OpenCV for Image Processing

Part-I

By,

R.Pratesh Kumar Reddy

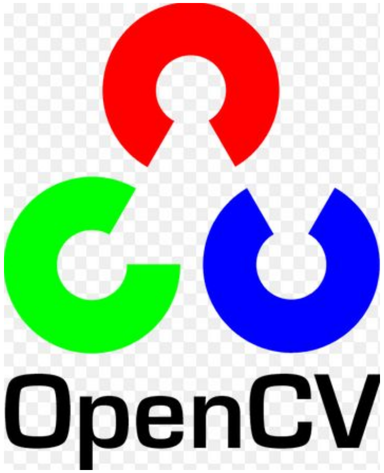
MT2014519

Contents of Presentation

- What is OpenCV?
- Uses of OpenCV
- Programming Language (why C++ api)
- Entire flow of the 2 presentations (strategy)
- Introduction to OpenCV 2.x
    - Modules
    - Include files
    - Basic Code Examples
- OpenCV data types (in detail)
    - Basic Data Types
    - Large Array Types
    - Array Operations
    - Utility Functions
    - Helper Objects
- GUI basics
- References/Furthur Reading

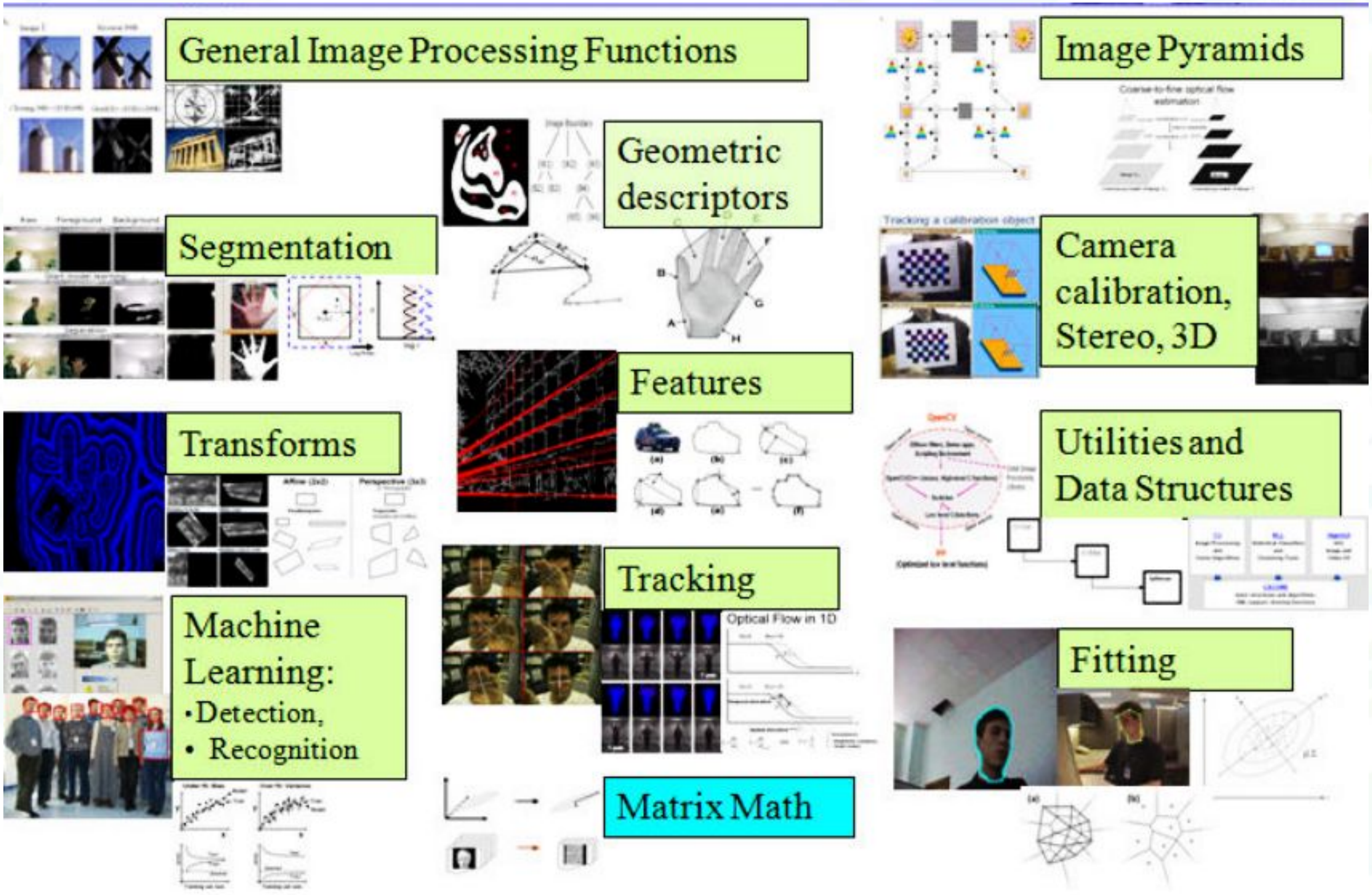
What is OpenCV ?

- OpenCV -- Open Source Computer Vision
- Open source library for real-time computer vision.
- Developed and launched initially by Intel Russia in 1999.
- Now taken over by OpenCV.org (NPO).
- Largest OpenCV library with 70 thousand people of user community.
- Has more than 500 optimised algorithms.
- Is Cross-platform and can run on Windows,Linux,Mac OS, iOS, Android, Blackberry 10, FreeBSD, OpenBSD.



OpenCV Use cases

- OpenCV Overview



General Image Processing Functions

Image Pyramids

Geometric descriptors

Segmentation

Camera calibration, Stereo, 3D

Features

Transforms

Utilities and Data Structures

Tracking

Machine Learning:
- Detection,
- Recognition

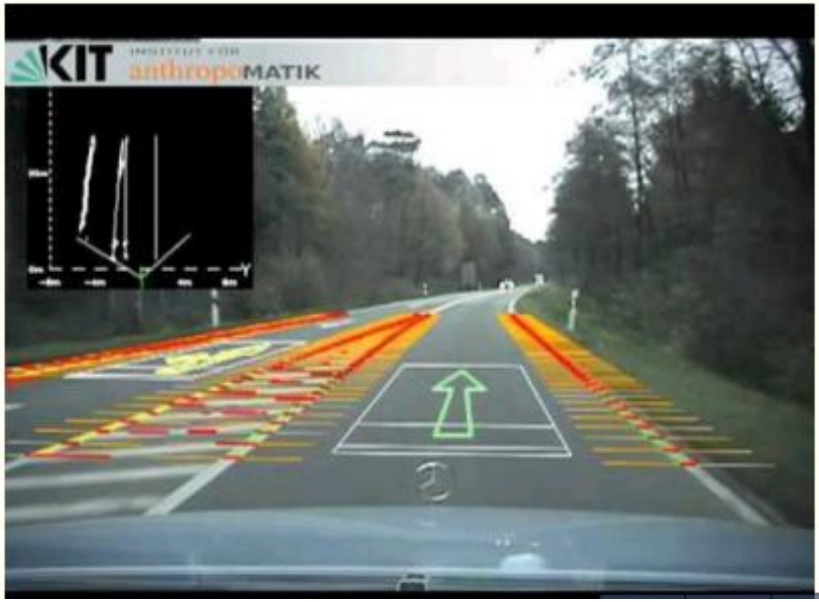
Fitting

Matrix Math

OpenCV Use cases

- OpenCV's application areas include:
  - 2D and 3D feature toolkits
  - Egomotion estimation
  - Facial recognition system
  - Gesture recognition
  - Human–computer interaction (HCI)
  - Mobile robotics
  - Motion understanding
  - Object identification
  - Segmentation and recognition
  - Stereopsis stereo vision: depth perception from 2 cameras
  - Structure from motion (SFM)
  - Motion tracking
  - Augmented reality
  - Machine Learning

