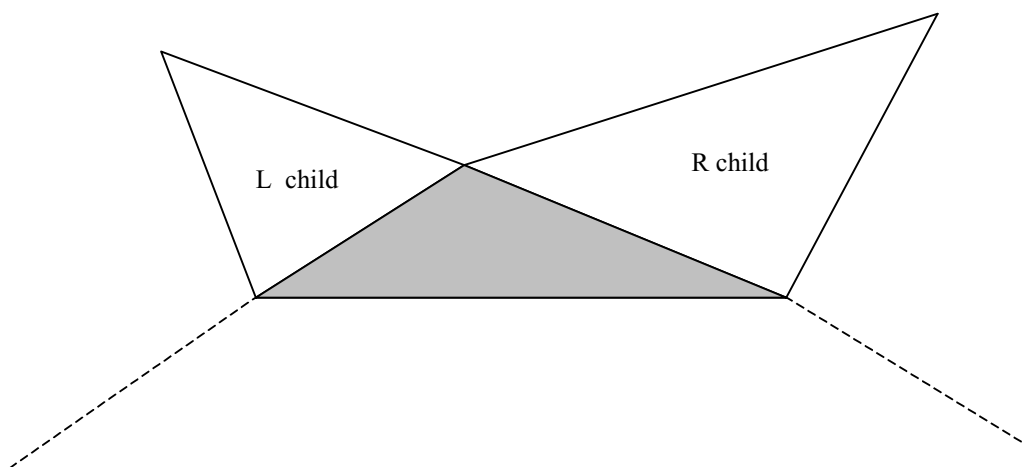
Figure 3-10 Interceptive Area



The shape-partitioning algorithm is multilevel. This procedure subsequently removes some points from the contour; the removed points become children nodes of the tree. On each iteration the procedure examines the triangles defined by all the pairs of the neighboring edges along the shape boundary and finds all LMIATs. After that all LMIATs whose areas are less than a reference value, which is the algorithm parameter, are removed. That actually means removing their middle points. If the user wants to get a precise representation, zero reference value could be passed. Other LMIATs are also removed, but the corresponding middle points are stored in the tree. After that another iteration is run. This process ends when the shape has been simplified to a quadrangle. The algorithm then determines a diagonal line that divides this quadrangle into two triangles in the most unbalanced way.

Thus the binary tree representation is constructed from the bottom to top levels. Every tree node is associated with one triangle. Except the root node, every node is connected to its parent node, and every node may have none, or single, or two child nodes. Each newly generated node becomes the parent of the nodes for which the two sides of the new node form the base line. The triangle that uses the left side of the parent triangle is the left child. The triangle that uses the right side of the parent triangle is the right child (See [Figure 3-11](#)).

Figure 3-11 Classification of Child Triangles



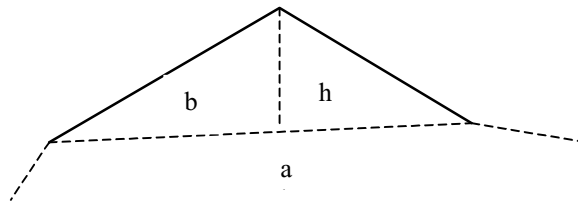
The root node is associated with the diagonal line of the quadrangle. This diagonal line divides the quadrangle into two triangles. The larger triangle is the left child and the smaller triangle is its right child.

For any tree node we record the following attributes:

- Coordinates x and y of the vertex P that do not lie on the base line of LMIAT, that is, coordinates of the middle (removed) point;
- Area of the triangle;
- Ratio of the height of the triangle h to the length of the base line a ([Figure 3-12](#));

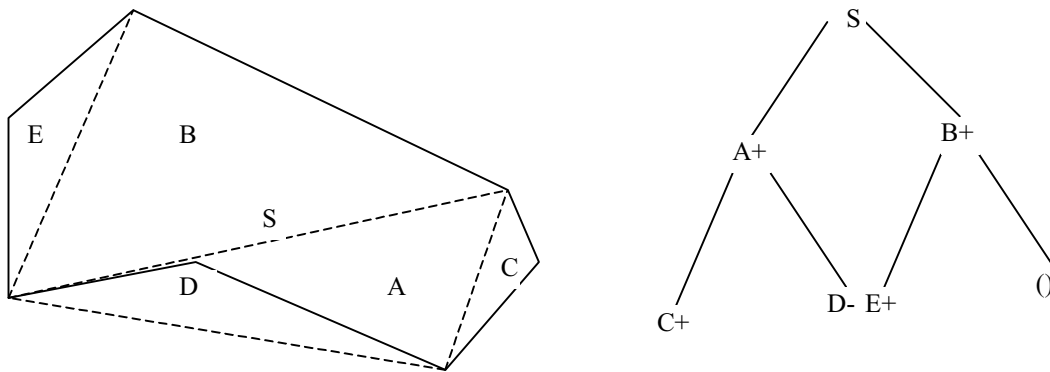
- Ratio of the projection of the left side of the triangle on the base line b to the length of the base line a ;
- Signs “+” or “-”; the sign “+” indicates that the triangle lies outside of the new shape due to the ‘cut’ type merge; the sign “-” indicates that the triangle lies inside the new shape.

Figure 3-12 Triangles Properties



[Figure 3-13](#) shows an example of the shape partitioning.

Figure 3-13 Shape Partitioning



It is necessary to note that only the first attribute is sufficient for source contour reconstruction; all other attributes may be calculated from it. However, the other four attributes are very helpful for efficient contour matching.

The shape matching process that compares two shapes to determine whether they are similar or not can be effected by matching two corresponding tree representations, e.g., two trees can be compared from top to bottom, node by node, using the breadth-first traversing procedure.

Let us define the *corresponding node pair* (CNP) of two binary tree representations TA and TB . The corresponding node pair is called $[A(i), B(i)]$, if $A(i)$ and $B(i)$ are at the same level and same position in their respective trees.

The next step is defining the *node weight*. The weight of $N(i)$ denoted as $W[N(i)]$ is defined as the ratio of the size of $N(i)$ to the size of the entire shape.

Let $N(i)$ and $N(j)$ be two nodes with heights $h(i)$ and $h(j)$ and base lengths $a(i)$ and $a(j)$ respectively. The projections of their left sides on their base lines are $b(i)$ and $b(j)$ respectively. The node distance $dn[N(i), N(j)]$ between $N(i)$ and $N(j)$ is defined as:

$$dn[N(i), N(j)] = |h(i)/a(i) \cdot W[N(i)] \mp h(j)/a(j) \cdot W[N(j)]| \\ + |b(i)/a(i) \cdot W[N(i)] \mp b(j)/a(j) \cdot W[N(j)]|$$

In the above equation, the “+” signs are used when the signs of attributes in two nodes are different and the “-” signs are used when the two nodes have the same sign.

For two trees TA and TB representing two shapes SA and SB and with the corresponding node pairs $[A(1), B(1)], [A(2), B(2)], \dots, [A(n), B(n)]$ the tree distance $dt(TA, TB)$ between TA and TB is defined as:

$$dt(TA, TB) = \sum_{i=1}^k dn[A(i), B(i)].$$

If the two trees are different in size, the smaller tree is enlarged with trivial nodes so that the two trees can be fully compared. A trivial node is a node whose size attribute is zero. Thus, the trivial node weight is also zero. The values of other node attributes are trivial and not used in matching. The sum of the node distances of the first k CNPs of TA and TB is called the cumulative tree distance $dt(TA, TB, k)$ and is defined as:

$$dc(TA, TB, k) = \sum_{i=1}^{\kappa} dn[A(i), B(i)] .$$

Cumulative tree distance shows the dissimilarity between the approximations of the two shapes and exhibits the multiresolution nature of the tree representation in shape matching.

The shape matching algorithm is quite straightforward. For two given tree representations the two trees are traversed according to the breadth-first sequence to find CNPs of the two trees. Next $dn[A(i), B(i)]$ and $dc(TA, TB, i)$ are calculated for every i . If for some i $dc(TA, TB, i)$ is larger than the tolerance threshold value, the matching procedure is terminated to indicate that the two shapes are dissimilar, otherwise it continues. If $dc(TA, TB)$ is still less than the tolerance threshold value, then the procedure is terminated to indicate that there is a good match between TA and TB .

Data Structures

The Computer Vision Library functions use special data structures to represent the contours and contour binary tree in memory, namely the structures `CvSeq` and `CvContourTree`. Below follows the definition of the structure `CvContourTree` in the C language.

Example 3-1 `CvContourTree` Structure Definition

```
typedef struct CvContourTree
{ CV_SEQUENCE_FIELDS( )
  CvPoint p1;           /*the start point of the binary tree
                        root*/
  CvPoint p2;           /*the end point of the binary tree
                        root*/
} CvContourTree;
```

Reference

cvFindContours

Finds contours in binary image.

```
int cvFindContours(IplImage* img, CvMemStorage* storage, CvSeq**
    firstContour, int headerSize=sizeof(CvContour),
    CvContourRetrievalMode mode=CV_RETR_LIST, CvChainApproxMethod
    method=CV_CHAIN_APPROX_SIMPLE);
```

<i>img</i>	Single channel image of <code>IPL_DEPTH_8U</code> type. Non-zero pixels are treated as 1-pixels. The function modifies the content of the input parameter.
<i>storage</i>	Contour storage location.
<i>firstContour</i>	Output parameter. Pointer to the first contour on the highest level.
<i>headerSize</i>	Size of the sequence header; must be equal to or greater than <code>sizeof(CvChain)</code> when the method <code>CV_CHAIN_CODE</code> is used, and equal to or greater than <code>sizeof(CvContour)</code> otherwise.
<i>mode</i>	Retrieval mode. <ul style="list-style-type: none"> • <code>CV_RETR_EXTERNAL</code> retrieves only the extreme outer contours (list); • <code>CV_RETR_LIST</code> retrieves all the contours (list); • <code>CV_RETR_CCOMP</code> retrieves the two-level hierarchy (list of connected components); • <code>CV_RETR_TREE</code> retrieves the complete hierarchy (tree).
<i>method</i>	Approximation method. <ul style="list-style-type: none"> • <code>CV_CHAIN_CODE</code> outputs contours in the Freeman chain code.

- `CV_CHAIN_APPROX_NONE` translates all the points from the chain code into points;
- `CV_CHAIN_APPROX_SIMPLE` compresses horizontal, vertical, and diagonal segments, that is, it leaves only their ending points;
- `CV_CHAIN_APPROX_TC89_L1`, `CV_CHAIN_APPROX_TC89_KCOS` are two versions of the Teh-Chin approximation algorithm.

Discussion

The function [cvFindContours](#) retrieves contours from the binary image and returns the pointer to the first contour. Other contours may be accessed through the `h_next` and `v_next` fields of the returned structure. The function returns total number of retrieved contours.

cvStartFindContours

Initializes contour scanning process.

```
CvContourScanner cvStartFindContours(IplImage* img, CvMemStorage* storage, int
    headerSize, CvContourRetrievalMode mode, CvChainApproxMethod method );
```

<i>img</i>	Single channel image of <code>IPL_DEPTH_8U</code> type. Non-zero pixels are treated as 1-pixels. The function damages the image.
<i>storage</i>	Contour storage location.
<i>headerSize</i>	Must be equal to or greater than <code>sizeof(CvChain)</code> when the method <code>CV_CHAIN_CODE</code> is used, and equal to or greater than <code>sizeof(CvContour)</code> otherwise.
<i>mode</i>	Retrieval mode. <ul style="list-style-type: none"> • <code>CV_RETR_EXTERNAL</code> retrieves only the extreme outer contours (list);

method

- `CV_RETR_LIST` retrieves all the contours (list);
- `CV_RETR_CCOMP` retrieves the two-level hierarchy (list of connected components);
- `CV_RETR_TREE` retrieves the complete hierarchy (tree).

Approximation method.

- `CV_CHAIN_CODE` codes the output contours in the chain code;
- `CV_CHAIN_APPROX_NONE` translates all the points from the chain code into points;
- `CV_CHAIN_APPROX_SIMPLE` substitutes ending points for horizontal, vertical, and diagonal segments;
- `CV_CHAIN_APPROX_TC89_L1`,
`CV_CHAIN_APPROX_TC89_KCOS` are two versions of the Teh-Chin approximation algorithm.

Discussion

The function [cvStartFindContours](#) initializes the contour scanner and returns the pointer to it. The structure is internal and no description is provided.

cvFindNextContour*Finds next contour on raster.*

```
CvSeq* cvFindNextContour( CvContourScanner scanner );
```

scanner Contour scanner initialized by the function `cvStartFindContours`.

Discussion

The function [cvFindNextContour](#) returns the next contour or 0, if the image contains no other contours.

cvSubstituteContour

Replaces retrieved contour.

```
void cvSubstituteContour( CvContourScanner scanner, CvSeq* newContour );
```

scanner Contour scanner initialized by the function `cvStartFindContours`.

newContour Substituting contour.

Discussion

The function [cvSubstituteContour](#) replaces the retrieved contour, that was returned from the preceding call of the function [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 *newContour* is 0, the retrieved contour is not included into the resulting structure, nor all of its children that might be added to this structure later.

cvEndFindContours

Finishes scanning process.

```
CvSeq* cvEndFindContours( CvContourScanner* scanner );
```

scanner Pointer to the contour scanner.

Discussion

The function [cvEndFindContours](#) finishes the scanning process and returns the pointer to the first contour on the highest level.

cvApproxChains

Approximates Freeman chain(s) with polygonal curve.

```
CvSeq* cvApproxChains( CvSeq* srcSeq, CvMemStorage* storage,
    CvChainApproxMethod method=CV_CHAIN_APPROX_SIMPLE,
    float parameter=0, int minimalPerimeter=0,
    int recursive=0 );
```

<i>srcSeq</i>	Pointer to the chain that can refer to other chains.
<i>storage</i>	Storage location for the resulting polylines.
<i>method</i>	Approximation method (see the description of the function cvFindContours).
<i>parameter</i>	Method parameter (not used now).
<i>minimalPerimeter</i>	Approximates only those contours whose perimeters are not less than <i>minimalPerimeter</i> . Other chains are removed from the resulting structure.
<i>recursive</i>	If not 0, the function approximates all chains that can be accessed from <i>srcSeq</i> by <i>h_next</i> or <i>v_next</i> links. If 0, the single chain is approximated.

Discussion

This is a stand-alone approximation routine. The function [cvApproxChains](#) works exactly in the same way as the functions [cvFindContours](#) / [cvFindNextContour](#) with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other contours, if any, can be accessed via *v_next* or *h_next* fields of the returned structure.

cvStartReadChainPoints

Initializes chain reader.

```
void cvStartReadChainPoints( CvChain* chain, CvChainPtReader* reader );
```

<i>chain</i>	Pointer to chain.
<i>reader</i>	Chain reader state.

Discussion

The function [cvStartReadChainPoints](#) initializes the special reader (see [Dynamic Data Structures](#) for more information on sets and sequences).

cvReadChainPoint

Gets next chain point.

```
CvPoint cvReadChainPoint( CvChainPtReader* reader );
```

<i>reader</i>	Chain reader state.
---------------	---------------------

Discussion

The function [cvReadChainPoint](#) returns the current chain point and moves to the next point.

cvApproxPoly

Approximates polygonal contour(s) with desired precision.

```
CvSeq* cvApproxPoly( CvSeq* srcSeq, int headerSize, CvMemStorage* storage,
    CvPolyApproxMethod method, float parameter, int recursive=0 );
```

<i>srcSeq</i>	Pointer to the contour that can refer to other chains.
<i>headerSize</i>	Size of the header for resulting sequences.
<i>storage</i>	Resulting contour storage location.
<i>method</i>	Approximation method; only CV_POLY_APPROX_DP is supported, that corresponds to Douglas-Peucker method.
<i>parameter</i>	Method-specific parameter; a desired precision for CV_POLY_APPROX_DP.
<i>recursive</i>	If not 0, the function approximates all contours that can be accessed from <i>srcSeq</i> by <i>h_next</i> or <i>v_next</i> links. If 0, the single contour is approximated.

Discussion

The function [cvApproxPoly](#) approximates one or more contours and returns pointer to the first resultant contour. Other contours, if any, can be accessed via *v_next* or *h_next* fields of the returned structure.

cvDrawContours

Draws contours in image.

```
void cvDrawContours( IplImage *img, CvSeq* contour, int externalColor, int
holeColor, int maxLevel, int thickness=1 );
```

<i>img</i>	Image where the contours will be drawn. Like in any other drawing function, every output is clipped with the ROI.
<i>contour</i>	Pointer to the first contour.
<i>externalColor</i>	Color to draw external contours with.
<i>holeColor</i>	Color to draw holes with.
<i>maxLevel</i>	Maximal level for drawn contours. If 0, only the contour is drawn. If 1, the contour and all contours after it on the same level are drawn. If 2, all contours after and all contours one level below the contours are drawn, etc.
<i>thickness</i>	Thickness of lines the contours are drawn with.

Discussion

The function [cvDrawContours](#) draws contour outlines in the image if the thickness is positive or zero or fills area bounded by the contours if thickness is negative, e.g. if `thickness==CV_FILLED`.

cvContoursMoments

Calculates contour moments up to order 3.

```
void cvContoursMoments(CvSeq* contour, CvMoments* moments);
```

<i>contour</i>	Pointer to the input contour header.
<i>moments</i>	Pointer to the output structure of contour moments; must be allocated by the caller.

Discussion

The function [cvContoursMoments](#) calculates unnormalized spatial and central moments of the contour up to order 3.

cvContourArea

Calculates region area inside contour or contour section.

```
double cvContourArea(CvSeq* contour, CvSlice slice=CV_WHOLE_SEQ(seq));
```

contour Pointer to the input contour header.

slice Starting and ending points of the contour section of interest.

Discussion

The function [cvContourArea](#) calculates the region area within the contour consisting of n points $p_i = (x_i, y_i)$, $0 \leq i \leq n$, $p_0 = p_n$, as a spatial moment:

$$\alpha_{00} = 1/2 \sum_{i=1}^n x_{i-1}y_i - x_iy_{i-1} \quad .$$

If a part of the contour is selected and the chord, connecting ending points, intersects the contour in several places, then the sum of all subsection areas is calculated. If the input contour has points of self-intersection, the region area within the contour may be calculated incorrectly.

cvMatchContours

Matches two contours.

```
double cvMatchContours (CvSeq *contour1, CvSeq* contour2, int method, long
parameter=0 );
```

contour1 Pointer to the first input contour header.

contour2 Pointer to the second input contour header.

parameter Method-specific parameter, currently ignored.

<i>method</i>	Method for the similarity measure calculation; must be any of
	<ul style="list-style-type: none"> • CV_CONTOURS_MATCH_I1; • CV_CONTOURS_MATCH_I2; • CV_CONTOURS_MATCH_I3.

Discussion

The function [cvMatchContours](#) calculates one of the three similarity measures between two contours.

Let two closed contours A and B have n and m points respectively:

$A = \{(x_i, y_i), 1 \leq i \leq n\}$ $B = \{(u_i, v_i), 1 \leq i \leq m\}$. Normalized central moments of a contour may be denoted as η_{pq} , $0 \leq p + q \leq 3$. M. Hu has shown that a set of the next seven features derived from the second and third moments of contours is an invariant to translation, rotation, and scale change [[Hu62](#)].

$$h_1 = \eta_{20} + \eta_{02},$$

$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2,$$

$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2,$$

$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2,$$

$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\ + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2],$$

$$h_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}),$$

$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\ + -(\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2].$$

From these seven invariant features the three similarity measures I_1 , I_2 , and I_3 may be calculated:

$$I_1(A, B) = \sum_{i=1}^7 \left| -1/m_i^A + 1/m_i^B \right|,$$

$$I_2(A, B) = \sum_{i=1}^7 \left| -m_i^A + m_i^B \right|,$$

$$I_3(A, B) = \max_i |(m_i^A - m_i^B) / m_i^A| ,$$

$$\text{where } m_i^A = \text{sgn}(h_i^A) \log_{10} |h_i^A| , \quad m_i^B = \text{sgn}(h_i^B) \log_{10} |h_i^B| .$$

cvCreateContourTree

Creates binary tree representation for input contour.

```
CvContourTree* cvCreateContourTree(CvSeq *contour, CvMemStorage* storage,
    double threshold);
```

contour Pointer to the input contour header.

storage Pointer to the storage block.

threshold Value of the threshold.

Discussion

The function [cvCreateContourTree](#) creates binary tree representation for the input contour *contour* and returns the pointer to its root. If the parameter *threshold* is less than or equal to 0, the function creates full binary tree representation. If the threshold is more than 0, the function creates 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 created tree.

cvContourFromContourTree

Restores contour from binary tree representation.

```
CvSeq* cvContourFromContourTree (CvContourTree *tree, CvMemStorage* storage,
    CvTermCriteria criteria);
```

<i>tree</i>	Pointer to the input tree.
<i>storage</i>	Pointer to the storage block.
<i>criterion</i>	Criteria for the definition of the threshold value for contour reconstruction (level of precision).

Discussion

The function [cvContourFromContourTree](#) restores the contour from its binary tree representation. The parameter *criterion* defines the threshold, that is, level of precision for the contour restoring. If *criterion.type* = CV_TERMCRIT_ITER, the function restores *criterion.maxIter* tree levels. If *criterion.type* = CV_TERMCRIT_EPS, the function restores the contour as long as *tri_area* > *criterion.epsilon* * *contour_area*, where *contour_area* is the magnitude of the contour area and *tri_area* is the magnitude of the current triangle area. If *criterion.type* = CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, the function restores the contour as long as one of these conditions is true. The function returns reconstructed contour.

cvMatchContourTrees

Compares two binary tree representations.

```
double cvMatchContourTrees (CvContourTree *tree1, CvContourTree *tree2,
    CvTreeMatchMethod method, double threshold);
```

<i>tree1</i>	Pointer to the first input tree.
<i>tree2</i>	Pointer to the second input tree.
<i>method</i>	Method for calculation of the similarity measure; now must be only CV_CONTOUR_TREES_MATCH_I1.
<i>threshold</i>	Value of the compared threshold.

Discussion

The function [cvMatchContourTrees](#) 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 the total calculating value of the similarity for levels from 0 to the specified one is more than the parameter *threshold*, the function stops calculations and value of the total similarity measure is returned as *result*. If the total calculating value of the similarity for levels from 0 to the specified one is less than or equal to *threshold*, the function continues calculation on the next tree level and returns the value of the total similarity measure for the binary trees.

This chapter describes functions from computational geometry field.

Overview

Ellipse Fitting

Fitting of primitive models to the image data is a basic task in pattern recognition and computer vision. A successful solution of this task results in reduction and simplification of the data for the benefit of higher level processing stages. One of the most commonly used models is the ellipse which, being a perspective projection of the circle, is of great importance for many industrial applications.

The representation of general conic by the second order polynomial is $F(\vec{a}, \vec{x}) = \vec{a}^T \vec{x} = ax^2 + bxy + cy^2 + dx + ey + f = 0$ with the vectors denoted as $\vec{a} = [a, b, c, d, e, f]^T$ and $\vec{x} = [x^2, xy, y^2, x, y, 1]^T$.

$F(\vec{a}, \vec{x})$ is called the “algebraic distance between point (x_0, y_0) and conic $F(a, x)$ ”.

Minimizing the sum of squared algebraic distances $\sum_{i=1}^n F(\vec{x}_i)^2$ may approach the fitting of conic.

In order to achieve ellipse-specific fitting polynomial coefficients must be constrained. For ellipse they must satisfy $b^2 - 4ac < 0$.

Moreover, the equality constraint $4ac - b^2 = 1$ can be imposed in order to incorporate coefficients scaling into constraint.

This constraint may be written as a matrix $\vec{a}^T C \vec{a} = 1$.

Finally, the problem could be formulated as minimizing $\|D\vec{a}\|^2$ with constraint $\vec{a}^T C \vec{a} = 1$, where D is the $n \times 6$ matrix $[\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n]^T$.

Introducing the Lagrange multiplier results in the system

$$2D^T D \vec{a} - 2\lambda C \vec{a} = 0 \quad , \text{ which can be re-written as } \\ \vec{a}^T C \vec{a} = 1$$

$$S \vec{a} = 2\lambda C \vec{a} \\ \vec{a}^T C \vec{a} = 1,$$

The system solution is described in [[Fitzgibbon95](#)].

After the system is solved, ellipse center and axis can be extracted.

Line Fitting

M-estimators are used for approximating a set of points with geometrical primitives e.g., conic section, in cases when the classical least squares method fails. For example, the image of a line from the camera contains noisy data with many outliers, that is, the points that lie far from the main group, and the least squares method fails if applied.

The least squares method searches for a parameter set that minimizes the sum of squared distances:

$$m = \sum_i d_i^2,$$

where d_i is the distance from the i^{th} point to the primitive. The distance type is specified as the function input parameter. If even a few points have a large d_i , then the perturbation in the primitive parameter values may be prohibitively big. The solution is to minimize

$$m = \sum_i \rho(d_i),$$

where $\rho(d_i)$ grows slower than d_i^2 . This problem can be reduced to *weighted least squares* [[Fitzgibbon95](#)], which is solved by iterative finding of the minimum of

$$m_k = \sum W(d_i^{k-1}) d_i^2,$$

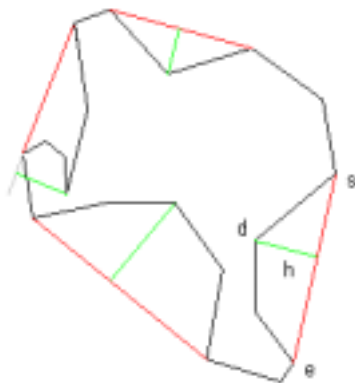
where k is the iteration number, d_i^{k-1} is the minimizer of the sum on the previous iteration, and $W(x) = \frac{1}{x} \frac{d\rho}{dx}$. If d_i is a linear function of parameters $p_j - d_i = \sum A_{ij} p_j$ then the minimization vector of the m_k is the eigenvector of $A^{T*} A$ matrix that corresponds to the smallest eigenvalue.

For more information see [[Zhang96](#)].

Convexity Defects

Let (p_1, p_2, \dots, p_n) be a closed simple polygon, or contour, and (h_1, h_2, \dots, h_m) a convex hull. A sequence of contour points exists normally between two consecutive convex hull vertices. This sequence forms the so-called convexity defect for which some useful characteristics can be computed. Computer Vision Library computes only one such characteristic, named “depth” (see [Figure 4-1](#)).

Figure 4-1 Convexity Defects



The black lines belong to the input contour. The red lines update the contour to its convex hull.

The symbols “s” and “e” signify the start and the end points of the convexity defect. The symbol “d” is a contour point located between “s” and “e” being the farthestmost from the line that includes the segment “se”. The symbol “h” stands for the convexity defect depth, that is, the distance from “d” to the “se” line.

The structure `CvConvexityDefect` represents the convexity defect.

Example 4-1 `CvConvexityDefect` structure definition

```
typedef struct
{
    CvPoint* start;           //start point of defect
    CvPoint* end;             //end point of defect
    CvPoint* depth_point;     //fathermost point
    float    depth;           //depth of defect
} CvConvexityDefect;
```

Reference

`cvFitEllipse`

Fits ellipse to set of 2D points.

```
void cvFitEllipse( CvPoint* points, int n, CvBox2D32f* box );
```

<code>points</code>	Pointer to the set of 2D points.
<code>n</code>	Number of points; must be more than or equal to 6.
<code>box</code>	Pointer to the structure for representation of the output ellipse.

Discussion

The function [`cvFitEllipse`](#) fills the output structure in the following way:

<code>box->center</code>	Point of the center of the ellipse;
<code>box->size</code>	Sizes of two ellipse axes;
<code>box->angle</code>	Angle between the horizontal axis and the ellipse axis with the length of <code>box->size.width</code> .

The output ellipse has the property of `box->size.width > box->size.height`.

cvFitLine2D

Fits 2D line to set of points on the plane.

```
void cvFitLine2D ( CvPoint2D32f* points, int count, CvDistType distType, void*
    param, float reps, float aeps float* line);
```

<i>points</i>	Array of 2D points.
<i>count</i>	Number of points.
<i>distType</i>	Type of the distance used to fit the data to a line.
<i>param</i>	Pointer to a user-defined function that calculates the weights for the type <code>CV_DIST_USER</code> , or the pointer to a float user-defined metric parameter <i>c</i> for the Fair and Welsch distance types.
<i>reps, aeps</i>	Used for iteration stop criteria. If zero, the default value of 0.01 is used.
<i>line</i>	Pointer to the array of 4 floats. When the function exits, the first two elements contain the direction vector of the line normalized to 1, the other two contain coordinates of a point that belongs to the line.

Discussion

The function [cvFitLine2D](#) fits 2D line to a set of points on the plane. Possible distance type values are listed below.

<code>CV_DIST_L2</code>	Standard least squares $\rho(x) = x^2$.
<code>CV_DIST_L1</code>	
<code>CV_DIST_L12</code>	
<code>CV_DIST_FAIR</code>	$c = 1.3998$.
<code>CV_DIST_WELSCH</code>	$\rho(x) = \frac{c^2}{2} \left[1 - \exp\left(-\left(\frac{x}{c}\right)^2\right) \right]$, $c = 2.9846$.
<code>CV_DIST_USER</code>	Uses a user-defined function to calculate the weight. The parameter <i>param</i> should point to the function.

The line equation is $[\vec{V} \times (\vec{r} - \vec{r}_0)] = 0$, where $\vec{V} = (line[0], line[1], line[2])$, $\vec{V} = 1$ and $\vec{r}_0 = (line[3], line[4], line[5])$.

In this algorithm \vec{r}_0 is the mean of the input vectors with weights, that is,

$$\vec{r}_0 = \frac{\sum_i W(d(\vec{r}_i)) \vec{r}_i}{\sum_i W(d(\vec{r}_i))}.$$

The parameters *reps* and *aeps* are iteration thresholds. If the distance of the type CV_DIST_C between two values of \vec{r}_0 calculated from two iterations is less than the value of the parameter *reps* and the angle in radians between two vectors \vec{V} is less than the parameter *aeps*, then the iteration is stopped.

The specification for the user-defined weight function is

```
void userWeight ( float* dist, int count, float* w );
```

dist Pointer to the array of distance values.

count Number of elements.

w Pointer to the output array of weights.

The function should fill the weights array with values of weights calculated from the distance values $w[i] = f(d[i])$. The function $f(x) = \frac{1}{x} \frac{dp}{dx}$ has to be monotone decreasing.

cvFitLine3D

Fits 3D line to set of points in 3D space.

```
void cvFitLine3D ( CvPoint3D32f* points, int count, CvDisType distType, void*
param, float reps, float aeps float* line);
```

points Array of 3D points.

count Number of points.

distType Type of the distance used to fit the data to a line.

param Pointer to a user-defined function that calculates the weights for the type CV_DIST_USER or the pointer to a float user-defined metric parameter *c* for the Fair and Welsch distance types.

<i>reps, aeps</i>	Used for iteration stop criteria. If zero, the default value of 0.01 is used.
<i>line</i>	Pointer to the array of 6 floats. When the function exits, the first three elements contain the direction vector of the line normalized to 1, the other three contain coordinates of a point that belongs to the line.

Discussion

The function `cvFitLine3D` fits 2D line to set of points on the plane. Possible distance type values are listed below.

CV_DIST_L2	Standard least squares $\rho(x) = x^2$.
CV_DIST_L1	
CV_DIST_L12	
CV_DIST_FAIR	$c = 1.3998$.
CV_DIST_WELSCH	$\rho(x) = \frac{c^2}{2} \left[1 - \exp\left(-\left(\frac{x}{c}\right)^2\right) \right]$, $c = 2.9846$.
CV_DIST_USER	Uses a user-defined function to calculate the weight. The parameter <i>param</i> should point to the function.

The line equation is $[\vec{V} \times (\vec{r} - \vec{r}_0)] = 0$, where $\vec{V} = (line[0], line[1], line[2])$, $\vec{V} = 1$ and $\vec{r}_0 = (line[3], line[4], line[5])$.

In this algorithm \vec{r}_0 is the mean of the input vectors with weights, that is,

$$\vec{r}_0 = \frac{\sum_i W(d(\vec{r}_i)) \vec{r}_i}{\sum_i W(d(\vec{r}_i))}.$$

The parameters *reps* and *aeps* are iteration thresholds. If the distance between two values of \vec{r}_0 calculated from two iterations is less than the value of the parameter *reps*, (the distance type CV_DIST_C is used in this case) and the angle in radians between two vectors \vec{V} is less than the parameter *aeps*, then the iteration is stopped.

The specification for the user-defined weight function is

```
void userWeight ( float* dist, int count, float* w );
```

dist Pointer to the array of distance values.

count Number of elements.
w Pointer to the output array of weights.

The function should fill the weights array with values of weights calculated from distance values $w[i] = f(d[i])$. The function $f(x) = \frac{1}{x} \frac{dp}{dx}$ has to be monotone decreasing.

cvProject3D

Projects array of 3D points to coordinate axis.

```
void cvProject3D ( CvPoint3D32f*  points3D, int count, CvPoint2D32f* points2D,  
                  int xindx, int yindx);
```

points3D Source array of 3D points.
count Number of points.
points2D Target array of 2D points.
xindx Index of the 3D coordinate from 0 to 2 that is to be used as
 x-coordinate.
yindx Index of the 3D coordinate from 0 to 2 that is to be used as
 y-coordinate.

Discussion

The function [cvProject3D](#) used with the function [cvmPerspectiveProject](#) is intended to provide a general way of projecting a set of 3D points to a 2D plane. The function copies two of the three coordinates specified by the parameters *xindx* and *yindx* of each 3D point to a 2D points array.

cvConvexHull

Finds convex hull of points set.

```
void cvConvexHull( CvPoint* points, int numPoints, CvRect* boundRect, int
orientation, int* hull, int* hullsize );
```

<i>points</i>	Pointer to the set of 2D points.
<i>numPoints</i>	Number of points.
<i>boundRect</i>	Pointer to the bounding rectangle of points set; not used.
<i>orientation</i>	Output order of the convex hull vertices CV_CLOCKWISE or CV_COUNTER_CLOCKWISE.
<i>hull</i>	Indices of convex hull vertices in the input array.
<i>hullsize</i>	Number of vertices in convex hull; output parameter.

Discussion

The function [cvConvexHull](#) takes an array of points and puts out indices of points that are convex hull vertices. The function uses Quicksort algorithm for points sorting.

The standard, that is, bottom-left *xy* coordinate system, is used to define the order in which the vertices appear in the output array.

cvContourConvexHull

Finds convex hull of points set.

```
CvSeq* cvContourConvexHull( CvSeq* contour, int orientation,
CvMemStorage* storage );
```

<i>contour</i>	Sequence of 2D points.
<i>orientation</i>	Output order of the convex hull vertices CV_CLOCKWISE or CV_COUNTER_CLOCKWISE.

storage Memory storage where the convex hull must be allocated.

Discussion

The function [cvContourConvexHull](#) takes an array of points and puts out indices of points that are convex hull vertices. The function uses QUICKSORT for points sorting.

The standard, that is, bottom-left xy coordinate system, defines the order in which the vertices appear in the output array.

The function returns *CvSeq* that is filled with pointers to those points of the source contour that belong to the convex hull.

cvConvexHullApprox

Finds approximate convex hull of points set.

```
void cvConvexHullApprox( CvPoint* points, int numPoints, CvRect* boundRect,
    int bandWidth, int orientation, int* hull, int* hullsize );
```

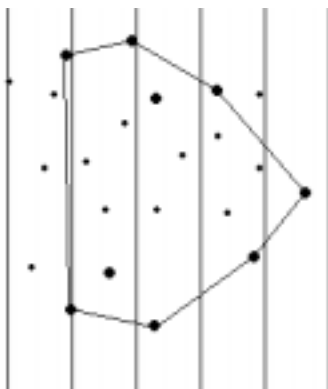
<i>points</i>	Pointer to the set of 2D points.
<i>numPoints</i>	Number of points.
<i>boundRect</i>	Pointer to the bounding rectangle of points set; not used.
<i>bandWidth</i>	Width of band used by the algorithm.
<i>orientation</i>	Output order of the convex hull vertices CV_CLOCKWISE or CV_COUNTER_CLOCKWISE.
<i>hull</i>	Indices of convex hull vertices in the input array.
<i>hullsize</i>	Number of vertices in the convex hull; output parameter.

Discussion

The function [cvConvexHullApprox](#) finds approximate convex hull of points set. The following algorithm is used: starting from the extreme left point of the input set, the plane is divided into vertical bands with the specified width. Within every band points

with maximal and minimal vertical coordinates are found and all other points are excluded. The next step is finding the exact convex hull of all remaining points (see [Figure 4-2](#)).

Figure 4-2 Finding Approximate Convex Hull



The algorithm can be used to find the exact convex hull; the value of the parameter *bandwidth* must then be equal to 1.

cvContourConvexHullApprox

Finds approximate convex hull of points set.

```
CvSeq* cvContourConvexHullApprox( CvSeq* contour, int bandwidth, int
orientation, CvMemStorage* storage );
```

<i>contour</i>	Sequence of 2D points.
<i>bandwidth</i>	Bandwidth used by the algorithm.
<i>orientation</i>	Output order of the convex hull vertices CV_CLOCKWISE or CV_COUNTER_CLOCKWISE.
<i>storage</i>	Memory storage where the convex hull must be allocated.

Discussion

The function [cvContourConvexHullApprox](#) finds approximate convex hull of points set. The following algorithm is used: starting from the extreme left point of the input set, the plane is divided into vertical bands with the specified width (bandwidth). Within every band points with maximal and minimal vertical coordinates are found and all other points are excluded. The next step is finding the exact convex hull of all remaining points (see [Figure 4-2](#)).

In case of points with integer coordinates, the algorithm can be used to find the exact convex hull; the value of the parameter *bandwidth* must then be equal to 1.

The function [cvContourConvexHullApprox](#) returns *CvSeq* that is filled with pointers to those points of the source contour that belong to the approximate convex hull.

cvCheckContourConvexity

Tests contour convex.

```
int cvCheckContourConvexity( CvSeq* contour );
```

contour Tested contour.

Discussion

The function [cvCheckContourConvexity](#) tests whether the input is a contour convex or not. The function returns 1 if the contour is convex, 0 otherwise.

cvConvexityDefects

Finds defects of convexity of contour.

```
CvSeq* cvConvexityDefects( CvSeq* contour, CvSeq* convexhull, CvMemStorage* storage );
```

contour Input contour, represented by a sequence of *CvPoint* structures.

<i>convexhull</i>	Exact convex hull of the input contour; must be computed by the function <code>cvContourConvexHull</code> .
<i>storage</i>	Memory storage where the sequence of convexity defects must be allocated.

Discussion

The function [cvConvexityDefects](#) finds all convexity defects of the input contour and returns a sequence of the `CvConvexityDefect` structures.

cvMinAreaRect

Finds circumscribed rectangle of minimal area for given convex contour.

```
void cvMinAreaRect ( CvPoint* points, int n, int left, int bottom, int right,
                    int top, CvPoint2D32f* anchor, CvPoint2D32f* vect1, CvPoint2D32f* vect2 );
```

<i>points</i>	Sequence of convex polygon points.
<i>n</i>	Number of input points.
<i>left</i>	Index of the extreme left point.
<i>bottom</i>	Index of the extreme bottom point.
<i>right</i>	Index of the extreme right point.
<i>top</i>	Index of the extreme top point.
<i>anchor</i>	Pointer to one of the output rectangle corners.
<i>vect1</i>	Pointer to the vector that represents one side of the output rectangle.
<i>vect2</i>	Pointer to the vector that represents another side of the output rectangle.

Discussion

The function [cvMinAreaRect](#) returns a circumscribed rectangle of the minimal area. The output parameters of this function are the corner of the rectangle and two incident edges of the rectangle (see [Figure 4-3](#)).

Figure 4-3 Minimal Area Bounding Rectangle



cvCalcPGH

Calculates pair-wise geometrical histogram for contour.

```
void cvCalcPGH( CvSeq* contour, CvHistogram* hist );
```

contour Input contour.

hist Calculated histogram; must be two-dimensional.

