Intel® Open Source Computer Vision Library

What is OpenCV

OpenCV means Intel® Open Source Computer Vision Library. It is a collection of C functions and few C++ classes that implement some popular algorithms of Image Processing and Computer Vision.

The key features

OpenCV is cross-platform middle-to-high level API that consists of a few hundreds (>300) C functions. It does not rely on external numerical libraries, though it can make use of some of them (see below) at runtime, if they are available.

OpenCV is free for both non-commercial and commercial use (see the license for details).

OpenCV provides transparent for user interface to Intel® Integrated Performance Primitives (IPP) (only ippcv for now). That is, it loads automatically IPP libraries optimized for specific processor at runtime, if they are available. More information about IPP can be retrieved at http://www.intel.com/software/products/ipp/ippvm20/index.htm

There are interfaces to OpenCV for some other languages/environments (more to come):

- EiC - ANSI C interpreter written by Ed Breen. AFAIK, it is now abandoned. Hawk and CvEnv are the interactive environments (written in MFC and TCL,respectively) that embedd EiC interpteter.
- Ch - ANSI C/C++ interpreter with some scripting capabilities, created and supported by SoftIntegration® company (http://www.softintegration.com) Wrappers for Ch are located at opencv/interfaces/ch.
- MATLAB® - great environment for numerical and symbolic computing by Mathworks. MATLAB® interface for some of OpenCV functions can be found at opencv/interfaces/matlab/toolbox

---

Who created it

The complete list of authors can be found in the file AUTHORS.

Besides, a lot of people helped with suggestions, patches, bug reports etc. The incomplete list is in THANKS file.

What's New

The most important things arrived in beta 3 are:

- Complete up-to-date HTML reference [we hope]
- Source code for stereo correspondence [p 12]
- Robust and fast face detection [p 4]
- 3D Object Tracker [p 14]

See ChangeLog file for full list of changes.

Where to get OpenCV

Go http://www.sourceforge.net/projects/opencvlibrary. If it does not work, type "OpenCV" in Google.

If you have a problem with installing/running/using OpenCV

1. Read FAQs [p 17]
2. Search through OpenCV archives at www.yahoogroups.com (http://groups.yahoo.com/group/OpenCV/)
3. Join OpenCV mailing list at yahoo groups (see FAQs on how to do it) and mail your questions (the mailing list will probably migrate to OpenCV's SourceForge site)
4. Look at the OpenCV sample code, read the reference manual :)

OpenCV Reference Manual

You may also look at the PDF manual, but do not trust it much - it is pretty out of date, especially, the reference part.

Other resources

OpenCV Applications (Windows only)

If you have questions/corrections/suggestions about these pages (not about the library ifself), mail to Vadim.Pisarevsky@intel.com.

Experimental Functionality Reference

The functionality resides in cvaux library. To use it in your application, place #include "cvaux.h" in your source files and:

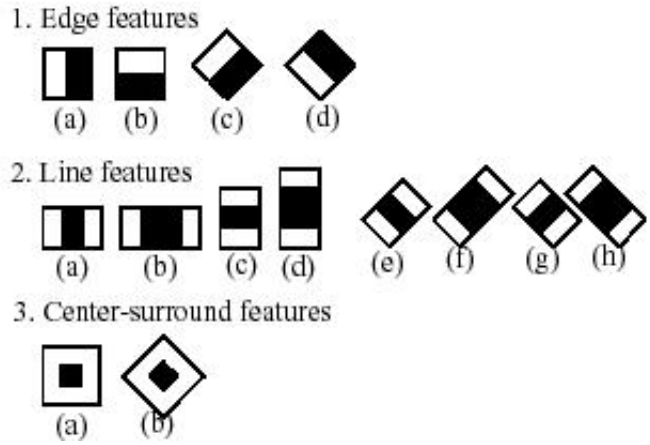
- In case of Win32 link the app against cvaux.lib that is import library for cvaux.dll
- In case of Linux use -lcvaux compiler option

Object Detection Functions

The object detector described below has been initially proposed by Paul Viola [Viola01] [p ??] and improved by Rainer Lienhart [Lienhart02] [p ??] . First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundreds of sample views of a particular object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.

After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a "1" if the region is likely to show the object (i.e., face/car), and "0" otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word "cascade" in the classifier name means that the resultant classifier consists of several simpler classifiers (`stages`) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word "boosted" means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different *boosting* techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifers, and are calculated as described below. The current algorithm uses the following Haar-like features:



The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and cvIntegral [p 167] description).

To see the object detector at work, have a look at HaarFaceDetect demo.

The following reference is for the detection part only. There is a separate application called *haartraining* that can train a cascade of boosted classifiers from a set of samples. See *opencv/apps/haartraining* for details.

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade

Boosted Haar classifier structures

```
#define CV_HAAR_FEATURE_MAX  3

/* a haar feature consists of 2-3 rectangles with appropriate weights */
typedef struct CvHaarFeature
```

```
{
    int  tilted; /* 0 means up-right feature, 1 means 45--rotated feature */

    /* 2-3 rectangles with weights of opposite signs and
       with absolute values inversely proportional to the areas of the rectangles.
       if rect[2].weight !=0, then
       the feature consists of 3 rectangles, otherwise it consists of 2 */
    struct
    {
        CvRect r;
        float weight;
    } rect[CV_HAAR_FEATURE_MAX];
} CvHaarFeature;

/* a single tree classifier (stump in the simplest case) that returns the response for the feature
   at the particular image location (i.e. pixel sum over subrectangles of the window) and gives out
   a value depending on the responce */
typedef struct CvHaarClassifier
{
    int count;
    /* number of nodes in the decision tree */
    CvHaarFeature* haarFeature;
    /* these are "parallel" arrays. Every index i
       corresponds to a node of the decision tree (root has 0-th index).

       left[i] - index of the left child (or negated index if the left child is a leaf)
       right[i] - index of the right child (or negated index if the right child is a leaf)
       threshold[i] - branch threshold. if feature responce is <= threshold, left branch
                      is chosen, otherwise right branch is chosed.
       alpha[i] - output value correponding to the leaf. */
    float* threshold; /* array of decision thresholds */
    int* left; /* array of left-branch indices */
    int* right; /* array of right-branch indices */
    float* alpha; /* array of output values */
}
CvHaarClassifier;

/* a boosted battery of classifiers(=stage classifier):
   the stage classifier returns 1
   if the sum of the classifiers' responces
   is greater than threshold and 0 otherwise */
typedef struct CvHaarStageClassifier
{
    int  count; /* number of classifiers in the battery */
    float threshold; /* threshold for the boosted classifier */
    CvHaarClassifier* classifier; /* array of classifiers */
}
CvHaarStageClassifier;

/* cascade of stage classifiers */
typedef struct CvHaarClassifierCascade
{
    int  count; /* number of stages */
    CvSize origWindowSize; /* original object size (the cascade is trained for) */
    CvHaarStageClassifier* stageClassifier; /* array of stage classifiers */
}
CvHaarClassifierCascade;
```

All the structures are used for representing a cascaded of boosted Haar classifiers. The cascade has the following hierarchical structure:

```
Cascade:
    Stage_1 :
        Classifier_11 :
            Feature_11
        Classifier_12 :
            Feature_12
        ...
    Stage_2 :
        Classifier_21 :
            Feature_21
        ...
    ...
```

The whole hierarchy can be constructed manually or loaded from a file or an embedded base using function cvLoadHaarClassifierCascade [p 7] .

---

cvLoadHaarClassifierCascade

Loads a trained cascade classifier from file or the classifier database embedded in OpenCV

```
CvHaarClassifierCascade*
cvLoadHaarClassifierCascade( const char* directory="<default_face_cascade>",
                             CvSize origWindowSize=cvSize(24,24));
```

directory
: Name of file containing the description of a trained cascade classifier; or name in angle brackets of a cascade in the classifier database embedded in OpenCV (only "<default_face_cascade>" is supported now).

origWindowSize
: Original size of objects the cascade has been trained on. Note that it is not stored in the cascade and therefore must be specified separately.

The function cvLoadHaarClassifierCascade [p 7] loads a trained cascade of haar classifiers from a file or the classifier database embedded in OpenCV. The base can be trained using *haartraining* application (see opencv/apps/haartraining for details).

---

cvReleaseHaarClassifierCascade

Releases haar classifier cascade

```
void cvReleaseHaarClassifierCascade( CvHaarClassifierCascade** cascade );
```

cascade
: Double pointer to the released cascade. The pointer is cleared by the function.

The function cvReleaseHaarClassifierCascade [p 7] deallocates the cascade that has been created manually or by cvLoadHaarClassifierCascade [p 7] .

---

cvCreateHidHaarClassifierCascade

Converts boosted classifier cascade to internal representation

```
/* hidden (optimized) representation of Haar classifier cascade */
typedef struct CvHidHaarClassifierCascade CvHidHaarClassifierCascade;

CvHidHaarClassifierCascade*
cvCreateHidHaarClassifierCascade( CvHaarClassifierCascade* cascade,
                                  const CvArr* sumImage=0,
                                  const CvArr* sqSumImage=0,
                                  const CvArr* tiltedSumImage=0,
                                  double scale=1 );
```

cascade
: original cascade that may be loaded from file using cvLoadHaarClassifierCascade [p 7] .

sumImage
: Integral (sum) single-channel image of 32-bit integer format. This image as well as the two subsequent images are used for fast feature evaluation and brightness/contrast normalization. They all can be retrieved from the input 8-bit single-channel image using function cvIntegral [p ??] . Note that all the images are 1 pixel wider and 1 pixel taller than the source 8-bit image.

sqSumImage
: Square sum single-channel image of 64-bit floating-point format.

tiltedSumImage
: Tilted sum single-channel image of 32-bit integer format.

scale
: Initial scale (see cvSetImagesForHaarClassifierCascade [p 11]).

The function cvCreateHidHaarClassifierCascade [p 8] converts pre-loaded cascade to internal faster representation. This step must be done before the actual processing. The integral image pointers may be NULL, in this case the images should be assigned later by cvSetImagesForHaarClassifierCascade [p 11] .

---

cvReleaseHidHaarClassifierCascade

Releases hidden classifier cascade structure

```
void cvReleaseHidHaarClassifierCascade( CvHidHaarClassifierCascade** cascade );
```

cascade
: Double pointer to the released cascade. The pointer is cleared by the function.

The function cvReleaseHidHaarClassifierCascade [p 8] deallocates structure that is an internal ("hidden") representation of haar classifier cascade.

cvHaarDetectObjects

Detects objects in the image

```
typedef struct CvAvgComp
{
    CvRect rect; /* bounding rectangle for the face (average rectangle of a group) */
    int neighbors; /* number of neighbor rectangles in the group */
}
CvAvgComp;

CvSeq* cvHaarDetectObjects( const IplImage* img, CvHidHaarClassifierCascade* cascade,
                            CvMemStorage* storage, double scale_factor=1.1,
                            int min_neighbors=3, int flags=0 );
```

img
: Image to detect objects in.

cascade
: Haar classifier cascade in internal representation.

storage
: Memory storage to store the resultant sequence of the object candidate rectangles.

scale_factor
: The factor by which the search window is scaled between the subsequent scans, for example, 1.1 means increasing window by 10%.

min_neighbors
: Minimum number (minus 1) of neighbor rectangles that makes up an object. All the groups of a smaller number of rectangles than `min_neighbors`-1 are rejected. If `min_neighbors` is 0, the function does not any grouping at all and returns all the detected candidate rectangles, which may be useful if the user wants to apply a customized grouping procedure.

flags
: Mode of operation. Currently the only flag that may be specified is `CV_HAAR_DO_CANNY_PRUNING`. If it is set, the function uses Canny edge detector to reject some image regions that contain too few or too much edges and thus can not contain the searched object. The particular threshold values are tuned for face detection and in this case the pruning speeds up the processing.

The function cvHaarDetectObjects [p 9] finds rectangular regions in the given image that are likely to contain objects the cascade has been trained for and returns those regions as a sequence of rectangles. The function scans the image several times at different scales (see cvSetImagesForHaarClassifierCascade [p 11]). Each time it considers overlapping regions in the image and applies the classifiers to the regions using cvRunHaarClassifierCascade [p 11] . It may also apply some heuristics to reduce number of analyzed regions, such as Canny prunning. After it has proceeded and collected the candidate rectangles (regions that passed the classifier cascade), it groups them and returns a sequence of average rectangles for each large enough group. The default parameters (`scale_factor`=1.1, `min_neighbors`=3, `flags`=0) are tuned for accurate yet slow face detection. For faster face detection on real video images the better settings are (`scale_factor`=1.2, `min_neighbors`=2, `flags`=CV_HAAR_DO_CANNY_PRUNING).

Example. Using cascade of Haar classifiers to find faces.

```
#include "cv.h"
#include "cvaux.h"
#include "highgui.h"

CvHidHaarClassifierCascade* new_face_detector(void)
{
    CvHaarClassifierCascade* cascade = cvLoadHaarClassifierCascade("<default_face_cascade>", cvSize(24,24));
    /* images are assigned inside cvHaarDetectObject, so pass NULL pointers here */
    CvHidHaarClassifierCascade* hid_cascade = cvCreateHidHaarClassifierCascade( cascade, 0, 0, 0, 1 );
    /* the original cascade is not needed anymore */
    cvReleaseHaarClassifierCascade( &cascade );
    return hid_cascade;
}

void detect_and_draw_faces( IplImage* image,
                            CvHidHaarClassifierCascade* cascade,
                            int do_pyramids )
{
    IplImage* small_image = image;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* faces;
    int i, scale = 1;

    /* if the flag is specified, down-scale the input image to get a
       performance boost w/o loosing quality (perhaps) */
    if( do_pyramids )
    {
        small_image = cvCreateImage( cvSize(image->width/2,image->height/2), IPL_DEPTH_8U, 3 );
        cvPyrDown( image, small_image, CV_GAUSSIAN_5x5 );
        scale = 2;
    }

    /* use the fastest variant */
    faces = cvHaarDetectObjects( small_image, cascade, storage, 1.2, 2, CV_HAAR_DO_CANNY_PRUNING );

    /* draw all the rectangles */
    for( i = 0; i < faces->total; i++ )
    {
        /* extract the rectanlges only */
        CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i, 0 );
        cvRectangle( image, cvPoint(face_rect.x*scale,face_rect.y*scale),
                     cvPoint((face_rect.x+face_rect.width)*scale,
                             (face_rect.y+face_rect.height)*scale),
                     CV_RGB(255,0,0), 3 );
    }

    if( small_image != image )
        cvReleaseImage( &small_image );
    cvReleaseMemStorage( &storage );
}

/* takes image filename from the command line */
int main( int argc, char** argv )
{
    IplImage* image;
    if( argc==2 && (image = cvLoadImage( argv[1], 1 )) != 0 )
    {
        CvHidHaarClassifierCascade* cascade = new_face_detector();
        detect_and_draw_faces( image, cascade, 1 );
        cvNamedWindow( "test", 0 );
        cvShowImage( "test", image );
        cvWaitKey(0);
        cvReleaseHidHaarClassifierCascade( &cascade );
        cvReleaseImage( &image );
    }

    return 0;
}
```

cvSetImagesForHaarClassifierCascade

Assigns images to the hidden cascade

```
void cvSetImagesForHaarClassifierCascade( CvHidHaarClassifierCascade* cascade,
                                          const CvArr* sumImage, const CvArr* sqSumImage,
                                          const CvArr* tiltedImage, double scale );
```

cascade
: Hidden Haar classifier cascade, created by cvCreateHidHaarClassifierCascade [p 8] .

sumImage
: Integral (sum) single-channel image of 32-bit integer format. This image as well as the two subsequent images are used for fast feature evaluation and brightness/contrast normalization. They all can be retrieved from input 8-bit single-channel image using function cvIntegral [p ??] . Note that all the images are 1 pixel wider and 1 pixel taller than the source 8-bit image.

sqSumImage
: Square sum single-channel image of 64-bit floating-point format.

tiltedSumImage
: Tilted sum single-channel image of 32-bit integer format.

scale
: Window scale for the cascade. If `scale`=1, original window size is used (objects of that size are searched) - the same size as specified in cvLoadHaarClassifierCascade [p 7] (24x24 in case of "<default_face_cascade>"), if `scale`=2, a two times larger window is used (48x48 in case of default face cascade). While this will speed-up search about four times, faces smaller than 48x48 cannot be detected.

The function cvSetImagesForHaarClassifierCascade [p 11] assigns images and/or window scale to the hidden classifier cascade. If image pointers are NULL, the previously set images are used further (i.e. NULLs mean "do not change images"). Scale parameter has no such a "protection" value, but the previous value can be retrieved by cvGetHaarClassifierCascadeScale [p 12] function and reused again. The function is used to prepare cascade for detecting object of the particular size in the particular image. The function is called internally by cvHaarDetectObjects [p 9] , but it can be called by user if there is a need in using lower-level function cvRunHaarClassifierCascade [p 11] .

---

cvRunHaarClassifierCascade

Runs cascade of boosted classifier at given image location

```
int cvRunHaarClassifierCascade( CvHidHaarClassifierCascade* cascade,
                                CvPoint pt, int startStage=0 );
```

cascade
: Hidden Haar classifier cascade.

pt
: Top-left corner of the analyzed region. Size of the region is a original window size scaled by the currenly set scale. The current window size may be retrieved using

cvGetHaarClassifierCascadeWindowSize [p 12] function.

startStage
: Initial zero-based index of the cascade stage to start from. The function assumes that all the previous stages are passed. This feature is used internally by cvHaarDetectObjects [p 9] for better processor cache utilization.

The function cvRunHaarHaarClassifierCascade [p ??] runs Haar classifier cascade at a single image location. Before using this function the integral images and the appropriate scale (=> window size) should be set using cvSetImagesForHaarClassifierCascade [p 11] . The function returns positive value if the analyzed rectangle passed all the classifier stages (it is a candidate) and zero or negative value otherwise.

---

cvGetHaarClassifierCascadeScale

Retrieves the current scale of cascade of classifiers

```
double cvGetHaarClassifierCascadeScale( CvHidHaarClassifierCascadeScale* cascade );
```

cascade
: Hidden Haar classifier cascade.

The function cvGetHaarHaarClassifierCascadeScale [p ??] retrieves the current scale factor for the search window of the Haar classifier cascade. The scale can be changed by cvSetImagesForHaarClassifierCascade [p 11] by passing NULL image pointers and the new scale value.

---

cvGetHaarClassifierCascadeWindowSize

Retrieves the current search window size of cascade of classifiers

```
CvSize cvGetHaarClassifierCascadeWindowSize( CvHidHaarClassifierCascadeWindowSize* cascade );
```

cascade
: Hidden Haar classifier cascade.

The function cvGetHaarHaarClassifierCascadeWindowSize [p ??] retrieves the current search window size for the Haar classifier cascade. The window size can be changed implicitly by setting appropriate scale.

Stereo Correspondence Functions

FindStereoCorrespondence

Calculates disparity for stereo-pair

```
cvFindStereoCorrespondence(
                   const  CvArr* leftImage, const  CvArr* rightImage,
                   int     mode, CvArr*  depthImage,
                   int     maxDisparity,
                   double  param1, double  param2, double  param3,
                   double  param4, double  param5  );
```

leftImage
: Left image of stereo pair, rectified grayscale 8-bit image

rightImage
: Right image of stereo pair, rectified grayscale 8-bit image

mode
: Algorithm used to find a disparity (now only CV_DISPARITY_BIRCHFIELD is supported)

depthImage
: Destination depth image, grayscale 8-bit image that codes the scaled disparity, so that the zero disparity (corresponding to the points that are very far from the cameras) maps to 0, maximum disparity maps to 255.

maxDisparity
: Maximum possible disparity. The closer the objects to the cameras, the larger value should be specified here. Too big values slow down the process significantly.

param1, param2, param3, param4, param5
: - parameters of algorithm. For example, param1 is the constant occlusion penalty, param2 is the constant match reward, param3 defines a highly reliable region (set of contiguous pixels whose reliability is at least param3), param4 defines a moderately reliable region, param5 defines a slightly reliable region. If some parameter is omitted default value is used. In Birchfield's algorithm param1 = 25, param2 = 5, param3 = 12, param4 = 15, param5 = 25 (These values have been taken from "Depth Discontinuities by Pixel-to-Pixel Stereo" Stanford University Technical Report STAN-CS-TR-96-1573, July 1996.)

The function cvFindStereoCorrespondence [p 13] calculates disparity map for two rectified grayscale images.

Example. Calculating disparity for pair of 8-bit color images

```
/*-----------------------------------------------------------------------------*/
IplImage* srcLeft = cvLoadImage("left.jpg",1);
IplImage* srcRight = cvLoadImage("right.jpg",1);
IplImage* leftImage = cvCreateImage(cvGetSize(srcLeft), IPL_DEPTH_8U, 1);
IplImage* rightImage = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);
IplImage* depthImage = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);

cvCvtColor(srcLeft, leftImage, CV_BGR2GRAY);
cvCvtColor(srcRight, rightImage, CV_BGR2GRAY);

cvFindStereoCorrespondence( leftImage, rightImage, CV_DISPARITY_BIRCHFIELD, depthImage, 50, 15, 3, 6, 8, 15 );
/*-----------------------------------------------------------------------------*/
```

And here is the example stereo pair that can be used to test the example

3D Tracking Functions

The section discusses functions for tracking objects in 3d space using a stereo camera. Besides C API, there is DirectShow 3dTracker filter and the wrapper application 3dTracker. you may find a description how to test the filter on sample data.

3dTrackerCalibrateCameras

Simultaneously determines position and orientation of multiple cameras

```
CvBool cv3dTrackerCalibrateCameras(int num_cameras,
           const Cv3dTrackerCameraIntrinsics camera_intrinsics[],
           CvSize checkerboard_size,
           IplImage *samples[],
           Cv3dTrackerCameraInfo camera_info[]);
```

num_cameras
: the number of cameras to calibrate. This is the size of each of the three array parameters.

camera_intrinsics
: camera intrinsics for each camera, such as determined by CalibFilter.

checkerboard_size
: the width and height (in number of squares) of the checkerboard.

samples
: images from each camera, with a view of the checkerboard.

camera_info
: filled in with the results of the camera calibration. This is passed into 3dTrackerLocateObjects [p 15] to do tracking.

The function cv3dTrackerCalibrateCameras [p 14] searches for a checkerboard of the specified size in each of the images. For each image in which it finds the checkerboard, it fills in the corresponding slot in `camera_info` with the position and orientation of the camera relative to the checkerboard and sets the `valid` flag. If it finds the checkerboard in all the images, it returns true; otherwise it returns false.

This function does not change the members of the `camera_info` array that correspond to images in which the checkerboard was not found. This allows you to calibrate each camera independently, instead of simultaneously. To accomplish this, do the following:

1. clear all the `valid` flags before calling this function the first time;
2. call this function with each set of images;
3. check all the `valid` flags after each call. When all the `valid` flags are set, calibration is complete.

Note that this method works well only if the checkerboard is rigidly mounted; if it is handheld, all the cameras should be calibrated simultanously to get an accurate result. To ensure that all cameras are calibrated simultaneously, ignore the `valid` flags and use the return value to decide when calibration is complete.

3dTrackerLocateObjects

Determines 3d location of tracked objects

```
int  cv3dTrackerLocateObjects(int num_cameras,
         int num_objects,
         const Cv3dTrackerCameraInfo camera_info[],
         const Cv3dTracker2dTrackedObject tracking_info[],
         Cv3dTrackerTrackedObject tracked_objects[]);
```

num_cameras
: the number of cameras.

num_objects
: the maximum number of objects found by any camera. (Also the maximum number of objects returned in `tracked_objects`.)

camera_info
: camera position and location information for each camera, as determined by 3dTrackerCalibrateCameras [p 14] .

tracking_info
: the 2d position of each object as seen by each camera. Although this is specified as a one-dimensional array, it is actually a two-dimensional array: `const Cv3dTracker2dTrackedObject tracking_info[num_cameras][num_objects]`. The `id` field of any unused slots must be -1. Ids need not be ordered or consecutive.

tracked_objects
: filled in with the results.

The function cv3dTrackerLocateObjects [p 15] determines the 3d position of tracked objects based on the 2d tracking information from multiple cameras and the camera position and orientation information computed by 3dTrackerCalibrateCameras [p ??] . It locates any objects with the same `id` that are tracked by more than one camera. It fills in the `tracked_objects` array and returns the number of objects located. The `id` fields of any unused slots in `tracked_objects` are set to -1.

Frequently Asked Questions / Troubleshootings / HOWTOs

General Questions

How to install OpenCV properly?

Read installation guide

How can I get acquainted with OpenCV fast?

- Try to run Hawk (under Windows), load opencv\samples\c scripts and run them.
- Then you can move to higher-weight applications like facedetection, lkdemo, camshift etc.
- Also, scan through reference manual [p ??] - it contains some example code as well.
- Search OpenCV archives at http://groups.yahoo.com/group/OpenCV for the topic you are interesting in.
- Create new project on base of sample script and an OpenCV demo application and modify it as needed. There are application wizards for Microsoft Developer Studio that create OpenCV-aware projects; look for them at http://groups.yahoo.com/group/OpenCV (Files section - you have to be registered OpenCV@yahoogroups.com user) or at OpenCV SourceForge page. Also read below [p ??] how to create such a project from scratch

Where do I submit Bug reports for the computer vision library?

Send email to OpenCV@yahoogroups.com Subject: BUG <....your title...>

How do I send bug reports for the Intel® Image Processing Library?

Send email to developer_support@intel.com

How do I join the web group for the library?

Send email to OpenCV-subscribe@yahoogroups.com, after you are a member and select your logon, you can read the web group at http://groups.yahoo.com/group/OpenCV

How do I modify the web group so that I don’t receive email everyday?

To get the messages real time, or once a day as a daily digest, you can go to http://groups.yahoo.com/mygroups and choose your setting from the pull down list to the right of OpenCV.;

Ok, I found the group completely useless for me. How can I unsubscribe?

Mail to OpenCV-unsubscribe@yahoogroups.com with subject **[OpenCV]** and arbitrary message contents.

How do I get support for the Image Processing Library (IPL)?

For the Image Processing Library, all support questions should go through:
http://support.intel.com/support/performancetools/support.htm (for release libraries)
https://premier.intel.com/scripts-quad/welcomeplsb.asp (for beta libraries)

In beta 3 IPL and OpenCV conflict. How to resolve it?

To be completely independent from IPL, OpenCV duplicates declarations of IplImage and few other structures and constants if it is not told explicitly that IPL is present. Defining HAVE_IPL before including OpenCV headers or putting "#include <ipl.h>" before OpenCV headers resolves the conflict.

Does OpenCV works on other processors?

Yes, OpenCV itself is open source and it is quite portable, especially across 32-bit platforms. On the other hand, OpenCV can run much faster on Intel processors because of IPP [p ??] .

Windows® OS related Qs:

When I try to build one of the apps, I get an error, streams.h not found.

You need DirectShow SDK that is now a part of DirectX SDK.

1. Download DirectX SDK from msdn.microsoft.com/directx/ (It's huge, but you can download it by parts). If it doesn't work for you, consider HighGUI that can capture video via VFW or MIL
2. Install it TOGETHER WITH SAMPLES.
3. Open <DirectXSDKInstallFolder>\samples\Multimedia\DirectShow\BaseClasses\baseclasses.dsw. If there is no such file, it is that you either didn't install samples or the path has changed, in the latter case search for streams.h and open a workspace file (workspace files for Developer Studio .NET have different extension) located in the same folder.
4. Build the library in both Release in Debug configurations.
5. Copy the built libraries (in DirectX 8.x they are called strmbase.lib and strmbasd.lib) to <DirectXSDKInstallFolder>\lib.
6. In Developer Studio add the following paths:

   <DirectXSDKInstallFolder>\include
   <DirectXSDKInstallFolder>\samples\Multimedia\DirectShow\BaseClasses
   to the includes' search path (at Tools->Options->Directories->Include files in case of Developer Studio 6.0)

Add <DirectXSDKInstallFolder>\lib to the libraries' search path (at Tools->Options->Directories->Library files in case of Developer Studio 6.0)

NOTE: PUT THE ADDED LINES ON THE VERY TOP OF THE LISTS, OTHERWISE YOU WILL STILL GET COMPILER OR LINKER ERRORS. This is necessary, because Developer Studio 6.0 includes some older DirectX headers and libs that conflict with new DirectX SDK versions.

7. Enjoy!

After installing DirectX SDK I'm still getting linker error about undefined or redefined "TransInPlace" filter class constructors etc.

Read the instructions from the previous answer, especially about the order of search directories.

When I use try to use cvcam, it just crashes

Make sure, you registered ProxyTrans.ax and SyncFilter.ax

CamShiftDemo can not be run

Make sure, you registered CamShift.ax and you have DirectShow-compatible camera

How to register *.ax (DirectShow filter)?

Open the file (within explorer) using regsvr32.exe (under Win2000 it is done by Open with->Choose Program...->Browse...->c:\windows\system32\regsvr32.exe (path may be different). You may remember association to save clicks later.

Filter couldn't be registered (regsvr32 reports an error)

The most probable reason is that the filter requires some DLLs that are not in the path. In case of OpenCV make sure <OpenInstallFolder>\bin is in the path

LKDemo / HMMDemo reports an error during startup and no the view is completely black

To run either of these apps you will need VFW-compatible camera. At startup the programs iterate through registered video capture devices. It might be that they could not find one. Try to select the camera manually by pressing "tune capture parameters" (camera) toolbar button. Then, try to setup video format (the button on the left from camera) to make the camera work.

cvd.lib or cvd.dll are not found

cvd.dll means Debug version of cv.dll and cvd.lib is the import library for cvd.dll. Open <OpenCVInstallFolder>_dsw\opencv.dsw, select "cv" as active project and select "Win32 Debug" configuration. Build the library and you will get bin\cvd.dll and lib\cvd.lib files. The same is true for *all* of OpenCV components - name of binary, ending with d means Debug version.

When compiling HighGUI I get the error message "mil.h is not found"

mil.h is a part of Matrox Imaging Library (MIL) that is usually supplied with Matrox (or compatible) framegrabbers, such as Meteor, Meteor II etc.

- If you have such a framegrabber and MIL installed, add mil\include and mil\lib to the search paths within Developer Studio (submenu Tools->Options->Directories).
- If you do not have MIL, just ignore the error. The file mil.h is only required to build MIL-aware version of Highgui "Win32 MIL Debug" or "Win32 MIL Release". Select "Win32 Debug" or "Win32 Release" configuration of highgui (submenu Build->Set Active Configuration...) instead - these versions of highgui can still be used to grab video via VFW interface, work with AVIs and still images.

How can I debug DirectShow filter?

- Open workspace with the filter (e.g. opencv.dsw),
- select the filter as active project and build it in debug configuration,
- switch to explorer for a minute to register debug version of the filter (e.g. regsvr32 camshiftd.ax) (it needs to be done only when debug/release version are switched - not every time when filter is recompiled, because registry stores only the filter name),
- get back to Developer Studio and start debugging session (F5). It will ask, what application do you want to run to debug the module. You may choose camshiftdemo to debug camshift.ax and DirectX SDK tool graphedit to debug arbitrary DirectShow filter.
- Within graphedit build filter graph (e.g. camera->camshift->renderer)
- Save the graph (you may just load it next time)
- Set the breakpoint inside ::Transform method of the filter or in other location.
- Run the filter and ... have fun

How can I create DeveloperStudio project to start playing with OpenCV

(note: this is a lengthy answer)

To create your own OpenCV-based project in Developer Studio from scratch do the following:

1. Within Developer Studio create new application:
    1. select from menu "File"->"New..."->"Projects" tab. Choose "Win32 Application" or "Win32 console application" - the latter is the easier variant and the both sample projects have this type.
    2. type the project name and choose location
    3. you may create own workspace for the project ("Create new workspace") or include the new

project into the currently loaded workspace ("Add to current workspace").

4. click "next" button
5. choose "An empty project", click "Finish", "OK".

After the above steps done Developer Studio will create the project folder (by default it has the same name as the project), <project name>.dsp file and, optionally, <project name>.dsw,.ncb ... files if you create own workspace.

2. Add a file to the project:
   - select from menu "File"->"New..."->"Files" tab.
   - choose "C++ Source File", type file name and press "OK"
   - add OpenCV-related #include directives:

     ```
     #include "cv.h"
     /* #inlcude "cvaux.h" // experimental stuff (if need) */
     #include "highgui.h"
     ```

     Or, you may copy some existing file (say, opencv\samples\c\morphology.c) to the project folder, open it and add to the project (right click in editor view -> "Insert File into Project" -> <your project name>).
3. Customize project settings:
   - Activate project setting dialog by choosing menu item "Project"->"Settings...".
   - Select your project in the right pane.
   - Tune settings, common to both Release and Debug configurations:
     - Select "Settings For:"->"All Configurations"
     - Choose "C/C++" tab -> "Preprocessor" category -> "Additional Include Directories:". Add comma-separated relative (to the .dsp file) or absolute paths to opencv\cv\include, opencv\otherlibs\highgui and, optionally, opencv\cvaux\include.
     - Choose "Link" tab -> "Input" category -> "Additional library path:". Add the paths to all neccessary import libraries
   - Tune settings for "Debug" configuration
     - Select "Settings For:"->"Win32 Debug".
     - Choose "Link" tab -> "General" category -> "Object/library modules". Add space-separated cvd.lib, highguid.lib, cvauxd.lib (optionally)
     - You may also want to change location and name of output file. For example, if you want the output .exe file to be put into the project folder, rather than Debug/ subfolder, you may type ./<exe-name>d.exe in "Link" tab -> "General" category -> "Output file name:".
   - Tune settings for "Release" configuration
     - Select "Settings For:"->"Win32 Release".
     - Choose "Link" tab -> "General" category -> "Object/library modules". Add space-separated cv.lib, highgui.lib, cvaux.lib (optionally)
     - Optionally, you may change name of the .exe file: type ./<exe-name>.exe in "Link" tab -> "General" category -> "Output file name:".
4. Add dependency projects into workspace:
   - Choose from menu: "Project" -> "Insert project into workspace".
   - Select opencv\cv\make\cv.dsp.
   - Do the same for opencv\cvaux\make\cvaux.dsp, opencv\otherlibs\highgui\highgui.dsp.
   - Set dependencies:

- Choose from menu: "Project" -> "Dependencies..."
- For "cvaux" choose "cv",
- for "highgui" choose "cv",
- for your project choose all: "cv", "cvaux", "highgui".

The dependencies customization allows to automatically build debug versions of opencv libraries and rebuild the binaries if the sources are changed somehow.

5. That's it. Now compile and run everything.

Linux Related Qs:

TODO

Technical Questions on Library use:

How to access image pixels

(The coordinates are 0-based and counted from image origin, either top-left (img->origin=IPL_ORIGIN_TL) or bottom-left (img->origin=IPL_ORIGIN_BL)

- Suppose, we have 8-bit 1-channel image I (IplImage* img):

```
I(x,y) ~ ((uchar*)(img->imageData + img->widthStep*y))[x]
```

- Suppose, we have 8-bit 3-channel image I (IplImage* img):

```
I(x,y)blue  ~ ((uchar*)(img->imageData + img->widthStep*y))[x*3]
I(x,y)green  ~ ((uchar*)(img->imageData + img->widthStep*y))[x*3+1]
I(x,y)red  ~ ((uchar*)(img->imageData + img->widthStep*y))[x*3+2]
```

e.g. increasing brightness of point (100,100) by 30 can be done this way:

```
CvPoint pt = {100,100};
((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3] += 30;
((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3+1] += 30;
((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3+2] += 30;
```

or more efficiently

```
CvPoint pt = {100,100};
uchar* temp_ptr = &((uchar*)(img->imageData + img->widthStep*pt.y))[x*3];
temp_ptr[0] += 30;
temp_ptr[1] += 30;
temp_ptr[2] += 30;
```

- Suppose, we have 32-bit floating point, 1-channel image I (IplImage* img):

```
I(x,y) ~ ((float*)(img->imageData + img->widthStep*y))[x]
```

- Now, the general case: suppose, we have N-channel image of type T:

```
I(x,y)c ~ ((T*)(img->imageData + img->widthStep*y))[x*N + c]
or you may use macro CV_IMAGE_ELEM( image_header, elemtype, y, x_Nc )
I(x,y)c ~ CV_IMAGE_ELEM( img, T, y, x*N + c )
```

There are functions that work with arbitrary (up to 4-channel) images and matrices (cvGet2D, cvSet2D), but they are pretty slow.

How to access matrix elements?

The technique is very similar. (In the samples below i - 0-based row index, j - 0-based column index)

- Suppose, we have 32-bit floating point real matrix M (CvMat* mat):

  ```
  M(i,j) ~ ((float*)(mat->data.ptr + mat->step*i))[j]
  ```

- Suppose, we have 64-bit floating point complex matrix M (CvMat* mat):

  ```
  Re M(i,j) ~ ((double*)(mat->data.ptr + mat->step*i))[j*2]
  Im M(i,j) ~ ((double*)(mat->data.ptr + mat->step*i))[j*2+1]
  ```

- For single-channel matrices there is a macro CV_MAT_ELEM(matrix, elemtype, row, col), i.e. for 32-bit floating point real matrix

  ```
  M(i,j) ~ CV_MAT_ELEM( mat, float, i, j ),
  ```

  e.g. filling 3x3 identity matrix:

  ```
  CV_MAT_ELEM( mat, float, 0, 0 ) = 1.f;
  CV_MAT_ELEM( mat, float, 0, 1 ) = 0.f;
  CV_MAT_ELEM( mat, float, 0, 2 ) = 0.f;
  CV_MAT_ELEM( mat, float, 1, 0 ) = 0.f;
  CV_MAT_ELEM( mat, float, 1, 1 ) = 1.f;
  CV_MAT_ELEM( mat, float, 1, 2 ) = 0.f;
  CV_MAT_ELEM( mat, float, 2, 0 ) = 0.f;
  CV_MAT_ELEM( mat, float, 2, 1 ) = 0.f;
  CV_MAT_ELEM( mat, float, 2, 2 ) = 1.f;
  ```

How to process my data with OpenCV

Suppose, you have 300x200 32-bit floating point array, that resides in 60000-element array.

```
int cols = 300, rows = 200;
float* myarr = new float[rows*cols];

// step 1) initializing CvMat header
CvMat mat = cvMat( rows, cols,
                   CV_32FC1, // 32-bit floating-point, single channel type
                   myarr // user data pointer (no data is copied)
                   );
// step 2) using cv functions, e.g. calculating l2 (Frobenius) norm
double norm = cvNorm( &mat, 0, CV_L2 );

...
delete myarr;
```

Other scenaria are described in the reference manual. See cvCreateMatHeader, cvInitMatHeader, cvCreateImageHeader, cvSetData etc.

How to load and display image

```
/* usage: prog <image_name> */
#include "cv.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    IplImage* img;
    if( argc == 2 && (img = cvLoadImage( argv[1], 1)) != 0 )
    {
        cvNamedWindow( "Image view", 1 );
        cvShowImage( "Image view", img );
        cvWaitKey(0); // very important
        cvDestroyWindow( "Image view" );
        cvReleaseImage( &img );
        return 0;
    }
    return -1;
}
```

How to find and process contours

Look at squares demo

How to calibrate camera using OpenCV

TODO

Basic Structures and Operations Reference

Helper structures

CvPoint

2D point with integer coordinates

```
typedef struct CvPoint
{
    int x; /* x-coordinate, usually zero-based */
    int y; /* y-coordinate, usually zero-based */
}
CvPoint;

/* the constructor function */
inline CvPoint cvPoint( int x, int y );

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

CvPoint2D32f

2D point with floating-point coordinates

```
typedef struct CvPoint2D32f
{
    float x; /* x-coordinate, usually zero-based */
    float y; /* y-coordinate, usually zero-based */
}
CvPoint2D32f;

/* the constructor function */
inline CvPoint2D32f cvPoint2D32f( double x, double y );

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

CvPoint3D32f

3D point with floating-point coordinates

```
typedef struct CvPoint3D32f
{
    float x; /* x-coordinate, usually zero-based */
    float y; /* y-coordinate, usually zero-based */
    float z; /* z-coordinate, usually zero-based */
}
CvPoint3D32f;

/* the constructor function */
inline CvPoint3D32f cvPoint3D32f( double x, double y, double z );
```

CvSize

pixel-accurate size of a rectangle

```
typedef struct CvSize
{
    int width; /* width of the rectangle */
    int height; /* height of the rectangle */
}
CvSize;

/* the constructor function */
inline CvSize cvSize( int width, int height );
```

CvSize2D32f

sub-pixel accurate size of a rectangle

```
typedef struct CvSize2D32f
{
    float width; /* width of the box */
    float height; /* height of the box */
}
CvSize2D32f;

/* the constructor function */
inline CvSize2D cvSize32f( double width, double height );
```

CvRect

offset and size of a rectangle

```
typedef struct CvRect
{
    int x; /* x-coordinate of the left-most rectangle corner[s] */
    int y; /* y-coordinate of the top-most or bottom-most
              rectangle corner[s] */
    int width; /* width of the rectangle */
    int height; /* height of the rectangle */
}
CvRect;

/* the constructor function */
inline CvRect cvRect( int x, int y, int width, int height );
```

CvScalar

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

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

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

/* the constructor function: initializes val[0] with val0, val[1]...val[3] with zeros */
inline CvScalar cvRealScalar( double val0 );
```

Array structures

CvMat

Multi-channel matrix

```
    typedef struct CvMat
    {
        int type; /* CvMat signature (CV_MAT_MAGIC_VAL), element type and flags */
        int step; /* full row length in bytes */

        int* refcount; /* underlying data reference counter */

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

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

        union
        {
            int cols;
            int width;
        };
    #else
        int rows; /* number of rows */
        int cols; /* number of columns */
    #endif

    } CvMat;
```

CvMatND

Multi-dimensional dense multi-channel array

```
    typedef struct CvMatND
    {
        int type; /* CvMatND signature (CV_MATND_MAGIC_VAL), element type and flags */
        int dims; /* number of array dimensions */

        int* refcount; /* underlying data reference counter */
```

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

    /* pairs (number of elements, distance between elements in bytes) for
       every dimension */
    struct
    {
        int size;
        int step;
    }
    dim[CV_MAX_DIM];

} CvMatND;
```

CvSparseMat

Multi-dimensional sparse multi-channel array

```
typedef struct CvSparseMat
{
    int type; /* CvSparseMat signature (CV_SPARSE_MAT_MAGIC_VAL), element type and flags */
    int dims; /* number of dimensions */
    int* refcount; /* reference counter - not used */
    struct CvSet* heap; /* a pool of hashtable nodes */
    void** hashtable; /* hashtable: each entry has a list of nodes
                         having the same "hashvalue modulo hashsize" */
    int hashsize; /* size of hashtable */
    int total; /* total number of sparse array nodes */
    int valoffset; /* value offset in bytes for the array nodes */
    int idxoffset; /* index offset in bytes for the array nodes */
    int size[CV_MAX_DIM]; /* array of dimension sizes */

} CvSparseMat;
```

IplImage

IPL image header

```
typedef struct _IplImage
{
    int  nSize;         /* sizeof(IplImage) */
    int  ID;            /* version (=0)*/
    int  nChannels;     /* Most of OpenCV functions support 1,2,3 or 4 channels */
    int  alphaChannel;  /* ignored by OpenCV */
    int  depth;         /* pixel depth in bits: IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16S,
                           IPL_DEPTH_32S, IPL_DEPTH_32F and IPL_DEPTH_64F are supported */
    char colorModel[4]; /* ignored by OpenCV */
    char channelSeq[4]; /* ditto */
    int  dataOrder;     /* 0 - interleaved color channels, 1 - separate color channels.
                           cvCreateImage can only create interleaved images */
```

```
    int  origin;        /* 0 - top-left origin,
                           1 - bottom-left origin (Windows bitmaps style) */
    int  align;         /* Alignment of image rows (4 or 8).
                           OpenCV ignores it and uses widthStep instead */
    int  width;         /* image width in pixels */
    int  height;        /* image height in pixels */
    struct _IplROI *roi;/* image ROI. when it is not NULL, this specifies image region to process */
    struct _IplImage *maskROI; /* must be NULL in OpenCV */
    void  *imageId;     /* ditto */
    struct _IplTileInfo *tileInfo; /* ditto */
    int  imageSize;     /* image data size in bytes
                           (=image->height*image->widthStep
                           in case of interleaved data)*/
    char *imageData;  /* pointer to aligned image data */
    int  widthStep;   /* size of aligned image row in bytes */
    int  BorderMode[4]; /* border completion mode, ignored by OpenCV */
    int  BorderConst[4]; /* ditto */
    char *imageDataOrigin; /* pointer to a very origin of image data
                              (not necessarily aligned) -
                              it is needed for correct image deallocation */
}
IplImage;
```

The structure *IplImage* came from *Intel Image Processing Library* where the format is native. OpenCV supports only subset of the possible IplImage formats:

- `alphaChannel` is ignored by OpenCV.
- `colorModel` and `channelSeq` are ignored by OpenCV. The single OpenCV function cvCvtColor [p ??] working with color spaces takes the source and destination color spaces as a parameter.
- `dataOrder` must be IPL_DATA_ORDER_PIXEL (the color channels are interleaved), however selected channels of planar images can be processed as well if COI is set.
- `align` is ignored by OpenCV, while `widthStep` is used to access to subsequent image rows.
- `maskROI` is not supported. The function that can work with mask take it as a separate parameter. Also the mask in OpenCV is 8-bit, whereas in IPL it is 1-bit.
- `tileInfo` is not supported.
- `BorderMode` and `BorderConst` are not supported. Every OpenCV function working with a pixel neigborhood uses a single hard-coded border mode (most often, replication).