OpenCV Use cases



Programming Language

- OpenCV 2.X is written in C++ and its primary interface is also C++.
- It mainly had C as its main interface in OpenCV 1.0 and still retains this interface.
- There are now full interfaces in Python, Java and MATLAB/OCTAVE (as of version 2.5).
- Wrappers in other languages such as C#, Perl, Ch and Ruby have been developed to encourage adoption by a wider audience.
- All of the new developments and algorithms in OpenCV are now developed in the C++ interface.
- It also has CUDA-based and OpenCL-based GPU interface which allow you to run the algorithms in GPU.
- Can take advantage of multiple core CPUs with its support for OpenMP.

- C++ is as near to application developers as is to embedded system developers.
- So that’s the reason the I have chosen to use C++ as main API.

Presentation Strategy

- There will be two presentations on this topic.
- First presentation will cover
  - Introduction
  - Basic codes in OpenCV
  - Data types and Objects (with their members)
  - GUI basics
- Second one will cover
  - Image filtering like blurring,morphological processes
  - Image Gradients, Edges and Corners detection.
  - Stretch,Shrink,Wrap and Rotate
  - Hough Transform
  - Histogram Equalization and Mapping
  - Image Segmentation
  - Frequency Analysis
- Will not be covering the Computer Vision Modules !!

Introduction to OpenCV 2.X - modules

- OpenCV is divided into Modules and each module deals with different fields of Computer Vision
- Every function in OpenCV is part of one or other module.
- The modules are
  - core: basic object types and their basic operations.
  - imgproc: image processing module contains basic transformations on images, including filters and similar convolutional operators.
  - highgui: user interface functions used to display images or take simple user input. A very light weight window UI toolkit.
  - video: To read and write video streams.
  - calib3d: To calibrate single cameras as well as stereo or multi-camera arrays.
  - features2d: For detecting, describing, and matching keypoint features.
  - objdetect: For detecting specific objects, such as faces or pedestrians. You can train the detectors to detect other objects as well.
  - ml: A wide array of machine learning algorithms to work with OpenCV datastructures.
  - Other : flann, GPU, photo, stitching, non-free, contrib, legacy, ocl
- The OpenCV documentation is also divided into same modules.

Introduction to OpenCV 2.X - include files

- Include files : The header files (.h) in OpenCV also follow the above modular structure.
- So if you want any function of a particular module you can just include that header file.

- The main header file is “ *.../include/opencv2/opencv.hpp*”
- #include "***opencv2/core/core_c.h***" : Old C data structures and arithmetic routines.
- #include "**opencv2/core/core.hpp**" : New C++ data structures and arithmetic routines.
- #include "**opencv2/imgproc/imgproc_c.h**" : Old C image processing functions.
- #include "**opencv2/imgproc/imgproc.hpp**" : New C++ image processing functions.
- "**opencv2/video/video.hpp**" : Video tracking and background segmentation routines.
- #include "**opencv2/features2d/features2d.hpp**" : Two-dimensional feature tracking support.
- #include "**opencv2/objdetect/objdetect.hpp**" :Cascade face detector; latent SVM; HoG; planar patch detector.
- #include "**opencv2/ml/ml.hpp"** : Machine learning: clustering, pattern recognition
- Some more
- Compile time : opencv.hpp v/s individual headers

`

Introduction to OpenCV 2.X - basic code

- ❖ **First program - Display a picture :**

```
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
using namespace std;
int main()
{
    cv::Mat im = cv::imread("image.jpg", CV_LOAD_IMAGE_COLOR);
    cv::namedWindow("Hello", cv::WINDOW_AUTOSIZE);
    cv::imshow("Hello", im);
    cout << "Press 'q' or ESC to quit..." << endl;
    int key;
    cv::waitKey(0)
    cv::destroyWindow(“Hello”);
    cout << "Got here" << endl;
    return 0;
}
```

`

Introduction to OpenCV 2.X - basic code

❖ Let’s break the code into chunks :

- header files
- using namespace
- main fucntion
- cv::Mat im = cv::imread("image.jpg", CV_LOAD_IMAGE_COLOR);
- namedWindow()
- imshow()
- waitkey()
- destroyWindow()

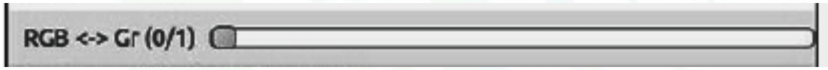
`

Introduction to OpenCV 2.X - basic code

- **Mat im = imread("image.jpg", CV_LOAD_IMAGE_COLOR);**
  - creates a variable im of type cv::Mat.
  - reads the image called image.jpg from the disk, and puts it into im through the function imread().
  - CV_LOAD_IMAGE_COLOR is a flag (a constant defined in the highgui.hpp header file) that tells imread() to load the image as a color image.
  - So, each pixel in im will have 3 channels of R,G,B.
- **cv::namedWindow("Hello",cv::WINDOW_AUTOSIZE);**
  cv::imshow("Hello", im);
  - creates a window called Hello (It is also the name displayed in the title bar of the window)
  - cv::WINDOW_AUTOSIZE flag such that window will resize itself as per image.
  - second one shows the image stored in im in the 'Hello' window
- **cv::waitKey(0)**
  - asks the program to stop and wait for a keystroke.
  - positive argument - program will wait for that milliseconds
  - 0 or negative argument - indefinite wait for a key stroke.
- **cv::destroyWindow()** will close the window and deallocate any associated memory usage

Introduction to OpenCV 2.X - basic code

- Let’s make some changes to above code:
- Add “***using namespace cv;***”
  - OpenCV functions live within a namespace called cv.
  - To get out of this bookkeeping chore, we can employ above snippet.
  - Compiler assumes that functions might belong to that namespace.
- Add “ ***(waitKey(1)) != 'q'*** ”
  - Checks every 1ms and waits for a particular keystroke to end the program.
  - Until then it has to be in a while(1) loop.
- Convert the image to grayscale using
  - ***cv::cvtColor()***
- Add slider (track bar) to the image
  - use ***createTrackbar()***



- So, our aim is to use this trackbar to change the image from grayscale to color and vice-versa. When we press ‘q’, the application should exit.

