Module 1.2.1

OpenCV Basics

Introduction to the Mat Class (C++)

Satya Mallick, Ph.D.
LearnOpenCV.com

Table of Contents

Introduction to the Mat Class

OpenCV was first released in 2001. At that time it had a C API which was extremely cumbersome to use.

OpenCV 2.0 was launched with a C++ interface which made the life of programmers easier. The C++ API is a substantial improvement over the C API with many nice features including automatic memory management.

The Mat class, introduced in OpenCV 2.0, has two very important purposes

1. It serves as the basic image container.
2. It also serves as the Matrix.

Mat as an Image container

OpenCV Mat class contains two parts

1. **Header** : The matrix header which includes details such as the size of the matrix, medium of storage, the address of storage, etc
2. **Data** : The second part is a pointer to the address in memory that stores the pixel values of the image.

Because the Mat class wraps a pointer to the actual pixel data, explicit copying of image data is prevented, leading to faster execution time. When a copy of actual pixel data matrix is needed, the Mat class provides easy methods to cloning as well.

Let us review some of the methods of the Mat class when used as an image container.

Read an image

You can read in an image from a stored file by using the function **imread**

```
// Read image in BGR format
Mat A = imread("image.jpg", IMREAD_COLOR);
// Read image in grayscale
Mat A = imread("image.jpg", IMREAD_GRAYSCALE);
```

Note: When using the C++ interface, data allocated in the Mat class is managed by OpenCV. You do not need to release this memory explicitly except in some rare circumstances.

Assignment operation

In an assignment operation, the data part of a matrix is not copied. In the examples below B and C are pointing to the same data as A, but they have different headers.

```
// Pointer of B points to the same data matrix as A
Mat B(A);
// C is assigned the same data matrix as A but a different header
Mat C = A;
```

Copy part of an image

In computer vision applications, many times we work with only part of an image and not the entire image. For example, in an application, we may first detect a face in an image. After the face is detected, we are not interested in the entire image. We want to work in the coordinate system of the rectangle around the face.

The copy operations in OpenCV provides a very efficient way of doing it. In the examples below, no data is copied; only the headers are changed. The image B is pointing to the same data as A, but to any OpenCV algorithm, B appears like a cropped version of A!

```
// Copy a Rectangle of width 50 and height 50 from point (15,15)
Mat B(A, Rect(15, 15, 50, 50));
/* Specific rows and columns can also be
   selected using the  Range function */
// Copies from rows 2 to 4 and columns 4 to 6
Mat B = A(Range(2, 4), Range(4,6));
```

Clone an image

If you want to copy the header and the data matrix you can use the **clone()** functions.

```
// The clone method is used to copy data and header.
```

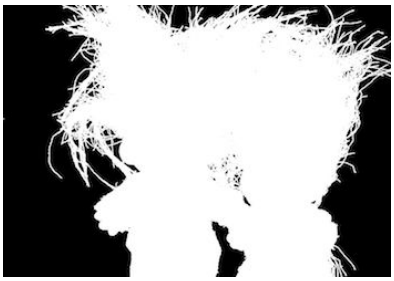
```
B = A.clone();
```

Clone an image with a mask

To clone an image with a mask, use **copyTo**

```
// The copyTo method allows you to clone with a binary mask.
A.copyTo(B,mask);
```

In the example below, the original image A is cloned with the binary mask in the center to produce output B. Only the pixels with value > 0 are copied over to B.

		
Original Image (A)	Mask	Clone (B) using copyTo

Find number of rows, columns, and channels

To find the number of rows, columns and channels of a Mat object, there are the following functions/attributes:

```
// Find rows and columns of a Mat object
```

```
Mat M;
// To find the number of rows
M.rows;
// To find the number of columns
M.cols;
// To find the number of channels
M.channels()
```

Alternatively, the **size()** method returns a Size object of type **Size(cols, rows)**.

Note: The number of columns equals width and the number of rows equals the height. So when specifying Size using height and width, we have to use **Size(height, width)** which is slightly counter-intuitive.

```
// Find height and width using Size
```

```
Mat M;
Size sz = M.size();
// number of rows
sz.height;
// number of columns
sz.width;
```

Mat as a Matrix

The Mat class also serves as the default Matrix class in OpenCV. Every Mat object has a datatype associated with it. The datatypes are specified by constants of the following form

CV_<bit-depth>{U|S|F}C<number_of_channels>

Where, U, S and F stand for Unsigned, Signed and Floating Point respectively.