Besides the above restrictions, OpenCV handles ROI differently. It requires that the sizes or ROI sizes of all source and destination images match exactly (according to the operation, e.g. for cvPyrDown [p ??] destination width(height) must be equal to source width(height) divided by 2 ±1), whereas IPL processes the intersection area - that is, the sizes or ROI sizes of all images may vary independently.

CvArr

Arbitrary array

```
typedef void CvArr;
```

CvArr* [p ??] is used *only* as a function parameter to specify that the function accepts arrays of more than a single type, for example IplImage* and CvMat*. The particular array type is determined in runtime from looking at the first 4-byte field of array header.

Arrays: Allocation, deallocation, copying; setting and retrieving parts

Alloc

Allocates memory buffer

```
void* cvAlloc( size_t size );
```

size
: Buffer size in bytes.

The function cvAlloc [p 36] allocates *size* bytes and returns pointer to the allocated buffer. In case of error the function reports an error and returns NULL pointer. By default cvAlloc calls icvAlloc which itself calls malloc, however it is possible to assign user-defined memory allocation/deallocation functions using cvSetMemoryManager [p ??] function.

Free

Deallocates memory buffer

```
void cvFree( void** buffer );
```

buffer
: Double pointer to released buffer.

The function cvFree [p 36] deallocates memory buffer allocated by cvAlloc [p 36] . It clears the pointer to buffer upon exit, that is why the double pointer is used. If *buffer is already NULL, the function does nothing

CreateImage

Creates header and allocates data

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

size
: Image width and height.

depth
: Bit depth of image elements. Can be one of:
IPL_DEPTH_8U - unsigned 8-bit integers
IPL_DEPTH_8S - signed 8-bit integers
IPL_DEPTH_16S - signed 16-bit integers

IPL_DEPTH_32S - signed 32-bit integers
IPL_DEPTH_32F - single precision floating-point numbers
IPL_DEPTH_64F - double precision floating-point numbers

channels
Number of channels per element(pixel). Can be 1, 2, 3 or 4. The channels are interleaved, for example the usual data layout of a color image is:
b0 g0 r0 b1 g1 r1 ...
Although in general IPL image format can store non-interleaved images as well and some of OpenCV can process it, this function can create interleaved images only.

The function cvCreateImage [p 36] creates the header and allocates data. This call is a shortened form of

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

CreateImageHeader

Allocates, initializes, and returns structure IplImage

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

size
Image width and height.

depth
Image depth (see CreateImage).

channels
Number of channels (see CreateImage).

The function cvCreateImageHeader [p 37] allocates, initializes, and returns the structure *IplImage*. This call is an analogue of

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

though it does not use IPL functions by default (see also *CV_TURN_ON_IPL_COMPATIBILITY* macro)

ReleaseImageHeader

Releases header

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

image
Double pointer to the deallocated header.

The function cvReleaseImageHeader [p 37] releases the header. This call is an analogue of

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

though it does not use IPL functions by default (see also *CV_TURN_ON_IPL_COMPATIBILITY*)

ReleaseImage

Releases header and image data

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

image
Double pointer to the header of the deallocated image.

The function cvReleaseImage [p 38] releases the header and the image data. This call is a shortened form of

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

InitImageHeader

Initializes allocated by user image header

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

image
Image header to initialize.
size
Image width and height.
depth
Image depth (see CreateImage).
channels
Number of channels (see CreateImage).
origin
IPL_ORIGIN_TL or *IPL_ORIGIN_BL*.

align
Alignment for image rows, typically 4 or 8 bytes.

The function cvInitImageHeader [p 38] initializes the image header structure without memory allocation.

CloneImage

Makes a full copy of image

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

image
Original image.

The function cvCloneImage [p 39] makes a full copy of the image including header, ROI and data

---

SetImageCOI

Sets channel of interest to given value

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

image
Image header.
coi
Channel of interest.

The function cvSetImageCOI [p 39] sets the channel of interest to a given value. Value 0 means that all channels are selected, 1 means that the first channel is selected etc. If ROI is *NULL* and *coi != 0*, ROI is allocated. Note that most of OpenCV functions do not support COI, so to process separate image/matrix channel one may copy (via cvCopy [p 58] or cvCvtPixToPlane [p 62]) the channel to separate image/matrix, process it and copy the result back (via cvCopy [p 58] or cvCvtPlaneToPix [p 63]) if need.

GetImageCOI

Returns index of channel of interest

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

image
Image header.

The function cvGetImageCOI [p 39] returns channel of interest of the image (it returns 0 if all the channels are selected).

SetImageROI

Sets image ROI to given rectangle

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

image
: Image header.

rect
: ROI rectangle.

The function cvSetImageROI [p 40] 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. Unlike COI, most of OpenCV functions do support ROI and treat it in a way as it would be a separate image (for example, all the pixel coordinates are counted from top-left or bottom-left (depending on the image origin) corner of ROI)

ResetImageROI

Releases image ROI

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

image
: Image header.

The function cvResetImageROI [p 40] releases image ROI. After that the whole image is considered selected. The similar result can be achieved by

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

But the latter variant does not deallocate *image->roi*.

GetImageROI

Returns image ROI coordinates

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

image
: Image header.

The function cvGetImageROI [p 40] returns image ROI coordinates. The rectangle cvRect [p ??] (0,0,image->width,image->height) is returned if there is no ROI

CreateMat

Creates new matrix

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

rows
: Number of rows in the matrix.

cols
: Number of columns in the matrix.

type
: Type of the matrix elements. Usually it is specified in form *CV_<bit_depth>(S|U|F)C<number_of_channels>*, for example: *CV_8UC1* means an 8-bit unsigned single-channel matrix, *CV_32SC2* means a 32-bit signed matrix with two channels.

The function cvCreateMat [p 41] allocates header for the new matrix and underlying data, and returns a pointer to the created matrix. It is a short form for:

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

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

CreateMatHeader

Creates new matrix header

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

rows
: Number of rows in the matrix.

cols
: Number of columns in the matrix.

type
: Type of the matrix elements (see cvCreateMat [p 41]).

The function cvCreateMatHeader [p 41] allocates new matrix header and returns pointer to it. The matrix data can further be allocated using cvCreateData [p 46] or set explicitly to user-allocated data via cvSetData [p 47] .

ReleaseMat

Deallocates matrix

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

mat
: Double pointer to the matrix.

The function cvReleaseMat [p 42] decrements the matrix data reference counter and releases matrix header:

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

InitMatHeader

Initializes matrix header

```
void cvInitMatHeader( CvMat* mat, int rows, int cols, int type,
                      void* data=0, int step=CV_AUTOSTEP );
```

mat
: Pointer to the matrix header to be initialized.

rows
: Number of rows in the matrix.

cols
: Number of columns in the matrix.

type
: Type of the matrix elements.

data
: Optional data pointer assigned to the matrix header.

step
: Full row width in bytes of the data assigned. By default, the minimal possible step is used, i.e., no gaps is assumed between subsequent rows of the matrix.

The function cvInitMatHeader [p 42] initializes already allocated CvMat [p 33] structure. It can be used to process raw data with OpenCV matrix functions.

For example, the following code computes matrix product of two matrices, stored as ordinary arrays.

Calculating Product of Two Matrices

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

   double b[] = { 1, 5, 9,
```

```
                2, 6, 10,
                3, 7, 11,
                4, 8, 12 };

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

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

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

Mat

Initializes matrix header (light-weight variant)

```
CvMat cvMat( int rows, int cols, int type, void* data = 0 );
```

rows
: Number of rows in the matrix.

cols
: Number of columns in the matrix.

type
: Type of the matrix elements (see CreateMat).

data
: Optional data pointer assigned to the matrix header.

The function cvMat [p 43] is a fast inline substitution for cvInitMatHeader [p 42] . Namely, it is equivalent to:

```
 CvMat mat;
 cvInitMatHeader( &mat, rows, cols, type, data, CV_AUTOSTEP );
```

CloneMat

Creates matrix copy

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

mat
: Input matrix.

The function cvCloneMat [p 43] creates a copy of input matrix and returns the pointer to it.

CreateMatND

Creates multi-dimensional dense array

```
CvMatND* cvCreateMatND( int dims, int* size, int type );
```

dims
: Number of array dimensions. It must not exceed CV_MAX_DIM (=16 by default, though it may be changed at build time)

size
: Array of dimension sizes.

type
: Type of array elements. The same as for CvMat [p 33]

The function cvCreateMatND [p 44] allocates header for multi-dimensional dense array and the underlying data, and returns pointer to the created array. It is a short form for:

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

Array data is stored row by row. All the rows are aligned by 4 bytes.

CreateMatNDHeader

Creates new matrix header

```
CvMatND* cvCreateMatNDHeader( int dims, int* size, int type );
```

dims
: Number of array dimensions.

size
: Array of dimension sizes.

type
: Type of array elements. The same as for CvMat

The function cvCreateMatND [p 44] allocates header for multi-dimensional dense array. The array data can further be allocated using cvCreateData [p 46] or set explicitly to user-allocated data via cvSetData [p 47] .

ReleaseMatND

Deallocates multi-dimensional array

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

mat
: Double pointer to the array.

The function cvReleaseMatND [p 44] decrements the array data reference counter and releases the array header:

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

InitMatNDHeader

Initializes multi-dimensional array header

```
void cvInitMatNDHeader( CvMatND* mat, int dims, int* size, int type, void* data=0 );
```

mat
: Pointer to the array header to be initialized.

rows
: Number of rows in the matrix.

cols
: Number of columns in the matrix.

type
: Type of the matrix elements.

data
: Optional data pointer assigned to the matrix header.

The function cvInitMatNDHeader [p 45] initializes already allocated CvMatND [p 33] structure.

CloneMatND

Creates full copy of multi-dimensional array

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

mat
: Input array.

The function cvCloneMatND [p 45] creates a copy of input array and returns pointer to it.

DecRefData

Decrements array data reference counter

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

array
: array header.

The function cvDecRefData [p 45] decrements CvMat [p 33] or CvMatND [p 33] data reference counter if the reference counter pointer is not NULL and deallocates the data if the counter reaches zero. In the current implementation the reference counter is not NULL only if the data was allocated using cvCreateData [p 46] function, in other cases such as:
external data was assigned to the header using cvSetData [p 47]
the matrix header presents a part of a larger matrix or image
the matrix header was converted from image or n-dimensional matrix header
the reference counter is set to NULL and thus it is not decremented. Whenever the data is deallocated or not, the data pointer and reference counter pointers are cleared by the function.

IncRefData

Increments array data reference counter

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

array
: array header.

The function cvIncRefData [p 46] increments CvMat [p 33] or CvMatND [p 33] data reference counter and returns the new counter value if the reference counter pointer is not NULL, otherwise it returns zero.

CreateData

Allocates array data

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

array
: Array header.

The function cvCreateData [p 46] allocates image, matrix or multi-dimensional array data. Note that in case of matrix types OpenCV allocation functions are used and in case of IplImage they are used too unless *CV_TURN_ON_IPL_COMPATIBILITY* was called. In the latter case IPL functions are used to allocate the data

---

ReleaseData

Releases array data

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

array
: Array header

The function cvReleaseData [p 47] releases the array data. In case of CvMat [p 33] or CvMatND [p 33] it simply calls cvDecRefData(), that is the function can not deallocate external data. See also the note to cvCreateData [p 46] .

SetData

Assigns user data to the array header

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

array
: Array header.

data
: User data.

step
: Full row length in bytes.

The function cvSetData [p 47] assigns user data to the array header. Header should be initialized before using cvCreate*Header, cvInit*Header or cvMat [p 43] (in case of matrix) function.

GetRawData

Retrieves low-level information about the array

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

array
: Array header.

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

step
: Output full row length in bytes.

roiSize
: Output ROI size.

The function cvGetRawData [p 47] fills output variables with low-level information about the array data. All output parameters are optional, so some of the pointers may be set to *NULL*. If the array is *IplImage* with ROI set, parameters of ROI are returned.

The following example shows how to get access to array elements using this function.

Using GetRawData to calculate absolute value of elements of a single-channel floating-point array.

```
float* data;
int step;

CvSize size;
int x, y;

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

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

GetMat

Returns matrix header for arbitrary array

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

arr
: Input array.

mat
: Pointer to CvMat [p 33] structure used as a temporary buffer.

coi
: Optional output parameter for storing COI.

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

The function cvGetMat [p 48] returns matrix header for the input array that can be matrix - CvMat* [p ??] , image - *IplImage** or multi-dimensional dense array - CvMatND* [p ??] (latter case is allowed only if *allowND != 0*) . In the case of matrix the function simply returns the input pointer. In the case of *IplImage** or CvMatND* [p ??] it initializes *mat* structure with parameters of the current image ROI and returns pointer to this temporary structure. Because COI is not supported by CvMat [p 33] , it is returned separately.

The function provides an easy way to handle both types of array - *IplImage* and CvMat [p 33] -, using the same code. Reverse transform from CvMat [p 33] to *IplImage* can be done using cvGetImage [p 49] function.

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

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

GetImage

Returns image header for arbitrary array

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

arr
: Input array.

image_header
: Pointer to *IplImage* structure used as a temporary buffer.

The function cvGetImage [p 49] returns image header for the input array that can be matrix - CvMat* [p ??] , or image - *IplImage**. In the case of image the function simply returns the input pointer. In the case of CvMat* [p ??] it initializes *image_header* structure with parameters of the input matrix. Note that if we transform *IplImage* to CvMat [p 33] and then transform CvMat back to IplImage, we can get different headers if the ROI is set, and thus some IPL functions that calculate image stride from its width and align may fail on the resultant image.

GetSubRect

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

```
CvMat* cvGetSubRect( const CvArr* array, CvMat* subarr, CvRect rect );
```

array
: Input array.

subarr
: Pointer to the resultant subarray header.

rect
: Zero-based coordinates of the rectangle of interest.

The function cvGetSubRect [p 49] returns header, corresponding to a specified rectangle of the input array. In other words, it allows the user to treat a rectangular part of input array as a stand-alone array. ROI is taken into account by the function so the sub-array of ROI is really extracted.

GetRow, GetRows

Returns array row or row span

```
CvMat* cvGetRow( const CvArr* arr, CvMat* subarr, int row );
CvMat* cvGetRows( const CvArr* arr, CvMat* subarr, int start_row, end_row );
```

arr
: Input array.

subarr
: Pointer to the resulting sub-array header.

row
: Zero-based index of the selected row.

start_row
: Zero-based index of the starting row (inclusive) of the span.

end_row
: Zero-based index of the ending row (exclusive) of the span.

The functions *GetRow* and *GetRows* return the header, corresponding to a specified row/row span of the input array. Note that *GetRow* is a shortcut for cvGetRows [p ??] :

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

GetCol, GetCols

Returns array column or column span

```
CvMat* cvGetCol( const CvArr* arr, CvMat* subarr, int col );
CvMat* cvGetCols( const CvArr* arr, CvMat* subarr, int start_col, end_col );
```

arr
: Input array.

subarr
: Pointer to the resulting sub-array header.

col
: Zero-based index of the selected column.

start_col
: Zero-based index of the starting column (inclusive) of the span.

end_col
: Zero-based index of the ending column (exclusive) of the span.

The functions *GetCol* and *GetCols* return the header, corresponding to a specified column/column span of the input array. Note that *GetCol* is a shortcut for cvGetCols [p ??] :

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

GetDiag

Returns one of array diagonals

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

arr
: Input array.

subarr
: Pointer to the resulting sub-array header.

diag
: Array diagonal. Zero corresponds to the main diagonal, -1 corresponds to the diagonal above the main etc., 1 corresponds to the diagonal below the main etc.

The function cvGetDiag [p 51] returns the header, corresponding to a specified diagonal of the input array.

GetSize

Returns size of matrix or image ROI

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

arr
: array header.

The function cvGetSize [p 51] returns number of rows (CvSize::height) and number of columns (CvSize::width) of the input matrix or image. In case of image the size of ROI is returned.

CreateSparseMat

Creates sparse array

```
CvSparseMat* cvCreateSparseMat( int dims, int* size, int type );
```

dims
: Number of array dimensions. It must not exceed CV_MAX_DIM (=16 by default, though it may be changed at build time)

size
: Array of dimension sizes.

type
: Type of array elements. The same as for CvMat

The function cvCreateSparseMat [p 51] allocates multi-dimensional sparse array. Initially the array contain no elements, that is cvGet*D will return zero for every index

---

ReleaseSparseMat

Deallocates sparse array

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

mat
: Double pointer to the array.

The function cvReleaseSparseMat [p 52] releases the sparse array and clears the array pointer upon exit

---

CloneSparseMat

Creates full copy of sparse array

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

mat
: Input array.

The function cvCloneSparseMat [p 52] creates a copy of the input array and returns pointer to the copy.

InitSparseMatIterator

Initializes sparse array elements iterator

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

mat
: Input array.

matIterator
: Initialized iterator.

The function cvInitSparseMatIterator [p 52] initializes iterator of sparse array elements and returns pointer to the first element, or NULL if the array is empty.

GetNextSparseNode

Initializes sparse array elements iterator

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

matIterator
Sparse array iterator.

The function cvGetNextSparseNode [p 52] moves iterator to the next sparse matrix element and returns pointer to it. In the current version there is no any particular order of the elements, because they are stored in hash table. The sample below demonstrates how to iterate through the sparse matrix:

Using cvInitSparseMatIterator [p 52] and cvGetNextSparseNode [p 52] to calculate sum of floating-point sparse array.

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

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

    sum += val;
}

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

GetElemType

Returns type of array elements

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

arr
Input array.

The functions *GetElemType* returns type of the array elements as it is described in cvCreateMat discussion:

```
CV_8UC1 ... CV_64FC4
```

GetDims, GetDimSize

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

```
int cvGetDims( const CvArr* arr, int* size=0 );
int cvGetDimSize( const CvArr* arr, int index );
```

arr
: Input array.

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

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

The function cvGetDims [p 54] returns number of array dimensions and their sizes. In case of *IplImage* or CvMat [p 33] it always returns 2 regardless of number of image/matrix rows. The function cvGetDimSize [p ??] returns the particular dimension size (number of elements per that dimension). For example, the following code calculates total number of array elements:

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

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

Ptr*D

Return pointer to the particular array element

```
uchar* cvPtr1D( const CvArr* arr, int idx0, int* type=0 );
uchar* cvPtr2D( const CvArr* arr, int idx0, int idx1, int* type=0 );
uchar* cvPtr3D( const CvArr* arr, int idx0, int idx1, int idx2, int* type=0 );
uchar* cvPtrND( const CvArr* arr, int* idx, int* type=0 );
```

arr
: Input array.

idx0
: The first zero-based component of the element index

idx1
: The second zero-based component of the element index

idx2
: The third zero-based component of the element index

idx
: Array of the element indices

type
: Optional output parameter: type of matrix elements

The functions cvPtr*D [p 54] return pointer to the particular array element. Number of array dimension should match to the number of indices passed to the function except for cvPtr1D [p ??] function that can be used for sequential access to 1D, 2D or nD dense arrays.

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

All these as well as other functions accessing array elements (cvGet[Real]*D [p ??] , cvSet[Real]*D [p ??]) raise an error in case if the element index is out of range.

Get*D

Return the particular array element

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

arr
: Input array.

idx0
: The first zero-based component of the element index

idx1
: The second zero-based component of the element index

idx2
: The third zero-based component of the element index

idx
: Array of the element indices

The functions cvGet*D [p 55] return the particular array element. In case of sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions)

GetReal*D

Return the particular element of single-channel array

```
double cvGetReal1D( const CvArr* arr, int idx0 );
double cvGetReal2D( const CvArr* arr, int idx0, int idx1 );
double cvGetReal3D( const CvArr* arr, int idx0, int idx1, int idx2 );
double cvGetRealND( const CvArr* arr, int* idx );
```

arr
: Input array. Must have a single channel.

idx0
: The first zero-based component of the element index

idx1
: The second zero-based component of the element index

idx2
: The third zero-based component of the element index

idx
: Array of the element indices

The functions cvGetReal*D [p 55] return the particular element of single-channel array. If the array has multiple channels, runtime error is raised. Note that cvGet*D [p 55] function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

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

---

mGet

Return the particular element of single-channel floating-point matrix

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

mat
: Input matrix.

row
: The zero-based index of row.

col
: The zero-based index of column.

The function cvmGet [p 56] is a fast replacement for cvGetReal2D [p ??] in case of single-channel floating-point matrices. It is faster because it is inline, it does less checks for array type and array element type and it checks for the row and column ranges only in debug mode.

Set*D

Change the particular array element

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

arr
: Input array.

idx0
: The first zero-based component of the element index

idx1
: The second zero-based component of the element index

idx2
: The third zero-based component of the element index

idx
: Array of the element indices

new_value
: The assigned value

The functions cvSet*D [p 56] assign the new value to the particular element of array. In case of sparse array the functions create the node if it does not exist yet

---

SetReal*D

Change the particular array element

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

arr
: Input array.

idx0
: The first zero-based component of the element index

idx1
: The second zero-based component of the element index

idx2
: The third zero-based component of the element index

idx
: Array of the element indices

new_value
: The assigned value

The functions cvSetReal*D [p 57] assign the new value to the particular element of single-channel array. If the array has multiple channels, runtime error is raised. Note that cvSet*D [p 56] function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

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

---

mSet

Return the particular element of single-channel floating-point matrix

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

mat
: The matrix.

row
: The zero-based index of row.

col
: The zero-based index of column.

value
: The new value of the matrix element

The function cvmSet [p 58] is a fast replacement for cvSetReal2D [p ??] in case of single-channel floating-point matrices. It is faster because it is inline, it does less checks for array type and array element type and it checks for the row and column ranges only in debug mode.

Clear*D

Clears the particular array element

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

arr
: Input array.

idx
: Array of the element indices

The function cvClearND [p 58] clears (sets to zero) the particular element of dense array or deletes the element of sparse array. If the element does not exists, the function does nothing.

Copy

Copies one array to another

```
void cvCopy( const CvArr* A, CvArr* B, const CvArr* mask =0);
```

A
: The source array.

B
: The destination array.

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

The function cvCopy [p 58] copies selected elements from input array to output array:

B(I)=A(I) if mask(I)!=0.

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

Set

Sets every element of array to given value

```
void cvSet( CvArr* A, CvScalar S, const CvArr* mask=0 );
```

A
The destination array.
S
Fill value.
mask
Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function cvSet [p 59] copies scalar *S* to every selected element of the destination array:

```
A(I)=S if mask(I)!=0
```

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

SetZero

Clears the array

```
void cvSetZero( CvArr* arr );
#define cvZero cvSetZero
```

arr
array to be cleared.

The function cvSetZero [p 59] clears the array. In case of dense arrays (CvMat, CvMatND or IplImage) cvZero(array) is equivalent to cvSet(array,cvScalarAll(0),0), but the function can clear sparse arrays by removing all the array elements

---

Arrays: Conversions, transformations, basic operations

Reshape

Changes shape of matrix/image without copying data

```
CvMat* cvReshape( const CvArr* array, CvMat* header, int new_cn, int new_rows=0 );
```

array
: Input array.

header
: Output header to be filled.

new_cn
: New number of channels. *new_cn = 0* means that number of channels remains unchanged.

new_rows
: New number of rows. *new_rows = 0* means that number of rows remains unchanged unless it needs to be changed according to *new_cn* value. destination array to be changed.

The function cvReshape [p 60] initializes CvMat header so that it points to the same data as the original array but has different shape - different number of channels, different number of rows or both.

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

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

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

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

ReshapeMatND

Changes shape of multi-dimensional array w/o copying data

```
CvArr* cvReshapeMatND( const CvArr* array,
                       int sizeof_header, CvArr* header,
                       int new_cn, int new_dims, int* new_sizes );

#define cvReshapeND( arr, header, new_cn, new_dims, new_sizes )   \
      cvReshapeMatND( (arr), sizeof(*(header)), (header),         \
                      (new_cn), (new_dims), (new_sizes))
```

array
Input array.
sizeof_header
Size of output header to distinguish between IplImage, CvMat and CvMatND output headers.
header
Output header to be filled.
new_cn
New number of channels. *new_cn = 0* means that number of channels remains unchanged.
new_dims
New number of dimensions. *new_dims = 0* means that number of dimensions remains the same.
new_sizes
Array of new dimension sizes. Only *new_dims-1* values are used, because the total number of elements must remain the same. Thus, if *new_dims = 1*, *new_sizes* array is not used

The function cvReshapeMatND [p 60] is an advanced version of cvReshape [p 60] that can work with multi-dimensional arrays as well (though, it can work with ordinary images and matrices) and change the number of dimensions. Below are the two samples from the cvReshape [p 60] description rewritten using cvReshapeMatND [p 60] :

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

...

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

Repeat

Fill destination array with tiled source array

```
void cvRepeat( const CvArr* A, CvArr* B );
```

A
Source array, image or matrix.
B
Destination array, image or matrix.

The function cvRepeat [p 61] fills the destination array with source array tiled:

```
B(i,j)=A(i%rows(A), j%cols(A))
```

where "%" means "modulo" operation. So the destination array may be as larger as well as smaller than the source array.

Flip

Flip a 2D array around vertical, horizontall or both axises

```
void  cvFlip( const CvArr* A, CvArr* B=0, int flip_mode=0);
#define cvMirror cvFlip
```

A
: Source array.

B
: Destination array. If *dst = NULL* the flipping is done inplace.

flip_mode
: Specifies how to flip the array.
flip_mode = 0 means flipping around x-axis, flip_mode > 0 (e.g. 1) means flipping around y-axis and flip_mode < 0 (e.g. -1) means flipping around both axises. See also the discussion below for the formulas

The function cvFlip [p 62] flips the array in one of different 3 ways (row and column indices are 0-based):

```
B(i,j)=A(rows(A)-i-1,j) if flip_mode = 0

B(i,j)=A(i,cols(A)-j-1) if flip_mode > 0

B(i,j)=A(rows(A)-i-1,cols(A)-j-1) if flip_mode < 0
```

The typical scenaria of the function use are:

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

CvtPixToPlane

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

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

src
: Source array.

dst0...dst3
: Destination channels.

The function cvCvtPixToPlane [p 62] divides a multi-channel array into separate single-channel arrays. Two modes are available for the operation. If the source array has N channels then if the first N destination channels are not NULL, all they are extracted from the source array, otherwise if only a single destination channel of the first N is not NULL, this particular channel is extracted, otherwise an error is raised. Rest of destination channels (beyond the first N) must always be NULL. For IplImage cvCopy [p 58] with COI set can be also used to extract a single channel from the image.

CvtPlaneToPix

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

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

src0... src3
: Input channels.

dst
: Destination array.

The function cvCvtPlaneToPix [p 63] is the opposite to the previous. If the destination array has N channels then if the first N input channels are not NULL, all they are copied to the destination array, otherwise if only a single source channel of the first N is not NULL, this particular channel is copied into the destination array, otherwise an error is raised. Rest of source channels (beyond the first N) must always be NULL. For IplImage cvCopy [p 58] with COI set can be also used to insert a single channel into the image.

ConvertScale

Converts one array to another with optional linear transformation

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

#define cvCvtScale cvConvertScale
#define cvScale  cvConvertScale
#define cvConvert( A, B )  cvConvertScale( (A), (B), 1, 0 )
```

A
: Source array.

B
: Destination array.

scale
: Scale factor.

shift
: Value added to the scaled source array elements.

The function cvConvertScale [p 63] has several different purposes and thus has several synonyms. It copies one array to another with optional scaling, which is performed first, and/or optional type conversion, performed after:

```
B(I)=A(I)*scale + (shift,shift,...)
```

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

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

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

ConvertScaleAbs

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

```
void cvConvertScaleAbs( const CvArr* A, CvArr* B, double scale=1, double shift=0 );
#define cvCvtScaleAbs cvConvertScaleAbs
```

A
: Source array.

B
: Destination array (should have 8u depth).

scale
: ScaleAbs factor.

shift
: Value added to the scaled source array elements.

The function cvConvertScaleAbs [p 64] is similar to the previous one, but it stores absolute values of the conversion results:

```
B(I)=abs(A(I)*scale + (shift,shift,...))
```

The function supports only destination arrays of 8u (8-bit unsigned integers) type, for other types the function can be emulated by combination of cvConvertScale [p 63] and cvAbs [p 76] functions.

Add

Computes per-element sum of two arrays

```
void cvAdd( const CvArr* A, const CvArr* B, CvArr* C, const CvArr* mask=0);
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

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

The function cvAdd [p 64] adds one array to another one:

```
C(I)=A(I)+B(I) if mask(I)!=0
```

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

AddS

Computes sum of array and scalar

```
void cvAddS( const CvArr* A, CvScalar S, CvArr* C, const CvArr* mask=0 );
```

A
: The source array.

S
: Added scalar.

C
: The destination array.

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

The function cvAddS [p 65] adds scalar *S* to every element in the source array *A* and stores the result in *C*

```
C(I)=A(I)+S if mask(I)!=0
```

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

Sub

Computes per-element difference of two arrays

```
void cvSub( const CvArr* A, const CvArr* B, CvArr* C, const CvArr* mask=0);
```

A
: The first source array.

B
The second source array.
C
The destination array.
mask
Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function cvSub [p 65] subtracts one array from another one:

```
C(I)=A(I)-B(I) if mask(I)!=0
```

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

SubS

Computes difference of array and scalar

```
void cvSubS( const CvArr* A, CvScalar S, CvArr* C, const CvArr* mask=0 );
```

A
The source array.
S
Subed scalar.
C
The destination array.
mask
Operation mask, 8-bit single channel array; specifies elements of destination array to be changed.

The function cvSubS [p 66] subtracts a scalar from every element of the source array:

```
C(I)=A(I)+S if mask(I)!=0
```

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

SubRS

Computes difference of scalar and array

```
void cvSubRS( const CvArr* A, CvScalar S, CvArr* C, const CvArr* mask=0 );
```

A
The first source array.
S
Scalar to subtract from.
C
The destination array.

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

The function cvSubRS [p 66] subtracts every element of source array from a scalar:

```
C(I)=S-A(I) if mask(I)!=0
```

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

Mul

Calculates per-element product of two arrays

```
void cvMul( const CvArr* A, const CvArr* B, CvArr* C, double scale=1 );
```

A
The first source array.
B
The second source array.
C
The destination array.
scale
Optional scale factor

The function cvMul [p 67] calculates per-element product of two arrays:

```
C(I)=scale•A(I)•B(I)
```

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

Div

Performs per-element division of two arrays

```
void cvDiv( const CvArr* A, const CvArr* B, CvArr* C, double scale=1 );
```

A
The first source array. If the pointer is NULL, the array is assumed to be all 1’s.
B
The second source array.
C
The destination array.
scale
Optional scale factor

The function cvDiv [p 67] divides one array by another:

```
C(I)=scale•A(I)/B(I), if A!=NULL
C(I)=scale/B(I),      if A=NULL
```

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

And

Calculates per-element bit-wise conjunction of two arrays

```
void cvAnd( const CvArr* A, const CvArr* B, CvArr* C, const CvArr* mask=0 );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

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

The function cvAnd [p 68] calculates per-element bit-wise logical conjunction of two arrays:

```
C(I)=A(I)&B(I) if mask(I)!=0
```

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

---

AndS

Calculates per-element bit-wise conjunction of array and scalar

```
void cvAndS( const CvArr* A, CvScalar S, CvArr* C, const CvArr* mask=0 );
```

A
: The source array.

S
: Scalar to use in the operation.

C
: The destination array.

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

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

```
C(I)=A(I)&S if mask(I)!=0
```

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

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

```
float a[] = { -1, 2, -3, 4, -5, 6, -7, 8, -9 };
CvMat A = cvMat( 3, 3, CV_32F, &a );
int i, abs_mask = 0x7fffffff;
cvAndS( &A, cvRealScalar(*(float*)&abs_mask), &A, 0 );
for( i = 0; i < 9; i++ )
    printf("%.1f ", a[i] );
```

The code should print:

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

Or

Calculates per-element bit-wise disjunction of two arrays

```
void cvOr( const CvArr* A, const CvArr* B, CvArr* C, const CvArr* mask=0 );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

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

The function cvOr [p 69] calculates per-element bit-wise disjunction of two arrays:

```
C(I)=A(I)|B(I)
```

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

---

OrS

Calculates per-element bit-wise disjunction of array and scalar

```
void cvOrS( const CvArr* A, CvScalar S, CvArr* C, const CvArr* mask=0 );
```

A
: The source array.

S
: Scalar to use in the operation.

C
: The destination array.

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

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

```
C(I)=A(I)|S if mask(I)!=0
```

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

---

Xor

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

```
void cvXor( const CvArr* A, const CvArr* B, CvArr* C, const CvArr* mask=0 );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

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

The function cvXor [p 70] calculates per-element bit-wise logical conjunction of two arrays:

```
C(I)=A(I)^B(I) if mask(I)!=0
```

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

---

XorS

Performs per-element bit-wise "exclusive or" operation on array and scalar

```
void cvXorS( const CvArr* A, CvScalar S, CvArr* C, const CvArr* mask=0 );
```

A
: The source array.

S
: Scalar to use in the operation.

C
: The destination array.

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

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

```
C(I)=A(I)^S if mask(I)!=0
```

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

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

```
float a[] = { 1, 0, 0, 1, -1, 0, 0, -1 }; /* 1, j, -1, -j */
CvMat A = cvMat( 4, 1, CV_32FC2, &a );
int i, neg_mask = 0x80000000;
cvXorS( &A, cvScalar( 0, *(float*)&neg_mask, 0, 0 ), &A, 0 );
for( i = 0; i < 4; i++ )
    printf("(%.1f, %.1f) ", a[i*2], a[i*2+1] );
```

The code should print:

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

Not

Performs per-element bit-wise inversion of array elements

```
void cvNot( const CvArr* A, CvArr* C );
```

A
: The source array.

C
: The destination array.

The function Not inverses every bit of every array element:

```
C(I)=~A(I)
```

Cmp

Performs per-element comparison of two arrays

```
void cvCmp( const CvArr* A, const CvArr* B, CvArr* C, int cmp_op );
```

A
: The first source array.

B
: The second source array. Both source array must have a single channel.

C
: The destination array, must have 8u or 8s type.

cmp_op
: The flag specifying the relation between the elements to be checked:
CV_CMP_EQ - A(I) "equal to" B(I)
CV_CMP_GT - A(I) "greater than" B(I)
CV_CMP_GE - A(I) "greater or equal" B(I)
CV_CMP_LT - A(I) "less than" B(I)
CV_CMP_GE - A(I) "less or equal" B(I)
CV_CMP_NE - A(I) "not equal to" B(I)

The function cvCmp [p 72] compares the corresponding elements of two arrays and fills the destination mask array:

```
C(I)=A(I) op B(I),
```

where *op* is '=', '>', '>=', '<', '<=' or '!='.

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

CmpS

Performs per-element comparison of array and scalar

```
void cvCmpS( const CvArr* A, double S, CvArr* C, int cmp_op );
```

A
: The source array, must have a single channel.

C
: The destination array, must have 8u or 8s type.

cmp_op
: The flag specifying the relation between the elements to be checked:
CV_CMP_EQ - A(I) "equal to" S
CV_CMP_GT - A(I) "greater than" S
CV_CMP_GE - A(I) "greater or equal" S
CV_CMP_LT - A(I) "less than" S

CV_CMP_GE - A(I) "less or equal" S
CV_CMP_NE - A(I) "not equal" S

The function cvCmpS [p 72] compares the corresponding elements of array and scalar and fills the destination mask array:

```
C(I)=A(I) op S,
```

where *op* is '=', '>', '>=', '<', '<=' or '!='.

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

InRange

Checks that array elements lie between elements of two other arrays

```
void cvInRange( const CvArr* A, const CvArr* L, const CvArr* U, CvArr* C );
```

A
The first source array.
L
The inclusive lower boundary array.
U
The exclusive upper boundary array.
C
The destination array, must have 8u or 8s type.

The function cvInRange [p 73] does the range check for every element of the input array:

$$C(I)=L(I)_0 <= A(I)_0 < U(I)_0$$

for single-channel arrays,

$$C(I)=L(I)_0 <= A(I)_0 < U(I)_0 \&\& L(I)_1 <= A(I)_1 < U(I)_1$$

for two-channel arrays etc.

C(I) is set to 0xff (all '1'-bits) if A(I) is within the range and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

InRangeS

Checks that array elements lie between two scalars

```
void cvInRangeS( const CvArr* A, CvScalar SL, CvScalar SU, CvArr* D );
```

A
: The first source array.

SL
: The inclusive lower boundary.

SU
: The exclusive upper boundary.

C
: The destination array, must have 8u or 8s type.

The function cvInRangeS [p 73] does the range check for every element of the input array:

$$C(I)=SL_0 <= A(I)_0 < SU_0$$

for a single-channel array,

$$C(I)=SL_0 <= A(I)_0 < SU_0 \&\& SL_1 <= A(I)_1 < SU_1$$

for a two-channel array etc.

C(I) is set to 0xff (all '1'-bits) if A(I) is within the range and 0 otherwise. All the arrays must have the same size (or ROI size)

Max

Finds per-element maximum of two arrays

```
void cvMax( const CvArr* A, const CvArr* B, CvArr* C );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

The function cvMax [p 74] calculates per-element maximum of two arrays:

```
C(I)=max(A(I), B(I))
```

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

MaxS

Finds per-element maximum of array and scalar

```
void cvMaxS( const CvArr* A, const CvArr* B, CvArr* C );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

The function cvMaxS [p 75] calculates per-element maximum of array and scalar:

```
C(I)=max(A(I), S)
```

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

Min

Finds per-element minimum of two arrays

```
void cvMin( const CvArr* A, const CvArr* B, CvArr* C );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

The function cvMin [p 75] calculates per-element minimum of two arrays:

```
C(I)=min(A(I),B(I))
```

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

MinS

Finds per-element minimum of array and scalar

```
void cvMinS( const CvArr* A, const CvArr* B, CvArr* C );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

The function cvMinS [p 75] calculates minimum of array and scalar:

```
C(I)=min(A(I), S)
```

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

AbsDiff

Calculates absolute difference between two arrays

```
void cvAbsDiff( const CvArr* A, const CvArr* B, CvArr* C );
```

A
: The first source array.

B
: The second source array.

C
: The destination array.

The function cvAbsDiff [p 76] calculates absolute difference between two arrays.

$C(I)_c = abs(A(I)_c - B(I)_c)$.

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

AbsDiffS

Calculates absolute difference between array and scalar

```
void cvAbsDiffS( const CvArr* A, CvArr* C, CvScalar S );
#define cvAbs(A, C) cvAbsDiffS(A, C, cvScalarAll(0))
```

A
: The source array.

C
: The destination array.

S
: The scalar.

The function cvAbsDiffS [p 76] calculates absolute difference between array and scalar.

$C(I)_c = abs(A(I)_c - S_c)$.

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

Array statistics

CountNonZero

Counts non-zero array elements

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

A
The array, must be single-channel array or multi-channel image with COI set.

The function cvCountNonZero [p 77] returns the number of non-zero elements in A:

$result = sum_I A(I)!=0$

In case of *IplImage* both ROI and COI are supported.

Sum

Summarizes array elements

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

A
The array.

The function cvSum [p 77] calculates sum *S* of array elements, independently for each channel:

$S_c = sum_I A(I)_c$

If the array is *IplImage* and COI is set, the function processes the selected channel only and stores the sum to the first scalar component (S_0).

Avg

Calculates average (mean) of array elements

```
CvScalar cvAvg( const CvArr* A, const CvArr* mask=0 );
```

A
The array.
mask
The optional operation mask.

The function cvAvg [p 77] calculates the average value *M* of array elements, independently for each channel:

$$N = \sum_I \texttt{mask}(I) \neq 0$$

$$M_c = 1/N \cdot \sum_I \texttt{A}(I)_c$$

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

AvgSdv

Calculates average (mean) of array elements

```
void cvAvgSdv( const CvArr* A, CvScalar* _M, CvScalar* _S, const CvArr* mask=0 );
```

A
The array.
_M
Pointer to the mean value, may be NULL if it is not needed.
_S
Pointer to the standard deviation.
mask
The optional operation mask.

The function cvAvgSdv [p 78] calculates the average value *M*=*_*M* and standard deviation *S*=*_*S* of array elements, independently for each channel:

$$N = \sum_I \texttt{mask}(I) \neq 0$$

$$M_c = 1/N \cdot \sum_I \texttt{A}(I)_c$$

$$S_c = \texttt{sqrt}(1/N \cdot \sum_I (\texttt{A}(I)_c - M_c)^2)$$

If the array is *IplImage* and COI is set, the function processes the selected channel only and stores the average and standard deviation to the first compoenents of output scalars (M_0 and S_0).

MinMaxLoc

Finds global minimum and maximum in array or subarray

```
void cvMinMaxLoc( const CvArr* A, double* minVal, double* maxVal,
                  CvPoint* minLoc, CvPoint* maxLoc, const CvArr* mask=0 );
```

A
: The source array, single-channel or multi-channel with COI set.

minVal
: Pointer to returned minimum value.

maxVal
: Pointer to returned maximum value.

minLoc
: Pointer to returned minimum location.

maxLoc
: Pointer to returned maximum location.

mask
: The optional mask that is used to select a subarray.

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

Norm

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

```
double cvNorm( const CvArr* A, const CvArr* B, int normType, const CvArr* mask=0 );
```

A
: The first source image.

B
: The second source image. If it is NULL, the absolute norm of *A* is calculated, otherwise absolute or relative norm of *A-B* is calculated.

normType
: Type of norm, see the discussion.

mask
: The optional operation mask.

The function cvNorm [p 79] calculates the absolute norm of *A* if *B* is NULL:

$norm = ||A||_C = \max_I abs(A(I)),$ if *normType* = CV_C

$norm = ||A||_{L1} = \sum_I abs(A(I)),$ if *normType* = CV_L1

$norm = ||A||_{L2} = \sqrt{\sum_I A(I)^2},$ if *normType* = CV_L2

And the function calculates absolute or relative difference norm if *B* is not NULL:

$norm = ||A-B||_C = \max_I abs(A(I)-B(I)),$ if *normType* = CV_C

$norm = ||A-B||_{L1} = \sum_I abs(A(I)-B(I)),$ if *normType* = CV_L1

$norm = ||A-B||_{L2} = \sqrt{\sum_I (A(I)-B(I))^2},$ if *normType* = CV_L2

or

$norm = ||A-B||_C / ||B||_C,$ if *normType* = CV_RELATIVE_C

$norm = ||A-B||_{L1} / ||B||_{L1},$ if *normType* = CV_RELATIVE_L1

$norm = ||A-B||_{L2} / ||B||_{L2},$ if *normType* = CV_RELATIVE_L2

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

Matrix Operations, Linear Algebra and Math Functions

SetIdentity

Initializes scaled identity matrix

```
void cvSetIdenity( CvArr* A, CvScalar S );
```

A
: The matrix to initialize (not necesserily square).

S
: The value to assign to the diagonal elements.

The function cvSetIdentity [p 80] initializes scaled identity matrix:

```
A(i,j)=S if i=j,
       0 otherwise
```

DotProduct

Calculates dot product of two arrays in Euclidian metrics

```
double cvDotProduct (const CvArr* A, const CvArr* B );
```

A
: The first source array.

B
: The second source array.

The function cvDotProduct [p 81] calculates and returns the Euclidean dot product of two arrays.

$A \bullet B = \sum_I (A(I)*B(I))$

In case of multiple channel arrays the results for all channels are accumulated. In particular, it gives a correct result for complex matrices. The function can process multi-dimensional arrays row by row.

CrossProduct

Calculates cross product of two 3D vectors

```
void cvCrossProduct( const CvArr* A, const CvArr* B, CvArr* C );
```

A
: The first source vector.

B
: The second source vector.

C
: The destination vector.

The function cvCrossProduct [p 81] calculates the cross product of two 3D vectors:

$C = A \times B, (C_1 = A_2 B_3 - A_3 B_2 , C_2 = A_3 B_1 - A_1 B_3 , C_3 = A_1 B_2 - A_2 B_1).$

ScaleAdd

Calculates sum of scaled array and another array

```
void cvScaleAdd( const CvArr* A, CvScalar S, const CvArr* B, CvArr* C );
#define cvMulAddS cvScaleAdd
```

A
: The first source array.