Introduction to OpenCV 2.X - basic code

- **Second Program - Change Color <-> Grayscale using trackbar**

```
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

// Global variables
int slider_max = 1;       // Maximum slider value const
int slider;               // Constantly updated slider value
Mat img;      // Original image
```

Introduction to OpenCV 2.X - basic code

```
// Callback function for trackbar event
void on_trackbar(int pos, void *)
{
        // Holds the image processed acording to value of slider
        Mat img_converted;
        // Convert color-spaces according to value of slider
        if(pos > 0) cvtColor(img, img_converted, CV_BGR2GRAY);
        else img_converted = img;
    imshow("Trackbar app", img_converted);
}
```

Introduction to OpenCV 2.X - basic code

```
int main()

{

        img = imread("image.jpg");

        namedWindow("Trackbar app");

        imshow("Trackbar app", img);


        slider = 0;                 // Initial value of slider

       // Create the trackbar

        createTrackbar("RGB <-> Grayscale", "Trackbar app", &slider, slider_max, on_trackbar);


        while(char(waitKey(1)) != 'q') {}          // Press 'q' to exit

        return 0;

}
```

Introduction to OpenCV 2.X - basic code

❖ **Third Program - Display Webcam or USBcam feed**

```
#include <opencv2/opencv.hpp>
using namespace cv;
using namespace std;

int main()
{
    // Create a VideoCapture object to read from video file
    // 0 is the ID of the built-in laptop camera, change if you want to use other camera
    VideoCapture cap(0);

//check if the file was opened properly
    if(!cap.isOpened())
    {
        cout << "Capture could not be opened succesfully" << endl;
        return -1;
    }

    namedWindow("Video");
```

Introduction to OpenCV 2.X - basic code

```
// Play the video in a loop till it ends

while(char(waitKey(1)) != 'q' && cap.isOpened())

{

        Mat frame;

        cap >> frame;

        // Check if the video is over

        if(frame.empty())

        {   cout << "Video over" << endl;

                break;            }

        imshow("Video", frame);

}

return 0; }
```

Introduction to OpenCV 2.X - basic code

- VideoCapture cap(0);
  - This creates a VideoCapture object that is linked to device number 0 (default device) on your computer.
- cap >> frame;
  - extracts a frame from the device to which the VideoCapture object cap is linked

What if you want to **read from a video file in our hard-disk** ??

Can use the same VideoCapture object.

But this time the arguments will contain the file path.

VideoCapture cap("video.mp4");

OpenCV supports almost every popular video container format.

All the other code remains same !!

Introduction to OpenCV 2.X - basic code

- Fourth Program - Writing Videos to disk from our WebCam

- For that we use the object of class “VideoWriter”.

- The object constructor of this class requires certain parameters for writing videos, viz.,

  - Output file name

  - Codec of the output file (like MPEG codec) - CV_FOURCC macro

  - Frames per second

  - Size of frames

Introduction to OpenCV 2.X - basic code

- The last two parameters can be obtained from the camera input frames. i.e
  - We are using VideoCapture object *cap* to get those camera frames
  - That object has member function "get"
  - Using that member fucntion "get", we can get various properties of a video (like the frame size, frame rate, brightness, contrast, exposure, etc.)
- CV_FOURCC macro needs four character codes of our codec. Note that to use a codec, you must have that codec installed on your computer

Introduction to OpenCV 2.X - basic code

- Fouth Program - write video to disk from the Webcam feed

```
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main()
{
    // 0 is the ID of the built-in laptop camera, change if you want to use other camera
    VideoCapture cap(0);
    //check if the file was opened properly
    if(!cap.isOpened())
    {
        cout << "Capture could not be opened succesfully" << endl;
        return -1;
    }


    // Get size of frames
    Size S = Size((int) cap.get(CV_CAP_PROP_FRAME_WIDTH), (int)
      cap.get(CV_CAP_PROP_FRAME_HEIGHT));
```

Introduction to OpenCV 2.X - basic code

```
// Make a video writer object and initialize it
VideoWriter put("output.mpg", CV_FOURCC('M','P','E','G'), 30, S);
if(!put.isOpened())
{
        cout << "File could not be created for writing. Check permissions" << endl;
        return -1;
}
namedWindow("Video");

// Play the video in a loop till it ends
while(char(waitKey(1)) != 'q' && cap.isOpened())
{
        Mat frame;
        cap >> frame;
        // Check if the video is over
        if(frame.empty())
        {
                cout << "Video over" << endl;
                break;
        }
        imshow("Video", frame);
        put << frame;
```

Introduction to OpenCV 2.X - basic code

```
}

      return 0;
}
```

❖ Till now we have played the video from both WebCam and Hard-disk.

Try this out ------> **Your own video player** !!

Writing a C++ program that loads the frames from video file in hard-disk and gives us the ability to control the video using a slider trackbar.

We have to use *cap.set (cv::CAP_PROP_POS_FRAMES, pos)* to set the current position of video frame to *pos* whenever the user changes the slider.

For every frame, we have to update our slider position. For that we have to use *cv::setTrackbarPos("SliderName", "WindowName", current_pos);*

To get this current position we'll use an already known function *int current_pos = (int)cap.get(cv::CAP_PROP_POS_FRAMES);*

Take help of many online OpenCV examples that show how to do the above task...

OpenCV 2.X datatypes

- One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly.
- For that we need to learn the basic building blocks of OpenCV so that we can create our own algorithms of them.
- Here comes the basic data types and all the objects we needed.
- The basic components in the library are enough to create a complete solution of your own to almost any computer vision problem.

Why do we need more data-types?

- The C/C++ has provided us with many data-types like int,float,char etc.,
- But these data-types are not enough when we have to handle image processing/Comp. Vision.
- For that OpenCV have used these basic data types and created many compound data-types.
- These are designed to make the representation and handling of computer vision concepts relatively easy and intuitive.
- You will also learn about templates and their aliases.
- Learn about possible type castings.
- OpenCV's basic data types are based on C++ STL

OpenCV 2.X datatypes

- Basic Data Types : Used to store vectors, points, small matrices, points, Scalars, Rectangles, Sizes
- Helper Objects : abstract concepts such as the garbage collecting pointer class, range objects used for slicing, and abstractions such as termination criteria.
- Large Array Types : to contain arrays or other assemblies of primitives (cv::Mat)
- Array Operations : Member functions of this cv::Mat objects
- Utility Functions : Extra functions

OpenCV 2.X datatypes - Basic Types

- Point classes
- cv::Scalar
- Size Classes
- cv::Rect
- cv::RotatedRect
- cv::Vec<>   (Fixed Vector Classes)
- cv::Matx<> (different from cv::Mat<>) (Fixed Matrix Classes)
- Complex Number Classes

MORE CONCENTRATION ON SPECIFIC BASIC DATA TYPES

Basic Types - Point Classes

- To store two-dimensional and three-dimensional points.

- Each of that point can be of any type : integer, floating-point, and so on.

- It has two templates (??) called Point_ & Point3_

- Out of these templates we can create aliases.