For example, when we read a color image into a Mat object, the datatype is CV_8UC3 which means each item in the matrix (i.e. a pixel) has 8 bits per channel (because pixel intensity can take 256 different values), unsigned (because pixel intensity values do not go below 0), and the 3, in the end stands for the three channels of the color image.

On the other hand, if you wanted to create a 2x2 matrix of floating point numbers, you would use the datatype to be CV_32FC1 which stands for 32-bit floating point data with a single channel.

Note: `CV_32FC1` and `CV_32F` point to the same datatype.

Create Mat object

A matrix can be created in several ways.

Using Constructor

Mat M(no. of rows, no. of cols, no. of channels, color)

```
// Create a 3x3 Matrix using the constructor
Mat M(3,3,CV_8UC3, Scalar(0,255,180));
```

Creates a 3 by 3 matrix having 9 elements. Each element has three components. If you cout << M; the output will look like this:

```
// Print a matrix using
// cout << M << endl;
 [0,255,180, 0,255,180, 0,255,180;
  0,255,180, 0,255,180, 0,255,180;
  0,255,180, 0,255,180, 0,255,180]
```

Using C/C++ arrays

A Mat object can also be created using C/C++ arrays.

The example below shows how to create multi-dimensional matrices. The first argument passed to the Mat constructor is the number of dimensions. A C/C++ array is first used to specify the size of each dimension. A pointer to the array is then passed to the Mat constructor. The Scalar::all(0) function initializes all the pixels / matrix elements with 0 value.

Example

```
// Create Mat using C/C++ arrays
```

```
// First define the size of the matrix
int size[]={4,4};

// Create a 2-dimensional matrix with the above size filled with 20.
Mat L(2, size, CV_8U, Scalar::all(20));

// It should be noted that here we have used a 2-dimensional matrix so that we can
print the output. You can also create higher dimensional matrices but then it
cannot be printed directly with a simple cout.

cout << L << endl;
```

```
// Output
```

```
[ 20,  20,  20,  20;
  20,  20,  20,  20;
  20,  20,  20,  20;
  20,  20,  20,  20]
```

Using create method

The create method of the Mat class can be used to create a matrix as well.

M.create(rows, cols, number of channels)

```
// Create a matrix using the create method
```

```
M.create(4, 4, CV_8UC(2));
cout << M << endl;
```

```
// Output
```

```
[  0,   0,   0,   0,   0,   0,   0,   0;
   0,   0,   0,   0,   0,   0,   0,   0;
   0,   0,   0,   0,   0,   0,   0,   0;
   0,   0,   0,   0,   0,   0,   0,   0]
```

Using this method we cannot initialize the matrix according to our choice of value.
This creates a 4 by 4 matrix with all values initialized to 0. Printing the matrix using cout would print the following output.

Ones, zeros and identity Matrices

Mat also has a number of handy functions that help create some basic matrices easily.

To create a Matrix with all data entries initialized to 1:
`Mat A = Mat::ones(no. of rows, no. cols, datatype and no. of channels)`

```
// Create a matrix with all elements initialized to 1
Mat M1 = Mat::ones(3, 3, CV_64F);
cout << M1 << endl;
// Output
[1,1,1;
 1,1,1;
 1,1,1]
```

To create a Matrix with all data entries initialized to 0:
`Mat A = Mat::zeros(no. of rows, no. cols, datatype and no. of channels)`

```
// Create a matrix with all elements initialized to 0
Mat M2 = Mat::zeros(3, 3, CV_64F);
cout << M2 << endl;
// Output
[0,0,0;
 0,0,0;
 0,0,0]
```

To create an identity matrix (an identity matrix has all the principal diagonal elements as 1 and the rest elements are 0)

```
Mat A = Mat::eye( no. rows, no. cols, datatype and no. of channels)
```

```
// Create 3x3 identity matrix
Mat M3 = Mat::eye(3, 3, CV_64F);
cout << M3 << endl;
// Output
[1,0,0;
 0,1,0;
 0,0,1]
```

Initialize small matrices

For small matrices we can initialize individual elements using a comma separated list.

```
// Small matrix initialization
Mat C = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, 6, 0, -3, 3);
cout << C << endl;
// Output
[ 0,-1, 0;
 -1, 5, 6;
  0, -3, 3]
```

References and Further reading

http://docs.opencv.org/3.0-beta/doc/tutorials/core/mat_the_basic_image_container/mat_the_basic_image_container.html

http://docs.opencv.org/3.0-beta/modules/core/doc/basic_structures.html

http://aishack.in/tutorials/opencvs-interface/

http://opencvexamples.blogspot.com/2013/09/creating-matrix-in-different-ways.html