Discussion

The function [cvCalcPGH](#) calculates pair-wise geometrical histogram for contour. The algorithm considers every pair of the 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. The histogram can be used for contour matching.

cvMinEnclosingCircle

Finds minimal enclosing circle for 2D-point set.

```
void cvFindMinEnclosingCircle( CvSeq* seq, CvPoint2D32f* center, float* radius
    );
```

<i>seq</i>	Sequence that contains the input point set. Only points with integer coordinates (<i>CvPoint</i>) are supported.
<i>center</i>	Output parameter. Center of the enclosing circle.
<i>radius</i>	Output parameter. Radius of the enclosing circle.

Discussion

The function [cvMinEnclosingCircle](#) finds the minimal enclosing circle for the planar point set. *Enclosing* means that all the points from the set are either inside or on the boundary of the circle. *Minimal* means that there is no enclosing circle with smaller radius.

This section describes various fixed filters, primarily derivative operators.

Sobel Derivatives

Figure 5-1 First x Derivative Sobel Operator

For example, first x derivative Sobel operator may be expressed as a polynomial $1 + 2q + q^2 - p^2 - 2p^2q - p^2q^2 = (1 + q)^2(1 - p^2) = (1 + q)(1 + q)(1 + p)(1 - p)$ and decomposed into convolution primitives as shown in [Figure 5-1](#).

This may be used to express a hierarchy of first x and y derivative Sobel operators as follows:

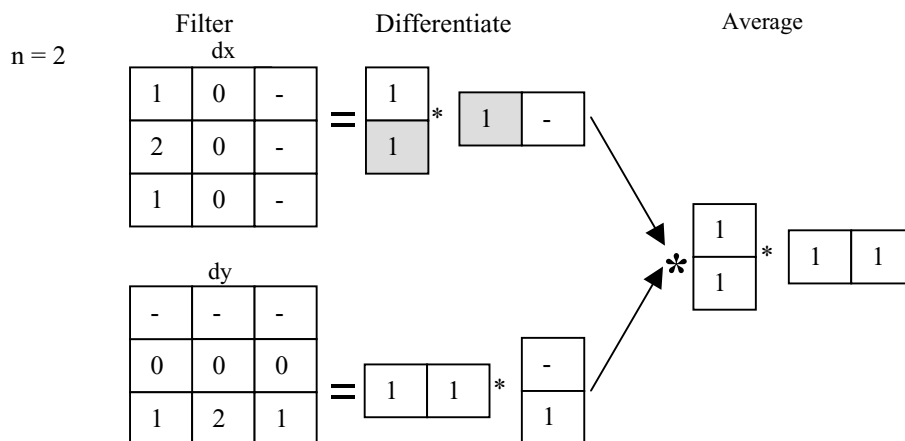
$$\frac{\partial}{\partial x} \Rightarrow (1+p)^{n-1}(1+q)^n(1-p) \quad (5.1)$$

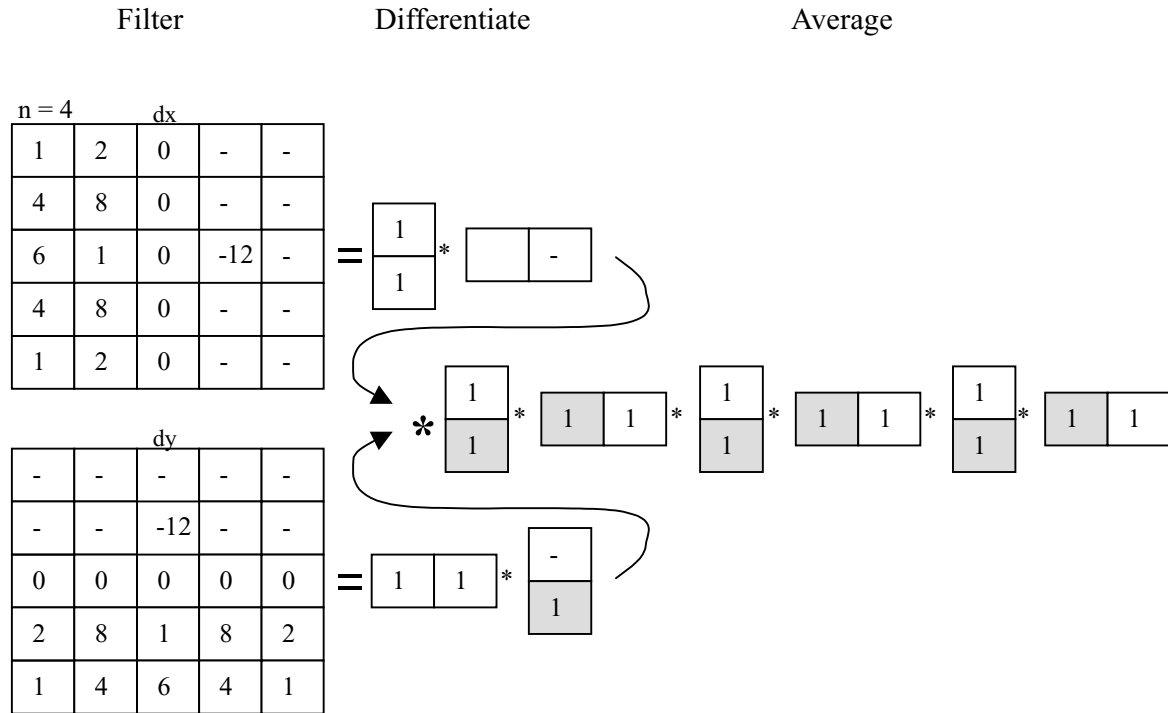
$$\frac{\partial}{\partial y} \Rightarrow (1+p)^n(1+q)^{n-1}(1-q) \quad (5.2)$$

for $n > 0$.

[Figure 5-2](#) shows the Sobel first derivative filters of equations (5.1) and (5.2) for $n = 2, 4$. The Sobel filter may be decomposed into simple “add-subtract” convolution primitives.

Figure 5-2 First Derivative Sobel Operators for $n=2$ and $n=4$





Second derivative Sobel operators can be expressed in polynomial decomposition similar to equations (5.1) and (5.2). The second derivative equations are:

$$\frac{\partial^2}{\partial x^2} \Rightarrow (1+p)^{n-2}(1+q)^n(1-p)^2, \quad (5.3)$$

$$\frac{\partial^2}{\partial y^2} \Rightarrow (1+p)^{n-1}(1+q)^{n-2}(1-q)^2, \quad (5.4)$$

$$\frac{\partial^2}{\partial x \partial y} \Rightarrow (1+p)^{n-1}(1+q)^{n-1}(1-p)(1-q) \quad (5.5)$$

for $n = 2, 3, \dots$

[Figure 5-3](#) shows the filters that result for $n = 2$ and 4. Just as shown in [Figure 5-2](#), these filters can be decomposed into simple “add-subtract” separable convolution operators as indicated by their polynomial form in the equations.

Figure 5-3 Sobel Operator Second Order Derivators for $n = 2$ and $n = 4$

The polynomial decomposition is shown above each operator.

$$\delta^2/\delta x^2 = (1+q)^2(1-p)^2$$

1	-2	1
2	-4	2
1	-2	1

$$\delta^2/\delta y^2 = (1+p)^2(1-q)^2$$

1	2	1
-2	-4	-2
1	2	1

$$\delta^2/\delta x \delta y = (1+q)(1+p)(1-q)(1-p)$$

-1	0	1
0	0	0
1	0	-1

$$\delta^2/\delta x^2 = (1+p)^2(1+q)^4(1-p)^2$$

1	0	-2	0	1
4	0	-4	0	4
6	0	-12	0	6
4	0	-8	0	4
1	0	-2	0	1

$$\delta^2/\delta y^2 = (1+q)^2(1+p)^4(1-q)^2$$

1	4	6	4	1
0	0	0	0	0
-2	-8	-12	-8	-2
0	0	0	0	0
1	4	6	4	1

$$\delta^2/\delta x \delta y = (1+p)^3(1+q)^3(1-p)(1-q)$$

-1	-2	0	2	1
-2	-4	0	4	2
0	0	0	0	0
2	4	0	-4	-2
1	2	0	-2	-1

Third derivative Sobel operators can also be expressed in the polynomial decomposition form:

$$\frac{\partial^3}{\partial x^3} \Rightarrow (1+p)^{n-3}(1+q)^n(1-p)^3, \quad (5.6)$$

$$\frac{\partial^3}{\partial y^3} \Rightarrow (1+p)^n(1+q)^{n-3}(1-q)^3, \quad (5.7)$$

$$\frac{\partial^3}{\partial x^2 \partial y} \Rightarrow (1-p)^2(1+p)^{n-2}(1+q)^{n-1}(1-q), \quad (5.8)$$

$$\frac{\partial^3}{\partial x \partial y^2} \Rightarrow (1-p)(1+p)^{n-1}(1+q)^{n-2}(1-q)^2 \quad (5.9)$$

for $n = 3, 4, \dots$. The third derivative filter needs to be applied only for the cases $n = 4$ and general.

Optimal Filter Kernels with Floating Point Coefficients

First Derivatives

[Table 5-1](#) gives coefficients for five increasingly accurate x derivative filters, the y filter derivative coefficients are just column vector versions of the x derivative filters.

Table 5-1 Coefficients for Accurate First Derivative Filters

Anchor	DX Mask Coefficients					
0	0.74038	-0.12019				
0	0.833812	-0.229945	0.0420264			
0	0.88464	-0.298974	0.0949175	-0.0178608		
0	0.914685	-0.346228	0.138704	-0.0453905	0.0086445	
0	0.934465	-0.378736	0.173894	-0.0727275	0.0239629	-0.00459622

Five increasingly accurate separable x derivative filter coefficients. The table gives half coefficients only. The full table can be obtained by mirroring across the central anchor coefficient. The greater the number of coefficients used, the less distortion from the ideal derivative filter.

Second Derivatives

[Table 5-2](#) gives coefficients for five increasingly accurate x second derivative filters. The y second derivative filter coefficients are just column vector versions of the x second derivative filters.

Table 5-2 Coefficients for Accurate Second Derivative Filters

Anchor	DX Mask Coefficients				
-2.20914	1.10457				
-2.71081	1.48229	-0.126882			
-2.92373	1.65895	-0.224751	0.0276655		
-3.03578	1.75838	-0.291985	0.0597665	-0.00827	
-3.10308	1.81996	-0.338852	0.088077	-0.0206659	0.00301915
The table gives half coefficients only. The full table can be obtained by mirroring across the central anchor coefficient. The greater the number of coefficients used, the less distortion from the ideal derivative filter.					

Laplacian Approximation

The Laplacian operator is defined as the sum of the second derivatives x and y :

$$L = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}. \quad (5.10)$$

Thus, any of the equations defined in the sections for second derivatives may be used to calculate the Laplacian for an image.

Reference

cvLaplace

Calculates convolution of input image with Laplacian operator.

```
void cvLaplace( IplImage* src, IplImage* dst, int apertureSize=3);
```

<i>src</i>	Input image.
<i>dst</i>	Destination image.
<i>apertureSize</i>	Size of the Laplacian kernel.

Discussion

The function [cvLaplace](#) calculates the convolution of the input image *src* with the Laplacian kernel of a specified size *apertureSize* and stores the result in *dst*.

cvSobel

Calculates convolution of input image with Sobel operator.

```
void cvSobel( IplImage* src, IplImage* dst, int dx, int dy, int apertureSize=3);
```

<i>src</i>	Input image.
<i>dst</i>	Destination image.
<i>dx</i>	Order of the derivative <i>x</i> .
<i>dy</i>	Order of the derivative <i>y</i> .

apertureSize Size of the extended Sobel kernel. The special value `CV_SCHARR`, equal to -1, corresponds to the Scharr filter $1/16[-3, -10, -3; 0, 0, 0; 3, 10, 3]$; may be transposed.

Discussion

The function [cvSobel](#) calculates the convolution of the input image *src* with a specified Sobel operator kernel and stores the result in *dst*.

Feature Detection Functions

Overview

A set of Sobel derivative filters may be used to find edges, ridges, and blobs, especially in a scale-space, or image pyramid, situation. Below follows a description of methods in which the filter set could be applied.

- D_x is the first derivative in the direction x just as D_y .
- D_{xx} is the second derivative in the direction x just as D_{yy} .
- D_{xy} is the partial derivative with respect to x and y .
- D_{xxx} is the third derivative in the direction x just as D_{yyy} .
- D_{xxy} and D_{xyy} are the third partials in the directions x, y .

Corner Detection

Method 1

Corners may be defined as areas where level curves multiplied by the gradient magnitude raised to the power of 3 assume a local maximum

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}. \quad (5.11)$$

Method 2

Sobel first derivative operators are used to take the derivatives x and y of an image, after which a small region of interest is defined to detect corners in. A 2x2 matrix of the sums of the derivatives x and y is subsequently created as follows:

$$C = \begin{bmatrix} \sum D_x^2 & \sum D_x D_y \\ \sum D_x D_y & \sum D_y^2 \end{bmatrix} \quad (5.12)$$

The eigenvalues are found by solving $\det(C - \lambda I) = 0$, where λ is a column vector of the eigenvalues and I is the identity matrix. For the 2x2 matrix of the equation above, the solutions may be written in a closed form:

$$\lambda = \frac{\sum D_x^2 + \sum D_y^2 \pm \sqrt{(\sum D_x^2 + \sum D_y^2)^2 - 4(\sum D_x^2 \sum D_y^2 - (\sum D_x D_y)^2)}}{2}. \quad (5.13)$$

If $\lambda_1, \lambda_2 > t$, where t is some threshold, then a corner is found at that location. This can be very useful for object or shape recognition.

Canny Edge Detector

Edges are the boundaries separating regions with different brightness or color. J.Canny suggested in [Canny86] a very good method for detecting edges. It takes grayscale image on input and returns bi-level image where non-zero pixels mark detected edges. Below the 4-stage algorithm is described.

Stage 1. Image Smoothing

The image data is smoothed by a Gaussian function of width specified by the user parameter.

Stage 2. Differentiation

The smoothed image, retrieved at Stage 1, is differentiated with respect to the directions x and y .

From the computed gradient values x and y , the magnitude and the angle of the gradient can be calculated using the hypotenuse and arctangen functionst.

In the OpenCV library smoothing and differentiation are joined in Sobel operator.

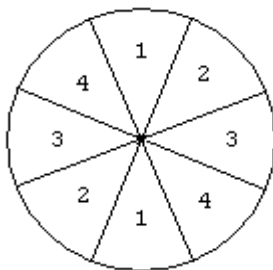
Stage 3. Non-Maximum Suppression

After the gradient has been calculated at each point of the image, the edges can be located at the points of local maximum gradient magnitude. It is done via suppression of non-maximums, that is points, whose gradient magnitudes are not local maximums. However, in this case the non-maximums perpendicular to the edge direction, rather than those in the edge direction, have to be suppressed, since the edge strength is expected to continue along an extended contour.

The algorithm starts off by reducing the angle of gradient to one of the four sectors shown in [Figure 5-4](#). The algorithm passes the 3×3 neighborhood across the magnitude array. At each point the center element of the neighborhood is compared with its two neighbors along line of the gradient given by the sector value.

If the central value is non-maximum, that is, not greater than the neighbors, it is suppressed.

Figure 5-4 Gradient Sectors



Stage 4. Edge Thresholding

The Canny operator uses the so-called “hysteresis” thresholding. Most thresholders use a single threshold limit, which means that if the edge values fluctuate above and below this value, the line appears broken. This phenomenon is commonly referred to as “streaking”. Hysteresis counters streaking by setting an upper and lower edge value limit. Considering a line segment, if a value lies above the upper threshold limit it is immediately accepted. If the value lies below the low threshold it is immediately rejected. Points which lie between the two limits are accepted if they are connected to pixels which exhibit strong response. The likelihood of streaking is reduced drastically since the line segment points must fluctuate above the upper limit and below the lower limit for streaking to occur. J. Canny recommends in [[Canny86](#)] the ratio of high to low limit to be in the range of two or three to one, based on predicted signal-to-noise ratios.

Reference

cvCanny

Implements Canny algorithm for edge detection.

```
void cvCanny( IplImage* img, IplImage* edges, double lowThresh, double
             highThresh, int apertureSize=3 );
```

<i>img</i>	Input image.
<i>edges</i>	Image to store the edges found by the function.
<i>apertureSize</i>	Size of the Sobel operator to be used in the algorithm.
<i>lowThresh</i>	Low threshold used for edge searching.
<i>highThresh</i>	High threshold used for edge searching.

Discussion

The function [cvCanny](#) finds the edges in the input image *img* and puts them into the output image *edges* using the Canny algorithm described above.

cvPreCornerDetect

Calculates two constraint images for corner detection.

```
void cvPreCornerDetect( IplImage* img, IplImage* corners, Int apertureSize );
```

img Input image.
corners Image to store the results.
apertureSize Size of the Sobel operator to be used in the algorithm.

Discussion

The function [cvPreCornerDetect](#) finds the corners on the input image *img* and stores them into the output image *corners* in accordance with [Method 1](#) for corner detection.

cvCornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
void cvCornerEigenValsAndVecs( IplImage* img, IplImage* eigenvv, int
    blockSize, int apertureSize=3 );
```

img Input image.
eigenvv Image to store the results.
blockSize Linear size of the square block over which derivatives averaging is done.
apertureSize Derivative operator aperture size in the case of byte source format. In the case of floating-point input format this parameter is the number of the fixed float filter used for differencing.

Discussion

For every raster pixel the function [cvCornerEigenValsAndVecs](#) takes a block of $blockSize \times blockSize$ pixels with the top-left corner, or top-bottom corner for bottom-origin images, at the pixel, computes first derivatives D_x and D_y within the block and then computes eigenvalues and eigenvectors of the matrix:

$$C = \begin{bmatrix} \sum D_x^2 & \sum D_x D_y \\ \sum D_x D_y & \sum D_y^2 \end{bmatrix}, \text{ where summations are performed over the whole block.}$$

The format of the frame *eigenvv* is the following: for every pixel of the input image the frame contains 6 float values ($\lambda_1, \lambda_2, x_1, y_1, x_2, y_2$).

λ_1, λ_2 are eigenvalues of the above matrix, not sorted by value.

x_1, y_1 are coordinates of the normalized eigenvector that corresponds to λ_1 .

x_2, y_2 are coordinates of the normalized eigenvector that corresponds to λ_2 .

In case of a singular matrix or if one of the eigenvalues is much less than another, all six values are set to 0. The Sobel operator with aperture width *apertureSize* is used for differentiation.

cvCornerMinEigenVal

Calculates minimal eigenvalues of image blocks for corner detection.

```
void cvCornerMinEigenVal( IplImage* img, IplImage* eigenv, int blockSize, int
    apertureSize=3 );
```

<i>img</i>	Input image.
<i>eigenvv</i>	Image to store the results.
<i>blockSize</i>	Linear size of the square block over which derivatives averaging is done.

apertureSize Derivative operator aperture size in the case of byte source format. In the case of floating-point input format this parameter is the number of the fixed float filter used for differencing.

Discussion

For every raster pixel the function [cvCornerMinEigenVal](#) takes a block of $blockSize \times blockSize$ pixels with the top-left corner, or top-bottom corner for bottom-origin images, at the pixel, computes first derivatives D_x and D_y within the block and then computes eigenvalues and eigenvectors of the matrix:

$$C = \begin{bmatrix} \sum D_x^2 & \sum D_x D_y \\ \sum D_x D_y & \sum D_y^2 \end{bmatrix}, \text{ where summations are made over the block.}$$

In case of a singular matrix the minimal eigenvalue is set to 0. The Sobel operator with aperture width *apertureSize* is used for differentiation.

cvFindCornerSubPix

Refines corner locations.

```
void cvFindCornerSubPix( IplImage* img, CvPoint2D32f* corners, int count,
CvSize win, CvSize zeroZone, CvTermCriteria criteria );
```

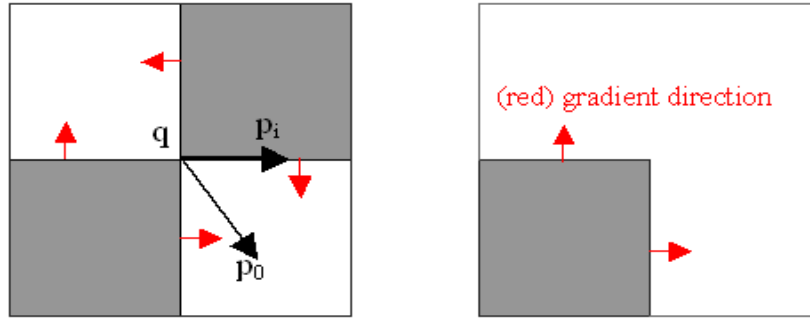
<i>img</i>	Input raster image.
<i>corners</i>	Initial coordinates of the input corners and refined coordinates on output.
<i>count</i>	Number of corners.
<i>win</i>	Half sizes of the search window. For example, if $win = (5, 5)$, then $5*2 + 1 \times 5*2 + 1 = 11 \times 11$ pixel window to be used.
<i>zeroZone</i>	Half size of the dead region in the middle of the search zone to avoid possible singularities of the autocorrelation matrix. The value of $(-1, -1)$ indicates that there is no such zone.

criteria Criteria for termination of the iterative process of corner refinement. Iterations may specify a stop when either required precision is achieved or the maximal number of iterations done.

Discussion.

The function [cvFindCornerSubPix](#) iterates to find the accurate sub-pixel location of a corner, or “radial saddle point”, as shown in [Figure 5-5](#).

Figure 5-5 Sub-Pixel Accurate Corner



The core idea of this algorithm 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. Thus:

$$\epsilon_i = \nabla I_{p_i}^T \cdot (q - p_i),$$

where ∇I_{p_i} is the image gradient at the one of the points p 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 ’s set to zero:

$$\left(\sum_i \nabla I_{p_i} \cdot \nabla I_{p_i}^T \right) \cdot q - \left(\sum_i \nabla I_{p_i} \cdot \nabla I_{p_i}^T \cdot p_i \right) = 0,$$

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

Determines strong corners on image.

```
void cvGoodFeaturesToTrack( IplImage* image, IplImage* eigImage, IplImage*
    tempImage, CvPoint2D32f* corners, int* cornerCount, double qualityLevel,
    double minDistance );
```

<i>image</i>	Source image; should be byte, signed byte, or floating-point depth single channel.
<i>eigImage</i>	Temporary image for minimal eigenvalues for pixels; must be floating-point, single channel.
<i>tempImage</i>	Another temporary image; must be floating-point, single channel.
<i>corners</i>	Output parameter. Detected corners.
<i>cornerCount</i>	Output parameter. Number of detected corners.
<i>qualityLevel</i>	Multiplier for the maxmin eigenvalue; specifies minimal accepted quality of image corners.
<i>minDistance</i>	Limit, specifying minimum possible distance between returned corners; Euclidian distance is used.

Discussion

The function [cvGoodFeaturesToTrack](#) finds corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every pixel of the source image and then performs non-maxima suppression (only local maxima in 3x3 neighborhood remain). The next step is rejecting the corners with the minimal eigenvalue less than $quality_level * \langle max_of_min_eigen_vals \rangle$. Finally, the

function ensures that all the corners found are distanced enough from one another by getting two strongest features and checking that the distance between the points is satisfactory. If not, the point is rejected.

Hough Transform

Overview

The Hough Transform (HT) is a popular method of extracting geometric primitives from raster images. The simplest version of the algorithm just detects lines, but it is easily generalized to find more complex features. There are several classes of HT that differ by the image information available. If the image is arbitrary, the Standard Hough Transform (SHT, [[Trucco98](#)]) should be used.

SHT, like all HT algorithms, considers a discrete set of single primitive parameters. If lines should be detected, then the parameters are ρ and θ , such that the line equation is $\rho = x \cos(\theta) + y \sin(\theta)$.

Here ρ is the distance from the origin to the line, and θ is the angle between the axis x and the perpendicular to the line vector that points from the origin to the line. Every pixel in the image may belong to many lines described by a set of parameters. In other words, the “*accumulator*” is defined which is an integer array $A(\rho, \theta)$ containing only zeroes initially. For each non-zero pixel in the image all accumulator elements corresponding to lines that contain the pixel are incremented by 1. Then a threshold is applied to distinguish lines and noise features, that is select all pairs (ρ, θ) for which $A(\rho, \theta)$ is greater than the threshold value. All such pairs characterize detected lines.

Multidimensional Hough Transform (MHT) is a modification of SHT. It performs precalculation of SHT on rough resolution in parameter space and detects the regions of parameter values that possibly have strong support, that is, correspond to lines in the source image. MHT should be applied to images with few lines and without noise.

[[Matas98](#)] presents advanced algorithm for detecting multiple primitives, Progressive Probabilistic Hough Transform (PPHT). The idea is to consider random pixels one by one. Every time the accumulator is changed, the highest peak is tested for threshold exceeding. If the test succeeds, points that belong to the corridor specified by the peak are removed. If the number of points exceeds the predefined value, that is, minimum

line length, then the feature is considered a line, otherwise it is considered a noise. Then the process repeats from the very beginning until no pixel remains in the image. The algorithm improves the result every step, so it can be stopped any time. [Matas98] claims that PPHT is easily generalized in almost all cases where SHT could be generalized. The disadvantage of this method is that, unlike SHT, it does not process some features, for instance, crossed lines, correctly.

For more information see [Matas98] and [Trucco98].

Reference

cvHoughLines

Finds lines in binary image, SHT algorithm.

```
void cvHoughLines ( IplImage* src, double rho, double theta, int threshold,
                    float* lines, int linesNumber);
```

<i>src</i>	Source image.
<i>rho</i>	Radius resolution.
<i>theta</i>	Angle resolution.
<i>threshold</i>	Threshold parameter.
<i>lines</i>	Pointer to the array of output lines parameters. The array should have $2 * linesNumber$ elements.
<i>linesNumber</i>	Maximum number of lines.

Discussion

The function [cvHoughLines](#) implements Standard Hough Transform (SHT) and demonstrates average performance on arbitrary images. The function returns number of detected lines. Every line is characterized by pair (ρ, θ) , where ρ is distance from line to point $(0, 0)$ and θ is the angle between the line and horizontal axis.

cvHoughLinesSDiv

Finds lines in binary image, MHT algorithm.

```
int cvHoughLinesSDiv ( IplImage* src, double rho, int srn, double theta, int
                      stn, int threshold, float* lines, int linesNumber );
```

<i>src</i>	Source image.
<i>rho</i>	Rough radius resolution.
<i>srn</i>	Radius accuracy coefficient, ρ/srn is accurate ρ resolution.
<i>theta</i>	Rough angle resolution.
<i>stn</i>	Angle accuracy coefficient, θ/stn is accurate angle resolution.
<i>threshold</i>	Threshold parameter.
<i>lines</i>	Pointer to array of the detected lines parameters. The array should have $2*linesNumber$ elements.
<i>linesNumber</i>	Maximum number of lines.

Discussion

The function [cvHoughLinesSDiv](#) implements coarse-to-fine variant of SHT and is significantly faster than the latter on images without noise and with a small number of lines. The output of the function has the same format as the output of the function [cvHoughLines](#).

cvHoughLinesP

Finds line segments in binary image, PPHT algorithm.

```
int cvHoughLinesP( IplImage* src, double rho, double theta, int threshold,
                  int lineLength, int lineGap, int* lines, int linesNumber );
```

<i>src</i>	Source image.
------------	---------------

<i>rho</i>	Rough radius resolution.
<i>theta</i>	Rough angle resolution.
<i>threshold</i>	Threshold parameter.
<i>lineLength</i>	Minimum accepted line length.
<i>lineGap</i>	Maximum length of accepted line gap.
<i>lines</i>	Pointer to array of the detected line segments' ending coordinates. The array should have <i>linesNumber*4</i> elements.
<i>linesNumber</i>	Maximum number of line segments.

Discussion

The function [cvHoughLinesP](#) implements Progressive Probabilistic Standard Hough Transform. It retrieves no more than *linesNumber* line segments; each of those must be not shorter than *lineLength* pixels. The method is significantly faster than SHT on noisy images, containing several long lines. The function returns number of detected segments. Every line segment is characterized by the coordinates of its ends(x_1, y_1, x_2, y_2).

Image Statistics

6

This chapter describes a set of functions that compute different information about images, considering their pixels as independent observations of a stochastic variable.

Overview

The computed values have statistical character and most of them depend on values of the pixels rather than on their relative positions. These statistical characteristics represent integral information about a whole image or its regions.

The first part of the chapter describes the characteristics that are typical for any stochastic variable or deterministic set of numbers, such as mean value, standard deviation, min and max values.

The second part describes the function for calculating the most widely used norms for a single image or a pair of images. The latter is often used to compare images.

The third part describes moments functions for calculating integral geometric characteristics of a 2D object, represented by grayscale or bi-level raster image, such as mass center, orientation, size, and rough shape description. As opposite to simple moments, that are used for characterization of any stochastic variable or other data, Hu invariants, described in the last function discussion, are unique for image processing because they are specifically designed for 2D shape characterization. They are invariant to several common geometric transformations.

Reference

cvCountNonZero

Counts non-zero pixels in image.

```
int cvCountNonZero (IplImage* image );
```

image Pointer to the source image.

Discussion

The function [cvCountNonZero](#) returns number of non-zero pixels in the whole image or selected image ROI.

cvSumPixels

Summarizes pixel values in image.

```
double cvSumPixels( IplImage* image );
```

image Pointer to the source image.

Discussion

The function [cvSumPixels](#) returns sum of pixel values in the whole image or selected image ROI.

cvMean

Calculates mean value in image region.

```
double cvMean( IplImage* image, IplImage* mask=0 );
```

image Pointer to the source image.

mask Mask image.

Discussion

The function [cvMean](#) calculates the mean of pixel values in the whole image, selected ROI or, if *mask* is not NULL, in an image region of arbitrary shape.

cvMean_StdDev

Calculates mean and standard deviation in image region.

```
void cvMean_StdDev( IplImage* image, double* mean, double* stddev,  
IplImage* mask=0 );
```

image Pointer to the source image.

mean Pointer to returned mean.

stddev Pointer to returned standard deviation.

mask Pointer to the single-channel mask image.

Discussion

The function [cvMean_StdDev](#) calculates mean and standard deviation of pixel values in the whole image, selected ROI or, if *mask* is not NULL, in an image region of arbitrary shape. If the image has more than one channel, the ROI must be selected.

cvMinMaxLoc

Finds global minimum and maximum in image region.

```
void cvMinMaxLoc( IplImage* image, double* minVal, double* maxVal,  
                  CvPoint* minLoc, CvPoint* maxLoc, IplImage* mask=0 );
```

<i>image</i>	Pointer to the source image.
<i>minVal</i>	Pointer to returned minimum value.
<i>maxVal</i>	Pointer to returned maximum value.
<i>minLoc</i>	Pointer to returned minimum location.
<i>maxLoc</i>	Pointer to returned maximum location.
<i>mask</i>	Pointer to the single-channel mask image.

Discussion

The function [cvMinMaxLoc](#) finds minimum and maximum pixel values and their positions. The extremums are searched over the whole image, selected ROI or, if *mask* is not `NULL`, over an image region of arbitrary shape. If the image has more than one channel, the ROI must be selected.

cvNorm

Calculates image norm, difference norm or relative difference norm.

```
double cvNorm( IplImage* imgA, IplImage* imgB, int normType, IplImage* mask=0  
              );
```

<i>imgA</i>	Pointer to the first source image.
<i>imgB</i>	Pointer to the second source image if any, <code>NULL</code> otherwise.
<i>normType</i>	Type of norm.

mask Pointer to the single-channel mask image.

Discussion

The function [cvNorm](#) calculates images norms defined below. If *imgB* = NULL, the following three norm types of image *A* are calculated:

$$NormType = CV_C: \|A\|_C = \max(|A_{ij}|),$$

$$NormType = CV_L1: \|A\|_{L_1} = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |A_{ij}|,$$

$$NormType = CV_L2: \|A\|_{L_2} = \sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} A_{ij}^2}.$$

If *imgB* ≠ NULL, the difference or relative difference norms are calculated:

$$NormType = CV_C: \|A - B\|_C = \max(|A_{ij} - B_{ij}|),$$

$$NormType = CV_L1: \|A - B\|_{L_1} = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |A_{ij} - B_{ij}|,$$

$$NormType = CV_L2: \|A - B\|_{L_2} = \sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} (A_{ij} - B_{ij})^2},$$

$$NormType = CV_RELATIVEC: \|A - B\|_C / \|B\|_C = \frac{\max(|A_{ij} - B_{ij}|)}{\max(|B_{ij}|)},$$

$$NormType = CV_RELATIVEL1: \|A - B\|_{L_1} / \|B\|_{L_1} = \frac{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |A_{ij} - B_{ij}|}{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |B_{ij}|},$$

$$\text{NormType} = \text{CV_RELATIVE_L2}: \|A - B\|_{L_2} / \|B\|_{L_2} = \frac{\sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} (A_{ij} - B_{ij})^2}}{\sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} (B_{ij})^2}}.$$

The function [cvNorm](#) returns calculated norm.

cvMoments

Calculates all moments up to third order of image plane and fills moment state structure.

```
void cvMoments( IplImage* image, CvMoments* moments, int isBinary=0 );
```

<i>image</i>	Pointer to the image or to top-left corner of its ROI.
<i>moments</i>	Pointer to returned moment state structure.
<i>isBinary</i>	If the flag is non-zero, all the zero pixel values are treated as zeroes, all the others are treated as ones.

Discussion

The function [cvMoments](#) calculates moments up to the third order and writes the result to the moment state structure. This structure is used then to retrieve a certain spatial, central, or normalized moment or to calculate Hu moments.

cvGetSpatialMoment

Retrieves spatial moment from moment state structure.

```
double cvGetSpatialMoment( CvMoments* moments, int x_order, int y_order );
```

moments Pointer to the moment state structure.
x_order Order *x* of required moment.
y_order Order *y* of required moment
 (0 ≤ *x_order*, *y_order*; *x_order* + *y_order* ≤ 3).

Discussion

The function [cvGetSpatialMoment](#) retrieves the spatial moment, which is defined as:

$$M_{x_order, y_order} = \sum_{x, y} I(x, y) x^{x_order} y^{y_order}, \text{ where}$$

$I(x, y)$ is the intensity of the pixel (x, y) .

cvGetCentralMoment

Retrieves central moment from moment state structure.

```
double cvGetCentralMoment( CvMoments* moments, int x_order, int y_order );
```

moments Pointer to the moment state structure.
x_order Order *x* of required moment.
y_order Order *y* of required moment
 (0 ≤ *x_order*, *y_order*; *x_order* + *y_order* ≤ 3).

Discussion

The function [cvGetCentralMoment](#) retrieves the central moment, which is defined as:

$$\mu_{x_order, y_order} = \sum_{x, y} I(x, y) (x - \bar{x})^{x_order} (y - \bar{y})^{y_order}, \text{ where}$$

$I(x, y)$ is the intensity of pixel (x, y) , \bar{x} is the coordinate x of the mass center, \bar{y} is the coordinate y of the mass center:

$$\bar{x} = \frac{M_{1,0}}{M_{0,0}}, \bar{y} = \frac{M_{0,1}}{M_{0,0}}.$$

cvGetNormalizedCentralMoment

Retrieves normalized central moment from moment state structure.

```
double cvGetNormalizedCentralMoment(CvMoments* moments, int x_order, int y_order);
```

<i>moments</i>	Pointer to the moment state structure.
<i>x_order</i>	Order x of required moment.
<i>y_order</i>	Order y of required moment
	($0 \leq x_order, y_order; x_order + y_order \leq 3$).

Discussion

The function [cvGetNormalizedCentralMoment](#) retrieves the normalized central moment, which is defined as:

$$\eta_{x_order, y_order} = \frac{\mu_{x_order, y_order}}{M_{0,0}^{((x_order + y_order)/2 + 1)}}.$$

cvGetHuMoments

Calculates seven moment invariants from moment state structure.