- So for e.g. we can create a 2D point that can hold two integers by

```
typedef Point_<int> Point2i;
```

- Now one can use the alias Point2i with any two integer values.

```
cv::Point2i p( x0, x1 );
```

Basic Types - Point Classes

- Many such aliases are pre-defined in OpenCV.
- The use of pre-defined aliases is that we don’t need to typedef every time we need to store two integer values.
- So, one can directly use Point2i and create an integer point object as above.
- The other pre-defined aliases of Point classes are

```
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;
typedef Point3_<int> Point3i;
typedef Point3_<float> Point3f;
typedef Point3_<double> Point3d;
```

Basic Types - Point Classes

- i is a 32-bit integer,
  f is a 32-bit floating-point number, and
  d is a 64-bit floating-point number

- We can define our own aliases that can hold unsigned characters or short integers.

  typedef Point_<short> Point2s; , typedef Point3_<uchar> Point2b;

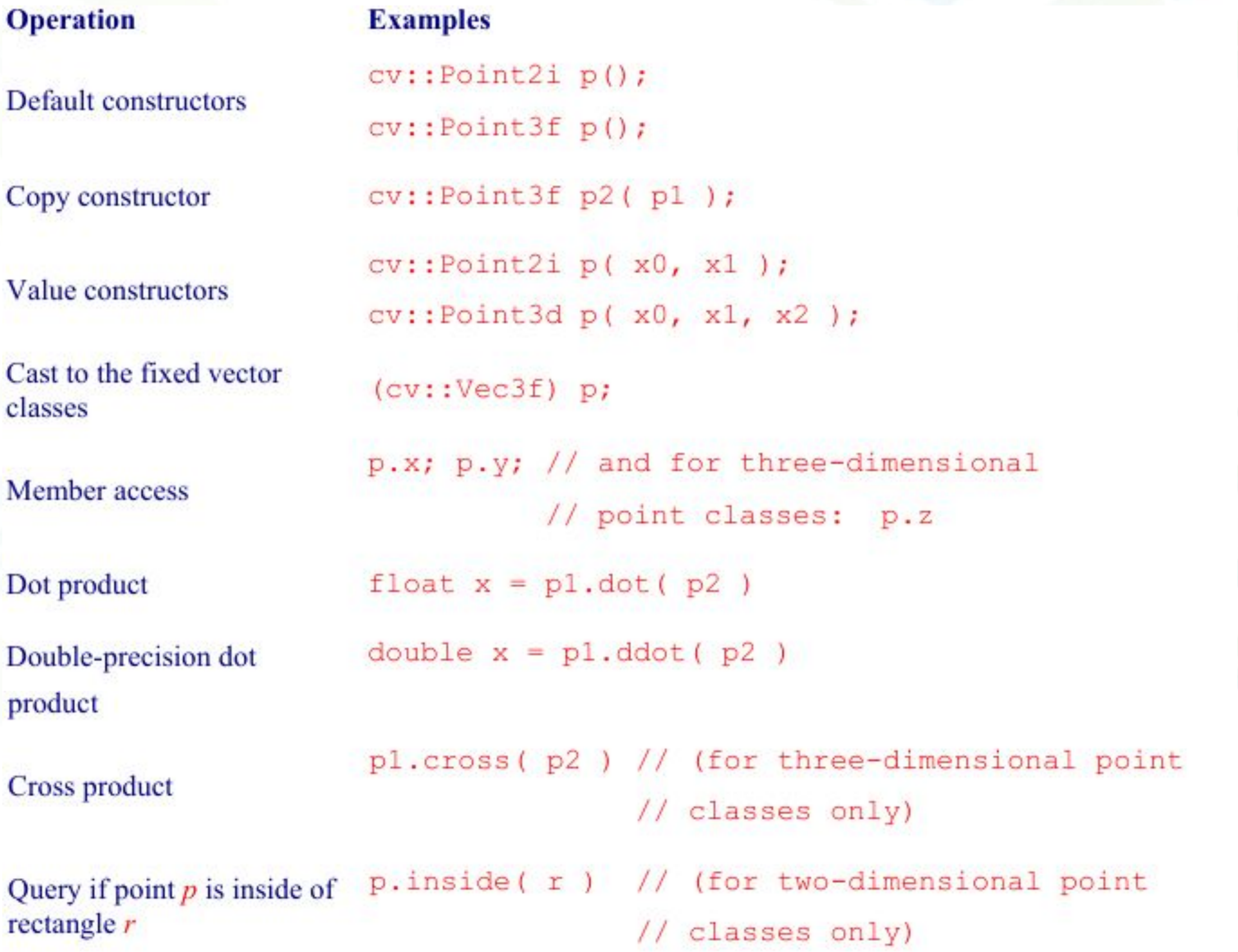
  b for unsigned character,
  s for short integer

Thereare lot of operations thatcan be performed on these point objects.
Using these operations, we can either access the point coordinates or can modify them.

```
For e.g.  cv::Point2i p( x0, x1 );   // create a point object p with coordinates x1 and x2
                p.x                              // to access the x coordinate
                p.y                              // to access the y coordinate
```

Basic Types - Point Classes

- Here is the (relatively short) list of functions natively supported by the point classes:



Operation	Examples

Default constructors	`cv::Point2i p();`
	`cv::Point3f p();`

Copy constructor	`cv::Point3f p2( p1 );`

Value constructors	`cv::Point2i p( x0, x1 );`
	`cv::Point3d p( x0, x1, x2 );`

Cast to the fixed vector classes	`(cv::Vec3f) p;`

Member access	`p.x; p.y; // and for three-dimensional`
	`// point classes:  p.z`

Dot product	`float x = p1.dot( p2 )`
Double-precision dot product	`double x = p1.ddot( p2 )`

Cross product	`p1.cross( p2 ) // (for three-dimensional point`
	`// classes only)`

Query if point *p* is inside of rectangle *r*	`p.inside( r )  // (for two-dimensional point`
	`// classes only)`

Basic Types - Point Classes

- ❖ The point classes natively support very less number of operations.
- ❖ But they are supported indirectly through implicit casting to the fixed vector classes

  So if you want more functions to be done on your point objects then first convert them to vector classes.

This conversion is generally called as “type casting”
So first use (cv::Vec3f) p; to convert it to vector.
As ‘p’ is now vector, we can use all the native functions of vectors on this ‘p’.
Infact, we can again convert this vector ‘p’ to other classes and use their native functions also.

Basic Types - cv::Scalar

- It’s a four-dimensional point class in which all of the members are double-precision floating-point numbers (d).
- Eventhough it’s called a point class, it is actually inherited from vector class.
- It was created using

```
typedef cv::Vec<double,4> cv::Scalar
```

- As a result, it inherits all of the vector algebra operations, member access functions and other properties from the fixed vector classes.

