OpenCV User Guide

v2.2

December, 2010

Contents

Part I

C++ API User Guide

Chapter 1

cv::Mat. Operations with images.

1.1 Input/Output

Images

Load an image from a file:

```
Mat img = imread(filename);
```

If you read a jpg file, a 3 channel image is created by default. If you need a grayscale image, use:

```
Mat img = imread(filename, 0);
```

Save an image to a file:

```
Mat img = imwrite(filename);
```

XML/YAML

1.2 Basic operations with images

Accessing pixel intensity values

In order to get pixel intensity value, you have to know the type of an image and the number of channels. Here is an example for a single channel grey scale image (type 8UC1) and pixel coordinates x and y:

```
Scalar intensity = img.at<uchar>(x, y);
```

`intensity.val[0]` contains a value from 0 to 255. Now let us consider a 3 channel image with `bgr` color ordering (the default format returned by imread):

```
Vec3b intensity = img.at<Vec3b>(x, y);
uchar blue = intensity.val[0];
uchar green = intensity.val[1];
uchar red = intensity.val[2];
```

You can use the same method for floating-point images (for example, you can get such an image by running Sobel on a 3 channel image):

```
Vec3f intensity = img.at<Vec3f>(x, y);
float blue = intensity.val[0];
float green = intensity.val[1];
float red = intensity.val[2];
```

The same method can be used to change pixel intensities:

```
img.at<uchar>(x, y) = 128;
```

There are functions in OpenCV, especially from calib3d module, such as `projectPoints`, that take an array of 2D or 3D points in the form of `Mat`. Matrix should contain exactly one column, each row corresponds to a point, matrix type should be 32FC2 or 32FC3 correspondingly. Such a matrix can be easily constructed from std::vector:

```
vector<Point2f> points;
//... fill the array
Mat pointsMat = Mat(points);
```

One can access a point in this matrix using the same method `Mat::at`:

```
Point2f point = pointsMat.at<Point2f>(i, 0);
```

Memory management and reference counting

`Mat` is a structure that keeps matrix/image characteristics (rows and columns number, data type etc) and a pointer to data. So nothing prevents us from having several instances of `Mat` corresponding to the same data. A `Mat` keeps a reference count that tells if data has to be deallocated when a particular instance of `Mat` is destroyed. Here is an example of creating two matrices without copying data:

```
std::vector<Point3f> points;
// .. fill the array
Mat pointsMat = Mat(points).reshape(1);
```

As a result we get a 32FC1 matrix with 3 columns instead of 32FC3 matrix with 1 column. `pointsMat` uses data from `points` and will not deallocate the memory when destroyed. In this particular instance, however, developer has to make sure that lifetime of `points` is longer than of `pointsMat`. If we need to copy the data, this is done using, for example, `Mat::copyTo` or `Mat::clone`:

```
Mat img = imread("image.jpg");
Mat img1 = img.clone();
```

To the contrary with C API where an output image had to be created by developer, an empty output `Mat` can be supplied to each function. Each implementation calls `Mat::create` for a destination matrix. This method allocates data for a matrix if it is empty. If it is not empty and has the correct size and type, the method does nothing. If, however, size or type are different from input arguments, the data is deallocated (and lost) and a new data is allocated. For example:

```
Mat img = imread("image.jpg");
Mat sobelx;
Sobel(img, sobelx, CV_32F, 1, 0);
```

Primitive operations

There is a number of convenient operators defined on a matrix. For example, here is how we can make a black image from an existing greyscale image `img`:

```
img = Scalar(0);
```

Selecting a region of interest:

```
Rect r(10, 10, 100, 100);
Mat smallImg = img(r);
```

A convertion from `Mat` to C API data structures:

```
Mat img = imread("image.jpg");
IplImage img1 = img;
CvMat m = img;
```

Note that there is no data copying here.