S
: Scale factor for the first array.

B
The second source array.
C
The destination array

The function cvScaleAdd [p 81] calculates sum of scaled array and another array:

```
C(I)=A(I)*S + B(I)
```

All array parameters should be of the same size and the same size

---

MatMulAdd

Calculates shifted matrix product

```
void cvMatMulAdd( const CvArr* A, const CvArr* B, const CvArr* C, CvArr* D );
#define cvMatMul(A, B, D) cvMatMulAdd(A, B, 0, D)
```

A
The first source array.
B
The second source array.
C
The third source array (shift). Can be NULL, if there is no shift.
D
The destination array.

The function cvMatMulAdd [p 82] calculates matrix product of two matrices and adds the third matrix to the product:

```
D = A*B + C   or  D(i,j)=sum_k (A(i,k)*B(k,j)) + C(i,j)
```

All the matrices should be of the same type and the coordinated sizes. Only real or complex floating-point matrices are supported

---

GEMM

Performs generalized matrix multiplication

```
void  cvGEMM( const CvArr* A, const CvArr* B, double alpha,
              const CvArr* C, double beta, CvArr* D, int tABC=0 );
#define cvMatMulAddEx cvGEMM
```

A
The first source array.
B
The second source array.

C
The third source array (shift). Can be NULL, if there is no shift.
D
The destination array.
tABC
The operation flags that can be 0 or combination of the following:
CV_GEMM_A_T - transpose A
CV_GEMM_B_T - transpose B
CV_GEMM_C_T - transpose B
for example, CV_GEMM_A_T+CV_GEMM_C_T corresponds to

$alpha*A^T*B + beta*C^T$

The function cvGEMM [p 82] performs generalized matrix multiplication:

$D = alpha*op(A)*op(B) + beta*op(C), where op(X) is X or X^T$

All the matrices should be of the same type and the coordinated sizes. Only real or complex floating-point matrices are supported

---

MatMulAddS

Performs matrix transform on every element of array

```
void cvMatMulAddS( const CvArr* A, CvArr* C, const CvArr* M, const CvArr* V=0 );
```

A
The first source array.
C
The destination array.
M
Transformation matrix.
V
Optional shift.

The function cvMatMulAddS [p 83] performs matrix transform on every element of array *A* and stores the result in *C*:

$C(i,j)=M*A(i,j) + V \quad or \quad C(i,j)(k)=sum_l(M(k,l)*A(i,j)(l)) + V(k)$

That is every element of *N*-channel array *A* is considered as *N*-element vector, which is transformed using matrix *N*×*N* matrix *M* and shift vector *V*. There is an option to code *V* into *A*. In this case *A* should be *N*×*N+1* matrix and the right-most column is used as the shift vector.

Both source and destination arrays should be of the same size or selected ROI size and of the same type. *M* and *V* should be real floating-point matrices. The function can be used for geometrical transforms of point sets and linear color transformations.

MulTransposed

Calculates product of array and transposed array

```
void cvMulTransposed( const CvArr* A, CvArr* C, int order );
```

A
: The source matrix.

C
: The destination matrix.

order
: Order of multipliers.

The function cvMulTransposed [p 84] calculates the product of A and its transposition.

The function evaluates

$C=A*A^T$

if order=0, and

$C=A^T*A$

otherwise

Trace

Returns trace of matrix

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

A
: The source matrix.

The function cvTrace [p 84] returns sum of diagonal elements of the matrix *A*.

$tr(A)=sum_i A(i,i)$

Transpose

Transposes matrix

```
void cvTranspose( const CvArr* A, CvArr* B );
#define cvT cvTranspose
```

A
The source matrix.
B
The destination matrix.

The function cvTranspose [p 84] transposes matrix *A*:

```
B(i,j)=A(j,i)
```

Note that no complex conjugation is done in case of complex matrix. Conjugation should be done separately: look at the sample code in cvXorS [p 70] for example

---

Det

Returns determinant of matrix

```
CvScalar cvDet( const CvArr* A );
```

A
The source matrix.

The function cvDet [p 85] returns determinant of the square matrix *A*. The direct method is used for small matrices and Gaussian elimination is used for larger matrices

---

Invert

Finds inverse or pseudo-inverse of matrix

```
double cvInvert( const CvArr* A, CvArr* B, int method );
#define cvInv cvInvert
```

A
The source matrix.
B
The destination matrix.
method
Inversion method:
CV_LU - Gaussian elimination with optimal pivot element chose CV_SVD - Singular decomposition method

The function cvInvert [p 85] inverts matrix *A* and stores the result in *B*

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

In case of *SVD* method the function returns the inversed condition number of *A* (ratio of the smallest singular value to the largest singular value) and 0 if *A* is all zeros. This method calculates a pseudo-inverse matrix if *A* is singular

---

Solve

Solves linear system or least-squares problem

```
int cvSolve( const CvArr* A, const CvArr* B, CvArr* X, int method );
#define cvInv cvSolve
```

A
: The source matrix.

B
: The right-hand part of the linear system.

method
: The solution (matrix inversion) method:
CV_LU - Gaussian elimination with optimal pivot element chose CV_SVD - Singular decomposition method

The function cvSolve [p 86] solves linear system or least-squares problem:

$X^* = \arg\min_X ||A*X-B||$

If *CV_LU* method is used, the function returns 1 if *A* is non-singular and 0 otherwise, in the latter case *X* is not valid

---

SVD

Performs singular value decomposition of real floating-point matrix

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

A
: Source *M*×*N* matrix.

W
: Resulting singular value matrix (*M*×*N* or *N*×*N*) or vector (*N*×*1*).

U
: Optional left orthogonal matrix (*M*×*M* or *M*×*N*). If CV_SVD_U_T is specified, the number of rows and columns in the sentence above should be swapped.

V
: Optional right orthogonal matrix (*N*×*N*)

flags
: Operation flags; can be 0 or combination of the following:
  - *CV_SVD_MODIFY_A* enables modification of matrix *A* during the operation. It speeds up the processing.

- *CV_SVD_U_T* means that the tranposed matrix *U* is returned. Specifying the flag speeds up the processing.
- *CV_SVD_V_T* means that the tranposed matrix *V* is returned. Specifying the flag speeds up the processing.

The function cvSVD [p 86] decomposes matrix *A* into a product of a diagonal matrix and two orthogonal matrices:

$A=U*W*V^T$

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

SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix *A* is square, symmetric and positively defined matrix, for example, when it is a covariation matrix. *W* in this case will be a vector of eigen values, and *U=V* is matrix of eigen vectors (thus, only one of *U* or *V* needs to be calculated if the eigen vectors are required)
- accurate solution of poor-conditioned linear systems
- least-squares solution of overdetermined linear systems. This and previous is done by cvSolve [p 86] function with *CV_SVD* method
- accurate calculation of different matrix characteristics such as rank (number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), determinant (absolute value of determinant is equal to the product of singular values). All the things listed in this item do not require calculation of *U* and *V* matrices.

SVBkSb

Performs singular value back substitution

```
void  cvSVBkSb( const CvArr* W, const CvArr* U, const CvArr* V,
                const CvArr* B, CvArr* X, int flags );
```

W
: Matrix or vector of singular values.

U
: Left orthogonal matrix (tranposed, perhaps)

V
: Right orthogonal matrix (tranposed, perhaps)

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

X
The destination matrix: result of back substitution.
flags
Operation flags, should match exactly to the *flags* passed to cvSVD [p 86] .

The function cvSVBkSb [p 87] calculates back substitution for decomposed matrix *A* (see cvSVD [p 86] description) and matrix *B*:

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

Where

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

And *epsilon* is a small number - $\approx 10^{-6}$ or $\approx 10^{-15}$ depending on the matrices element type.

This function together with cvSVD [p 86] is used inside cvInvert [p 85] and cvSolve [p 86] , and the possible reason to use these (svd & bksb) "low-level" function is to avoid temporary matrices allocation inside the high-level counterparts (inv & solve).

EigenVV

Computes eigenvalues and eigenvectors of symmetric matrix

```
void cvEigenVV( CvArr* A, CvArr* evects, CvArr* evals, double eps );
```

A
The source symmetric square matrix. It is modified during the processing.
evects
The output matrix of eigenvectors, stored as a subsequent rows.
evals
The output vector of eigenvalues, stored in the descenting order (order of eigenvalues and eigenvectors is syncronized, of course).
eps
Accuracy of diagonalization (typically, DBL_EPSILON=$\approx 10^{-15}$ is enough).

The function cvEigenVV [p 88] computes the eigenvalues and eigenvectors of the matrix *A*:

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

The contents of matrix *A* is destroyed by the function.

Currently the function is slower than cvSVD [p 86] yet less accurate, so if *A* is known to be positively-defined (for example, it is a covariation matrix), it is recommended to use cvSVD [p 86] to find eigenvalues and eigenvectors of *A*, especially if eigenvectors are not required.

PerspectiveTransform

Performs perspective matrix transform on 3D vector array

```
void cvPerspectiveTransform( const CvArr* A, CvArr* B, const CvArr* M );
```

A
: The source three-channel floating-point array.

B
: The destination three-channel floating-point array.

M
: 4 × 4 transformation matrix.

The function cvPerspectiveTransform [p 89] transforms every element of *A* considering it a 3D vector as:

```
(x, y, z) -> (x'/w, y'/w, z'/w),

where
(x', y', z', w') = M*(x, y, z, 1)

and w = 1/w' if w'!=0,
          1  otherwise
```

CalcCovarMatrix

Calculates covariation matrix out of the set of arrays

```
void cvCalcCovarMatrix( CvArr** Vs, CvArr* M, CvArr* A );
```

Vs
: The set of input arrays. All the arrays must have the same type and the same size.

M
: The output covariation matrix that should be floating-point and square. Number of arrays is implicitly assumed to be equal to number of the matrix rows/columns.

A
: The output array that is set to the average of the input arrays.

The function cvCalcCovarMatrix [p 89] calculates the covariation matrix and average array out of the set of input arrays:

$A(I)=\sum_k Vs^{(k)}(I)$

$M(i,j)=(Vs^{(i)}-A)\bullet(Vs^{(j)}-A)$

Where the upper index in parentheses means the particular array from the set and "•" means dot product. The covariation matrix may be used then (after inversion) in cvMahalonobis [p 90] function to measure a distance between vectors, to find eigen objects via cvSVD [p 86] etc.

Mahalonobis

Calculates Mahalonobis distance between vectors

```
double cvMahalonobis( const CvArr* A, const CvArr* B, CvArr* T );
```

A
: The first 1D source vector.

B
: The second 1D source vector.

T
: The inverse covariation matrix.

The function cvMahalonobis [p 90] calculates the weighted distance between two vectors and returns it:

$$d(A,B)=\sqrt{\sum_{i,j} T(i,j)\cdot(A(i)-B(i))\cdot(A(j)-B(j))}$$

The covariation matrix may be calculated using cvCalcCovarMatrix [p 89] function and further inverted using cvInvert [p 85] function (CV_SVD method is the preffered one, because the matrix might be singular).

CartToPolar

Calculates magnitude and/or angle of 2d vectors

```
void cvCartToPolar( const CvArr* X, const CvArr* Y, CvArr* M, CvArr* A,
                    int angle_in_degrees=0 );
```

X
: The array of x-coordinates

Y
: The array of y-coordinates

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

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

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

The function cvCartToPolar [p 90] calculates either magnitude, angle, or both of every vector *(X(I),Y(I))*:

$$M(I)=\sqrt{X(I)^2+Y(I)^2},$$
$$A(I)=\operatorname{atan}(Y(I)/X(I))$$

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

PolarToCart

Calculates cartesian coordinates of 2d vectors represented in polar form

```
void cvPolarToCart( const CvArr* M, const CvArr* A, CvArr* X, CvArr* Y,
                    int angle_in_degrees=0 );
```

M
: The array of magnitudes. If it is NULL, the magnitudes are assumed all 1's.

A
: The array of angles, whether in radians or degrees.

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

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

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

The function cvPolarToCart [p 91] calculates either x-coodinate, y-coordinate or both of every vector *M(I)*exp(A(I)*j)*:

```
X(I)=M(I)*cos(A(I)),
Y(I)=M(I)*sin(A(I))
```

Pow

Raises every array element to power

```
void cvPow( const CvArr* X, CvArr* Y, double p );
```

X
: The source array.

Y
: The destination array, should be the same type as the source.

p
: The exponent of power.

The function cvPow [p 91] raises every element of input array to *p*:

$Y(I)=X(I)^p$, if *p* is integer
$Y(I)=abs(X(I))^p$, otherwise

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

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

For some values of *power*, such as integer values, 0.5 and -0.5, an optimized algorithm is used.

Exp

Calculates exponent of every array element

```
void cvExp( const CvArr* X, CvArr* Y );
```

X
: The source array.

Y
: The destination array, it should have *double* type or the same type as the source.

The function cvExp [p 92] calculates exponent of every element of input array:

```
Y(I)=exp(X(I))
```

Maximum relative error is ≈7e-6. Currently, the function converts denormalized values to zeros on output.

Log

Calculates natural logarithm of every array element absolute value

```
void cvLog( const CvArr* X, CvArr* Y );
```

X
: The source array.

Y
: The destination array, it should have *double* type or the same type as the source.

The function cvLog [p 92] calculates natural logarithm of absolute value of every element of input array:

```
Y(I)=log(abs(X(I))), X(I)!=0
Y(I)=C,  X(I)=0
```

Where *C* is large negative number (≈-700 in the current implementation)

CheckArr

Checks every element of input array for invalid values

```
int  cvCheckArr( const CvArr* X, int flags=0,
                 double minVal=0, double maxVal=0);
#define cvCheckArray cvCheckArr
```

X
: The array to check.

flags
: The operation flags, 0 or combination of:
CV_CHECK_RANGE - if set, the function checks that every value of array is within [minVal,maxVal) range, otherwise it just checks that every element is neigther NaN nor ±∞.
CV_CHECK_QUIET - if set, the function does not raises an error if an element is invalid or out of range

minVal
: The inclusive lower boundary of valid values range. It is used only if *CV_CHECK_RANGE* is set.

maxVal
: The exclusive upper boundary of valid values range. It is used only if *CV_CHECK_RANGE* is set.

The function cvCheckArr [p 93] checks that every array element is neither NaN nor ±∞. If *CV_CHECK_RANGE* is set, it also checks that every element is greater than or equal to *minVal* and less than *maxVal*. The function returns nonzero if the check succeeded, i.e. all elements are valid and within the range, and zero otherwise. In the latter case if *CV_CHECK_QUIET* flag is not set, the function raiser runtime error.

RandInit

Initializes random number generator state

```
void cvRandInit( CvRandState* state, double param1, double param2, int seed,
                 int distType=CV_RAND_UNI );
```

state
: Pointer to the initialized random number generator state structure.

param1
: The first distribution parameter. In case of uniform distribution it is the inclusive lower boundary of random numbers range. In case of normal distribution it is the standard deviation of random numbers.

param2
: The second distribution parameter. In case of uniform distribution it is the exclusive upper boundary of random numbers range. In case of normal distribution it is the mean value of random numbers.

seed
: Initial 32-bit value to start a random sequence.

distType
: Distribution type:
CV_RAND_UNI - uniform distribution

CV_RAND_NORMAL - normal or Gaussian distribution

The function cvRandInit [p 93] initializes the *state* structure that is used for generating uniformly distributed numbers in the range *[param1, param2)* or normally distributed numbers with *param1* mean and *param2* standard deviation. The parameters are set for all the dimensions simultaneously - resemble that RNG has separate parameters for each of 4 dimensions. A multiply-with-carry generator is used.

RandSetRange

Changes the range of generated random numbers without touching RNG state

```
void cvRandSetRange( CvRandState* state, double param1, double param2, int index=-1 );
```

state
: State of random number generator (RNG).

param1
: New lower boundary/deviation of generated numbers.

param2
: New upper boundary/mean value of generated numbers.

index
: The 0-based index of dimension/channel for which the parameter are changed, -1 means changing the parameters for all dimensions.

The function cvRandSetRange [p 94] changes the range of generated random numbers without reinitializing RNG state. It is useful if a few arrays of different types need to initialized with random numbers within a loop. Alternatively, you may have a separate generator for each array, but then you should provide several uncorrelated initialization seeds - one per each generator.

Rand

Fills array with random numbers and updates the RNG state

```
void cvRand( CvRandState* state, CvArr* arr );
```

state
: RNG state initialized by *RandInit* and, optionally, customized by *RandSetRange*.

arr
: The destination array.

The function cvRand [p 94] fills the destination array with uniformly or normally distributed random numbers within the pre-set range and updates RNG state. In the sample below this and two functions above are used to put a few normally distributed floating-point numbers to random locations within a 2d array

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

cvRandInit( &rng_state,
            0, 1, /* use dummy parameters now and adjust them further */
            0xffffffff /* just use a fixed seed here */,
            CV_RAND_UNI /* specify uniform type */ );

/* customize the RNG to use it for initialiazing locations:
   the 0-th dimension is used for x’s and the 1st - for y’s */
cvRandSetRange( &rng_state, 0, size.width, 0 );
cvRandSetRange( &rng_state, 0, size.height, 1 );

/* initialize the locations */
cvRand( &rng_state, locations );

/* modify RNG to make it produce normally distributed values */
rng_state.disttype = CV_RAND_NORMAL;
cvRandSetRange( &rng_state,
                30 /* deviation */,
                100 /* average point brightness */,
                -1 /* initialize all the dimensions */ );
/* generate values */
cvRand( &rng_state, values );

/* set the points */
for( i = 0; i < pointCount; i++ )
{
    CvPoint pt = *(CvPoint*)cvPtr1D( locations, i, 0 );
    float value = *(float*)cvPtr1D( values, i, 0 );
    cvSetReal2D( noisy_screen, pt.y, pt.x, value );
}

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

/* cvRandInit does not allocate any memory, so there is no need
  (and no function) to deinitialize it */
```

RandNext

Returns 32-bit unsigned integer and updates RNG

```
unsigned cvRandNext( CvRandState* state );
```

state

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

The function cvRandNext [p 95] returns uniformly-distributed (regardless of the RNG distribution type settings) "plain" integer random number and updates RNG state. It is similar to rand() function from C runtime library, but it always generates 32-bit number whereas rand() returns a number in between 0 and *RAND_MAX* which is 2**16 or 2**32, depending on the platform.

The function is useful for generating scalar random numbers, such as points, patch sizes, table indices etc, where integer numbers of a certain range can be generated using modulo operation and floating-point numbers can be generated by scaling to 0..1 of any other specific range. Here is the example from the previous function discussion rewritten using cvRandNext [p 95] :

```
/* the input and the task is the same as in the previous sample. */
CvRandState rng_state;
int i, pointCount = 1000;
/* ... - no arrays are allocated here */
CvSize size = cvGetSize( noisy_screen );
/* make a buffer for normally distributed numbers to reduce call overhead */
#define bufferSize 16
float normalValueBuffer[bufferSize];
CvMat normalValueMat = cvMat( bufferSize, 1, CV_32F, normalValueBuffer );
int valuesLeft = 0;

/* initialize RNG to produce normally distributed values.
   Coordinates will be uniformly distributed within 0..2**32
   anyway as they are generated using cvRandNext */
cvRandInit( &rng_state,
            100,
            30,
            0xffffffff /* just use a fixed seed here */,
            CV_RAND_NORMAL /* specify uniform type */ );

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

    if( valuesLeft <= 0 )
    {
        /* fulfill the buffer with normally distributed numbers if the buffer is empty */
        cvRand( &rng_state, &normalValueMat );
        valuesLeft = bufferSize;
    }
    cvSetReal2D( noisy_screen, pt.y, pt.x, normalValueBuffer[--valuesLeft]);
}

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

DFT

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

```
#define CV_DXT_INVERSE  1
#define CV_DXT_SCALE 2
#define CV_DXT_INV_SCALE (CV_DXT_SCALE|CV_DXT_INVERSE)
#define CV_DXT_INVERSE_SCALE CV_DXT_INV_SCALE

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

src
: Source array, real or complex.

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

flags
: Transformation flags, 0 or a combination of the following flags:
CV_DXT_INVERSE - perform inverse transform (w/o post-scaling>
CV_DXT_SCALE - divide the result by the number of array elements
For convenience, the constant CV_DXT_FORWARD may be used instead of literal 0.

The function cvDFT [p 97] performs forward or inverse transform of 1D or 2D floating-point array:

Forward Fourier transform of 1D vector of N elements:
$y = F \bullet x$, where $F_{ik} = \exp(-j \bullet Pi \bullet i \bullet k/N)$, $j = \sqrt{-1}$

Inverse Fourier transform of 1D vector of N elements:
$x = F^{-1} \bullet y = F^{T} \bullet y$

Forward Fourier transform of 2D vector of M×N elements:
$Y = F \bullet X \bullet F^{*}$

Inverse Fourier transform of 2D vector of M×N elements:
$X = F^{*} \bullet Y \bullet F$

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

$$
\begin{array}{lllllllll}
Re\,Y_{0,0} & Re\,Y_{0,1} & Im\,Y_{0,1} & Re\,Y_{0,2} & Im\,Y_{0,2} & \ldots & Re\,Y_{0,N/2-1} & Im\,Y_{0,N/2-1} & Re\,Y_{0,N/2} \\
Re\,Y_{1,0} & Re\,Y_{1,1} & Im\,Y_{1,1} & Re\,Y_{1,2} & Im\,Y_{1,2} & \ldots & Re\,Y_{1,N/2-1} & Im\,Y_{1,N/2-1} & Re\,Y_{1,N/2} \\
Im\,Y_{2,0} & Re\,Y_{2,1} & Im\,Y_{2,1} & Re\,Y_{2,2} & Im\,Y_{2,2} & \ldots & Re\,Y_{2,N/2-1} & Im\,Y_{2,N/2-1} & Im\,Y_{2,N/2} \\
\ldots & & & & & & & & \\
Re\,Y_{M/2-1,0} & Re\,Y_{M-3,1} & Im\,Y_{M-3,1} & Re\,Y_{M-3,2} & Im\,Y_{M-3,2} & \ldots & Re\,Y_{M-3,N/2-1} & Im\,Y_{M-3,N/2-1} & Re\,Y_{M-3,N/2} \\
Im\,Y_{M/2-1,0} & Re\,Y_{M-2,1} & Im\,Y_{M-2,1} & Re\,Y_{M-2,2} & Im\,Y_{M-2,2} & \ldots & Re\,Y_{M-2,N/2-1} & Im\,Y_{M-2,N/2-1} & Im\,Y_{M-2,N/2} \\
Re\,Y_{M/2,0} & Re\,Y_{M-1,1} & Im\,Y_{M-1,1} & Re\,Y_{M-1,2} & Im\,Y_{M-1,2} & \ldots & Re\,Y_{M-1,N/2-1} & Im\,Y_{M-1,N/2-1} & Im\,Y_{M-1,N/2}
\end{array}
$$

Note: the last column is present if N is even, the last row is present if M is even.

In case of 1D real transform the result looks like the first row of the above matrix

---

MulCss

Performs per-element multiplication of two Fourier spectrums of two real arrays

```
void cvMulCss( const CvArr* srcA, const CvArr* srcB, CvArr* dst );
```

srcA
: The first source array.

srcB
: The second source array.

dst
: The destination array of the same type and the same size of the sources.

The function cvMulCss [p 98] performs per-element multiplication of the two packed matrices that are produced by forward real Fourier transform (1D or 2D).

Calculating DFT's of two real arrays, then multiplying results by *cvMulCss* and performing inverse DFT on the product is equivalent yet faster way to find cyclic convolution of the two original arrays.

DCT

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

```
#define CV_DXT_INVERSE  1
#define CV_DXT_SCALE 2
#define CV_DXT_INV_SCALE (CV_DXT_SCALE|CV_DXT_INVERSE)
#define CV_DXT_INVERSE_SCALE CV_DXT_INV_SCALE

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

src
: Source array, real 1D or 2D array.

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

flags
: Transformation flags, 0 or a combination of the following flags:
CV_DXT_INVERSE - perform inverse transform
CV_DXT_SCALE - divide the result by the number of array elements
For convenience, the constant CV_DXT_FORWARD may be used instead of literal 0.

The function cvDCT [p 98] performs forward or inverse transform of 1D or 2D floating-point array:

```
Forward Cosine transform of 1D vector of N elements:
y = C•x, where C ik =sqrt((i==0?1:2)/N)•cos(Pi•(2i+1)•k/N), j=sqrt(-1)

Inverse Cosine transform of 1D vector of N elements:
x = C -1 •y = C T •y

Forward Cosine transform of 2D vector of M×N elements:
Y = C•X•C T

Inverse Fourier transform of 2D vector of M×N elements:
X = C T •Y•C
```

Dynamic Data Structures

CvMemStorage

Growing memory storage

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

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

New memory buffer that may be allocated explicitly by cvMemStorageAlloc [p 102] function or implicitly by higher-level functions, such as cvSeqPush [p 107] , cvGraphAddEdge [p 125] etc., *always* starts in the end of the current block if it fits there. After allocation *free_space* is decremented by the size of the allocated buffer plus some padding to keep the proper alignment. When the allocated buffer does not fit into the available part of *top*, the next storage block from the list is taken as *top* and *free_space* is reset to the whole block size prior to the allocation.

If there is no more free blocks, a new block is allocated (or borrowed from parent, see cvCreateChildMemStorage [p 101]) and added to the end of list. Thus, the storage behaves as a stack with *bottom* indicating bottom of the stack and the pair (*top, free_space*) indicating top of the stack. The stack top may be saved via cvSaveMemStoragePos [p 103] , restored via cvRestoreMemStoragePos [p 103] or reset via cvClearStorage [p ??] .

CvMemBlock

Memory storage block

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

The structure CvMemBlock [p 100] represents a single block of memory storage. 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, normally there is no need to access the storage structure fields directly.

CvMemStoragePos

Memory storage position

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

The structure described below stores the position of the stack top that can be saved via cvSaveMemStoragePos [p 103] and restored via cvRestoreMemStoragePos [p 103] .

CreateMemStorage

Creates memory storage

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

blockSize
: Size of the storage blocks in bytes. If it is 0, the block size is set to default value - currently it is ≈64K.

The function cvCreateMemStorage [p 100] creates a memory storage and returns pointer to it. Initially the storage is empty. All fields of the header, except the *block_size*, are set to 0.

CreateChildMemStorage

Creates child memory storage

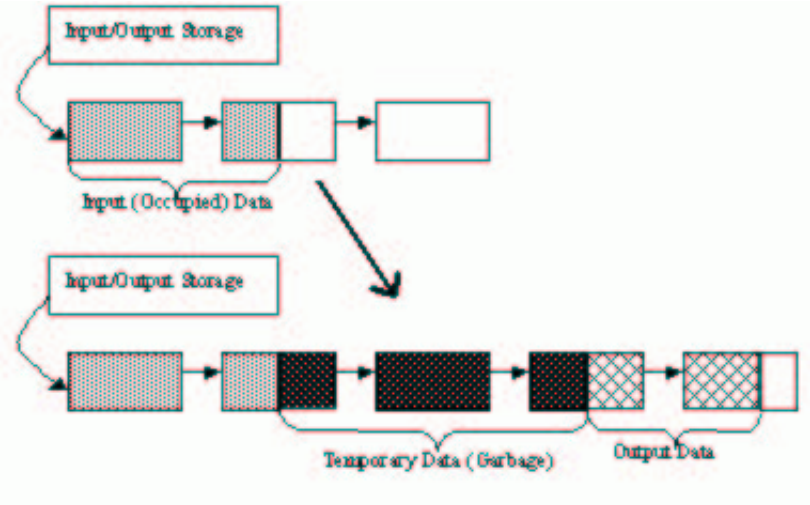
```
CvMemStorage* cvCreateChildMemStorage( CvMemStorage* parent );
```

parent
Parent memory storage.

The function cvCreateChildMemStorage [p 101] creates a child memory storage that is similar to simple memory storage except for the differences in the memory allocation/deallocation mechanism. When a child storage needs a new block to add to the block list, it tries to get this block from the parent. The first unoccupied parent block available is taken and excluded from the parent block list. If no blocks are available, the parent either allocates a block or borrows one from its own parent, if any. In other words, the chain, or a more complex structure, of memory storages where every storage is a child/parent of another is possible. When a child storage is released or even cleared, it returns all blocks to the parent. In other aspects, the child storage is the same as the simple storage.

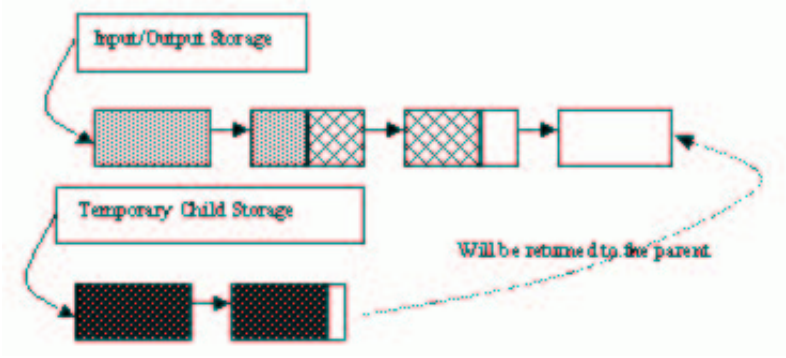
The children storages are useful in the following situation. Imagine that user needs to process dynamical data resided in some storage and put the result back to the same storage. With the simplest approach, when temporary data is resided in the same storage as the input and output data, the storage will look as following after processing:

Dynamic data processing without using child storage



That is, garbage appears in the middle of the storage. However, if one creates a child memory storage in the beginning of the processing, writes temporary data there and releases the child storage in the end, no garbage will appear in the source/destination storage:

Dynamic data processing using a child storage



ReleaseMemStorage

Releases memory storage

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

storage
: Pointer to the released storage.

The function cvReleaseMemStorage [p 102] deallocates all storage memory blocks or returns them to the parent, if any. Then it deallocates the storage header and clears the pointer to the storage. All children of the storage must be released before the parent is released.

ClearMemStorage

Clears memory storage

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

storage
: Memory storage.

The function cvClearMemStorage [p 102] resets the top (free space boundary) of the storage to the very beginning. This function does not deallocate any memory. If the storage has a parent, the function returns all blocks to the parent.

MemStorageAlloc

Allocates memory buffer in the storage

```
void* cvMemStorageAlloc( CvMemStorage* storage, int size );
```

storage
Memory storage.
size
Buffer size.

The function cvMemStorageAlloc [p 102] allocates memory buffer in the storage. The buffer size must not exceed the storage block size, otherwise runtime error is raised. The buffer address is aligned by *CV_STRUCT_ALIGN* (=*sizeof(double)* for the moment) bytes.

SaveMemStoragePos

Saves memory storage position

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

storage
Memory storage.
pos
The output position of the storage top.

The function cvSaveMemStoragePos [p 103] saves the current position of the storage top to the parameter *pos*. The function cvRestoreMemStoragePos [p 103] can further retrieve this position.

RestoreMemStoragePos

Restores memory storage position

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

storage
Memory storage.
pos
New storage top position.

The function cvRestoreMemStoragePos [p 103] restores the position of the storage top from the parameter *pos*. This function and the function cvClearMemStorage [p 102] are the only methods to release memory occupied in memory blocks. Note again that there is no way to free memory in the middle of the occupied part of the storage.

Sequences

CvSeq

Growable sequence of elements

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


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

The structure CvSeq [p 104] is a base for all of OpenCV dynamic data structures.

Such an unusual definition via a helper macro simplifies the extension of the structure CvSeq [p 104] with additional parameters. To extend CvSeq [p 104] the user may define a new structure and put user-defined fields after all CvSeq [p 104] fields that are included via the macro *CV_SEQUENCE_FIELDS()*.

There are two types of sequences - dense and sparse. Base type for dense sequences is CvSeq [p 104] and such sequences are used to represent growable 1d arrays - vectors, stacks, queues, deques. They have no gaps in the middle - if an element is removed from the middle or inserted into the middle of the sequence the elements from the closer end are shifted. Sparse sequences have CvSet [p 119] base class and they are discussed later in more details. They are sequences of nodes each of those may be either occupied or free as indicated by the node flag. Such sequences are used for unordered data structures such as sets of elements, graphs, hash tables etc.

The field *header_size* contains the actual size of the sequence header and should be greater 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 *total* contains the actual number of dense sequence elements and number of allocated nodes in sparse sequence.

The field *flags*contain the particular dynamic type signature (*CV_SEQ_MAGIC_VAL* for dense sequences and *CV_SET_MAGIC_VAL* for sparse sequences) in the highest 16 bits and miscellaneous information about the sequence. The lowest *CV_SEQ_ELTYPE_BITS* bits contain the ID of the element type. Most of sequence processing functions do not use element type but element size stored in *elem_size*. If sequence contains the numeric data of one of CvMat [p 33] type then the element type matches to the corresponding CvMat [p 33] element type, e.g. CV_32SC2 may be used for sequence of 2D points, CV_32FC1 for sequences of floating-point values etc. *CV_SEQ_ELTYPE(seq_header_ptr)* macro retrieves the type of sequence elements. Processing function that work with numerical sequences check that *elem_size* is equal to the calculated from the type element size. Besides CvMat [p 33] compatible types, there are few extra element types defined in cvtypes.h [p ??] header:

Standard Types of Sequence Elements

```
#define CV_SEQ_ELTYPE_POINT          CV_32SC2  /* (x,y) */
#define CV_SEQ_ELTYPE_CODE           CV_8UC1   /* freeman code: 0..7 */
#define CV_SEQ_ELTYPE_GENERIC        0 /* unspecified type of sequence elements */
#define CV_SEQ_ELTYPE_PTR            CV_USRTYPE1 /* =6 */
#define CV_SEQ_ELTYPE_PPOINT         CV_SEQ_ELTYPE_PTR  /* &elem: pointer to element of other sequence */
#define CV_SEQ_ELTYPE_INDEX          CV_32SC1  /* #elem: index of element of some other sequence */
#define CV_SEQ_ELTYPE_GRAPH_EDGE     CV_SEQ_ELTYPE_GENERIC  /* &next_o, &next_d, &vtx_o, &vtx_d */
#define CV_SEQ_ELTYPE_GRAPH_VERTEX   CV_SEQ_ELTYPE_GENERIC  /* first_edge, &(x,y) */
#define CV_SEQ_ELTYPE_TRIAN_ATR      CV_SEQ_ELTYPE_GENERIC  /* vertex of the binary tree   */
#define CV_SEQ_ELTYPE_CONNECTED_COMP CV_SEQ_ELTYPE_GENERIC  /* connected component  */
#define CV_SEQ_ELTYPE_POINT3D        CV_32FC3  /* (x,y,z)  */
```

The next *CV_SEQ_KIND_BITS* bits specify the kind of the sequence:

Standard Kinds of Sequences

```
/* generic (unspecified) kind of sequence */
#define CV_SEQ_KIND_GENERIC     (0 << CV_SEQ_ELTYPE_BITS)

/* dense sequence suntypes */
#define CV_SEQ_KIND_CURVE       (1 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_BIN_TREE    (2 << CV_SEQ_ELTYPE_BITS)

/* sparse sequence (or set) subtypes */
#define CV_SEQ_KIND_GRAPH       (3 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_SUBDIV2D    (4 << CV_SEQ_ELTYPE_BITS)
```

The remaining bits are used to identify different features specific to certain sequence kinds and element types. For example, curves made of points (*CV_SEQ_KIND_CURVE|CV_SEQ_ELTYPE_POINT*), together with the flag *CV_SEQ_FLAG_CLOSED* belong to the type *CV_SEQ_POLYGON* or, if other flags are used, to its subtype. Many contour processing functions check the type of the input sequence and report an error if they do not support this type. The file cvtypes.h [p ??] 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.

CvSeqBlock

Continuous sequence block

```
typedef struct CvSeqBlock
{
    struct CvSeqBlock* prev; /* previous sequence block */
    struct CvSeqBlock* next; /* next sequence block */
    int start_index; /* index of the first element in the block +
    sequence->first->start_index */
    int count; /* number of elements in the block */
    char* data; /* pointer to the first element of the block */
} CvSeqBlock;
```

Sequence blocks make up a circular double-linked list, so the pointers *prev* and *next* are never *NULL* and point to the previous and the next sequence blocks within the sequence. It means that *next* of the last block is the first block and *prev* of the first block is the last block. The fields *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 *(start_index, count)* for the sequence blocks are *(2,3), (5, 5)*, and *(10, 2)* correspondingly. The parameter *start_index* of the first block is usually *0* unless some elements have been inserted at the beginning of the sequence.

CreateSeq

Creates sequence

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

seqFlags
: Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be set to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.

headerSize
: Size of the sequence header; must be greater or equal to *sizeof(CvSeq)*. If a specific type or its extension is indicated, this type must fit the base type header.

elemSize
: Size of the sequence elements in bytes. The size must be consistent with the sequence type. For example, for a sequence of points to be created, the element type *CV_SEQ_ELTYPE_POINT* should be specified and the parameter *elemSize* must be equal to *sizeof(CvPoint)*.

storage
: Sequence location.

The function cvCreateSeq [p 106] 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 *delta_elems* to the default value (that may be reassigned using cvSetSeqBlockSize [p 107] function), and clears other fields, including the space behind

sizeof(CvSeq).

SetSeqBlockSize

Sets up sequence block size

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

seq
: Sequence.

blockSize
: Desirable block size.

The function cvSetSeqBlockSize [p 107] affects the memory allocation granularity. When the free space in the 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 is, the lower the possible sequence fragmentation, 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 block header size plus the sequence element size.

SeqPush

Adds element to sequence end

```
char* cvSeqPush( CvSeq* seq, void* element=0 );
```

seq
: Sequence.

element
: Added element.

The function cvSeqPush [p 107] adds an element to the end of sequence and retuns pointer to the allocated element. If the input *element* is NULL, the function simply allocates a space for one more element.

The following code demonstrates how to create a new sequence using this function:

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, /* sequence of integer elements */
                          sizeof(CvSeq), /* header size - no extra fields */
                          sizeof(int), /* element size */
                          storage /* the container storage */ );
int i;
for( i = 0; i < 100; i++ )
{
    int* added = (int*)cvSeqPush( seq, &i );
    printf( "%d is added\n", *added );
```

```
}

...
/* release memory storage in the end */
cvReleaseMemStorage( &storage );
```

The function cvSeqPush [p 107] has O(1) complexity, but there is a faster method for writing large sequences (see cvStartWriteSeq [p 116] and related functions).

SeqPop

Removes element from sequence end

```
void cvSeqPop( CvSeq* seq, void* element=0 );
```

seq
: Sequence.

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

The function cvSeqPop [p 108] removes an element from the sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

SeqPushFront

Adds element to sequence beginning

```
char* cvSeqPushFront( CvSeq* seq, void* element=0 );
```

seq
: Sequence.

element
: Added element.

The function cvSeqPushFront [p 108] is similar to cvSeqPush [p 107] but it adds the new element to the beginning of the sequence. The function has O(1) complexity.

SeqPopFront

Removes element from sequence beginning

```
void cvSeqPopFront( CvSeq* seq, void* element=0 );
```

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

The function cvSeqPopFront [p 108] removes an element from the beginning of the sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

SeqPushMulti

Pushes several elements to the either end of sequence

```
void cvSeqPushMulti( CvSeq* seq, void* elements, int count, int in_front=0 );
```

seq
Sequence.
elements
Added elements.
count
Number of elements to push.
in_front
The flags specifying the modified sequence end:
CV_BACK (=0) - the elements are added to the end of sequence
CV_FRONT(!=0) - the elements are added to the beginning of sequence

The function cvSeqPushMulti [p 109] adds several elements to either 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.

SeqPopMulti

Removes several elements from the either end of sequence

```
void cvSeqPopMulti( CvSeq* seq, void* elements, int count, int in_front=0 );
```

seq
Sequence.
elements
Removed elements.
count
Number of elements to pop.
in_front
The flags specifying the modified sequence end:
CV_BACK (=0) - the elements are removed from the end of sequence
CV_FRONT(!=0) - the elements are removed from the beginning of sequence

The function cvSeqPopMulti [p 109] removes several elements from either end of the sequence. If the number of the elements to be removed exceeds the total number of elements in the sequence, the function removes as many elements as possible.

SeqInsert

Inserts element in sequence middle

```
char* cvSeqInsert( CvSeq* seq, int beforeIndex, void* element=0 );
```

seq
: Sequence.

beforeIndex
: Index before which the element is inserted. Inserting before 0 (the minimal allowed value of the parameter) is equal to cvSeqPushFront [p 108] and inserting before *seq->total* (the maximal allowed value of the parameter) is equal to cvSeqPush [p 107] .

element
: Inserted element.

The function cvSeqInsert [p 110] shifts the sequence elements from the inserted position to the nearest end of the sequence and copies the *element* content there if the pointer is not NULL. The function returns pointer to the inserted element.

SeqRemove

Removes element from sequence middle

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

seq
: Sequence.

index
: Index of removed element.

The function cvSeqRemove [p 110] removes elements with the given index. If the index is out of range the function reports an error. An attempt to remove an element from an empty sequence is a partitial case of this situation. The function removes an element by shifting the sequence elements between the nearest end of the sequence and the *index*-th position, not counting the latter.

ClearSeq

Clears sequence

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

seq
: Sequence.

The function cvClearSeq [p 110] removes all elements from the sequence. The function does not return the memory to the storage, but this memory is reused later when new elements are added to the sequence. This function time complexity is *O(1)*.

GetSeqElem

Returns pointer to sequence element by its index

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

seq
: Sequence.

index
: Index of element.

block
: Optional output parameter. If it is not *NULL*, the pointer to the sequence block containing the requested element is stored in this location.

The function cvGetSeqElem [p 111] 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 the sequence is most likely to consist of a single sequence block or the desired element is likely to be located in the first block, then the macro *CV_GET_SEQ_ELEM(elemType, seq, index)* should be used, where the parameter *elemType* is the type of sequence elements (CvPoint [p 30] 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 *GetSeqElem*. Negative indices always cause the cvGetSeqElem [p 111] call. The function has O(1) time complexity assuming that number of blocks is much smaller than the number of elements.

SeqElemIdx

Returns index of concrete sequence element

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

seq
: Sequence.

element
: Pointer to the element within the sequence.

block
: Optional argument. If the pointer is not *NULL*, the address of the sequence block that contains the element is stored in this location.

The function cvSeqElemIdx [p 111] returns the index of a sequence element or a negative number if the element is not found.

CvtSeqToArray

Copies sequence to one continuous block of memory

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

seq
: Sequence.

array
: Pointer to the destination array that must fit all the sequence elements.

slice
: The sequence part to copy to the array.

The function cvCvtSeqToArray [p ??] copies the entire sequence or subsequence to the specified buffer and returns the pointer to the buffer.

MakeSeqHeaderForArray

Constructs sequence from array

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

seqType
: Type of the created sequence.

headerSize
: Size of the header of the sequence. Parameter sequence must point to the structure of that size or greater size.

elemSize
: Size of the sequence element.

array
: Pointer to the array that makes up the sequence.

total
: Total number of elements in the sequence. The number of array elements must be equal to the value of this parameter.

sequence
: Pointer to the local variable that is used as the sequence header.

block
Pointer to the local variable that is the header of the single sequence block.

The function cvMakeSeqHeaderForArray [p 112] initializes sequence header for array. The sequence header as well as the sequence block are allocated by the user (for example, on stack). No data is copied by the function. The resultant sequence will consists of a single block and have NULL storage pointer, thus, it is possible to read its elements, but the attempts to add elements to the sequence will raise an error in most cases.

SeqSlice

Makes separate header for the sequence slice

```
CvSeq* cvSeqSlice( CvSeq* seq, CvSlice slice=CV_WHOLE_SEQ,
                   CvMemStorage* storage=0, int copyData=0 );

cvCloneSeq(seq[,storage]) ~ cvSeqSlice(seq,CV_WHOLE_SEQ,storage,1)
```

seq
Sequence.
slice
The part of the sequence to extract.
storage
The destination storage to keep the new sequence header and the copied data if any. If it is NULL, the function uses the storage containing the input sequence.
copyData
The flag that indicates whether to copy the elements of the extracted slice (*copyData*!=0) or not (*copyData*=0)

The function cvSeqSlice [p 113] creates another sequence and either makes it share the elements of the specified slice with the original sequence or creates another copy of the slice. So if one needs to process a part of sequence but the processing function does not have a slice parameter, the required sequence slice may be represented as a separate sequence using this function. Another purpose of the function is to make a copy of entire sequence that is done by cvCloneSeq [p ??] inline shortcut to cvSeqSlice [p 113]

SeqRemoveSlice

Removes sequence slice

```
void cvSeqRemoveSlice( CvSeq* seq, CvSlice slice );
```

seq
Sequence.
slice
The part of the sequence to remove.

The function cvSeqRemoveSlice [p 113] removes slice from the sequence.

SeqInsertSlice

Inserts array in the middle of sequence

```
void cvSeqInsertSlice( CvSeq* seq, int beforeIndex, const CvArr* fromArr );
```

seq
: Sequence.

slice
: The part of the sequence to remove.

The function cvSeqInsertSlice [p 114] inserts all *fromArr* array elements at the specified position of the sequence. The array may be matrix or another sequence.

SeqInvert

Reverses the order of sequence elements

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

seq
: Sequence.

The function cvSeqInvert [p 114] reverses the sequence in-place - makes the first element go last, the last element go first etc.

SeqSort

Sorts sequence element using the specified comparison function

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);

void cvSeqSort( CvSeq* seq, CvCmpFunc func, void* userdata );
```

seq
: The sequence to sort

func
: The comparison function that returns negative, zero or positive value depending on the elements relation (see the above declaration and the example below) - similar function is used by *qsort* from C runline except that in the latter *userdata* is not used

userdata
: The user parameter passed to the compasion function; helps to avoid global variables in some cases.