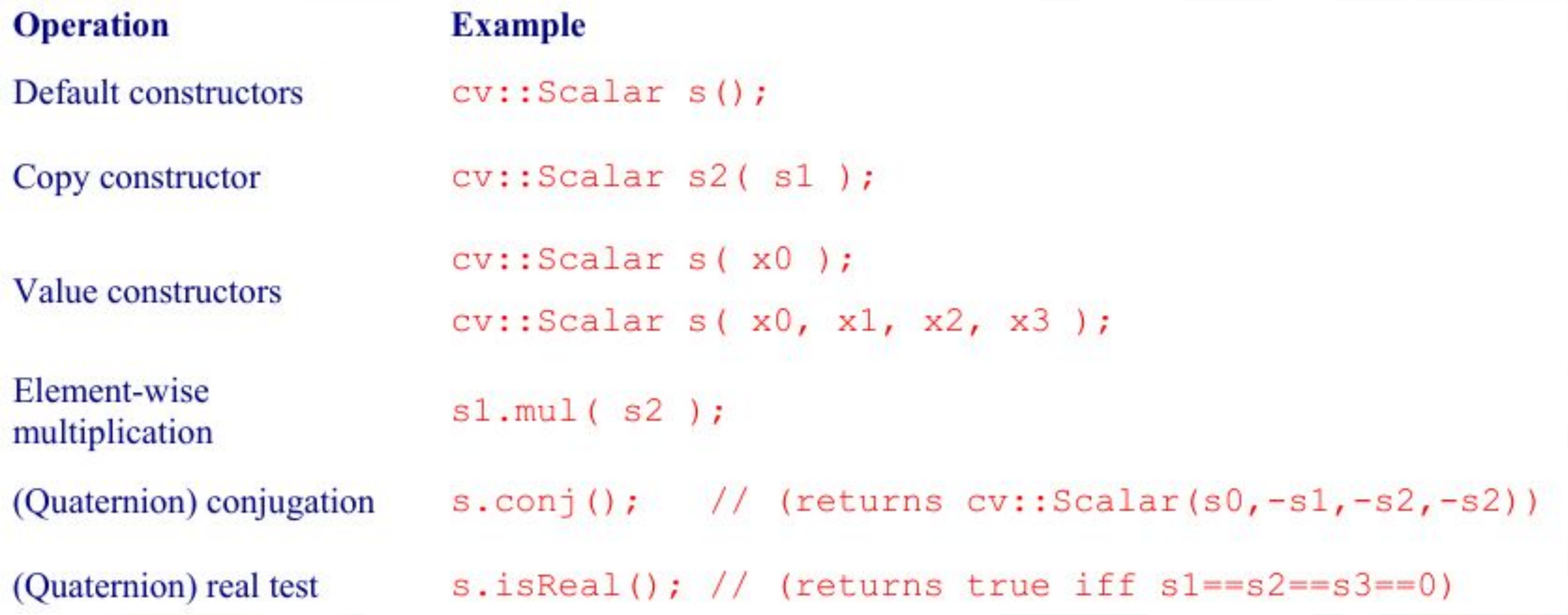
Four-dimensional double-precision vectors have some special uses in Comp. Vision applications.

So this cv::Scalar class has been defined seperatly.

This class has a few special member functions attached that are useful for various kinds of four component vectors in computer vision.

Basic Types - cv::Scalar

- Here is a short list of functions natively supported by the scalar class:



Operation	Example
Default constructors	`cv::Scalar s();`
Copy constructor	`cv::Scalar s2( s1 );`

Value constructors	`cv::Scalar s( x0 );`
	`cv::Scalar s( x0, x1, x2, x3 );`

Element-wise multiplication	`s1.mul( s2 );`
(Quaternion) conjugation	`s.conj();    // (returns cv::Scalar(s0,-s1,-s2,-s2))`
(Quaternion) real test	`s.isReal(); // (returns true iff s1==s2==s3==0)`

- You will notice that for cv::Scalar, the operation “cast to the fixed vector classes” does not appear. The reason (as noted earlier) is that, it inherits everthing from vector classes, so we don’t need to type cast.

Basic Types - Size Classes

- Similar to the point classes.
- The primary difference is that the point class' data members are named as x and y.
- While the corresponding data members in the size classes are named as width and height.

The three aliases for the size classes are

cv::Size, cv::Size2i, and cv::Size2f.

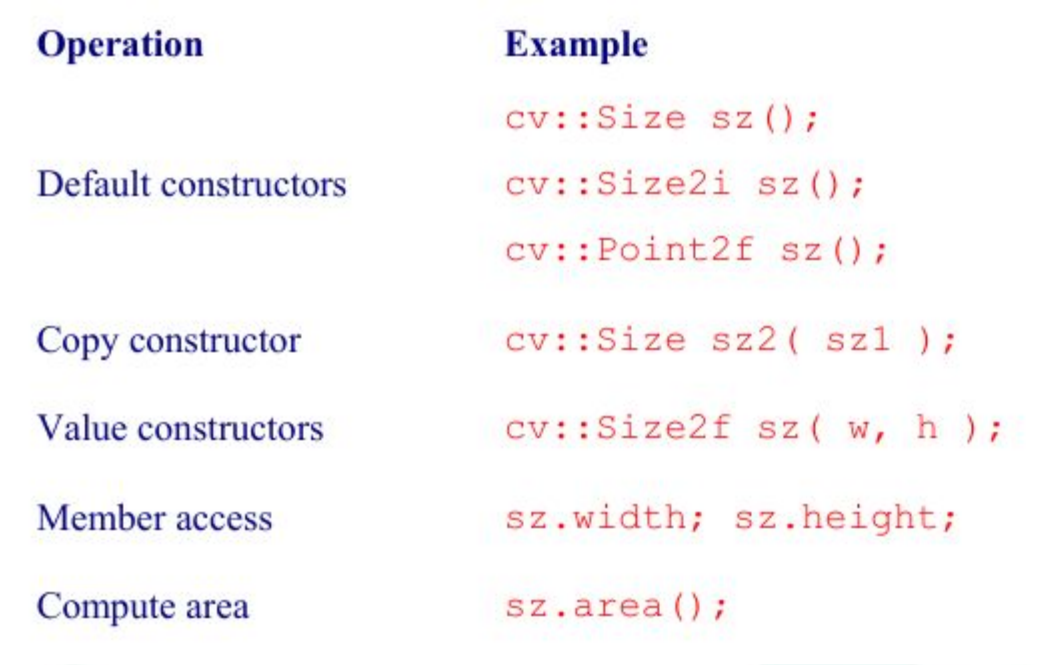
```
typedef Size_<int> Size2i;
typedef Size2i Size;
typedef Size_<float> Size2f;
```

The first two of these are equivalent and imply integer size
The last is for 32-bit floating-point sizes.

The size classes do not support casting to the fixed vector classes.
This means that the size classes have more restricted utility.
But the vice-versa is possible !!
Which means, the point classes and the fixed vector classes can be cast to the size classes without any problem.

Basic Types - Size Classes

- Operations supported directly by the size classes :