```
void cvGetHuMoments( CvMoments* moments, CvHuMoments* HuMoments );
```

moments Pointer to the moment state structure.

HuMoments Pointer to Hu moments structure.

Discussion

The function [cvGetHuMoments](#) calculates seven Hu invariants using the following formulas:

$$h_1 = \eta_{20} + \eta_{02},$$

$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2,$$

$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2,$$

$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2,$$

$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

$$h_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}),$$

$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

These values are proved to be invariants to the image scale, rotation, and reflection except the first one, whose sign is changed by reflection.

Pyramids

7

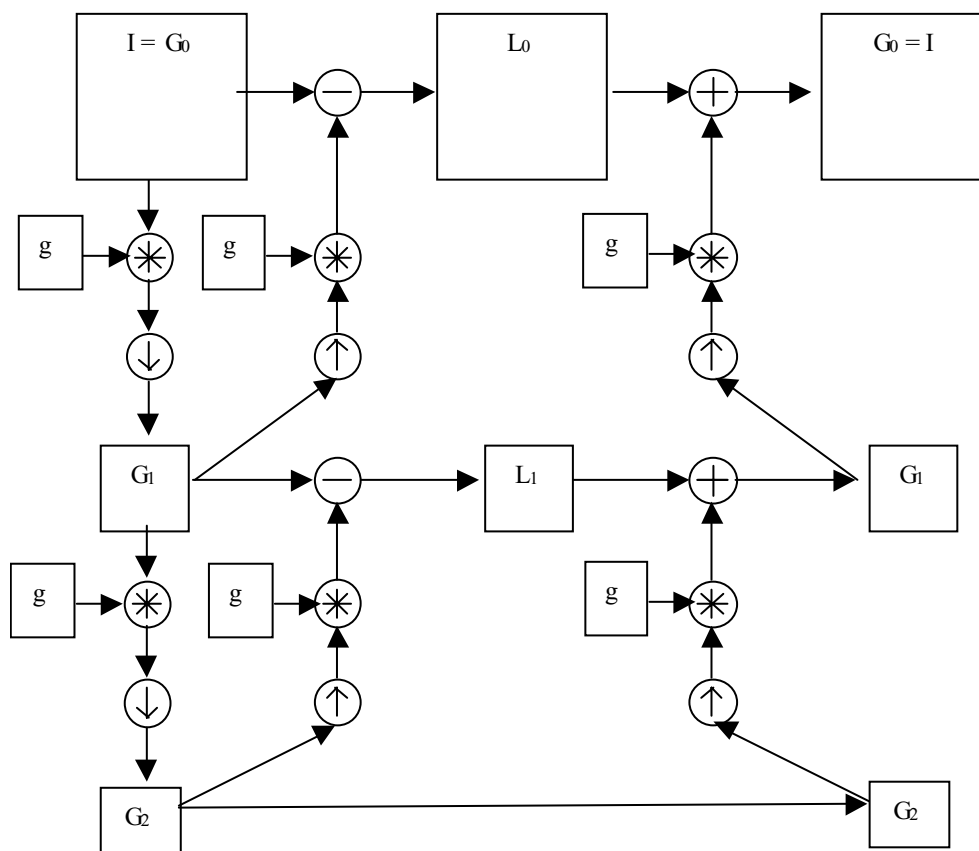
This chapter describes functions that support generation and reconstruction of Gaussian and Laplacian Pyramids.

Overview

[Figure 7-1](#) shows the basics of creating Gaussian or Laplacian pyramids. The original image G_0 is convolved with a Gaussian, then down-sampled to get the reduced image G_1 . This process can be continued as far as desired or until the image size is one pixel.

The Laplacian pyramid can be built from a Gaussian pyramid as follows: Laplacian level “ k ” can be built by up-sampling the lower level image G_{k+1} . Convolution with a Gaussian kernel “ g ” interpolates the pixels “missing” after up-sampling. The resulting image is subtracted from the image G_k . To rebuild the original image, the process is reversed as [Figure 7-1](#) shows.

Figure 7-1 A Three-Level Gaussian and Laplacian Pyramid.



The Gaussian image pyramid on the left is used to create the Laplacian pyramid in the center, which is used to reconstruct the Gaussian pyramid and the original image on the right. In the figure, I is the original image, G is the Gaussian image, L is the Laplacian image. Subscripts denote level of the pyramid. A Gaussian kernel g is used to convolve the image before down-sampling or after up-sampling.

Image Segmentation by Pyramid

Computer vision uses pyramid based image processing techniques on a wide scale now. The pyramid provides a hierarchical smoothing, segmentation, and hierarchical computing structure that supports fast analysis and search algorithms.

P. J. Burt suggested a pyramid-linking algorithm as an effective implementation of a combined segmentation and feature computation algorithm [Burt81]. This algorithm, described also in [Jahne97], finds connected components without preliminary threshold, that is, it works on grayscale image. It is an iterative algorithm.

Burt's algorithm includes the following steps:

1. Computation of the Gaussian pyramid.
2. Segmentation by pyramid-linking.
3. Averaging of linked pixels.

Steps 2 and 3 are repeated iteratively until a stable segmentation result is reached.

After computation of the Gaussian pyramid a son-father relationship is defined between nodes (pixels) in adjacent levels. The following attributes may be defined for every node (i, j) on the level l of the pyramid:

$c[i, j, l][t]$ is the value of the local image property, e.g., intensity;

$a[i, j, l][t]$ is the area over which the property has been computed;

$p[[i, j, l][t]$ is pointer to the node's father, which is at level $l+1$;

$s[i, j, l][t]$ is the segment property, the average value for the entire segment containing the node.

The letter t stands for the iteration number ($t \geq 0$). For $t = 0$, $c[i, j, l][0] = G_{i, j}^l$.

For every node (i, j) at level l there are 16 candidate son nodes at level $l-1$ (i', j') , where

$$i' \in \{2i-1, 2i, 2i+1, 2i+2\}, j' \in \{2j-1, 2j, 2j+1, 2j+2\}. \quad (7.1)$$

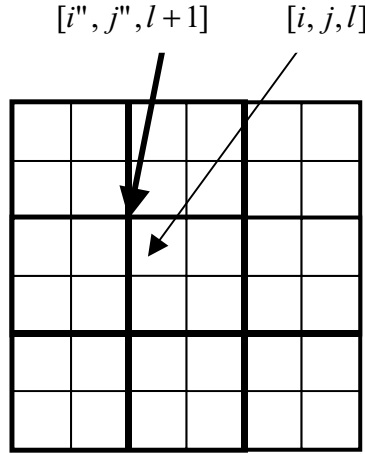
For every node (i, j) at level l there are 4 candidate father nodes at level $l+1$ (i'', j'') , (see [Figure 7-2](#)), where

$$i'' \in \{(i-1)/2, (i+1)/2\}, j'' \in \{(j-1)/2, (j+1)/2\}. \quad (7.2)$$

Son-father links are established for all nodes below the top of pyramid for every iteration t . Let $d[n][t]$ be the absolute difference between the c value of the node (i, j) at level l and its n^{th} candidate father, then

$$p[i, j, l][t] = \underset{1 \leq n \leq 4}{\operatorname{argmin}} d[n][t] \quad (7.3)$$

Figure 7-2 Connections between Adjacent Pyramid Levels



After the son-father relationship is defined, the t , c , and a values are computed from bottom to the top for the $0 \leq l \leq n$ as

$$a[i, j, 0][t] = 1, \quad c[i, j, 0][t] = c[i, j, 0][0], \quad i, j, l][t] = \sum a[i', j', l-1][t],$$

where sum is calculated over all (i', j') node sons, as indicated by the links p in (7.3).

If $a[i, j, l][t] > 0$ then $i, l][t] = \sum ([i', j', l-1][t] \cdot c[i', j', l-1][t]) / a[i, j, l][t]$, but if $a[i, j, 0][t] = 0$, the node has no sons, $c[i, j, 0][t]$ is set to the value of one of its candidate sons selected at random. No segment values are calculated in the top down order. The value of the initial level L is an input parameter of the algorithm. At the level L the segment value of each node is set equal to its local property value:

$$s[i, j, L][t] = c[i, j, L][t].$$

For lower levels $l < L$ each node value is just that of its father

$$s[i, j, l][t] = c[i'', j'', l+1][t].$$

Here node (i'', j'') is the father of (i, j) , as established in Equation (7.3).

After this the current iteration t finishes and the next iteration $t+1$ begins. Any changes in pointers in the next iteration result in changes in the values of local image properties.

The iterative process is continued until no changes occur between two successive iterations.

The choice of L only determines the maximum possible number of segments. If the number of segments less than the numbers of nodes at the level L , the values of $c[i, j, L][t]$ are clustered into a number of groups equal to the desired number of segments. The group average value is computed from the c values of its members, weighted by their areas a , and replaces the value c for each node in the group.

Data Structures

The pyramid functions use the data structure `IplImage` for image representation and the data structure `CvSeq` for the sequence of the connected components representation. Every element of this sequence is the data structure `CvConnectedComp` for the single connected component representation in memory.

The C language definition for the `CvConnectedComp` structure is given below.

Example 7-1 CvConnectedComp Structure Definition

```
typedef struct CvConnectedComp
{
    double area;           /* area of the segmented
                           component */
    float value;           /* gray scale value of the
                           segmented component */
    CvRect rect;           /* ROI of the segmented component
                           */
} CvConnectedComp;
```

Reference

cvPyrDown

Downsamples image.

```
void cvPyrDown(IplImage* src, IplImage* dst, IplFilter
filter=IPL_GAUSSIAN_5x5);
```

<i>src</i>	Pointer to the source image.
<i>dst</i>	Pointer to the destination image.
<i>filter</i>	Type of the filter used for convolution; only IPL_GAUSSIAN_5x5 is currently supported.

Discussion

The function [cvPyrDown](#) performs downsampling step of Gaussian pyramid decomposition. First it convolves source image with the specified filter and then downsamples the image by rejecting even rows and columns. So the destination image is four times smaller than the source image.

cvPyrUp

Upsamples image.

```
void cvPyrUp(IplImage* src, IplImage* dst, IplFilter filter=IPL_GAUSSIAN_5x5);
```

<i>src</i>	Pointer to the source image.
<i>dst</i>	Pointer to the destination image.
<i>filter</i>	Type of the filter used for convolution; only IPL_GAUSSIAN_5x5 is currently supported.

Discussion

The function [cvPyrUp](#) performs upsampling step of Gaussian pyramid decomposition. First it upsamples the source image by injecting even zero rows and columns and then convolves result with the specified filter multiplied by 4 for interpolation. So the destination image is four times larger than the source image.

cvPyrSegmentation

Implements image segmentation by pyramids.

```
void cvPyrSegmentation(IplImage* srcImage, IplImage* dstImage, CvMemStorage*
    storage, CvSeq** comp, int level, double threshold1, double threshold2);
```

<i>srcImage</i>	Pointer to the input image data.
<i>dstImage</i>	Pointer to the output segmented data.
<i>storage</i>	Storage; stores the resulting sequence of connected components.
<i>comp</i>	Pointer to the output sequence of the segmented components.
<i>level</i>	Maximum level of the pyramid for the segmentation.
<i>threshold1</i>	Error threshold for establishing the links.
<i>threshold2</i>	Error threshold for the segments clustering.

Discussion

The function [cvPyrSegmentation](#) 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

$\rho(c(a), c(b)) < threshold1$. After the connected components are defined, they are joined into several clusters. Any two segments *A* and *B* belong to the same cluster, if $\rho(c(A), c(B)) < threshold2$. The input image has only one channel, then $\rho(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then $\rho(c^1, c^2) = 0,3 \cdot (c_r^1 - c_r^2) + 0,59 \cdot (c_g^1 - c_g^2) + 0,11 \cdot (c_b^1 - c_b^2)$. There may be more than one connected component per a cluster.

Input *srcImage* and output *dstImage* should have the identical `IPL_DEPTH_8U` depth and identical number of channels (1 or 3).

Morphology

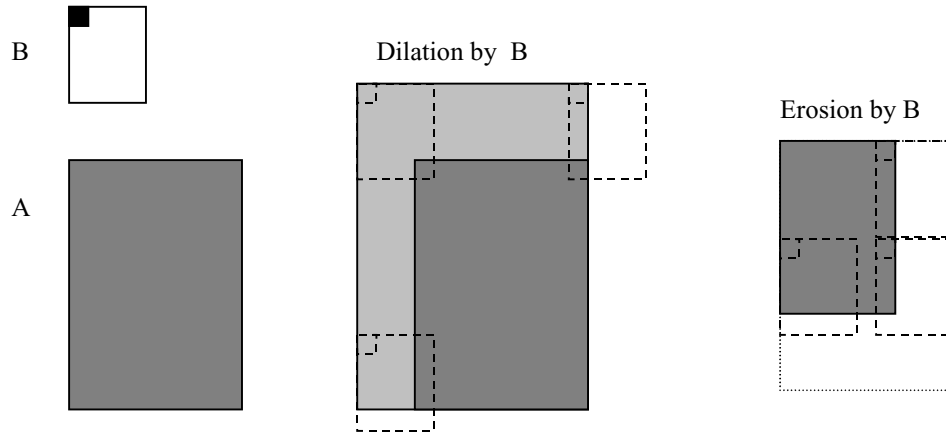
8

This chapter describes an expanded set of morphological operators that can be used for noise filtering, merging or splitting image regions, as well as for region boundary detection.

Overview

Mathematical Morphology is a set-theory method of image analysis first developed by Matheron and Serra at the Ecole des Mines, Paris [[Serra82](#)]. The two basic morphological operations are erosion, or thinning, and dilation, or thickening. All operations involve an image A , called the *object of interest*, and a kernel element B , called the *structuring element*. The image and structuring element could be in any number of dimensions, but the most common use is with a 2D binary image, or with a 3D gray scale image. The element B is most often a square or a circle, but could be any shape. Just like in convolution, B is a kernel or template with an anchor point.

[Figure 8-1](#) shows dilation and erosion of object A by B . The element B is rectangular with an anchor point at upper left shown as a dark square.

Figure 8-1 Dilation and Erosion of A by B.

If B_t is the translation of B around the image, then dilation of object A by structuring element B is

$$A \oplus B = \{t : B_t \cap A \neq \emptyset\}.$$

It means every pixel is in the set, if the intersection is not null. That is, a pixel under the anchor point of B is marked “on”, if at least one pixel of B is inside of A .

$A \oplus nB$ indicates the dilation is done n times.

Erosion of object A by structuring element B is

$$A \ominus B = \{t : B_t \subseteq A\}.$$

That is, a pixel under the anchor of B is marked “on”, if B is entirely within A .

$A \ominus nB$ indicates the erosion is done n times and can be useful in finding ∂A , the boundary of A :

$$\partial A = A - (A \ominus nB).$$

Opening of A by B is

$$A \circ B = (A \ominus nB) \oplus nB. \quad (8.1)$$

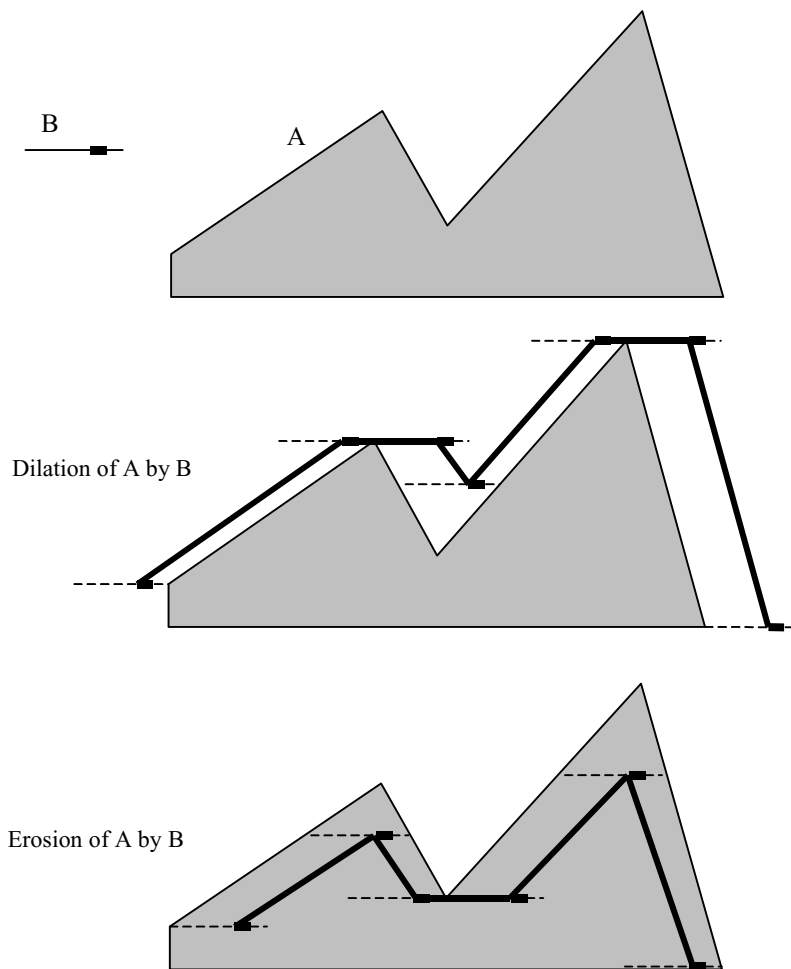
Closing of A by B is

$$A \bullet B = (A \oplus nB) \ominus nB, \quad (8.2)$$

where $n > 0$.

Flat Structuring Elements for Gray Scale

Erosion and dilation can be done in 3D, that is, with gray levels. 3D structuring elements can be used, but the simplest and the best way is to use a flat structuring element B as shown in [Figure 8-2](#). In the figure, B has an anchor slightly to the right of the center as shown by the dark mark on B . [Figure 8-2](#) shows 1D cross-section of both dilation and erosion of a gray level image A by a flat structuring element B .

Figure 8-2 Dilation and Erosion of Gray Scale Image.

In [Figure 8-2](#), dilation is mathematically

$$\sup_{y \in B_t} A$$

and erosion is

$$\inf_{y \in B_t} A$$

Open and Close Gray Level with Flat Structuring Element

The typical position of the anchor of the structuring element B for opening and closing is in the center. Subsequent opening and closing could be done in the same manner as in the Opening (8.1) and Closing (8.2) equations above to smooth off jagged objects as opening tends to cut off peaks and closing tends to fill in valleys.

Morphological Gradient Function

A morphological gradient may be taken with the flat gray scale structuring elements as follows:

$$\text{grad}(A) = \frac{(A \oplus B_{\text{flat}}) - (A \ominus B_{\text{flat}})}{2}.$$

Top Hat and Black Hat

Top Hat (TH) is a function that isolates bumps and ridges from gray scale objects. In other words, it can detect areas that are lighter than the surrounding neighborhood of A and smaller compared to the structuring element. The function subtracts the opened version of A from the gray scale object A :

$$TH_B(A) = A - (A \circ B_{\text{flat}}).$$

Black Hat (TH^d) is the dual function of Top Hat in that it isolates valleys and “cracks off” ridges of a gray scale object A , that is, the function detects dark and thin areas by subtracting A from the closed image A :

$$TH_B^d(A) = (A \bullet B_{\text{flat}}) - A.$$

Thresholding often follows both Top Hat and Black Hat operations.

Reference

cvCreateStructuringElementEx

Creates structuring element.

```
IplConvKernel* cvCreateStructuringElementEx(int nCols, int nRows, int anchorX,
int anchorY, CvElementShape shape, int* values );
```

<i>nCols</i>	Number of columns in the structuring element.
<i>nRows</i>	Number of rows in the structuring element.
<i>anchorX</i>	Relative horizontal offset of the anchor point.
<i>anchorY</i>	Relative vertical offset of the anchor point.
<i>shape</i>	Shape of the structuring element; may have the following values: <ul style="list-style-type: none"> • 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 <i>values</i> specifies the mask, that is, which neighbors of the pixel must be considered.
<i>values</i>	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 <code>NULL</code> , 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.

Discussion

The function [cvCreateStructuringElementEx](#) allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the morphological operations.

cvReleaseStructuringElement

Deletes structuring element.

```
void cvReleaseStructuringElement(IplConvKernel** ppElement);
```

ppElement Pointer to the deleted structuring element.

Discussion

The function [cvReleaseStructuringElement](#) releases the structure `IplConvKernel` that is no longer needed. If **ppElement* is `NULL`, the function has no effect. The function returns created structuring element.

cvErode

Erodes image by using arbitrary structuring element.

```
void cvErode( IplImage* src, IplImage* dst, IplConvKernel* B, int iterations);
```

src Source image.

dst Destination image.

B Structuring element used for erosion. If `NULL`, a 3x3 rectangular structuring element is used.

iterations Number of times erosion is applied.

Discussion

The function [cvErode](#) erodes the source image. The function takes the pointer to the structuring element, consisting of “zeros” and “minus ones”; the minus ones determine neighbors of each pixel from which the minimum is taken and put to the corresponding destination pixel. The function supports the in-place mode when the source and

destination pointers are the same. Erosion can be applied several times (*iterations* parameter). Erosion on a color image means independent transformation of all channels.

cvDilate

Dilates image by using arbitrary structuring element.

```
void cvDilate( IplImage* pSrc, IplImage* pDst, IplConvKernel* B, int
               iterations);
```

<i>pSrc</i>	Source image.
<i>pDst</i>	Destination image.
<i>B</i>	Structuring element used for dilation. If <code>NULL</code> , a 3x3 rectangular structuring element is used.
<i>iterations</i>	Number of times dilation is applied.

Discussion

The function [cvDilate](#) performs dilation of the source image. It takes pointer to the structuring element that consists of “zeros” and “minus ones”; the minus ones determine neighbors of each pixel from which the maximum is taken and put to the corresponding destination pixel. Function supports in-place mode. Dilation can be applied several times (*iterations* parameter). Dilation of color image means independent transformation of all channels.

cvMorphologyEx

Performs advanced morphological transformations.

```
void cvMorphologyEx( IplImage* src, IplImage* dst, IplImage* temp,
IplConvKernel* B, CvMorphOp op, int iterations );
```

<i>src</i>	Source image.
<i>dst</i>	Destination image.
<i>temp</i>	Temporary image, required in some cases.
<i>B</i>	Structuring element.
<i>op</i>	Type of morphological operation: <ul style="list-style-type: none">• CV_MOP_OPEN, opening;• CV_MOP_CLOSE, closing;• CV_MOP_GRADIENT, morphological gradient;• CV_MOP_TOPHAT, top hat;• CV_MOP_BLACKHAT, black hat. (See Overview for description of these operations).
<i>iterations</i>	Number of times erosion and dilation are applied during the complex operation.

Discussion

The function [cvMorphologyEx](#) performs advanced morphological transformations. The function uses [cvErode](#) and [cvDilate](#) to perform more complex operations. The parameter *temp* must be non-NULL and point to the image of the same size and same format as *src* and *dst* when *op* is CV_MOP_GRADIENT, or when *op* is CV_MOP_TOPHAT or *op* is CV_MOP_BLACKHAT and *src* is equal to *dst* (in-place operation).

Background Subtraction

9

The chapter describes basic functions that enable building statistical model of background for its further subtraction.

Overview

In this chapter the term "background" stands for a set of motionless image pixels, that is, pixels that do not belong to any object, moving in front of the camera. This definition can vary if considered in other techniques of object extraction. For example, if a depth map of the scene is obtained, background can be determined as parts of scene that are located far enough from the camera.

The simplest background model assumes that every background pixel brightness varies independently, according to normal distribution. The background characteristics can be calculated by accumulating several dozens of frames, as well as their squares. That means finding a sum of pixel values in the location $S_{(x,y)}$ and a sum of squares of the values $Sq_{(x,y)}$ for every pixel location.

Then mean is calculated as $m_{(x,y)} = \frac{S_{(x,y)}}{N}$, where N is the number of the frames collected, and

standard deviation as $\sigma_{(x,y)} = \sqrt{\frac{Sq_{(x,y)}}{N} - \left(\frac{S_{(x,y)}}{N}\right)^2}$.

After that the pixel in certain pixel location in certain frame is regarded as belonging to a moving object if condition $abs(m_{(x,y)} - p_{(x,y)}) > C\sigma_{(x,y)}$ is met, where C is a certain constant. If C is equal to 3, it is the well-known "three sigmas" rule. To obtain that background model, one should put any objects away from camera for a few seconds, so that a whole image from camera represents subsequent background observation.

There can be improvement of the simplest technique that was just described. First, it is reasonable to provide adaptation of background differencing model to changes of lighting conditions and background scenes, e.g., when camera moves or some object is passing behind the front object. The simple accumulation in order to calculate mean brightness can be replaced with running average. Also, several techniques can be used to identify moving parts of the scene and exclude them in the course of background information accumulation. The techniques include change detection, e.g., via `cvAbsDiff` with `cvThreshold`, optical flow and, probably, others.

The functions from the chapter are simply the basic functions for background information accumulation and they can not make up the complete background differencing module alone.

Reference

cvAcc

Adds frame to accumulator.

```
void cvAcc( IplImage* img, IplImage* sum, IplImage* mask=0 );
```

<i>img</i>	Input image.
<i>sum</i>	Accumulating image.
<i>mask</i>	Mask image.

Discussion

The function [cvAcc](#) adds a new image *img* to the accumulating sum *sum*. If *mask* is not NULL, it specifies what accumulator pixels are affected.

cvSquareAcc

Calculates square of source image and adds it to destination image.

```
void cvSquareAcc( IplImage* img, IplImage* sqSum, IplImage* mask=0 );
```

<i>img</i>	Input image.
<i>sqSum</i>	Accumulating image.
<i>mask</i>	Mask image.

Discussion

The function [cvSquareAcc](#) adds the square of the new image *img* to the accumulating sum *sqSum* of image squares. If *mask* is not NULL, it specifies what accumulator pixels are affected.

cvMultiplyAcc

Calculates product of two input images and adds it to destination image.

```
void cvMultiplyAcc( IplImage* imgA, IplImage* imgB, IplImage* acc, IplImage* mask=0 );
```

<i>imgA</i>	First input image.
<i>imgB</i>	Second input image.
<i>acc</i>	Accumulating image.
<i>mask</i>	Mask image.

Discussion

The function [cvMultiplyAcc](#) multiplies input *imgA* by *imgB* and adds the result to the accumulating sum *acc* of the image products. If *mask* is not NULL, it specifies what accumulator pixels are affected.

cvRunningAvg

Calculates weighted sum of two images.

```
void cvRunningAvg( IplImage* imgY, IplImage* imgU, double alpha,
IplImage* mask=0 )
```

<i>imgY</i>	Input image.
<i>imgU</i>	Destination image.
<i>alpha</i>	Weight of input image.
<i>mask</i>	Mask image.

Discussion

The function [cvRunningAvg](#) calculates weighted sum of two images. Once a statistical model is available, there is often a need to update the value slowly to account for slowly changing lighting, etc. This can be done by using a simple adaptive filter:

$$\mu_t = \alpha y + (1 - \alpha)\mu_{t-1},$$

where μ (*imgU*) is the updated value, $0 \leq \alpha \leq 1$ is an averaging constant, typically set to a small value such as 0.05, and y (*imgY*) is a new observation at time t . When the function is applied to a frame sequence, the result is called “the running average of the sequence”.

If *mask* is not NULL, it specifies what accumulator pixels are affected.

Distance Transform

10

This chapter describes the distance transform functions group.

Overview

Distance transform is used for calculating the distance to an object. The input is an image with feature and non-feature pixels. The function labels every non-feature pixel in the output image with a distance to the closest feature pixel. Feature pixels are marked with zero. Distance transform is used for a wide variety of subjects including skeleton finding and shape analysis. The [[Borgefors86](#)] two-pass algorithm is implemented.

Reference

cvDistTransform

Calculates distance to closest zero pixel for all non-zero pixels of source image.

```
void cvDistTransform ( IplImage* src, IplImage* dst, CvDisType distType,
                      CvDisMaskType maskType, float* mask);
```

<i>src</i>	Source image.
<i>dst</i>	Output image with calculated distances.
<i>distType</i>	Type of distance; can be CV_DIST_L1, CV_DIST_L2, CV_DIST_C or CV_DIST_USER.

<i>maskType</i>	Size of distance transform mask; can be <code>CV_DIST_MASK_3x3</code> or <code>CV_DIST_MASK_5x5</code> .
<i>mask</i>	Pointer to the user-defined mask used with the distance type <code>CV_DIST_USER</code> .

Discussion

The function [cvDistTransform](#) approximates the actual distance from the closest zero pixel with a sum of fixed distance values: two for 3x3 mask and three for 5x5 mask. [Figure 10-1](#) shows the result of the distance transform of a 7x7 image with a zero central pixel.

Figure 10-1 3x3 Mask

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	0	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

This example corresponds to a 3x3 mask; in case of user-defined distance type the user sets the distance between two pixels, that share the edge, and the distance between the pixels, that share the corner only. For this case the values are 1 and 1.5 correspondingly. [Figure 10-2](#) shows the distance transform for the same image, but for a 5x5 mask. For the 5x5 mask the user sets the additional distance that is the distance

between pixels corresponding to the chess knight move. In this example the additional distance is equal to 2. For `CV_DIST_L1`, `CV_DIST_L2`, and `CV_DIST_C` the optimal precalculated distance values are used.

Figure 10-2 5x5 Mask

4.5	3.5	3	3	3	3.5	4
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	0	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

Threshold Functions

11

This chapter describes threshold functions group.

Overview

Thresholding functions are used mainly for two purposes:

- masking out some pixels that do not belong to a certain range, for example, to extract blobs of certain brightness or color from the image;
- converting grayscale image to bi-level or black-and-white image.

Usually, the resultant image is used as a mask or as a source for extracting higher-level topological information, e.g., contours (see [Active Contours](#)), skeletons (see [Distance Transform](#)), lines (see [Hough Transform](#) functions), etc.

Generally, threshold is a determined function $t(x, y)$ on the image:

$$t(x, y) = \begin{cases} A(p(x, y)), f(x, y, p(x, y)) = true \\ B(p(x, y)), f(x, y, p(x, y)) = false \end{cases}$$

The predicate function $f(x, y, p(x, y))$ is typically represented as $g(x, y) < p(x, y) < h(x, y)$, where g and h are some functions of pixel value and in most cases they are simply constants.

There are two basic types of thresholding operations. The first type uses a predicate function, independent from location, that is, $g(x, y)$ and $h(x, y)$ are constants over the image. However, for concrete image some optimal, in a sense, values for the constants can be calculated using image histograms (see [Histogram](#)) or other statistical criteria

(see [Image Statistics](#)). The second type of the functions chooses $g(x, y)$ and $h(x, y)$ depending on the pixel neighborhood in order to extract regions of varying brightness and contrast.

The functions, described in this chapter, implement both these approaches. They support single-channel images with depth IPL_DEPTH_8U, IPL_DEPTH_8S or IPL_DEPTH_32F and can work in-place.

Reference

cvAdaptiveThreshold

Provides adaptive thresholding binary image.

```
void cvAdaptiveThreshold( IplImage* src, IplImage* dst, double max,
    CvAdaptiveThreshMethod method, CvThreshType type, double* parameters);
```

<i>src</i>	Source image.
<i>dst</i>	Destination image.
<i>max</i>	Max parameter, used with the types CV_THRESH_BINARY and CV_THRESH_BINARY_INV only.
<i>method</i>	Method for the adaptive threshold definition; now CV_STDDEF_ADAPTIVE_THRESH only.
<i>type</i>	Thresholding type; must be one of <ul style="list-style-type: none"> CV_THRESH_BINARY, $val = (val > Thresh ? MAX : 0)$; CV_THRESH_BINARY_INV, $val = (val > Thresh ? 0 : MAX)$; CV_THRESH_TOZERO, $val = (val > Thresh ? val : 0)$; CV_THRESH_TOZERO_INV, $val = (val > Thresh ? 0 : val)$.
<i>parameters</i>	Pointer to the list of method-specific input parameters. For the method CV_STDDEF_ADAPTIVE_THRESH the value <i>parameters</i> [0] is the size of the neighborhood: 1- (3x3), 2- (5x5), or 3- (7x7), and <i>parameters</i> [1] is the value of the minimum variance.

Discussion

The function [cvAdaptiveThreshold](#) calculates the adaptive threshold for every input image pixel and segments image. The algorithm is as follows.

Let $\{f_{ij}\}$, $1 \leq i \leq I$, $1 \leq j \leq J$ be the input image. For every pixel i, j the mean m_{ij} and variance v_{ij} are calculated as follows:

$$m_{ij} = 1/2p \cdot \sum_{s=-p}^p \sum_{t=-p}^p f_{i+s, j+t}, \quad v_{ij} = 1/2p \cdot \sum_{s=-p}^p \sum_{t=-p}^p |f_{i+s, j+t} - m_{ij}|,$$

where $p \times p$ is the neighborhood.

Local threshold for pixel i, j is $t_{ij} = m_{ij} + v_{ij}$ for $v_{ij} > v_{min}$, and $t_{ij} = t_{ij-1}$ for $v_{ij} \leq v_{min}$, where v_{min} is the minimum variance value. If $j = 1$, then $t_{ij} = t_{i-1, j}$, $t_{11} = t_{i_0 j_0}$, where $v_{i_0 j_0} > v_{min}$ and $v_{ij} \leq v_{min}$ for $(i < i_0) \vee ((i = i_0) \wedge (j < j_0))$.

Output segmented image is calculated as in the function [cvThreshold](#).

cvThreshold

Thresholds binary image.

```
void cvThreshold( IplImage* src, IplImage* dst, float thresh, float maxvalue,
CvThreshType type);
```

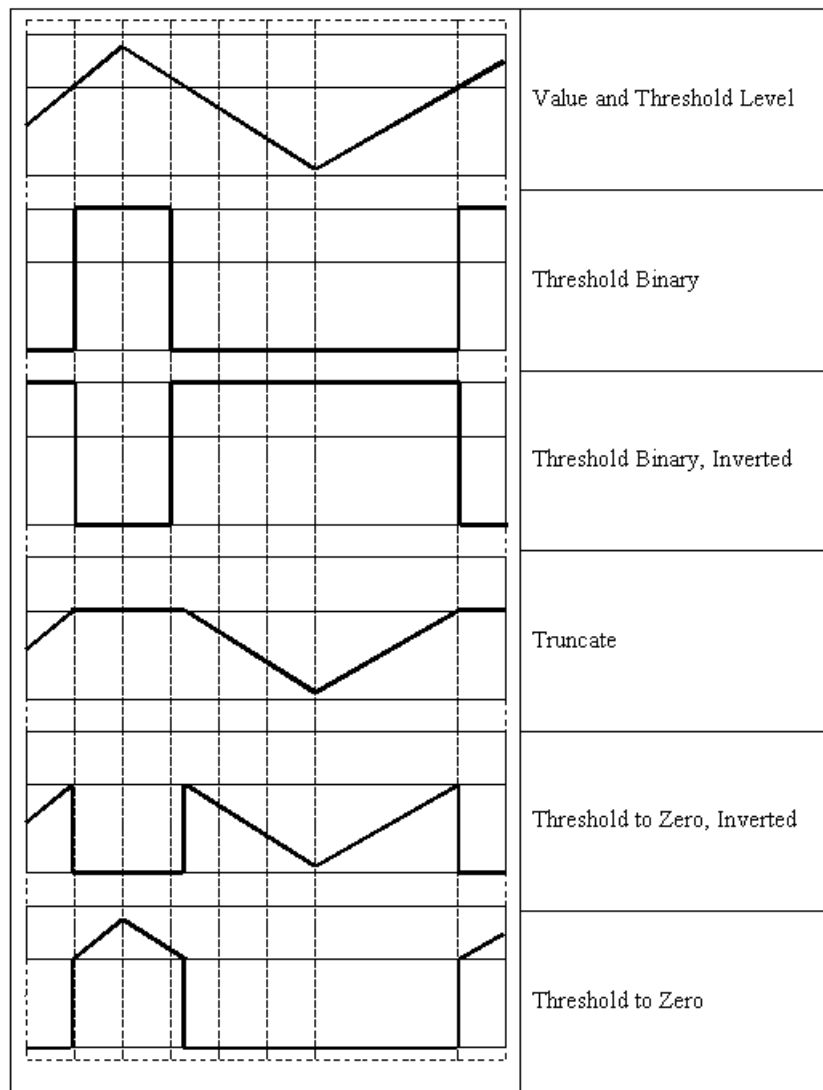
<i>src</i>	Source image.
<i>dst</i>	Destination image; can be the same as the parameter <i>src</i> .
<i>thresh</i>	Threshold parameter.
<i>maxvalue</i>	Maximum value; parameter, used with threshold types CV_THRESH_BINARY, CV_THRESH_BINARY_INV, and CV_THRESH_TRUNC.
<i>type</i>	Thresholding type; must be one of <ul style="list-style-type: none"> CV_THRESH_BINARY, <i>val</i> = (<i>val</i> > <i>thresh</i> <i>maxvalue</i>:0);

- `CV_THRESH_BINARY_INV, val = (val > thresh 0: maxvalue);`
- `CV_THRESH_TRUNC, val = (val > thresh ? thresh: maxvalue);`
- `CV_THRESH_TOZERO, val = (val > thresh val: 0);`
- `CV_THRESH_TOZERO_INV, val = (val > thresh 0: val).`

Discussion

The function [cvThreshold](#) applies fixed-level thresholding to grayscale image. The result is either a grayscale image or a bi-level image. The former variant is typically used to remove noise from the image, while the latter one is used to represent a grayscale image as composition of connected components and after that build contours on the components via the function [cvFindContours](#). [Figure 11-1](#) illustrates meanings of different threshold types:

Figure 11-1 Meanings of Threshold Types



This chapter describes the function performing flood filling of a connected domain.

Overview

Flood filling means that a group of connected pixels with close values is filled with, or is set to, a certain value. The flood filling process starts with some point, called “seed”, that is specified by function caller and then it propagates until it reaches the image ROI boundary or cannot find any new pixels to fill due to a large difference in pixel values. For every pixel that is just filled the function analyses:

- 4 neighbors, that is, excluding the diagonal neighbors; this kind of connectivity is called 4-connectivity, or
- 8 neighbors, that is, including the diagonal neighbors; this kind of connectivity is called 8-connectivity.

The parameter *connectivity* of the function specifies the type of connectivity.

The function can be used for:

- segmenting a grayscale image into a set of uni-color areas,
- marking each connected component with individual color for bi-level images.

The function supports single-channel images with the depth `IPL_DEPTH_8U` or `IPL_DEPTH_32F`.

Reference

cvFloodFill

Makes flood filling of image connected domain.

```
void cvFloodFill( IplImage* img, CvPoint seedPoint, double newVal, double
    loDiff, double upDiff, CvConnectedComp* comp, int connectivity=4 );
```

<i>img</i>	Input image; repainted by the function.
<i>seedPoint</i>	Coordinates of the seed point inside the image ROI.
<i>newVal</i>	New value of repainted domain pixels.
<i>loDiff</i>	Maximal lower difference between the values of pixel belonging to the repainted domain and one of the neighboring pixels to identify the latter as belonging to the same domain.
<i>upDiff</i>	Maximal upper difference between the values of pixel belonging to the repainted domain and one of the neighboring pixels to identify the latter as belonging to the same domain.
<i>comp</i>	Pointer to structure the function fills with the information about the repainted domain.
<i>connectivity</i>	Type of connectivity used within the function. If it is 4, which is default value, the function tries out four neighbors of the current pixel, otherwise the function tries out all the 8 neighbors.

Discussion

The function [cvFloodFill](#) fills the seed pixel neighborhoods inside which all pixel values are close to each other. The pixel is considered to belong to the repainted domain if its value v meets the following conditions:

$$v_0 - d_{lw} \leq v \leq v_0 + d_{up},$$

where v_0 is the value of at least one of the current pixel neighbors, which already belongs to the repainted domain. The function checks 4-connected neighborhoods of each pixel, that is, its side neighbors.

Camera Calibration

13

This chapter describes camera calibration and undistortion functions.

Overview

Camera Parameters

Camera calibration functions are used for calculating intrinsic and extrinsic camera parameters.

Camera parameters are the numbers describing a particular camera configuration. The *intrinsic* camera parameters are those that specify the camera characteristics proper; these parameters include the focal length, that is, the distance between the camera lens and the image plane, the location of the image center in pixel coordinates, the effective pixel size, and the radial distortion coefficient of the lens. The *extrinsic* camera parameters describe the spatial relationship between the camera and the world; they are the rotation matrix and translation vector specifying the transformation between the camera and world reference frames.

A camera is modeled by the usual pinhole: the relationship between a 3D point M and its image projection m is given by the formula

$$m = A[Rt]M,$$

where A is the camera intrinsic matrix:

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

where (c_x, c_y) are coordinates of the principal point;

(f_x, f_y) are the focal lengths by the axes x and y ;

(R, t) are extrinsic parameters, the rotation matrix R and translate vector t that relates the world coordinate system to the camera coordinate system:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \quad t = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}.$$

Camera usually exhibits significant lens distortion, especially radial distortion. The distortion has 4 coefficients: k_1, k_2, k_3, k_4 .

Use the function [cvUndistortInit](#) to correct the camera lens distortion (see [Figure 13-2](#)).

The following algorithm was used for camera calibration:

1. Find homography for all points on series of images.
2. Initialize intrinsic parameters; distortion is set to 0.
3. Find extrinsic parameters for each image of pattern.
4. Make main optimization by minimizing error of projection points with all parameters.

Homography

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \text{ is the matrix of homography.}$$

Without any loss of generality, the model plane may be assumed to be on $z = 0$ of the world coordinate system. If r_i denotes the i^{th} column of the rotation matrix R , then:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A[r_1 \ r_2 \ r_3 \ t] = \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = A[r_1 \ r_2 \ t] \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}.$$

By abuse of notation, M is still used to denote a point on the model plane, but $M = [X, Y]^T$, since Z is always equal to 0. In its turn, $\tilde{M} = [X, Y, 1]^T$. Therefore, a model point M and its image m are related by the homography H :

$$s\tilde{m} = H\tilde{M} \text{ with } H = A[r_1 \ r_2 \ t].$$

As is clear, the 3×3 matrix H is defined without specifying a scalar factor.

Pattern

Calibration may be made using pattern (see [Figure 13-1](#)). Pattern has black and white squares on white background. The geometry of pattern must be known. The pattern may be printed using a high-quality printer and put on a glass substrate.

Figure 13-1 Pattern



Lens Distortion

Any camera usually exhibits significant lens distortion, especially radial distortion. The distortion is described by 4 coefficients: two radial distortion coefficients k_1, k_2 , and two tangential ones p_1, p_2 .

Let (u, v) be true pixel image coordinates, that is, coordinates with ideal projection, and (\tilde{u}, \tilde{v}) be corresponding real observed (distorted) image coordinates. Similarly, (x, y) are ideal (distortion-free) and (\tilde{x}, \tilde{y}) are real (distorted) image physical coordinates. Taking into account two expansion terms gives the following:

$$\begin{aligned}\tilde{x} &= x + x[k_1 r^2 + k_2 r^4] + [2p_1 x y + p_2(r^2 + 2x^2)] \\ \tilde{y} &= y + y[k_1 r^2 + k_2 r^4] + [2p_2 x y + p_1(r^2 + 2y^2)],\end{aligned}$$

where $r^2 = x^2 + y^2$. Second addends in the above relations describe radial distortion and the third ones - tangential. The center of the radial distortion is the same as the principal point. If $\tilde{u} = u_0 + \alpha \tilde{x}$ and $\tilde{v} = v_0 + \beta \tilde{y}$, where c_x, c_y, f_x , and f_y are components of the camera intrinsic matrix, then:

$$\begin{aligned}\tilde{u} &= u + (u - c_x) \left[k_1 r^2 + k_2 r^4 + 2p_1 y + p_2 \left(\frac{r^2}{x} + 2x \right) \right] \\ \tilde{v} &= v + (v - c_y) \left[k_1 r^2 + k_2 r^4 + 2p_2 x + p_1 \left(\frac{r^2}{y} + 2y \right) \right].\end{aligned}$$

The latter relations are the basic ones for the group of undistortion functions.

This group consists of three functions: [cvUndistortOnce](#), [cvUndistortInit](#), and [cvUndistort](#). If only a single image is required to be corrected, [cvUndistortOnce](#) function may be used. When dealing with a number of images possessing similar parameters, e.g., a sequence of video frames, it is better to use the other two functions. In this case the following sequence of actions must take place. First allocate *data* array in the main function; length of this array must be *N* or *3N* elements, where $N = N_x \cdot N_y$ — full number of pixels (see Discussion after [cvUndistortInit](#) description). Then call the function [cvUndistortInit](#) that fills the *data* array. After that call the [cvUndistortInit](#) function for each frame inside the cycle.

Figure 13-2 Correcting Lens Distortion



Image with Lens Distortion



Image with Corrected Lens Distortion

Rotation Matrix and Rotation Vector

Rodrigues conversion function [cvRodrigues](#) is a method to convert rotation vector to rotation matrix or vice versa.

Reference

cvCalibrateCamera

Calibrates camera with single precision.