The function cvSeqSort [p 114] sorts the sequence in-place using the specified criteria. Below is the example of the function use:

```
/* Sort 2d points in top-to-bottom left-to-right order */
static int cmp_func( const void* _a, const void* _b, void* userdata )
{
    CvPoint* a = (CvPoint*)_a;
    CvPoint* b = (CvPoint*)_b;
    int y_diff = a->y - b->y;
    int x_diff = a->x - b->x;
    return y_diff ? y_diff : x_diff;
}

...

CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq), sizeof(CvPoint), storage );
int i;

for( i = 0; i < 10; i++ )
{
    CvPoint pt;
    pt.x = rand() % 1000;
    pt.y = rand() % 1000;
    cvSeqPush( seq, &pt );
}

cvSeqSort( seq, cmp_func, 0 /* userdata is not used here */ );

/* print out the sorted sequence */
for( i = 0; i < seq->total; i++ )
{
    CvPoint* pt = (CvPoint*)cvSeqElem( seq, i );
    printf( "(%d,%d)\n", pt->x, pt->y );
}

cvReleaseMemStorage( &storage );
```

StartAppendToSeq

Initializes process of writing to sequence

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

seq
: Pointer to the sequence.

writer
: Writer state; initialized by the function.

The function cvStartAppendToSeq [p 115] initializes the writer to write to the sequence. Written elements are added to the end of the sequence by *CV_WRITE_SEQ_ELEM(written_elem, writer)* macro. Note that during the writing process other operations on the sequence may yield incorrect result or even corrupt the sequence (see description of cvFlushSeqWriter [p 117] that helps to avoid some of that difficulties).

StartWriteSeq

Creates new sequence and initializes writer for it

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

seqFlags
: Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be equal to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.

headerSize
: Size of the sequence header. The parameter value may not be less than *sizeof(CvSeq)*. If a certain type or extension is specified, it must fit the base type header.

elemSize
: 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 *CV_SEQ_ELTYPE_POINT*), then the parameter elemSize must be equal to *sizeof(CvPoint)*.

storage
: Sequence location.

writer
: Writer state; initialized by the function.

The function cvStartWriteSeq [p 116] is a composition of cvCreateSeq [p 106] and cvStartAppendToSeq [p 115] . The pointer to the created sequence is stored at *writer->seq* and is also returned by cvEndWriteSeq [p 116] function that should be called in the end.

EndWriteSeq

Finishes process of writing sequence

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

writer
: Writer state

The function cvEndWriteSeq [p 116] finishes the writing process and returns the pointer to the written sequence. The function also truncates the last incomplete sequence block to return the remaining part of the block to the memory storage. After that the sequence can be read and modified safely.

FlushSeqWriter

Updates sequence headers from the writer state

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

writer
: Writer state

The function cvFlushSeqWriter [p 117] 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. In some algorithm requires often flush'es, consider using cvSeqPush [p 107] instead.

StartReadSeq

Initializes process of sequential reading from sequence

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

seq
: Sequence.

reader
: Reader state; initialized by the function.

reverse
: Determines the direction of the sequence traversal. If *reverse* is 0, the reader is positioned at the first sequence element, otherwise it is positioned at the last element.

The function cvStartReadSeq [p 117] initializes the reader state. 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(read_elem, reader)* in case of forward reading and by using *CV_REV_READ_SEQ_ELEM(read_elem, reader)* in case of reversed reading. Both macros put the sequence element to *read_elem* and move the reading pointer toward the next element. A circular structure of sequence blocks is used for the reading process, that is, after the last element has been read by the macro *CV_READ_SEQ_ELEM*, the first element is read when the macro is called again. The same applies to *CV_REV_READ_SEQ_ELEM* . There is no function to finish the reading process, since it neither changes the sequence nor creates any temporary buffers. The reader field *ptr* points to the current element of the sequence that is to be read next. The code below demonstrates how to use sequence writer and reader.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, sizeof(CvSeq), sizeof(int), storage );
CvSeqWriter writer;
CvSeqReader reader;
int i;

cvStartAppendToSeq( seq, &writer );
for( i = 0; i < 10; i++ )
{
```

```
    int val = rand()%100;
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("%d is written\n", val );
}
cvEndWriteSeq( &writer );

cvStartReadSeq( seq, &reader, 0 );
for( i = 0; i < seq->total; i++ )
{
    int val;
    CV_READ_SEQ_ELEM( val, reader );
    printf("%d is read\n", val );
}
...

cvReleaseStorage( &storage );
```

GetSeqReaderPos

Returns the current reader position

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

reader
: Reader state.

The function cvGetSeqReaderPos [p 118] returns the current reader position (within 0 ... *reader->seq->total* - 1).

SetSeqReaderPos

Moves the reader to specified position

```
void cvSetSeqReaderPos( CvSeqReader* reader, int index, int is_relative=0 );
```

reader
: Reader state.

index
: The destination position. If the positioning mode is used (see the next parameter) the actual position will be *index* mod *reader->seq->total*.

is_relative
: If it is not zero, then *index* is a relative to the current position.

The function cvSetSeqReaderPos [p 118] moves the read position to the absolute position or relative to the current position.

Sets

CvSet

Collection of nodes

```
typedef struct CvSetElem
{
    int flags; /* it is negative if the node is free and zero or positive otherwise */
    struct CvSetElem* next_free; /* if the node is free, the field is a
                                    pointer to next free node */
}
CvSetElem;

#define CV_SET_FIELDS()    \
    CV_SEQUENCE_FIELDS()   /* inherits from CvSeq */ \
    struct CvSetElem* free_elems; /* list of free nodes */

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

The structure CvSet [p 119] is a base for OpenCV sparse data structures.

As follows from the above declaration CvSet [p 119] inherits from CvSeq [p 104] and it adds *free_elems* field it to, which is a list of free nodes. Every set node, whether free or not, is the element of the underlying sequence. While there is no restrictions on elements of dense sequences, the set (and derived structures) elements must start with integer field and be able to fit CvSetElem structure, because these two fields (integer followed by the pointer) are required for organization of node set with the list of free nodes. If a node is free, *flags* field is negative (the most-significant bit, or MSB, of the field is set), and *next_free* points to the next free node (the first free node is referenced by *free_elems* field of CvSet [p 119]). And if a node is occupied, *flags* field is positive and contains the node index that may be retrieved using (set_elem->flags & CV_SET_ELEM_IDX_MASK) expression, the rest of the node content is determined by the user. In particular, the occupied nodes are not linked as the free nodes are, so the second field can be used for such a link as well as for some different purpose. The macro *CV_IS_SET_ELEM(set_elem_ptr)* can be used to determined whether the specified node is occupied or not.

Initially the set and the list are empty. When a new node is requiested from the set, it is taken from the list of free nodes, which is updated then. If the list appears to be empty, a new sequence block is allocated and all the nodes within the block are joined in the list of free nodes. Thus, *total* field of the set is the total number of nodes both occupied and free. When an occupied node is released, it is added to the list of free nodes. The node released last will be occupied first.

In OpenCV CvSet [p 119] is used for representing graphs (CvGraph [p 122]), sparse multi-dimensional arrays (CvSparseMat [p 34]), planar subdivisions (CvSubdiv2D [p ??]) etc.

CreateSet

Creates empty set

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

setFlags
: Type of the created set.

headerSize
: Set header size; may not be less than *sizeof(CvSet)*.

elemSize
: Set element size; may not be less than CvSetElem [p ??] .

storage
: Container for the set.

The function cvCreateSet [p 120] creates an empty set with a specified header size and element size, and returns the pointer to the set. The function is just a thin layer on top of cvCreateSeq [p 106] .

SetAdd

Occupies a node in the set

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

set
: Set.

elem
: Optional input argument, inserted element. If not NULL, the function copies the data to the allocated node (The MSB of the first integer field is cleared after copying).

insertedElem
: Optional output argument; the pointer to the allocated cell.

The function cvSetAdd [p 120] allocates a new node, optionally copies input element data to it, and returns the pointer and the index to the node. The index value is taken from the lower bits of *flags* field of the node. The function has O(1) complexity, however there exists a faster function for allocating set nodes (see cvSetNew [p 121]).

SetRemove

Removes element from set

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

set
Set.
index
Index of the removed element.

The function cvSetRemove [p 120] removes an element with a specified index from the set. If the node at the specified location is not occupied the function does nothing. The function has O(1) complexity, however, cvSetRemoveByPtr [p 121] provides yet faster way to remove a set element if it is located already.

SetNew

Adds element to set (fast variant)

```
CvSetElem* cvSetNew( CvSet* set );
```

set
Set.

The function cvSetNew [p 121] is inline light-weight variant of cvSetAdd [p 120] . It occupies a new node and returns pointer to it rather than index.

SetRemoveByPtr

Removes set element given its pointer

```
void cvSetRemoveByPtr( CvSet* set, void* elem );
```

set
Set.
elem
Removed element.

The function cvSetRemoveByPtr [p 121] is inline light-weight variant of cvSetRemove [p 120] that takes element pointer. The function does not check whether the node is occupied or not - the user should take care of it.

GetSetElem

Finds set element by its index

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

set
Set.
index
Index of the set element within a sequence.

The function cvGetSetElem [p 121] finds a set element by index. The function returns the pointer to it or 0 if the index is invalid or the corresponding node is free. The function supports negative indices as it uses cvGetSeqElem [p 111] to locate the node.

ClearSet

Clears set

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

set
Cleared set.

The function cvClearSet [p 122] removes all elements from set. It has O(1) time complexity.

Graphs

CvGraph

Oriented or unoriented weigted graph

```
#define CV_GRAPH_VERTEX_FIELDS()    \
    int flags; /* vertex flags */   \
    struct CvGraphEdge* first; /* the first incident edge */

typedef struct CvGraphVtx
{
    CV_GRAPH_VERTEX_FIELDS()
}
CvGraphVtx;

#define CV_GRAPH_EDGE_FIELDS()      \
    int flags; /* edge flags */     \
    float weight; /* edge weight */ \
    struct CvGraphEdge* next[2]; /* the next edges in the incidence lists for staring (0) */ \
                                 /* and ending (1) vertices */ \
    struct CvGraphVtx* vtx[2]; /* the starting (0) and ending (1) vertices */

typedef struct CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS()
}
CvGraphEdge;

#define  CV_GRAPH_FIELDS()                  \
    CV_SET_FIELDS() /* set of vertices */   \
```

```
    CvSet* edges;   /* set of edges */

typedef struct CvGraph
{
    CV_GRAPH_FIELDS()
}
CvGraph;
```

The structure CvGraph [p 122] is a base for graphs used in OpenCV.

Graph structure inherits from CvSet [p 119] - this part describes common graph properties and the graph vertices, and contains another set as a member - this part describes the graph edges.

The vertex, edge and the graph header structures are declared using the same technique as other extendible OpenCV structures - via macros, that simplifies extension and customization of the structures. While the vertex and edge structures do not inherit from CvSetElem [p ??] explicitly, they satisfy both conditions on the set elements - have an integer field in the beginning and fit CvSetElem structure. The *flags* fields are used as for indicating occupied vertices and edges as well as for other purposes, for example, for graph traversal (see cvStartScanGraph [p 130] et al.), so it is better not to use them directly.

The graph is represented as a set of edges each of whose has the list of incident edges. The incidence lists for different vertices are interleaved to avoid information duplication as much as posssible.

The graph may be oriented or unoriented. In the latter case there is no distiction between edge connecting vertex A with vertex B and the edge connecting vertex B with vertex A - only one of them can exist in the graph at the same moment and it represents both <A, B> and <B, A> edges..

CreateGraph

Creates empty graph

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

graphFlags
: Type of the created graph. Usually, it is either *CV_SEQ_KIND_GRAPH* for generic unoriented graphs and *CV_SEQ_KIND_GRAPH* | *CV_GRAPH_FLAG_ORIENTED* for generic oriented graphs.

headerSize
: Graph header size; may not be less than *sizeof(CvGraph).*

vertexSize
: Graph vertex size; the custom vertex structure must start with CvGraphVtx [p ??] (use *CV_GRAPH_VERTEX_FIELDS()*)

edgeSize
: Graph edge size; the custom edge structure must start with CvGraphEdge [p ??] (use *CV_GRAPH_EDGE_FIELDS()*)

storage
: The graph container.

The function cvCreateGraph [p 123] creates an empty graph and returns pointer to it.

GraphAddVtx

Adds vertex to graph

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

graph
: Graph.

vtx
: Optional input argument used to initialize the added vertex (only user-defined fields beyond *sizeof(CvGraphVtx)* are copied).

insertedVtx
: Optional output argument. If not *NULL*, the address of the new vertex is written there.

The function cvGraphAddVtx [p 124] adds a vertex to the graph and returns the vertex index.

GraphRemoveVtx

Removes vertex from graph

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

graph
: Graph.

vtxIdx
: Index of the removed vertex.

The function cvGraphRemoveAddVtx [p ??] removes a vertex from the graph together with all the edges incident to it. The function reports an error, if the input vertex does not belong to the graph.

GraphRemoveVtxByPtr

Removes vertex from graph

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

graph
: Graph.

vtx
: Pointer to the removed vertex.

The function cvGraphRemoveVtxByPtr [p 124] removes a vertex from the graph together with all the edges incident to it. The function reports an error, if the vertex does not belong to the graph.

GetGraphVtx

Finds graph vertex by index

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

graph
: Graph.

vtxIdx
: Index of the vertex.

The function cvGetGraphVtx [p 125] finds the graph vertex by index and returns the pointer to it or NULL if the vertex does not belong to the graph.

GraphVtxIdx

Returns index of graph vertex

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

graph
: Graph.

vtx
: Pointer to the graph vertex.

The function cvGraphVtxIdx [p 125] returns index of the graph vertex.

GraphAddEdge

Adds edge to graph

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

graph
: Graph.

startIdx
: Index of the starting vertex of the edge.

endIdx
: Index of the ending vertex of the edge. For unoriented graph the order of the vertex parameters does not matter.

edge
Optional input parameter, initialization data for the edge.
insertedEdge
Optional output parameter to contain the address of the inserted edge.

The function cvGraphAddEdge [p 125] connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e. when the result is negative) the function also reports an error by default.

GraphAddEdgeByPtr

Adds edge to graph

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

graph
Graph.
startVtx
Pointer to the starting vertex of the edge.
endVtx
Pointer to the ending vertex of the edge. For unoriented graph the order of the vertex parameters does not matter.
edge
Optional input parameter, initialization data for the edge.
insertedEdge
Optional output parameter to contain the address of the inserted edge within the edge set.

The function cvGraphAddEdge [p 125] connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e. when the result is negative) the function also reports an error by default.

GraphRemoveEdge

Removes edge from graph

```
void cvGraphRemoveEdge( 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. For unoriented graph the order of the vertex parameters does not matter.

The function cvGraphRemoveEdge [p 126] removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

GraphRemoveEdgeByPtr

Removes edge from graph

```
void 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. For unoriented graph the order of the vertex parameters does not matter.

The function cvGraphRemoveEdgeByPtr [p 127] removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

FindGraphEdge

Finds edge in graph

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

graph
: Graph.

startIdx
: Index of the starting vertex of the edge.

endIdx
: Index of the ending vertex of the edge. For unoriented graph the order of the vertex parameters does not matter.

The function cvFindGraphEdge [p 127] finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

FindGraphEdgeByPtr

Finds edge in graph

```
CvGraphEdge* cvFindGraphEdgeByPtr( CvGraph* graph, CvGraphVtx* startVtx, CvGraphVtx* endVtx );
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

graph
: Graph.

startVtx
: Pointer to the starting vertex of the edge.

endVtx
: Pointer to the ending vertex of the edge. For unoriented graph the order of the vertex parameters does not matter.

The function cvFindGraphEdge [p 127] finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

GraphEdgeIdx

Returns index of graph edge

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

graph
: Graph.

edge
: Pointer to the graph edge.

The function cvGraphEdgeIdx [p 128] returns index of the graph edge.

GraphVtxDegree

Counts edges indicent to the vertex

```
int cvGraphVtxDegree( CvGraph* graph, int vtxIdx );
```

graph
: Graph.

vtx
: Index of the graph vertex.

The function cvGraphVtxDegree [p 128] returns the number of edges incident to the specified vertex, both incoming and outcoming. To count the edges, the following code is used:

```
CvGraphEdge* edge = vertex->first; int count = 0;
while( edge )
{
    edge = CV_NEXT_GRAPH_EDGE( edge, vertex );
    count++;
}
```

The macro *CV_NEXT_GRAPH_EDGE(edge, vertex)* returns the edge incident to *vertex* that follows after *edge*.

GraphVtxDegreeByPtr

Finds edge in graph

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

graph
: Graph.

vtx
: Pointer to the graph vertex.

The function cvGraphVtxDegree [p 128] returns the number of edges incident to the specified vertex, both incoming and outcoming.

ClearGraph

Clears graph

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

graph
: Graph.

The function cvClearGraph [p 129] removes all vertices and edges from the graph. The function has O(1) time complexity.

CloneGraph

Clone graph

```
CvGraph* cvCloneGraph( const CvGraph* graph, CvMemStorage* storage );
```

graph
: The graph to copy.

storage
: Container for the copy.

The function cvCloneGraph [p 129] creates full copy of the graph. If the graph vertices or edges have pointers to some external data, it still be shared between the copies. The vertex and edge indices in the new graph may be different from the original, because the function defragments the vertex and edge sets.

CvGraphScanner

Graph traversal state

```
typedef struct CvGraphScanner
{
    CvGraphVtx* vtx;       /* current graph vertex (or current edge origin) */
    CvGraphVtx* dst;       /* current graph edge destination vertex */
    CvGraphEdge* edge;     /* current edge */

    CvGraph* graph;        /* the graph */
    CvSeq*   stack;        /* the graph vertex stack */
    int      index;        /* the lower bound of certainly visited vertices */
    int      mask;         /* event mask */
}
CvGraphScanner;
```

The structure CvGraphScanner [p 130] is used for depth-first graph traversal. See discussion of the functions below.

StartScanGraph

Initializes graph traverser state

```
void cvStartScanGraph( CvGraph* graph, CvGraphScanner* scanner,
                       CvGraphVtx* vtx=0, int mask=CV_GRAPH_ALL_ITEMS );
```

graph
: Graph.

scanner
: Graph traversal state. It is initialized by the function.

vtx
: Initial vertex to start from.

mask
: Event mask indicating which events are interesting to the user (where cvNextGraphItem [p 131] function returns control to the user) It can be *CV_GRAPH_ALL_ITEMS* (all events are interesting) or combination of the following flags:
  - CV_GRAPH_VERTEX - stop at the graph vertices visited for the first time
  - CV_GRAPH_TREE_EDGE - stop at tree edges (*tree edge* is the edge connecting the last visited vertex and the vertex to be visited next)
  - CV_GRAPH_BACK_EDGE - stop at back edges (*back edge* is the edge connecting the last visited vertex and the vertex that was visited before)
  - CV_GRAPH_FORWARD_EDGE - stop at forward edges (*forward edge* is the edge connecting the vertex not visited yet and the last visited vertex (in that order). The *forward edges* are

possible only during oriented graph traversal)

- CV_GRAPH_CROSS_EDGE - stop at cross edges (*cross edge* is similar to *back edge* but the ending vertex belongs to another traversal tree). The *cross edges* are also possible only during oriented graphs traversal)
- CV_GRAPH_ANY_EDGE - stop and any edge (*tree, back, forward and cross edges*)
- CV_GRAPH_NEW_TREE - stop in the beginning of every new traversal tree. When the traversal procedure visits all vertices and edges reachible from the initial vertex (the visited vertices together with tree edges make up a tree), it search for some unvisited vertices in the graph and resumes the traversal process from the vertex. Before starting the new tree (including the initial call of the traversal procedure) it generates *CV_GRAPH_NEW_TREE* event.
  For unoriented graphs traversal tree corresponds to a connected component of the graph.
- CV_GRAPH_BACKTRACKING - stop at every already visited vertex during backtracking - returning to visited already visited vertexes of the traversal tree.

The function cvStartScanGraph [p 130] initializes graph traverser state. The initialized structure is used in cvNextGraphItem [p 131] function - the incremental traversal procedure.

NextGraphItem

Makes one or more steps of the graph traversal procedure

```
int cvNextGraphItem( CvGraphScanner* scanner );
```

scanner
: Graph traversal state. It is updated by the function.

The function cvNextGraphItem [p 131] traverses through the graph until an event interesting to the user (that is, an event, marked in the *mask* in cvStartScanGraph [p 130] call) is met or the traversal is over. In the first case it returns one of the events, listed in the description of *mask* parameter above and with the next call with the same state it resumes the traversal. In the latter case it returns CV_GRAPH_OVER (-1). When the event is *CV_GRAPH_VERTEX*, or *CV_GRAPH_BACKTRACKING* or *CV_GRAPH_NEW_TREE*, the currently observed vertex is stored in *scanner->vtx*. And if the event is edge-related, the edge itself is stored at *scanner->edge*, the previously visited vertex - at *scanner->vtx* and the other ending vertex of the edge - at *scanner->dst*.

EndScanGraph

Finishes graph traversal procedure

```
void cvEndScanGraph( CvGraphScanner* scanner );
```

scanner
: Graph traversal state.

The function cvEndScanGraph [p 131] finishes graph traversal procedure. It must be called after CV_GRAPH_OVER event is received or if the traversal is interrupted somewhere before, because the traverser state contains dynamically allocated structures that need to be released

---

Trees

CV_TREE_NODE_FIELDS

Helper macro for a tree node type declaration

```
#define CV_TREE_NODE_FIELDS(node_type)                          \
    int       flags;         /* micsellaneous flags */          \
    int       header_size;   /* size of sequence header */      \
    struct    node_type* h_prev; /* previous sequence */        \
    struct    node_type* h_next; /* next sequence */            \
    struct    node_type* v_prev; /* 2nd previous sequence */    \
    struct    node_type* v_next; /* 2nd next sequence */
```

The macro *CV_TREE_NODE_FIELDS()* is used to declare structures that can be organized into hierarchical strucutures (trees). Although, it is not shown, the macro is used to declared CvSeq [p 104] - the basic type for all dynamical structures and CvFileNode [p 140] - XML node type used in reading/writing functions (see *Persistence* section below). The trees made of nodes declared using this macro can be processed using the functions described below in this section.

CvTreeNodeIterator

Opens existing or creates new file storage

```
typedef struct CvTreeNodeIterator
{
    const void* node;
    int level;
    int maxLevel;
}
CvTreeNodeIterator;
```

The structure CvTreeNodeIterator [p 132] is used to traverse trees. The tree node declaration should start with *CV_TREE_NODE_FIELDS(...)* macro.

InitTreeNodeIterator

Initializes tree node iterator

```
void cvInitTreeNodeIterator( CvTreeNodeIterator* treeIterator,
                             const void* first, int maxLevel );
```

treeIterator
: Tree iterator initialized by the function.

first
: The initial node to start traversing from.

maxLevel
: The maximal level of the tree (*first* node assumed to be at the first level) to traverse up to. For example, 1 means that only nodes at the same level as *first* should be visited, 2 means that the nodes on the same level as *first* and their direct children should be visited etc.

The function cvInitTreeNodeIterator [p 132] initializes tree iterator. The tree is traversed in depth-first order.

NextTreeNode

Returns the currently observed node and moves iterator toward the next node

```
void* cvNextTreeNode( CvTreeNodeIterator* treeIterator );
```

treeIterator
: Tree iterator initialized by the function.

The function cvNextTreeNode [p 133] returns the currently observed node and then updates the iterator - moves it toward the next node. In other words, the function behavior is similar to *p++ expression on usual C pointer or C++ collection iterator. The function returns NULL if there is no more nodes.

PrevTreeNode

Returns the currently observed node and moves iterator toward the previous node

```
void* cvPrevTreeNode( CvTreeNodeIterator* treeIterator );
```

treeIterator
: Tree iterator initialized by the function.

The function cvPrevTreeNode [p 133] returns the currently observed node and then updates the iterator - moves it toward the previous node. In other words, the function behavior is similar to *p-- expression on usual C pointer or C++ collection iterator. The function returns NULL if there is no more nodes.

TreeToNodeSeq

Gathers all node pointers to the single sequence

```
CvSeq* cvTreeToNodeSeq( const void* first, int header_size, CvMemStorage* storage );
```

first
: The initial tree node.

header_size
: Header size of the created sequence (sizeof(CvSeq) is the most used value).

storage
: Container for the sequence.

The function cvTreeToNodeSeq [p 134] puts pointers of all nodes reacheable from *first* to the single sequence. The pointers are written subsequently in the depth-first order.

InsertNodeIntoTree

Adds new node to the tree

```
void cvInsertNodeIntoTree( void* node, void* parent, void* frame );
```

node
: The inserted node.

parent
: The parent node that is already in the tree.

frame
: The top level node. If *parent* and *frame* are the same, *v_prev* field of *node* is set to NULL rather than *parent*.

The function cvInsertNodeIntoTree [p 134] adds another node into tree. The function does not allocate any memory, it can only modify links of the tree nodes.

RemoveNodeFromTree

Removes node from tree

```
void cvRemoveNodeFromTree( void* node, void* frame );
```

node
: The removed node.

frame
: The top level node. If *node->v_prev = NULL* and *node->h_prev* is NULL (i.e. if *node* is the first child of *frame*), *frame->v_next* is set to *node->h_next* (i.e. the first child or frame is changed).

The function cvRemoveNodeFromTree [p 134] removes node from tree. The function does not deallocate any memory, it can only modify links of the tree nodes.

Persistence (Writing and Reading Structures)

OpenFileStorage

Opens existing or creates new file storage

```
CvFileStorage*  cvOpenFileStorage( const char* filename, CvMemStorage* memstorage, int flags );
```

filename
: The storage file name.

memstorage
: Memory storage used for storing temporary data and read dynamic structures. If it is NULL, the temporary memory storage is created and used.

flags
: Can be one of the following:
CV_STORAGE_READ - the storage is open for reading
CV_STORAGE_WRITE_TEXT - the storage is open for writing data in text format
CV_STORAGE_WRITE_BINARY - the storage is open for writing data in XDR and base64-encoded binary format

The function cvOpenFileStorage [p 135] open existing file storage or creates a new storage. The file has XML format and it allows user to store as standard OpenCV arrays and dynamic structures as well as custom data structures. The function returns pointer to CvFileStorage [p ??] structure, which declaration is hidden, though not needed to access directly.

ReleaseFileStorage

Releases file storage

```
void  cvReleaseFileStorage( CvFileStorage** storage );
```

storage
: Double pointer to the released file storage.

The function cvReleaseFileStorage [p 135] closes the file on disk that has been written or read and releases all temporary structures. It must be called after all I/O operations with the storage are finished.

Write

Writes array or dynamic structure to the file storage

```
void  cvWrite( CvFileStorage* storage, const char* name,
               const void* structPtr,
               CvAttrList attributes=cvAttrList(),
               int flags=0 );
```

storage
: File storage.

name
: Name, or ID, of the written structure. It is used to index the written information and then access it using these ID's. If it is NULL or empty (""), no ID is written. If it has special value "<auto>", the address of the written structure in heximal notation will be used as the name.

structPtr
: The written structure - CvMat*, IplImage*, CvSeq*, CvGraph* etc.

attributes
: The list of attributes that can be formed from NULL-terminated array of *<attr_name, attr_value>* pairs using cvAttrList [p ??] () function (see the example below). Most often it is just empty. The passed attributes override standard attributes with the same name, e.g. user may specify *header_dt* or *dt* attributes to write dynamic structures with custom headers and element types.

flags
: The operation flags passed into the specific loading/storing function for the particular data type. Usually it is not used and may be set to 0. In case of contour trees (for details see cvFindContours [p 176] function in *Structural Analysis* chapter of the reference) it may be set to CV_WRITE_TREE to force the whole contour tree to be written.

The function cvWrite [p 136] writes a passed structure to OpenCV file storage. The sample below demonstrates how to write different types of data to storage.

```
/****************** Writing Data *************************/
#include <cv.h>

int main( int argc, char** argv )
{
    CvMemStorage* memstorage = cvCreateMemStorage(0);
    CvFileStorage* storage = cvOpenFileStorage( "sample.xml", 0, CV_STORAGE_WRITE_TEXT );
    CvMat* mat = cvCreateMat( 3, 3, CV_32FC1 );
    CvSeq* seq = cvCreateSeq( CV_32SC1, sizeof(CvSeq), sizeof(int), memstorage );
    char* seq_attr[] =
    {
    "created_by", argv[0],
    "the_sequence_creation_date", "1 Sep 2002",
    "comment", "just a comment",
    0
    };
    CvTermCriteria criteria = { CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 10, 0.1 };
    const char* string1 = "test";
    int i;

    cvSetIdentity( mat );
    cvWrite( storage, "The identity matrix", mat, cvAttrList(), 0 );

    for( i = 0; i < 10; i++ )
```

```
        cvSeqPush( seq, &i );

    cvWrite( storage, "SmallSequence", seq, cvAttrList(seq_attr,0), 0 );
    cvWriteElem( storage, "SampleStructure", "iid", &criteria ); /* writing C structure, see below */
    cvWriteElem( storage, "SampleString", "a", string1 ); /* writing C string, see below */

    cvReleaseFileStorage( &storage );
    cvReleaseMemStorage( &memstorage );
    return 0;
}
/*****************************************************/
```

After compiling and runnning the sample the file *sample.xml* will contain something like this:

```
<?xml version="1.0"?>
<opencv_storage>
<struct id="The identity matrix" type="CvMat" dt="f" size="3 3" format="text">
  1.000000e+000  0.000000e+000  0.000000e+000  0.000000e+000  1.000000e+000
  0.000000e+000  0.000000e+000  0.000000e+000  1.000000e+000
</struct>
<struct id="SmallSequence" type="CvSeq" flags="42990003" dt="i" format="text"
 created_by="D:\OpenCV\bin\test.exe" the_sequence_creation_date="1 Sep 2002"
 comment="just a comment">
      0      1      2      3      4      5      6      7      8      9
</struct>
<elem id="SampleStructure" dt="iid" value="      3     10  1.000000000000000e-001"/>
<elem id="SampleString" dt="a" value="test"/>
</opencv_storage>
```

StartWriteStruct

Writes the opening tag of a compound structure

```
void  cvStartWriteStruct( CvFileStorage* storage, const char* name,
                          const char* typeName=0, const void* structPtr=0,
                          CvAttrList attributes=cvAttrList());
```

storage
: File storage.

name
: Name, or ID, of the written structure. It is used to index the written information and then access it using these ID's. If it is NULL or empty (""), no ID is written. If it has special value "<auto>", the address of the written structure in heximal notation will be used as the name.

structPtr
: The written structure pointer. It is not used unless *name = "<auto>"*.

attributes
: The list of attributes (the same as in the previous function)

The function cvStartWriteStruct [p 137] writes the opening tag of a compound structure. It is used by cvWrite [p 136] function and can be used explicitly to group some structures or write an writer for some custom data structure.

EndWriteStruct

Closes the last opened tag

```
void  cvEndWriteStruct( CvFileStorage* storage );
```

storage
: File storage.

The function cvEndWriteStruct [p 138] closes the most recent opened tag.

WriteElem

Writes a scalar variable

```
void  cvWriteElem( CvFileStorage* storage, const char* name,
                   const char* elem_spec, const void* data_ptr );
```

storage
: File storage

name
: Name, or ID, of the written scalar. As usual, "<auto>" means that the data pointer will be used as a name.

elem_spec
: A sequence of character each of whose specifies a type of particualar field of the written structure:
  - ’a’ - NULL-terminated C string. It must be the only character of specification string.
  - ’u’ - 8-bit unsigned number
  - ’c’ - 8-bit signed number
  - ’s’ - 16-bit signed number
  - ’i’ - 32-bit signed number
  - ’f’ - single precision floating-point number
  - ’d’ - double precision floating-point number
  - ’p’ - pointer, it is not stored, but it takes some space in the input structure, so it must be specified in order to write the subsequent fields correctly.
  - ’r’ - the same as pointer, but the integer number, but the lowest 32 bit of the pointer are written as an integer. This is useful for storing dynamic structures where different nodes reference each other. In this case the pointers are replaced with some indices, the structure is written and, the pointers are restored back.

data_ptr
: Pointer to the written data. The written data can be a single value of one of basic numerical types (*unsigned char, char, short, int, float* or *double*), C structure containing one or more numerical fields or a character string. In case of C structures an ideal alignment must be used - *short*’s must be aligned by 2 bytes, *integer*’s and *float*’s by 4 bytes and *double*’s by 8 bytes. Usually such an alignment is used by C/C++ compiler by default, however some structures, e.g. *BITMAPFILEHEADER* Win32 structure break this rule. If you want to store such a structure, you may use sprintf(elem_spec,

"%du", sizeof(my_struct)) to form an element spec that allows to store arbitrary structure, though the representation will neither be readable nor portable.

The function cvWriteElem [p 138] writes a single numberical value, a structure of numerical values or a character string. Here are some examples (see cvWrite [p 136] function discussion for complete sample):

```
CvScalar scalar = { 1., 2., 3.14, 4. };
cvWriteElem( filestorage, "scalar1", "4d", &scalar );

CvPoint pt = { 100, 50 };
cvWriteElem( filestorage, "feature_point", "2i", &pt };

struct
{
    char c;
    uchar u;
    short s;
    int i;
    float f;
    double d;
}
big_twos = { 20, 200, 20000, 2000000, 2e10, 2e100 };
cvWriteElem( filestorage, "big_twos", "cusifd", &big_twos );

cvWriteElem( filestorage, "string1", "a", "Hello, world!" };
```

Read

Reads array or dynamic structure from the file storage

```
void*  cvRead( CvFileStorage* storage, const char* name, CvAttrList** list=0 );
```

storage
: File storage.

name
: Name of the structure to read.

list
: Optional output parameter that is filled with the node attributes list.

The function cvRead [p 139] reads a structure with the specified name from OpenCV file storage. The structure is stored inside the file storage so it is be deallocated when the file storage is released, except the case when it is dynamic strucutre and non-NULL memory storage was passed to cvOpenFileStorage [p 135] function. If you want to keep the sturcture, use cvClone* [p ??] .

ReadElem

Reads a scalar variable

```
void  cvReadElem( CvFileStorage* storage, const char* name, void* data_ptr );
```

storage
: File storage

name
: Name of the variable to read.

data_ptr
: Pointer to the destination structure. In case of strings, *data_ptr* should be *char*** - pointer to the string pointer that is filled by the function.

The function cvReadElem [p 139] reads a single numberical value, a structure of numerical values or a character string. The order the variables are read in may be different from the order they are written. Here are examples - counterparts for examples from cvWriteElem [p 138] discussion:

```
CvScalar scalar;
cvWriteElem( filestorage, "scalar1", &scalar );

CvPoint pt;
cvWriteElem( filestorage, "feature_point", &pt };

struct
{
    char c;
    uchar u;
    short s;
    int i;
    float f;
    double d;
}
big_twos;

cvReadElem( filestorage, "big_twos", &big_twos );

const char* string1 = 0;
cvReadElem( filestorage, "string1", (void*)&string1 };
```

CvFileNode

XML node representation

```
typedef struct CvFileNode
{
    int       flags;         /* micsellaneous flags */
    int       header_size;   /* size of node header */
    struct    node_type* h_prev; /* previous node having the same parent */
    struct    node_type* h_next; /* next node having the same parent */
    struct    node_type* v_prev; /* the parent node */
    struct    node_type* v_next; /* the first child node */
    const char* tagname; /* INTERNAL: XML tag name */
    const char* name; /* the node name */
    CvAttrList* attr; /* list of attributes */
    struct CvFileNode* hash_next; /* INTERNAL: next entry in hash table */
    unsigned hash_val; /* INTERNAL: hash value */
    int elem_size; /* size of a structure elements */
```

```
    struct CvTypeInfo* typeinfo; /* INTERNAL: type information */
    const char* body; /* INTERNAL: pointer to the structure or scalar content within XML file,
                         filled by XML parser */
    const void* content; /* the read structure, NULL if it is read yet */
}
CvFileNode;
```

The structure CvFileNode [p 140] represents an XML node - building block of OpenCV file storage. The nodes are organized into tree that has a root node correponding to <opencv_storage> tag. When a file storage is written, the tree is constructed by cvWrite [p 136] , cvStartWriteStruct [p 137] , cvEndWriteStruct [p 138] and cvWriteElem [p 138] functions, and it does not to be accessed directly in most of cases. When the file storage is read, the whole tree is reconstructed by XML parser in cvOpenFileStorage [p 135] function, but none of nodes are decoded. Instead, a particular node can be decoded via cvReadFileNode [p 141] function, after its pointer is retrived by the node name using cvGetFileNode [p 141] function or the node is reached from the root node. The functions cvInitTreeNodeIterator [p 132] , cvNextTreeNode [p 133] and cvPrevTreeNode [p 133] can be used to traverse the XML tree. When the node is decoded, *content* field becomes valid and the same pointer will be returned by sebsequent calls to cvReadFileNode [p 141] or cvRead [p 139] for this node.

GetFileNode

Finds XML node by name

```
CvFileNode* cvGetFileNode( CvFileStorage* storage, const char* name );
```

storage
: File storage.

name
: The searched node name. If it is NULL, the root of the XML tree is returned.

The function cvGetFileNode [p 141] locates XML node by name and returns pointer to the node or NULL if the node is not found.

.

ReadFileNode

Decodes XML node content

```
void* cvReadFileNode( CvFileStorage* storage, CvFileNode* node, CvAttrList** list=0 );
```

storage
: File storage.

node
: The node to decode.

list
: Optional output parameter that is filled with the node attributes list.

The function cvReadFileNode [p 141] decodes the file node content. If *content* field of the node is not NULL, no decoding is done and the pointer is simply returned.

.

Image Processing and Analysis Reference

- Utility Functions [p ??]
  - MatchTemplate [p ??]

Note:
The chapter describes functions for image processing and analysis. Most of the functions work with 2d arrays of pixels. We refer the arrays as "images" however they do not neccesserily have to be IplImage's, they may be CvMat's or CvMatND's as well.

Drawing Functions

Drawing functions work with arbitrary 8-bit images or single-channel images with larger depth: 16s, 32s, 32f, 64f All the functions include parameter color that means rgb value (that may be constructed with `CV_RGB` macro) for color images and brightness for grayscale images.

If a drawn figure is partially or completely outside the image, it is clipped.

CV_RGB

Constructs a color value

```
#define CV_RGB( r, g, b )  (int)((uchar)(b) + ((uchar)(g) << 8) + ((uchar)(r) << 16))
```

Line

Draws simple or thick line segment

```
void cvLine( CvArr* img, CvPoint pt1, CvPoint pt2, double color, int thickness=1, int connectivity=8 );
```

img
: The image.

pt1
: First point of the line segment.

pt2
: Second point of the line segment.

color
: Line color (RGB) or brightness (grayscale image).

thickness
: Line thickness.

connectivity
: Line connectivity, 8 (by default) or 4. It is possible to pass 0 instead of 8.

The function cvLine [p 145] draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. The 8-connected or 4-connected 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 )`.

LineAA

Draws antialiased line segment

```
void cvLineAA( CvArr* img, CvPoint pt1, CvPoint pt2, double color, int scale=0 );
```

img
: Image.

pt1
: First point of the line segment.

pt2
: Second point of the line segment.

color
: Line color (RGB) or brightness (grayscale image).

scale
: Number of fractional bits in the end point coordinates.

The function cvLineAA [p 146] draws the 8-connected line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. The algorithm includes some sort of Gaussian filtering to get a smooth picture. To specify the line color, the user may use the macro `CV_RGB( r, g, b )`.

Rectangle

Draws simple, thick or filled rectangle

```
void cvRectangle( CvArr* img, CvPoint pt1, CvPoint pt2, double color, int thickness=1 );
```

img
: Image.

pt1
: One of the rectangle vertices.

pt2
: Opposite rectangle vertex.

color
: Line color (RGB) or brightness (grayscale image).

thickness
: Thickness of lines that make up the rectangle. Negative values, e.g. CV_FILLED, make the function to draw a filled rectangle.

The function cvRectangle [p 146] draws a rectangle with two opposite corners `pt1` and `pt2`.

Circle

Draws simple, thick or filled circle

```
void cvCircle( CvArr* img, CvPoint center, int radius, double color, int thickness=1 );
```

img
: Image where the line is drawn.

center
: Center of the circle.

radius
: Radius of the circle.

color
: Circle color (RGB) or brightness (grayscale image).

thickness
: Thickness of the circle outline if positive, otherwise indicates that a filled circle has to be drawn.

The function cvCircle [p 147] 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 )`.

Ellipse

Draws simple or thick elliptic arc or fills ellipse sector

```
void cvEllipse( CvArr* img, CvPoint center, CvSize axes, double angle,
                double startAngle, double endAngle, double color, int thickness=1 );
```

img
: Image.

center
: Center of the ellipse.

axes
: Length of the ellipse axes.

angle
: Rotation angle.

startAngle
: Starting angle of the elliptic arc.

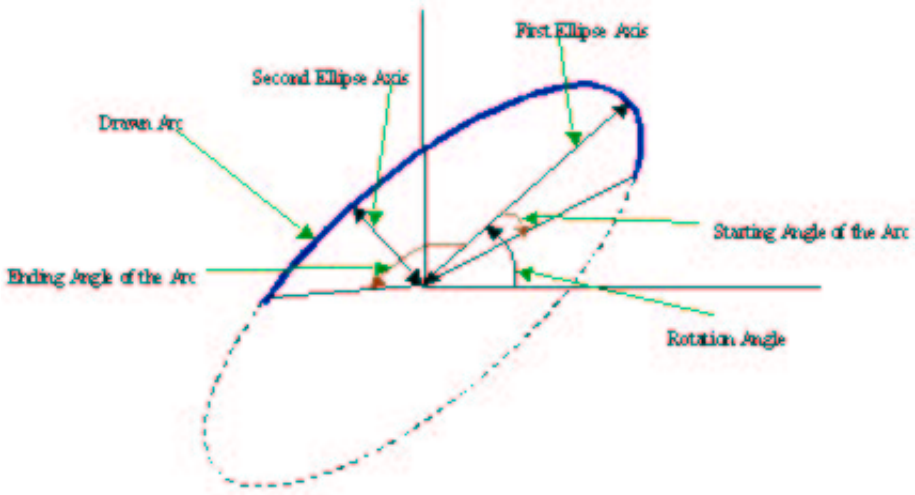
endAngle
: Ending angle of the elliptic arc.

color
: Ellipse color (RGB) or brightness (grayscale image).

thickness
: Thickness of the ellipse arc.

The function cvEllipse [p 147] draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by ROI rectangle. The 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 picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



EllipseAA

Draws antialiased elliptic arc

```
void cvEllipseAA( CvArr* img, CvPoint center, CvSize axes, double angle,
                  double startAngle, double endAngle, double color, int scale=0 );
```

img
: Image.

center
: Center of the ellipse.

axes
: Length of the ellipse axes.

angle
: Rotation angle.

startAngle
: Starting angle of the elliptic arc.

endAngle
: Ending angle of the elliptic arc.

color
: Ellipse color (RGB) or brightness (grayscale image).

scale
Specifies the number of fractional bits in the center coordinates and axes sizes.

The function cvEllipseAA [p 148] draws an antialiased elliptic arc. The arc is clipped by ROI rectangle. The 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.

FillPoly

Fills polygons interior

```
void cvFillPoly( CvArr* img, CvPoint** pts, int* npts, int contours, double color );
```

img
Image.
pts
Array of pointers to polygons.
npts
Array of polygon vertex counters.
contours
Number of contours that bind the filled region.
color
Polygon color (RGB) or brightness (grayscale image).

The function cvFillPoly [p 149] fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, etc.

FillConvexPoly

Fills convex polygon

```
void cvFillConvexPoly( CvArr* img, CvPoint* pts, int npts, double color );
```

img
Image.
pts
Array of pointers to a single polygon.
npts
Polygon vertex counter.
color
Polygon color (RGB) or brightness (grayscale image).

The function cvFillConvexPoly [p 149] fills convex polygon interior. This function is much faster than the function cvFillPoly [p 149] 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.

PolyLine

Draws simple or thick polygons