Operation	Example

Default constructors	`cv::Size sz();`
	`cv::Size2i sz();`
	`cv::Point2f sz();`

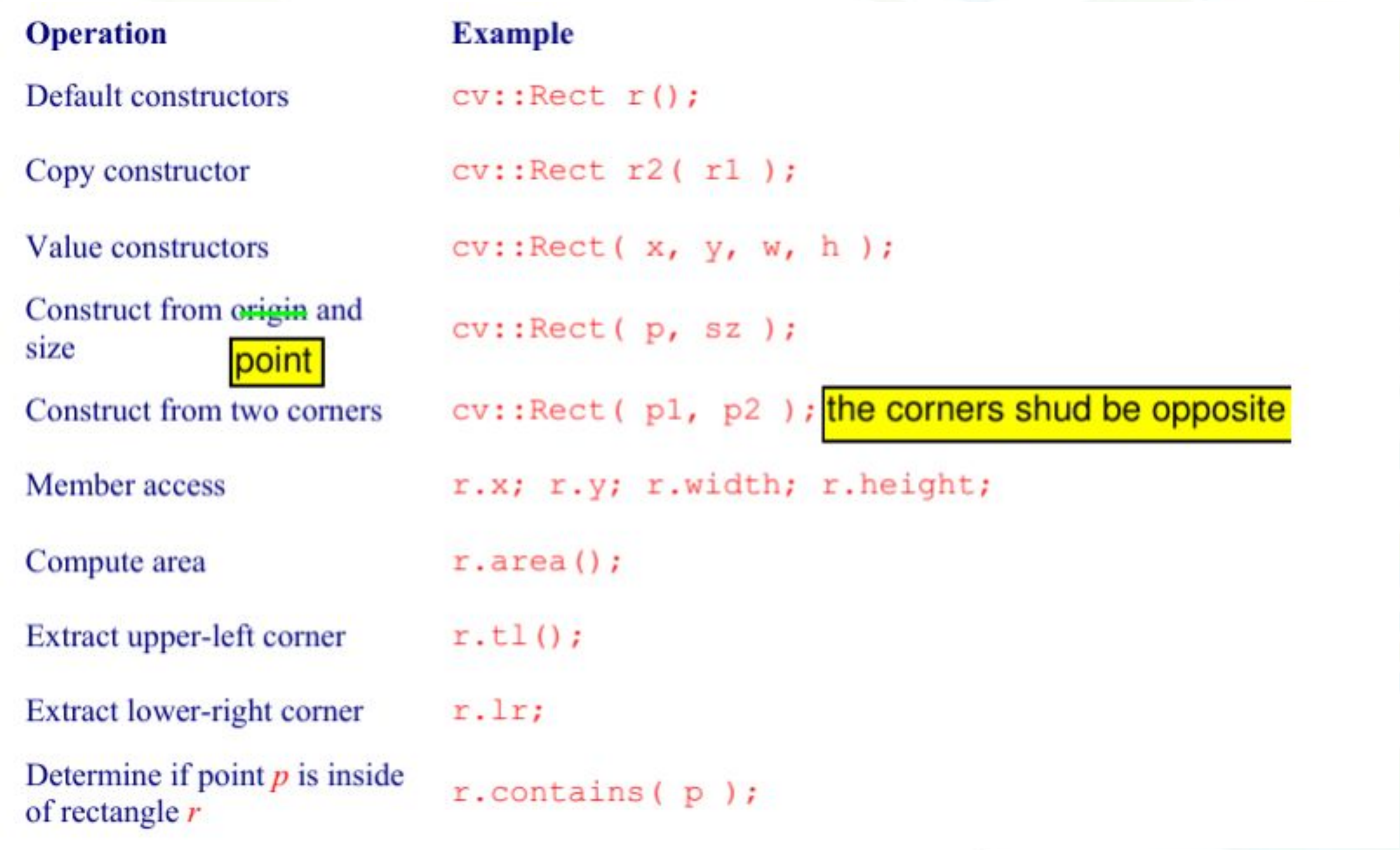
Copy constructor	`cv::Size sz2( sz1 );`
Value constructors	`cv::Size2f sz( w, h );`
Member access	`sz.width; sz.height;`
Compute area	`sz.area();`

Basic Types - cv::Rect

- To represent rectangles:
- The rectangle classes include
  - the members x and y of the point class (representing the upper-left corner of the rectangle)
  - the members width and height of the size class (representing the extent of the rectangle)
- The rectangle classes do not inherit operators from the point or size classes.

Basic Types - cv::Rect

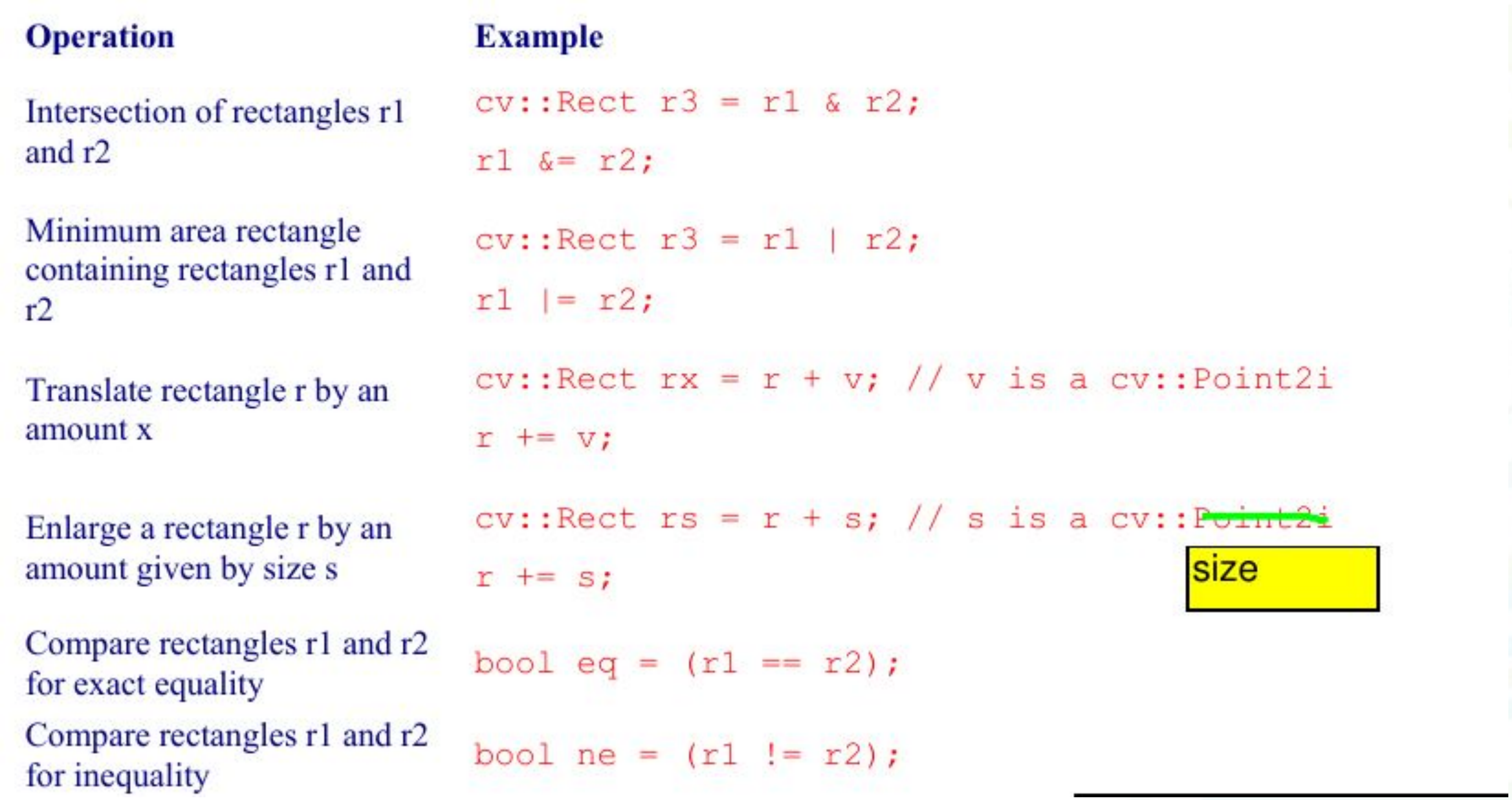
- Operations supported by class cv::Rect