```
void cvCalibrateCamera( int numImages, int* numPoints, CvSize imageSize,
    CvPoint2D32f* imagePoints32f, CvPoint3D32f* objectPoints32f, CvVect32f
    distortion32f, CvMatr32f cameraMatrix32f, CvVect32f transVects32f,
    CvMatr32f rotMatrs32f, int useIntrinsicGuess);
```

<i>numImages</i>	Number of images.
<i>numPoints</i>	Array of the number of points in each image.
<i>imageSize</i>	Size of image.

<i>imagePoints32f</i>	Pointer to the images.
<i>objectPoints32f</i>	Pointer to the pattern.
<i>distortion32f</i>	Array of four distortion coefficients found.
<i>cameraMatrix32f</i>	Camera matrix found.
<i>transVects32f</i>	Array of translate vectors for each pattern position in the image.
<i>rotMatrs32f</i>	Array of the rotation matrix for each pattern position in the image.
<i>useIntrinsicGuess</i>	Intrinsic guess. If equal to 1, intrinsic guess is needed.

Discussion

The function [cvCalibrateCamera](#) calculates the camera parameters using information points on the pattern object and pattern object images.

cvCalibrateCamera_64d

Calibrates camera with double precision.

```
void cvCalibrateCamera_64d( int numImages, int* numPoints, CvSize imageSize,
    CvPoint2D64d* imagePoints, CvPoint3D64d* objectPoints, CvVect64d
    distortion, CvMatr64d cameraMatrix, CvVect64d transVects, CvMatr64d
    rotMatrs, int useIntrinsicGuess);
```

<i>numImages</i>	Number of images.
<i>numPoints</i>	Array of the number of points in each image.
<i>imageSize</i>	Size of the image.
<i>imagePoints</i>	Pointer to the images.
<i>objectPoints</i>	Pointer to the pattern.
<i>distortion</i>	Distortion coefficients found.
<i>cameraMatrix</i>	Camera matrix found.

<i>transVects</i>	Array of translate vectors for each pattern position on the image.
<i>rotMatrs</i>	Array of the rotation matrix for each pattern position on the image.
<i>useIntrinsicGuess</i>	Intrinsic guess. If equal to 1, intrinsic guess is needed.

Discussion

The function [cvCalibrateCamera_64d](#) is basically the same as the function [cvCalibrateCamera](#), but uses double precision.

cvFindExtrinsicCameraParams

Finds extrinsic camera parameters for pattern.

```
void cvFindExtrinsicCameraParams( int numPoints, CvSize imageSize,
    CvPoint2D32f* imagePoints32f, CvPoint3D32f* objectPoints32f, CvVect32f
    focalLength32f, CvPoint2D32f principalPoint32f, CvVect32f distortion32f,
    CvVect32f rotVect32f, CvVect32f transVect32f);
```

<i>NumPoints</i>	Number of points.
<i>ImageSize</i>	Size of image.
<i>imagePoints32f</i>	Pointer to the image.
<i>objectPoints32f</i>	Pointer to the pattern.
<i>focalLength32f</i>	Focal length.
<i>principalPoint32f</i>	Principal point.
<i>distortion32f</i>	Distortion.
<i>rotVect32f</i>	Rotation vector.
<i>transVect32f</i>	Translate vector.

Discussion

The function [cvFindExtrinsicCameraParams](#) finds the extrinsic parameters for pattern.

cvFindExtrinsicCameraParams_64d

Finds extrinsic camera parameters for pattern with double precision.

```
void cvFindExtrinsicCameraParams_64d( int numPoints, CvSize imageSize,
    CvPoint2D64d* imagePoints, CvPoint3D64d* objectPoints, CvVect64d
    focalLength, CvPoint2D64d principalPoint, CvVect64d distortion, CvVect64d
    rotVect, CvVect64d transVect);
```

<i>NumPoints</i>	Number of points.
<i>ImageSize</i>	Size of image.
<i>imagePoints</i>	Pointer to the image.
<i>objectPoints</i>	Pointer to the pattern.
<i>focalLength</i>	Focal length.
<i>principalPoint</i>	Principal point.
<i>distortion</i>	Distortion.
<i>rotVect</i>	Rotation vector.
<i>transVect</i>	Translate vector.

Discussion

The function [cvFindExtrinsicCameraParams_64d](#) finds the extrinsic parameters for pattern with double precision.

cvRodrigues

Converts rotation matrix to rotation vector and vice versa with single precision.

```
void cvRodrigues( CvMatr32f rotMatr32f, CvVect32f rotVect32f, CvMatr32f  
Jacobian32f, CvRodriguesType convType);
```

rotMatr32f Rotation matrix.

rotVect32f Rotation vector.

Jacobian32f Jacobian matrix 3 x 9.

ConvType Type of conversion; must be CV_RODRIGUES_M2V for converting the matrix to the vector or CV_RODRIGUES_V2M for converting the vector to the matrix.

Discussion

The function [cvRodrigues](#) converts the rotation matrix to the rotation vector or vice versa.

cvRodrigues_64d

Converts rotation matrix to rotation vector and vice versa with double precision.

```
void cvRodrigues_64d( CvMatr64d rotMatr, CvVect64d rotVect, CvMatr64d  
Jacobian, CvRodriguesType convType);
```

rotMatr Rotation matrix.

rotVect Rotation vector.

Jacobian Jacobian matrix 3 x 9.

ConvType Type of conversion must be `CV_RODRIGUES_M2V` for converting the matrix to the vector or `CV_RODRIGUES_V2M` for converting the vector to the matrix.

Discussion

The function [cvRodrigues_64d](#) converts the rotation matrix to the rotation vector or vice versa with double precision.

cvUndistortOnce

Corrects camera lens distortion.

```
void cvUndistortOnce ( IplImage* srcImage, IplImage* dstImage, float*
intrMatrix, float* distCoeffs, int interpolate=1 );
```

srcImage Source (distorted) image.
dstImage Destination (corrected) image.
intrMatrix Matrix of the camera intrinsic parameters.
distCoeffs Vector of the 4 distortion coefficients k_1 , k_2 , p_1 and p_2 .
interpolate Interpolation toggle (optional).

Discussion

The function [cvUndistortOnce](#) corrects camera lens distortion using known matrix of the camera intrinsic parameters and distortion coefficients. It is used if a single image is to be corrected.

Preliminarily, the function [cvCalibrateCamera](#) calculates matrix of the camera intrinsic parameters and distortion coefficients k_1 , k_2 , p_1 and p_2 .

If *interpolate* = 0, inter-pixel interpolation is disabled; otherwise default bilinear interpolation is used.

cvUndistortInit

Calculates arrays of distorted points indices and interpolation coefficients.

```
void cvUndistortInit ( IplImage* srcImage, float* IntrMatrix, float*  
distCoeffs, int* data, int interpolate=1 );
```

<i>srcImage</i>	Source (distorted) image.
<i>intrMatrix</i>	Matrix of the camera intrinsic parameters.
<i>distCoeffs</i>	Vector of the 4 distortion coefficients k_1 , k_2 , p_1 and p_2 .
<i>data</i>	Distortion data array.
<i>interpolate</i>	Interpolation toggle (optional).

Discussion

The function [cvUndistortInit](#) calculates arrays of distorted points indices and interpolation coefficients using known matrix of the camera intrinsic parameters and distortion coefficients. It must be used before calling the function [cvUndistort](#).

Preliminarily, the function [cvCalibrateCamera](#) calculates matrix of the camera intrinsic parameters and distortion coefficients k_1 , k_2 , p_1 and p_2 .

The *data* array must be allocated in the main function before use of the function [cvUndistortInit](#). If *interpolate* = 0, its length must be *size.width*size.height* elements; otherwise $3*size.width*size.height$ elements.

If *interpolate* = 0, inter-pixel interpolation is disabled; otherwise default bilinear interpolation is used.

cvUndistort

Corrects camera lens distortion.

```
void cvUndistort ( IplImage* srcImage, IplImage* dstImage, int* data, int
interpolate=1 );
```

<i>srcImage</i>	Source (distorted) image.
<i>dstImage</i>	Destination (corrected) image.
<i>data</i>	Distortion data array.
<i>interpolate</i>	Interpolation toggle (optional).

Discussion

The function [cvUndistort](#) corrects camera lens distortion using previously calculated arrays of distorted points indices and undistortion coefficients. It is used if a sequence of frames must be corrected.

Preliminarily, the function [cvUndistortInit](#) calculates the array *data* .

If *interpolate* = 0, then inter-pixel interpolation is disabled; otherwise bilinear interpolation is used. In the latter case the function acts slower, but quality of the corrected image increases.

cvFindChessBoardCornerGuesses

*Finds approximate positions of internal corners
of the chessboard.*

```
int cvFindChessBoardCornerGuesses(IplImage* img, IplImage* thresh, CvSize
etalonSize, CvPoint2D32f* corners, int *cornerCount );
```

<i>img</i>	Source chessboard view; must have depth of IPL_DEPTH_8U.
<i>thresh</i>	Temporary image of the same size and format as the source image.

<i>etalonSize</i>	Number of inner corners per chessboard row and column. Width (number of columns) must be less or equal to height (number of rows). For chessboard see Figure 13-1 .
<i>corners</i>	Pointer to the corner array found.
<i>cornerCount</i>	Signed value whose absolute value is a number of corners found. A positive number means that a whole chessboard has been found and a negative number means that not all the corners have been found.

Discussion

The function [cvFindChessBoardCornerGuesses](#) attempts to determine whether the input image is a chessboard pattern and locate internal chessboard corners. The function returns non-zero value if all 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 corners or reorder them, the function returns 0. For example, a simple chessboard has 8×8 squares and 7×7 internal corners, that is, points, where the squares are tangent. The word “approximate” in the above description means that the corner coordinates found may differ from the actual coordinates by a couple of pixels. To get more precise coordinates, the user may use the function [cvFindCornerSubPix](#).

This chapter describes functions for morphing views from two cameras.

Overview

The View Morphing technique is used to get image from a virtual camera that can be placed between two real cameras. The input for View Morphing algorithms are two images from real cameras and information about correspondence between regions in the two images. The output of the algorithms is a synthesized image - "view from virtual camera".

This section addresses the problem of synthesizing images of real scenes under three-dimensional transformation in viewpoint and appearance. Solving this problem enables interactive viewing of remote scenes on a computer, in which a user can move the virtual camera through the environment. The point to make here is that a three-dimensional scene transformation can be rendered on a video display device by applying simple transformation to a set of basis images of the scene. The virtue of these transformations is that they operate directly on the image and recover only the scene information that is required to accomplish the desired effect. Consequently, the transformations are applicable in a situation when accurate three-dimensional models are difficult or impossible to obtain.

A central topic is the problem of view synthesis, that is, rendering images of a real scene from different camera viewpoints by processing a set of basis images.

Algorithm

1. Find fundamental matrix, for example, using correspondence points in images.
2. Find scanlines for each image.
3. Warp images across scanlines.

4. Find correspondence of warped images.
5. Morph warped images across position of virtual camera.
6. Unwarp image.
7. Delete moire from resulting image.

Figure 14-1 Original Images

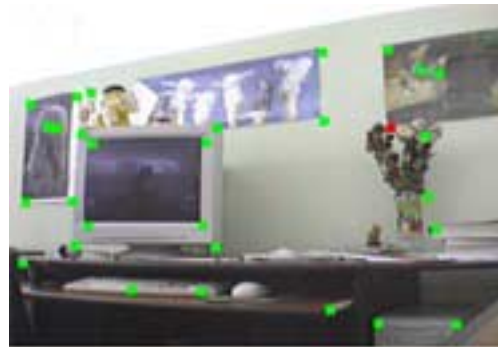
Original Image from Left Camera



Original Image from Right Camera

Figure 14-2 Correspondence Points

Correspondence Points on Left Image



Correspondence Points on Right Image

Figure 14-3 Scan Lines

Some Scanlines on Left Image



Some Scanlines on Right Image

Figure 14-4 Moire in Morphed Image

Figure 14-5 Resulting Morphed Image

Morphed Image from Virtual Camera with Deleted Moire.

Using Functions for View Morphing Algorithm

1. Find the fundamental matrix using the correspondence points in the two images of cameras by calling the function [cvFindFundamentalMatrix](#).
2. Find the number of scanlines in the images for the given fundamental matrix by calling the function [cvFindFundamentalMatrix](#) with null pointers to the scanlines.
3. Allocate enough memory for:
 - scanlines in the first image, scanlines in the second image, scanlines in the virtual image (for each `numscan*2*4*sizeof(int)`);
 - lengths of scanlines in the first image, lengths of scanlines in the second image, lengths of scanlines in the virtual image (for each `numscan*2*4*sizeof(int)`);
 - buffer for the prewarp first image, the second image, the virtual image (for each `width*height*2*sizeof(int)`);
 - data runs for the first image and the second image (for each `width*height*4*sizeof(int)`);
 - correspondence data for the first image and the second image (for each `width*height*2*sizeof(int)`);

- numbers of lines for the first and second images (for each $width*height*4*sizeof(int)$).
- 4. Find scanlines coordinates by calling the function [cvFindFundamentalMatrix](#).
- 5. Prewarp the first and second images using scanlines data by calling the function [cvPreWarpImage](#).
- 6. Find runs on the first and second images scanlines by calling the function [cvFindRuns](#).
- 7. Find correspondence information by calling the function [cvDynamicCorrespondMulti](#).
- 8. Find coordinates of scanlines in the virtual image for the virtual camera position $alpha$ by calling the function [cvMakeAlphaScanlines](#).
- 9. Morph the prewarp virtual image from the first and second images using correspondence information by calling the function [cvMorphEpilinesMulti](#).
- 10. Postwarp the virtual image by calling the function [cvPostWarpImage](#).
- 11. Delete moire from the resulting virtual image by calling the function [cvDeleteMoire](#).

Reference

cvFindFundamentalMatrix

Finds fundamental matrix from correspondence pair points in two images.

```
void cvFindFundamentalMatrix( int* points1, int* points2, int numpoints, int method, CvMatrix3* matrix);
```

<i>points1</i>	Pointer to the array of correspondence points in the first image.
<i>points2</i>	Pointer to the array of correspondence points in the second image.
<i>numpoints</i>	Number of point pairs.

<i>method</i>	Method for finding the fundamental matrix; currently not used, must be zero.
<i>matrix</i>	Resulting fundamental matrix.

Discussion

The function [cvFindFundamentalMatrix](#) finds the fundamental matrix from correspondence pair points in two images. If the number of points is too small or the point positions are not good, that is, they lie very close or on the same planar surface, the matrix is not found correctly.

cvMakeScanlines

Calculates scanlines coordinates for two cameras by fundamental matrix.

```
void cvMakeScanlines( CvMatrix3* matrix, CvSize imgSize, int* scanlines_1,
int* scanlines_2, int* lens_1, int* lens_2, int* numlines);
```

<i>matrix</i>	Fundamental matrix.
<i>imgSize</i>	Size of the image.
<i>scanlines_1</i>	Pointer to the array of calculated scanlines of the first image.
<i>scanlines_2</i>	Pointer to the array of calculated scanlines of the second image.
<i>lens_1</i>	Pointer to the array of calculated lengths (in pixels) of the first image scanlines.
<i>lens_2</i>	Pointer to the array of calculated lengths (in pixels) of the second image scanlines.
<i>numlines</i>	Pointer to the variable that stores the number of scanlines.

Discussion

The function [cvMakeScanlines](#) finds coordinates of scanlines for two images.

This function returns the number of scanlines. The function does nothing except calculating the number of scanlines if the pointers *scanlines_1* or *scanlines_2* are equal to zero.

Memory for all arrays must be allocated before calling this function. Let *numscan* be the number of scanlines. Memory must be allocated for:

1. Scanlines in the first image, scanlines in the second image, and scanlines in the virtual image (for each $numscan * 2 * 4 * \text{sizeof}(\text{int})$).
2. Lengths of scanlines in the first image, lengths of scanlines in the second image, and lengths of scanlines in the virtual image (for each $numscan * 2 * 4 * \text{sizeof}(\text{int})$).
3. Buffer for the prewarp of first image, second image, and virtual image (for each $width * height * 2 * \text{sizeof}(\text{int})$).
4. Data runs for the first and second images (for each $width * height * 4 * \text{sizeof}(\text{int})$).
5. Correspondence data for the first image and the second image (for each $width * height * 2 * \text{sizeof}(\text{int})$).

cvPreWarpImage

Finds prewarp of given image.

```
void cvPreWarpImage( int numLines, IplImage* img, uchar* dst, int* dstNums,
                    int* scanlines);
```

<i>numLines</i>	Number of scanlines for the image.
<i>img</i>	Image to prewarp.
<i>dst</i>	Data to store for the prewarp image.
<i>dstNums</i>	Pointer to the array of lengths of scanlines.
<i>scanlines</i>	Pointer to the array of coordinates of scanlines.

Discussion

The function [cvPreWarpImage](#) finds prewarp of the given image across scanlines. Memory must be allocated before calling this function. Memory size is $\max(\text{width}, \text{height}) * \text{numscanlines} * \text{size}(\text{char}) * 3$.

cvFindRuns

Finds runs in two prewarp images.

```
void cvFindRuns( int numLines, uchar* prewarp_1, uchar* prewarp_2, int*
    lineLens_1, int* lineLens_2, int* runs_1, int* runs_2, int* numRuns_1,
    int* numRuns_2);
```

<i>numLines</i>	Number of scanlines.
<i>prewarp_1</i>	Prewarp data of the first image.
<i>prewarp_2</i>	Prewarp data of the second image.
<i>lineLens_1</i>	Array of lengths of scanlines in the first image.
<i>lineLens_2</i>	Array of lengths of scanlines in the second image.
<i>runs_1</i>	Array of runs in each scanline in the first image.
<i>runs_2</i>	Array of runs in each scanline in the second image.
<i>numRuns_1</i>	Array of numbers of runs in each scanline in the first image.
<i>numRuns_2</i>	Array of numbers of runs in each scanline in the second image.

Discussion

The function [cvFindRuns](#) finds runs in the two prewarp images. Memory must be allocated before calling this function. Memory size for one array of runs is $\max(\text{width}, \text{height}) * \text{numscanlines} * 3 * \text{sizeof}(\text{int})$.

cvDynamicCorrespondMulti

Finds correspondence between two sets of runs of two warped images.

```
void cvDynamicCorrespondMulti( int   lines, int* first, int* firstRuns, int*
                             second, int* secondRuns, int* firstCorr, int* secondCorr);
```

<i>lines</i>	Number of scanlines.
<i>first</i>	Array of runs of the first image.
<i>firstRuns</i>	Array of numbers of runs in each scanline of the first image.
<i>second</i>	Array of runs of the second image.
<i>secondRuns</i>	Array of numbers of runs in each scanline of the second image.
<i>firstCorr</i>	Array of find correspondence information for the first image.
<i>secondCorr</i>	Array of find correspondence information for the second image.

Discussion

The function [cvDynamicCorrespondMulti](#) finds correspondence between two sets of runs of two images. The function finds runs in the two prewarp images. Memory must be allocated before calling this function. Memory size for one array of correspondence information is `max(width,height)*numscanlines*3*sizeof(int)`.

cvMakeAlphaScanlines

Finds coordinates of scanlines for image for virtual camera position.

```
void cvMakeAlphaScanlines( int* scanlines_1, int* scanlines_2, int*
                          scanlinesA, int* lens, int numlines, float alpha);
```

<i>scanlines_1</i>	Pointer to the array of the first scanlines.
<i>scanlines_2</i>	Pointer to the array of the second scanlines.

<i>scanlinesA</i>	Pointer to the array of the scanlines found in the virtual image.
<i>lens</i>	Pointer to the array of lengths of the scanlines found in the virtual image.
<i>numlines</i>	Number of scanlines.
<i>alpha</i>	Position of virtual camera (0.0 - 1.0).

Discussion

The function [cvMakeAlphaScanlines](#) finds coordinates of scanlines for the virtual camera with the given camera position.

Memory must be allocated before calling this function. Memory size for the array of correspondence runs is $numscanlines * 2 * 4 * sizeof(int)$. Memory size for the array of the scanline lengths is $numscanlines * 2 * 4 * sizeof(int)$.

cvMorphEpilinesMulti

Morphs two prewarp images using corresponding information.

```
void cvMorphEpilinesMulti( int lines, uchar* firstPix, int* firstNum, uchar*
    secondPix, int* secondNum, uchar* dstPix, int* dstNum, float alpha, int*
    first, int* firstRuns, int* second, int* secondRuns, int* firstCorr, int*
    secondCorr);
```

<i>lines</i>	Number of scanlines in the prewarp image.
<i>firstPix</i>	Pointer to the first prewarp image.
<i>firstNum</i>	Pointer to the array of numbers of points in each scanline in the first image.
<i>secondPix</i>	Pointer to the second prewarp image.
<i>secondNum</i>	Pointer to the array of numbers of points in each scanline in the second image.
<i>dstPix</i>	Pointer to the resulting morphed warped image.
<i>dstNum</i>	Pointer to the array of numbers of points in each line.

<i>alpha</i>	Virtual camera position (0.0 - 1.0).
<i>first</i>	First sequence of runs.
<i>firstRuns</i>	Pointer to the number of runs in each scanline in the first image.
<i>second</i>	Second sequence of runs.
<i>secondRuns</i>	Pointer to the number of runs in each scanline in the second image.
<i>firstCorr</i>	Pointer to the array of correspondence information found for the first runs.
<i>secondCorr</i>	Pointer to the array of correspondence information found for the second runs

Discussion

The function [cvMorphEpilinesMulti](#) morphs two prewarp images using corresponding information.

cvPostWarpImage

Finds postwarp for given image data.

```
void cvPostWarpImage( int numLines, uchar* src, int* srcNums, IplImage* img,
    int* scanlines);
```

<i>numLines</i>	Number of scanlines.
<i>src</i>	Pointer to the prewarp image virtual image.
<i>srcNums</i>	Number of scanlines in the image.
<i>img</i>	Resulting unwarp image.
<i>scanlines</i>	Pointer to the array of scanlines data.

Discussion

The function [cvPostWarpImage](#) finds postwarp for the given image data.

cvDeleteMoire

Deletes moire in given image.

```
void cvDeleteMoire( IplImage* img);  
    img            Image.
```

Discussion

The function [cvDeleteMoire](#) deletes moire from the given image. The post-morphing post-warped image has black points: the postwarped image is created by lines, which means that every point may not be filled. The function deletes moire (black points) from the given image by the color of neighbor points. If all scanlines are horizontal, this function may be omitted.

Motion Templates

15

This chapter describes Motion Templates functions.

Overview

The functions described in this section are designed to generate motion template images that can be used to rapidly determine where a motion occurred, how it occurred, and in which direction it occurred. The algorithms are based on papers by Davis and Bobick [[Davis97](#)] and Bradski and Davis [[Bradsky00](#)]. These functions operate on images that are the output of frame or background differencing, or other image segmentation operations; thus the input and output image types are all grayscale, that is, one color channel.

Motion Representation and Normal Optical Flow Method

Motion Representation

[Figure 15-1](#) (left) shows capturing a foreground silhouette of the moving object or person. Obtaining a clear silhouette is achieved through application of some of background subtraction techniques briefly described in [Overview](#) of the chapter on Background Subtraction. As the person or object moves, copying the most recent foreground silhouette as the highest values in the motion history image creates a “layered history” of the resulting motion; typically this “highest value” is just a floating point timestamp of time elapsing since the code was run in milliseconds. [Figure 15-1](#) (right) shows the result that may be called the *Motion History Image (MHI)*. A pixel level or a time delta threshold, as appropriate, is set such that pixel values in the *MHI* image that fall below that threshold are set to zero.

Figure 15-1 Motion History Image from Moving Silhouette

The most recent motion has the highest value, earlier motions have decreasing values subject to a threshold below which the value is set to zero. Different stages of creating and processing motion templates are described below.

A) Updating MHI Images

Generally, we work with floating point images since we read system time differences in milliseconds from application launch time, convert the time differences into a floating point number and use that number as the value of our most recent silhouette. We write this current silhouette over the past silhouettes and threshold away pixels that are too old (beyond a maximum *mhiDuration*) to create the *Motion History Image (MHI)*.

B) Making Motion Gradient Image

1. Start with the MHI image as shown in [Figure 15-1](#)(left).
2. Apply 3×3 Sobel operators x and y to the image.

3. If the resulting response at a pixel location (x, y) is $S_x(x, y)$ to the Sobel operator x and $S_y(x, y)$ to the operator y , then the orientation of the gradient is calculated as:

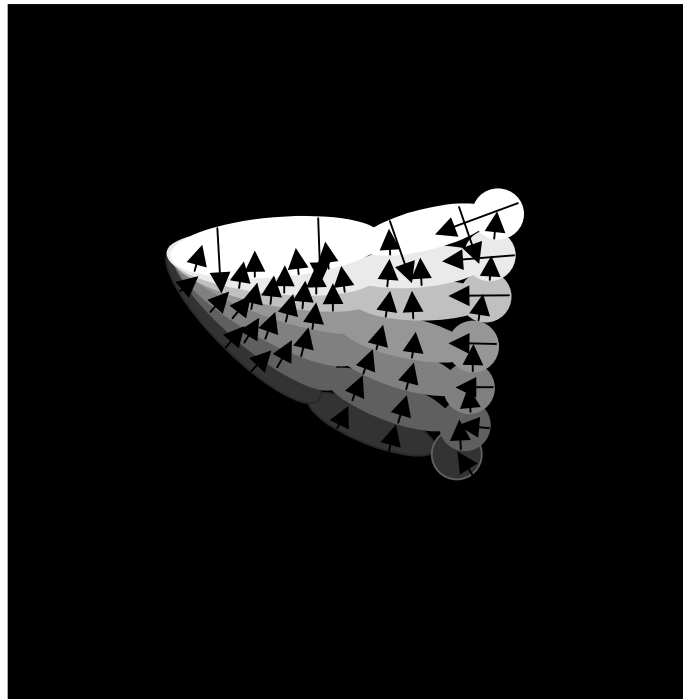
$$A(x, y) = \arctan(S_y(x, y) / S_x(x, y)),$$

and the magnitude of the gradient is:

$$M(x, y) = \sqrt{S_x^2(x, y) + S_y^2(x, y)}.$$

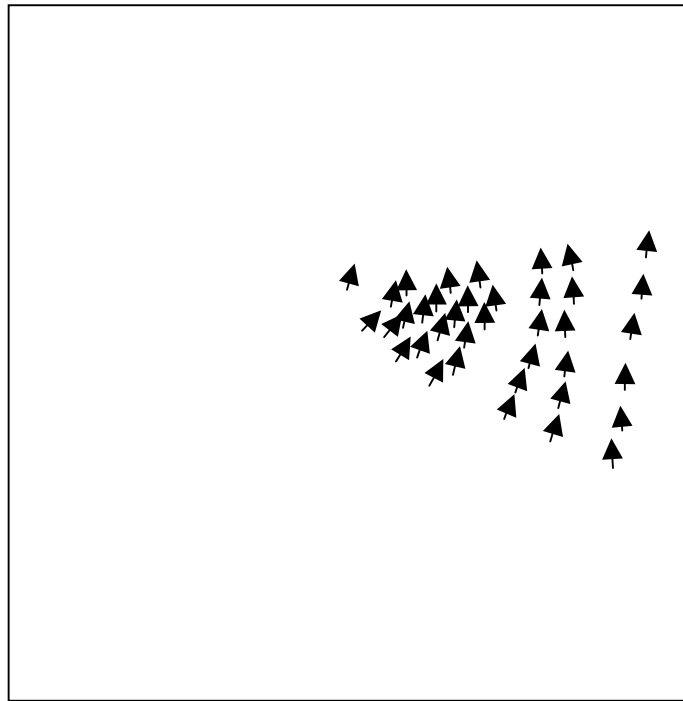
4. The equations are applied to the image yielding direction or angle of flow image superimposed (just for reference) over the *MHI* image as shown in [Figure 15-2](#).

Figure 15-2 Direction of Flow Image



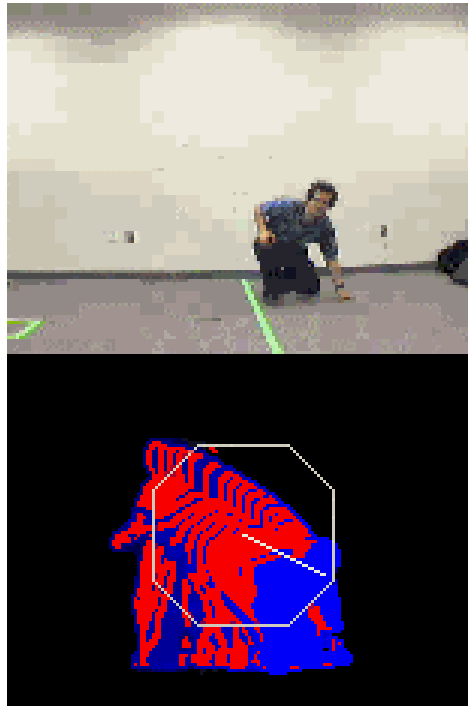
5. The boundary pixels of the *MH* region may give incorrect motion angles and magnitudes, as [Figure 15-2](#) shows. Thresholding away magnitudes that are either too large or too small can be a remedy in this case. [Figure 15-3](#) shows the ultimate results.

Figure 15-3 Resulting Normal Motion Directions.



C) Finding Regional Orientation or Normal Optical Flow

[Figure 15-4](#) shows the output of the motion gradient function described in the section above together with the marked direction of motion flow.

Figure 15-4 MHI Image of Kneeling Person

The current silhouette is in bright blue with past motions in dimmer and dimmer blue. Red lines show where valid normal flow gradients were found. The white line shows computed direction of global motion weighted towards the most recent direction of motion.

To determine the most recent, salient global motion:

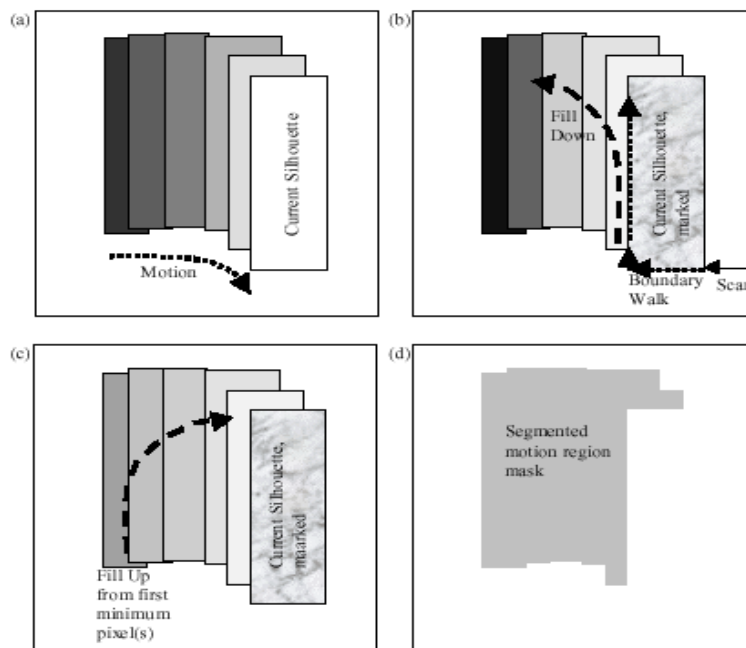
1. Calculate a histogram of the motions resulting from processing (see [Figure 15-3](#)).
2. Find the average orientation of a circular function: angle in degrees.
 - a. Find the maximal peak in the orientation histogram.
 - b. Find the average of minimum differences from this base angle. The more recent movements are taken with larger weights.

Motion Segmentation

Usually, it is not necessary to calculate the motion orientation for the whole image. So certain motion regions, produced by the movement of parts or the whole of the object of interest, may be grouped. Using then a downward stepping floodfill to label motion regions connected to the current silhouette helps identify areas of motion directly attached to parts of the object of interest.

Once *MHI* image is constructed, the most recent silhouette acquires the maximal values, e.g., most recent timestamp, in that image. The image is scanned until this value is found, and then walking along the silhouette's contour helps find attached areas of motion. The algorithm for creating masks to segment motion region is as follows:

1. Scan the *MHI* until finding a pixel of the current timestamp (most recent silhouette), mark that region by a floodfill (see [Figure 15-5 \(a\)](#));
2. Walk around the boundary of the current silhouette region looking outside for recent (within a threshold) unmarked motion history "steps". When a suitable step is found, mark it with a downward floodfill. If the size of the fill is not big enough, zero out the area (see [Figure 15-5 \(b\)](#)).
3. [Optional]:
 - Record locations of minimums (or record locations of predetermined values) within each downfill (see [Figure 15-5 \(c\)](#));
 - Perform separate floodfills up from each detected location (see [Figure 15-5 \(d\)](#));
 - Combine separately (by logical AND) each upfill with downfill it belonged to.
4. Store the detected segmented motion regions into the mask.
5. Continue the boundary "walk" until the silhouette has been circumnavigated.
6. [Optional] Go to 1 until all current silhouette regions are found.

Figure 15-5 Creating Masks to Segment Motion Region

The functions that do all of the above are described below.

Reference

cvUpdateMotionHistory

Updates motion history image.

```
void cvUpdateMotionHistory (IplImage* silhouette, IplImage* mhi, double
    timestamp, double mhiDuration);
```

<i>silhouette</i>	Silhouette image that has non-zero pixels where the motion occurs.
<i>mhi</i>	Motion history image, both an input and output parameter.
<i>timestamp</i>	Floating point current time in milliseconds.
<i>mhiDuration</i>	Maximal duration of motion track in milliseconds.

Discussion

The function [cvUpdateMotionHistory](#) updates the motion history image with silhouette of floating point current system time, assigning the current *timestamp* value to those *mhi* pixels that have corresponding non-zero silhouette pixels. The function also clears *mhi* pixels older than $timestamp - mhiDuration$ if the corresponding silhouette values are 0.

cvCalcMotionGradient

Calculates gradient orientation of motion history image.

```
void cvCalcMotionGradient( IplImage* mhi, IplImage* mask, IplImage*
    orientation, double maxTDelta, double minTDelta, int apertureSize=3 );
```

<i>mhi</i>	Motion history image.
<i>mask</i>	Mask image; marks pixels where motion gradient data is correct. Output parameter.

<i>orientation</i>	Motion gradient orientation image; contains angles from 0 to ~360 degrees.
<i>apertureSize</i>	Size of aperture used to calculate derivatives. Value should be odd, e.g., 3, 5, etc.
<i>maxTDelta</i>	Upper threshold. The function considers the gradient orientation valid if the difference between the maximum and minimum <i>mhi</i> values within a pixel neighborhood is lower than this threshold.
<i>minTDelta</i>	Lower threshold. The function considers the gradient orientation valid if the difference between the maximum and minimum <i>mhi</i> values within a pixel neighborhood is greater than this threshold.

Discussion

The function [cvCalcMotionGradient](#) calculates the derivatives D_x and D_y for the image *mhi* and then calculates orientation of the gradient using the formula

$$\varphi = \begin{cases} 0, & x = 0, y = 0 \\ \arctan(y/x) & \text{else} \end{cases}$$

Finally, the function masks off pixels with a very small (less than *minTDelta*) or very large (greater than *maxTDelta*) difference between the minimum and maximum *mhi* values in their neighborhood. The neighborhood for determining the minimum and maximum has the same size as aperture for derivative kernels - *apertureSize* \times *apertureSize* pixels.

cvCalcGlobalOrientation

Calculates global motion orientation of some selected region.

```
void cvCalcGlobalOrientation( IplImage* orientation, IplImage* mask, IplImage*
    mhi, double curr_mhi_timestamp, double mhiDuration );
```

<i>orientation</i>	Motion gradient orientation image; calculated by the function cvCalcMotionGradient .
<i>mask</i>	Mask image. It is a conjunction of valid gradient mask, calculated by the function cvCalcMotionGradient and mask of the region, whose direction needs to be calculated.
<i>mhi</i>	Motion history image.
<i>curr_mhiTimestamp</i>	Current time in milliseconds.
<i>mhiDuration</i>	Maximal duration of motion track in milliseconds.

Discussion

The function [cvCalcGlobalOrientation](#) calculates the general motion direction in the selected region.

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 orientation vectors (the more recent is the motion, the greater is the weight). The resultant angle is `<basic orientation> + <shift>`.

cvSegmentMotion

Segments whole motion into separate moving parts.