```
void cvPolyLine( CvArr* img, CvPoint** pts, int* npts, int contours, int isClosed,
                 double color, int thickness=1, int connectivity=8 );
```

img
: Image.

pts
: Array of pointers to polylines.

npts
: Array of polyline vertex counters.

contours
: Number of polyline contours.

isClosed
: 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.

connectivity
: The connectivity of polyline segments, 8 (by default) or 4.

The function cvPolyLine [p 150] draws a set of simple or thick polylines.

PolyLineAA

Draws antialiased polygons

```
void cvPolyLineAA( CvArr* img, CvPoint** pts, int* npts, int contours,
                   int isClosed, int color, int scale =0);
```

img
: Image.

pts
: Array of pointers to polylines.

npts
: Array of polyline vertex counters.

contours
: Number of polyline contours.

isClosed
: 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.

The function cvPolyLineAA [p 150] draws a set of antialiased polylines.

InitFont

Initializes font structure

```
void cvInitFont( CvFont* font, CvFontFace fontFace, float hscale,
                 float vscale, float italicScale, int thickness );
```

font
Pointer to the font structure initialized by the function.
fontFace
Font name identifier. Only the font `CV_FONT_VECTOR0` is currently supported.
hscale
Horizontal scale. If equal to `1.0f`, the characters have the original width depending on the font type. If equal to `0.5f`, the characters are of half the original width.
vscale
Vertical scale. If equal to `1.0f`, the characters have the original height depending on the font type. If equal to `0.5f`, the characters are of half the original height.
italicScale
Approximate tangent of the character slope relative to the vertical line. Zero value means a non-italic font, `1.0f` means ≈`45°` slope, etc. thickness Thickness of lines composing letters outlines. The function cvLine [p 145] is used for drawing letters.

The function cvInitFont [p 151] 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.

PutText

Draws text string

```
void cvPutText( CvArr* img, const char* text, CvPoint org, CvFont* font, int color );
```

img
Input image.
text
String to print.
org
Coordinates of the bottom-left corner of the first letter.

font
: Pointer to the font structure.

color
: Text color (RGB) or brightness (grayscale image).

The function cvPutText [p 151] 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.

GetTextSize

Retrieves width and height of text string

```
void cvGetTextSize( CvFont* font, const char* textString, CvSize* textSize, int* ymin );
```

font
: Pointer to the font structure.

textString
: Input string.

textSize
: Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.

ymin
: Lowest y coordinate of the text relative to the baseline. Negative, if the text includes such characters as g, j, p, q, y, etc., and zero otherwise.

The function cvGetTextSize [p 152] calculates the binding rectangle for the given text string when a specified font is used.

Gradients, Edges and Corners

Sobel

Calculates first, second, third or mixed image derivatives using extended Sobel operator

```
void cvSobel( const CvArr* I, CvArr* J, int dx, int dy, int apertureSize=3 );
```

I
: Source image.

J
: Destination image.

ox
: Order of the derivative x .

oy

Order of the derivative y .

apertureSize

Size of the extended Sobel kernel, must be 1, 3, 5 or 7. In all cases except 1, apertureSize × apertureSize separable kernel will be used to calculate the derivative. For `apertureSize`=1 3x1 or 1x3 kernel is used (Gaussian smoothing is not done). There is also special value `CV_SCHARR` (=-1) that corresponds to 3x3 Scharr filter that may give more accurate results than 3x3 Sobel. Scharr aperture is:

```
| -3 0  3|
|-10 0 10|
| -3 0  3|
```

for x-derivative or transposed for y-derivative.

The function cvSobel [p 152] calculates the image derivative by convolving the image with the appropriate kernel:

$$J(x,y) = d^{ox+oy} I/dx^{ox} \bullet dy^{oy} \quad |_{(x,y)}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less robust to the noise. Most often, the function is called with (ox=1, oy=0, apertureSize=3) or (ox=0, oy=1, apertureSize=3) to calculate first x- or y- image derivative. The first case corresponds to

```
  |-1  0  1|
  |-2  0  2|
  |-1  0  1|
```

kernel and the second one corresponds to

```
  |-1 -2 -1|
  | 0  0  0|
  | 1  2  1|
or
  | 1  2  1|
  | 0  0  0|
  |-1 -2 -1|
```

kernel, depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger by absolute value numbers than the source image. To avoid overflow, the function requires 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using cvConvertScale [p ??] or cvConvertScaleAbs [p 64] functions. Besides 8-bit images the function can process 32-bit floating-point images. Both source and destination must be single-channel images of equal size or ROI size.

Laplace

Calculates Laplacian of the image

```
void cvLaplace( const CvArr* I, CvArr* J, int apertureSize=3 );
```

I
: Source image.

J
: Destination image.

apertureSize
: Aperture parameter for Sobel operator (see cvSobel [p 152]).

The function cvLaplace [p 154] calculates Laplacian of the source image by summing second x- and y-derivatives calcualted using Sobel operator:

$$J(x,y) = d^2 I/dx^2 + d^2 I/dy^2$$

Specifying `apertureSize`=1 gives the fastest variant that is equal to convolving the image with the following kernel:

```
|0  1  0|
|1 -4  1|
|0  1  0|
```

As well as in cvSobel [p 152] function, no scaling is done and the same combinations of input and output formats are supported.

Canny

Implements Canny algorithm for edge detection

```
void cvCanny( const CvArr* img, CvArr* edges, double threshold1,
              double threshold2, int apertureSize=3 );
```

img
: Input image.

edges
: Image to store the edges found by the function.

threshold1
: The first threshold.

threshold2
: The second threshold.

apertureSize
: Aperture parameter for Sobel operator (see cvSobel [p 152]).

The function cvCanny [p 154] finds the edges on the input image `img` and marks them in the output image `edges` using the Canny algorithm. The smallest of `threshold1` and `threshold2` is used for edge linking, the largest - to find initial segments of strong edges.

PreCornerDetect

Calculates two constraint images for corner detection

```
void cvPreCornerDetect( const CvArr* img, CvArr* corners, int apertureSize=3 );
```

img
: Input image.

corners
: Image to store the corner candidates.

apertureSize
: Aperture parameter for Sobel operator (see cvSobel [p 152]).

The function cvPreCornerDetect [p 155] finds the corners on the input image `img` and stores them in the `corners` image in accordance with `Method 1` for corner detection desctibed in the guide.

CornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection

```
void cvCornerEigenValsAndVecs( const CvArr* I, CvArr* eigenvv,
                               int blockSize, int apertureSize=3 );
```

I
: Input image.

eigenvv
: Image to store the results. It must be 6 times wider than the input image.

blockSize
: Neighborhood size (see discussion).

apertureSize
: Aperture parameter for Sobel operator (see cvSobel [p 152]).

For every pixel the function `cvCornerEigenValsAndVecs` considers `blockSize` × `blockSize` neigborhood S(p). It calcualtes covariation matrix of derivatives over the neigborhood as:

$$M = \begin{vmatrix} \sum_{S(p)} (dI/dx)^2 & \sum_{S(p)} (dI/dx \cdot dI/dy) \\ \sum_{S(p)} (dI/dx \cdot dI/dy) & \sum_{S(p)} (dI/dy)^2 \end{vmatrix}$$

After that it finds eigenvectors and eigenvalues of the resultant matrix and stores them into destination image in form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$, where
λ_1, λ_2 - eigenvalues of `M`; not sorted

(x_1, y_1) - eigenvector corresponding to λ_1
(x_2, y_2) - eigenvector corresponding to λ_2

CornerMinEigenVal

Calculates minimal eigenvalue of image blocks for corner detection

```
void cvCornerMinEigenVal( const CvArr* img, CvArr* eigenvv, int blockSize, int apertureSize=3 );
```

img
: Input image.

eigenvv
: Image to store the minimal eigen values. Should have the same size as `img`

blockSize
: Neighborhood size (see discussion of cvCornerEigenValsAndVecs [p 155]).

apertureSize
: Aperture parameter for Sobel operator (see cvSobel [p 152]). format. In the case of floating-point input format this parameter is the number of the fixed float filter used for differencing.

The function cvCornerMinEigenVal [p 156] is similar to cvCornerEigenValsAndVecs [p 155] but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. $\min(\lambda_1, \lambda_2)$ in terms of the previous function.

FindCornerSubPix

Refines corner locations

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

I
: Input image.

corners
: Initial coordinates of the input corners and refined coordinates on output.

count
: Number of corners.

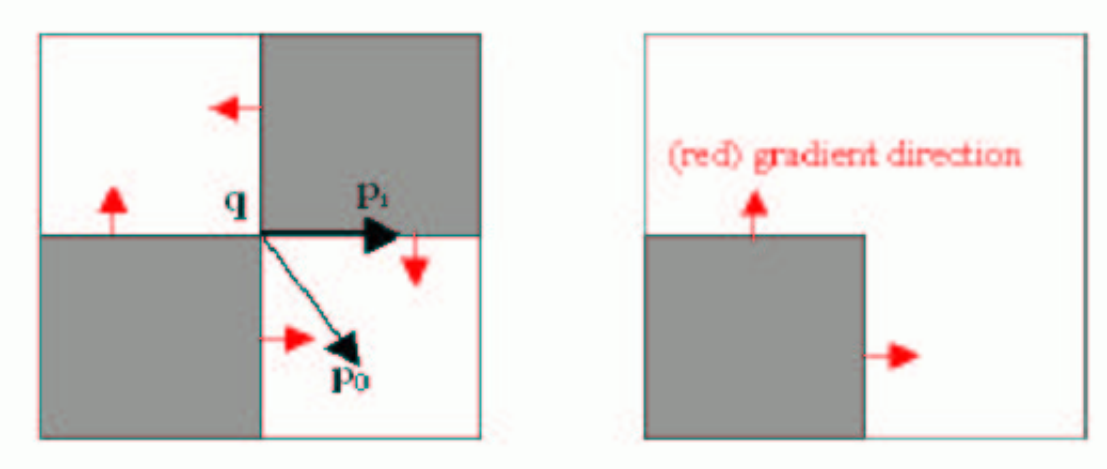
win
: Half sizes of the search window. For example, if `win`=(5,5) then 5*2+1 × 5*2+1 = 11 × 11 search window is used.

zeroZone
: Half size of the dead region in the middle of the search zone over which the summation in formulae below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size.

criteria

Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after certain number of iteration or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy.

The function cvFindCornerSubPix [p 156] iterates to find the sub-pixel accurate location of a corner, or "radial saddle point", as shown in on the picture below.



Sub-pixel accurate corner (radial saddle point) locator is based on the observation that any vector from q to p is orthogonal to the image gradient.

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:

$$\varepsilon_i = DI_{p_i}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ε_i is minimized. A system of equations may be set up with ε_i' set to zero:

$$\text{sum}_i (DI_{p_i} \cdot DI_{p_i}^T) \cdot q - \text{sum}_i (DI_{p_i} \cdot DI_{p_i}^T \cdot p_i) = 0$$

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.

GoodFeaturesToTrack

Determines strong corners on image

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

image
: The source 8-bit or floating-point 32-bit, single-channel image.

eigImage
: Temporary floating-point 32-bit image of the same size as `image`.

tempImage
: Another temporary image of the same size and same format as `eigImage`.

corners
: Output parameter. Detected corners.

cornerCount
: Output parameter. Number of detected corners.

qualityLevel
: Multiplier for the maxmin eigenvalue; specifies minimal accepted quality of image corners.

minDistance
: Limit, specifying minimum possible distance between returned corners; Euclidian distance is used.

The function cvGoodFeaturesToTrack [p 158] finds corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using cvCornerMinEigenVal [p 156] function and stores them in `eigImage`. Then it performs non-maxima suppression (only local maxima in 3x3 neighborhood remain). The next step is rejecting the corners with the minimal eigenvalue less than `qualityLevel`•max(`eigImage`(x,y)). Finally, the function ensures that all the corners found are distanced enough from one another by considering the corners (the most strongest corners are considered first) and checking that the distance between the newly considered feature and the features considered earlier is larger than `minDistance`. So, the function removes the features than are too close to the stronger features.

Sampling, Interpolation and Geometrical Transforms

InitLineIterator

Initializes line iterator

```
int cvInitLineIterator( const CvArr* img, CvPoint pt1, CvPoint pt2,
                        CvLineIterator* lineIterator, int connectivity=8 );
```

img
: Image.

pt1
: Starting the line point.

pt2
: Ending the line point.

lineIterator
: Pointer to the line iterator state structure.

connectivity
: The scanned line connectivity, 4 or 8.

The function cvInitLineIterator [p 158] initializes the line iterator and returns the number of pixels between two end points. Both points must be inside the image. After the iterator has been initialized, all the points on the raster line that connects the two ending points may be retrieved by successive calls of `CV_NEXT_LINE_POINT` point. The points on the line are calculated one by one using 4-connected or 8-connected Bresenham algorithm.

Example. Using line iterator to calculate pixel values along the color line

```
CvScalar sum_line_pixels( IplImage* img, CvPoint pt1, CvPoint pt2 )
{
    CvLineIterator iterator;
    int blue_sum = 0, green_sum = 0, red_sum = 0;
    int count = cvInitLineIterator( img, pt1, pt2, &iterator, 8 );

    for( int i = 0; i < count; i++ ){
        blue_sum += iterator.ptr[0];
        green_sum += iterator.ptr[1];
        red_sum += iterator.ptr[2];
        CV_NEXT_LINE_POINT(iterator);

        /* print the pixel coordinates: demonstrates how to calculate the coordinates */
        {
        int offset, x, y;
        /* assume that ROI is not set, otherwise need to take it into account. */
        offset = iterator.ptr - (uchar*)(img->imageData);
        y = offset/img->widthStep;
        x = (offset - y*img->widthStep)/(3*sizeof(uchar) /* size of pixel */);
        printf("(%d,%d)\n", x, y );
        }
    }
    return cvScalar( blue_sum, green_sum, red_sum );
}
```

SampleLine

Reads raster line to buffer

```
int cvSampleLine( const CvArr* img, CvPoint pt1, CvPoint pt2,
                  void* buffer, int connectivity=8 );
```

img
: Image.

pt1
: Starting the line point.

pt2
: Ending the line point.

buffer
: Buffer to store the line points; must have enough size to store max(|`pt2.x-pt1.x`|+1, |`pt2.y-pt1.y`|+1) points in case of 8-connected line and |`pt2.x-pt1.x`|+|`pt2.y-pt1.y`|+1 in case of 4-connected line.

connectivity
: The line connectivity, 4 or 8.

The function cvSampleLine [p 159] implements a 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.

GetRectSubPix

Retrieves pixel rectangle from image with sub-pixel accuracy

```
void cvGetRectSubPix( const CvArr* I, CvArr* J, CvPoint2D32f center );
```

I
: Source image.

J
: Extracted rectangle.

center
: Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image.

The function cvGetRectSubPix [p 160] extracts pixels from `I`:

```
J( x+width(J)/2, y+height(J)/2 )=I( x+center.x, y+center.y )
```

where the values of pixels at non-integer coordinates (x+center.x, y+center.y) are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. Whereas the rectangle center must be inside the image, the whole rectangle may be partially occluded. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

GetQuadrangeSubPix

Retrieves pixel quadrangle from image with sub-pixel accuracy

```
void cvGetQuadrangeSubPix( const CvArr* I, CvArr* J, const CvArr* M,
                           int fillOutliers=0, CvScalar fillValue=cvScalarAll(0) );
```

I
: Source image.

J
: Extracted quadrangle.

M
: The transformation 3 × 2 matrix [A|b] (see the discussion).

fillOutliers
: The flag indicating whether to interpolate values of pixel taken from outside of the source image using replication mode (`fillOutliers`=0) or set them a fixed value (`fillOutliers`=1).

fillValue
: The fixed value to set the outlier pixels to if `fillOutliers`=1.

The function cvGetQuadrangleSubPix [p ??] extracts pixels from `I` at sub-pixel accuracy and stores them to `J` as follows:

$$J(x+width(J)/2, y+height(J)/2)= I(A_{11} x+A_{12} y+b_1 , A_{21} x+A_{22} y+b_2),$$

where A and b are taken from M

$$M = \begin{vmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{vmatrix}$$

where the values of pixels at non-integer coordinates $A \bullet (x,y)^T + b$ are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently.

Example. Using cvGetQuadrangeSubPix for image rotation.

```
#include "cv.h"
#include "highgui.h"
#include "math.h"

int main( int argc, char** argv )
{
    IplImage* src;
    /* the first command line parameter must be image file name */
    if( argc==2 && (src = cvLoadImage(argv[1], -1))!=0)
    {
        IplImage* dst = cvCloneImage( src );
        int delta = 1;
        int angle = 0;

        cvNamedWindow( "src", 1 );
        cvShowImage( "src", src );

        for(;;)
        {
            float m[6];
            double factor = (cos(angle*CV_PI/180.) + 1.1)*3;
            CvMat M = cvMat( 2, 3, CV_32F, m );
            int w = src->width;
            int h = src->height;

            m[0] = (float)(factor*cos(-angle*2*CV_PI/180.));
```

```
            m[1] = (float)(factor*sin(-angle*2*CV_PI/180.));
            m[2] = w*0.5f;
            m[3] = -m[1];
            m[4] = m[0];
            m[5] = h*0.5f;

            cvGetQuadrangleSubPix( src, dst, &M, 1, cvScalarAll(0));

            cvNamedWindow( "dst", 1 );
            cvShowImage( "dst", dst );

            if( cvWaitKey(5) == 27 )
                break;

            angle = (angle + delta) % 360;
        }
    }
    return 0;
}
```

Resize

Resizes image

```
void cvResize( const CvArr* I, CvArr* J, int interpolation=CV_INTER_LINEAR );
```

I
: Source image.

J
: Destination image.

interpolation
: Interpolation method:
  - CV_INTER_NN - nearest-neigbor interpolation,
  - CV_INTER_LINEAR - bilinear interpolation (used by default)

The function cvResize [p 162] resizes image `I` so that it fits exactly to `J`. If ROI is set, the function consideres the ROI as supported as usual. the source image using the specified structuring element B that determines the shape of a pixel neighborhood over which the minimum is taken:

$C=erode(A,B): C(I)=\min_{(K \text{ in } B_I)} A(K)$

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 the channels.

Morphological Operations

CreateStructuringElementEx

Creates structuring element

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

nCols
: Number of columns in the structuring element.

nRows
: Number of rows in the structuring element.

anchorX
: Relative horizontal offset of the anchor point.

anchorY
: Relative vertical offset of the anchor point.

shape
: Shape of the structuring element; may have the following values:
  - `CV_SHAPE_RECT` , a rectangular element;
  - `CV_SHAPE_CROSS` , a cross-shaped element;
  - `CV_SHAPE_ELLIPSE` , an elliptic element;
  - `CV_SHAPE_CUSTOM` , a user-defined element. In this case the parameter `values` specifies the mask, that is, which neighbors of the pixel must be considered.

values
: Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non-zero values indicate points that belong to the element. If the pointer is `NULL` , then all values are considered non-zero, that is, the element is of a rectangular shape. This parameter is considered only if the shape is `CV_SHAPE_CUSTOM` .

The function cv CreateStructuringElementEx [p ??] allocates and fills the structure `IplConvKernel` , which can be used as a structuring element in the morphological operations.

ReleaseStructuringElement

Deletes structuring element

```
void cvReleaseStructuringElement( IplConvKernel** ppElement );
```

ppElement
: Pointer to the deleted structuring element.

The function cv ReleaseStructuringElement [p ??] releases the structure `IplConvKernel` that is no longer needed. If `*ppElement` is `NULL` , the function has no effect. The function returns created structuring element.

Erode

Erodes image by using arbitrary structuring element

```
void cvErode( const CvArr* A, CvArr* C, IplConvKernel* B=0, int iterations=1 );
```

A
Source image.
C
Destination image.
B
Structuring element used for erosion. If it is NULL, a 3×3 rectangular structuring element is used.
iterations
Number of times erosion is applied.

The function cvErode [p 164] erodes the source image using the specified structuring element B that determines the shape of a pixel neighborhood over which the minimum is taken:

$C=erode(A,B): \quad C(x,y)=\min_{((x',y') \text{ in } B(x,y))} A(x',y')$

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 the channels.

Dilate

Dilates image by using arbitrary structuring element

```
void cvDilate( const CvArr* A, CvArr* C, IplConvKernel* B=0, int iterations=1 );
```

A
Source image.
C
Destination image.
B
Structuring element used for erosion. If it is NULL, a 3×3 rectangular structuring element is used.
iterations
Number of times erosion is applied.

The function cvDilate [p 164] dilates the source image using the specified structuring element B that determines the shape of a pixel neighborhood over which the maximum is taken:

$C=dilate(A,B): \quad C(x,y)=\max_{((x',y') \text{ in } B(x,y))} A(x',y')$

The function supports the in-place mode when the source and destination pointers are the same. Dilation can be applied several times `iterations` parameter. Dilation on a color image means independent transformation of all the channels.

MorphologyEx

Performs advanced morphological transformations

```
void cvMorphologyEx( const CvArr* A, CvArr* C, CvArr* temp,
                     IplConvKernel* B, CvMorphOp op, int iterations );
```

A
: Source image.

C
: Destination image.

temp
: Temporary image, required in some cases.

B
: Structuring element.

op
: Type of morphological operation (see the discussion).

iterations
: Number of times erosion and dilation are applied.

The function cvMorphologyEx [p 165] performs advanced morphological transformations using on erosion and dilation as basic operations.

```
Opening:
C=open(A,B)=dilate(erode(A,B),B),   if op=CV_MOP_OPEN

Closing:
C=close(A,B)=erode(dilate(A,B),B),  if op=CV_MOP_CLOSE

Morphological gradient:
C=morph_grad(A,B)=dilate(A,B)-erode(A,B),  if op=CV_MOP_GRADIENT

"Top hat":
C=tophat(A,B)=A-erode(A,B),   if op=CV_MOP_TOPHAT

"Black hat":
C=blackhat(A,B)=dilate(A,B)-A,   if op=CV_MOP_BLACKHAT
```

The temporary image `temp` is required if `op=CV_MOP_GRADIENT` or if `A=C` (inplace operation) and `op=CV_MOP_TOPHAT` or `op=CV_MOP_BLACKHAT`

Filters and Color Conversion

Smooth

Smooths the image in one of several ways

```
void cvSmooth( const CvArr* src, CvArr* dst,
               int smoothtype=CV_GAUSSIAN,
               int param1=3, int param2=0 );
```

src
: The source image.

dst
: The destination image.

smoothtype
: Type of the smoothing:
  - CV_BLUR_NO_SCALE (simple blur with no scaling) - summation over a pixel `param1`×`param2` neighborhood. If the neighborhood size is not fixed, one may use cvIntegral [p 167] function.
  - CV_BLUR (simple blur) - summation over a pixel `param1`×`param2` neighborhood with subsequent scaling by 1/(`param1`•`param2`).
  - CV_GAUSSIAN (gaussian blur) - convolving image with `param1`×`param2` Gaussian.
  - CV_MEDIAN (median blur) - finding median of `param1`×`param1` neighborhood (i.e. the neighborhood is square).
  - CV_BILATERAL (bilateral filter) - applying bilateral 3x3 filtering with color sigma=`param1` and space sigma=`param2`. Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1
: The first parameter of smoothing operation.

param2
: The second parameter of smoothing operation. In case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1`.

The function cvSmooth [p 166] smooths image using one of several methods. Every of the methods has some features and restrictions listed below

Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to cvSobel [p 152] and cvLaplace [p 154]) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

Integral

Calculates integral images

```
void cvIntegral( const CvArr* I, CvArr* S, CvArr* Sq=0, CvArr* T=0 );
```

I
: The source image, w×h, single-channel, 8-bit, or floating-point (32f or 64f).

S
: The sum image, w+1×h+1, single-channel, 32-bit integer or double precision floating-point (64f).

Sq
: The square sum image, w+1×h+1, single-channel, double precision floating-point (64f).

T
: The tilted sum image (sum of rotated by 45° image), w+1×h+1, single-channel, the same data type as sum.

The function cvIntegral [p 167] calculates one or more integral images for the source image as following:

$$S(X,Y)=\sum_{x<X,\,y<Y} I(x,y)$$

$$Sq(X,Y)=\sum_{x<X,\,y<Y} I(x,y)^2$$

$$T(X,Y)=\sum_{y<Y,\,abs(x-X)<y} I(x,y)$$

After that the images are calculated, they can be used to calculate sums of pixels over an arbitrary rectangles, for example:

$$\sum_{x1<=x<x2,\,y1<=y<y2} I(x,y)=S(x2,y2)-S(x1,y2)-S(x2,y1)+S(x1,x1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size etc.

CvtColor

Converts image from one color space to another

```
void cvCvtColor( const CvArr* src, CvArr* dst, int code );
```

src
: The source 8-bit image.

dst
: The destination 8-bit image.

code
: Color conversion operation that can be specifed using CV_<src_color_space>2<dst_color_space> constants (see below).

The function cvCvtColor [p 167] converts input image from one color space to another. The function ignores `colorModel` and `channelSeq` fields of `IplImage` header, so the source image color space should be specified correctly (including order of the channels in case of RGB space, e.g. BGR means 24-bit format with $B_0\ G_0\ R_0\ B_1\ G_1\ R_1$ ... layout, whereas RGB means 24-format with $R_0\ G_0\ B_0\ R_1\ G_1\ B_1$ ... layout). The function can do the following transformations:

- Transformations within RGB space like adding/removing alpha channel, reversing the channel order, conversion to/from 16-bit (Rx5:Gx6:Rx5) color, as well as conversion to/from grayscale using:

  ```
  RGB[A]->Gray: Y=0.212671*R + 0.715160*G + 0.072169*B + 0*A
  Gray->RGB[A]: R=Y G=Y B=Y A=0
  ```

  All the possible combinations of input and output format (except equal) are allowed here.

- RGB<=>XYZ (CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB):

  ```
  |X|   |0.412411  0.357585  0.180454| |R|
  |Y| = |0.212649  0.715169  0.072182|*|G|
  |Z|   |0.019332  0.119195  0.950390| |B|

  |R|   | 3.240479  -1.53715  -0.498535| |X|
  |G| = |-0.969256   1.875991  0.041556|*|Y|
  |B|   | 0.055648  -0.204043  1.057311| |Z|
  ```

- RGB<=>YCrCb (CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB)

  ```
  Y=0.299*R + 0.587*G + 0.114*B
  Cr=(R-Y)*0.713 + 128
  Cb=(B-Y)*0.564 + 128

  R=Y + 1.403*(Cr - 128)
  G=Y - 0.344*(Cr - 128) - 0.714*(Cb - 128)
  B=Y + 1.773*(Cb - 128)
  ```

- RGB=>HSV (CV_BGR2HSV,CV_RGB2HSV)

  ```
  V=max(R,G,B)
  S=(V-min(R,G,B))*255/V   if V!=0, 0 otherwise

        (G - B)*60/S,  if V=R
  H= 180+(B - R)*60/S,  if V=G
     240+(R - G)*60/S,  if V=B

  if H<0 then H=H+360
  ```

  The hue values calcualted using the above formulae vary from 0° to 360° so they are divided by 2 to fit into 8-bit destination format.

- RGB=>Lab (CV_BGR2Lab, CV_RGB2Lab)

```
|X|   |0.433910  0.376220  0.189860| |R/255|
|Y| = |0.212649  0.715169  0.072182|*|G/255|
|Z|   |0.017756  0.109478  0.872915| |B/255|

L = 116*Y^1/3       for Y>0.008856
L = 903.3*Y      for Y<=0.008856

a = 500*(f(X)-f(Y))
b = 200*(f(Y)-f(Z))
where f(t)=t^1/3              for t>0.008856
      f(t)=7.787*t+16/116   for t<=0.008856
```

The above formulae have been taken from
http://www.cica.indiana.edu/cica/faq/color_spaces/color.spaces.html

- Bayer=>RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerRG2BGR, CV_BayerGB2RGB, CV_BayerGR2BGR, CV_BayerRG2RGB, CV_BayerBG2BGR, CV_BayerGR2RGB, CV_BayerGB2BGR)

  Bayer pattern is widely used in CCD and CMOS cameras. It allows to get color picture out of a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

R	G	R	G	R
G	B	G	B	G
R	G	R	G	R
G	B	G	B	G
R	G	R	G	R
G	B	G	B	G

  The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants CV_Bayer$C_1C_2$2{BGR|RGB} indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

Threshold

Applies fixed-level threshold to array elements

```
void cvThreshold( const CvArr* src, CvArr* dst, double threshold,
                  double maxValue, int thresholdType );
```

src
: Source array (single-channel, 8-bit of 32-bit floating point).

dst
: Destination array; must be either the same type as `src` or 8-bit.

threshold
: Threshold value.

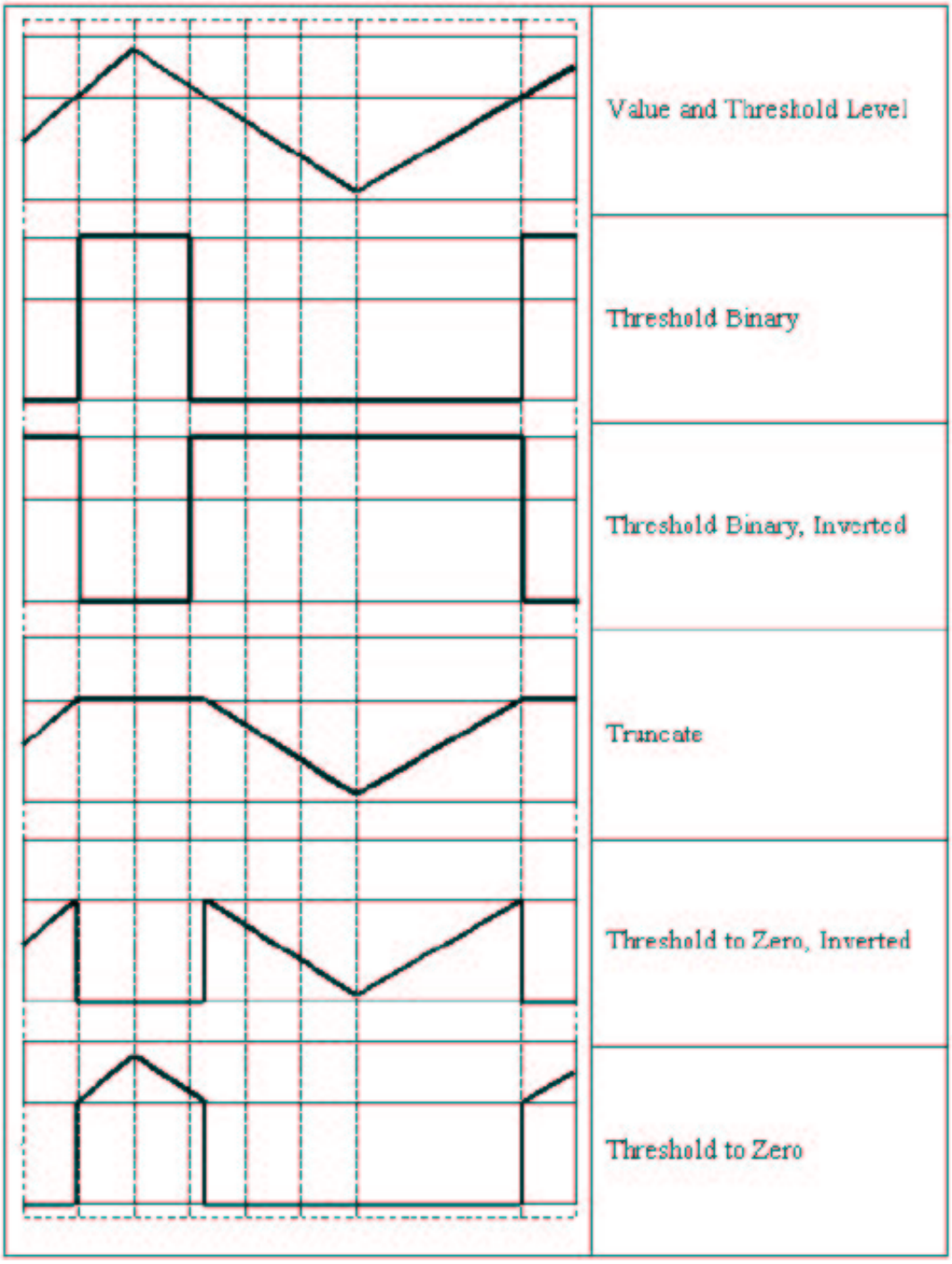
maxValue
: Maximum value to use with `CV_THRESH_BINARY`, `CV_THRESH_BINARY_INV`, and `CV_THRESH_TRUNC` thresholding types.

thresholdType
: Thresholding type (see the discussion)

The function cvThreshold [p 169] applies fixed-level thresholding to single-channel array. The function is typically used to get bi-level (binary) image out of grayscale image or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding the function supports that are determined by `thresholdType`:

```
thresholdType=CV_THRESH_BINARY:
dst(x,y) = maxValue, if src(x,y)>threshold
           0, otherwise

thresholdType=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>threshold
           maxValue, otherwise

thresholdType=CV_THRESH_TRUNC:
dst(x,y) = threshold, if src(x,y)>threshold
           src(x,y), otherwise

thresholdType=CV_THRESH_TOZERO:
dst(x,y) = src(x,y), if (x,y)>threshold
           0, otherwise

thresholdType=CV_THRESH_TOZERO_INV:
dst(x,y) = 0, if src(x,y)>threshold
           src(x,y), otherwise
```

And this is the visual description of thresholding types:


Value and Threshold Level
Threshold Binary
Threshold Binary, Inverted
Truncate
Threshold to Zero, Inverted
Threshold to Zero

AdaptiveThreshold

Applies adaptive threshold to array

```
void cvAdaptiveThreshold( const CvArr* src, CvArr* dst, double maxValue,
                          int adaptiveMethod, int thresholdType,
                          int blockSize, double param1 );
```

src
: Source image.

dst
: Destination image.

maxValue
: Maximum value that is used with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV`.

adaptiveMethod
: Adaptive thresholding algorithm to use: `CV_ADAPTIVE_THRESH_MEAN_C` or `CV_ADAPTIVE_THRESH_GAUSSIAN_C` (see the discussion).

thresholdType
: Thresholding type; must be one of
  - `CV_THRESH_BINARY,`
  - `CV_THRESH_BINARY_INV,`

blockSize
: The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, ...

param1
: The method-dependent parameter. For the methods `CV_ADAPTIVE_THRESH_MEAN_C` and `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a constant subtracted from mean or weighted mean (see the discussion), though it may be negative.

The function cvAdaptiveThreshold [p 172] transforms grayscale image to binary image according to the formulae:

```
thresholdType=CV_THRESH_BINARY:
dst(x,y) = maxValue, if src(x,y)>T(x,y)
           0, otherwise

thresholdType=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>T(x,y)
           maxValue, otherwise
```

where T_I is a threshold calculated individually for each pixel.

For the method `CV_ADAPTIVE_THRESH_MEAN_C` it is a mean of `blockSize` × `blockSize` pixel neighborhood, subtracted by `param1`.

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a weighted sum (gaussian) of `blockSize` × `blockSize` pixel neighborhood, subtracted by `param1`.

LUT

Performs look-up table transformation on image

```
CvMat* cvLUT( const CvArr* A, CvArr* B, const CvArr* lut );
```

A
: Source array of 8-bit elements.

B
: Destination array of arbitrary depth and of the same number of channels as the source array.

lut
: Look-up table of 256 elements; should be of the same depth as the destination array.

The function cvLUT [p 173] fills the destination array with values of look-up table entries. Indices of the entries are taken from the source array. That is, the function processes each pixel as follows:

```
B(x,y)=lut[A(x,y)+Δ]
```

where Δ is 0 for 8-bit `unsigned` source image type and 128 for 8-bit `signed` source image type.

Pyramids and the Applications

PyrDown

Downsamples image

```
void cvPyrDown( const CvArr* src, CvArr* dst, int filter=CV_GAUSSIAN_5x5 );
```

src
: The source image.

dst
: The destination image, should have 2x smaller width and height than the source.

filter
: Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported.

The function cvPyrDown [p 173] 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.

PyrUp

Upsamples image

```
void cvPyrUp( const CvArr* src, CvArr* dst, int filter=CV_GAUSSIAN_5x5 );
```

src
: The source image.

dst
: The destination image, should have 2x smaller width and height than the source.

filter
: Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported.

The function cvPyrUp [p 173] performs up-sampling 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.

PyrSegmentation

Implements image segmentation by pyramids

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

src
: The source image.

dst
: The destination image.

storage
: Storage; stores the resulting sequence of connected components.

comp
: Pointer to the output sequence of the segmented components.

level
: Maximum level of the pyramid for the segmentation.

threshold1
: Error threshold for establishing the links.

threshold2
: Error threshold for the segments clustering.

The function cvPyrSegmentation [p 174] implements image segmentation by pyramids. The pyramid builds up to the level `level`. The links between any pixel `a` on level `i` and its candidate father pixel `b` on the adjacent level are established if
`p(c(a),c(b))<threshold1`. After the connected components are defined, they are joined into several clusters. Any two segments A and B belong to the same cluster, if
`p(c(A),c(B))<threshold2`. The input image has only one channel, then
$p(c^1,c^2)=|c^1-c^2|$. If the input image has three channels (red, green and blue), then
$p(c^1,c^2)=0,3\cdot(c^1{}_r-c^2{}_r)+0,59\cdot(c^1{}_g-c^2{}_g)+0,11\cdot(c^1{}_b-c^2{}_b)$. There may be more than one connected component per a cluster.
The images `src` and `dst` should be 8-bit single-channel or 3-channel images or equal size

Connected components

CvConnectedComp

Connected component

```
    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;
```

FloodFill

Fills a connected component with given color

```
void cvFloodFill( CvArr* img, CvPoint seed, double newVal,
                  double lo=0, double up=0, CvConnectedComp* comp=0,
                  int flags=4, CvArr* mask=0 );
#define CV_FLOODFILL_FIXED_RANGE (1 << 16)
#define CV_FLOODFILL_MASK_ONLY   (1 << 17)
```

img
: Input image, either 1-,3-channel 8-bit, or single-channel floating-point image. It is modified by the function unless CV_FLOODFILL_MASK_ONLY flag is set (see below).

seed
: Coordinates of the seed point inside the image ROI.

newVal
: New value of repainted domain pixels. For 8-bit color images it is a packed color (e.g. using `CV_RGB` macro).

lo
: Maximal lower brightness/color difference between the currently observed pixel and one of its neighbor belong to the component or seed pixel to add the pixel to component. In case of 8-bit color images it is packed value.

up
: Maximal upper brightness/color difference between the currently observed pixel and one of its neighbor belong to the component or seed pixel to add the pixel to component. In case of 8-bit color images it is packed value.

comp
: Pointer to structure the function fills with the information about the repainted domain.

flags
: The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or combination of the following flags:
    - CV_FLOODFILL_FIXED_RANGE - if set the difference between the current pixel and seed

pixel is considered, otherwise difference between neighbor pixels is considered (the range is floating).

- CV_FLOODFILL_MASK_ONLY - if set, the function does not fill the image (`newVal` is ignored), but the fills mask (that must be non-NULL in this case).

mask

Operation mask, should be singe-channel 8-bit image, 2 pixels wider and 2 pixels taller than `img`. If not NULL, the function uses and updates the mask, so user takes responsibility of initializing `mask` content. Floodfilling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. Or it is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap.

The function cvFloodFill [p 175] fills a connected component starting from the seed pixel where all pixels within the component have close to each other values (prior to filling). The pixel is considered to belong to the repainted domain if its value `I(x,y)` meets the following conditions (the particular cases are specifed after commas):

```
I(x',y')-lo<=I(x,y)<=I(x',y')+up, grayscale image + floating range
I(seed.x,seed.y)-lo<=I(x,y)<=I(seed.x,seed.y)+up, grayscale image + floating range

I(x',y')r -lor <=I(x,y)r <=I(x',y')r +upr  and
I(x',y')g -log <=I(x,y)g <=I(x',y')g +upg  and
I(x',y')b -lob <=I(x,y)b <=I(x',y')b +upb , color image + floating range

I(seed.x,seed.y)r -lor <=I(x,y)r <=I(seed.x,seed.y)r +upr  and
I(seed.x,seed.y)g -log <=I(x,y)g <=I(seed.x,seed.y)g +upg  and
I(seed.x,seed.y)b -lob <=I(x,y)b <=I(seed.x,seed.y)b +upb , color image + fixed range
```

where `I(x',y')` is value of one of pixel neighbors (to be added to the connected component in case of floating range, a pixel should have at least one neigbor with similar brightness)

FindContours

Finds contours in binary image

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

image

The source 8-bit single channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - that is image treated as `binary`. To get such a binary image from grayscale, one may use cvThreshold [p 169] , cvAdaptiveThreshold [p 172] or cvCanny [p 154] . The function modifies the source image content.

storage

Container of the retrieved contours.

firstContour

Output parameter, will contain the pointer to the first outer contour.

headerSize
Size of the sequence header, >=sizeof(CvChain [p ??]) if `method`=CV_CHAIN_CODE, and >=sizeof(CvContour) otherwise.

mode
Retrieval mode.

- `CV_RETR_EXTERNAL`retrives only the extreme outer contours
- `CV_RETR_LIST`retrieves all the contours and puts them in the list
- `CV_RETR_CCOMP`retrieves all the contours and organizes them into two-level hierarchy: top level are external boundaries of the components, second level are bounda boundaries of the holes
- `CV_RETR_TREE`retrieves all the contours and reconstructs the full hierarchy of nested contours

method
Approximation method.

- `CV_CHAIN_CODE`outputs contours in the Freeman chain code. All other methods output polygons (sequences of vertices).
- `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, the function leaves only their ending points;
- `CV_CHAIN_APPROX_TC89_L1`, `CV_CHAIN_APPROX_TC89_KCOS` applies one of the flavors of Teh-Chin chain approximation algorithm.
- `CV_LINK_RUNS` uses completely different (from the previous methods) algorithm - linking of horizontal segments of 1's. Only `CV_RETR_LIST` retrieval mode is allowed by the method.

The function cvFindContours [p 176] retrieves contours from the binary image and returns the number of retrieved contours. The pointer `firstContour` is filled by the function. It will contain pointer to the first most outer contour or NULL if no contours is detected (if the image is completely black). Other contours may be reached from `firstContour` using `h_next` and `v_next` links. The sample in cvDrawContours [p 179] discussion shows how to use contours for connected component detection. Contours can be also used for shape analysis and object recognition - see `squares` sample in CVPR 2001 tutorial course located at SourceForge site.

StartFindContours

Initializes contour scanning process

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

image
The source 8-bit single channel binary image.

storage
Container of the retrieved contours.

headerSize
: Size of the sequence header, >=sizeof(CvChain [p ??]) if `method`=CV_CHAIN_CODE, and >=sizeof(CvContour) otherwise.

mode
: Retrieval mode, has the same meaning as in cvFindContours [p 176] .

method
: Approximation method, the same as in cvFindContours [p 176] except that CV_LINK_RUNS can not be used here.

The function cvStartFindContours [p 177] initializes and returns pointer to the contour scanner. The scanner is used further in cvFindNextContour [p 178] to retrieve the rest of contours.

FindNextContour

Finds next contour in the image

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

scanner
: Contour scanner initialized by the function cvStartFindContours [p 177] .

The function cvFindNextContour [p 178] locates and retrieves the next contour in the image and returns pointer to it. The function returns NULL, if there is no more contours.

SubstituteContour

Replaces retrieved contour

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

scanner
: Contour scanner initialized by the function cvStartFindContours .

newContour
: Substituting contour.

The function cvSubstituteContour [p 178] replaces the retrieved contour, that was returned from the preceding call of the function cvFindNextContour [p 178] 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`=NULL, the retrieved contour is not included into the resulting structure, nor all of its children that might be added to this structure later.

EndFindContours

Finishes scanning process

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

scanner
: Pointer to the contour scanner.

The function cvEndFindContours [p 179] finishes the scanning process and returns the pointer to the first contour on the highest level.

DrawContours

Draws contour outlines or interiors in the image

```
void cvDrawContours( CvArr *image, CvSeq* contour,
                     double external_color, double hole_color,
                     int max_level, int thickness=1,
                     int connectivity=8 );
```

image
: Image where the contours are to be drawn. Like in any other drawing function, the contours are clipped with the ROI.

contour
: Pointer to the first contour.

externalColor
: Color to draw external contours with.

holeColor
: Color to draw holes with.

maxLevel
: Maximal level for drawn contours. If 0, only `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. If the value is negative, the function does not draw the contours following after `contour` but draws child contours of `contour` up to abs(`maxLevel`)-1 level.

thickness
: Thickness of lines the contours are drawn with. If it is negative (e.g. =CV_FILLED), the contour interiors are drawn.

connectivity
: Connectivity of line segments of the contour outlines.

The function cvDrawContours [p 179] draws contour outlines in the image if `thickness`>=0 or fills area bounded by the contours if `thickness`<0.

Example. Connected component detection via contour functions