Operation	Example
Default constructors	`cv::Rect r();`
Copy constructor	`cv::Rect r2( r1 );`
Value constructors	`cv::Rect( x, y, w, h );`
Construct from ~~origin~~ and size point	`cv::Rect( p, sz );`
Construct from two corners	`cv::Rect( p1, p2 );` the corners shud be opposite
Member access	`r.x; r.y; r.width; r.height;`
Compute area	`r.area();`
Extract upper-left corner	`r.tl();`
Extract lower-right corner	`r.lr;`
Determine if point *p* is inside of rectangle *r*	`r.contains( p );`

Basic Types - cv::Rect

- **Overloaded operators that take objects of type cv::Rect**



Operation	Example

Intersection of rectangles r1 and r2	`cv::Rect r3 = r1 & r2;`
	`r1 &= r2;`

Minimum area rectangle containing rectangles r1 and r2	`cv::Rect r3 = r1 | r2;`
	`r1 |= r2;`

Translate rectangle r by an amount x	`cv::Rect rx = r + v; // v is a cv::Point2i`
	`r += v;`

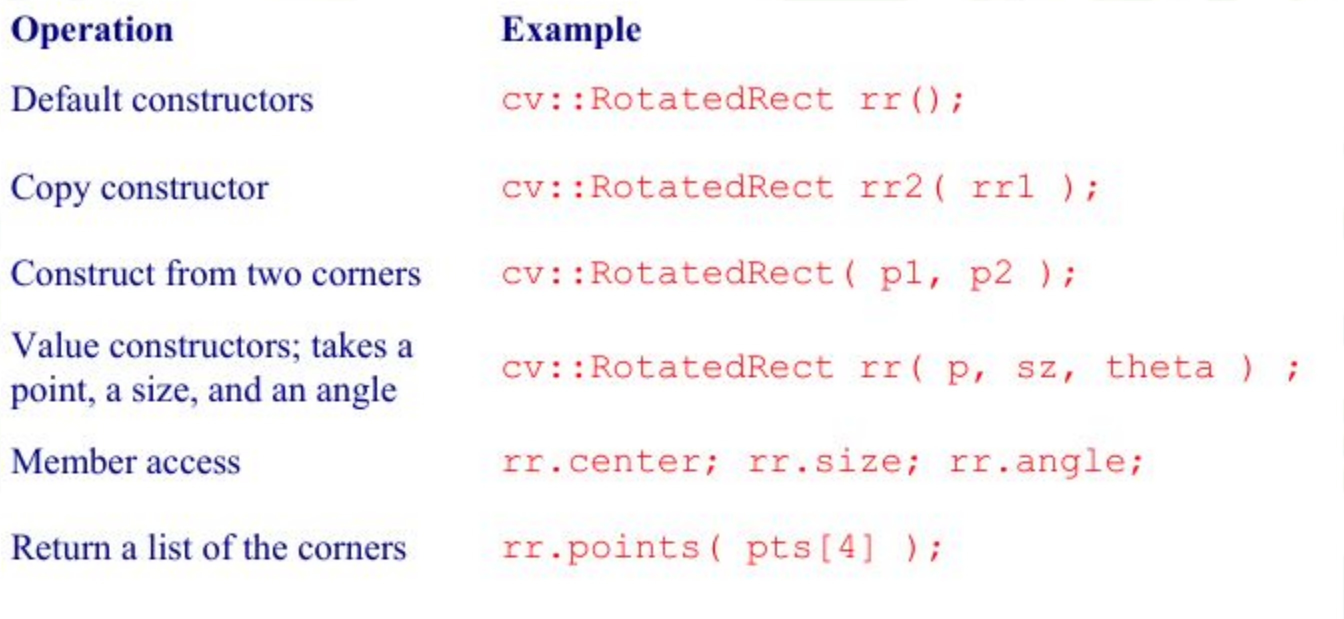
Enlarge a rectangle r by an amount given by size s	`cv::Rect rs = r + s; // s is a cv::`~~`Point2i`~~
	size
	`r += s;`

Compare rectangles r1 and r2 for exact equality	`bool eq = (r1 == r2);`
Compare rectangles r1 and r2 for inequality	`bool ne = (r1 != r2);`

Basic Types - cv::RotatedRect

- It holds cv::Point2f called center,
  cv::Size2f called size, and
  one additional float called angle, representing the rotation of the rectangle around center

Operations supported directly by class cv::RotatedRect

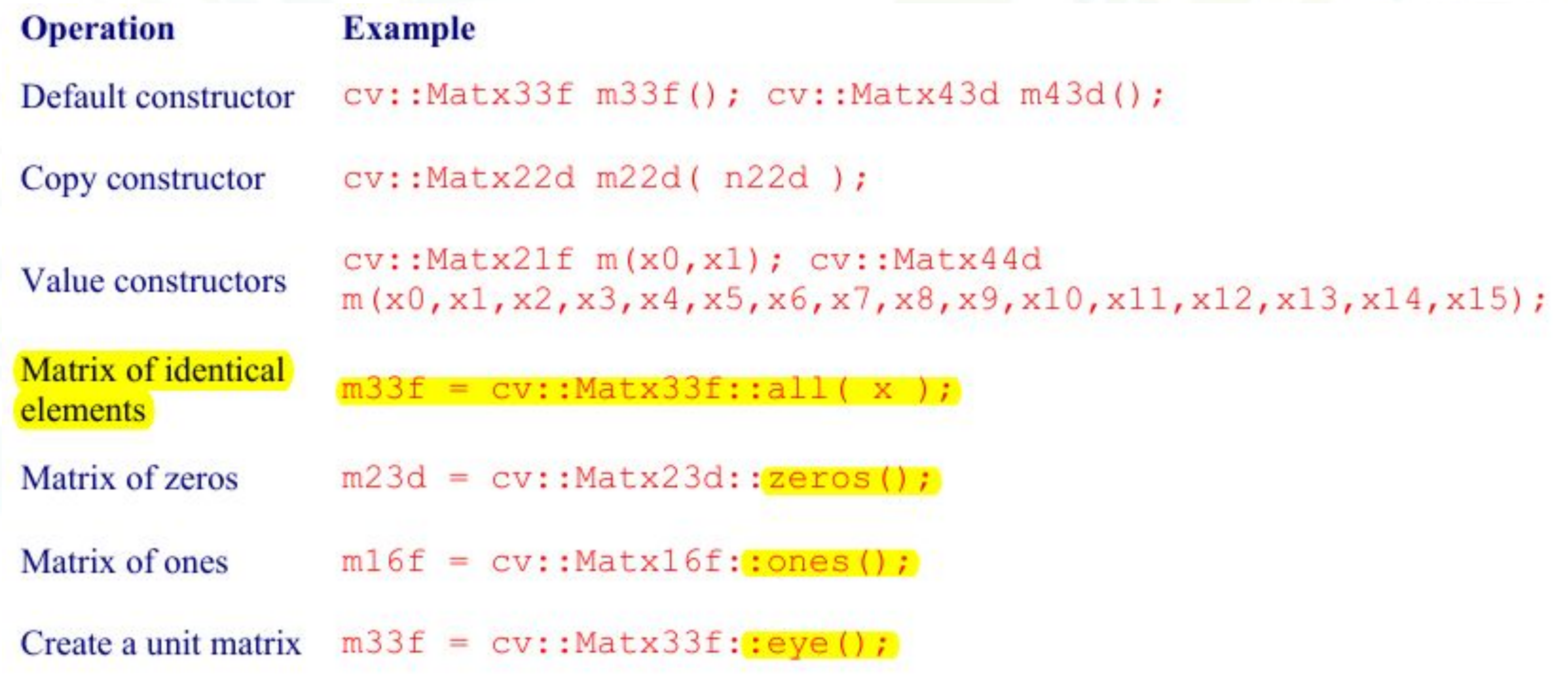


Operation	Example
Default constructors	`cv::RotatedRect rr();`
Copy constructor	`cv::RotatedRect rr2( rr1 );`
Construct from two corners	`cv::RotatedRect( p1, p2 );`
Value constructors; takes a point, a size, and an angle	`cv::RotatedRect rr( p, sz, theta ) ;`
Member access	`rr.center; rr.size; rr.angle;`
Return a list of the corners	`rr.points( pts[4] );`