```
void cvSegmentMotion( IplImage* mhi, IplImage* segMask, CvMemStorage* storage,
    CvSeq** components, double timestamp, double segThresh );
```

<i>mhi</i>	Motion history image.
<i>segMask</i>	Image where the mask found should be stored.
<i>Storage</i>	Pointer to the memory storage, where the sequence of components should be saved.
<i>components</i>	Sequence of components found by the function.
<i>timestamp</i>	Floating point current time in milliseconds.

segThresh Segmentation threshold; recommended to be equal to the interval between motion history “steps” or greater.

Discussion

The function [cvSegmentMotion](#) finds all the motion segments, starting from connected components in the image `mhi` that have value of the current timestamp. Each of the resulting segments is marked with an individual value (1,2 ...).

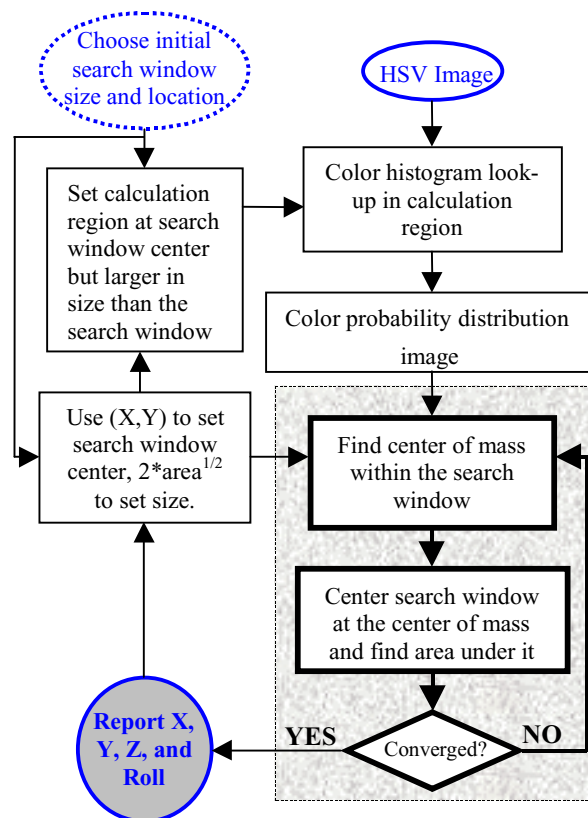
The function stores information about each resulting motion segment in the structure `CvConnectedComp`. The function returns a sequence of such structures.

This chapter describes CamShift algorithm realization functions.

Overview

CamShift stands for the “Continuously Adaptive Mean-SHIFT” algorithm. [Figure 16-1](#) summarizes the CamShift algorithm. For each video frame, the raw image is converted to a color probability distribution image via a color histogram model of the color being tracked (flesh for face tracking). The center and size of the color object are found via the CamShift algorithm operating on the color probability image. The current size and location of the tracked object are reported and used to set the size and location of the search window in the next video image. The process is then repeated for continuous tracking. The algorithm is a generalization of the Mean Shift algorithm, highlighted in gray in [Figure 16-1](#).

Figure 16-1 Block Diagram of CamShift Algorithm



CamShift operates on a 2D color probability distribution image produced from histogram back-projection (see [Histogram](#), this document). The core part of the CamShift algorithm is the Mean Shift algorithm.

The Mean Shift part of the algorithm (gray area in [Figure 16-1](#)) is as follows:

1. Choose the search window size.
2. Choose the initial location of the search window.
3. Compute the mean location in the search window.
4. Center the search window at the mean location computed in Step 3.

5. Repeat Steps 3 and 4 until the search window center converges, i.e., until it has moved for a distance less than the preset threshold.

Mass Center Calculation for 2D Probability Distribution

For discrete 2D image probability distributions, the mean location (the centroid) within the search window (Steps 3 and 4 above) is found as follows:

Find the zeroth moment

$$M_{00} = \sum_x \sum_y I(x, y).$$

Find the first moment for x and y

$$M_{10} = \sum_x \sum_y x I(x, y); M_{01} = \sum_x \sum_y y I(x, y).$$

Mean search window location (the centroid) then is found as

$$x_c = \frac{M_{10}}{M_{00}}; y_c = \frac{M_{01}}{M_{00}},$$

where $I(x, y)$ is the pixel (probability) value in the position (x, y) in the image, and x and y range over the search window.

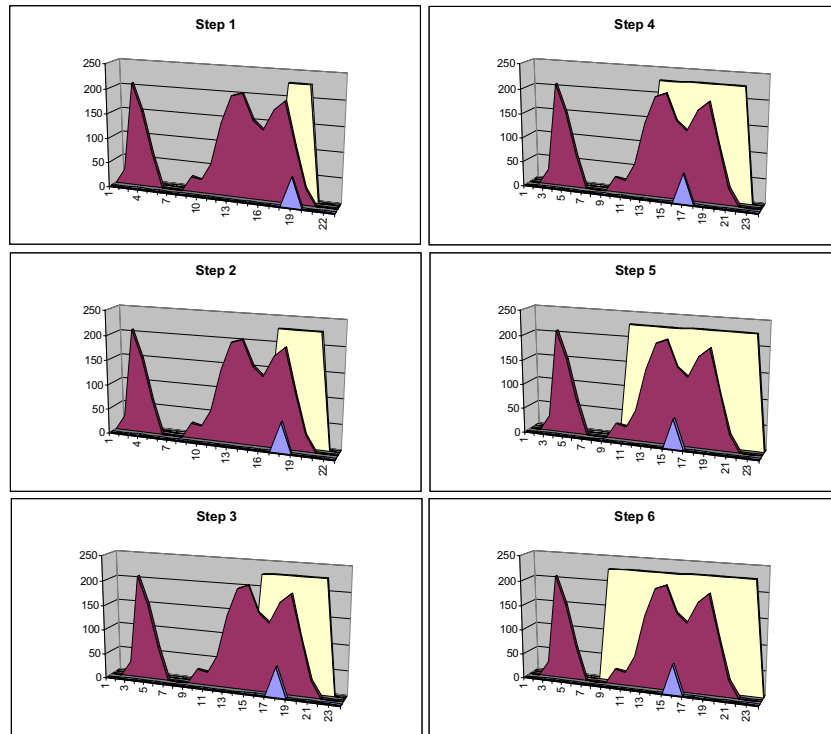
Unlike the Mean Shift algorithm, which is designed for static distributions, CamShift is designed for dynamically changing distributions. These occur when objects in video sequences are being tracked and the object moves so that the size and location of the probability distribution changes in time. The CamShift algorithm adjusts the search window size in the course of its operation. Initial window size can be set at any reasonable value. For discrete distributions (digital data), the minimum window length or width is three. Instead of a set, or externally adapted window size, CamShift relies on the zeroth moment information, extracted as part of the internal workings of the algorithm, to continuously adapt its window size within or over each video frame.

CamShift Algorithm

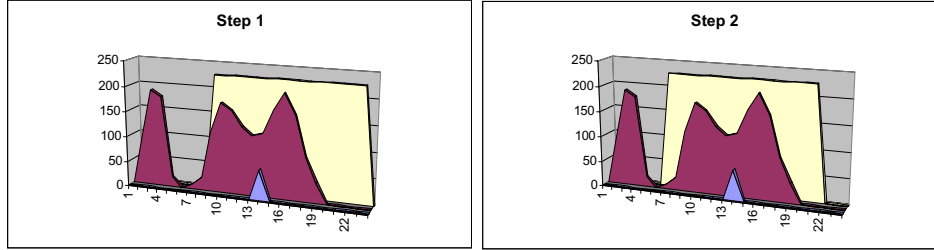
1. Set the calculation region of the probability distribution to the whole image.
2. Choose the initial location of the 2D mean shift search window.

3. Calculate the color probability distribution in the 2D region centered at the search window location in an ROI slightly larger than the mean shift window size.
4. Run mean shift algorithm to find the search window center. Store the zeroth moment (area or size) and center location.
5. For the next video frame, center the search window at the mean location stored in Step 4 and set the window size to a function of the zeroth moment found there. Go to Step 3.

[Figure 16-2](#) shows CamShift finding the face center on a 1D slice through a face and hand flesh hue distribution. [Figure 16-3](#) shows the next frame when the face and hand flesh hue distribution has moved, and convergence is reached in two iterations.

Figure 16-2 Cross Section of Flesh Hue Distribution

Rectangular CamShift window is shown behind the hue distribution, while triangle in front marks the window center. CamShift is shown iterating to convergence down the left then right columns.

Figure 16-3 Flesh Hue Distribution (Next Frame)

Starting from the converged search location in [Figure 16-2](#) bottom right, CamShift converges on new center of distribution in two iterations.

Calculation of 2D Orientation

The 2D orientation of the probability distribution is also easy to obtain by using the second moments in the course of CamShift operation, where (x, y) range over the search window, and $I(x, y)$ is the pixel (probability) value at (x, y) .

Second moments are

$$M_{20} = \sum_x \sum_y x^2 I(x, y), \quad M_{02} = \sum_x \sum_y y^2 I(x, y).$$

Then the object orientation (major axis) is

$$\theta = \frac{\arctan \left(\frac{2 \left(\frac{M_{11}}{M_{00}} - x_c y_c \right)}{\left(\frac{M_{20}}{M_{00}} - x_c^2 \right) - \left(\frac{M_{02}}{M_{00}} - y_c^2 \right)} \right)}{2}.$$

The first two eigenvalues (major length and width) of the probability distribution “blob” found by CamShift may be calculated in closed form as follows. Let

$$a = \frac{M_{20}}{M_{00}} - x_c^2, \quad b = 2 \left(\frac{M_{11}}{M_{00}} - x_c y_c \right), \quad \text{and} \quad c = \frac{M_{02}}{M_{00}} - y_c^2.$$

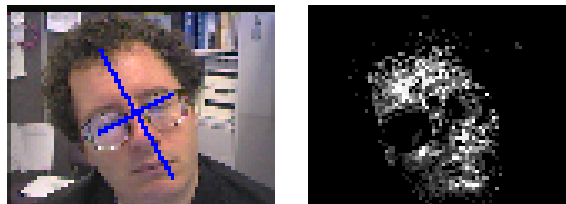
Then length l and width w from the distribution centroid are

$$l = \sqrt{\frac{(a+c) + \sqrt{b^2 + (a-c)^2}}{2}},$$

$$w = \sqrt{\frac{(a+c) - \sqrt{b^2 + (a-c)^2}}{2}}.$$

When used in face tracking, the above equations give head roll, length, and width as marked in the source video image in [Figure 16-4](#).

Figure 16-4 Orientation of Flesh Probability Distribution



Reference

cvCamShift

Finds object center, size, and orientation.

```
int cvCamShift( IplImage* imgProb, CvRect windowIn, CvTermCriteria criteria,
                CvConnectedComp* out, CvBox2D* box=0 );
```

<i>imgProb</i>	2D object probability distribution.
<i>windowIn</i>	Initial search window.
<i>criteria</i>	Criteria applied to determine when the window search should be finished.

<i>out</i>	Resultant structure that contains converged search window coordinates (<i>rect</i> field) and sum of all pixels inside the window (<i>area</i> field).
<i>box</i>	Circumscribed box for the object. If not <code>NULL</code> , contains object size and orientation.

Discussion

The function [cvCamShift](#) finds an object center using the Mean Shift algorithm and, after that, calculates the object size and orientation. The function returns number of iterations made within the Mean Shift algorithm.

cvMeanShift

Iterates to find object center.

```
int cvMeanShift( IplImage* imgProb, CvRect windowIn, CvTermCriteria criteria,  
CvConnectedComp* out );
```

<i>imgProb</i>	2D object probability distribution.
<i>windowIn</i>	Initial search window.
<i>criteria</i>	Criteria applied to determine when the window search should be finished.
<i>out</i>	Resultant structure that contains converged search window coordinates (<i>rect</i> field) and sum of all pixels inside the window (<i>area</i> field).

Discussion

The function [cvMeanShift](#) iterates to find the object center given its 2D color probability distribution image. 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.

Active Contours

17

This chapter describes a function for working with active contours (snakes).

Overview

The snake was presented in [Kass88] as an energy-minimizing parametric closed curve guided by external forces. Energy function associated with the snake $E = E_{int} + E_{ext}$,

where E_{int} is the internal energy formed by the snake configuration, E_{ext} is the external energy formed by external forces affecting the snake. The aim of the snake is to find a location that can minimize the energy.

Let p_1, \dots, p_n be a discrete representation of a snake, that is, a sequence of points on an image plane.

In OpenCV the internal energy function is the sum of the contour continuity energy and the contour curvature energy, as follows:

$E_{int} = E_{cont} + E_{curv}$, where

E_{cont} is the contour continuity energy. This energy is $E_{cont} = |\bar{d} - \|p_i - p_{i-1}\||$, where \bar{d} is the average distance between all pairs $(p_i - p_{i-1})$. Minimizing E_{cont} over all the snake points p_1, \dots, p_n , causes the snake points become more equidistant.

E_{curv} is the contour curvature energy. The smoother the contour is, the less is the curvature energy. $E_{curv} = \|p_{i-1} - 2p_i + p_{i+1}\|^2$.

In [Kass88] external energy was represented as $E_{ext} = E_{img} + E_{con}$, where

E_{img} – image energy and E_{con} – energy of additional constraints.

Two variants of image energy are proposed:

$$1. \quad E_{img} = -I,$$

where I is the image intensity. In this case the snake is attracted to the bright lines of image.

$$2. \quad E_{img} = -\|grad(I)\|. \text{ The snake is attracted to image edges.}$$

A variant of external constraint is described in [Kass88]. Imagine the snake points connected by springs with certain points on the image. Spring force $k(x - x_0)$ will produce the energy $\frac{kx^2}{2}$.

This force pulls snake points to fixed positions, which can be useful when snake points need to be fixed.

OpenCV does not support this opportunity now.

Summary energy at every point can be written as

$$E_i = \alpha_i E_{cont,i} + \beta_i E_{curv,i} + \gamma_i E_{img,i}, \quad (17.1)$$

where α, β, γ are the weights of every kind of energy. The full snake energy is the sum of E_i over all the points.

The meanings of α, β, γ are as follows:

α is responsible for contour continuity, that is, a big α makes snake points more evenly spaced.

β is responsible for snake corners, that is, a big β for a certain point makes the angle between snake edges more obtuse.

γ is responsible for making the snake point more sensitive to the image energy, rather than to continuity or curvature.

Only relative values of α, β, γ in the snake point are relevant.

The following way of working with snakes is proposed:

- create a snake with initial configuration;
- define weights α, β, γ at every point;
- allow the snake to minimize its energy;
- evaluate the snake position. If required, adjust α, β, γ , and possibly image data, and repeat the previous step.

There are three well-known algorithms for minimizing snake energy. In [Kass88] the minimization is based on variational calculus. In [Yuille89] dynamic programming is used. The greedy algorithm is proposed in [Williams92].

The latter algorithm is the most efficient and yields quite good results. The scheme of this algorithm for each snake point is as follows:

- Use [Equation \(17.1\)](#) to compute E for every location from point neighborhood. Before computing E , each energy term E_{cont} , E_{curv} , E_{img} must be normalized using formula $E_{normalized} = (E_{img} - min)/(max - min)$, where max and min are maximal and minimal energy in scanned neighborhood.
- Choose location with minimum energy.
- Move snakes point to this location.
- Repeat all the steps until convergence is reached.

Criteria of convergence are as follows:

- maximum number of iterations is achieved;
- number of points, moved at last iteration, is less than given threshold.

In [Williams92] the authors proposed a way, called high-level feedback, to adjust b coefficient for corner estimation during minimization process. Although this feature is not available in the implementation, the user may build it, if needed.

Reference

cvSnakeImage

Changes contour position to minimize its energy.

```
void cvSnakeImage( IplImage* image, CvPoint* points, int length,
                  float* alpha, float* beta, float* gamma, int coeffUsage, CvSize win,
                  CvTermCriteria criteria, int calcGradient=1 );
```

<i>image</i>	Pointer to the source image.
<i>points</i>	Points of the contour.

<i>length</i>	Number of points in the contour.
<i>alpha</i>	Weight of continuity energy.
<i>beta</i>	Weight of curvature energy.
<i>gamma</i>	Weight of image energy.
<i>coeffUsage</i>	<p>Variant of usage of the previous three parameters:</p> <ul style="list-style-type: none">• <code>CV_VALUE</code> indicates that each of <i>alpha</i>, <i>beta</i>, <i>gamma</i> is pointer to a single value to be used for all points;• <code>CV_ARRAY</code> indicates that each of <i>alpha</i>, <i>beta</i>, <i>gamma</i> is pointer to an array of coefficients different for all the points of the snake. All the arrays must have the size equal to the snake size.
<i>win</i>	Size of neighborhood of every point used to search the minimum; must be odd.
<i>criteria</i>	Termination criteria.
<i>calcGradient</i>	Gradient flag. If not 0, the function counts source image gradient magnitude as external energy, otherwise the image intensity is considered.

Discussion

The function [cvSnakeImage](#) uses image intensity as image energy.

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 the number of moved points is less than `criteria.epsilon` or the function performed `criteria.maxIter` iterations, the function terminates.

This chapter describes functions used for calculation of optical flow implementing Lucas & Kanade, Horn & Schunck, and Block Matching techniques.

Overview

Most papers devoted to motion estimation use the term “*optical flow*”. Optical flow is defined as an apparent motion of image brightness. If $I(x, y, t)$ is the image brightness that changes in time to provide an image sequence, then two main assumptions can be made:

1. Brightness $I(x, y, t)$ depends on coordinates x, y in greater part of the image.
2. Brightness of every point of a moving or static object does not change in time.

Let some object in the image, or some point of an object, move and after time dt the object displacement is (dx, dy) . Using Taylor series for brightness $I(x, y, t)$ gives the following:

$$I(x + dx, y + dy, t + dt) = I(x, y, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt + \dots, \quad (18.1)$$

where “...” are higher order terms.

Next, according to Assumption 2:

$$I(x + dx, y + dy, t + dt) = I(x, y, t), \quad (18.2)$$

and

$$\frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt + \dots = 0. \quad (18.3)$$

Dividing (18.3) by dt and defining

$$\frac{dx}{dt} = u, \quad \frac{dy}{dt} = v \quad (18.4)$$

gives an equation

$$-\frac{\partial i}{\partial t} = \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v, \quad (18.5)$$

usually called *optical flow constraint equation*, where u and v are components of optical flow field in x and y coordinates respectively. Since [Equation \(18.5\)](#) has more than one solution, more constraints are required.

Some variants of further steps may be chosen. Below follows a brief overview of the options available.

Lucas & Kanade Technique

Using the optical flow equation for group of adjacent pixels and assuming that all of them have the same velocity, we can make a system of linear equations.

In a non-singular system for two pixels we can compute a velocity vector to solve the system. However, combining equations for more than two pixels is more effective. We might get a system that has no solution; yet we can solve it roughly, using the least square method. We will use weighted combination of equations. This method involves the solution of 2×2 linear system.

$$\begin{aligned} \sum_{x,y} W(x,y) I_x I_y u + \sum_{x,y} W(x,y) I_y^2 v &= - \sum_{x,y} W(x,y) I_y I_t, \\ \sum_{x,y} W(x,y) I_x^2 u + \sum_{x,y} W(x,y) I_x I_y v &= - \sum_{x,y} W(x,y) I_x I_t, \end{aligned}$$

where $W(x,y)$ is the Gaussian window. The Gaussian window may be represented as a composition of two separable kernels with binomial coefficients. Iterating through the system can yield even better results. That is, retrieved offset is used to determine a new window in the second image from which the window in the first image is subtracted while I_t is calculated.

Horn & Schunck Technique

Horn and Schunck propose a technique that assumes the smoothness of the estimated optical flow field. This constraint can be formulated as

$$S = \iint_{\text{image}} \left[\left(\frac{\partial u}{\partial x} \right)^2 + \left(\frac{\partial u}{\partial y} \right)^2 + \left(\frac{\partial v}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial y} \right)^2 \right] (dx) dy \quad . \quad (18.6)$$

This optical flow solution can deviate from the optical flow constraint. To express this deviation the following integral can be used:

$$C = \iint_{\text{image}} \left(\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} \right)^2 dx dy. \quad (18.7)$$

The value $S + \lambda C$, where λ is a parameter, called Lagrangian multiplier, is to be minimized. Typically, a smaller λ must be taken for a noisy image and a larger one for a quite accurate image.

To minimize $S + \lambda C$, a system of two second-order differential equations for the whole image must be solved:

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= \lambda \left(\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} \right) \frac{\partial I}{\partial x}, \\ \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} &= \lambda \left(\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} \right) \frac{\partial I}{\partial y}. \end{aligned} \quad (18.8)$$

Iterative method could be applied for the purpose when a number of iterations are made for each pixel. This technique for two consecutive images seems to be computationally expensive because of iterations, but for a long sequence of images only an iteration for two images must be done, if the result of the previous iteration is chosen as initial approximation.

Block Matching

This technique does not use an optical flow equation directly. If, for example, an image tiled with small, possibly overlapping blocks is considered, then for every block in the first image the algorithm tries to find a block of the same size in the second image that is most similar to the block in the first image. The function searches in the neighborhood of some given point in the second image. So we assume that all the points in the block move by the same offset and find that offset, just like in [Lucas & Kanade](#) method. Different metrics can be used to measure similarity or difference between blocks - cross correlation, squared difference, etc.

Reference

cvCalcOpticalFlowHS

Calculates optical flow for two images.

```
void cvCalcOpticalFlowHS( IplImage* srcA, IplImage* srcB, int usePrevious,
    IplImage* velx, IplImage* vely, double lambda, CvTermCriteria criteria);
```

<i>imgA</i>	First image.
<i>imgB</i>	Second image.
<i>usePrevious</i>	Uses previous (input) velocity field.
<i>velx</i>	Horizontal component of the optical flow.
<i>vely</i>	Vertical component of the optical flow.
<i>lambda</i>	Lagrangian multiplier.
<i>criteria</i>	Criteria of termination of velocity computing.

Discussion

The function [cvCalcOpticalFlowHS](#) computes flow for every pixel, thus output images must have the same size as input. [Horn & Schunck technique](#) is implemented.

cvCalcOpticalFlowLK

Calculates optical flow for two images.

```
void cvCalcOpticalFlowLK( IplImage* srcA, IplImage* srcB, CvSize winSize,
    IplImage* velx, IplImage* vely);
```

<i>imgA</i>	First image.
<i>imgB</i>	Second image.
<i>winSize</i>	Size of the averaging window used for grouping pixels.

`velx` Horizontal component of the optical flow.
`vely` Vertical component of the optical flow.

Discussion

The function [cvCalcOpticalFlowLK](#) computes flow for every pixel, thus output images must have the same size as input. [Lucas & Kanade technique](#) is implemented.

cvCalcOpticalFlowBM

Calculates optical flow for two images by block matching method.

```
void cvCalcOpticalFlowBM( IplImage* srcA, IplImage* srcB, CvSize blockSize,  
    CvSize shiftSize, CvSize maxRange, int usePrevious, IplImage* velx,  
    IplImage* vely);
```

`imgA` First image.
`imgB` Second image.
`blockSize` Size of basic blocks that are compared.
`shiftSize` Block coordinate increments.
`maxRange` Size of the scanned neighborhood in pixels around block.
`usePrevious` Uses previous (input) velocity field.
`velx` Horizontal component of the optical flow.
`vely` Vertical component of the optical flow.

Discussion

The function [cvCalcOpticalFlowBM](#) calculates optical flow for two images using the [Block Matching algorithm](#). Velocity is computed for every block (not every pixel), so velocity image pixels correspond to input image blocks and the velocity image must have the following size:

$$\text{velocityFrameSize.width} = \left\lceil \frac{\text{imageSize.width}}{\text{blockSize.width}} \right\rceil,$$

$$\text{velocityFrameSize.height} = \left\lceil \frac{\text{imageSize.height}}{\text{blockSize.height}} \right\rceil.$$

cvCalcOpticalFlowPyrLK

Calculates optical flow for two images using iterative Lucas-Kanade method in pyramids.

```
void cvCalcOpticalFlowPyrLK(IplImage* imgA, IplImage* imgB, IplImage* pyrA,
    IplImage* pyrB, CvPoint2D32f* featuresA, CvPoint2D32f* featuresB, int
    count, CvSize winSize, int level, char* status, float* error,
    CvTermCriteria criteria, int flags );
```

<i>imgA</i>	First frame (time <i>t</i>).
<i>imgB</i>	Second frame (time <i>t</i> + <i>dt</i>).
<i>pyrA</i>	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 (<i>imgSize.width</i> +8)* <i>imgSize.height</i> /3 bytes is sufficient.
<i>pyrB</i>	Similar to <i>pyrA</i> , applies to the second frame.
<i>featuresA</i>	Array of points for which the flow needs to be found.
<i>featuresB</i>	Array of 2D points containing calculated new positions of input features in the second image.
<i>count</i>	Number of feature points.
<i>winSize</i>	Size of the search window of each pyramid level.
<i>level</i>	Maximal pyramid level number. If 0, pyramids are not used (single level), if 1, two levels are used, etc.
<i>status</i>	Array. Every element of the array is set to 1 if the flow for the corresponding feature has been found, 0 otherwise.
<i>error</i>	Array of double numbers containing difference between patches around the original and moved points. Optional parameter; can be NULL.

<i>criteria</i>	Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped.
<i>flags</i>	Miscellaneous flags: <ul style="list-style-type: none">• <code>CV_LKFLOW_PYR_A_READY</code>, pyramid for the first frame is precalculated before the call;• <code>CV_LKFLOW_PYR_B_READY</code>, pyramid for the second frame is precalculated before the call;• <code>CV_LKFLOW_INITIAL_GUESSES</code>, features <i>B</i> array holds initial guesses about new feature locations before the function call.

Discussion

The function [cvCalcOpticalFlowPyrLK](#) calculates the optical flow between two images for the given set of points. The function finds the flow with sup-pixel accuracy.

Both parameters *pyrA* and *pyrB* 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, if the image is large enough, the function calculates the pyramid and stores it in the buffer unless the flag `CV_LKFLOW_PYR_A[B]_READY` is set. After the function call both pyramids are calculated and the ready flag for the corresponding image can be set in the next call.

This chapter describes group of functions for estimating stochastic models state.

Overview

Definitions and Motivation

State estimation programs implement a model and an estimator. A model is analogous to a data structure representing relevant information about the visual scene. An estimator is analogous to the software engine that manipulates this data structure to compute beliefs about the world. The OpenCV routines provide two estimators: standard Kalman and condensation.

Models

Many computer vision applications involve repeated estimating, that is, tracking, of the system quantities that change over time. These dynamic quantities are called the system *state*. The system in question can be anything that happens to be of interest to a particular vision task.

To estimate the state of a system, reasonably accurate knowledge of the system *model* and *parameters* may be assumed. Parameters are the quantities that describe the model configuration but change at a rate much slower than the state. Parameters are often assumed known and static.

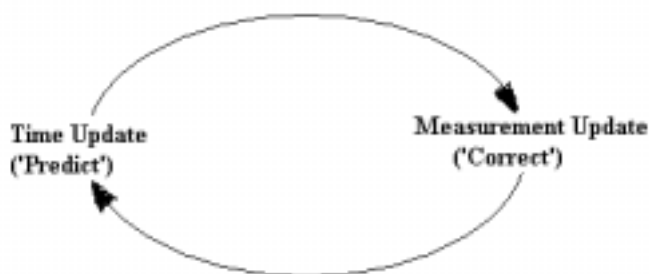
In OpenCV a state is represented with a vector. In addition to this output of the state estimate routines, there is another vector representing *measurements* that are input to the routines from the sensor data.

For the model, two main parts need to be represented. The first describes the dynamics of how the state is expected to change from one time step to the next. The other thing that needs to be represented is the model of how a measurement vector z_t is obtained from the state.

Estimators

Most estimators have the same general form with repeated propagation and update phases that modify the state's uncertainty as illustrated in [Figure 19-1](#).

Figure 19-1 Ongoing Discrete Kalman Filter Cycle



The time update projects the current state estimate ahead in time. The measurement update adjusts the projected estimate using an actual measurement at that time.

A common, desirable property of an estimator is being unbiased when the probability density of estimate errors has an expected value of 0. There exists an optimal propagation and update formulation that is the best, linear, unbiased estimator (BLUE) for any given model of the form. This formulation is known as the discrete Kalman estimator, whose standard form is implemented in OpenCV.

Kalman Filtering

The Kalman filter addresses the general problem of trying to estimate the state x of a discrete-time process that is governed by the linear stochastic difference equation

$$x_{k+1} = Ax_k + w_k \quad (19.1)$$

with a measurement z , that is

$$z_k = Hx_k + v_k \quad (19.2)$$

The random variables w_k and v_k respectively represent the process and measurement noise. They are assumed to be independent of each other, white, and with normal probability distributions

$$p(w) = N(0, Q), \quad (19.3)$$

$$p(v) = N(0, R). \quad (19.4)$$

The $N \times N$ matrix A in the [difference equation \(19.1\)](#) relates the state at time step k to the state at step $k+1$, in the absence of process noise. The $M \times N$ matrix H in the [measurement equation \(19.2\)](#) relates the state to the measurement z_k .

If the "super minus" $x_{\bar{k}}$ is defined as a priori state estimate at step k provided the process prior to step k is known, and x_k is a posteriori state estimate at step k provided measurement z_k is known, then a priori and a posteriori estimate errors can be defined as $e_{\bar{k}} = x_k - x_{\bar{k}}$. The a priori estimate error covariance is then $P_{\bar{k}} = E[e_{\bar{k}}e_{\bar{k}}^{-T}]$ and the a posteriori estimate error covariance is $P_k = E[e_k e_k^T]$.

The Kalman filter estimates the process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of noisy measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward in time the current state and error covariance estimates to obtain the a priori estimates for the next time step. The measurement update equations are responsible for the feedback, that is, for incorporating a new measurement into the a priori estimate to obtain an improved a posteriori estimate. The time update equations can also be viewed as predictor equations, while the measurement update equations can be thought of as corrector equations. Indeed, the final estimation algorithm resembles that of a predictor-corrector algorithm for solving numerical problems as shown in [Figure 19-1](#). The specific equations for the time and measurement updates are presented below.

Time Update Equations

$$x_{\bar{k}+1} = A_k x_k,$$

$$P_{\bar{k}+1} = A_k P_k A_k^T + Q_k.$$

Measurement Update Equations:

$$K_k = P_{\bar{k}} H_k^T (H_k P_{\bar{k}} H_k^T + R_k)^{-1},$$

$$X_k = X_{\bar{k}} + K_k (z_k - H_k X_{\bar{k}}),$$

$$P_k = (I - K_k H_k) P_{\bar{k}},$$

where K is the so-called Kalman gain matrix and I is the identity operator.

Example 19-1 CvKalman Structure Definition

```
typedef struct CvKalman
{
    int MP;        //Dimension of measurement vector
    int DP;        // Dimension of state vector
    float* PosterState; // Vector of State of the System in k-th step
    float* PriorState;  // Vector of State of the System in (k-1)-th step
    float* DynamMatr;   // Matrix of the linear Dynamics system
    float* MeasurementMatr; // Matrix of linear measurement
    float* MNCovariance; // Matrix of measurement noise covariance
    float* PNCovariance; // Matrix of process noise covariance
    float* KalmGainMatr; // Kalman Gain Matrix
    float* PriorErrorCovariance; //Prior Error Covariance matrix
    float* PosterErrorCovariance; //Poster Error Covariance matrix
    float* Temp1;       // Temporary Matrixes
    float* Temp2;
}CvKalman;
```

Reference

cvCreateKalman

Allocates Kalman filter structure.

```
CvKalman* cvCreateKalman( int DynamParams, int MeasureParams );
```

DynamParams Dimension of the state vector.

MeasureParams Dimension of the measurement vector.

Discussion

The function [cvCreateKalman](#) creates `CvKalman` structure and returns pointer to the structure.

cvReleaseKalman

Deallocates Kalman filter structure.

```
void cvReleaseKalman(CvKalman** Kalman);
```

Kalman Double pointer to the structure to be released.

Discussion

The function [cvReleaseKalman](#) releases the structure `CvKalman` (see [Example](#)) and frees the memory previously allocated for the structure.

cvKalmanUpdateByTime

Estimates subsequent model state.

```
void cvKalmanUpdateByTime (CvKalman* Kalman);
```

Kalman Pointer to the structure to be updated.

Discussion

The function [cvKalmanUpdateByTime](#) estimates the subsequent stochastic model state by its current state.

cvKalmanUpdateByMeasurement

Adjusts model state.

```
void cvKalmanUpdateByMeasurement (CvKalman* Kalman, CvMat* Measurement);
```

Kalman Pointer to the structure to be updated.

Measurement Pointer to the structure CvMat containing the measurement vector.

Discussion

The function [cvKalmanUpdateByMeasurement](#) adjusts stochastic model state on basis of the true measurements of the model state.

ConDensation Algorithm

This section describes the ConDensation (conditional density propagation) algorithm, based on factored sampling. The main idea of the algorithm is using the set of randomly generated samples for probability density approximation. For simplicity, general principles of ConDensation algorithm are described below for linear stochastic dynamical system:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{w}_k \quad (19.5)$$

with a measurement z .

For the algorithm to start a set of samples x^n must be generated. The samples are randomly generated vectors of states. The function `cvInitSampleSet` does it in OpenCV implementation.

During the first phase of the condensation algorithm every sample in the set is updated according to [Equation \(19.5\)](#).

Further, when the vector of measurement z is obtained, the algorithm estimates conditional probability densities of every sample $P(x^n|z)$. The OpenCV implementation of the condensation algorithm enables the user to define various probability density functions. There is no such special function in the library. After the probabilities are calculated, the user may evaluate, for example, moments of tracked process at the current time step.

Implementation of Nonlinear Models

If dynamics or measurement of the stochastic system is non-linear, the user may update the dynamics (A) or measurement (H) matrices, using their Taylor series at each time step.

Example 19-2 CvConDensation Structure Definition

```
typedef struct
{
    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;
```

Reference

cvCreateConDensation

Allocates ConDensation filter structure.

```
CvConDensation* cvCreateConDensation( int DP, int MP, int SamplesNum);
```

DynamParams Dimension of the state vector.

MeasureParams Dimension of the state vector.

SamplesNum Number of samples.

Discussion

The function [cvCreateConDensation](#) creates `CvConDensation` structure and returns pointer to the structure.

cvReleaseConDensation

Deallocates ConDensation filter structure.

```
void cvReleaseConDensation(CvConDensation** ConDens);
```

ConDens Pointer to the pointer to the structure to be released.

Discussion

The function [cvReleaseConDensation](#) releases the structure `CvConDensation` (see [Example](#)) and frees all memory previously allocated for the structure.

cvConDensInitSampleSet

Initializes sample set for condensation algorithm.

```
void cvConDensInitSampleSet(CvConDensation* ConDens, CvMat* lowerBound CvMat*  
upperBound);
```

ConDens Pointer to a structure to be initialized.

lowerBound Vector of the lower boundary for each dimension.

upperBound Vector of the upper boundary for each dimension.

Discussion

The function [cvConDensInitSampleSet](#) fills the samples arrays in the structure `CvConDensation` (see [Example](#)) with values within specified ranges.

cvConDensUpdatebyTime

Estimates subsequent model state.

```
void cvConDensUpdateByTime(CvConDensation* ConDens);
```

ConDens Pointer to the structure to be updated.

Discussion

The function [cvConDensUpdatebyTime](#) estimates the subsequent stochastic model state from its current state.

This chapter describes functions that together perform POSIT algorithm.

Overview

The POSIT algorithm determines the six degree-of-freedom pose of a known tracked 3D rigid object. Given the projected image coordinates of uniquely identified points on the object, the algorithm refines an initial pose estimate by iterating with a weak perspective camera model to construct new image points; the algorithm terminates when it reaches a converged image, the pose of which is the solution.

Background

Camera parameters

Camera parameters are the numbers describing a particular camera configuration. The *intrinsic* camera parameters are those that specify the camera itself; they include the focal length, that is, the distance between the camera lens and the image plane, the location of the image center in pixel coordinates, the effective pixel size, and the radial distortion coefficient of the lens. To simplify pose recovery, the focal length is the only intrinsic parameter considered as it is the only one contributing to the geometric image formation model. The *extrinsic* camera parameters describe the spatial relationship between the camera and the world; they are the rotation matrix and translation vector specifying the transformation between the camera and world reference frames. In the case of pose recovery of a rigid object, the six degree-of-freedom extrinsic parameters are exactly the pose being sought.

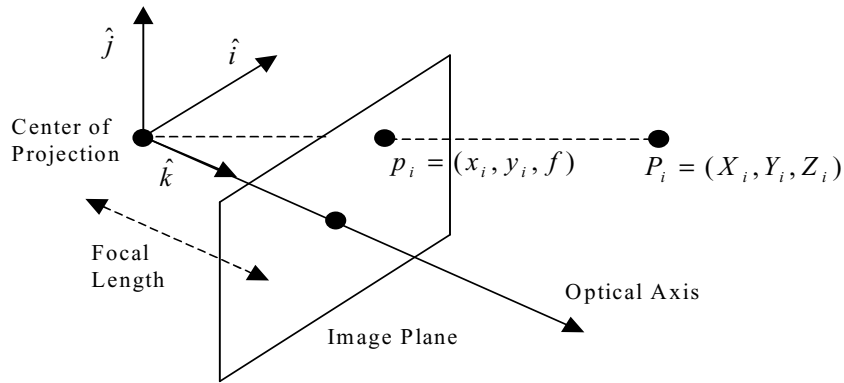
Geometric Image Formation

The link between world points and their corresponding image points is the projection from world space to image space. [Figure 20-1](#) depicts the *perspective* (or *pinhole*) model, which is the most common projection model because of its generality and usefulness.

The points in the world are projected onto the image plane according to their distance from the center of projection. Using similar triangles, the relationship between the coordinates of an image point $p_i = (x_i, y_i)$ and its world point $P_i = (X_i, Y_i, Z_i)$ can be determined as

$$x_i = \frac{f}{Z_i} X_i, y_i = \frac{f}{Z_i} Y_i. \quad (20.1)$$

Figure 20-1 Perspective Geometry Projection



The *weak-perspective* projection model simplifies the projection equation by replacing all z_i with a representative \bar{z} so that $s = f/\bar{z}$ is a constant scale for all points. The projection equations are then

$$x_i = sX_i, y_i = sY_i. \quad (20.2)$$

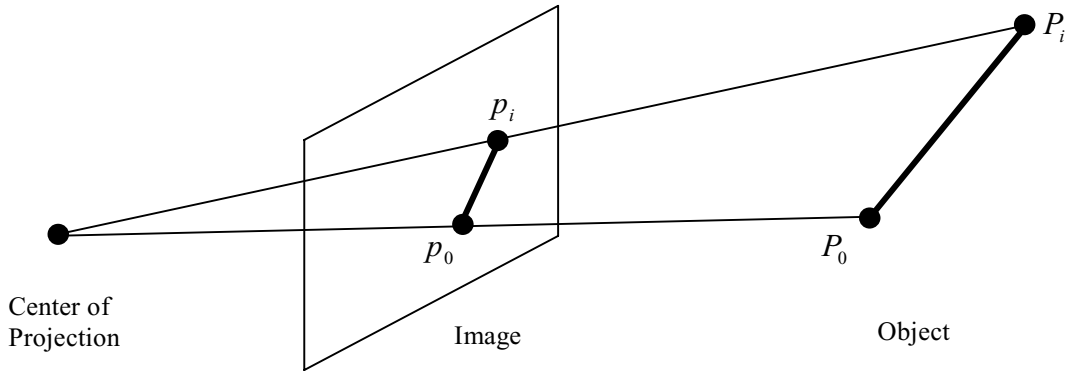
Because this situation can be modeled as an orthographic projection ($x_i = X_i$, $y_i = Y_i$) followed by isotropic scaling, weak-perspective projection is sometimes called *scaled orthographic projection*. Weak-perspective is a valid assumption only when the distances between any Z_i are much smaller than the distance between the Z_i and the center of projection; in other words, the world points are clustered and far enough from the camera. Possible \tilde{z} include any Z_i or the average over all Z_i .

More detailed explanations of this material can be found in [Trucco98].

Pose Approximation Method

Using weak-perspective projection, a method for determining approximate pose, termed *Pose from Orthography and Scaling* (POS) in [DeMenthon92], can be derived. First, a reference point P_0 in the world is chosen from which all other world points can be described as vectors: $\vec{P} = P_i - P_0$ (see Figure 20-2).

Figure 20-2 Scaling of Vectors in Weak-Perspective Projection



Similarly, the projection of this point, namely p_0 , is a reference point for the image points: $\vec{p}_i = p_i - p_0$. Proceeding from the weak-perspective assumption, the x component of \vec{p}_i is a scaled-down form of the x component of \vec{P}_i :

$$x_i - x_0 = s(X_i - X_0) = s(\vec{P}_0 \cdot \hat{i}). \quad (20.3)$$

This is also true for their y components. If I and J are defined as scaled-up versions of the unit vectors \hat{i} and \hat{j} ($I = s\hat{i}$ and $J = s\hat{j}$), then

$$x_i - x_0 = \vec{p}_i \cdot I \text{ and } y_i - y_0 = \vec{p}_i \cdot J \quad (20.4)$$

as two equations for each point for which I and J are unknown. These equations, collected over all the points, can be put into matrix form as

$$\underline{x} = MI \text{ and } \underline{y} = MJ, \quad (20.5)$$

where \underline{x} is a vector of \vec{p}_i x components, \underline{y} is a vector of \vec{p}_i y components, and M is a matrix whose rows are the \vec{p}_i vectors. These two sets of equations can be further joined to construct a single set of linear equations:

$$[\underline{x} \ \underline{y}] = M[I \ J] \Rightarrow \vec{p}_i C = M[I \ J], \quad (20.6)$$

where \vec{p}_i is a matrix whose rows are \vec{p}_i . Now that we have an overconstrained system of linear equations, we can solve for I and J in a least-squares sense as

$$[I \ J] = M^+ \vec{p}_i, \quad (20.7)$$

where M^+ is the pseudo-inverse of M .

Now that we have I and J , we construct the pose estimate as follows. First, \hat{i} and \hat{j} are estimated as I and J normalized, that is, scaled to unit length. By construction, these are the first two rows of the rotation matrix, and their cross-product is the third row:

$$R = \begin{bmatrix} \hat{i}^T \\ \hat{j}^T \\ (\hat{i} \times \hat{j})^T \end{bmatrix}. \quad (20.8)$$

The average of the magnitudes of I and J is an estimate of the weak-perspective scale s . From the weak-perspective equations, the world point P_0 in camera coordinates is the image point p_0 in camera coordinates scaled by s :

$$P_0 = p_0/s = [x_0 \ y_0 \ f]/s, \quad (20.9)$$

which is precisely the translation vector being sought.

Algorithm

The POSIT algorithm was first presented in the paper by DeMenthon and Davis [DeMenthon92]. In this paper, the authors first describe their POS (Pose from Orthography and Scaling) algorithm. By approximating perspective projection with weak-perspective projection POS produces a pose estimate from a given image. POS can be repeatedly used by constructing a new weak perspective image from each pose estimate and feeding it into the next iteration. The calculated images are estimates of the initial perspective image with successively smaller amounts of “perspective distortion” so that the final image contains no such distortion. The authors term this iterative use of POS as POSIT (POS with Iterations).

POSIT requires three pieces of known information. First, the *object model* consists of N points, each with unique 3D coordinates. N must be greater than 3, and the points must be non-degenerate (non-coplanar) to avoid algorithmic difficulties. Better results are achieved by using more points and by choosing points as far from coplanarity as possible. The object model is an $N \times 3$ matrix. Second, the *object image* is the set of 2D points resulting from a camera projection of the model points onto an image plane; it is a function of the object current pose. The object image is an $N \times 2$ matrix. Finally, the focal length of the camera must be known.

Given the object model and the object image, the algorithm proceeds as follows. First, the object image is assumed to be a weak perspective image of the object, from which a least-squares pose approximation is calculated via the object model pseudoinverse. From this approximate pose the object model is projected onto the image plane to construct a new weak perspective image. From this image a new approximate pose is found using least-squares, which in turn determines another weak perspective image, and so on. For well-behaved inputs, this procedure converges to an unchanging weak perspective image, whose corresponding pose is the final calculated object pose.

Example 20-1 POSIT Algorithm in Pseudo-Code

```
POSIT (imagePoints, objectPoints, focalLength) {
    count = converged = 0;
    modelVectors = modelPoints - modelPoints(0);
    oldWeakImagePoints = imagePoints;
    while (!converged) {
        if (count == 0)
            imageVectors = imagePoints - imagePoints(0);
        else {
            weakImagePoints = imagePoints .*
```

Example 20-1 POSIT Algorithm in Pseudo-Code (continued)

```

                                ((1 + modelVectors*row3/translation(3)) * [1
1]);
    imageDifference = sum(sum(abs( round(weakImagePoints) -
                                round(oldWeakImagePoints))));
    oldWeakImagePoints = weakImagePoints;
    imageVectors = weakImagePoints - weakImagePoints(0);
}
[I J] = pseudoinverse(modelVectors) * imageVectors;
row1 = I / norm(I);
row2 = J / norm(J);
row3 = crossproduct(row1, row2);
rotation = [row1; row2; row3];
scale = (norm(I) + norm(J)) / 2;
translation = [imagePoints(1,1); imagePoints(1,2); focalLength] /
              scale;
converged = (count > 0) && (diff < 1);
count = count + 1;
}
return {rotation, translation};
}

```

As the first step assumes, the object image is a weak perspective image of the object. It is a valid assumption only for an object that is far enough from the camera so that “perspective distortions” are insignificant. For such objects the correct pose is recovered immediately and convergence occurs at the second iteration. For less ideal situations, the pose is quickly recovered after several iterations. However, convergence is not guaranteed when perspective distortions are significant, for example, when an object is close to the camera with pronounced foreshortening. DeMenthon and Davis state that “convergence seems to be guaranteed if the image features are at a distance from the image center shorter than the focal length.”[\[DeMenthon92\]](#) Fortunately, this occurs for most realistic camera and object configurations.

Reference

cvCreatePOSITObject

Initializes structure containing object information.