```
#include "cv.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    IplImage* src;
    // the first command line parameter must be file name of binary (black-n-white) image
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* contour = 0;

        cvThreshold( src, src, 1, 255, CV_THRESH_BINARY );
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvFindContours( src, storage, &contour, sizeof(CvContour), CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );
        cvZero( dst );

        for( ; contour != 0; contour = contour->h_next )
        {
            int color = CV_RGB( rand(), rand(), rand() );
            /* replace CV_FILLED with 1 to see the outlines */
            cvDrawContours( dst, contour, color, color, -1, CV_FILLED, 8 );
        }

        cvNamedWindow( "Components", 1 );
        cvShowImage( "Components", dst );
        cvWaitKey(0);
    }
}
```

Replace CV_FILLED with 1 in the sample below to see the contour outlines

---

Image and contour moments

Moments

Calculates all moments up to third order of a polygon or rasterized shape

```
void cvMoments( const CvArr* arr, CvMoments* moments, int isBinary=0 );
```

arr
: Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points of a vector of points).

moments
: Pointer to returned moment state structure.

isBinary
: (For images only) If the flag is non-zero, all the zero pixel values are treated as zeroes, all the others are treated as ones.

The function cvMoments [p 180] calculates spatial and central moments up to the third order and writes them to `moments`. The moments may be used then to calculate gravity center of the shape, its area, main axises and various shape characeteristics including 7 Hu invariants.

GetSpatialMoment

Retrieves spatial moment from moment state structure

```
double cvGetSpatialMoment( CvMoments* moments, int j, int i );
```

moments
: The moment state, calculated by cvMoments [p 180] .

j
: x-order of the retrieved moment, j >= 0.

i
: y-order of the retrieved moment, i >= 0 and i + j <= 3.

The function cvGetSpatialMoment [p 181] retrieves the spatial moment, which in case of image moments is defined as:

$M_{ji} = sum_{x,y}(I(x,y) \bullet x^j \bullet y^i)$

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

GetCentralMoment

Retrieves central moment from moment state structure

```
double cvGetCentralMoment( CvMoments* moments, int j, int i );
```

moments
: Pointer to the moment state structure.

j
: x-order of the retrieved moment, j >= 0.

i
: y-order of the retrieved moment, i >= 0 and i + j <= 3.

The functioncvGetCentralMoment [p 181] retrieves the central moment, which in case of image moments is defined as:

$\mu_{ij} = sum_{x,y}(I(x,y) \bullet (x-x_c)^j \bullet (y-y_c)^i)$,

where $x_c = M_{10}/M_{00}$, $y_c = M_{01}/M_{00}$ - coordinates of the gravity center

GetNormalizedCentralMoment

Retrieves normalized central moment from moment state structure

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

moments
: Pointer to the moment state structure.

j
: x-order of the retrieved moment, j >= 0.

i
: y-order of the retrieved moment, i >= 0 and i + j <= 3.

The function cvGetNormalizedCentralMoment [p 181] retrieves the normalized central moment, which in case of image moments is defined as:

$$\eta_{ij} = \mu_{ij} / M_{00}^{((i+j)/2+1)}$$

GetHuMoments

Calculates seven Hu invariants

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

moments
: Pointer to the moment state structure.

HuMoments
: Pointer to Hu moments structure.

The function cvGetHuMoments [p 182] calculates seven Hu invariants that are defined as:

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

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

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

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

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

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

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

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

Image Processing and Analysis Reference

- Utility Functions [p 202]
  - MatchTemplate [p 202]

Note:
The chapter describes functions for image processing and analysis. Most of the functions work with 2d arrays of pixels. We refer the arrays as "images" however they do not neccesserily have to be IplImage's, they may be CvMat's or CvMatND's as well.

Special Image Transforms

HoughLines

Finds lines in binary image using Hough transform

```
CvSeq* cvHoughLines2( CvArr* image, void* lineStorage, int method,
                      double dRho, double dTheta, int threshold,
                      double param1=0, double param2 );
```

image
: Source 8-bit single-channel (binary) image. It may be modified by the function.

lineStorage
: The storage for the lines detected. It can be a memory storage (in this case a sequence of lines is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of a particular type (see below) where the lines' parameters are written. The matrix header is modified by the function so its `cols`/`rows` contains a number of lines detected (that is a matrix is truncated to fit exactly the detected lines, though no data is deallocated - only the header is modified). In the latter case if the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (the lines are not sorted by length, confidence or whatever criteria).

method
: The Hough transform variant, one of:
  - `CV_HOUGH_STANDARD` - classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ), where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of CV_32FC2 type.
  - `CV_HOUGH_PROBABILISTIC` - probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole lines. Every segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of CV_32SC4 type.
  - `CV_HOUGH_MULTI_SCALE` - multi-scale variant of classical Hough transform. The lines are encoded the same way as in CV_HOUGH_CLASSICAL.

dRho
: Distance resolution in pixel-related units.

dTheta
: Angle resolution measured in radians.

threshold
: Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`.

param1
: The first method-dependent parameter:
  - For classical Hough transform it is not used (0).
  - For probabilistic Hough transform it is the minimum line length.
  - For multi-scale Hough transform it is divisor for distance resolution `dRho`. (The coarse distance resolution will be `dRho` and the accurate resolution will be (`dRho` / `param1`)).

param2
: The second method-dependent parameter:
  - For classical Hough transform it is not used (0).
  - For probabilistic Hough transform it is the maximum gap between line segments lieing on the same line to treat them as the single line segment (i.e. to join them).
  - For multi-scale Hough transform it is divisor for angle resolution `dTheta`. (The coarse angle resolution will be `dTheta` and the accurate resolution will be (`dTheta` / `param2`)).

The function cvHoughLines2 [p ??] implements a few variants of Hough transform for line detection.

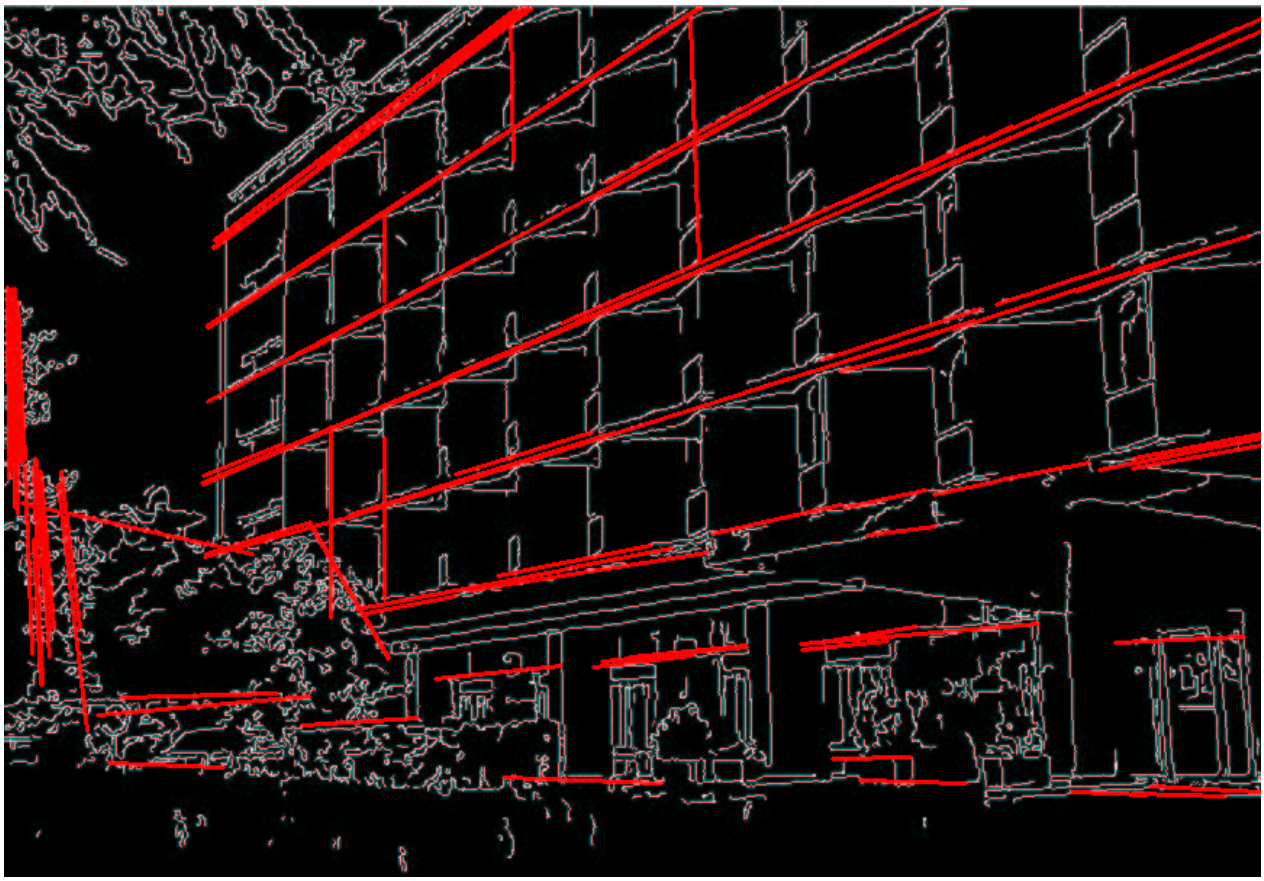
Example. Detecting lines with Hough transform.

```
/* This is a standalone program. Pass an image name as a first parameter of the program.
   Switch between standard and probabilistic Hough transform by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* lines = 0;
        int i;
        cvCanny( src, dst, 50, 200, 3 );
        cvCvtColor( dst, color_dst, CV_GRAY2BGR );
#if 1
        lines = cvHoughLines2( dst, storage, CV_HOUGH_CLASSICAL, 1, CV_PI/180, 150, 0, 0 );

        for( i = 0; i < lines->total; i++ )
        {
            float* line = (float*)cvGetSeqElem(lines,i);
            float rho = line[0];
            float theta = line[1];
            CvPoint pt1, pt2;
            double a = cos(theta), b = sin(theta);
            if( fabs(a) < 0.001 )
            {
                pt1.x = pt2.x = cvRound(rho);
                pt1.y = 0;
                pt2.y = color_dst->height;
            }
            else if( fabs(b) < 0.001 )
            {
                pt1.y = pt2.y = cvRound(rho);
```

```
                pt1.x = 0;
                pt2.x = color_dst->width;
            }
            else
            {
                pt1.x = 0;
                pt1.y = cvRound(rho/b);
                pt2.x = cvRound(rho/a);
                pt2.y = 0;
            }
            cvLine( color_dst, pt1, pt2, CV_RGB(255,0,0), 3, 8 );
        }
#else
        lines = cvHoughLines2( dst, storage, CV_HOUGH_PROBABILISTIC, 1, CV_PI/180, 80, 30, 10 );
        for( i = 0; i < lines->total; i++ )
        {
            CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
            cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3, 8 );
        }
#endif
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "Hough", 1 );
        cvShowImage( "Hough", color_dst );

        cvWaitKey(0);
    }
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in case of probabilistic Hough transform ("#if 0" case):



DistTransform

Calculates distance to closest zero pixel for all non-zero pixels of source image

```
void cvDistTransform( const CvArr* src, CvArr* dst, CvDisType disType=CV_DIST_L2,
                      int maskSize=3, float* mask=0 );
```

src
: Source 8-bit single-channel (binary) image.

dst
: Output image with calculated distances (32-bit floating-point, single-channel).

disType
: Type of distance; can be `CV_DIST_L1, CV_DIST_L2, CV_DIST_C` or `CV_DIST_USER`.

maskSize
: Size of distance transform mask; can be 3 or 5. In case if `CV_DIST_L1` or `CV_DIST_C` the parameter is forced to 3, because 5×5 mask gives the same result as 3×3 in this case yet it is slower.

mask
: User-defined mask in case of user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in case of 3×3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight's move cost) in case of 5×5 mask.

The function cvDistTransform [p 189] calculates the approximated distance from every binary image pixel to the nearest zero pixel. For zero pixels the function sets the zero distance, for others it finds the shortest path consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for 5×5 mask). The overal distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all the horizontal and vertical shifts must have the same cost (that is denoted as `a`), all the diagonal shifts must have the same cost (denoted `b`), and all knight's moves' must have the same cost (denoted `c`). For `CV_DIST_C` and `CV_DIST_L1` types the distance is calculated precisely, whereas for `CV_DIST_L2` (Euclidian distance) the distance can be calculated only with some relative error (5×5 mask gives more accurate results), OpenCV uses the values suggested in [Borgefors86] [p 191] :

```
CV_DIST_C (3×3):
a=1, b=1

CV_DIST_L1 (3×3):
a=1, b=2

CV_DIST_L2 (3×3):
a=0.955, b=1.3693

CV_DIST_L2 (5×5):
a=1, b=1.4, c=2.1969
```

And below are samples of distance field (black (0) pixel is in the middle of white square) in case of user-defined distance:

User-defined 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1	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

User-defined 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	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

Typically, for fast coarse distance estimation CV_DIST_L2, 3×3 mask is used, and for more accurate distance estimation CV_DIST_L2, 5×5 mask is used.

[Borgefors86] Gunilla Borgefors, "Distance Transformations in Digital Images". Computer Vision, Graphics and Image Processing 34, 344-371 (1986).

Histogram Functions

CvHistogram

Muti-dimensional histogram

```
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; /* root of balanced tree storing histogram bins */
    CvSet* set; /* pointer to memory storage (for the balanced tree) */
    int* chdims[CV_HIST_MAX_DIM]; /* cache data for fast calculating */
} CvHistogram;
```

CreateHist

Creates histogram

```
CvHistogram* cvCreateHist( int cDims, int* dims, int type,
                           float** ranges=0, int uniform=1 );
```

cDims
: Number of histogram dimensions.

dims
: Array of histogram dimension sizes.

type
: Histogram representation format: `CV_HIST_ARRAY` means that histogram data is represented as an multi-dimensional dense array CvMatND [p ??] ; `CV_HIST_TREE` means that histogram data is represented as a multi-dimensional sparse array CvSparseMat [p ??] .

ranges
: Array of ranges for histogram bins. Its meaning depends on the `uniform` parameter value. The ranges are used for when histogram is calculated or backprojected to determine, which histogram bin corresponds to which value/tuple of values from the input image[s].

uniform
: Uniformity flag; if not 0, the histogram has evenly spaced bins and for every `0<=i<cDims` `ranges[i]` is array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is split then into `dims[i]` equal parts to determine `i-th` input tuple value ranges for every histogram bin. And if `uniform=0`, then `i-th` element of `ranges` array contains `dims[i]+1` elements: $\texttt{lower}_0, \texttt{upper}_0, \texttt{lower}_1, \texttt{upper}_1 == \texttt{lower}_2, \ldots, \texttt{upper}_{\texttt{dims[i]-1}}$, where $\texttt{lower}_j$ and $\texttt{upper}_j$ are lower and upper boundaries of `i-th` input tuple value for `j-th` bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin, are not counted by cvCalcHist [p 197] and filled with 0 by cvCalcBackProject [p 198] .

The function cvCreateHist [p 192] 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 [p 192] , though cvCalcHist [p 197] and cvCalcBackProject [p 198] may process 8-bit images without setting bin ranges, they assume equally spaced in 0..255 bins.

SetHistBinRanges

Sets bounds of histogram bins

```
void cvSetHistBinRanges( CvHistogram* hist, float** ranges, int uniform=1 );
```

hist
: Histogram.

ranges
: Array of bin ranges arrays, see cvCreateHist [p 192] .

uniform
Uniformity flag, see cvCreateHist [p 192] .

The function cvSetHistBinRanges [p 192] is a stand-alone function for setting bin ranges in the histogram. For more detailed description of the parameters `ranges` and `uniform` see cvCalcHist [p 197] 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.

ReleaseHist

Releases histogram

```
void cvReleaseHist( CvHistogram** hist );
```

hist
Double pointer to the released histogram.

The function cvReleaseHist [p 193] releases the histogram (header and the data). The pointer to histogram is cleared by the function. If `*hist` pointer is already `NULL`, the function does nothing.

ClearHist

Clears histogram

```
void cvClearHist( CvHistogram* hist );
```

hist
Histogram.

The function cvClearHist [p 193] sets all histogram bins to 0 in case of dense histogram and removes all histogram bins in case of sparse array.

MakeHistHeaderForArray

Makes a histogram out of array

```
void cvMakeHistHeaderForArray( int cDims, int* dims, CvHistogram* hist,
                               float* data, float** ranges=0, int uniform=1 );
```

cDims
Number of histogram dimensions.
dims
Array of histogram dimension sizes.
hist
The histogram header initialized by the function.

data
Array that will be used to store histogram bins.
ranges
Histogram bin ranges, see cvCreateHist [p 192] .
uniform
Uniformity flag, see cvCreateHist [p 192] .

The function cvMakeHistHeaderForArray [p 193] initializes the histogram, which header and bins are allocated by user. No cvReleaseHist [p 193] need to be called afterwards. The histogram will be dense, sparse histogram can not be initialized this way.

QueryHistValue_1D

Queries value of histogram bin

```
#define cvQueryHistValue_1D( hist, idx0 ) \
    cvGetReal1D( (hist)->bins, (idx0) )
#define cvQueryHistValue_2D( hist, idx0, idx1 ) \
    cvGetReal2D( (hist)->bins, (idx0), (idx1) )
#define cvQueryHistValue_3D( hist, idx0, idx1, idx2 ) \
    cvGetReal3D( (hist)->bins, (idx0), (idx1), (idx2) )
#define cvQueryHistValue_nD( hist, idx ) \
    cvGetRealND( (hist)->bins, (idx) )
```

hist
Histogram.
idx0, idx1, idx2, idx3
Indices of the bin.
idx
Array of indices

The macros cvQueryHistValue_*D [p ??] return the value of the specified bin of 1D, 2D, 3D or nD histogram. In case of sparse histogram the function returns 0, if the bin is not present in the histogram, and no new bin is created.

GetHistValue_1D

Returns pointer to histogram bin

```
#define cvGetHistValue_1D( hist, idx0 ) \
    ((float*)(cvPtr1D( (hist)->bins, (idx0), 0 ))
#define cvGetHistValue_2D( hist, idx0, idx1 ) \
    ((float*)(cvPtr2D( (hist)->bins, (idx0), (idx1), 0 ))
#define cvGetHistValue_3D( hist, idx0, idx1, idx2 ) \
    ((float*)(cvPtr3D( (hist)->bins, (idx0), (idx1), (idx2), 0 ))
#define cvGetHistValue_nD( hist, idx ) \
    ((float*)(cvPtrND( (hist)->bins, (idx), 0 ))
```

hist
: Histogram.

idx0, idx1, idx2, idx3
: Indices of the bin.

idx
: Array of indices

The macros cvGetHistValue_*D [p ??] return pointer to the specified bin of 1D, 2D, 3D or nD histogram. In case of sparse histogram the function creates a new bins and fills it with 0, if it does not exists.

GetMinMaxHistValue

Finds minimum and maximum histogram bins

```
void cvGetMinMaxHistValue( const CvHistogram* hist,
                           float* minVal, float* maxVal,
                           int* minIdx =0, int* maxIdx =0);
```

hist
: Histogram.

minVal
: Pointer to the minimum value of the histogram; can be NULL.

maxVal
: Pointer to the maximum value of the histogram; can be NULL.

minIdx
: Pointer to the array of coordinates for minimum. If not NULL, must have `hist->c_dims` elements to store the coordinates.

maxIdx
: Pointer to the array of coordinates for maximum. If not NULL, must have `hist->c_dims` elements to store the coordinates.

The function cvGetMinMaxHistValue [p 195] finds the minimum and maximum histogram bins and their positions. In case of several maximums or minimums the earliest in lexicographical order extrema locations are returned.

NormalizeHist

Normalizes histogram

```
void cvNormalizeHist( CvHistogram* hist, double factor );
```

hist
: Pointer to the histogram.

factor
: Normalization factor.

The function cvNormalizeHist [p 195] normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor`.

ThreshHist

Thresholds histogram

```
void cvThreshHist( CvHistogram* hist, double thresh );
```

hist
: Pointer to the histogram.

thresh
: Threshold level.

The function cvThreshHist [p 196] clears histogram bins that are below the specified level.

CompareHist

Compares two dense histograms

```
double cvCompareHist( const CvHistogram* H1, const CvHistogram* H2,
                      CvCompareMethod method );
```

H1
: The first dense histogram.

H2
: The second dense histogram.

method
: Comparison method, one of:
  - CV_COMP_CORREL;
  - CV_COMP_CHISQR;
  - CV_COMP_INTERSECT.

The function cvCompareHist [p 196] compares two histograms using specified method and returns the comparison result. It processes as following:

```
Correlation (method=CV_COMP_CORREL):
d(H1,H2)=sumI(H'1(I)•H'2(I))/sqrt(sumI[H'1(I)2]•sumI[H'2(I)2])
where
H'k(I)=Hk(I)-1/N•sumJHk(J) (N=number of histogram bins)

Chi-Square (method=CV_COMP_CHISQR):
d(H1,H2)=sumI[(H1(I)-H2(I))/(H1(I)+H2(I))]

Intersection (method=CV_COMP_INTERSECT):
d(H1,H2)=sumImax(H1(I),H2(I))
```

Note, that the function can operate on dense histogram only. To compare sparse histogram or more general sparse configurations of weighted points, consider cvCalcEMD [p ??] function.

CopyHist

Copies histogram

```
void cvCopyHist( CvHistogram* src, CvHistogram** dst );
```

src
: Source histogram.

dst
: Pointer to destination histogram.

The function cvCopyHist [p 197] makes a copy of the histogram. If the second histogram pointer `*dst` is NULL, a new histogram of the same size as `src` is created. Otherwise, both histograms must have equal types and sizes. Then the function copies the source histogram bins values to destination histogram and sets the same as `src`'s value ranges.

CalcHist

Calculates histogram of image(s)

```
void cvCalcHist( IplImage** img, CvHistogram* hist,
                 int doNotClear=0, const CvArr* mask=0 );
```

img
: Source images (though, you may pass CvMat** as well).

hist
: Pointer to the histogram.

doNotClear
: Clear flag, if it is non-zero, the histogram is not cleared before calculation. It may be useful for iterative histogram update.

mask
: The operation mask, determines what pixels of the source images are counted.

The function cvCalcHist [p 197] calculates the histogram of one or more single-channel images. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input images.

Sample. Calculating and displaying 2D Hue-Saturation histogram of a color image

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv )
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 1))!= 0)
    {
```

```
        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* v_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* planes[] = { h_plane, s_plane };
        IplImage* hsv = cvCreateImage( cvGetSize(src), 8, 3 );
        int h_bins = 30, s_bins = 32;
        int hist_size[] = {h_bins, s_bins};
        float h_ranges[] = { 0, 180 }; /* hue varies from 0 (~0°red) to 180 (~360°red again) */
        float s_ranges[] = { 0, 255 }; /* saturation varies from 0 (black-gray-white) to 255 (pure spectrum color) */
        float* ranges[] = { h_ranges, s_ranges };
        int scale = 10;
        IplImage* hist_img = cvCreateImage( cvSize(h_bins*scale,s_bins*scale), 8, 3 );
        CvHistogram* hist;
        float max_value = 0;
        int h, s;

        cvCvtColor( src, hsv, CV_BGR2HSV );
        cvCvtPixToPlane( hsv, h_plane, s_plane, v_plane, 0 );
        hist = cvCreateHist( 2, hist_size, CV_HIST_ARRAY, ranges, 1 );
        cvCalcHist( planes, hist, 0, 0 );
        cvGetMinMaxHistValue( hist, 0, &max_value, 0, 0 );
        cvZero( hist_img );

        for( h = 0; h < h_bins; h++ )
        {
            for( s = 0; s < s_bins; s++ )
            {
                float bin_val = cvQueryHistValue_2D( hist, h, s );
                int intensity = cvRound(bin_val*255/max_value);
                cvRectangle( hist_img, cvPoint( h*scale, s*scale ),
                             cvPoint( (h+1)*scale - 1, (s+1)*scale - 1),
                             CV_RGB(intensity,intensity,intensity), /* graw a grayscale histogram.
                                                                       if you have idea how to do it
                                                                       nicer let us know */
                             CV_FILLED );
            }
        }

        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "H-S Histogram", 1 );
        cvShowImage( "H-S Histogram", hist_img );

        cvWaitKey(0);
    }
}
```

CalcBackProject

Calculates back projection

```
void cvCalcBackProject( IplImage** img, CvArr* backProject, const CvHistogram* hist );
```

img
: Source images (though you may pass CvMat** as well).

backProject
: Destination back projection image of the same type as the source images.

hist
: Histogram.

The function cvCalcBackProject [p 198] calculates the back project of the histogram. For each tuple of pixels at the same position of all input single-channel images the function puts the value of the histogram bin, corresponding to the tuple, to the destination image. In terms of statistics, the value of each output image pixel is probability of the observed tuple given the distribution (histogram). For example, to find a

red object in the picture, one may do the following:

1. Calculate a hue histogram for the red object assuming the image contains only this object. The histogram is likely to have a strong maximum, corresponding to red color.
2. Calculate back projection of a hue plane of input image where the object is searched, using the histogram. Threshold the image.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

That is the approximate algorithm of Camshift color object tracker, except for the last step, where CAMSHIFT algorithm is used to locate the object on the back projection given the previous object position.

CalcBackProjectPatch

Locates a template within image by histogram comparison

```
void cvCalcBackProjectPatch( IplImage** img, CvArr* dst,
                             CvSize patchSize, CvHistogram* hist,
                             int method, float normFactor );
```

img
: Source images (though, you may pass CvMat** as well)

dst
: Destination image.

patchSize
: Size of patch slid though the source image.

hist
: Histogram

method
: Compasion method, passed to cvCompareHist [p 196] (see description of that function).

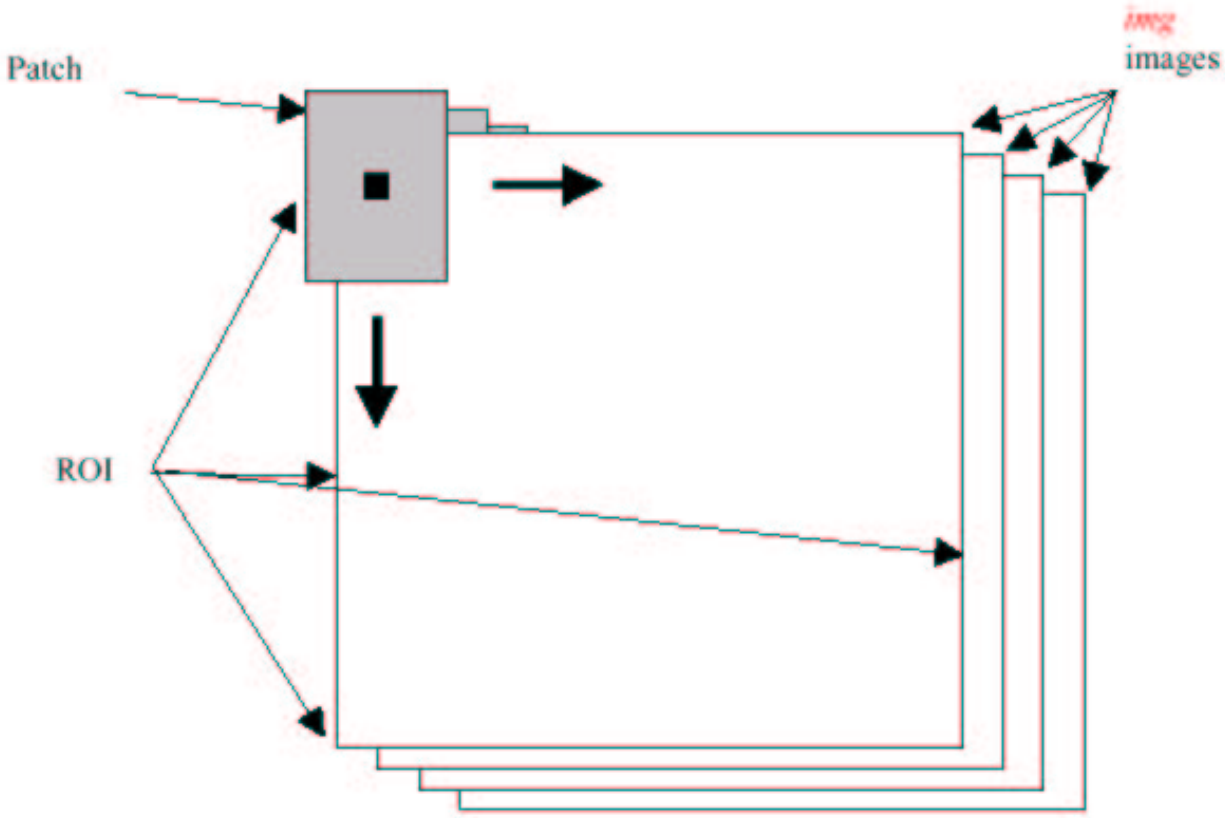
normFactor
: Normalization factor for histograms, will affect normalization scale of destination image, pass 1. if unsure.

The function cvCalcBackProjectPatch [p 199] 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 the number of 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 the picture below. The histogram is normalized using the parameter `norm_factor` so that it may be compared with `hist`. The calculated histogram is compared to the model histogram; `hist` uses the

function cvCompareHist [p 196] with the comparison method=`method`). The resulting output is placed at the location corresponding to the patch anchor in the probability image `dst`. This process is repeated as the patch is slid over the ROI. Iterative histogram update by subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can save a lot of operations, though it is not implemented yet.

Back Project Calculation by Patches



CalcProbDensity

Divides one histogram by another

```
void  cvCalcProbDensity( const CvHistogram* hist1, const CvHistogram* hist2,
                         CvHistogram* histDens, double scale=255 );
```

hist1
: first histogram (divisor).

hist2
: second histogram.

histDens
: destination histogram.

The function cvCalcProbDensity [p 200] calculates the object probability density from the two histograms as:

```
histDens(I)=0  if hist1(I)==0
            scale  if hist1(I)!=0 && hist2(I)>hist1(I)
            hist2(I)*scale/hist1(I) if hist1(I)!=0 && hist2(I)<=hist1(I)
```

So the destination histogram bins are within [0,scale).

CalcEMD2

Computes "minimal work" distance between two weighted point configurations

```
float cvCalcEMD2( const CvArr* signature1, const CvArr* signature2, CvDisType distType,
                  float (*distFunc)(const float* f1, const float* f2, void* userParam ),
                  const CvArr* costMatrix, CvArr* flow,
                  float* lowerBound, void* userParam );
```

signature1
: First signature, `size1×dims+1` floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used.

signature2
: Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different, in this case an extra "dummy" point is added to either `signature1` or `signature2`.

distType
: 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 `distFunc` or pre-calculated `costMatrix` is used.

distFunc
: The user-defined distance function. It takes coordinates of two points and returns the distance between the points.

costMatrix
: The user-defined `size1×size2` cost matrix. At least one of `costMatrix` and `distFunc` must be NULL. Also, if a cost matrix is used, lower boundary (see below) can not be calculated, because it needs a metric function.

flow
: The resultant `size1×size2` flow matrix: $\texttt{flow}_{ij}$ is a flow from i-th point of `signature1` to j-th point of `signature2`

lowerBound
: Optional output parameter: lower boundary of distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or there is the signatures consist of weights only (i.e. the matrices have a single column).

userParam
Pointer to optional data that is passed into the user-defined distance function.

The function cvCalcEMD2 [p 201] computes earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the application desctibed in [RubnerSept98] [p 202] is multi-dimensional histogram comparison for image retrieval. EMD is a transportation problem that is solved using some modification of simplex algorithm, thus the complexity is exponential in the worst case, though, it is much faster in average. In case of real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

[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.

Utility Functions

MatchTemplate

Compares template against overlapped image regions

```
void cvMatchTemplate( const CvArr* I, const CvArr* T,
                      CvArr* result, int method );
```

I
Image where the search is running. It should be single-chanel, 8-bit or 32-bit floating-point.
T
Searched template; must be not greater than the source image and the same data type as the image.
R
Image of comparison results; single-channel 32-bit floating-point. If `I` is `W×H` and `T` is `w×h` then `R` must be `W-w+1×H-h+1`.
method
Specifies the way the template must be compared with image regions (see below).

The function cvMatchTemplate [p 202] is similiar to cvCalcBackProjectPatch [p 199] . It slids through `I`, compares `w×h` patches against `T` using the specified method and stores the comparison results to `result`. Here are the formular for the different comparison methods one may use (the summation is done over template and/or the image patch: `x'=0..w-1, y'=0..h-1`):

```
method=CV_TM_SQDIFF:
R(x,y)=sum x',y' [T(x',y')-I(x+x',y+y')] 2

method=CV_TM_SQDIFF_NORMED:
R(x,y)=sum x',y' [T(x',y')-I(x+x',y+y')] 2 /sqrt[sum x',y' T(x',y') 2 •sum x',y' I(x+x',y+y') 2 ]

method=CV_TM_CCORR:
R(x,y)=sum x',y' [T(x',y')•I(x+x',y+y')]
```

method=CV_TM_CCORR_NORMED:
$R(x,y)=\sum_{x',y'}[T(x',y') \cdot I(x+x',y+y')]/\sqrt{[\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2]}$

method=CV_TM_CCOEFF:
$R(x,y)=\sum_{x',y'}[T'(x',y') \cdot I'(x+x',y+y')]$,

where $T'(x',y')=T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$ (mean template brightness=>0)
$I'(x+x',y+y')=I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$ (mean patch brightness=>0)

method=CV_TM_CCOEFF_NORMED:
$R(x,y)=\sum_{x',y'}[T'(x',y') \cdot I'(x+x',y+y')]/\sqrt{[\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2]}$

After the function finishes comparison, the best matches can be found as global minimums (CV_TM_SQDIFF*) or maximums (CV_TM_CCORR* and CV_TM_CCOEFF*) using cvMinMaxLoc [p 79] function.

Structural Analysis Reference

- ClearSubdivVoronoi2D [p 226]

Contour Processing Functions

ApproxChains

Approximates Freeman chain(s) with polygonal curve

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

srcSeq
: Pointer to the chain that can refer to other chains.

storage
: Storage location for the resulting polylines.

method
: Approximation method (see the description of the function cvFindContours [p 176]).

parameter
: Method parameter (not used now).

minimalPerimeter
: Approximates only those contours whose perimeters are not less than *minimalPerimeter*. Other chains are removed from the resulting structure.

recursive
: If not 0, the function approximates all chains that access can be obtained to from *srcSeq* by *h_next* or *v_next links*. If 0, the single chain is approximated.

This is a stand-alone approximation routine. The function cvApproxChains [p 205] works exactly in the same way as cvFindContours [p 176] with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via *v_next* or *h_next* fields of the returned structure.

StartReadChainPoints

Initializes chain reader

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

chain Pointer to chain. reader Chain reader state.

The function cvStartReadChainPoints [p 205] initializes a special reader (see Dynamic Data Structures [p 99] for more information on sets and sequences).

ReadChainPoint

Gets next chain point

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

reader
: Chain reader state.

The function cvReadChainPoint [p 206] returns the current chain point and updates the reader position.

ApproxPoly

Approximates polygonal curve(s) with desired precision

```
CvSeq* cvApproxPoly( const void* srcSeq, int headerSize, CvMemStorage* storage,
                     int method, double parameter,
                     int parameter2=0 );
```

srcSeq
: Sequence of array of points.

headerSize
: Header size of approximated curve[s].

storage
: Container for approximated contours. If it is NULL, the input sequences' storage is used.

method
: Approximation method; only *CV_POLY_APPROX_DP* is supported, that corresponds to Douglas-Peucker algorithm.

parameter
: Method-specific parameter; in case of *CV_POLY_APPROX_DP* it is a desired approximation accuracy.

parameter2
: If case if *srcSeq* is sequence it means whether the single sequence should be approximated or all sequences on the same level or below *srcSeq* (see cvFindContours [p 176] for description of hierarchical contour structures). And if *srcSeq* is array (CvMat [p ??] *) of points, the parameter specifies whether the curve is closed (*parameter2*!=0) or not (*parameter2*=0).

The function cvApproxPoly [p 206] approximates one or more curves and returns the approximation result[s]. In case of multiple curves approximation the resultant tree will have the same structure as the input one (1:1 correspondence).

BoundingRect

Calculates up-right bounding rectangle of point set

```
CvRect cvBoundingRect( CvArr* contour, int update );
```

contour
: Sequence or array of points.

update
: The update flag. Here is list of possible combination of the flag values and type of *contour*:
  - update=0, contour ~ CvContour*: the bounding rectangle is not calculated, but it is taken from *rect* field of the contour header.
  - update=1, contour ~ CvContour*: the bounding rectangle is calculated and written to *rect* field of the contour header.
  - update=0, contour ~ CvSeq* or CvMat*: the bounding rectangle is calculated and returned.
  - update=1, contour ~ CvSeq* or CvMat*: runtime error is raised.

The function cvBoundingRect [p 206] returns the up-right bounding rectangle for 2d point set.

ContourArea

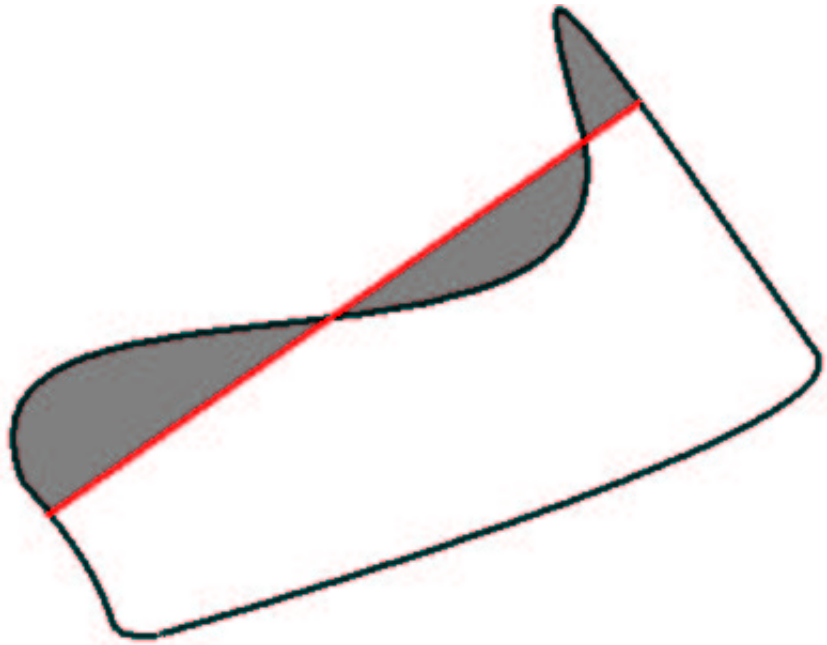
Calculates area of the whole contour or contour section

```
double cvContourArea( const CvArr* contour, CvSlice slice=CV_WHOLE_SEQ );
```

contour
: Contour (sequence or array of vertices).

slice
: Starting and ending points of the contour section of interest, by default area of the whole contour is calculated.

The function cvContourArea [p 207] calculates area of the whole contour or contour section. In the latter case the total area bounded by the contour arc and the chord connecting the 2 selected points is calculated as shown on the picture below:



NOTE: Orientation of the contour affects the area sign, thus the function may return *negative* result. Use *fabs()* function from C runtime to get the absolute value of area.

ArcLength

Calculates contour perimeter or curve length

```
double cvArcLength( const void* curve, CvSlice slice=CV_WHOLE_SEQ, int isClosed=-1 );
```

curve
: Sequence or array of the curve points.

slice
: Starting and ending points of the curve, by default the whole curve length is calculated.

isClosed
: Indicates whether the curve is closed or not. There are 3 cases:
  - isClosed=0 - the curve is assumed to be unclosed.
  - isClosed>0 - the curve is assumed to be closed.
  - isClosed<0 - if curve is sequence, the flag CV_SEQ_FLAG_CLOSED of ((CvSeq*)curve)->flags is checked to determine if the curve is closed or not, otherwise (curve is represented by array (CvMat*) of points) it is assumed to be unclosed.

The function cvArcLength [p 208] calculates length or curve as sum of lengths of segments between subsequent points

---

MatchShapes

Compares two shapes

```
double cvMatchShapes( const void* A, const void* B,
                      int method, double parameter=0 );
```

A
: First contour or grayscale image

B
: Second contour or grayscale image

method
: Comparison method, one of CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3.

parameter
: Method-specific parameter (is not used now).

The function cvMatchShapes [p 209] compares two shapes. The 3 implemented methods all use Hu moments (see cvGetHuMoments [p 182]):

```
method=CV_CONTOUR_MATCH_I1:
I1(A,B)=sum i=1..7 abs(1/mA i - 1/mB i)

method=CV_CONTOUR_MATCH_I2:
I2(A,B)=sum i=1..7 abs(mA i - mB i)

method=CV_CONTOUR_MATCH_I3:
I3(A,B)=sum i=1..7 abs(mA i - mB i)/abs(mA i)

where
mA i=sign(hA i)•log(hA i),
mB i=sign(hB i)•log(hB i),
hA i, hB i - Hu moments of A and B, respectively.
```

CreateContourTree

Creates hierarchical representation of contour

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

contour
: Input contour.

storage
: Container for output tree.

threshold
: Approximation accuracy.

The function cvCreateContourTree [p 209] creates binary tree representation for the input *contour* and returns the pointer to its root. If the parameter *threshold* is less than or equal to 0, the function creates full binary tree representation. If the threshold is greater 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 the created tree.

ContourFromContourTree

Restores contour from tree

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

tree
: Contour tree.

storage
: Container for the reconstructed contour.

criteria
: Criteria, where to stop reconstruction.

The function cvContourFromContourTree [p 210] restores the contour from its binary tree representation. The parameter *criteria* determines the accuracy and/or the number of tree levels used for reconstruction, so it is possible to build approximated contour. The function returns reconstructed contour.

MatchContourTrees

Compares two contours using their tree representations

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

tree1
: First contour tree.

tree2
: Second contour tree.

method
: Similarity measure, only *CV_CONTOUR_TREES_MATCH_I1* is supported.

threshold
: Similarity threshold.

The function cvMatchContourTrees [p 210] calculates the value of the matching measure for two contour trees. The similarity measure is calculated level by level from the binary tree roots. If at the certain level difference between contours becomes less than *threshold*, the reconstruction process is interrupted and the current difference is returned.

Geometry Functions

MaxRect

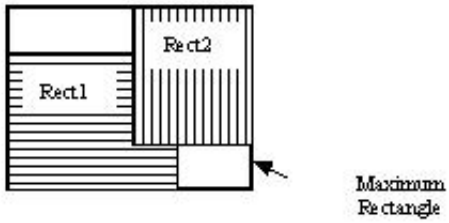
Finds bounding rectangle for two given rectangles

```
CvRect cvMaxRect( const CvRect* rect1, const CvRect* rect2 );
```

rect1
: First rectangle

rect2
: Second rectangle

The function cvMaxRect [p 211] finds minimum area rectangle that contains both input rectangles inside:



CvBox2D

Rotated 2D box

```
typedef struct CvBox2D
{
    CvPoint2D32f center;  /* center of the box */
    CvSize2D32f  size;    /* box width and length */
    float angle;          /* angle between the horizontal axis
                             and the first side (i.e. length) in radians */
}
CvBox2D;
```

BoxPoints

Finds box vertices

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] );
```

box
: Box

pt
: Array of vertices

The function cvBoxPoints [p 211] calculates vertices of the input 2d box. Here is the function code:

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    float a = (float)cos(box.angle)*0.5f;
    float b = (float)sin(box.angle)*0.5f;

    pt[0].x = box.center.x - a*box.size.height - b*box.size.width;
    pt[0].y = box.center.y + b*box.size.height - a*box.size.width;
    pt[1].x = box.center.x + a*box.size.height - b*box.size.width;
    pt[1].y = box.center.y - b*box.size.height - a*box.size.width;
    pt[2].x = 2*box.center.x - pt[0].x;
    pt[2].y = 2*box.center.y - pt[0].y;
    pt[3].x = 2*box.center.x - pt[1].x;
    pt[3].y = 2*box.center.y - pt[1].y;
}
```

FitEllipse

Fits ellipse to set of 2D points

```
CvBox2D cvFitEllipse2( const CvArr* points );
```

points
: Sequence or array of points.

The function cvFitEllipse [p 212] calculates ellipse that fits best (in least-squares sense) to a set of 2D points. The meaning of the returned structure fields is similar to those in cvEllipse [p 147] except that *size* stores the full lengths of the ellipse axises, not half-lengths

---

FitLine

Fits line to 2D or 3D point set