Conversion from color to grey scale:

```
Mat img = imread("image.jpg"); // loading a 8UC3 image
Mat grey;
cvtColor(img, grey, CV_BGR2GRAY);
```

Change image type from 8UC1 to 32FC1:

```
convertTo(src, dst, CV_32F);
```

Visualizing images

It is very useful to see indermediate results of your algorithm during development process. OpenCV provides a convenient way of visualizing images. A 8U image can be shown using:

```
Mat img = imread("image.jpg");

namedWindow("image", CV_WINDOW_AUTOSIZE);
imshow("image", img);
waitKey();
```

A call to waitKey() starts a message passing cycle that waits for a key stroke in the "image" window. A 32F image needs to be converted to 8U type. For example:

```
Mat img = imread("image.jpg");
Mat grey;
cvtColor(img, grey, CV_BGR2GREY);

Mat sobelx;
Sobel(grey, sobelx, CV_32F, 1, 0);

double minVal, maxVal;
minMaxLoc(sobelx, &minVal, &maxVal); //find minimum and maximum intensities
Mat draw;
sobelx.convertTo(draw, CV_8U, 255.0/(maxVal - minVal), -minVal);

namedWindow("image", CV_WINDOW_AUTOSIZE);
imshow("image", draw);
waitKey();
```

Chapter 2

Features2d.

2.1 Detectors

2.2 Descriptors

2.3 Matching keypoints

The code

We will start with a short sample opencv/samples/cpp/matcher_simple.cpp:

```
Mat img1 = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
Mat img2 = imread(argv[2], CV_LOAD_IMAGE_GRAYSCALE);
if(img1.empty() || img2.empty())
{
        printf("Can't read one of the images\n");
        return -1;
}

// detecting keypoints
SurfFeatureDetector detector(400);
vector<KeyPoint> keypoints1, keypoints2;
detector.detect(img1, keypoints1);
detector.detect(img2, keypoints2);

// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1, descriptors2;
extractor.compute(img1, keypoints1, descriptors1);
extractor.compute(img2, keypoints2, descriptors2);
```

```
// matching descriptors
BruteForceMatcher<L2<float> > matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);

// drawing the results
namedWindow("matches", 1);
Mat img_matches;
drawMatches(img1, keypoints1, img2, keypoints2, matches, img_matches);
imshow("matches", img_matches);
waitKey(0);
```

The code explained

Let us break the code down.

```
Mat img1 = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
Mat img2 = imread(argv[2], CV_LOAD_IMAGE_GRAYSCALE);
if(img1.empty() || img2.empty())
{
        printf("Can't read one of the images\n");
        return -1;
}
```

We load two images and check if they are loaded correctly.

```
// detecting keypoints
FastFeatureDetector detector(15);
vector<KeyPoint> keypoints1, keypoints2;
detector.detect(img1, keypoints1);
detector.detect(img2, keypoints2);
```

First, we create an instance of a keypoint detector. All detectors inherit the abstract FeatureDetector interface, but the constructors are algorithm-dependent. The first argument to each detector usually controls the balance between the amount of keypoints and their stability. The range of values is different for different detectors [1] so use defaults in case of doubt.

```
// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1, descriptors2;
extractor.compute(img1, keypoints1, descriptors1);
extractor.compute(img2, keypoints2, descriptors2);
```

[1] For instance, FAST threshold has the meaning of pixel intensity difference and usually varies in the region $[0, 40]$. SURF threshold is applied to a Hessian of an image and usually takes on values larger than 100.

We create an instance of descriptor extractor. The most of OpenCV descriptors inherit DescriptorExtractor abstract interface. Then we compute descriptors for each of the keypoints. The output `Mat` of the `DescriptorExtractor::compute` method contains a descriptor in a row i for each i-th keypoint. Note that the method can modify the keypoints vector by removing the keypoints such that a descriptor for them is not defined (usually these are the keypoints near image border). The method makes sure that the ouptut keypoints and descriptors are consistent with each other (so that the number of keypoints is equal to the descriptors row count).

```
// matching descriptors
BruteForceMatcher<L2<float> > matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
```

Now that we have descriptors for both images, we can match them. First, we create a matcher that for each descriptor from image 2 does exhaustive search for the nearest descriptor in image 1 using Eucledian metric. Manhattan distance is also implemented as well as a Hamming distance for Brief descriptor. The output vector `matches` contains pairs of corresponding points indices.

```
// drawing the results
namedWindow("matches", 1);
Mat img_matches;
drawMatches(img1, keypoints1, img2, keypoints2, matches, img_matches);
imshow("matches", img_matches);
waitKey(0);
```

The final part of the sample is about visualizing the matching results.