```
CvPOSITObject* cvCreatePOSITObject( CvPoint3D32f* points, int numPoints );
```

points Pointer to the points of the 3D object model.

numPoints Number of object points.

Discussion

The function [cvCreatePOSITObject](#) allocates memory for the object structure and computes the object inverse matrix.

This data is stored in the structure `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.

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.

cvPOSIT

Implements POSIT algorithm.

```
void cvPOSIT( CvPoint2D32f* imagePoints, CvPOSITObject* pObject, double  
              focalLength, CvTermCriteria criteria, CvMatrix3* rotation, CvPoint3D32f*  
              translation);
```

<i>imagePoints</i>	Pointer to the object points projections on the 2D image plane.
<i>pObject</i>	Pointer to the object structure.
<i>focalLength</i>	Focal length of the camera used.
<i>criteria</i>	Termination criteria of the iterative POSIT algorithm.
<i>rotation</i>	Matrix of rotations.
<i>translation</i>	Translation vector.

Discussion

The function [cvPOSIT](#) implements POSIT algorithm. Image coordinates are given in a camera-related coordinate system. Camera calibration functions must define the focal length of the camera. At every iteration of the algorithm new perspective projection of estimated pose is computed.

Difference norm between two projections is the maximal distance between correspondent points. The parameter *criteria.epsilon* serves to stop the algorithm if the difference is small.

cvReleasePOSITObject

Deallocates 3D object structure.

```
void cvReleasePOSITObject( CvPOSITObject** ppObject );
```

ppObject Address of the pointer to the object structure.

Discussion

The function [cvReleasePOSITObject](#) is used to release memory previously allocated by the function [cvCreatePOSITObject](#).

This chapter describes functions that operate on multi-dimensional histograms.

Overview

Histogram is a discrete approximation of stochastic variable probability distribution. The variable can be both a scalar value and a vector. Histograms are widely used in image processing and computer vision. For example, one-dimensional histograms can be used for:

- grayscale image enhancement,
- determining optimal threshold levels (see [Threshold Functions](#)),
- selecting color objects via hue histograms back projection (see [CamShift](#)), and other operations.

Two-dimensional histograms can be used, for example, for:

- analyzing and segmenting color images, normalized to brightness (e.g. red-green or hue-saturation images),
- analyzing and segmenting motion fields (x - y or magnitude-angle histograms),
- analyzing shapes (see [cvCalcPGH](#) in [Geometry](#) chapter) or textures.

Multi dimensional histograms can be used for:

- content based retrieval (see the function [cvCalcEMD](#)),
- bayesian-based object recognition (see [[Schiele2000](#)]).

To store all the types of histograms (1D, 2D, nD), OpenCV introduces special structure `CvHistogram` described in [Example 21-1](#).

Example 21-1 `CvHistogram` Structure Definition

```
typedef struct CvHistogram
{
    int      header_size; /* header's size */
    CvHistType type; /* type of histogram */
    int      flags; /* histogram's flags */
    int      c_dims; /* histogram's dimension */
    int      dims[CV_HIST_MAX_DIM]; /* every dimension size */
    int      mdims[CV_HIST_MAX_DIM]; /* coefficients for fast
                                     access to element
                                     /* &m[a,b,c] = m + a*mdims[0] +
                                     b*mdims[1] + c*mdims[2] */
    float*   thresh[CV_HIST_MAX_DIM]; /* bin boundaries arrays for every
                                     dimension */
    float*   array; /* all the histogram data, expanded into
                     the single row */
    struct CvNode* root; /* tree - histogram data */
    CvSet*   set; /* pointer to memory storage
                  (for tree data) */
    int*     chdims[CV_HIST_MAX_DIM]; /* cache data for fast calculating */
} CvHistogram;
```

It is possible to store any histogram either in a dense form (as a multi-dimensional array) or in a sparse form (now a balanced tree is used), however, it is reasonable to store 4D (or even 3D) histograms and higher dimensional histograms in a sparse form and 1D or 2D histograms in a dense form.

The type of histogram representation is passed into histogram creation function and then it is stored in `type` field of `CvHistogram`. It is possible to use histogram processing functions from this chapter on histograms created by the user. Use the function [cvMakeHistHeaderForArray](#).

Histograms and Signatures

Histograms represent a simple statistical description of an object, e.g., an image. The object characteristics are measured during iterating through that object: for example, color histograms for an image are built from pixel values in one of the color spaces.

We quantize all the possible values of that multi-dimensional characteristic on each coordinate. If the quantized characteristic can take different k_1 values on the first coordinate, k_2 values on second, and k_n on the last one, the resulting histogram has the size $size = \prod_{i=1}^n k_i$.

The histogram can be viewed as a multi-dimensional array. Each dimension corresponds to a certain object feature. An array element with coordinates $[i_1, i_2 \dots i_n]$, otherwise called a histogram bin, contains a number of measurements done for the object with quantized value equal to i_1 on first coordinate, i_2 on the second coordinate, and so on. We can compare objects using their histograms:

$$D_{L_1}(H, K) = \sum_i |h_i - k_i|, \text{ or}$$

$$D(H, K) = \sqrt{(\bar{h} - \bar{k})^T A (\bar{h} - \bar{k})}.$$

But these methods suffer from several disadvantages. D_{L_1} sometimes gives too small difference when there is no exact correspondence between histogram bins, that is, if the bins of one histogram are slightly shifted. On the other hand, D_{L_2} gives too large difference due to cumulative property.

Another drawback of pure histograms is large space required, especially for higher-dimensional characteristics. The solution is to store not all histogram bins, but only the ones that are non-zero, or just the ones with the highest score. Generalization of histograms is termed *signature* and defined in the following way:

1. Characteristic values with rather fine quantization are gathered.
2. Only non-zero bins are dynamically stored.

This can be implemented using hash-tables, balanced trees, or other “sparse” structures. After processing, a set of “clusters” is obtained. Each of them is characterized by the coordinates and weight, that is, a number of measurements in the neighborhood. Removing clusters with small weight can further reduce the signature size. Although these structures cannot be compared using formulas written above, there exists a robust comparison method described in [RubnerJan98] called Earth Mover Distance.

Earth Mover Distance (EMD)

Physically, two signatures can be viewed as two systems - earth masses, spread into several localized pieces. Each piece, or cluster, has some coordinates in space and weight, that is, the earth mass it contains. The distance between two systems can be measured then as a minimal work needed to get the second configuration from the first or vice versa. To get metric, invariant to scale, the result is to be divided by the total mass of the system.

Mathematically, it can be formulated as follows.

Consider m suppliers and n consumers. Let the capacity of i^{th} supplier be x_i and the capacity of j^{th} consumer be y_j . Also, let the ground distance between i^{th} supplier and j^{th} consumer be $c_{i,j}$. The following restrictions must be met:

$$x_i \geq 0, y_j \geq 0, c_{i,j} \geq 0,$$

$$\sum_i x_i \geq \sum_j y_j,$$

$$0 \leq i < m, 0 \leq j < n.$$

Then the task is to find the flow matrix $\|f_{i,j}\|$, where $f_{i,j}$ is the amount of earth, transferred from i^{th} supplier to j^{th} consumer. This flow must satisfy the restrictions below:

$$f_{i,j} \geq 0,$$

$$\sum_i f_{i,j} \leq x_i,$$

$$\sum_j f_{i,j} = y_j$$

and minimize the overall cost:

$$\min \sum_i \sum_j c_{i,j} f_{i,j}.$$

If $\|f_{i,j}\|$ is the optimal flow, then Earth Mover Distance is defined as

$$EMD(x, y) = \frac{\sum_i \sum_j c_{i,j} f_{i,j}}{\sum_i \sum_j f_{i,j}}.$$

The task of finding the optimal flow is a well known transportation problem, which can be solved, for example, using the simplex method.

Example Ground Distances

As shown in the section above, physically intuitive distance between two systems can be found if the distance between their elements can be measured. The latter distance is called ground distance and, if it is a true metric, then the resultant distance between systems is a metric too. The choice of the ground distance depends on the concrete task as well as the choice of the coordinate system for the measured characteristic. In [\[RubnerSept98\]](#), [\[RubnerOct98\]](#) three different distances are considered.

- The first is used for human-like color discrimination between pictures. CIE Lab model represents colors in a way when a simple Euclidean distance gives true human-like discrimination between colors. So, converting image pixels into CIE Lab format, that is, representing colors as 3D-vectors (L,a,b), and quantizing them (in 25 segments on each coordinate in [\[RubnerSept98\]](#)), produces a color-based signature of the image. Although in experiment, made in [\[RubnerSept98\]](#), the maximal number of non-zero bins could be $25 \times 25 \times 25 = 15625$, the average number of clusters was ~ 8.8 , that is, resulting signatures were very compact.
- The second example is more complex. Not only the color values are considered, but also the coordinates of the corresponding pixels, which makes it possible to differentiate between pictures of similar color palette but representing different color regions placements: e.g., green grass at the bottom and blue sky on top vs. green forest on top and blue lake at the bottom. 5D space is used and metric is: $[(\Delta L)^2 + (\Delta a)^2 + (\Delta b)^2 + \lambda((\Delta x)^2 + (\Delta y)^2)]^{1/2}$, where λ regulates importance of the spatial correspondence. When $\lambda = 0$, the first metric is obtained.
- The third example is related to texture metrics. In the example Gabor transform is used to get the 2D-vector texture descriptor (l, m) , which is a log-polar characteristic of the texture. Then, no-invariance ground distance is defined as: $d((l_1, m_1), (l_2, m_2)) = |\Delta l| + \alpha |\Delta m|$, $\Delta l = \min(|l_1 - l_2|, L - |l_1 - l_2|)$, $\Delta m = |m_1 - m_2|$, where α is the scale parameter of Gabor transform, L is the number of different angles used (angle resolution), and M is the number of scales used (scale resolution). To get invariance to scale and rotation, the user may calculate minimal *EMD* for several scales and rotations:
 $(l_1, m_1), (l_2, m_2)$

$$EMD(t_1, t_2) = \min_{\substack{0 \leq l_0 < L \\ -M < m_0 < M}} EMD(t_1, t_2, l_0, m_0),$$

where d is measured as in the previous case, but Δl and Δm look slightly different:

$$\Delta l = \min(|l_1 - l_2 + l_0(\text{mod } L)|, L - |l_1 - l_2 + l_0(\text{mod } L)|), \Delta m = |m_1 - m_2 + m_0|.$$

Lower Boundary for EMD

If ground distance is metric and distance between points can be calculated via the norm of their difference, and total suppliers' capacity is equal to total consumers' capacity, then it is easy to calculate lower boundary of *EMD* because:

$$\begin{aligned} \sum_i \sum_j c_{i,j} f_{i,j} &= \sum_i \sum_j \|p_i - q_j\| f_{i,j} = \sum_i \sum_j \|p_i - q_j\| f_{i,j} \\ &\geq \left\| \sum_i \sum_j \|p_i - q_i\| f_{i,j} \right\| = \left\| \sum_i \left(\sum_j f_{i,j} \right) p_i - \sum_j \left(\sum_i f_{i,j} \right) q_j \right\| \\ &= \left\| \sum_i x_i p_i - \sum_j y_j q_j \right\| \end{aligned}$$

As it can be seen, the latter expression is the distance between the mass centers of the systems.

Poor candidates can be easily rejected using this lower boundary for *EMD* distance, when searching in the large image database.

Reference

cvCreateHist

Creates histogram.

```
CvHistogram* CreateHist( int c_dims, int* dims, CvHistType type,
float** ranges=0, int uniform=1);
```

<i>c_dims</i>	Number of histogram dimensions.
<i>dims</i>	Array with numbers of bins per each dimension.

<i>type</i>	Histogram representation format: <code>CV_HIST_ARRAY</code> means that histogram data is represented as an array; <code>CV_HIST_TREE</code> means that histogram data is represented as a sparse structure, that is, the balanced tree in this implementation.
<i>ranges</i>	2-D array, or more exactly, an array of arrays, of bin ranges for every histogram dimension. Its meaning depends on the uniform parameter value.
<i>uniform</i>	If not 0, the histogram has evenly spaced bins and every element of <i>ranges</i> array is an array of two numbers - lower and upper boundaries for the corresponding histogram dimension. If the parameter is equal to 0, then i^{th} element of ranges array contains $dims[i]+1$ elements: $l(0), u(0) == l(1), u(1) == l(2), \dots, u(n-1)$, where $l(i)$ and $u(i)$ are lower and upper boundaries for the i^{th} bin, respectively.

Discussion

The function [cvCreateHist](#) creates a histogram of the specified size and returns the pointer to the created histogram. If the array *ranges* is 0, the histogram bin ranges must be specified later via the function [cvSetHistBinRanges](#).

cvReleaseHist

Releases histogram header and underlying data.

```
void cvReleaseHist( CvHistogram** hist );
```

hist Pointer to the released histogram.

Discussion

The function [cvReleaseHist](#) releases the histogram header and underlying data. The pointer to histogram is cleared by the function. If **hist* pointer is already `NULL`, the function has no effect.

cvMakeHistHeaderForArray

Initializes histogram header.

```
void cvMakeHistHeaderForArray( int c_dims, int* dims, CvHistogram* hist,
    float* data, float** ranges=0, int uniform=1 );
```

<i>c_dims</i>	Histogram dimension number.
<i>dims</i>	Dimension size array.
<i>hist</i>	Pointer to the histogram to be created.
<i>data</i>	Pointer to the source data histogram.
<i>ranges</i>	2D array of bin ranges.
<i>uniform</i>	If not 0, the histogram has evenly spaced bins.

Discussion

The function [cvMakeHistHeaderForArray](#) initializes the histogram header and sets the data pointer to the given value *data*. The histogram must have the type CV_HIST_ARRAY. If the array *ranges* is 0, the histogram bin ranges must be specified later via the function [cvSetHistBinRanges](#).

cvQueryHistValue_1D

Queries value of histogram bin.

```
float cvQueryHistValue_1D( CvHistogram* hist, int idx0 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Index of the bin.

Discussion

The function [cvQueryHistValue_1D](#) returns the value of the specified bin of *1D* histogram. If the histogram representation is a sparse structure and the specified bin is not present, the function return 0.

cvQueryHistValue_2D

Queries value of histogram bin.

```
float cvQueryHistValue_2D( CvHistogram* hist, int idx0, int idx1 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Index of the bin in the first dimension.
<i>idx1</i>	Index of the bin in the second dimension.

Discussion

The function [cvQueryHistValue_2D](#) returns the value of the specified bin of *2D* histogram. If the histogram representation is a sparse structure and the specified bin is not present, the function return 0.

cvQueryHistValue_3D

Queries value of histogram bin.

```
float cvQueryHistValue_3D( CvHistogram* hist, int idx0, int idx1, int idx2 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Index of the bin in the first dimension.
<i>idx1</i>	Index of the bin in the second dimension.
<i>idx2</i>	Index of the bin in the third dimension.

Discussion

The function [cvQueryHistValue_3D](#) returns the value of the specified bin of 3D histogram. If the histogram representation is a sparse structure and the specified bin is not present, the function return 0.

cvQueryHistValue_nD

Queries value of histogram bin.

```
float cvQueryHistValue_nD( CvHistogram* hist, int* idx );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx</i>	Array of bin indices, that is, a multi-dimensional index.

Discussion

The function [cvQueryHistValue_nD](#) returns the value of the specified bin of *nD* histogram. If the histogram representation is a sparse structure and the specified bin is not present, the function return 0. The function is the most general in the family of [QueryHistValue](#) functions.

cvGetHistValue_1D

Returns pointer to histogram bin.

```
float* cvGetHistValue_1D( CvHistogram* hist, int idx0 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Index of the bin.

Discussion

The function [cvGetHistValue_1D](#) returns the pointer to the histogram bin, given its coordinates. If the bin is not present, it is created and initialized with 0. The function returns `NULL` pointer if the input coordinates are invalid.

cvGetHistValue_2D

Returns pointer to histogram bin.

```
float* cvGetHistValue_2D( CvHistogram* hist, int idx0, int idx1 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Index of the bin in the first dimension.
<i>idx1</i>	Index of the bin in the second dimension.

Discussion

The function [cvGetHistValue_2D](#) returns the pointer to the histogram bin, given its coordinates. If the bin is not present, it is created and initialized with 0. The function returns `NULL` pointer if the input coordinates are invalid.

cvGetHistValue_3D

Returns pointer to histogram bin.

```
float* cvGetHistValue_3D( CvHistogram* hist, int idx0, int idx1, int idx2 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Index of the bin in the first dimension.
<i>idx1</i>	Index of the bin in the second dimension.
<i>idx2</i>	Index of the bin in the third dimension.

Discussion

The function [cvGetHistValue_3D](#) returns the pointer to the histogram bin, given its coordinates. If the bin is not present, it is created and initialized with 0. The function returns `NULL` pointer if the input coordinates are invalid.

cvGetHistValue_nD

Returns pointer to histogram bin.

```
float* cvGetHistValue_nD( CvHistogram* hist, int* idx );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx</i>	Array of bin indices, that is, a multi-dimensional index.

Discussion

The function [cvGetHistValue_nD](#) returns the pointer to the histogram bin, given its coordinates. If the bin is not present, it is created and initialized with 0. The function returns `NULL` pointer if the input coordinates are invalid.

cvGetMinMaxHistValue

Finds minimum and maximum histogram bins.

```
void cvGetMinMaxHistValue( CvHistogram* hist, float* minVal, float* maxVal,  
    int* minIdx=0, int* maxIdx=0 );
```

<i>hist</i>	Pointer to the histogram.
<i>minVal</i>	Pointer to the minimum value of the histogram; can be <code>NULL</code> .
<i>maxVal</i>	Pointer to the maximum value of the histogram; can be <code>NULL</code> .
<i>minIdx</i>	Pointer to the array of coordinates for minimum. If not <code>NULL</code> , must have <i>hist</i> -> <i>c_dims</i> elements.

maxIdx Pointer to the array of coordinates for maximum. If not `NULL`, must have *hist->c_dims* elements.

Discussion

The function [cvGetMinMaxHistValue](#) finds the minimum and maximum histogram bins and their positions.

cvNormalizeHist

Normalizes histogram.

```
void cvNormalizeHist( CvHistogram* hist, float factor );
```

hist Pointer to the histogram.

factor Normalization factor.

Discussion

The function [cvNormalizeHist](#) normalizes the histogram, such that the sum of histogram bins becomes equal to *factor*.

cvThreshHist

Thresholds histogram.

```
void cvThreshHist( CvHistogram* hist, float thresh);
```

hist Pointer to the histogram.

thresh Threshold level.

Discussion

The function [cvThreshHist](#) clears histogram bins that are below the specified level.

cvCompareHist

Compares two histograms.

```
double cvCompareHist( CvHistogram* hist1, CvHistogram* hist2, CvCompareMethod
method );
```

hist1 First histogram.

hist2 Second histogram.

method Comparison method; may be any of those listed below:

- CV_COMP_CORREL;
- CV_COMP_CHISQR;
- CV_COMP_INTERSECT.

Discussion

The function [cvCompareHist](#) compares two histograms using specified method.

$$\text{CV_COMP_CORREL } result = \frac{\sum_i \hat{q}_i \hat{v}_i}{\sqrt{\sum_i \hat{q}_i^2 * \sum_i \hat{v}_i^2}},$$

$$\text{CV_COMP_CHISQR } result = \sum_i \frac{(q_i - v_i)^2}{q_i + v_i},$$

$$\text{CV_COMP_INTERSECT } result = \sum_i \min(q_i, v_i).$$

The function returns the comparison result.

cvCopyHist

Copies histogram.

```
void cvCopyHist( CvHistogram* src, CvHistogram** dst );
```

src Source histogram.

dst Pointer to destination histogram.

Discussion

The function [cvCopyHist](#) makes a copy of the histogram. If the second histogram pointer **dst* is null, it is allocated and the pointer is stored at **dst*. Otherwise, both histograms must have equal types and sizes, and the function simply copies the source histogram bins values to destination histogram.

cvSetHistBinRanges

Sets bounds of histogram bins.

```
void cvSetHistBinRanges( CvHistogram* hist, float** ranges, int uniform=1 );
```

hist Destination histogram.

ranges 2D array of bin ranges.

uniform If not 0, the histogram has evenly spaced bins.

Discussion

The function [cvSetHistBinRanges](#) is a stand-alone function for setting bin ranges in the histogram. For more detailed description of the parameters *ranges* and *uniform* see [cvCreateHist](#) function, that can initialize the ranges as well. Ranges for histogram bins must be set before the histogram is calculated or backproject of the histogram is calculated.

cvCalcHist

Calculates histogram of image(s).

```
void cvCalcHist( IplImage** img, CvHistogram* hist, int doNotClear=0,
IplImage* mask=0 );
```

<i>img</i>	Source images.
<i>hist</i>	Pointer to the histogram.
<i>doNotClear</i>	Clear flag.
<i>mask</i>	Mask; determines what pixels of the source images are considered in process of histogram calculation.

Discussion

The function [cvCalcHist](#) calculates the histogram of the array of single-channel images. If the parameter *doNotClear* is 0, then the histogram is cleared before calculation; otherwise the histogram is simply updated.

cvCalcBackProject

Calculates back project.

```
void cvCalcBackProject( IplImage** img, IplImage* dstImg, CvHistogram* hist);
```

<i>img</i>	Source images array.
<i>dstImg</i>	Destination image.
<i>hist</i>	Source histogram.

Discussion

The function [cvCalcBackProject](#) calculates the back project of the histogram. For each group of pixels taken from the same position from all input single-channel images the function puts the histogram bin value to the destination image, where the

coordinates of the bin are determined by the values of pixels in this input group. From the statistical point of view, an output image pixel value characterizes probability of the corresponding input pixels group belonging to an object, whose local features distribution histogram is used.

For example, to find a red object in the picture the procedure is as follows:

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 project using the histogram and get the picture, where bright pixel corresponds to typical colors (for example, red) in the searched object.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

cvCalcBackProjectPatch

Calculates back project patch of histogram.

```
void cvCalcBackProjectPatch( IplImage** img, IplImage* dst, CvSize patchSize,
    CvHistogram* hist, CvCompareMethod method, float normFactor );
```

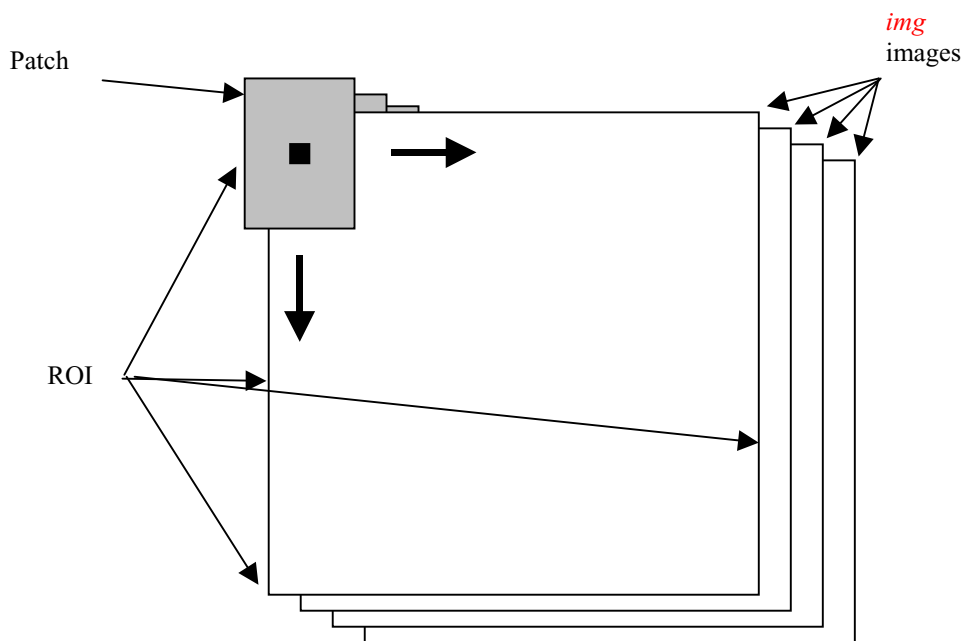
<i>img</i>	Source images array.
<i>dst</i>	Destination image.
<i>patchSize</i>	Size of patch slid though the source image.
<i>hist</i>	Probabilistic model.
<i>method</i>	Method of comparison.
<i>normFactor</i>	Normalization factor.

Discussion

The function [cvCalcBackProjectPatch](#) calculates 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 *img*. 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 *img* image array is a collection of these measurement images. A multi-dimensional histogram *hist* is constructed by sampling from the *img* image array. The final histogram is normalized. The *hist* histogram has as many dimensions as elements in *img* array.

Each new image is measured and then converted into an *img* image array over a chosen ROI. Histograms are taken from this *img* image in an area covered by a “patch” with anchor at center as shown in [Figure 21-1](#). 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` (the parameter *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. Subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can make many calculations redundant.

Figure 21-1 Back Project Calculation by Patches

Each image of the image array *img* shown in the figure stores the corresponding element of a multi-dimensional measurement vector. Histogram measurements are drawn from measurement vectors over a patch with anchor at the center. A multi-dimensional histogram *hist* is used via the function [cvCompareHist](#) to calculate the output at the patch anchor. The patch is slid around until the values are calculated over the whole ROI.

cvCalcEMD

Computes earth mover distance.

```
void cvCalcEMD(float* signature1, int size1, float* signature2, int size2, int
    dims, CvDisType distType, float (*dist_func)( float* f1, float* f2, void*
    user_param), float* emd, float* lowerBound, void* user_param);
```

<i>signature1</i>	First signature, array of <i>size1</i> * (<i>dims</i> + 1) elements.
<i>signature2</i>	Second signature, array of <i>size2</i> * (<i>dims</i> + 1) elements.
<i>dims</i>	Number of dimensions in feature space.
<i>distType</i>	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 is used as the metric. The function takes two coordinate vectors and user parameter and returns the distance between two vectors.
<i>emd</i>	Pointer to the calculated <i>emd</i> distance.
<i>lowerBound</i>	Pointer to the calculated lower boundary.

Discussion

The function [cvCalcEMD](#) computes earth mover distance and/or a lower boundary of the distance. The lower boundary can be calculated only if *dims* > 0, and it has sense only if the metric used satisfies all metric axioms. The lower boundary is calculated very fast and can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object. If the parameter *dims* is equal to 0, then *signature1* and *signature2* are considered simple *ID* histograms. Otherwise, both signatures must look as follows:

```
(weight_i0, x0_i0, x1_i0, ..., x(dims-1)_i0,
weight_i1, x0_i1, x1_i1, ..., x(dims-1)_i1,
...
weight_(size1-1), x0_(size1-1), x1_(size1-1), ..., x(dims-1)_(size1-1)),
```

where *weight_{ik}* is the weight of *ik cluster*, while *x0_{ik}*, ..., *x(dims-1)_{ik}* are coordinates of the cluster *ik*.

If the parameter *lower_bound* is equal to 0, only *emd* is calculated. If the calculated lower boundary is greater than or equal to the value stored at this pointer, then the true *emd* is not calculated, but is set to that *lower_bound*.

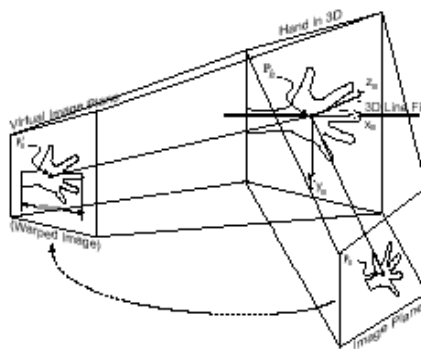
This chapter describes specific functions for the static gesture recognition technology.

Overview

The gesture recognition algorithm can be divided into four main components as illustrated in [Figure 22-1](#).

The first component computes the 3D arm pose from range image data that may be obtained from the standard stereo correspondence algorithm. The process includes 3D line fitting, finding the arm position along the line and creating the arm mask image.

Figure 22-1 Gesture Recognition Algorithm



The second component produces a frontal view of the arm image and arm mask through a planar homograph transformation. The process consists of the homograph matrix calculation and warping image and image mask (See [Figure 22-2](#)).

The third component segments the arm from the background based on the probability density estimate that a pixel with a given hue and saturation value belongs to the arm. For this *2D* image histogram, image mask histogram, and probability density histogram are calculated. Following that, initial estimate is iteratively refined using the maximum likelihood approach and morphology operations (See [Figure 22-3](#)).

Figure 22-2 Arm Location and Image Warping

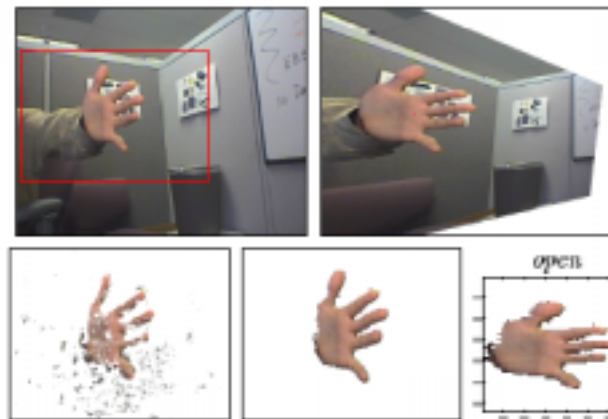
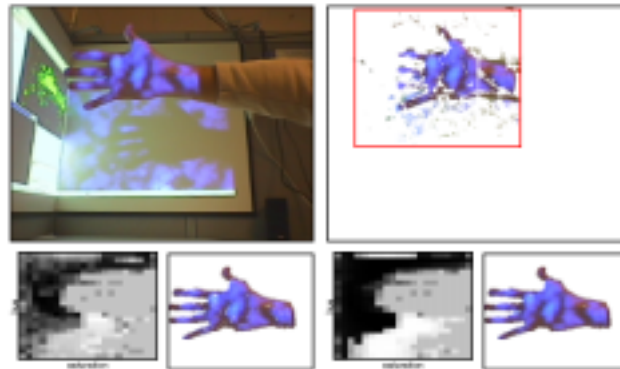


Figure 22-3 Arm Segmentation by Probability Density Estimation

The fourth step is the recognition step when normalized central moments or seven Hu moments are calculated using the resulting image mask. These invariants are used to match masks by the Mahalanobis distance metric calculation.

The functions operate with specific data of several types. Range image data is a set of 3D points in the world coordinate system calculated via the stereo correspondence algorithm. The second data type is a set of the original image indices of this set of 3D points, that is, projections on the image plane. The functions of this group enable the user:

- to locate the arm region in a set of 3D points (the functions [cvFindHandRegion](#), [cvFindHandRegionA](#)),
- create image mask from a subset of 3D points and associated subset indices around the arm center (the function [cvCreateHandMask](#)),
- calculate the homography matrix for the initial image transformation from the image plane to the plane defined by the frontal arm plane (the function [cvCalcImageHomography](#)),
- and calculate the probability density histogram for the arm location (the function [cvCalcProbDensity](#)).

Reference

cvFindHandRegion

Finds arm region in 3D range image data.

```
void cvFindHandRegion( CvPoint3D32f* points, int count, CvSeq* indexes, float*
    line, CvSize2D32f size, int flag, CvPoint3D32f* center, CvMemStorage*
    storage, CvSeq** numbers);
```

<i>points</i>	Pointer to the input 3D point data.
<i>count</i>	Numbers of the input points.
<i>indexes</i>	Sequence of the input points indices in the initial image.
<i>line</i>	Pointer to the input points approximation line.
<i>size</i>	Size of the initial image.
<i>flag</i>	Flag of the arm orientation.
<i>center</i>	Pointer to the output arm center.
<i>storage</i>	Pointer to the memory storage.
<i>numbers</i>	Pointer to the output sequence of the points indices.

Discussion

The function [cvFindHandRegion](#) finds the arm region in 3D range image data. The coordinates of the points must be defined in the world coordinates system. Each input point has user-available transform indices in the initial image (*indexes*). The function finds the arm region along the approximation line from the left, if *flag* = 0, or from the right, if *flag* = 1, in the points maximum accumulation by the points projection histogram calculation. Also the function calculates the center of the arm region and the indices of the points that lie near the arm center. The function [cvFindHandRegion](#) assumes that the arm length is equal to about 0.25m in the world coordinate system.

cvFindHandRegionA

Finds arm region in 3D range image data and defines arm orientation.

```
void cvFindHandRegionA( CvPoint3D32f* points, int count, CvSeq* indexs, float*  
    line, CvSize2D32f size, int jCenter, CvPoint3D32f* center, CvMemStorage*  
    storage, CvSeq** numbers);
```

<i>points</i>	Pointer to the input 3D point data.
<i>count</i>	Number of the input points.
<i>indexs</i>	Sequence of the input points indices in the initial image.
<i>line</i>	Pointer to the input points approximation line.
<i>size</i>	Size of the initial image.
<i>jCenter</i>	Input <i>j</i> -index of the initial image center.
<i>center</i>	Pointer to the output arm center.
<i>storage</i>	Pointer to the memory storage.
<i>numbers</i>	Pointer to the output sequence of the points indices.

Discussion

The function [cvFindHandRegionA](#) finds the arm region in the 3D range image data and defines the arm orientation (left or right). The coordinates of the points must be defined in the world coordinates system. The input parameter *jCenter* is the index *j* of the initial image center in pixels (*width*/2). Each input point has user-available transform indices on the initial image (*indexs*). The function finds the arm region along approximation line from the left or from the right in the points maximum accumulation by the points projection histogram calculation. Also the function calculates the center of the arm region and the indices of points that lie near the arm center. The function [cvFindHandRegionA](#) assumes that the arm length is equal to about 0.25m in the world coordinate system.

cvCreateHandMask

Creates arm mask on image plane.

```
void cvCreateHandMask(CvSeq* numbers, IplImage *img_mask, CvRect *roi);
```

<i>numbers</i>	Sequence of the input points indices in the initial image.
<i>img_mask</i>	Pointer to the output image mask.
<i>roi</i>	Pointer to the output arm ROI.

Discussion

The function [cvCreateHandMask](#) creates the arm mask on the image plane. The pixels of the resulting mask associated with the set of indices on the initial image (*indexes*) will have the maximum unsigned char value (255). All remaining pixels will have the minimum unsigned char value (0). The output image mask (*img_mask*) has to have the `IPL_DEPTH_8U` type and the number of channels is 1.

cvCalcImageHomography

Calculates homography matrix.

```
void cvCalcImageHomography(float *line, CvPoint3D32f* center, float  
    intrinsic[3][3], float homography[3][3]);
```

<i>line</i>	Pointer to the input 3D line.
<i>center</i>	Pointer to the input arm center.
<i>intrinsic</i>	Matrix of the intrinsic camera parameters.
<i>homography</i>	Output homography matrix.

Discussion

The function [cvCalcImageHomography](#) calculates the homograph matrix for the initial image transformation from image plane to the plane, defined by 3D arm line (See [Figure 22-1](#)). If $n_1 = (n_x, n_y)$ and $n_2 = (n_x, n_z)$ are coordinates of the normals of the 3D line projection of planes XY and XZ , then the result image homography matrix is calculated as $H = A \cdot (R_h + (I_{3 \times 3} - R_h) \cdot \bar{x}_h \cdot [0, 0, 1]) \cdot A^{-1}$, where R_h is the 3×3 matrix $R_h = R_1 \cdot R_2$, and

$$R_1 = [n_1 \times u_z, n_1, u_z], R_2 = [u_y \times n_2, u_y, n_2], u_z = [0, 0, 1]^T, u_y = [0, 1, 0]^T, \bar{x}_h = \frac{T_h}{T_z} = \left[\frac{T_x}{T_z}, \frac{T_y}{T_z}, 1 \right]^T,$$

where (T_x, T_y, T_z) is the arm center coordinates in the world coordinate system. A is the intrinsic camera parameters matrix

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

The diagonal entries f_x and f_y are the camera focal length in units of horizontal and vertical pixels and the two remaining entries c_x, c_y are the principal point image coordinates.

cvCalcProbDensity

Calculates arm mask probability density on image plane.