```
void  cvFitLine( const CvArr* points, CvDisType disType, double C,
                 double reps, double aeps, float* line );
```

points
: Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates.

disType
: The distance used for fitting (see the discussion).

C
: Numerical parameter for some types of distances, if 0 then some optimal value is chosen.

reps, aeps
: Sufficient accuracy for radius (distance between the coordinate origin and the line) and angle, respectively, 0.01 would be a good defaults for both. is used.

line
: The output line parameters. In case of 2d fitting it is array of 4 floats *(vx, vy, x0, y0)* where *(vx, vy)* is a normalized vector collinear to the line and *(x0, y0)* is some point on the line. In case of 3D fitting it is array of 6 floats *(vx, vy, vz, x0, y0, z0)* where *(vx, vy, vz)* is a normalized vector collinear to the line and *(x0, y0, z0)* is some point on the line.

The function cvFitLine [p ??] fits line to 2D or 3D point set by minimizing $\text{sum}_i \rho(r_i)$, where r_i is distance between i-th point and the line and $\rho(r)$ is a distance function, one of:

```
disType=CV_DIST_L2 (L2):
ρ(r)=r²/2 (the simplest and the fastest least-squares method)

disType=CV_DIST_L1 (L1):
ρ(r)=r

disType=CV_DIST_L12 (L1-L2):
ρ(r)=2•[sqrt(1+r²/2) - 1]

disType=CV_DIST_FAIR (Fair):
ρ(r)=C²•[r/C - log(1 + r/C)],  C=1.3998

disType=CV_DIST_WELSCH (Welsch):
ρ(r)=C²/2•[1 - exp(-(r/C)²)],  C=2.9846

disType=CV_DIST_HUBER (Huber):
ρ(r)= r²/2,   if r < C
      C•(r-C/2),   otherwise;   C=1.345
```

ConvexHull2

Finds convex hull of points set

```
CvSeq* cvConvexHull2( const void* points, void* hullStorage=0,
                      int orientation=CV_CLOCKWISE, int returnPoints=0 );
```

points
: Sequence or array of 2D points with 32-bit integer or floating-point coordinates.

hullStorage
: The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified so to truncate the array downto the hull size.

orientation
: Desired orientation of convex hull: *CV_CLOCKWISE* or *CV_COUNTER_CLOCKWISE*.

returnPoints
: If non-zero, the points themselves will be stored in the hull instead of indices if *hullStorage* is array, or pointers if *hullStorage* is memory storage.

The function cvConvexHull2 [p 213] finds convex hull of 2D point set using Sklansky's algorithm. If *hullStorage* is memory storage, the function creates a sequence containing the hull points or pointers to them, depending on *returnPoints* value and returns the sequence on output.

Example. Building convex hull for a sequence or array of points

```
#include "cv.h"
#include "highgui.h"
#include <stdlib.h>

#define ARRAY  0 /* switch between array/sequence method by replacing 0<=>1 */

void main( int argc, char** argv )
{
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    cvNamedWindow( "hull", 1 );

#if !ARRAY
        CvMemStorage* storage = cvCreateMemStorage();
#endif

    for(;;)
    {
        int i, count = rand()%100 + 1, hullcount;
        CvPoint pt0;
#if !ARRAY
        CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2, sizeof(CvContour),
                                    sizeof(CvPoint), storage );
        CvSeq* hull;

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            cvSeqPush( ptseq, &pt0 );
        }
        hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
        hullcount = hull->total;
#else
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
        int* hull = (int*)malloc( count * sizeof(hull[0]));
        CvMat pointMat = cvMat( 1, count, CV_32SC2, points );
        CvMat hullMat = cvMat( 1, count, CV_32SC1, hull );

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            points[i] = pt0;
        }
        cvConvexHull2( &pointMat, &hullMat, CV_CLOCKWISE, 0 );
```

```
        hullcount = hullMat.cols;
#endif
        cvZero( img );
        for( i = 0; i < count; i++ )
        {
#if !ARRAY
            pt0 = *CV_GET_SEQ_ELEM( CvPoint, ptseq, i );
#else
            pt0 = points[i];
#endif
            cvCircle( img, pt0, 2, CV_RGB( 255, 0, 0 ), CV_FILLED );
        }

#if !ARRAY
        pt0 = **CV_GET_SEQ_ELEM( CvPoint*, hull, hullcount - 1 );
#else
        pt0 = points[hull[hullcount-1]];
#endif

        for( i = 0; i < hullcount; i++ )
        {
#if !ARRAY
            CvPoint pt = **CV_GET_SEQ_ELEM( CvPoint*, hull, i );
#else
            CvPoint pt = points[hull[i]];
#endif
            cvLine( img, pt0, pt, CV_RGB( 0, 255, 0 ));
            pt0 = pt;
        }

        cvShowImage( "hull", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;

#if !ARRAY
        cvClearMemStorage( storage );
#else
        free( points );
        free( hull );
#endif
    }
}
```

CheckContourConvexity

Tests contour convex

```
int cvCheckContourConvexity( const void* contour );
```

contour
: Tested contour (sequence or array of points).

The function cvCheckContourConvexity [p 215] tests whether the input contour is convex or not. The contour must be simple, i.e. without self-intersections.

CvConvexityDefect

Structure describing a single contour convexity detect

```
typedef struct CvConvexityDefect
{
    CvPoint* start; /* point of the contour where the defect begins */
    CvPoint* end; /* point of the contour where the defect ends */
    CvPoint* depth_point; /* the farthest from the convex hull point within the defect */
    float depth; /* distance between the farthest point and the convex hull */
} CvConvexityDefect;
```

Picture. Convexity defects for hand contour.

ConvexityDefects

Finds convexity defects of contour

```
CvSeq* cvConvexityDefects( const void* contour, const void* convexhull,
                           CvMemStorage* storage=0 );
```

contour
: Input contour.

convexhull
: Convex hull obtained using cvConvexHull2 [p 213] that should contain pointers or indices to the contour points, not the hull points themselves, i.e. *returnPoints* parameter in cvConvexHull2 [p 213] should be 0.

storage
: Container for output sequence of convexity defects. If it is NULL, contour or hull (in that order) storage is used.

The function cvConvexityDefects [p 216] finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect [p 216] structures.

MinAreaRect2

Finds circumscribed rectangle of minimal area for given 2D point set

```
CvBox2D  cvMinAreaRect2( const void* points, CvMemStorage* storage=0 );
```

points
: Sequence or array of points.

storage
: Optional temporary memory storage.

The function cvMinAreaRect2 [p 217] finds a circumscribed rectangle of the minimal area for 2D point set by building convex hull for the set and applying rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour

MinEnclosingCircle

Finds circumscribed circle of minimal area for given 2D point set

```
void cvMinEnclosingCircle( const void* points, CvPoint2D32f* center, float* radius );
```

points
Sequence or array of 2D points.
center
Output parameter. The center of the enclosing circle.
radius
Output parameter. The radius of the enclosing circle.

The function cvMinEnclosingCircle [p 217] finds the minimal circumscribed circle for 2D point set using iterative algorithm.

CalcPGH

Calculates pair-wise geometrical histogram for contour

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

contour
Input contour. Currently, only integer point coordinates are allowed.
hist
Calculated histogram; must be two-dimensional.

The function cvCalcPGH [p 218] calculates 2D pair-wise geometrical histogram (PGH), described in [Iivarinen97] [p ??] , for the 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 (that is, the histogram is transposed relatively to [Iivarninen97] definition). The histogram can be used for contour matching.

[Iivarinen97] Jukka Iivarinen, Markus Peura, Jaakko Srel, and Ari Visa. Comparison of Combined Shape Descriptors for Irregular Objects, 8th British Machine Vision Conference, BMVC'97. You may find online version at http://www.cis.hut.fi/research/IA/paper/publications/bmvc97/bmvc97.html

---

KMeans

Splits set of vectors by given number of clusters

```
void cvKMeans2( const CvArr* samples, int numClusters,
                CvArr* clusterIdx, CvTermCriteria termcrit );
```

samples
Floating-point matrix of input samples, one row per sample.

numClusters
: Number of clusters to split the set by.

clusterIdx
: Output integer vector storing cluster indices for every sample.

termcrit
: Specifies maximum number of iterations and/or accuracy (distance the centers move by between the subsequent iterations).

The function cvKMeans2 [p ??] implements k-means algorithm that finds centers of *numClusters* clusters and groups the input samples around the clusters. On output *clusterIdx(i)* contains a cluster index for sample stored in i-th rows of *samples*.

Example. Clustering random samples of multi-gaussian distribution with k-means

```
#include "cv.h"
#include "highgui.h"

void main( int argc, char** argv )
{
    #define MAX_CLUSTERS 5
    static const int color_tab[MAX_CLUSTERS] =
    {
        CV_RGB(255,0,0), CV_RGB(0,255,0), CV_RGB(100,100,255),
        CV_RGB(255,0,255), CV_RGB(255,255,0)
    };
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRandState rng;
    cvRandInit( &rng, 0, 1, -1, CV_RAND_NORMAL );

    cvNamedWindow( "clusters", 1 );

    for(;;)
    {
        int k, cluster_count = cvRandNext(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandNext(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

        /* generate random sample from multigaussian distribution */
        for( k = 0; k < cluster_count; k++ )
        {
            CvPoint center;
            CvMat point_chunk;
            center.x = cvRandNext(&rng)%img->width;
            center.y = cvRandNext(&rng)%img->height;
            cvRandSetRange( &rng, center.x, img->width/6, 0 );
            cvRandSetRange( &rng, center.y, img->height/6, 1 );
            cvGetRows( points, &point_chunk, k*sample_count/cluster_count,
                       k == cluster_count - 1 ? sample_count : (k+1)*sample_count/cluster_count );

            cvRand( &rng, &point_chunk );
        }

        /* shuffle samples */
        for( i = 0; i < sample_count/2; i++ )
        {
            CvPoint2D32f* pt1 = (CvPoint2D32f*)points->data.fl + cvRandNext(&rng)%sample_count;
            CvPoint2D32f* pt2 = (CvPoint2D32f*)points->data.fl + cvRandNext(&rng)%sample_count;
            CvPoint2D32f temp;
            CV_SWAP( *pt1, *pt2, temp );
```

```
        }

        cvKMeans2( points, cluster_count, clusters,
                   cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

        cvZero( img );

        for( i = 0; i < sample_count; i++ )
        {
            CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
            int cluster_idx = clusters->data.i[i];
            cvCircle( img, cvPointFrom32f(pt), 2, color_tab[cluster_idx], CV_FILLED );
        }

        cvReleaseMat( &points );
        cvReleaseMat( &clusters );

        cvShowImage( "clusters", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;
    }
}
```

MinEnclosingCircle

Finds circumscribed circle of minimal area for given 2D point set

```
void cvMinEnclosingCircle( const void* points, CvPoint2D32f* center, float* radius );
```

points
: Sequence or array of 2D points.

center
: Output parameter. The center of the enclosing circle.

radius
: Output parameter. The radius of the enclosing circle.

The function cvMinEnclosingCircle [p 220] finds the minimal circumscribed circle for 2D point set using iterative algorithm.

Planar Subdivisions

CvSubdiv2D

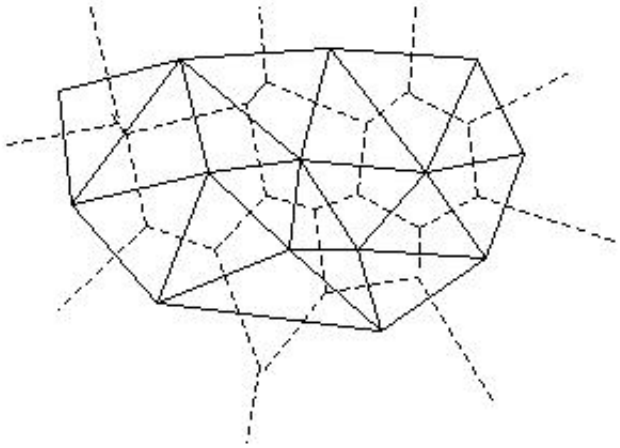
Planar subdivision

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
```

```
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
CvSubdiv2D;
```

Planar subdivision is a subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on 2d point set, where the points are linked together and form a planar graph, which, together with a few edges connecting exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists dual subdivision there facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called virtual point below) of dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dot lines



OpenCV subdivides plane into triangles using Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case the dual subdivision is Voronoi diagram of input 2d point set. The subdivisions can be used for 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) etc.

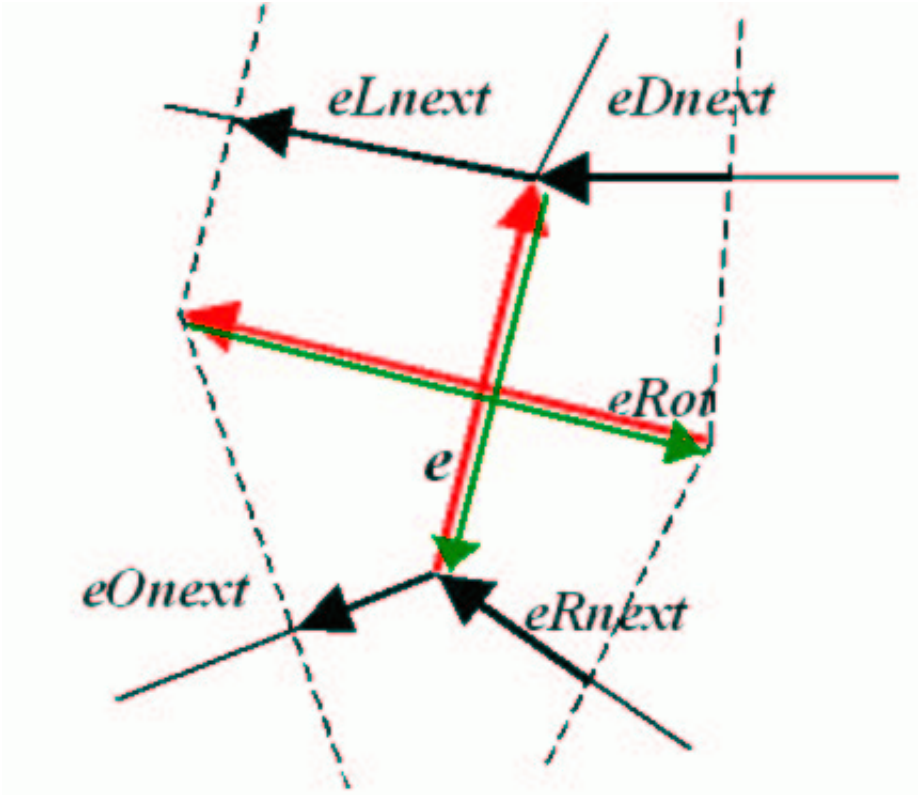
CvQuadEdge2D

Quad-edge of planar subdivision

```
/* one of edges within quad-edge, lower 2 bits is index (0..3)
   and upper bits are quad-edge pointer */
typedef long CvSubdiv2DEdge;

/* quad-edge structure fields */
```

```
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```

Quad-edge is a basic element of subdivision, it contains four edges (e, eRot and reversed e & eRot):



CvSubdiv2DPoint

Point of original or dual subdivision

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int            flags;      \
    CvSubdiv2DEdge first;      \
    CvPoint2D32f   pt;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
```

```
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

Subdiv2DGetEdge

Returns one of edges related to given

```
CvSubdiv2DEdge  cvSubdiv2DGetEdge( CvSubdiv2DEdge edge, CvNextEdgeType type );
#define cvSubdiv2DNextEdge( edge ) cvSubdiv2DGetEdge( edge, CV_NEXT_AROUND_ORG )
```

edge
: Subdivision edge (not a quad-edge)

type
: Specifies, which of related edges to return, one of:
  - CV_NEXT_AROUND_ORG - next around the edge origin (*eOnext* on the picture above if *e* is the input edge)
  - CV_NEXT_AROUND_DST - next around the edge vertex (*eDnext*)
  - CV_PREV_AROUND_ORG - previous around the edge origin (reversed *eRnext*)
  - CV_PREV_AROUND_DST - previous around the edge destination (reversed *eLnext*)
  - CV_NEXT_AROUND_LEFT - next around the left facet (*eLnext*)
  - CV_NEXT_AROUND_RIGHT - next around the right facet (*eRnext*)
  - CV_PREV_AROUND_LEFT - previous around the left facet (reversed *eOnext*)
  - CV_PREV_AROUND_RIGHT - previous around the right facet (reversed *eDnext*)

The function cvSubdiv2DGetEdge [p 223] returns one the edges related to the input edge.

Subdiv2DRotateEdge

Returns another edge of the same quad-edge

```
CvSubdiv2DEdge  cvSubdiv2DRotateEdge( CvSubdiv2DEdge edge, int rotate );
```

edge
: Subdivision edge (not a quad-edge)

type
: Specifies, which of edges of the same quad-edge as the input one to return, one of:
  - 0 - the input edge (*e* on the picture above if *e* is the input edge)
  - 1 - the rotated edge (*eRot*)
  - 2 - the reversed edge (reversed *e* (in green))
  - 3 - the reversed rotated edge (reversed *eRot* (in green))

The function cvSubdiv2DRotateEdge [p 223] returns one the edges of the same quad-edge as the input edge.

Subdiv2DEdgeOrg

Returns edge origin

```
CvSubdiv2DPoint* cvSubdiv2DEdgeOrg( CvSubdiv2DEdge edge );
```

edge
: Subdivision edge (not a quad-edge)

The function cvSubdiv2DEdgeOrg [p 224] returns the edge origin. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using function cvCalcSubdivVoronoi2D [p 226] .

Subdiv2DEdgeDst

Returns edge destination

```
CvSubdiv2DPoint* cvSubdiv2DEdgeDst( CvSubdiv2DEdge edge );
```

edge
: Subdivision edge (not a quad-edge)

The function cvSubdiv2DEdgeDst [p 224] returns the edge destination. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using function cvCalcSubdivVoronoi2D [p 226] .

CreateSubdivDelaunay2D

Creates empty Delaunay triangulation

```
CvSubdiv2D* cvCreateSubdivDelaunay2D( CvRect rect, CvMemStorage* storage );
```

rect
: Rectangle that includes all the 2d points that are to be added to subdivision.

storage
: Container for subdivision.

The function cvCreateSubdivDelaunay2D [p 224] creates an empty Delaunay subdivision, where 2d points can be added further using function cvSubdivDelaunay2DInsert [p 225] . All the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

SubdivDelaunay2DInsert

Inserts a single point to Delaunay triangulation

```
CvSubdiv2DPoint*  cvSubdivDelaunay2DInsert( CvSubdiv2D* subdiv, CvPoint2D32f pt);
```

subdiv
: Delaunay subdivision created by function cvCreateSubdivDelaunay2D [p 224] .

pt
: Inserted point.

The function cvSubdivDelaunay2DInsert [p 225] inserts a single point to subdivision and modifies the subdivision topology appropriately. If a points with same coordinates exists already, no new points is added. The function returns pointer to the allocated point. No virtual points coordinates is calculated at this stage.

Subdiv2DLocate

Inserts a single point to Delaunay triangulation

```
CvSubdiv2DPointLocation  cvSubdiv2DLocate( CvSubdiv2D* subdiv, CvPoint2D32f pt,
                                           CvSubdiv2DEdge *edge,
                                           CvSubdiv2DPoint** vertex=0 );
```

subdiv
: Delaunay or another subdivision.

pt
: The point to locate.

edge
: The output edge the point falls onto or right to.

vertex
: Optional output vertex double pointer the input point coinsides with.

The function cvSubdiv2DLocate [p 225] locates input point within subdivision. There are 5 cases:

- point falls into some facet. The function returns CV_PTLOC_INSIDE and **edge* will contain one of edges of the facet.
- point falls onto the edge. The function returns CV_PTLOC_ON_EDGE and **edge* will contain this edge.
- point coinsides with one of subdivision vertices. The function returns CV_PTLOC_VERTEX and **vertex* will contain pointer to the vertex.
- point is outside the subdivsion reference rectangle. The function returns CV_PTLOC_OUTSIDE_RECT and no pointers is filled.
- one of input arguments is invalid. Runtime error is raised or, if silent or "parent" error processing mode is selected, CV_PTLOC_ERROR is returnd.

FindNearestPoint2D

Finds the closest subdivision vertex to given point

```
CvSubdiv2DPoint* cvFindNearestPoint2D( CvSubdiv2D* subdiv, CvPoint2D32f pt );
```

subdiv
: Delaunay or another subdivision.

pt
: Input point.

The function cvFindNearestPoint2D [p 226] is another function that locates input point within subdivision. It finds subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using cvSubdiv2DLocate [p 225]) is used as a starting point. The function returns pointer to the found subdivision vertex

---

CalcSubdivVoronoi2D

Calculates coordinates of Voronoi diagram cells

```
void cvCalcSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv
: Delaunay subdivision, where all the points are added already.

The function cvCalcSubdivVoronoi2D [p 226] calculates coordinates of virtual points. All virtual points corresponding to some vertex of original subdivision form (when connected together) a boundary of Voronoi cell of that point.

ClearSubdivVoronoi2D

Removes all virtual points

```
void cvClearSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv
: Delaunay subdivision.

The function cvClearSubdivVoronoi2D [p 226] removes all virtual points. It is called internally in cvCalcSubdivVoronoi2D [p 226] if the subdivision was modified after previous call to the function.

There are a few other lower-level functions that work with planar subdivisions, see cv.h and the sources. Demo script delaunay.c that builds Delaunay triangulation and Voronoi diagram of random 2d point set can be found at opencv/samples/c.

Motion Analysis and Object Tracking Reference

Accumulation of Background Statistics

Acc

Adds frame to accumulator

```
void cvAcc( const CvArr* I, CvArr* S, const CvArr* mask=0 );
```

I
: Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently).

S
: Accumulator of the same number of channels as input image, 32-bit or 64-bit floating-point.

mask
: Optional operation mask.

The function cvAcc [p 227] adds the whole image *I* or its selected region to accumulator *S*:

```
S(x,y)=S(x,y)+I(x,y) if mask(x,y)!=0
```

SquareAcc

Adds the square of source image to accumulator

```
void cvSquareAcc( const CvArr* img, CvArr* sqSum, const CvArr* mask=0 );
```

I
: Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently).

Sq
: Accumulator of the same number of channels as input image, 32-bit or 64-bit floating-point.

mask
: Optional operation mask.

The function cvSquareAcc [p 228] adds the square of input image *I* or its selected region to accumulator *Sq*:

$$Sq(x,y)=Sq(x,y)+I(x,y)^2 \text{ if } mask(x,y)!=0$$

MultiplyAcc

Adds product of two input images to accumulator

```
void cvMultiplyAcc( const CvArr* I, const CvArr* J, CvArr* Sp, const CvArr* mask=0 );
```

I
: First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently).

J
: Second input image, the same format as *I*.

Sp
: Accumulator of the same number of channels as input images, 32-bit or 64-bit floating-point.

mask
: Optional operation mask.

The function cvMultiplyAcc [p 228] adds product of the whole images *I* and *J* or their selected regions to accumulator *Sp*:

```
Sp(x,y)=Sp(x,y)+I(x,y)•J(x,y) if mask(x,y)!=0
```

RunningAvg

Updates running average

```
void cvRunningAvg( const CvArr* I, CvArr* R, double alpha, const CvArr* mask=0 );
```

I
: Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently).

R
: Accumulator of the same number of channels as input image, 32-bit or 64-bit floating-point.

alpha
: Weight of input image.

mask
: Optional operation mask.

The function cvRunningAvg [p 229] calculates weighted sum of input image *I* and accumulator *R* so that *R* becomes a running average of frame sequence:

```
R(x,y)=(1-α)•R(x,y) + α•I(x,y) if mask(x,y)!=0
```

where α (alpha) regulates update speed (how fast accumulator forgets about previous frames).

Motion Templates

UpdateMotionHistory

Updates motion history image by moving silhouette

```
void cvUpdateMotionHistory( const CvArr* S, CvArr* MHI,
                            double timestamp, double duration );
```

S
: Silhouette mask that has non-zero pixels where the motion occurs.

MHI
: Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp
: Current time in milliseconds or other units.

duration
: Maximal duration of motion track in the same units as *timestamp*.

The function cvUpdateMotionHistory [p 229] updates the motion history image as following:

```
MHI(x,y)=timestamp  if S(x,y)!=0
         0          if S(x,y)=0 and MHI(x,y)<timestamp-duration
         MHI(x,y)   otherwise
```

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

CalcMotionGradient

Calculates gradient orientation of motion history image

```
void cvCalcMotionGradient( const CvArr* MHI, CvArr* mask, CvArr* orientation,
                           double delta1, double delta2, int apertureSize=3 );
```

MHI
: Motion history image.

mask
: Mask image; marks pixels where motion gradient data is correct. Output parameter.

orientation
: Motion gradient orientation image; contains angles from 0 to ~360°.

delta1, delta2
: The function finds minimum (m(x,y)) and maximum (M(x,y)) MHI values over each pixel (x,y) neihborhood and assumes the gradient is valid only if

  ```
  min(delta1,delta2) <= M(x,y)-m(x,y) <= max(delta1,delta2).
  ```

apertureSize
: Aperture size of derivative operators used by the function: CV_SCHARR, 1, 3, 5 or 7 (see cvSobel [p 152]).

The function cvCalcMotionGradient [p 230] calculates the derivatives *Dx* and *Dy* of *MHI* and then calculates gradient orientation as:

```
orientation(x,y)=arctan(Dy(x,y)/Dx(x,y))
```

where both *Dx(x,y)*' and *Dy(x,y)*' signs are taken into account (as in cvCartToPolar [p 90] function). After that *mask* is filled to indicate where the orientation is valid (see *delta1* and *delta2* description).

CalcGlobalOrientation

Calculates global motion orientation of some selected region

```
double cvCalcGlobalOrientation( const CvArr* orientation, const CvArr* mask, const CvArr* MHI,
                                double currTimestamp, double mhiDuration );
```

orientation
: Motion gradient orientation image; calculated by the function cvCalcMotionGradient [p 230] .

mask
: Mask image. It may be a conjunction of valid gradient mask, obtained with cvCalcMotionGradient [p 230] and mask of the region, whose direction needs to be calculated.

MHI
: Motion history image.

timestamp
: Current time in milliseconds or other units, it is better to store time passed to cvUpdateMotionHistory [p 229] before and reuse it here, because running cvUpdateMotionHistory [p 229] and cvCalcMotionGradient [p 230] on large images may take some time.

duration
: Maximal duration of motion track in milliseconds, the same as in cvUpdateMotionHistory [p 229] .

The function cvCalcGlobalOrientation [p 230] calculates the general motion direction in the selected region and returns the angle between 0° and 360°. 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 a circular sum of the basic orientation and the shift.

SegmentMotion

Segments whole motion into separate moving parts

```
CvSeq* cvSegmentMotion( const CvArr* MHI, CvArr* segMask, CvMemStorage* storage,
                        double timestamp, double segthresh );
```

mhi
: Motion history image.

segMask
: Image where the mask found should be stored, single-channel, 32-bit floating-point.

storage
: Memory storage that will contain a sequence of motion connected components.

timestamp
: Current time in milliseconds or other units.

segthresh
: Segmentation threshold; recommended to be equal to the interval between motion history "steps" or greater.

The function cvSegmentMotion [p 231] finds all the motion segments and marks them in *segMask* with individual values each (1,2,...). It also returns a sequence of CvConnectedComp [p 175] structures, one per each motion components. After than the motion direction for every component can be calculated with cvCalcGlobalOrientation [p 230] using extracted mask of the particular component (using cvCmp [p 72])

Object Tracking

MeanShift

Finds object center on back projection

```
int cvMeanShift( const CvArr* imgProb, CvRect windowIn,
                 CvTermCriteria criteria, CvConnectedComp* comp );
```

imgProb
: Back projection of object histogram (see cvCalcBackProject [p 198]).

windowIn
: Initial search window.

criteria
: Criteria applied to determine when the window search should be finished.

comp
: Resultant structure that contains converged search window coordinates (*comp->rect* field) and sum of all pixels inside the window (*comp->area* field).

The function cvMeanShift [p 232] iterates to find the object center given its back projection and initial position of search window. The iterations are made until the search window center moves by less than the given value and/or until the function has done the maximum number of iterations. The function returns the number of iterations made.

CamShift

Finds object center, size, and orientation

```
int cvCamShift( const CvArr* imgProb, CvRect windowIn, CvTermCriteria criteria,
                CvConnectedComp* comp, CvBox2D* box=0 );
```

imgProb
: Back projection of object histogram (see cvCalcBackProject [p 198]).

windowIn
: Initial search window.

criteria
: Criteria applied to determine when the window search should be finished.

comp
: Resultant structure that contains converged search window coordinates (*comp->rect* field) and sum of all pixels inside the window (*comp->area* field).

box
: Circumscribed box for the object. If not *NULL*, contains object size and orientation.

The function cvCamShift [p 232] implements CAMSHIFT object tracking algrorithm ([Bradski98] [p 237]). First, it finds an object center using cvMeanShift [p 232] and, after that, calculates the object size and orientation. The function returns number of iterations made within cvMeanShift [p 232] .

CvCamShiftTracker [p ??] class declared in cv.hpp implements color object tracker that uses the function.

[Bradski98] G.R. Bradski. Computer vision face tracking as a component of a perceptual user interface. In Workshop on Applications of Computer Vision, pages 214219, Princeton, NJ, Oct. 1998.
Updated version can be viewed online at http://www.intel.com/technology/itj/q21998/articles/art_2.htm. Also, it is included into OpenCV distribution (camshift.pdf)

SnakeImage

Changes contour position to minimize its energy

```
void cvSnakeImage( const IplImage* image, CvPoint* points, int length,
                   float* alpha, float* beta, float* gamma, int coeffUsage,
                   CvSize win, CvTermCriteria criteria, int calcGradient=1 );
```

image
: The source image or external energy field.

points
: Contour points (snake).

length
: Number of points in the contour.

alpha
: Weight[s] of continuity energy, single float or array of *length* floats, one per each contour point.

beta
: Weight[s] of curvature energy, similar to *alpha*.

gamma
: Weight[s] of image energy, similar to *alpha*.

coeffUsage
: Variant of usage of the previous three parameters:
  - *CV_VALUE* indicates that each of *alpha, beta, gamma* is a pointer to a single value to be used for all points;
  - *CV_ARRAY* indicates that each of *alpha, beta, gamma* is a pointer to an array of coefficients different for all the points of the snake. All the arrays must have the size equal to the contour size.

win
: Size of neighborhood of every point used to search the minimum, both *win.width* and *win.height* must be odd.

criteria
: Termination criteria.

calcGradient
: Gradient flag. If not 0, the function calculates gradient magnitude for every image pixel and consideres it as the energy field, otherwise the input image itself is considered.

The function cvSnakeImage [p 233] updates snake in order to minimize its total energy that is a sum of internal energy that depends on contour shape (the smoother contour is, the smaller internal energy is) and external energy that depends on the energy field and reaches minimum at the local energy extremums that correspond to the image edges in case of image gradient.

The parameter *criteria.epsilon* is used to define the minimal number of points that must be moved during any iteration to keep the iteration process running.

If at some iteration the number of moved points is less than *criteria.epsilon* or the function performed *criteria.maxIter* iterations, the function terminates.

Optical Flow

CalcOpticalFlowHS

Calculates optical flow for two images

```
void cvCalcOpticalFlowHS( const CvArr* imgA, const CvArr* imgB, int usePrevious,
                          CvArr* velx, CvArr* vely, double lambda,
                          CvTermCriteria criteria );
```

imgA
: First image, 8-bit, single-channel.

imgB
: Second image, 8-bit, single-channel.

usePrevious
: Uses previous (input) velocity field.

velx
: Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.

vely
: Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.

lambda
: Lagrangian multiplier.

criteria
: Criteria of termination of velocity computing.

The function cvCalcOpticalFlowHS [p 234] computes flow for every pixel of the first input image using Horn & Schunck algorithm [Horn81] [p ??] .

[Horn81] Berthold K.P. Horn and Brian G. Schunck. Determining Optical Flow. Artificial Intelligence, 17, pp. 185-203, 1981.

CalcOpticalFlowLK

Calculates optical flow for two images

```
void cvCalcOpticalFlowLK( const CvArr* imgA, const CvArr* imgB, CvSize winSize,
                          CvArr* velx, CvArr* vely );
```

imgA
: First image, 8-bit, single-channel.

imgB
: Second image, 8-bit, single-channel.

winSize
: Size of the averaging window used for grouping pixels.

velx
: Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.

vely
: Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel.

The function cvCalcOpticalFlowLK [p 235] computes flow for every pixel of the first input image using Lucas & Kanade algorithm [Lucas81] [p 235] .

[Lucas81] Lucas, B., and Kanade, T. An Iterative Image Registration Technique with an Application to Stereo Vision, Proc. of 7th International Joint Conference on Artificial Intelligence (IJCAI), pp. 674-679.

CalcOpticalFlowBM

Calculates optical flow for two images by block matching method

```
void cvCalcOpticalFlowBM( const CvArr* imgA, const CvArr* imgB, CvSize blockSize,
                          CvSize shiftSize, CvSize maxRange, int usePrevious,
                          CvArr* velx, CvArr* vely );
```

imgA
: First image, 8-bit, single-channel.

imgB
: Second image, 8-bit, single-channel.

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 of floor((imgA->width - blockSize.width)/shiftSize.width) × floor((imgA->height - blockSize.height)/shiftSize.height) size, 32-bit floating-point, single-channel.
vely
Vertical component of the optical flow of the same size *velx*, 32-bit floating-point, single-channel.

The function cvCalcOpticalFlowBM [p 235] calculates optical flow for overlapped blocks *blockSize.width*×*blockSize.height* pixels each, thus the velocity fields are smaller than the original images. For every block in *imgA* the functions tries to find a similar block in *imgB* in some neighborhood of the original block or shifted by (velx(x0,y0),vely(x0,y0)) block as has been calculated by previous function call (if *usePrevious=1*)

CalcOpticalFlowPyrLK

Calculates optical flow for a sparse feature set using iterative Lucas-Kanade method in pyramids

```
void cvCalcOpticalFlowPyrLK( const CvArr* imgA, const CvArr* imgB, CvArr* pyrA, CvArr* pyrB,
                             CvPoint2D32f* featuresA, CvPoint2D32f* featuresB,
                             int count, CvSize winSize, int level, char* status,
                             float* error, CvTermCriteria criteria , int flags );
```

imgA
First frame, at time *t*.
imgB
Second frame, at time *t* + *dt* .
pyrA
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 *(imgSize.width +8)* imgSize.height /3* bytes is sufficient.
pyrB
Similar to *pyrA* , applies to the second frame.
featuresA
Array of points for which the flow needs to be found.
featuresB
Array of 2D points containing calculated new positions of input
features
in the second image.
count
Number of feature points.
winSize
Size of the search window of each pyramid level.
level
Maximal pyramid level number. If *0* , pyramids are not used (single level), if *1* , two levels are used, etc.

status
: Array. Every element of the array is set to *1* if the flow for the corresponding feature has been found, *0* otherwise.

error
: Array of double numbers containing difference between patches around the original and moved points. Optional parameter; can be *NULL* .

criteria
: Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped.

flags
: Miscellaneous flags:
  - *CV_LKFLOW_PYR_A_READY* , pyramid for the first frame is precalculated before the call;
  - *CV_LKFLOW_PYR_B_READY* , pyramid for the second frame is precalculated before the call;
  - *CV_LKFLOW_INITIAL_GUESSES* , array B contains initial coordinates of features before the function call.

The function cvCalcOpticalFlowPyrLK [p 236] implements sparse iterative version of Lucas-Kanade optical flow in pyramids ([Bouguet00] [p ??]). Calculates the optical flow between two images for the given set of points. The function finds the flow with sub-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, the function calculates the pyramid and stores it in the buffer unless the flag *CV_LKFLOW_PYR_A[B]_READY* is set. The image should be large enough to fit the Gaussian pyramid data. After the function call both pyramids are calculated and the ready flag for the corresponding image can be set in the next call.

[Bouguet00] Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker. The paper is included into OpenCV distribution (algo_tracking.pdf)

Estimators

CvKalman

Kalman filter state

```
typedef struct CvKalman
{
    int MP;                     /* number of measurement vector dimensions */
    int DP;                     /* number of state vector dimensions */
    int CP;                     /* number of control vector dimensions */

    /* backward compatibility fields */
#if 1
    float* PosterState;         /* =state_pre->data.fl */
    float* PriorState;          /* =state_post->data.fl */
    float* DynamMatr;           /* =transition_matrix->data.fl */
```

```
    float* MeasurementMatr;     /* =measurement_matrix->data.fl */
    float* MNCovariance;        /* =measurement_noise_cov->data.fl */
    float* PNCovariance;        /* =process_noise_cov->data.fl */
    float* KalmGainMatr;        /* =gain->data.fl */
    float* PriorErrorCovariance;/* =error_cov_pre->data.fl */
    float* PosterErrorCovariance;/* =error_cov_post->data.fl */
    float* Temp1;               /* temp1->data.fl */
    float* Temp2;               /* temp2->data.fl */
#endif

    CvMat* state_pre;           /* predicted state (x'(k)):
                                   x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* corrected state (x(k)):
                                   x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* state transition matrix (A) */
    CvMat* control_matrix;      /* control matrix (B)
                                   (it is not used if there is no control)*/
    CvMat* measurement_matrix;  /* measurement matrix (H) */
    CvMat* process_noise_cov;   /* process noise covariance matrix (Q) */
    CvMat* measurement_noise_cov; /* measurement noise covariance matrix (R) */
    CvMat* error_cov_pre;       /* priori error estimate covariance matrix (P'(k)):
                                   P'(k)=A*P(k-1)*At + Q)*/
    CvMat* gain;                /* Kalman gain matrix (K(k)):
                                   K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* posteriori error estimate covariance matrix (P(k)):
                                   P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

The structure CvKalman [p 237] is used to keep Kalman filter state. It is created by cvCreateKalman [p 239] function, updated by cvKalmanPredict [p 239] and cvKalmanCorrect [p 240] functions and released by cvReleaseKalman [p 239] functions. Normally, the structure is used for standard Kalman filter (notation and formulae are borrowed from excellent Kalman tutorial [Welch95] [p ??]):

$x_k = A \bullet x_{k-1} + B \bullet u_k + w_k$
$z_k = H \bullet x_k + v_k ,$

where:

x_k (x_{k-1}) - state of the system at the moment k (k-1)
z_k - measurement of the system state at the moment k
u_k - external control applied at the moment k

w_k and v_k are normally-distributed process and measurement noise, respectively:
p(w) ~ N(0,Q)
p(v) ~ N(0,R),

that is,
Q - process noise covariance matrix, constant or variable,
R - measurement noise covariance matrix, constant or variable

In case of standard Kalman filter, all the matrices: A, B, H, Q and R are initialized once after CvKalman [p 237] structure is allocated via cvCreateKalman [p 239] . However, the same structure and the same functions may be used to simulate extended Kalman filter by linearizing extended Kalman filter equation in the current system state neighborhood, in this case A, B, H (and, probably, Q and R) should be updated on every step.

[Welch95] Greg Welch, Gary Bishop. An Introduction To the Kalman Filter. Technical Report TR95-041, University of North Carolina at Chapel Hill, 1995. Online version is available at http://www.cs.unc.edu/~welch/kalman/kalman_filter/kalman.html

---

CreateKalman

Allocates Kalman filter structure

```
CvKalman* cvCreateKalman( int dynamParams, int measureParams, int controParams=0 );
```

dynamParams
: dimensionality of the state vector

measureParams
: dimensionality of the measurement vector

controlParams
: dimensionality of the control vector

The function cvCreateKalman [p 239] allocates CvKalman [p 237] and all its matrices and initializes them somehow.

ReleaseKalman

Deallocates Kalman filter structure

```
void cvReleaseKalman(CvKalman** kalman );
```

kalman
: double pointer to the Kalman filter structure.

The function cvReleaseKalman [p 239] releases the structure CvKalman [p 237] and all underlying matrices.

KalmanPredict

Estimates subsequent model state

```
const CvMat* cvKalmanPredict( CvKalman* kalman, const CvMat* control=NULL );
#define cvKalmanUpdateByTime cvKalmanPredict
```

kalman
: Kalman filter state.

control
: Control vector (u_k), should be NULL iff there is no external control (*controlParams*=0).

The function cvKalmanPredict [p 239] estimates the subsequent stochastic model state by its current state and stores it at *kalman->state_pre*:

$$x'_k = A \cdot x_k + B \cdot u_k$$
$$P'_k = A \cdot P_{k-1} \cdot A^T + Q,$$

where

x'_k is predicted state (kalman->state_pre),

x_{k-1} is corrected state on the previous step (kalman->state_post)
(should be initialized somehow in the beginning, zero vector by default),

u_k is external control (*control* parameter),

P'_k is priori error covariance matrix (kalman->error_cov_pre)

P_{k-1} is posteriori error covariance matrix on the previous step (kalman->error_cov_post)
(should be initialized somehow in the beginning, identity matrix by default),

The function returns the estimated state.

KalmanCorrect

Adjusts model state

```
void cvKalmanCorrect( CvKalman* kalman, const CvMat* measurement=NULL );
#define cvKalmanUpdateByMeasurement cvKalmanCorrect
```

kalman
: Pointer to the structure to be updated.

measurement
: Pointer to the structure CvMat containing the measurement vector.

The function cvKalmanCorrect [p 240] adjusts stochastic model state on the basis of the given measurement of the model state:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

where

z_k - given measurement (*mesurement* parameter)

K_k - Kalman "gain" matrix.

The function stores adjusted state at *kalman->state_post* and returns it on output.

Example. Using Kalman filter to track a rotating point

```
#include "cv.h"
#include "highgui.h"
#include <math.h>

int main(int argc, char** argv)
{
    /* A matrix data */
    const float A[] = { 1, 1, 0, 1 };

    IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
    CvKalman* kalman = cvCreateKalman( 2, 1, 0 );
    /* state is (phi, delta_phi) - angle and angle increment */
    CvMat* state = cvCreateMat( 2, 1, CV_32FC1 );
    CvMat* process_noise = cvCreateMat( 2, 1, CV_32FC1 );
    /* only phi (angle) is measured */
    CvMat* measurement = cvCreateMat( 1, 1, CV_32FC1 );
    CvRandState rng;
    int code = -1;

    cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

    cvZero( measurement );
    cvNamedWindow( "Kalman", 1 );

    for(;;)
    {
        cvRandSetRange( &rng, 0, 0.1, 0 );
        rng.disttype = CV_RAND_NORMAL;

        cvRand( &rng, state );

        memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
        cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1) );
        cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-5) );
        cvSetIdentity( kalman->measurement_noise_cov, cvRealScalar(1e-1) );
        cvSetIdentity( kalman->error_cov_post, cvRealScalar(1));
        /* choose random initial state */
        cvRand( &rng, kalman->state_post );

        rng.disttype = CV_RAND_NORMAL;

        for(;;)
        {
            #define calc_point(angle)                                      \
                cvPoint( cvRound(img->width/2 + img->width/3*cos(angle)),  \
                         cvRound(img->height/2 - img->width/3*sin(angle)))

            float state_angle = state->data.fl[0];
            CvPoint state_pt = calc_point(state_angle);

            /* predict point position */
            const CvMat* prediction = cvKalmanPredict( kalman, 0 );
            float predict_angle = prediction->data.fl[0];
            CvPoint predict_pt = calc_point(predict_angle);
            float measurement_angle;
            CvPoint measurement_pt;
```

```
            cvRandSetRange( &rng, 0, sqrt(kalman->measurement_noise_cov->data.fl[0]), 0 );
            cvRand( &rng, measurement );

            /* generate measurement */
            cvMatMulAdd( kalman->measurement_matrix, state, measurement, measurement );

            measurement_angle = measurement->data.fl[0];
            measurement_pt = calc_point(measurement_angle);

            /* plot points */
            #define draw_cross( center, color, d )                                \
                cvLine( img, cvPoint( center.x - d, center.y - d ),                \
                             cvPoint( center.x + d, center.y + d ), color, 1, 0 ); \
                cvLine( img, cvPoint( center.x + d, center.y - d ),                \
                             cvPoint( center.x - d, center.y + d ), color, 1, 0 )

            cvZero( img );
            draw_cross( state_pt, CV_RGB(255,255,255), 3 );
            draw_cross( measurement_pt, CV_RGB(255,0,0), 3 );
            draw_cross( predict_pt, CV_RGB(0,255,0), 3 );
            cvLine( img, state_pt, predict_pt, CV_RGB(255,255,0), 3, 0 );

            /* adjust Kalman filter state */
            cvKalmanCorrect( kalman, measurement );

            cvRandSetRange( &rng, 0, sqrt(kalman->process_noise_cov->data.fl[0]), 0 );
            cvRand( &rng, process_noise );
            cvMatMulAdd( kalman->transition_matrix, state, process_noise, state );

            cvShowImage( "Kalman", img );
            code = cvWaitKey( 100 );

            if( code > 0 ) /* break current simulation by pressing a key */
                break;
        }
        if( code == 27 ) /* exit by ESCAPE */
            break;
    }

    return 0;
}
```

CvConDensation

ConDenstation state

```
    typedef struct CvConDensation
    {
        int MP;     //Dimension of measurement vector
        int DP;     // Dimension of state vector
        float* DynamMatr;       // Matrix of the linear Dynamics system
        float* State;           // Vector of State
        int SamplesNum;         // Number of the Samples
        float** flSamples;      // array of the Sample Vectors
        float** flNewSamples;   // temporary array of the Sample Vectors
```

```
    float* flConfidence;    // Confidence for each Sample
    float* flCumulative;    // Cumulative confidence
    float* Temp;            // Temporary vector
    float* RandomSample;    // RandomVector to update sample set
    CvRandState* RandS;     // Array of structures to generate random vectors
} CvConDensation;
```

The structure CvConDensation [p ??] stores CONditional DENSity propagATION tracker state. The information about the algorithm can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/ISARD1/condensation.html

---

CreateConDensation

Allocates ConDensation filter structure

```
CvConDensation* cvCreateConDensation( int DynamParams, int MeasureParams, int SamplesNum );
```

DynamParams
: Dimension of the state vector.