```
void cvCalcProbDensity (CvHistogram* hist, CvHistogram* hist_mask,
                        CvHistogram* hist_dens );
```

<i>hist</i>	Input image histogram.
<i>hist_mask</i>	Input image mask histogram.
<i>hist_dens</i>	Result probability density histogram.

Discussion

The function [cvCalcProbDensity](#) calculates the arm mask probability density from the two 2D-histograms. The input histograms have to be calculated in two channels on the initial image. If $\{h_{ij}\}$ and $\{hm_{ij}\}$, $1 \leq i \leq B_i$, $1 \leq j \leq B_j$ are input histogram and mask histogram respectively, then the result probability density histogram p_{ij} is calculated as

$$p_{ij} = \begin{cases} \frac{m_{ij}}{h_{ij}} \cdot 255, & \text{if } h_{ij} \neq 0, \\ 0, & \text{if } h_{ij} = 0, \\ 255, & \text{if } m_{ij} > h_{ij} \end{cases}$$

So the values of the p_{ij} are between 0 and 255.

cvMaxRect

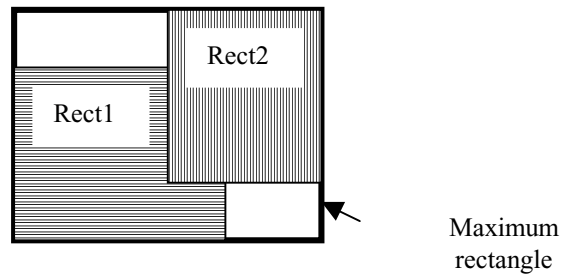
Calculates the maximum rectangle.

```
void cvMaxRect (CvRect* rect1, CvRect* rect2, CvRect* max_rect );
```

<code>rect1</code>	First input rectangle.
<code>rect2</code>	Second input rectangle.
<code>max_rect</code>	Result maximum rectangle.

Discussion

The function [cvMaxRect](#) calculates the maximum rectangle for two input rectangles ([Figure 22-4](#)).

Figure 22-4 Maximum Rectangular for Two Input Rectangles

Matrix Operations

23

This chapter describes functions for matrix operations.

Overview

OpenCV introduces special type `CvMat` for storing real single-precision or double-precision matrices. Operations supported include basic matrix arithmetics, eigen problem solution, SVD, 3D geometry and recognition-specific functions. To reduce time call overhead the special type `CvMatArray` (array of matrices) and support functions are also introduced.

Example 23-1 `CvMat` Structure Definition

```
typedef struct CvMat
{
    int rows;           // number of rows
    int cols;           // number of cols
    CvMatType type;     // type of matrix
    int step;           // not used
    union
    {
        float* fl; //pointer to the float data
        double* db; //pointer to double-precision data
    }data;
}CvMat
```

Example 23-2 `CvMatArray` Structure Definition

```
typedef struct CvMatArray
{
    int rows; //number of rows
    int cols; //number pf cols
    int type; // type of matrices
    int step; // not used
    int count; // number of matrices in aary
```

Example 23-2 CvMatArray Structure Definition (continued)

```
union
{
    float* fl;
    float* db;
}data; // pointer to matrix array data
}CvMatArray
```

Reference

cvmAlloc

Allocates memory for matrix data.

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

mat Pointer to the matrix for which memory must be allocated.

Discussion

The function [cvmAlloc](#) allocates memory for matrix data.

cvmAllocArray

Allocates memory for matrix array data.

```
void cvmAllocArray (CvMatArray*  matAr);
```

matAr Pointer to the matrix array for which memory must be allocated.

Discussion

The function [cvmAllocArray](#) allocates memory for matrix array data.

cvmFree

Frees memory allocated for matrix data.

```
void cvmFree (CvMat*  matAr);
```

mat Pointer to the matrix.

Discussion

The function [cvmFree](#) releases the memory allocated by the function [cvmAlloc](#).

cvmFreeArray

Frees memory allocated for matrix array data.

```
void cvmFreeArray (CvMat*  matAr);
```

mat Pointer to the matrix array.

Discussion

The function [cvmFreeArray](#) releases the memory allocated by the function [cvmAllocArray](#).

cvmAdd

Computes sum of two matrices.

```
void cvmAdd ( CvMat* SrcA, CvMat* SrcB, CvMat* Dst);
```

SrcA Pointer to the first source matrix.
SrcB Pointer to the second source matrix.

Dst Pointer to the destination matrix.

Discussion

The function [cvmAdd](#) adds the matrix *SrcA* to *SrcB* and stores the result in *Dst*.

$$(c = a + b, c_i = a_i + b_i).$$

cvmSub

Computes difference of two matrices.

```
void cvmSub ( CvMat* SrcA, CvMat* SrcB, CvMat* Dst );
```

SrcA Pointer to the first source matrix.

SrcB Pointer to the second source matrix.

Dst Pointer to the destination matrix.

Discussion

The function [cvmSub](#) subtracts the matrix *SrcB* from the matrix *SrcA* and stores the result in *Dst*.

$$(c = a - b, c_i = a_i - b_i).$$

cvmScale

Multiplies matrix by scalar value.

```
void cvmScale ( CvMat* Src, CvMat* Dst, double value );
```

Src Pointer to the source matrix.

Dst Pointer to the destination matrix.

value Factor.

Discussion

The function [cvmScale](#) multiplies every element of the matrix by a scalar value

$$c = \alpha a, c_i = \alpha a_i.$$

cvmDotProduct

Calculates dot product of two vectors in Euclidian metrics.

```
double cvmDotProduct( CvMat* Src1, CvMat* Src2 );
```

Src1 Pointer to the first source vector.

Src2 Pointer to the second source vector.

Discussion

The function [cvmDotProduct](#) calculates and returns the Euclidean dot product of two vectors.

$$DP = \sum_{i=1}^N a_i b_i.$$

cvmCrossProduct

Calculates cross product of two 3D vectors.

```
void cvmCrossProduct( CvMat* Src1, CvMat* Src2, CvMat* Dest );
```

Src1 Pointer to the first source vector.

Src2 Pointer to the second source vector.

Dest Pointer to the destination vector.

Discussion

The function [cvmCrossProduct](#) calculates the cross product of two 3-D vectors:

$$c = axb \quad (c_1 = a_2b_3 - a_3b_2, c_2 = a_3b_1 - a_1b_3, c_3 = a_1b_2 - a_2b_1).$$

cvmMul

Multiplies matrices.

```
void cvmMul ( CvMat* SrcA, CvMat* SrcB, CvMat* Dst );
```

<i>SrcA</i>	Pointer to the first source matrix.
<i>SrcB</i>	Pointer to the second source matrix.
<i>Dst</i>	Pointer to the destination matrix

Discussion

The function [cvmMul](#) multiplies *SrcA* by *SrcB* and stores the result in *Dst*.

$$C = AB, C_{ij} = \sum_k A_{ik} B_{kj}.$$

cvmMulTransposed

Calculates product of matrix and transposition.

```
void cvmMulTransposed (CvMat* Src, CvMat* Dst, Int order);
```

<i>Src</i>	Pointer to the source matrix.
<i>DestMatr</i>	Pointer to the destination matrix.
<i>Order</i>	Order of multipliers.

Discussion

The function [cvmMulTransposed](#) calculates the product of *SrcMatr* and its transposition.

The function evaluates $B = A^T A$ if *Order* is non-zero, $B = AA^T$ otherwise.

cvmTranspose

Transposes matrix.

```
void cvmTranspose ( CvMat* Src, CvMat*Dst );
```

Src Pointer to the source matrix.

Dst Pointer to the destination matrix.

Discussion

The function [cvmTranspose](#) transposes *Src* and stores result in *Dst*.

$B = A^T$, $B_{ij} = A_{ji}$.

cvmInvert

Inverses matrix.

```
void cvmInvert ( CvMat* Src, CvMat*Dst );
```

Src Pointer to the source matrix.

Dst Pointer to the destination matrix.

Discussion

The function [cvmInvert](#) inverts *Src* and stores the result in *Dst*.

$B = A^{-1}$, $AB = BA = I$,

cvmTrace

Returns trace of matrix.

```
double cvmTrace ( CvMat* mat );
```

mat Pointer to the source matrix.

Discussion

The function [cvmTrace](#) returns the sum of diagonal elements of the matrix *mat* .

cvmDet

Returns determinant of matrix.

```
double cvmDet ( CvMat* mat );
```

mat Pointer to the source matrix.

Discussion

The function [cvmDet](#) returns the determinant of the matrix *mat*.

cvmCopy

Copies one matrix to another.

```
void cvmCopy ( CvMat* Src, CvMat* Dst );
```

Src Pointer to the source matrix.
Dest Pointer to the destination matrix.

Discussion

The function [cvmCopy](#) copies the matrix *Src* to the matrix *Dest*.

$$B = A, B_{ij} = A_{ij}.$$

cvmSetZero_32f

Sets matrix to zero.

```
void cvmSetZero_32f ( CvMat* mat );
```

mat Pointer to the matrix to be set to zero.

Discussion

The function [cvmSetZero_32f](#) sets the matrix to zero.

$$A = 0, A_{ij} = 0.$$

cvmSetIdentity

Sets matrix to identity.

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

mat Pointer to the matrix to be set to identity.

Discussion

The function [cvmSetIdentity](#) sets the matrix to identity.

$$A = E, A_{ij} = \delta_{ij}.$$

cvmMahalonobis

Calculates Mahalonobis distance between vectors.

```
double cvmMahalonobis ( CvMat* SrcA, CvMat* SrcB, CvMat* mat );
```

SrcA Pointer to the first source vector.

SrcB Pointer to the second source vector.

Matr Pointer to the weighted matrix.

Discussion

The function [cvmMahalonobis](#) calculates the weighted distance between two vectors and returns it:

$$Dist = \sqrt{\sum_i \sum_j T_{ij} (a_i - b_i)(a_j - b_j)}.$$

Here, T matrix is supposed to be inverse of covariation matrix.

cvmSVD

Calculates singular value decomposition.

```
void cvmSVD ( CvMat* Src, CvMat* Orth, CvMat* Diag );
```

Src Pointer to the source matrix.

Orth Pointer to the matrix where the orthogonal matrix will be saved.

Diag Pointer to the matrix where the diagonal matrix will be saved.

Discussion

The function [cvmSVD](#) decomposes the source matrix to product of two orthogonal and one diagonal matrices.

$A = A1' \times Diag \times Orth$, where $A1$ is orthogonal matrix and stored in A , $Diag$ is diagonal matrix and $Orth$ is another orthogonal matrix. If A is square matrix, $A1$ and $Orth$ will be the same.



NOTE. The function `cvmSVD` destroys the source matrix `Src`. Therefore, in case the source matrix is needed after decomposition, the user is advised to clone it before running this function.

cvmEigenVV

Computes eigenvalues and eigenvectors.

```
void cvmEigenVV ( CvMat* Src, CvMat* evecs CvMat* evals, Double eps );
```

<i>Src</i>	Pointer to the source matrix.
<i>evecs</i>	Pointer to the matrix where eigenvectors must be stored.
<i>evals</i>	Pointer to the matrix where eigenvalues must be stored.
<i>eps</i>	Accuracy of diagonalization.

Discussion

The function [cvmEigenVV](#) computes the eigenvalues and eigenvectors of the matrix *Src* and stores them in the parameters *evals* and *evecs* correspondingly. Jacobi method is used.



NOTE. The function `cvmEigenVV` destroys the source matrix `Src`. Therefore, if the source matrix is needed after eigenvalues have been calculated, the user is advised to clone it before running the function `cvmEigenVV`.

cvmPerspectiveProject

Implements general transform of 3D vector array.

```
void cvmPerspectiveProject (CvMat* mat, CvMatArray src, CvMatArray dst);
```

<i>mat</i>	4x4 matrix.
<i>src</i>	Source array of 3D vectors.
<i>dst</i>	Destination array of 3D vectors.

Discussion

The function [cvmPerspectiveProject](#) maps every input 3D vector $(x, y, z)^T$ to $(x'/w, y'/w, z'/w)^T$, where

$$(x', y', z', w')^T = (mat) \times (x, y, z, 1)^T \quad \text{and} \quad w = \begin{cases} w', & w' \neq 0 \\ 1, & w' = 0 \end{cases}.$$

This chapter describes functions that operate on eigen objects.

Overview

Let us define an object $u = \{u_1, u_2, \dots, u_n\}$ in the n -dimensional space as a sequence of values u_1 that could be vectors, images, etc. Images may either have or not have ROI. Let us assume that we have a group of input objects $u^i = \{u_1^i, u_2^i, \dots, u_n^i\}$, $i = 1, \dots, m$; usually $m \ll n$. Averaged, or mean, object $\bar{u} = \{\bar{u}_1, \bar{u}_2, \dots, \bar{u}_n\}$ of this group is defined as follows:

$$\bar{u}_l = \frac{1}{m} \sum_{k=1}^m u_l^k.$$

Covariance matrix $C = [c_{ij}]$ is a square symmetric matrix $m \times m$:

$$c_{ij} = \sum_{l=1}^m (u_l^i - \bar{u}_l) \cdot (u_l^j - \bar{u}_l).$$

Eigen objects basis $e^i = \{e_1^i, e_2^i, \dots, e_n^i\}$, $i = 1, \dots, m_1 \leq m$ of the input objects group may be calculated using the following relation:

$$e_l^i = \frac{1}{\sqrt{\lambda_i}} \sum_{k=1}^m v_k^i \cdot (u_l^k - \bar{u}_l),$$

where λ_i and $v^i = \{v_1^i, v_2^i, \dots, v_m^i\}$ are eigenvalues and the corresponding eigenvectors of matrix C .

Any input object u^i as well as any other object u may be decomposed in the eigen objects m_1-D sub-space. Decomposition coefficients of the object u are:

$$w_i = \sum_{l=1} e_l^i \cdot (u_l - \bar{u}_l) \quad .$$

Using these coefficients, we may calculate projection $\tilde{u} = \{\tilde{u}_1, \tilde{u}_2, \dots, \tilde{u}_n\}$ of the object u to the eigen objects sub-space, or, in other words, restore the object u in that sub-space:

$$\tilde{u}_l = \sum_{k=1}^{m_1} w_k e_l^k + \bar{u}_l$$

Reference

cvCalcCovarMatrixEx

Calculates covariance matrix for group of input objects.

```
void cvCalcCovarMatrixEx( int nObjects, void* input, int ioFlags, int
    ioBufSize, uchar* buffer, void* userData, IplImage* avg, float*
    covarMatrix );
```

<i>nObjects</i>	Number of source objects.
<i>input</i>	Pointer either to the array of <i>IplImage</i> input objects or to the read callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags.
<i>ioBufSize</i>	Input/output buffer size.
<i>buffer</i>	Pointer to input/output buffer.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.

<i>avg</i>	Averaged object.
<i>covarMatrix</i>	Covariance matrix. Output parameter; must be allocated before the call.

Discussion

The function [cvCalcCovarMatrixEx](#) calculates a covariance matrix of the input objects group using previously calculated averaged object. Depending on *ioFlags* parameter it may be used either in direct access or callback mode. If *ioFlags* is not `CV_EIGOBJ_NO_CALLBACK`, buffer must be allocated before the function [cvCalcCovarMatrixEx](#).

cvCalcEigenObjects

Calculates orthonormal eigen basis and averaged object for group of input objects.

```
void cvCalcEigenObjects ( int nObjects, void* input, void* output, int ioFlags,
    int ioBufSize, void* userData, CvTermCriteria* calcLimit, IplImage* avg,
    float* eigVals;
```

<i>nObjects</i>	Number of source objects.
<i>input</i>	Pointer either to the array of <i>IplImage</i> input objects or to the read callback function (depending on the parameter <i>ioFlags</i>).
<i>output</i>	Pointer either to the array of eigen objects or to the write callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags.
<i>ioBufSize</i>	Input/output buffer size in bytes. The size is zero, if unknown.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>calcLimit</i>	Determines conditions for the calculation to be finished.
<i>avg</i>	Averaged object.

eigVals Pointer to the eigenvalues array in the descending order; may be NULL.

Discussion

The function [cvCalcEigenObjects](#) calculates orthonormal eigen basis and averaged object for group of input objects. Depending on *ioFlags* parameter it may be used either in direct access or callback mode. Depending on the parameter *calcLimit*, calculations are finished either if the eigen faces number reaches a certain value or if the relation between the current and the largest eigenvalues comes down to a certain value, or any of the above conditions takes place. The value *calcLimit->type* must be CV_TERMCRIT_NUMB, CV_TERMCRIT_EPS, or CV_TERMCRIT_NUMB | CV_TERMCRIT_EPS. The function returns the real values *calcLimit->maxIter* and *calcLimit->epsilon*.

Averaged object is also calculated by the function [cvCalcEigenObjects](#), but it must be created previously. Calculated eigen objects are ordered according to the corresponding eigenvalues in the descending order.

The parameter *eigVals* may be equal to NULL, if eigenvalues are not needed.

The function [cvCalcEigenObjects](#) uses the function [cvCalcCovarMatrixEx](#).

cvCalcDecompCoeff

Calculates decomposition coefficient of input object.

```
double cvCalcDecompCoeff( IplImage* obj, IplImage* eigObj, IplImage* avg );
```

obj Input object.
eigObj Eigen object.
avg Averaged object.

Discussion

The function [cvCalcDecompCoeff](#) calculates one decomposition coefficient of the input object using the previously calculated eigen object and the averaged object.

cvEigenDecomposite

Calculates all decomposition coefficients for input object.

```
void cvEigenDecomposite( IplImage* obj, int nEigObjs, void* eigInput, int  
    ioFlags, void* userData, IplImage* avg, float* coeffs );
```

<i>obj</i>	Input object.
<i>nEigObjs</i>	Number of eigen objects.
<i>eigInput</i>	Pointer either to the array of <i>IplImage</i> eigen objects or to the read callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>avg</i>	Averaged object.
<i>coeffs</i>	Calculated coefficients; output parameter.

Discussion

The function [cvEigenDecomposite](#) calculates all decomposition coefficients for the input object using the previously calculated eigen objects basis and the averaged object. Depending on *ioFlags* parameter it may be used either in direct access or callback mode.

cvEigenProjection

Calculates object projection to the eigen sub-space.

```
void cvEigenProjection ( int nEigObjs, void* eigInput, int ioFlags, void*
    userData, float* coeffs, IplImage* avg, IplImage* proj );
```

<i>nEigObjs</i>	Number of eigen objects.
<i>eigInput</i>	Pointer either to the array of <i>IplImage</i> input objects or to the read callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>coeffs</i>	Previously calculated decomposition coefficients.
<i>avg</i>	Averaged object.
<i>proj</i>	Decomposed object projection to the eigen sub-space.

Discussion

The function [cvEigenProjection](#) calculates an object projection to the eigen sub-space or, in other words, restores an object using previously calculated eigen objects basis, averaged object, and decomposition coefficients of the restored object. Depending on *ioFlags* parameter it may be used either in direct access or callback mode.

Use of Functions

The functions of the eigen objects group have been developed to be used for any number of objects, even if their total size exceeds free RAM size. So the functions may be used in two main modes.

Direct access mode is the best choice if the size of free RAM is sufficient for all input and eigen objects allocation. This mode is set if the parameter *ioFlags* is equal to `CV_EIGOBJ_NO_CALLBACK`. In this case *input* and *output* parameters are pointers to

arrays of input (output) objects of `IplImage*` type. The parameters `ioBufSize` and `userData` are not used. An example of the function [cvCalcEigenObjects](#) used in direct access mode is given below.

Example 24-1 Use of function `cvCalcEigenObjects` in Direct Access Mode

```
IplImage** objects;
IplImage** eigenObjects;
IplImage*  avg;
float*     eigVals;
CvSize     size = cvSize( nx, ny );
.....
if( !( eigVals = (float*) cvAlloc( nObjects*sizeof(float) ) ) )
    __ERROR_EXIT__;
if( !( avg = cvCreateImage( size, IPL_DEPTH_32F, 1 ) ) )
    __ERROR_EXIT__;
for( i=0; i< nObjects; i++ )
{
    objects[i]      = cvCreateImage( size, IPL_DEPTH_8U, 1 );
    eigenObjects[i] = cvCreateImage( size, IPL_DEPTH_32F, 1 );
    if( !( objects[i] & eigenObjects[i] ) )
        __ERROR_EXIT__;
}
.....
cvCalcEigenObjects ( nObjects,
                    (void*)objects,
                    (void*)eigenObjects,
                    CV_EIGOBJ_NO_CALLBACK,
                    0,
                    NULL,
                    calcLimit,
                    avg,
                    eigVals );
```

The *callback mode* is the right choice in case when the number and the size of objects are large, which happens when all objects and/or eigen objects cannot be allocated in free RAM. In this case input/output information may be read/written and developed by portions. Such regime is called callback mode and is set by the parameter `ioFlags`. Three kinds of the callback mode may be set:

`IoFlag = CV_EIGOBJ_INPUT_CALLBACK`, only input objects are read by portions;

`IoFlag = CV_EIGOBJ_OUTPUT_CALLBACK`, only eigen objects are calculated and written by portions;

IoFlag = CV_EIGOBJ_BOTH_CALLBACK, or *IoFlag* = CV_EIGOBJ_INPUT_CALLBACK | CV_EIGOBJ_OUTPUT_CALLBACK, both processes take place. If one of the above modes is realized, the parameters *input* and *output*, both or either of them, are pointers to read/write callback functions. These functions must be written by the user; their prototypes are the same:

```
CvStatus callback_read ( int ind, void* buffer, void* userData);
```

```
CvStatus callback_write( int ind, void* buffer, void* userData);
```

<i>ind</i>	Index of the read or written object.
<i>buffer</i>	Pointer to the start memory address where the object will be allocated.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.

The user must define the user data structure which may carry all information necessary to read/write procedure, such as the start address or file name of the first object on the HDD or any other device, row length and full object length, etc.

If *ioFlag* is not equal to CV_EIGOBJ_NO_CALLBACK, the function [cvCalcEigenObjects](#) allocates a buffer in RAM for objects/eigen objects portion storage. The size of the buffer may be defined either by the user or automatically. If the parameter *ioBufSize* is equal to 0, or too large, the function will define the buffer size. The read data must be located in the buffer compactly, that is, row after row, without alignment and gaps.

An example of the user data structure, i/o callback functions, and the use of the function [cvCalcEigenObjects](#) in the callback mode is shown below.

Example 24-2 User Data Structure, I/O Callback Functions, and Use of Function [cvCalcEigenObjects](#) in Callback Mode

```
// User data structure
typedef struct _UserData
{
    int      objLength; /* Obj. length (in elements, not in bytes !) */
    int      step;      /* Obj. step (in elements, not in bytes !) */
    CvSize   size;      /* ROI or full size */
    CvPoint  roiIndent;
    char*    read_name;
    char*    write_name;
} UserData;
```

Example 24-2 User Data Structure, I/O Callback Functions, and Use of Function cvCalcEigenObjects in Callback Mode (continued)

```
//-----
--
// Read callback function
CvStatus callback_read_8u ( int ind, void* buffer, void* userData)
{
    int i, j, k = 0, m;
    UserData* data = (UserData*)userData;
    uchar* buff = (uchar*)buf;
    char name[32];
    FILE *f;

    if( ind<0 ) return CV_StsBadArg;
    if( buf==NULL || userData==NULL ) CV_StsNullPtr;

    for(i=0; i<28; i++)
    {
        name[i] = data->read_name[i];
        if(name[i]=='.' || name[i]==' '))break;
    }
    name[i] = 48 + ind/100;
    name[i+1] = 48 + (ind%100)/10;
    name[i+2] = 48 + ind%10;
    if((f=fopen(name, "r"))==NULL) return CV_BadCallBack;
    m = data->roiIndent.y*step + data->roiIndent.x;

    for( i=0; i<data->size.height; i++, m+=data->step )
    {
        fseek(f, m , SEEK_SET);
        for( j=0; j<data->size.width; j++, k++ )
            fread(buff+k, 1, 1, f);
    }

    fclose(f);
    return CV_StsOk;
}
//-----
// Write callback function
cvStatus callback_write_32f ( int ind, void* buffer, void* userData)
{
    int i, j, k = 0, m;
    UserData* data = (UserData*)userData;
    float* buff = (float*)buf;
    char name[32];
    FILE *f;

    if( ind<0 ) return CV_StsBadArg;
    if( buf==NULL || userData==NULL ) CV_StsNullPtr;
```

Example 24-2 User Data Structure, I/O Callback Functions, and Use of Function `cvCalcEigenObjects` in Callback Mode (continued)

```

    for(i=0; i<28; i++)
    {
        name[i] = data->read_name[i];
        if(name[i]!='.' || name[i]!=' '))break;
    }
    if((f=fopen(name, "w"))==NULL) return CV_BadCallBack;
    m = 4 * (ind*data->objLength + data->roiIndent.y*step
            + data->roiIndent.x);

    for( i=0; i<data->size.height; i++, m+=4*data->step )
    {
        fseek(f, m , SEEK_SET);
        for( j=0; j<data->size.width; j++, k++ )
            fwrite(buff+k, 4, 1, f);
    }

    fclose(f);
    return CV_StsOk;
}
//-----
--
// fragments of the main function
{
    . . . . .
    int bufSize = 32*1024*1024; //32 MB RAM for i/o buffer
    float* avg;
    cv UserData data;
    cvStatus r;
    cvStatus (*read_callback)( int ind, void* buf, void* userData)=
        read_callback_8u;
    cvStatus (*write_callback)( int ind, void* buf, void* userData)=
        write_callback_32f;
    cvInput* u_r = (cvInput*)&read_callback;
    cvInput* u_w = (cvInput*)&write_callback;
    void* read_   = (u_r)->data;
    void* write_  = (u_w)->data;
    . . . . .
    data->read_name = "input";
    data->write_name = "eigens";
    avg = (float*)cvAlloc(sizeof(float) * obj_width * obj_height );

    cvCalcEigenObjects( obj_number,
                        read_,
                        write_,
                        CV_EIGOBJ_BOTH_CALLBACK,
                        bufSize,

```

**Example 24-2 User Data Structure, I/O Callback Functions, and Use of Function
cvCalcEigenObjects in Callback Mode (continued)**

```
        (void*)&data,  
        &limit,  
        avg,  
        eigVal );  
    . . . . .  
}
```

Embedded Hidden Markov Models

25

This chapter describes functions for using Embedded Hidden Markov Models (HMM) in face recognition task.

Overview

HMM Structures

In order to support embedded models the user must define structures to represent 1D HMM and 2D embedded HMM model.

```
typedef struct _CvEHMM
{
    int level;
    int num_states;
    float* transP;
    float** obsProb;
    union
    {
        CvEHMMState* state;
        struct _CvEHMM* ehmm;
    } u;
}CvEHMM;
```

Below is the description of the `CvEHMM` fields:

<i>level</i>	Level of embedded HMM. If <i>level</i> ==0, HMM is most external. In 2D HMM there are two types of HMM: 1 external and several embedded. External HMM has <i>level</i> ==1, embedded HMMs have <i>level</i> ==0.
<i>num_states</i>	Number of states in 1D HMM.
<i>transP</i>	State-to-state transition probability, square matrix (<i>num_state</i> × <i>num_state</i>).
<i>obsProb</i>	Observation probability matrix.
<i>state</i>	Array of HMM states. For the last-level HMM, that is, an HMM without embedded HMMs, HMM states are “real”.
<i>ehmm</i>	Array of embedded HMMs. If HMM is not last-level, then HMM states are not “real” and they are HMMs.

For representation of observations the following structure is defined:

```
typedef struct CvImgObsInfo
{
    int obs_x;
    int obs_y;
    int obs_size;
    float** obs;
    int* state;
    int* mix;
}CvImgObsInfo;
```

This structure is used for storing observation vectors extracted from 2D image.

<i>obs_x</i>	Number of observations in the horizontal direction.
<i>obs_y</i>	Number of observations in the vertical direction.
<i>obs_size</i>	Length of every observation vector.
<i>obs</i>	Pointer to observation vectors stored consequently. Number of vectors is <i>obs_x</i> * <i>obs_y</i> .
<i>state</i>	Array of indices of states, assigned to every observation vector.

mix Index of mixture component, corresponding to the observation vector within an assigned state.

Reference

cvCreate2DHMM

Creates 2D embedded HMM.

```
CvEHMM* cvCreate2DHMM( int* stateNumber, int* numMix, int obsSize );
```

stateNumber Array, the first element of the which specifies the number of superstates in the HMM. All subsequent elements specify the number of states in every embedded HMM, corresponding to each superstate. So, the length of the array is *stateNumber*[0]+1.

numMix Array with numbers of Gaussian mixture components per each internal state. The number of elements in the array is equal to number of internal states in the HMM, that is, superstates (or external states) are not counted here.

obsSize Size of observation vectors to be used with created HMM.

Discussion

The function [cvCreate2DHMM](#) returns created structure of the type `CvEHMM` with specified parameters.

cvRelease2DHMM

Releases 2D embedded HMM.

```
void cvRelease2DHMM(CvEHMM** hmm);
```

hmm Address of pointer to HMM to be released.

Discussion

The function [cvRelease2DHMM](#) frees all memory used by HMM and clears the pointer to HMM.

cvCreateObsInfo

Creates structure to store image observation vectors.

```
CvImgObsInfo* cvCreateObsInfo( CvSize numObs, int obsSize );
```

<i>numObs</i>	Numbers of observations in the horizontal and vertical directions. For the given image and scheme of extracting observations the parameter can be computed via the macro <code>CV_COUNT_OBS(roi, dctSize, delta, numObs)</code> , where <i>roi</i> , <i>dctSize</i> , <i>delta</i> , <i>numObs</i> are the pointers to structures of the type <code>CvSize</code> . The pointer <i>roi</i> means size of <i>roi</i> of image observed, <i>numObs</i> is the output parameter of the macro.
<i>obsSize</i>	Size of observation vectors to be stored in the structure.

Discussion

The function [cvCreateObsInfo](#) creates new structures to store image observation vectors. For definitions of the parameters *roi*, *dctSize*, and *delta* see the specification of the function [cvImgToObs_DCT](#).

cvReleaseObsInfo

Releases observation vectors structure.

```
void cvReleaseObsInfo( CvImgObsInfo** obs_info );
```

<i>obs_info</i>	Address of the pointer to the structure <code>CvImgObsInfo</code> .
-----------------	---

Discussion

The function [cvReleaseObsInfo](#) frees all memory used by observations and the clears pointer to the structure `CvImgObsInfo`.

cvImgToObs_DCT

Extracts observation vectors from image.

```
void cvImgToObs_DCT( IplImage* image, float* obs, CvSize dctSize, CvSize
                    obsSize, CvSize delta );
```

<i>image</i>	Input image.
<i>obs</i>	Pointer to consequently stored observation vectors.
<i>dctSize</i>	Size of image blocks for which DCT coefficients are to be computed.
<i>obsSize</i>	Number of the lowest DCT coefficients in the horizontal and vertical directions that will be put into the observation vector.
<i>delta</i>	Shift in pixels between two consecutive image blocks in the horizontal and vertical directions.

Discussion

The function [cvImgToObs_DCT](#) extracts observation vectors, that is, DCT coefficients, from the image. The user must pass `obs_info.obs` as the parameter `obs` to use this function with other HMM functions and use the structure `obs_info` of the `CvImgObsInfo` type.

Example 25-1

```
CvImgObsInfo* obs_info;
.....
CvImgToObs_DCT( image,
                obs_info->obs, //!!!
                dctSize, obsSize, delta );
```

cvUniformImgSegm

Performs uniform segmentation of image observations by HMM states.

```
void cvUniformImgSegm( CvImgObsInfo* obs_info, CvEHMM* hmm);
```

obs_info Observations structure.

hmm HMM structure.

Discussion

The function [cvUniformImgSegm](#) segments image observations by HMM states uniformly (see [Figure 25-1](#) for 2D embedded HMM with 5 superstates and 3, 6, 6, 6, 3 internal states of every corresponding superstate).

Figure 25-1 Initial Segmentation for 2D Embedded HMM



cvInitMixSegm

Segments all observations within every internal state of HMM by state mixture components.

```
void cvInitMixSegm( CvImgObsInfo** obs_info_array, int num_img, CvEHMM* hmm);
```

obs_info_array Array of pointers to the observation structures.

num_img Length of above array.

hmm HMM.

Discussion

The function [cvInitMixSegm](#) takes a group of observations from several training images already segmented by states and splits a set of observation vectors within every internal HMM state into as many clusters as number of mixture components in the state.

cvEstimateHMMStateParams

Estimates all parameters of every HMM state.

```
void cvEstimateHMMStateParams(CvImgObsInfo** obs_info_array, int num_img,
                             CvEHMM* hmm);
```

<i>obs_info_array</i>	Array of pointers to the observation structures.
<i>num_img</i>	Length of the array.
<i>hmm</i>	HMM.

Discussion

The function [cvEstimateHMMStateParams](#) computes all inner parameters of every HMM state, including Gaussian means, variances, etc.

cvEstimateTransProb

Computes transition probability matrices for embedded HMM.

```
void cvEstimateTransProb( CvImgObsInfo** obs_info_array, int num_img, CvEHMM*
                          hmm);
```

<i>obs_info_array</i>	Array of pointers to the observation structures.
-----------------------	--

num_img Length of above array.
hmm HMM.

Discussion

The function [cvEstimateTransProb](#) uses current segmentation of image observations to compute transition probability matrices for all embedded and external HMMs.

cvEstimateObsProb

Computes probability of every observation of several images.

```
void cvEstimateObsProb( CvImgObsInfo* obs_info, CvEHMM* hmm );
```

obs_info Observation structure.
hmm HMM structure.

Discussion

The function [cvEstimateObsProb](#) computes Gaussian probabilities of each observation to occur in each of the internal HMM states.

cvEViterbi

Executes Viterbi algorithm for embedded HMM.

```
Float cvEViterbi( CvImgObsInfo* obs_info, CvEHMM* hmm );
```

obs_info Observation structure.
hmm HMM structure.

Discussion

The function [cvEViterbi](#) executes Viterbi algorithm for embedded HMM. Viterbi algorithm evaluates the likelihood of the best match between given image observations and given HMM and performs segmentation of image observations by HMM states. The segmentation is done on the basis of the match found.

cvMixSegmL2

*Segments observations from all training images
by mixture components of newly assigned states.*

```
void cvMixSegmL2( CvImgObsInfo** obs_info_array, int num_img, CvEHMM* hmm);
```

<i>obs_info_array</i>	Array of pointers to the observation structures.
<i>num_img</i>	Length of the array.
<i>hmm</i>	HMM.

Discussion

The function [cvMixSegmL2](#) segments observations from all training images by mixture components of newly Viterbi algorithm-assigned states. The function uses Euclidean distance to group vectors around existing mixtures centers.

This chapter describes simple drawing functions.

Overview

The functions described in this chapter are intended mainly to mark out recognized or tracked features in the image. With tracking or recognition pipeline implemented it is often necessary to represent results of the processing in the image. Despite the fact that most Operating Systems have advanced graphic capabilities, they often require an image, where one is going to draw, to be created by special system functions. For example, under Win32 a graphic context (DC) must be created in order to use GDI draw functions. Therefore, several simple functions for 2D vector graphic rendering have been created. All of them are platform-independent and work with `IplImage` structure. Now supported image formats include byte-depth (`depth == IPL_DEPTH_8U` or `depth == IPL_DEPTH_8S`) single channel (grayscale) or three channel (RGB or, more exactly, BGR (that is, blue channel goes first) images).

There are several notes that can be made for each drawing function in the library, therefore, they are put below - not in discussion sections:

- All of the functions take `color` parameter that means brightness for grayscale images and RGB color for color images. In the latter case a value, passed to the function, can be composed via `CV_RGB` macro that is defined as:

```
#define CV_RGB(r,g,b) (((r)&255) << 16)|(((g)&255) << 8)|((b)&255))
```

- Any function in the group takes one or more points (`CvPoint` structure instance(s)) as input parameters. Point coordinates are counted from top-left ROI corner for top-origin images and from bottom-left ROI corner for bottom-origin images.

- All the functions are divided into two classes - with or without antialiasing. For several functions there exist antialiased versions that end with AA suffix. The coordinates, passed to AA-functions, can be specified with sub-pixel accuracy, that is they can have several fractional bits, which number is passed via *scale* parameter. For example, if `cvCircleAA` function is passed `center = cvPoint(34,18)` and `scale = 2` then the actual center coordinates will be `(34/4.,19/4.)==(16.5,4.75)`.
- Simple (that is, non-antialiased) functions have *thickness* parameter that specifies thickness of lines a figure is drawn with. For some functions the parameter may take negative values. It causes the functions to draw a filled figure instead of drawing its outline. To improve code readability one may use constant `CV_FILLED = -1` as a *thickness* value to draw filled figures.

Reference

cvLine

Draws simple or thick line segment.

```
void cvLine( IplImage* img, CvPoint pt1, CvPoint pt2, int color, int
            thickness=1 );
```

<i>img</i>	Image.
<i>pt1</i>	First point of the line segment.
<i>pt2</i>	Second point of the line segment.
<i>color</i>	Line color (RGB) or brightness (grayscale image).
<i>thickness</i>	Line thickness.

Discussion

The function [cvLine](#) draws the line segment between *pt1* and *pt2* points in the image. The line is clipped by the image or ROI rectangle. The Bresenham algorithm is used for simple line segments. Thick lines are drawn with rounding endings. To specify the line color the user may use the macro `CV_RGB (r, g, b)` that makes a 32-bit color value from the color components.

cvLineAA

Draws antialiased line segment.

```
void cvLineAA( IplImage* img, CvPoint pt1, CvPoint pt2, int color, int scale=0
);
```

<i>img</i>	Image.
<i>pt1</i>	First point of the line segment.
<i>pt2</i>	Second point of the line segment.
<i>color</i>	Line color (RGB) or brightness (grayscale image).
<i>scale</i>	Number of fractional bits in the end point coordinates.

Discussion

The function [cvLineAA](#) draws the line segment between *pt1* and *pt2* points in the image. The line is clipped by the image or ROI rectangle. Drawing algorithm includes some sort of Gaussian filtering to get smooth picture. To specify the line color the user may use the macro `CV_RGB (r, g, b)` that makes a 32-bit color value from the color components.

cvRectangle

Draws simple, thick or filled rectangle.

```
void cvRectangle( IplImage* img, CvPoint pt1, CvPoint pt2,  
                 int color, int thickness );
```

<i>img</i>	Image.
<i>pt1</i>	One of the rectangle vertices.
<i>pt2</i>	Opposite rectangle vertex.
<i>color</i>	Line color (RGB) or brightness (grayscale image).
<i>thickness</i>	Thickness of lines that make up the rectangle.

Discussion

The function [cvRectangle](#) draws rectangle with two opposite corners *pt1* and *pt2*. If the parameter *thickness* is positive or zero, the outline of the rectangle is drawn with that thickness, otherwise a filled rectangle is drawn.

cvCircle

Draws simple, thick or filled circle.

```
void cvCircle( IplImage* img, CvPoint center, int radius, int color,  
              int thickness=1 );
```

<i>img</i>	Image where the line is drawn.
<i>center</i>	Center of the circle.
<i>radius</i>	Radius of the circle.
<i>color</i>	Circle color (RGB) or brightness (grayscale image).
<i>thickness</i>	Thickness of the circle outline if positive, otherwise indicates that a filled circle should be drawn.

Discussion

The function [cvCircle](#) draws a simple or filled circle with given center and radius. The circle is clipped by ROI rectangle. The Bresenham algorithm is used both for simple and filled circles. To specify the circle color the user may use the macro `CV_RGB(r, g, b)` that makes a 32-bit color value from the color components.

cvEllipse

Draws simple or thick elliptic arc or fills ellipse sector:

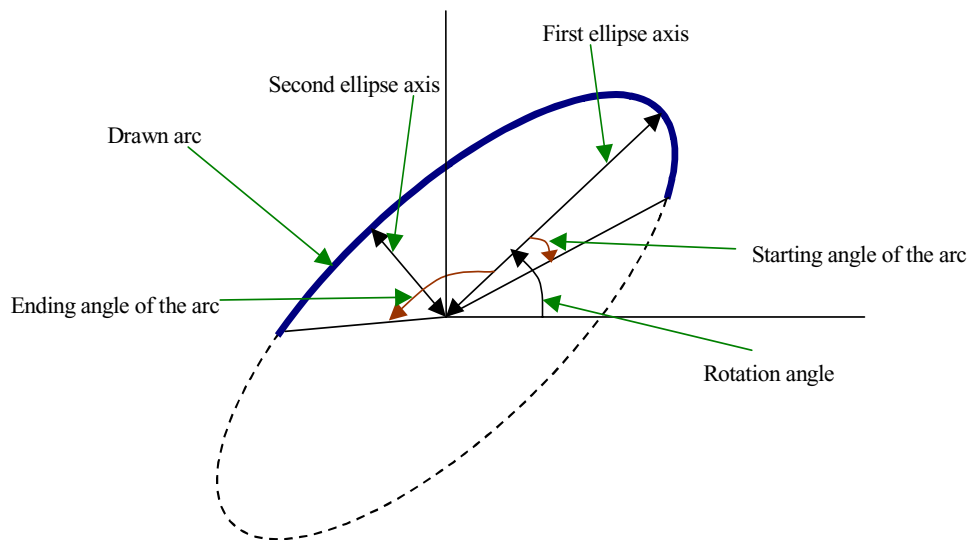
```
void cvEllipse( IplImage* img, CvPoint center, CvSize axes, double angle,
                double start_angle, double end_angle, int color, int thickness=1 );
```

<i>img</i>	Image.
<i>center</i>	Center of the ellipse.
<i>axes</i>	Length of ellipse axes.
<i>angle</i>	Rotation angle.
<i>start_angle</i>	Starting angle of the elliptic arc.
<i>end_angle</i>	Ending angle of the elliptic arc.
<i>color</i>	Ellipse color (RGB) or brightness (grayscale image).
<i>thickness</i>	Thickness of the ellipse arc.

Discussion

The function [cvEllipse](#) draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by ROI rectangle. Generalized Bresenham algorithm for conic section is used for simple elliptic arcs here, and piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are given in degrees. The meaning of parameters is shown in [Figure 26-1](#):

Figure 26-1



cvEllipseAA

Draws antialiased elliptic arc.

```
void cvEllipseAA( IplImage* img, CvPoint center, CvSize axes, double angle,
    double start_angle, double end_angle, int color, int scale=0 );
```

<i>img</i>	Image.
<i>center</i>	Center of the ellipse.
<i>axes</i>	Length of ellipse axes.
<i>angle</i>	Rotation angle.
<i>start_angle</i>	Starting angle of the elliptic arc.
<i>end_angle</i>	Ending angle of the elliptic arc.

<i>color</i>	Ellipse color (RGB) or brightness (grayscale image).
<i>scale</i>	Specifies the number of fractional bits in the center coordinates and axes sizes.

Discussion

The function [cvEllipseAA](#) draws an antialiased elliptic arc. The arc is clipped by ROI rectangle. Generalized Bresenham algorithm for conic section is used for simple elliptic arcs here, and piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are in degrees. The meaning of parameters is shown in [Figure 26-1](#).

cvFillPoly

Fills polygons interior.

```
void cvFillPoly( IplImage* img, CvPoint** pts, int* npts, int contours,
                int color );
```

<i>img</i>	Image.
<i>pts</i>	Array of pointers to polygons.
<i>npts</i>	Array of array counters or a single counter.
<i>contours</i>	Number of contours that bind the filled region.
<i>color</i>	Polygon color (RGB) or brightness (grayscale image).

Discussion

The function [cvFillPoly](#) fills an area, bounded by several polygonal contours. The function fills complex areas, e.g., areas with holes, contour self-intersection, etc.

cvFillConvexPoly

Fills convex polygon.

```
void cvFillConvexPoly( IplImage* img, CvPoint* pts, int npts, int color );
```

<i>img</i>	Image.
<i>pts</i>	Array of pointers to a single polygon.
<i>npts</i>	Array of array counters or a single counter.
<i>color</i>	Polygon color (RGB) or brightness (grayscale image).

Discussion

The function [cvFillConvexPoly](#) fills convex polygon interior. The function [cvFillConvexPoly](#) is much faster than the function [cvFillPoly](#) and fills not only the convex polygon but any monotonic polygon, that is, a polygon, whose contour intersects every horizontal line (scan line) twice at the most.

cvPolyLine

Draws simple or thick polygons.

```
void cvPolyLine( IplImage* img, CvPoint** pts, int* npts, int contours,  
is_closed, int color, int thickness=1 );
```

<i>img</i>	Image.
<i>pts</i>	Array of pointers to polylines.
<i>npts</i>	Array of polyline counters or a single counter.
<i>contours</i>	Number of polyline contours.
<i>is_closed</i>	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 Polygon color (RGB) or brightness (grayscale image).
thickness Thickness of the polyline edges.

Discussion

The function [cvPolyLine](#) draws a set of simple or thick polylines.

cvPolyLineAA

Draws antialiased polygons.

```
void cvPolyLineAA( IplImage* img, CvPoint** pts, int* npts, int contours,  
                  is_closed, int color, int scale=0 );
```

img Image.
pts Array of pointers to polylines.
npts Array of polyline counters or a single counter.
contours Number of polyline contours.
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 Polygon color (RGB) or brightness (grayscale image).
scale Specifies number of fractional bits in the coordinates of polyline vertices.

Discussion

The function [cvPolyLineAA](#) draws a set of antialiased polylines.

cvInitFont

Initializes font structure.

```
void cvInitFont( CvFont* font, CvFontFace font_face, float hscale, float
                vscale, float italic_scale, int thickness );
```

<i>font</i>	Pointer to the resultant font structure.
<i>font_face</i>	Font name identifier. Only the font CV_FONT_VECTOR0 is currently supported.
<i>hscale</i>	Horizontal scale. If equal to 1.0f, the characters will have the original width depending on the font type. If equal to 0.5f, the characters will be half of the original width.
<i>vscale</i>	Vertical scale. If equal to 1.0f, the characters will have the original height depending on the font type. If equal to 0.5f, the characters will be half of the original height.
<i>italic_scale</i>	Approximate tangent of the character slope relative to the vertical line. Zero value means a non-italic font, 1.0f means ~45° slope, etc.
<i>thickness</i>	Thickness of lines composing letters outlines. The function cvLine is used for drawing letters.

Discussion

The function [cvInitFont](#) initializes the font structure that can be passed further into text drawing functions. Although only one font is supported, it is possible to get different font “flavors” by varying the scale parameters, slope, and thickness.

cvPutText

Draws the text string.

```
void cvPutText( IplImage* img, const char* text, CvPoint org, CvFont* font, int
               color );
```

<i>img</i>	Input image.
<i>text</i>	String to print.
<i>org</i>	Coordinates of bottom-left corner of the first letter.
<i>font</i>	Pointer to the font structure.
<i>color</i>	Text color (RGB) or brightness (grayscale image).

Discussion

The function [cvPutText](#) renders the text in the image with the specified font and color. The printed text is clipped by ROI rectangle. Symbols that do not belong to the specified font are replaced with the “rectangle” symbol.

cvGetTextSize

Retrieves width and height of text string.

```
void cvGetTextSize( CvFont* font, const char* text_string, CvSize* text_size,
int* ymin );
```

<i>font</i>	Pointer to the font structure.
<i>text_string</i>	Input string.
<i>text_size</i>	Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.
<i>ymin</i>	Lowest <i>y</i> coordinate of the text relative to the baseline. Negative, if the text includes such characters as <i>g</i> , <i>j</i> , <i>p</i> , <i>q</i> , <i>y</i> , etc., and zero otherwise.

Discussion

The function [cvGetTextSize](#) calculates the binding rectangle for the given text string when the specified font is used.

System Functions

27

This chapter describes system library functions.

Reference

cvLoadPrimitives

Loads optimized versions of functions for specific platform.

```
int cvLoadPrimitives (char* dllName, char* processorType);
```

<i>dllName</i>	Name of dynamically linked library without postfix that contains the optimized versions of functions
<i>processorType</i>	Postfix that specifies the platform type: “w7” for Pentium® 4 processor, “A6” for Intel® Pentium® II processor, “M6” for Intel® Pentium® II processor, NULL for auto detection of the platform type.

Discussion

The function [cvLoadPrimitives](#) loads the versions of functions that are optimized for a specific platform. The function is automatically called before the first call to the library function, if not called earlier.

cvGetLibraryInfo

Gets the library information string.

```
void cvGetLibraryInfo (char** version, int* loaded, char** dllName);
```

<i>version</i>	Pointer to the string that will receive the build date information; can be NULL.
<i>loaded</i>	Postfix that specifies the platform type: “W7” for Pentium® 4 processor, “A6” for Intel® Pentium® III processor, “M6” for Intel® Pentium® II processor, NULL for auto detection of the platform type.
<i>dllName</i>	Pointer to the full name of dynamically linked library without path, could be NULL.

Discussion

The function [cvGetLibraryInfo](#) retrieves information about the library: the build date, the flag that indicates whether optimized *DLLS* have been loaded or not, and their names, if loaded.

The chapter describes unclassified OpenCV functions.

Reference

cvAbsDiff

Calculates absolute difference between two images and between image and scalar value.

```
void cvAbsDiff( IplImage* srcA, IplImage* srcB, IplImage* dst );
```

<i>srcA</i>	First compared image.
<i>srcB</i>	Second compared image.
<i>dst</i>	Destination image.
<i>value</i>	Value to compare.

Discussion

The function [cvAbsDiff](#) calculates the absolute difference between two images or between an image and a scalar value.

cvAbsDiff: $dst[i] = abs(src[i] - dst[i])$.

cvAbsDiffS

Calculates absolute difference between two images and between image and scalar value.

```
void cvAbsDiffS( IplImage* srcA, IplImage* dst, double value );
```

<i>srcA</i>	First compared image.
<i>srcB</i>	Second compared image.
<i>dst</i>	Destination image.
<i>value</i>	Value to compare.

Discussion

The function [cvAbsDiffS](#) calculates the absolute difference between two images or between an image and a scalar value.

cvAbsDiffS: $dst[i] = abs(src[i] - value)$.

cvMatchTemplate

Fills characteristic image for given image and template.

```
void cvMatchTemplate( IplImage* img, IplImage* templ, IplImage* result,
    CvTemplMatchMethod method );
```

<i>img</i>	Image where the search is running.
<i>templ</i>	Searched template; must be not greater than the source image. The parameters <i>img</i> and <i>templ</i> must be single-channel images and have the same depth (IPL_DEPTH_8U, IPL_DEPTH_8S, or IPL_DEPTH_32F).

<i>result</i>	Output characteristic image. It has to be a single-channel image with depth equal to <code>IPL_DEPTH_32F</code> . If the parameter <i>img</i> has the size of $W \times H$ and the template has the size of $w \times h$, the resulting image must have the size or selected ROI $W - w + 1 \times H - h + 1$.
<i>method</i>	Specifies the way the template must be compared with image regions.

Discussion

The function [cvMatchTemplate](#) implements a set of methods for finding regions in the image that are similar to the given template.

Given a source image with $W \times H$ pixels and template with $w \times h$ pixels, we get the resulting image with $W - w + 1 \times H - h + 1$ pixels, and the pixel value in each location (x, y) characterizes the similarity between the template and the image rectangle with the top-left corner at (x, y) and the right-bottom corner at $(x + w - 1, y + h - 1)$. Similarity can be calculated in several ways:

Squared difference (`method == CV_TM_SQDIFF`)

$$S(x, y) = \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} [T(x', y') - I(x + x', y + y')]^2,$$

where $I(x, y)$ is the value of the image pixel in the location (x, y) , while $T(x, y)$ is the value of the template pixel in the location (x, y) .

Normalized squared difference (`method == CV_TM_SQDIFF_NORMED`)

$$S(x, y) = \frac{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} [T(x', y') - I(x + x', y + y')]^2}{\sqrt{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y')^2 \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} I(x + x', y + y')^2}}.$$

Cross correlation (`method == CV_TM_CCORR`):

$$C(x, y) = \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y') I(x+x', y+y') .$$

Cross correlation, normalized (method == CV_TM_CCORR_NORMED):

$$\tilde{C}(x, y) = \frac{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y') I(x+x', y+y')}{\sqrt{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y')^2 \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} I(x+x', y+y')^2}} .$$

Correlation coefficient (method == CV_TM_CCOEFF):

$$R(x, y) = \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{T}(x', y') \tilde{I}(x+x', y+y') ,$$

where $\tilde{T}(x', y') = T(x', y') - \bar{T}$, $\tilde{I}(x+x', y+y') = I(x+x', y+y') - \bar{I}(x, y)$, and where \bar{T} stands for the average value of pixels in the template raster and $\bar{I}(x, y)$ stands for the average value of the pixels in the current “window” of the image.

Correlation coefficient, normalized (method == CV_TM_CCOEFF_NORMED):

$$\tilde{R}(x, y) = \frac{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{T}(x', y') \tilde{I}(x+x', y+y')}{\sqrt{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{T}(x', y')^2 \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{I}(x+x', y+y')^2}} .$$

After the function [cvMatchTemplate](#) returns the resultant image, probable positions of the template in the image could be located as the local or global maximums of the resultant image brightness.

cvCvtPixToPlane

Divides pixel image into separate planes.

```
void cvCvtPixToPlane( IplImage* src, IplImage* dst0, IplImage* dst1, IplImage*  
dst2, IplImage* dst3);
```

src Source image.
dst0...dst4 Destination planes.

Discussion

The function [cvCvtPixToPlane](#) divides a color image into separate planes. Two modes are available for the operation. Under the first mode the parameters *dst0*, *dst1*, and *dst2* are non-zero, while *dst3* must be zero for the three-channel source image. For the four-channel source image all the destination image pointers are non-zero, in this case the function splits the three/four channel image into separate planes and writes them to destination images. Under the second mode only one of the destination images is not `NULL`; in this case, the corresponding plane is extracted from the image and placed into destination image.

cvCvtPlaneToPix

Composes color image from separate planes.

```
void cvCvtPlaneToPix( IplImage* src0, IplImage* src1, IplImage* src2,  
IplImage* src3, IplImage* dst );
```

src0...src4 Source planes.
dst Destination image.

Discussion

The function [cvCvtPlaneToPix](#) composes color image from separate planes. If the *dst* has three channels, then *src0*, *src1*, and *src2* must be non-zero, otherwise *dst* must have four channels and all the source images must be non-zero.

cvConvertScale

Converts one image to another with linear transformation.

```
void cvConvertScale( IplImage* src, IplImage* dst, double scale, double
                    shift);
```

src Source image.

dst Destination image.

Discussion

The function [cvConvertScale](#) applies linear transform to all pixels in the source image and puts the result into the destination image with appropriate type conversion. The following conversions are supported: IPL_DEPTH_8U <-> IPL_DEPTH_32F, IPL_DEPTH_8U <-> IPL_DEPTH_16S, IPL_DEPTH_8S <-> IPL_DEPTH_32F, IPL_DEPTH_8S <-> IPL_DEPTH_16S, IPL_DEPTH_16S <-> IPL_DEPTH_32F and IPL_DEPTH_32S <-> IPL_DEPTH_32F. The unsigned char to float conversion is effected by the formula

$$dst(x,y) = (float)(src(x,y)*scale + shift);$$

The float is converted to unsigned char by the following algorithm:

```
t = round(src(x,y)*scale + shift);
if( t < 0 )
    dst(x,y) = 0;
else if( t > 255 )
    dst(x,y) = 255;
else
```

```
dst(x,y) = (unsigned char)t;
```

cvInitLineIterator

Initializes line iterator.

```
int cvInitLineIterator( IplImage* img, CvPoint pt1, CvPoint pt2,
    CvLineIterator* lineIterator);
```

<i>img</i>	Image.
<i>pt1</i>	Starting line point.
<i>pt2</i>	Ending line point.
<i>lineIterator</i>	Pointer to the line iterator state structure.

Discussion

The function [cvInitLineIterator](#) initializes the line iterator and returns the number of pixels between two ending 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 the 8-point connected Bresenham algorithm. Below follows an example how to draw the line on the RGB image, such that the image pixels that belong to the line are mixed with the given color using the XOR operation.

```
void put_xor_line( IplImage* img, CvPoint pt1, CvPoint pt2, int r, int
g, int b ) {
    CvLineIterator iterator;
    int count = cvInitLineIterator( img, pt1, pt2, &iterator);
    for( int i = 0; i < count; i++ ){
        iterator.ptr[0] ^= (uchar)b;
        iterator.ptr[1] ^= (uchar)g;
        iterator.ptr[2] ^= (uchar)r;
        CV_NEXT_LINE_POINT(iterator);
    }
```

```
}  
}
```

cvSampleLine

Reads raster line to buffer.

```
int cvSampleLine( IplImage* img, CvPoint pt1, CvPoint pt2, void* buffer );
```

<i>img</i>	Image.
<i>pt1</i>	Starting line point.
<i>pt2</i>	Ending line point.
<i>buffer</i>	Buffer to store the line points; must have enough size to store $\text{MAX}(pt2.x - pt1.x + 1, pt2.y - pt1.y + 1)$ points.

Discussion

The function [cvSampleLine](#) implements one particular case of application of line iterators. The function reads all the image points, lying on the line between *pt1* and *pt2*, including the ending points, and stores them into the buffer.

cvGetRectSubPix

Retrieves raster rectangle from image with sub-pixel accuracy.

```
void cvGetRectSubPix( IplImage* src, IplImage* rect, CvPoint2D32f center );
```

<i>src</i>	Source image.
<i>rect</i>	Extracted rectangle; must have odd width and height.
<i>center</i>	Floating point coordinates of the rectangle center. The center must be inside the image.

buffer Buffer to store the line points; must have enough size to store $\text{MAX}(|pt2.x - pt1.x| + 1, |pt2.y - pt1.y| + 1)$ points.

Discussion

The function [cvGetRectSubPix](#) extracts pixels from *src*, if the pixel coordinates satisfy the conditions below:

```
center.x - (widthrect-1)/2 <= x <= center.x + (widthrect-1)/2;
center.y - (heightrect-1)/2 <= y <= center.y + (heightrect-1)/2.
```

Since the center coordinates are not integer, bilinear interpolation is applied to get the values of pixels in non-integer locations. Although the rectangle center must be inside the image, the whole rectangle may be partially occluded. In this case, the pixel values are spread from the boundaries outside the image to approximate values of occluded pixels.

cvbFastArctan

Calculates fast arctangent approximation for arrays of abscissas and ordinates.

```
void cvbFastArctan( const float* y, const float* x, float* angle, int len );
```

y Array of ordinates.
x Array of abscissas.
angle Calculated angles of points (*x*[*i*], *y*[*i*]).
len Number of elements in the arrays.

Discussion

The function [cvbFastArctan](#) calculates an approximate arctangent value, the angle of the point (*x*, *y*). The angle is in the range from 0° to 360°. Accuracy is about 0.1°. For point (0, 0) the resultant angle is 0.

cvSqrt

Calculates square root of single float argument or array of floats.

```
float cvSqrt( float x );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

The function [cvSqrt](#) calculates the square root of arguments. The arguments should be non-negative, otherwise the result is unpredictable. The relative error for the scalar version is less than $9e-6$, for the vector function less than $3e-7$.

cvbSqrt

Calculates square root of single float argument or array of floats.

```
void cvbSqrt( const float* x, float* y, int len );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

The function [cvbSqrt](#) calculates the square root of arguments. The arguments should be non-negative, otherwise the result is unpredictable. The relative error for the scalar version is less than $9e-6$, for the vector function less than $3e-7$.

cvInvSqrt

Calculates inverse square root of single float argument or array of floats.

```
float cvInvSqrt( float x );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

The function [cvInvSqrt](#) calculates the inverse square root of arguments. The arguments should be non-negative, otherwise the result is unpredictable. The relative error for the scalar version is less than $9e-6$, for the vector function less than $3e-7$.

cvbInvSqrt

Calculates inverse square root of single float argument or array of floats.

```
void cvbInvSqrt( const float* x, float* y, int len );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

The function [cvbInvSqrt](#) calculates the inverse square root of their arguments. The arguments should be non-negative, otherwise the result is unpredictable. The relative error for the scalar version is less than $9e-6$, for the vector function less than $3e-7$.

cvbReciprocal

Calculates inverse of array of floats.

```
void cvbReciprocal( const float* x, float* y, int len );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

The function [cvbReciprocal](#) calculates the inverse ($1/x$) of arguments. The arguments should be non-zero. The function gives a very precise result with the relative error less than $1e-7$.

cvbCartToPolar

Calculates magnitude and angle for array of abscissas and ordinates.

```
void cvbCartToPolar( const float* y, const float* x, float* mag, float* angle,  
int len );
```

<i>y</i>	Array of ordinates.
<i>x</i>	Array of abscissas.
<i>mag</i>	Calculated magnitudes of points ($x[i], y[i]$).
<i>angle</i>	Calculated angles of points ($x[i], y[i]$).
<i>len</i>	Number of elements in the arrays.

Discussion

The function [cvbCartToPolar](#) calculates the magnitude $\sqrt{x[i]^2 + y[i]^2}$ and the angle $\arctan(y[i]/x[i])$ of each point $(x[i], y[i])$. The angle is measured in degrees and varies from 0° to 360° . The function is a combination of the functions [cvbFastArctan](#) and [cvbSqrt](#), so the accuracy is the same as in these functions. If pointers to the angle array or the magnitude array are `NULL`, the corresponding part is not calculated.

cvbFastExp

Calculates fast exponent approximation for array of floats.

```
void cvbFastExp( const float* x, double* exp_x, int len);
```

<code>x</code>	Array of arguments.
<code>exp_x</code>	Array of results.
<code>len</code>	Number of elements in the arrays.

Discussion

The function [cvbFastExp](#) calculates fast exponent approximation for each element of the input array. Maximal relative error is about $7e-6$.

cvbFastLog

Calculates fast approximation of natural logarithm for array of doubles.

```
void cvbFastLog( const double* x, float* log_x, int len);
```

<code>x</code>	Array of arguments.
<code>exp_x</code>	Array of results.

len Number of elements in the arrays.

Discussion

The function [cvbFastLog](#) calculates fast logarithm approximation for each element of the input array. Maximal relative error is about $7e-6$.

cvRandInit

Initializes state of random number generator.

```
void cvRandInit(   CvRandState* state, float lower, float upper, int seed );
```

state Pointer to the initialized random number generator state.
lower Lower boundary of uniform distribution.
upper Upper boundary of uniform distribution.
seed Initial 32-bit value to start a random sequence.

Discussion

The function [cvRandInit](#) initializes the `state` structure that is used for generating uniformly distributed numbers in the range `[lower, upper)`. A multiply-with-carry generator is used.

cvbRand

Fills array with random numbers

```
void cvbRand( CvRandState* state, float* x, int len );
```

state Random number generator state.
x Destination array.
len Number of elements in the array.

Discussion

The function [cvbRand](#) fills the array with random numbers and updates generator state.

cvFillImage

Fills image with constant value.

```
void cvFillImage( IplImage* img, double val );
```

<i>img</i>	Filled image.
<i>val</i>	Value to fill the image.

Discussion

The function [cvFillImage](#) is equivalent to either `iplSetFP` or `iplSet`, depending on the pixel type, that is, floating-point or integer.

cvRandSetRange

Sets range of generated random numbers without reinitializing RNG state.

```
void cvRandSetRange( CvRandState* state, double lower, double upper );
```

<i>state</i>	State of random number generator (RNG).
<i>lower</i>	New lower bound of generated numbers.
<i>upper</i>	New upper bound of generated numbers.

Discussion

The function [cvRandSetRange](#) changes the range of generated random numbers without reinitializing RNG state. For the current implementation of RNG the function is equivalent to the following code:

```
unsigned seed = state.seed;
unsigned carry = state.carry;
cvRandInit( &state, lower, upper, 0 );
state.seed = seed;
state.carry = carry;
```

However, the function is preferable because of compatibility with the next versions of the library.

cvKMeans

Splits set of vectors into given number of clusters.

```
void cvKMeans ( int num_clusters, CvVect32f* samples, int num_samples, int
vec_size, CvTermCriteria termcrit, int* cluster );
```

<i>num_clusters</i>	Number of required clusters.
<i>samples</i>	Pointer to array of input vectors.
<i>num_samples</i>	Number of input vectors.
<i>vec_size</i>	Size of every input vector.
<i>termcrit</i>	Criteria of iterative algorithm termination.
<i>cluster</i>	Characteristic array of cluster numbers, corresponding to each input vector.

Discussion

The function [cvKMeans](#) iteratively adjusts mean vectors of every cluster. Termination criteria must be used to stop the execution of the algorithm. At every iteration the convergence value is computed as follows:

$$\sum_{i=1}^K \|old_mean_i - new_mean_i\|^2.$$

The function terminates if $E < Termcrit.epsilon$.

This bibliography provides a list of publications that might be useful to the Intel® Computer Vision Library users. This list is not complete; it serves only as a starting point.

- [[Borgefors86](#)] Gunilla Borgefors. *Distance Transformations in Digital Images*. Computer Vision, Graphics and Image Processing 34, 344-371 (1986).
- [[Bradski00](#)] G. Bradski and J. Davis. *Motion Segmentation and Pose Recognition with Motion History Gradients*. IEEE WACV'00, 2000.
- [[Burt81](#)] P. J. Burt, T. H. Hong, A. Rosenfeld. *Segmentation and Estimation of Image Region Properties Through Cooperative Hierarchical Computation*. IEEE Tran. On SMC, Vol. 11, N.12, 1981, pp. 802-809.
- [[Canny86](#)] J. Canny. *A Computational Approach to Edge Detection*, IEEE Trans. on Pattern Analysis and Machine Intelligence, 8(6), pp. 679-698 (1986).
- [[Davis97](#)] J. Davis and Bobick. *The Representation and Recognition of Action Using Temporal Templates*. MIT Media Lab Technical Report 402, 1997.
- [[DeMenthon92](#)] Daniel F. DeMenthon and Larry S. Davis. *Model-Based Object Pose in 25 Lines of Code*. In Proceedings of ECCV '92, pp. 335-343, 1992.
- [[Fitzgibbon95](#)] Andrew W. Fitzgibbon, R.B.Fisher. *A Buyer's Guide to Conic Fitting*, Proc.5th British Machine Vision Conference, Birmingham, pp. 513-522, 1995.
- [[Hu62](#)] M. Hu. *Visual Pattern Recognition by Moment Invariants*, IRE Transactions on Information Theory, 8:2, pp. 179-187, 1962.

- [[Jahne97](#)] B. Jahne. *Digital Image Processing*. Springer, New York, 1997.
- [[Kass88](#)] M. Kass, A. Witkin, and D. Terzopoulos. *Snakes: Active Contour Models*, International Journal of Computer Vision, pp. 321-331, 1988.
- [[Matas98](#)] J. Matas, C. Galambos, J. Kittler. *Progressive Probabilistic Hough Transform*. British Machine Vision Conference, 1998.
- [[Rosenfeld73](#)] A. Rosenfeld and E. Johnston. *Angle Detection on Digital Curves*. IEEE Trans. Computers, 22:875-878, 1973.
- [[RubnerJan98](#)] Y. Rubner. C. Tomasi, L.J. Guibas. *Metrics for Distributions with Applications to Image Databases*. Proceedings of the 1998 IEEE International Conference on Computer Vision, Bombay, India, January 1998, pp. 59-66.
- [[RubnerSept98](#)] Y. Rubner. C. Tomasi, L.J. Guibas. *The Earth Mover's Distance as a Metric for Image Retrieval*. Technical Report STAN-CS-TN-98-86, Department of Computer Science, Stanford University, September 1998.
- [[RubnerOct98](#)] Y. Rubner. C. Tomasi. *Texture Metrics*. Proceeding of the IEEE International Conference on Systems, Man, and Cybernetics, San-Diego, CA, October 1998, pp. 4601-4607.
<http://robotics.stanford.edu/~rubner/publications.html>
- [[Serra82](#)] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, 1982.
- [[Schiele2000](#)] Bernt Schiele and James L. Crowley. *Recognition without Correspondence Using Multidimensional Receptive Field Histograms*. In International Journal of Computer Vision 36 (1), pp. 31-50, January 2000.
- [[Suzuki85](#)] S. Suzuki, K. Abe. *Topological Structural Analysis of Digital Binary Images by Border Following*. CVGIP, v.30, n.1. 1985, pp. 32-46.
- [[Teh89](#)] C.H. Teh, R.T. Chin. *On the Detection of Dominant Points on Digital Curves*. - IEEE Tr. PAMI, 1989, v.11, No.8, p. 859-872.
- [[Trucco98](#)] Emanuele Trucco, Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, Inc., 1998.

- [[Williams92](#)] D. J. Williams and M. Shah. *A Fast Algorithm for Active Contours and Curvature Estimation*. CVGIP: Image Understanding, Vol. 55, No. 1, pp. 14-26, Jan., 1992._
<http://www.cs.ucf.edu/~vision/papers/shah/92/WIS92A.pdf>.
- [[Yuille89](#)] A.Y.Yuille, D.S.Cohen, and P.W.Hallinan. *Feature Extraction from Faces Using Deformable Templates in CVPR*, pp. 104-109, 1989.
- [[Zhang96](#)] Zhengyou Zhang. *Parameter Estimation Techniques: A Tutorial with Application to Conic Fitting*, Image and Vision Computing Journal, 1996.

Index

A

Active Contours
 cvSnakeImage, 17-3

B

Background Subtraction, 9-1
 cvAcc, 9-2
 cvMultiplyAcc, 9-3
 cvRunningAvg, 9-4
 cvSquareAcc, 9-3

C

Camera Calibration
 cvCalibrateCamera, 13-5
 cvCalibrateCamera_64d, 13-6
 cvFindChessBoardCornerGuesses, 13-12
 cvFindExtrinsicCameraParams, 13-7
 cvFindExtrinsicCameraParams_64d, 13-8
 cvRodrigues, 13-9
 cvRodrigues_64d, 13-9
 cvUnDistort, 13-12
 cvUnDistortInit, 13-11
 cvUnDistortOnce, 13-10
CamShift
 cvCamShift, 16-7
 cvMeanShift, 16-8
Contour Processing
 cvApproxChains, 3-24
 cvApproxPoly, 3-26
 cvContourArea, 3-28
 cvContourFromContourTree, 3-30

 cvContoursMoments, 3-27
 cvCreateContourTree, 3-30
 cvDrawContours, 3-26
 cvEndFindContours, 3-23
 cvFindContours, 3-20
 cvFindNextContour, 3-22
 cvMatchContours, 3-28
 cvMatchContourTrees, 3-31
 cvReadChainPoint, 3-25
 cvStartFindContours, 3-21
 cvStartReadChainPoints, 3-25
 cvSubstituteContour, 3-23
cvCalcPGH, 4-14
cvCheckContourConvexity, 4-12
cvCreateStructuringElementEx, 8-6
cvEndFindContours, 3-23
cvFitLine2D, 4-5
cvLaplace, 5-7
cvReleaseStructuringElement, 8-7
cvSeqElemIdx, 2-17
cvSeqPopFront, 2-13

D

Distance Transform
 cvDistTransform, 10-1
Drawing Primitives
 cvCircle, 26-4
 cvEllipse, 26-5
 cvEllipseAA, 26-6
 cvFillConvexPoly, 26-8
 cvFillPoly, 26-7

- cvGetTextSize, 26-11
 - cvInitFont, 26-10
 - cvLine, 26-2
 - cvLineAA, 26-3
 - cvPolyLine, 26-8
 - cvPolyLineAA, 26-9
 - cvPutText, 26-10
 - cvRectangle, 26-4
 - Dynamic Data Structures
 - Graphs
 - cvClearGraph, 2-42
 - cvCreateGraph, 2-35
 - cvFindGraphEdge, 2-40
 - cvFindGraphEdgeByPtr, 2-40
 - cvGetGraphVtx, 2-43
 - cvGraphAddEdge, 2-37
 - cvGraphAddEdgeByPtr, 2-38
 - cvGraphAddVtx, 2-36
 - cvGraphEdgeIdx, 2-44
 - cvGraphRemoveEdge, 2-39
 - cvGraphRemoveEdgeByPtr, 2-39
 - cvGraphRemoveVtx, 2-36
 - cvGraphRemoveVtxByPtr, 2-37
 - cvGraphVtxDegree, 2-41
 - cvGraphVtxDegreeByPtr, 2-42
 - cvGraphVtxIdx, 2-43
 - Memory Functions
 - cvClearMemStorage, 2-4
 - cvCreateChildMemStorage, 2-3
 - cvCreateMemStorage, 2-3
 - cvReleaseMemStorage, 2-4
 - cvRestoreMemStoragePos, 2-5
 - Sequences
 - cvClearSeq, 2-16
 - cvCreateSeq, 2-10
 - cvCvtSeqToArray, 2-18
 - cvGetSeqElem, 2-17
 - cvMakeSeqHeaderForArray, 2-18
 - cvSeqElemIdx, 2-17
 - cvSeqInsert, 2-15
 - cvSeqPop, 2-13
 - cvSeqPopFront, 2-13
 - cvSeqPopMulti, 2-14
 - cvSeqPush, 2-12
 - cvSeqPushFront, 2-13
 - cvSeqPushMulti, 2-14
 - cvSeqRemove, 2-16
 - cvSetSeqBlockSize, 2-11
 - Sets
 - cvClearSet, 2-31
 - cvCreateSet, 2-29
 - cvGetSetElem, 2-30
 - cvSetAdd, 2-29
 - cvSetRemove, 2-30
 - Writing and Reading Sequences
 - cvEndWriteSeq, 2-22
 - cvFlushSeqWriter, 2-23
 - cvGetSeqReaderPos, 2-24
 - cvSetSeqReaderPos, 2-25
 - cvStartAppendToSeq, 2-21
 - cvStartReadSeq, 2-23
 - cvStartWriteSeq, 2-21
- ## E
- Eigen Objects
 - cvCalcCovarMatrixEx, 24-2
 - cvCalcDecompCoeff, 24-4
 - cvCalcEigenObjects, 24-3
 - cvEigenDecomposite, 24-5
 - cvEigenProjection, 24-6
 - Estimators
 - cvConDensInitSampleSet, 19-8
 - cvConDensUpdatebyTime, 19-9
 - cvCreateConDensation, 19-7
 - cvCreateKalman, 19-4
 - cvKalmanUpdateByMeasurement, 19-6
 - cvKalmanUpdateByTime, 19-5
 - cvReleaseConDensation, 19-8
 - cvReleaseKalman, 19-5
- ## F
- Features
 - Feature Detection Functions
 - cvCanny, 5-11
 - cvCornerEigenValsandVecs, 5-12

- cvCornerMinEigenVal, 5-13
- cvFindCornerSubPix, 5-14
- cvGoodFeaturesToTrack, 5-16
- cvPreCornerDetect, 5-12
- Hough Transform
 - cvHoughLines, 5-18
 - cvHoughLinesP, 5-19
 - cvHoughLinesSDiv, 5-19
- Optimal Filter Kernels
 - cvLaplace, 5-7
 - cvSobel, 5-7
- Flood Fill
 - cvFloodFill, 12-2

G

- Geometry
 - cvCalcPGH, 4-14
 - cvCheckContourConvexity, 4-12
 - cvContourConvexHull, 4-9
 - cvContourConvexHullApprox, 4-11
 - cvConvexHull, 4-9
 - cvConvexHullApprox, 4-10
 - cvConvexityDefects, 4-12
 - cvFitEllipse_32f, 4-4
 - cvFitLine, 4-5
 - cvMinAreaRect, 4-13
 - cvMinEnclosingCircle, 4-15
 - cvProject3D, 4-8

H

- Histogram
 - cvCalcBackProject, 21-16
 - cvCalcBackProjectPatch, 21-17
 - cvCalcEMD, 21-20
 - cvCalcHist, 21-16
 - cvCompareHist, 21-14
 - cvCopyHist, 21-15
 - cvCreateHist, 21-6
 - cvGetHistValue_1D, 21-10
 - cvGetHistValue_2D, 21-11
 - cvGetHistValue_3D, 21-11

- cvGetHistValue_nD, 21-12
- cvGetMinMaxHistValue, 21-12
- cvMakeHistHeaderForArray, 21-8
- cvNormalizeHist, 21-13
- cvQueryHistValue_1D, 21-8
- cvQueryHistValue_2D, 21-9
- cvQueryHistValue_3D, 21-9
- cvQueryHistValue_nD, 21-10
- cvReleaseHist, 21-7
- cvSetHistThresh, 21-15
- cvThreshHist, 21-13

I

Image Function Reference

- cvCopyImage, 1-10
- cvCreateImage, 1-5
- cvCreateImageData, 1-6
- cvCreateImageHeader, 1-4
- cvGetImageRawData, 1-9
- cvInitImageHeader, 1-9
- cvReleaseImage, 1-6
- cvReleaseImageData, 1-7
- cvReleaseImageHeader, 1-5
- cvSetImageCOI, 1-8
- cvSetImageData, 1-7
- cvSetImageROI, 1-8

Image Statistics

- cvCountNonZero, 6-2
- cvGetCentralMoment, 6-7
- cvGetHuMoments, 6-9
- cvGetNormalizedCentralMoment, 6-8
- cvGetSpatialMoment, 6-7
- cvMean, 6-3
- cvMean_StdDev, 6-3
- cvMinMaxLoc, 6-4
- cvMinMaxLocMask, 6-4
- cvMoments, 6-6
- cvNorm, 6-4
- cvSumPixels, 6-2

M

Morphology, 8-1

- cvCreateStructuringElementEx, 8-6
- cvDilate, 8-8
- cvErode, 8-7
- cvMorphologyEx, 8-9
- cvReleaseStructuringElement, 8-7

Motion Templates

- cvCalcGlobalOrientation, 15-9
- cvCalcMotionGradient, 15-8
- cvSegmentMotion, 15-10

O

Optical Flow

- cvCalcOpticalFlowBM, 18-5
- cvCalcOpticalFlowHS, 18-4
- cvCalcOpticalFlowLK, 18-4
- cvCalcOpticalFlowPyrLK, 18-6

P

Pixel Access Macro Reference, 1-10

- CV_INIT_PIXEL_POS, 1-12
- CV_MOVE, 1-13
- CV_MOVE_PARAM, 1-14
- CV_MOVE_PARAM_WRAP, 1-15
- CV_MOVE_TO, 1-13
- CV_MOVE_WRAP, 1-14

POSIT

- cvCreatePOSITObject, 20-7
- cvPOSIT, 20-7
- cvReleasePOSITObject, 20-8

Pyramids

- cvPyrDown, 7-6
- cvPyrSegmentation, 7-7
- cvPyrUp, 7-6

S

System Functions

- cvGetLibraryInfo, 27-2

- cvLoadPrimitives, 27-1

T

Threshold Functions

- cvAdaptiveThreshold, 11-2
- cvThreshold, 11-3

U

Utility

- cvAbsDiff, 28-1
- cvAbsDiffS, 28-2
- cvbCartToPolar, 28-12
- cvbFastArctan, 28-9
- cvbFastExp, 28-13
- cvbFastLog, 28-13
- cvbInvSqrt, 28-11
- cvbRand, 28-14
- cvbReciprocal, 28-12
- cvbSqrt, 28-10
- cvConvertScale, 28-6
- cvCvtPixToPlane, 28-5
- cvCvtPlaneToPix, 28-5
- cvFillImage, 28-15
- cvGetRectSubPix, 28-8
- cvInitLineIterator, 28-7
- cvInvSqrt, 28-11
- cvKMeans, 28-16
- cvMatchTemplate, 28-2
- cvRandInit, 28-14
- cvRandSetRange, 28-15
- cvSampleLine, 28-8
- cvSqrt, 28-10

V

View Morphing

- cvDeleteMoire, 14-12
- cvDynamicCorrespondMulti, 14-9
- cvFindFundamentalMatrix, 14-5
- cvFindRuns, 14-8
- cvMakeAlphaScanlines, 14-9

cvMakeScanlines, 14-6
cvMorphEpilinesMulti, 14-10
cvPostWarpImage, 14-11
cvPreWarpImage, 14-7