MeasureParams
: Dimension of the measurement vector.

SamplesNum
: Number of samples.

The function cvCreateConDensation [p 243] creates CvConDensation [p ??] structure and returns pointer to the structure.

ReleaseConDensation

Deallocates ConDensation filter structure

```
void cvReleaseConDensation( CvConDensation** ConDens );
```

ConDens
: Pointer to the pointer to the structure to be released.

The function cvReleaseConDensation [p 243] releases the structure CvConDensation [p ??] (see cvConDensation [p ??]) and frees all memory previously allocated for the structure.

ConDensInitSampleSet

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.

The function cvConDensInitSampleSet [p 243] fills the samples arrays in the structure CvConDensation [p ??] with values within specified ranges.

ConDensUpdateByTime

Estimates subsequent model state

```
void cvConDensUpdateByTime( CvConDensation* ConDens );
```

ConDens
Pointer to the structure to be updated.

The function cvConDensUpdateByTime [p 244] estimates the subsequent stochastic model state from its current state.

Object Recognition Reference

Eigen Objects (PCA) Functions

The functions described in this section do PCA analysis and compression for a set of 8-bit images that may not fit into memory all together. If your data fits into memory and the vectors are not 8-bit (or you want a simpler interface), use cvCalcCovarMatrix [p 89] , cvSVD [p 86] and cvGEMM [p 82] to do PCA

CalcCovarMatrixEx

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 );
```

nObjects
: Number of source objects.

input
Pointer either to the array of *IplImage* input objects or to the read callback function according to the value of the parameter *ioFlags*.
ioFlags
Input/output flags.
ioBufSize
Input/output buffer size.
buffer
Pointer to the input/output buffer.
userData
Pointer to the structure that contains all necessary data for the
callback
functions.
avg
Averaged object.
covarMatrix
Covariance matrix. An output parameter; must be allocated before the call.

The function cvCalcCovarMatrixEx [p 245] 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 calling the function.

CalcEigenObjects

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 );
```

nObjects
Number of source objects.
input
Pointer either to the array of *IplImage* input objects or to the read callback function according to the value of the parameter *ioFlags*.
output
Pointer either to the array of eigen objects or to the write callback function according to the value of the parameter ioFlags .
ioFlags
Input/output flags.
ioBufSize
Input/output buffer size in bytes. The size is zero, if unknown.
userData
Pointer to the structure that contains all necessary data for the callback functions.

calcLimit
: Criteria that determine when to stop calculation of eigen objects.

avg
: Averaged object.

eigVals
: Pointer to the eigenvalues array in the descending order; may be *NULL* .

The function cvCalcEigenObjects [p 246] calculates orthonormal eigen basis and the averaged object for a group of the 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 after first *calcLimit.maxIters* dominating eigen objects are retrieved or if the ratio of the current eigenvalue to the largest eigenvalue comes down to *calcLimit.epsilon* threshold. 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* .

The function also calculates the averaged object, which must be created previously. Calculated eigen objects are arranged 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 [p 246] uses the function cvCalcCovarMatrixEx [p 245] .

CalcDecompCoeff

Calculates decomposition coefficient of input object

```
double cvCalcDecompCoeff( IplImage* obj, IplImage* eigObj, IplImage* avg );
```

obj
: Input object.

eigObj
: Eigen object.

avg
: Averaged object.

The function cvCalcDecompCoeff [p 247] calculates one decomposition coefficient of the input object using the previously calculated eigen object and the averaged object.

EigenDecomposite

Calculates all decomposition coefficients for input object

```
void cvEigenDecomposite( IplImage* obj, int nEigObjs, void* eigInput,
                         int ioFlags, void* userData, IplImage* avg, float* coeffs );
```

obj
Input object.
nEigObjs
Number of eigen objects.
eigInput
Pointer either to the array of *IplImage* input objects or to the read callback function according to the value of the parameter *ioFlags*.
ioFlags
Input/output flags.
userData
Pointer to the structure that contains all necessary data for the callback functions.
avg
Averaged object.
coeffs
Calculated coefficients; an output parameter.

The function cvEigenDecomposite [p 247] 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.

EigenProjection

Calculates object projection to the eigen sub-space

```
void cvEigenProjection( int nEigObjs, void* eigInput, int ioFlags,
                        void* userData, float* coeffs,
                        IplImage* avg, IplImage* proj );
```

nEigObjs
Number of eigen objects.
eigInput
Pointer either to the array of *IplImage* input objects or to the read callback function according to the value of the parameter *ioFlags*.
ioFlags
Input/output flags.
userData
Pointer to the structure that contains all necessary data for the callback functions.
coeffs
Previously calculated decomposition coefficients.
avg
Averaged object.
proj
Decomposed object projection to the eigen sub-space.

The function cvEigenProjection [p 248] 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.

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.

Embedded Hidden Markov Models Functions

In order to support embedded models the user must define structures to represent 1D HMM and 2D embedded HMM model.

CvHMM

Embedded HMM Structure

```
typedef struct _CvEHMM
{
    int level;
    int num_states;
    float* transP;
    float** obsProb;
    union
    {
        CvEHMMState* state;
        struct _CvEHMM* ehmm;
    } u;
} CvEHMM;
```

level
: Level of embedded HMM. If *level ==0*, HMM is most external. In 2D HMM there are two types of HMM: 1 external and several embedded. External HMM has *level ==1*, embedded HMMs have *level ==0* .

num_states
: Number of states in 1D HMM.

transP
: State-to-state transition probability, square matrix *(num_state×num_state)*.

obsProb
: Observation probability matrix.

state
: Array of HMM states. For the last-level HMM, that is, an HMM without embedded HMMs, HMM states are real.

ehmm
: 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:

CvImgObsInfo

Image Observation Structure

```
typedef struct CvImgObsInfo
{
    int obs_x;
    int obs_y;
    int obs_size;
    float** obs;
    int* state;
    int* mix;
} CvImgObsInfo;
```

obs_x
: Number of observations in the horizontal direction.

obs_y
: Number of observations in the vertical direction.

obs_size
: Length of every observation vector.

obs
: Pointer to observation vectors stored consequently. Number of vectors is *obs_x*obs_y* .

state
: Array of indices of states, assigned to every observation vector.

mix
: Index of mixture component, corresponding to the observation vector within an assigned state.

Create2DHMM

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 are not counted here.

obsSize
: Size of observation vectors to be used with created HMM.

The function cvCreate2DHMM [p 250] returns the created structure of the type CvEHMM [p ??] with specified parameters.

Release2DHMM

Releases 2D embedded HMM

```
void cvRelease2DHMM(CvEHMM** hmm );
```

hmm
: Address of pointer to HMM to be released.

The function cvRelease2DHMM [p 251] frees all the memory used by HMM and clears the pointer to HMM.

CreateObsInfo

Creates structure to store image observation vectors

```
CvImgObsInfo* cvCreateObsInfo( CvSize numObs, int obsSize );
```

numObs
: 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 *CV_COUNT_OBS(roi, dctSize, delta, numObs)*, where *roi, dctSize, delta, numObs* are the pointers to structures of the type CvSize [p ??] . The pointer *roi* means size of *roi* of image observed, *numObs* is the output parameter of the macro.

obsSize
: Size of observation vectors to be stored in the structure.

The function cvCreateObsInfo [p 251] 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 [p 252] .

ReleaseObsInfo

Releases observation vectors structure

```
void cvReleaseObsInfo( CvImgObsInfo** obsInfo );
```

obsInfo
: Address of the pointer to the structure CvImgObsInfo [p 250] .

The function cvReleaseObsInfo [p 251] frees all memory used by observations and clears pointer to the structure CvImgObsInfo [p 250] .

ImgToObs_DCT

Extracts observation vectors from image

```
void cvImgToObs_DCT( IplImage* image, float* obs, CvSize dctSize,
                     CvSize obsSize, CvSize delta );
```

image
: Input image.

obs
: Pointer to consequently stored observation vectors.

dctSize
: Size of image blocks for which DCT (Discrete Cosine Transform) coefficients are to be computed.

obsSize
: Number of the lowest DCT coefficients in the horizontal and vertical directions to be put into the observation vector.

delta
: Shift in pixels between two consecutive image blocks in the horizontal and vertical directions.

The function cvImgToObs_DCT [p 252] extracts observation vectors, that is, DCT coefficients, from the image. The user must pass *obsInfo.obs* as the parameter *obs* to use this function with other HMM functions and use the structure *obsInfo* of the CvImgObsInfo [p 250] type.

Calculating Observations for HMM

```
CvImgObsInfo* obs_info;

    ...

    cvImgToObs_DCT( image,obs_info->obs, //!!!

    dctSize, obsSize, delta );
```

UniformImgSegm

Performs uniform segmentation of image observations by HMM states

```
void cvUniformImgSegm( CvImgObsInfo* obsInfo, CvEHMM* hmm );
```

obsInfo
: Observations structure.

hmm
: HMM structure.

The function cvUniformImgSegm [p 252] segments image observations by HMM states uniformly (see Initial Segmentation for 2D Embedded HMM for 2D embedded HMM with 5 superstates and 3, 6, 6, 6, 3 internal states of every corresponding superstate).

Initial Segmentation for 2D Embedded HMM

InitMixSegm

Segments all observations within every internal state of HMM by state mixture components

```
void cvInitMixSegm( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray
: Array of pointers to the observation structures.

numImg
: Length of above array.

hmm
: HMM.

The function cvInitMixSegm [p 253] 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 the number of mixture components in the state.

EstimateHMMStateParams

Estimates all parameters of every HMM state

```
void cvEstimateHMMStateParams( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray
: Array of pointers to the observation structures.

numImg
: Length of the array.

hmm
HMM.

The function cvEstimateHMMStateParams [p 253] computes all inner parameters of every HMM state, including Gaussian means, variances, etc.

EstimateTransProb

Computes transition probability matrices for embedded HMM

```
void cvEstimateTransProb( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray
Array of pointers to the observation structures.
numImg
Length of the above array.
hmm
HMM.

The function cvEstimateTransProb [p 254] uses current segmentation of image observations to compute transition probability matrices for all embedded and external HMMs.

EstimateObsProb

Computes probability of every observation of several images

```
void cvEstimateObsProb( CvImgObsInfo* obsInfo, CvEHMM* hmm );
```

obsInfo
Observation structure.
hmm
HMM structure.

The function cvEstimateObsProb [p 254] computes Gaussian probabilities of each observation to occur in each of the internal HMM states.

EViterbi

Executes Viterbi algorithm for embedded HMM

```
float cvEViterbi( CvImgObsInfo* obsInfo, CvEHMM* hmm );
```

obsInfo
Observation structure.

hmm
HMM structure.

The function cvEViterbi [p 254] executes Viterbi algorithm for embedded HMM. Viterbi algorithm evaluates the likelihood of the best match between the given image observations and the given HMM and performs segmentation of image observations by HMM states. The segmentation is done on the basis of the match found.

MixSegmL2

Segments observations from all training images by mixture components of newly assigned states

```
void cvMixSegmL2( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray
Array of pointers to the observation structures.
numImg
Length of the array.
hmm
HMM.

The function cvMixSegmL2 [p 255] segments observations from all training images by mixture components of newly Viterbi algorithm-assigned states. The function uses Euclidean distance to group vectors around the existing mixtures centers.

Camera Calibration and 3D Reconstruction Reference

Camera Calibration Functions

CalibrateCamera

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 );
```

numImages
Number of the images.
numPoints
Array of the number of points in each image.
imageSize
Size of the image.
imagePoints32f
Pointer to the images.
objectPoints32f
Pointer to the pattern.
distortion32f
Array of four distortion coefficients found.
cameraMatrix32f
Camera matrix found.
transVects32f
Array of translate vectors for each pattern position in the image.
rotMatrs32f
Array of the rotation matrix for each pattern position in the image.
useIntrinsicGuess
Intrinsic guess. If equal to 1, intrinsic guess is needed.

The function cvCalibrateCamera [p 256] calculates the camera parameters using information points on the pattern object and pattern object images.

CalibrateCamera_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 );
```

numImages
Number of the images.
numPoints
Array of the number of points in each image.
imageSize
Size of the image.
imagePoints
Pointer to the images.
objectPoints
Pointer to the pattern.
distortion
Distortion coefficients found.

cameraMatrix
Camera matrix found.
transVects
Array of the translate vectors for each pattern position on the image.
rotMatrs
Array of the rotation matrix for each pattern position on the image.
useIntrinsicGuess
Intrinsic guess. If equal to 1, intrinsic guess is needed.

The function cvCalibrateCamera_64d [p 257] is basically the same as the function cvCalibrateCamera [p 256] , but uses double precision.

Rodrigues

Converts rotation matrix to rotation vector and vice versa with single precision

```
void  cvRodrigues( CvMat* rotMatrix, CvMat* rotVector,
                   CvMat* jacobian, int convType);
```

rotMatrix
Rotation matrix (3x3), 32-bit or 64-bit floating point.
rotVector
Rotation vector (3x1 or 1x3) of the same type as *rotMatrix*.
jacobian
Jacobian matrix 3 × 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.

The function cvRodrigues [p 258] converts the rotation matrix to the rotation vector or vice versa.

UnDistortOnce

Corrects camera lens distortion

```
void cvUnDistortOnce( const CvArr* srcImage, CvArr* dstImage,
                      const float* intrMatrix,
                      const float* distCoeffs,
                      int interpolate=1 );
```

srcImage
Source (distorted) image.
dstImage
Destination (corrected) image.
intrMatrix
Matrix of the camera intrinsic parameters (3x3).

distCoeffs
Vector of the four distortion coefficients k_1, k_2, p_1 and p_2.

interpolate
Bilinear interpolation flag.

The function cvUnDistortOnce [p 258] corrects camera lens distortion in case of a single image. Matrix of the camera intrinsic parameters and distortion coefficients k_1, k_2 , p_1 , and p_2 must be preliminarily calculated by the function cvCalibrateCamera [p 256] .

UnDistortInit

Calculates arrays of distorted points indices and interpolation coefficients

```
void cvUnDistortInit( const CvArr* srcImage, CvArr* undistMap,
                      const float* intrMatrix,
                      const float* distCoeffs,
                      int interpolate=1 );
```

srcImage
Artibtrary source (distorted) image, the image size and number of channels do matter.

undistMap
32-bit integer image of the same size as the source image (if *interpolate=0*) or 3 times wider than the source image (if *interpolate=1*).

intrMatrix
Matrix of the camera intrinsic parameters.

distCoeffs
Vector of the 4 distortion coefficients k_1, k_2, p_1 and p_2.

interpolate
Bilinear interpolation flag.

The function cvUnDistortInit [p 259] calculates arrays of distorted points indices and interpolation coefficients using known matrix of the camera intrinsic parameters and distortion coefficients. It calculates undistortion map for cvUnDistort [p 259] .

Matrix of the camera intrinsic parameters and the distortion coefficients may be calculated by cvCalibrateCamera [p 256] .

UnDistort

Corrects camera lens distortion

```
void cvUnDistort( const void* srcImage, void* dstImage,
                  const void* undistMap, int interpolate=1 );
```

srcImage
Source (distorted) image.
dstImage
Destination (corrected) image.
undistMap
Undistortion map, pre-calculated by cvUnDistortInit [p 259] .
interpolate
Bilinear interpolation flag, the same as in cvUnDistortInit [p 259] .

The function cvUnDistort [p 259] corrects camera lens distortion using previously calculated undistortion map. It is faster than cvUnDistortOnce [p 258] .

FindChessBoardCornerGuesses

Finds approximate positions of internal corners of the chessboard

```
int cvFindChessBoardCornerGuesses( IplImage* img, IplImage* thresh, CvSize etalonSize,
                                   CvPoint2D32f* corners, int* cornerCount );
```

img
Source chessboard view; must have the depth of *IPL_DEPTH_8U*.
thresh
Temporary image of the same size and format as the source image.
etalonSize
Number of inner corners per chessboard row and column. The width (the number of columns) must be less or equal to the height (the number of rows).
corners
Pointer to the corner array found.
cornerCount
Signed value whose absolute value is the 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.

The function cvFindChessBoardCornerGuesses [p 260] attempts to determine whether the input image is a view of the 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 the corners or reorder them, the function returns 0. For example, a simple chessboard has 8 x 8 squares and 7 x 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 [p ??] .

Pose Estimation

FindExtrinsicCameraParams

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 );
```

numPoints
: Number of the points.

ImageSize
: Size of the image.

imagePoints32f
: Pointer to the image.

objectPoints32f
: Pointer to the pattern.

focalLength32f
: Focal length.

principalPoint32f
: Principal point.

distortion32f
: Distortion.

rotVect32f
: Rotation vector.

transVect32f
: Translate vector.

The function cvFindExtrinsicCameraParams [p 261] finds the extrinsic parameters for the pattern.

FindExtrinsicCameraParams_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 );
```

numPoints
: Number of the points.

ImageSize
Size of the image.
imagePoints
Pointer to the image.
objectPoints
Pointer to the pattern.
focalLength
Focal length.
principalPoint
Principal point.
distortion
Distortion.
rotVect
Rotation vector.
transVect
Translate vector.

The function cvFindExtrinsicCameraParams_64d [p 261] finds the extrinsic parameters for the pattern with double precision.

CreatePOSITObject

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.

The function cvCreatePOSITObject [p 262] allocates memory for the object structure and computes the object inverse matrix.

The preprocessed object data is stored in the structure CvPOSITObject [p ??] , 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 [p 263] 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 [p 263] must be called to free memory.

POSIT

Implements POSIT algorithm

```
void cvPOSIT( CvPoint2D32f* imagePoints, CvPOSITObject* pObject,
              double focalLength, CvTermCriteria criteria,
              CvMatrix3* rotation, CvPoint3D32f* translation );
```

imagePoints
: Pointer to the object points projections on the 2D image plane.

pObject
: Pointer to the object structure.

focalLength
: Focal length of the camera used.

criteria
: Termination criteria of the iterative POSIT algorithm.

rotation
: Matrix of rotations.

translation
: Translation vector.

The function cvPOSIT [p 263] implements POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using camera calibration functions. At every iteration of the algorithm new perspective projection of estimated pose is computed.

Difference norm between two projections is the maximal distance between corresponding points. The parameter *criteria.epsilon* serves to stop the algorithm if the difference is small.

ReleasePOSITObject

Deallocates 3D object structure

```
void cvReleasePOSITObject( CvPOSITObject** ppObject );
```

ppObject
: Address of the pointer to the object structure.

The function cvReleasePOSITObject [p 263] releases memory previously allocated by the function cvCreatePOSITObject [p 262] .

CalcImageHomography

Calculates homography matrix for oblong planar object (e.g. arm)

```
void cvCalcImageHomography( float* line, CvPoint3D32f* center,
                            float* intrinsic, float homography[3][3]);
```

line
: the main object axis direction (vector (dx,dy,dz)).

center
: object center ((cx,cy,cz)).

intrinsic
: intrinsic camera parameters (3x3 matrix).

homography
: output homography matrix (3x3).

The function cvCalcImageHomography [p 263] calculates the homography matrix for the initial image transformation from image plane to the plane, defined by 3D oblong object line (See Figure 6-10 in OpenCV Guide 3D Reconstruction Chapter).

View Morphing Functions

MakeScanlines

Calculates scanlines coordinates for two cameras by fundamental matrix

```
void cvMakeScanlines( CvMatrix3* matrix, CvSize imgSize, int* scanlines1,
                      int* scanlines2, int* lens1, int* lens2, int* numlines );
```

matrix
: Fundamental matrix.

imgSize
: Size of the image.

scanlines1
: Pointer to the array of calculated scanlines of the first image.

scanlines2
: Pointer to the array of calculated scanlines of the second image.

lens1
: Pointer to the array of calculated lengths (in pixels) of the first image scanlines.

lens2
: Pointer to the array of calculated lengths (in pixels) of the second image scanlines.

numlines
: Pointer to the variable that stores the number of scanlines.

The function cvMakeScanlines [p 264] 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 *scanlines1* or *scanlines2* are equal to zero.

PreWarpImage

Rectifies image

```
void cvPreWarpImage( int numLines, IplImage* img, uchar* dst,
                     int* dstNums, int* scanlines );
```

numLines
: Number of scanlines for the image.

img
: Image to prewarp.

dst
: Data to store for the prewarp image.

dstNums
: Pointer to the array of lengths of scanlines.

scanlines
: Pointer to the array of coordinates of scanlines.

The function cvPreWarpImage [p 265] rectifies the image so that the scanlines in the rectified image are horizontal. The output buffer of size *max(width,height)*numscanlines*3* must be allocated before calling the function.

FindRuns

Retrieves scanlines from rectified image and breaks them down into runs

```
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 );
```

numLines
: Number of the scanlines.

prewarp_1
: Prewarp data of the first image.

prewarp_2
: Prewarp data of the second image.

lineLens_1
: Array of lengths of scanlines in the first image.

lineLens_2
: Array of lengths of scanlines in the second image.

runs_1
: Array of runs in each scanline in the first image.

runs_2
: Array of runs in each scanline in the second image.

numRuns_1
Array of numbers of runs in each scanline in the first image.
numRuns_2
Array of numbers of runs in each scanline in the second image.

The function cvFindRuns [p 265] retrieves scanlines from the rectified image and breaks each scanline down into several runs, that is, series of pixels of almost the same brightness.

DynamicCorrespondMulti

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 );
```

lines
Number of scanlines.
first
Array of runs of the first image.
firstRuns
Array of numbers of runs in each scanline of the first image.
second
Array of runs of the second image.
secondRuns
Array of numbers of runs in each scanline of the second image.
firstCorr
Pointer to the array of correspondence information found for the first runs.
secondCorr
Pointer to the array of correspondence information found for the second runs.

The function cvDynamicCorrespondMulti [p 266] finds correspondence between two sets of runs of two 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)* .

MakeAlphaScanlines

Calculates coordinates of scanlines of image from virtual camera

```
void cvMakeAlphaScanlines( int* scanlines_1, int* scanlines_2,
                           int* scanlinesA, int* lens,
                           int numlines, float alpha );
```

scanlines_1
Pointer to the array of the first scanlines.
scanlines_2
Pointer to the array of the second scanlines.
scanlinesA
Pointer to the array of the scanlines found in the virtual image.
lens
Pointer to the array of lengths of the scanlines found in the virtual image.
numlines
Number of scanlines.
alpha
Position of virtual camera *(0.0 - 1.0)* .

The function cvMakeAlphaScanlines [p 266] 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)* .

MorphEpilinesMulti

Morphs two pre-warped images using information about stereo correspondence

```
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 );
```

lines
Number of scanlines in the prewarp image.
firstPix
Pointer to the first prewarp image.
firstNum
Pointer to the array of numbers of points in each scanline in the first image.
secondPix
Pointer to the second prewarp image.
secondNum
Pointer to the array of numbers of points in each scanline in the second image.
dstPix
Pointer to the resulting morphed warped image.
dstNum
Pointer to the array of numbers of points in each line.

alpha
Virtual camera position *(0.0 - 1.0)* .
first
First sequence of runs.
firstRuns
Pointer to the number of runs in each scanline in the first image.
second
Second sequence of runs.
secondRuns
Pointer to the number of runs in each scanline in the second image.
firstCorr
Pointer to the array of correspondence information found for the first runs.
secondCorr
Pointer to the array of correspondence information found for the second runs.

The function cvMorphEpilinesMulti [p 267] morphs two pre-warped images using information about correspondence between the scanlines of two images.

PostWarpImage

Warps rectified morphed image back

```
void cvPostWarpImage( int numLines, uchar* src, int* srcNums,
                      IplImage* img, int* scanlines );
```

numLines
Number of the scanlines.
src
Pointer to the prewarp image virtual image.
srcNums
Number of the scanlines in the image.
img
Resulting unwarp image.
scanlines
Pointer to the array of scanlines data.

The function cvPostWarpImage [p 268] warps the resultant image from the virtual camera by storing its rows across the scanlines whose coordinates are calculated by cvMakeAlphaScanlines [p 266] .

DeleteMoire

Deletes moire in given image

```
void cvDeleteMoire( IplImage* img );
```

img
Image.

The function cvDeleteMoire [p 268] deletes moire from the given image. The post-warped image may have black (un-covered) points because of possible holes between neighboring scanlines. The function deletes moire (black pixels) from the image by substituting neighboring pixels for black pixels. If all the scanlines are horizontal, the function may be omitted.

Stereo Correspondence and Epipolar Geometry Functions

FindFundamentalMat

Calculates fundamental matrix from corresponding points in two images

```
int cvFindFundamentalMat( CvMat* points1,
                          CvMat* points2,
                          CvMat* fundMatr,
                          int    method,
                          double param1,
                          double param2,
                          CvMat* status=0);
```

points1
Array of the first image points of 2xN/Nx2 or 3xN/Nx3 size (N is number of points). The point coordinates should be floating-point (single or double precision)
points2
Array of the second image points of the same size and format as *points1*
fundMatr
The output fundamental matrix or matrices. Size 3x3 or 9x3 (7-point method can returns up to 3 matrices).
method
Method for computing fundamental matrix
CV_FM_7POINT - for 7-point algorithm. Number of points == 7
CV_FM_8POINT - for 8-point algorithm. Number of points >= 8
CV_FM_RANSAC - for RANSAC algorithm. Number of points >= 8
CV_FM_LMEDS - for LMedS algorithm. Number of points >= 8
param1
The parameter is used for RANSAC or LMedS methods only. It is the maximum distance from point to epipolar line, beyound which the point is considered bad and is not considered in further calculations. Usually it is set to 0.5 or 1.0.
param2
The parameter is used for RANSAC or LMedS methods only. It denotes the desirable level of confidense the matrix is the correct (up to some precision). It can be set to 0.99 for example.
status
Array of N elements, every element of which is set to 1 if the point was not rejected during the computation, 0 otherwise. The array is computed only in RANSAC and LMedS methods. For other

methods it is set to all 1's. This is the optional parameter.

The epipolar geometry is described by the following equation:

$p_2^T * F * p_1 = 0,$

where F is fundamental matrix, p_1 and p_2 are corresponding points on the two images.

The function *FindFundamentalMat* calculates fundamental matrix using one of four methods listed above and returns the number of fundamental matrix found: 0 if the matrix could not be found, 1 or 3 if the matrix or matrices have been found successfully.

The calculated fundamental matrix may be passed further to *ComputeCorrespondEpilines* function that computes coordinates of corresponding epilines on two images.

For 7-point method uses exactly 7 points. It can find 1 or 3 fundamental matrices. It returns number of the matrices found and if there is a room in the destination array to keep all the detected matrices, stores all of them there, otherwise it stores only one of the matrices.

All other methods use 8 or more points and return a single fundamental matrix.

Example. Fundamental matrix calculation

```
int numPoints = 100;
CvMat* points1;
CvMat* points2;
CvMat* status;
CvMat* fundMatr;

points1  = cvCreateMat(2,numPoints,CV_32F);
points2  = cvCreateMat(2,numPoints,CV_32F);
status   = cvCreateMat(1,numPoints,CV_32F);

/* Fill the points here ... */

fundMatr = cvCreateMat(3,3,CV_32F);
int num = cvFindFundamentalMat(points1,points2,fundMatr,CV_FM_RANSAC,1.0,0.99,status);
if( num == 1 )
{
    printf("Fundamental matrix was found\n");
}
else
{
    printf("Fundamental matrix was not found\n");
}


/*====== Example of code for three matrixes ======*/
CvMat* points1;
CvMat* points2;
CvMat* fundMatr;

points1  = cvCreateMat(2,7,CV_32F);
points2  = cvCreateMat(2,7,CV_32F);
```

```
/* Fill the points here... */

fundMatr = cvCreateMat(9,3,CV_32F);
int num = cvFindFundamentalMat(points1,points2,fundMatr,CV_FM_7POINT,0,0,0);
printf("Found %d matrixes\n",num);
```

ComputeCorrespondEpilines

For every input point on one of image computes the corresponding epiline on the other image

```
void cvComputeCorrespondEpilines( const CvMat* points,
                                  int pointImageID,
                                  CvMat* fundMatr,
                                  CvMat* corrLines);
```

points
: The input points: 2xN or 3xN array (N number of points)

pointImageID
: Image ID there are points are located, 1 or 2

fundMatr
: Fundamental matrix

corrLines
: Computed epilines, 3xN array

The function *ComputeCorrespondEpilines* computes the corresponding epiline for every input point using the basic equation of epipolar line geometry:

If points located on first image (ImageID=1), corresponding epipolar line can be computed as:

$l_2 = F * p_1$

where F is fundamental matrix, p_1 point on first image, l_2 corresponding epipolar line on second image.

If points located on second image (ImageID=2):

$l_1 = F^T * p_2$

where F is fundamental matrix, p_2 point on second image, l_1 corresponding epipolar line on first image

Each epipolar line is present by coefficients a,b,c of line equation:

```
a*x + b*y + c = 0
```

Also computed line normalized by $a^2 + b^2 = 1$. It's useful if distance from point to line must be computed later.

GUI and Video Acquisition Reference

HighGUI overview

TODO

Window functions

cvNamedWindow

Creates a window (image placeholder)

```
int cvNamedWindow( const char* name, unsigned long flags );
```

name
: Name of the window which is used as window identifier and appears in the window caption.

flags
: Defines window properties. Currently the only supported property is ability to automatically change the window size to fit the image being hold by the window. Use CV_WINDOW_AUTOSIZE for enabling the automatical resizing or 0 otherwise.

The function cvNamedWindow [p 273] creates a window which can be used as a placeholder for images and trackbars. Created windows are reffered by their names.

cvDestroyWindow

Destroys a window

```
void cvDestroyWindow( const char* name );
```

name
: Name of the window to be destroyed.

The function cvDestroyWindow [p 273] destroyes the window with the given name.

cvResizeWindow

Sets window sizes

```
void cvResizeWindow( const char* name, int width, int height );
```

name
: Name of the window to be resized.

width
: New width

height
: New height

The function cvResizeWindow [p 273] changes the sizes of the window.

---

cvGetWindowHandle

Gets window handle by name

```
void* cvGetWindowHandle( const char* name );
```

name
: Name of the window.

The function cvGetWindowHandle [p 274] returns native window handle (HWND in case of Win32 and Widget in case of X Window).

---

cvGetWindowName

Gets window name by handle

```
const char* cvGetWindowName( void* window_handle );
```

window_handle
: Handle of the window.

The function cvGetWindowName [p 274] returns the name of window given its native handle(HWND in case of Win32 and Widget in case of X Window).

---

cvCreateTrackbar

Creates the trackbar and attaches it to the specified window

```
CV_EXTERN_C_FUNCPTR( void (*CvTrackbarCallback)(int pos) );

int cvCreateTrackbar( const char* trackbar_name, const char* window_name,
                      int* value, int count, CvTrackbarCallback on_change );
```

trackbar_name
: Name of created trackbar.

window_name
: Name of the window which will be used as a parent for created trackbar.

value
: Pointer to the integer variable, which value will reflect the position of the slider. Upon the creation the slider position is defined by this variable.

count
: Maximal position of the slider. Minimal position is always 0.

on_change
Pointer to the function to be called every time the slider changes the position. This function should be prototyped as
void Foo(int);
Can be NULL if callback is not required.

The function cvCreateTrackbar [p 274] creates the trackbar(slider) with the specified name and range, assigns the variable to be syncronized with trackbar position and specifies callback function to be called on trackbar position change. The created trackbar is displayed on top of given window.

---

cvGetTrackbarPos

Retrieves trackbar position

```
int cvGetTrackbarPos( const char* trackbar_name, const char* window_name );
```

trackbar_name
Name of trackbar.
window_name
Name of the window which is the parent of trackbar.

The function cvGetTrackbarPos [p 275] returns the ciurrent position of the specified trackbar.

---

cvSetTrackbarPos

Sets trackbar position

```
void cvSetTrackbarPos( const char* trackbar_name, const char* window_name, int pos );
```

trackbar_name
Name of trackbar.
window_name
Name of the window which is the parent of trackbar.
pos
New position.

The function cvSetTrackbarPos [p 275] sets the position of the specified trackbar.

---

cvSetMouseCallback

Assigns callback for mouse events

```
#define CV_EVENT_MOUSEMOVE      0
#define CV_EVENT_LBUTTONDOWN    1
#define CV_EVENT_RBUTTONDOWN    2
#define CV_EVENT_MBUTTONDOWN    3
```

```
#define CV_EVENT_LBUTTONUP      4
#define CV_EVENT_RBUTTONUP      5
#define CV_EVENT_MBUTTONUP      6
#define CV_EVENT_LBUTTONDBLCLK  7
#define CV_EVENT_RBUTTONDBLCLK  8
#define CV_EVENT_MBUTTONDBLCLK  9

#define CV_EVENT_FLAG_LBUTTON   1
#define CV_EVENT_FLAG_RBUTTON   2
#define CV_EVENT_FLAG_MBUTTON   4
#define CV_EVENT_FLAG_CTRLKEY   8
#define CV_EVENT_FLAG_SHIFTKEY  16
#define CV_EVENT_FLAG_ALTKEY    32

CV_EXTERN_C_FUNCPTR( void (*CvMouseCallback )(int event, int x, int y, int flags) );

HIGHGUI_API void cvSetMouseCallback( const char* window_name, CvMouseCallback on_mouse );
```

window_name
: Name of the window.

on_mouse
: Pointer to the function to be called every time mouse event occurs in the specified window. This function should be prototyped as

  ```
  void Foo(int event, int x, int y, int flags);
  ```

  where *event* is one of *CV_EVENT_**, *x* and *y* are coordinates of mouse pointer in image coordinates (not window coordinates) and *flags* is a combination of *CV_EVENT_FLAG*.

The function cvSetMouseCallback [p 275] sets the callback function for mouse events occuting within the specified window. To see how it works, look at opencv/samples/c/ffilldemo.c demo

---

Image handling functions

---

cvLoadImage

Loads an image from file

```
IplImage* cvLoadImage( const char* filename, int iscolor CV_DEFAULT(1));
```

filename
: Name of file to be loaded.

iscolor
: If >0, the loaded image will always have 3 channels;
  if 0, the loaded image will always have 1 channel;
  if <0, the loaded image will be loaded as is (with number of channels depends on the file).

The function cvLoadImage [p 276] loads an image from the specified file and returns the pointer to the loaded image. Currently the following file formats are supported: Windows bitmaps - BMP, DIB; JPEG files - JPEG, JPG, JPE; Portable Network Graphics - PNG; Portable image format - PBM, PGM, PPM; Sun rasters - SR, RAS; TIFF files - TIFF, TIF.

If "filename" does not contain full path, the file is searched in the current directory and in directories specified by cvAddSearchPath [p 283]

---

cvSaveImage

Saves an image to the file

```
int cvSaveImage( const char* filename, const CvArr* image );
```

filename
: Name of the file.

image
: Image to be saved.

The function cvSaveImage [p 277] saves the image to the specified file.

---

cvShowImage

Shows the image in the specified window

```
void cvShowImage( const char* name, const CvArr* image );
```

name
: Name of the window to attach the image to.

image
: Image to be shown.

The function cvShowImage [p 277] shows the image in the specified window. If the window was created with CV_WINDOW_AUTOSIZE flag then the image will be shown with its original size otherwise the image will be scaled to fit the window.

---

cvConvertImage

Converts one image to another with optional vertical flip

```
void cvConvertImage( const CvArr* src, CvArr* dst, int flip CV_DEFAULT(0));
```

src
: Source image.

dst
: Destination image.

flip
: 1 - to flip image vertically,
0 - not to flip.

The function cvConvertImage [p 277] converts one image to another and flips the result vertically if required. This function does the same conversions as cvCvtColor [p ??] function, but do this automatically accordingly to formats of input and output images.

Video I/O functions

CvCapture

Structure for getting video from camera or AVI file

```
typedef struct CvCapture CvCapture;
```

The structure CvCapture [p 278] does not have public interface and is used only as a parameter for video capture functions.

---

cvCaptureFromAVI

Allocates CvCapture structure binds it to the specified AVI file

```
CvCapture* cvCaptureFromAVI( const char* filename );
```

filename
: Name of the AVI file.

The function cvCaptureFromAVI [p 278] allocates and initialized the CvCapture structure for reading the video stream from the specified AVI file.

After the allocated structure is not used any more it should be released by cvReleaseCapture [p 279] function.

---

cvCaptureFromCAM

Allocates CvCapture structure and binds it to the video camera

```
CvCapture* cvCaptureFromCAM( int index );
```

index
: Index of the camera to be used. If there is only one camera or it does not matter what camera to use, -1 may be passed.

The function cvCaptureFromCAM [p 278] allocates and initialized the CvCapture structure for reading a video stream from the camera. Currently two camera interfaces can be used: Video for Windows (VFW) and Matrox Imaging Library (MIL). To connect to VFW camera the parameter "index" should be in range 0-10, to connect to MIL camera the parameter "index" should be in range 100-115. If -1 is passed then the function searches for VFW camera first and then for MIL camera.

After the allocated CvCapture structure is not used any more it should be released by cvReleaseCapture [p 279] function.

---

cvReleaseCapture

Releases the CvCapture structure

```
void cvReleaseCapture( CvCapture** capture );
```

capture
: Address of the pointer to CvCapture structure to be released.

The function cvReleaseCapture [p 279] releases the CvCapture structure allocated by cvCaptureFromAVI [p 278] or cvCaptureFromCAM [p 278] .

---

cvGrabFrame

Grabs frame from camera or AVI

```
int cvGrabFrame( CvCapture* capture );
```

capture
: CvCapture representing camera or AVI file.

The function cvGrabFrame [p 279] grabs the frame from camera or AVI. The grabbed frame is stored internally. The purpose of this function is to grab frame fast what is important for syncronization in case of reading from several cameras simultaneously. The grabbed frames are not exposed because they may be stored in compressed format (as defined by camera/driver). To get access to the grabbed frame cvGrabFrame [p ??] should be followed by cvRetrieveFrame [p 279] .

---

cvRetrieveFrame

Gets the image grabbed with cvGrabFrame

```
IplImage* cvRetrieveFrame( CvCapture* capture );
```

capture
: CvCapture representing camera or AVI file.

The function cvRetrieveFrame [p 279] returns the pointer to the image grabbed with cvGrabFrame [p 279] function. The returned image should not be released by the user.

---

cvQueryFrame

Grabs and returns a frame from camera or AVI

```
IplImage* cvQueryFrame( CvCapture* capture );
```

capture
: CvCapture representing camera or AVI file.

The function cvQueryFrame [p 280] grabs a frame from camera or AVI and returns the pointer to grabbed image. Actually this function just successively calls cvGrabFrame [p ??] and cvRetrieveFrame [p 279] . The returned image should not be released by the user.

---

cvGetCaptureProperty

Gets camera/AVI properties

```
double cvGetCaptureProperty( CvCapture* capture, int property_id );
```

capture
: CvCapture representing camera or AVI file.

property_id
: property identifier. Can be one of the following:
CV_CAP_PROP_POS_MSEC - film current position in milliseconds or video capture timestamp
CV_CAP_PROP_POS_FRAMES - 0-based index of the frame to be decoded/captured next
CV_CAP_PROP_POS_AVI_RATIO - relative position of AVI file (0 - start of the film, 1 - end of the film)
CV_CAP_PROP_FRAME_WIDTH - width of frames in the video stream
CV_CAP_PROP_FRAME_HEIGHT - height of frames in the video stream
CV_CAP_PROP_FPS - frame rate
CV_CAP_PROP_FOURCC - 4-character code of codec. CV_CAP_PROP_FRAME_COUNT - number of frames in AVI file.

The function cvGetCaptureProperty [p 280] retrieves the specified property of camera or AVI.

---

cvSetCaptureProperty

Sets camera/AVI properties

```
int cvSetCaptureProperty( CvCapture* capture, int property_id, double value );
```

capture
: CvCapture representing camera or AVI file.

property_id
: property identifier. Can be one of the following:
CV_CAP_PROP_POS_MSEC - (only for AVI)
CV_CAP_PROP_POS_MSEC - set position (only for AVIs)
CV_CAP_PROP_POS_FRAMES - set position (only for AVIs)
CV_CAP_PROP_POS_AVI_RATIO - set position (only for AVIs)
CV_CAP_PROP_FRAME_WIDTH - width of frames in the video stream
CV_CAP_PROP_FRAME_HEIGHT - height of frames in the video stream
CV_CAP_PROP_FPS - frame rate
CV_CAP_PROP_FOURCC - 4-character code of codec.

value
: value of the property.

The function cvSetCaptureProperty [p 281] sets the specified property of camera or AVI. Currently the function works only for setting some AVI properties: CV_CAP_PROP_POS_MSEC, CV_CAP_PROP_POS_FRAMES, CV_CAP_PROP_POS_AVI_RATIO

---

cvCreateAVIWriter

Creates AVI writer

```
typedef struct CvAVIWriter CvAVIWriter;
CvAVIWriter* cvCreateAVIWriter( const char* filename, int fourcc, double fps, CvSize frameSize )
```

filename
: Name of AVI file to be written to. If file does not exist it is created.

fourcc
: 4-character code of codec used to compress the frames. For example, CV_FOURCC('P','I','M','1') is MPEG-1 codec, CV_FOURCC('M','J','P','G') is motion-jpeg codec etc. Under Win32 it is possible to pass -1 in order to choose compression method and additional compression parameters from dialog.

fps
: Framerate of the created video stream.

frameSize
: Size of the frames of AVI file.

The function cvCreateAVIWriter [p 281] allocates and initializes the hidden structure CvAVIWriter that is used for writing AVI files frame by frame.

NOTE: Writing to AVIs works under Win32 only

---

cvReleaseAVIWriter

Releases AVI writer

```
void cvReleaseAVIWriter( CvAVIWriter** writer );
```

writer
: address of pointer to the released CvAVIWriter structure.

The function cvReleaseAVIWriter [p 282] closes the AVI file being written and deallocates the memory used by CvAVIWriter structure.

---

cvWriteToAVI

Writes a frame to AVI file

```
int cvWriteToAVI( CvAVIWriter* writer, const IplImage* image );
```

writer
: Pointer to CvAVIWriter structure.

image
: Frame to be written/appended to AVI file

The function cvWriteToAVI [p 282] writes/appends one frame to AVI file binded to "writer".

Support/system functions

---

cvInitSystem

Initializes HighGUI

```
void cvInitSystem( int argc, char** argv );
```

argc
: Number of command line arguments.

argv
: Array of command line arguments

The function cvInitSystem [p 282] initializes HighGUI. If it wasn't called explicitly by the user before the first window is created, it is called implicitly then with *argc*=0, *argv*=NULL. Under Win32 there is no need to call it explicitly. Under X Window the arguments are used for creating Application Shell that is a standard way to define a look of HighGUI windows and controls.

cvWaitKey

Waits for pressed key

```
int cvWaitKey(int delay CV_DEFAULT(0));
```

delay
: Delay in milliseconds.

The function cvWaitKey [p 283] waits for key event infinitely (delay<=0) or for "delay" milliseconds. Returns the code of pressed key or -1 if key was not pressed until the specified timeout has elapsed.

Note: This function is the only method in HighGUI to fetch and handle events so it needs to be called periodically for normal event processing.

---

cvAddSearchPath

Adds the specified path to the list of search paths;

```
/* add folder to the image search path (used by cvLoadImage) */
void cvAddSearchPath( const char* path );
```

path
: Path to add to the search list.

The function cvAddSearchPath [p 283] adds the specified folder to the search path list. The search path list is used by cvLoadImage [p 276] function.

Bibliography

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.

[Horn81] Berthold K.P. Horn and Brian G. Schunck. Determining Optical Flow. Artificial Intelligence, 17, pp. 185-203, 1981.

[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.

[Schiele00] 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] Z. Zhang. Parameter Estimation Techniques: A Tutorial with Application to Conic Fitting, Image and Vision Computing Journal, 1996.

[Zhang99] Z. Zhang. Flexible Camera Calibration By Viewing a Plane From Unknown Orientations. International Conference on Computer Vision (ICCV'99), Corfu, Greece, pages 666-673, September 1999.

[Zhang00] Z. Zhang. A Flexible New Technique for Camera Calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(11):1330-1334, 2000.