Basic Types - Fixed Matrix Classes

- ❖ **Fixed matrix classes are for matrices whose dimensions are known at compile time (hence "fixed").**
- ❖ **Quick to allocate and clean up.**

The template is called cv::Matx<>
The basic pre-defined aliases are cv::Matx{1,2,...}{1,2,...}{f,d}
Numbers can be any number from one to six (except five)
Create your own using " typedef cv::Matx<5,5,float> "



Operation	Example
Default constructor	`cv::Matx33f m33f(); cv::Matx43d m43d();`
Copy constructor	`cv::Matx22d m22d( n22d );`
Value constructors	`cv::Matx21f m(x0,x1); cv::Matx44d m(x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15);`
Matrix of identical elements	`m33f = cv::Matx33f::all( x );`
Matrix of zeros	`m23d = cv::Matx23d::zeros();`
Matrix of ones	`m16f = cv::Matx16f::ones();`
Create a unit matrix	`m33f = cv::Matx33f::eye();`

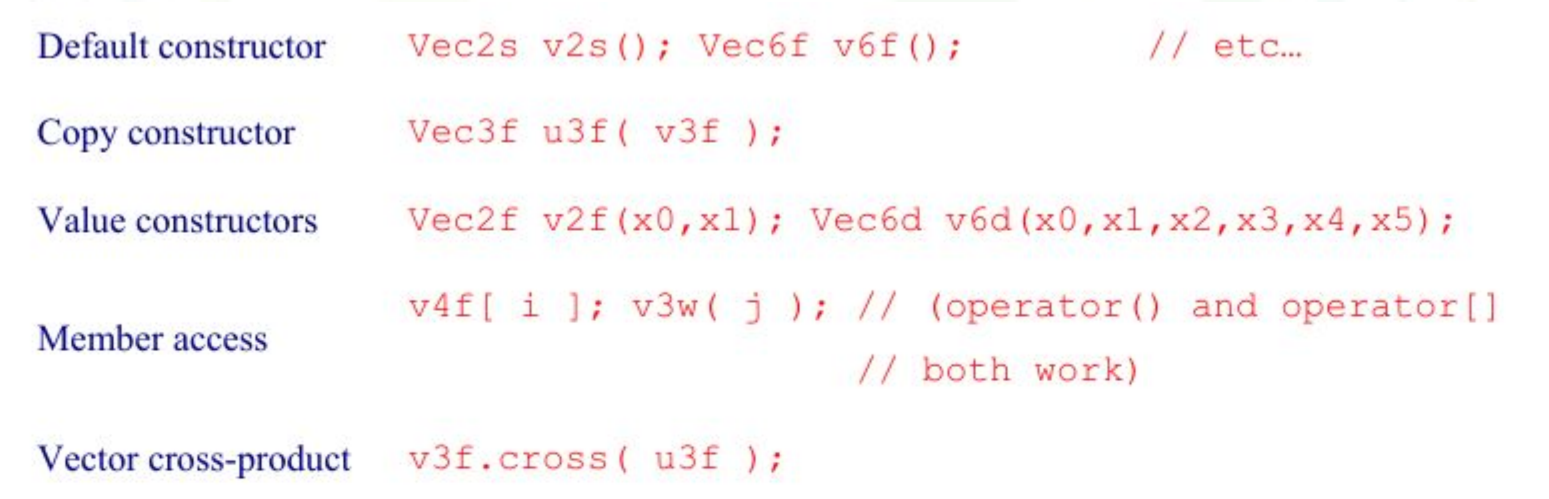
- ❖ Still, very large number of operations are there on this fixed matrix classes : you can get from ref.man

Basic Types - Fixed Vector Classes

- Derived from the fixed matrix classes.
- The fixed vector template cv::Vec<> is a cv::Matx<> whose number of columns is one.

readily available aliases are cv::Vec{2,3,4,6}{b,s,w,i,f,d}
the new addition, w indicates an unsigned short integer

They also inherit all the methods from ‘Fixed Matrix Class’



Default constructor	`Vec2s v2s(); Vec6f v6f();           // etc...`
Copy constructor	`Vec3f u3f( v3f );`
Value constructors	`Vec2f v2f(x0,x1); Vec6d v6d(x0,x1,x2,x3,x4,x5);`

Member access	`v4f[ i ]; v3w( j ); // (operator() and operator[]`
	`                    // both work)`

Vector cross-product	`v3f.cross( u3f );`

Large Array Types - cv::Mat

- cv::Mat is considered as the heart of the entire C++ implementation of the OpenCV library.
- Majority of functions in the OpenCV library
  - are members of the cv::Mat class
  - or take a cv::Mat as an argument
  - or return cv::Mat as a return value.

cv::Mat class is used to represent dense arrays of any number of dimensions.
Dimensions vs Channels -- Take color image as example

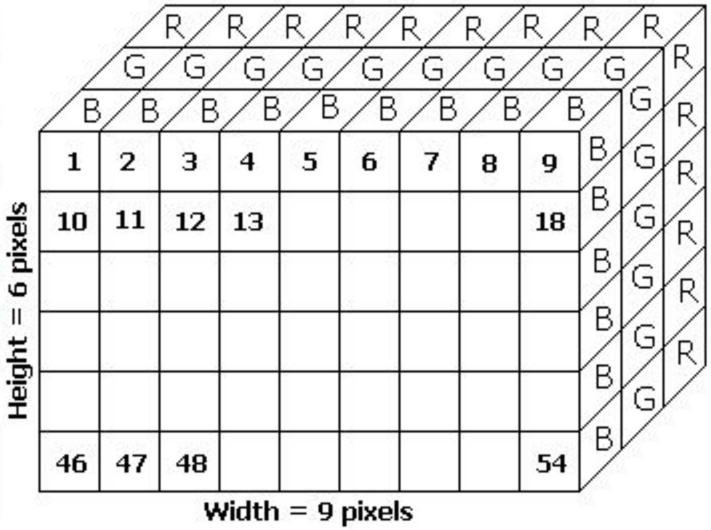
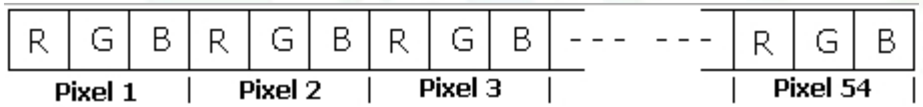
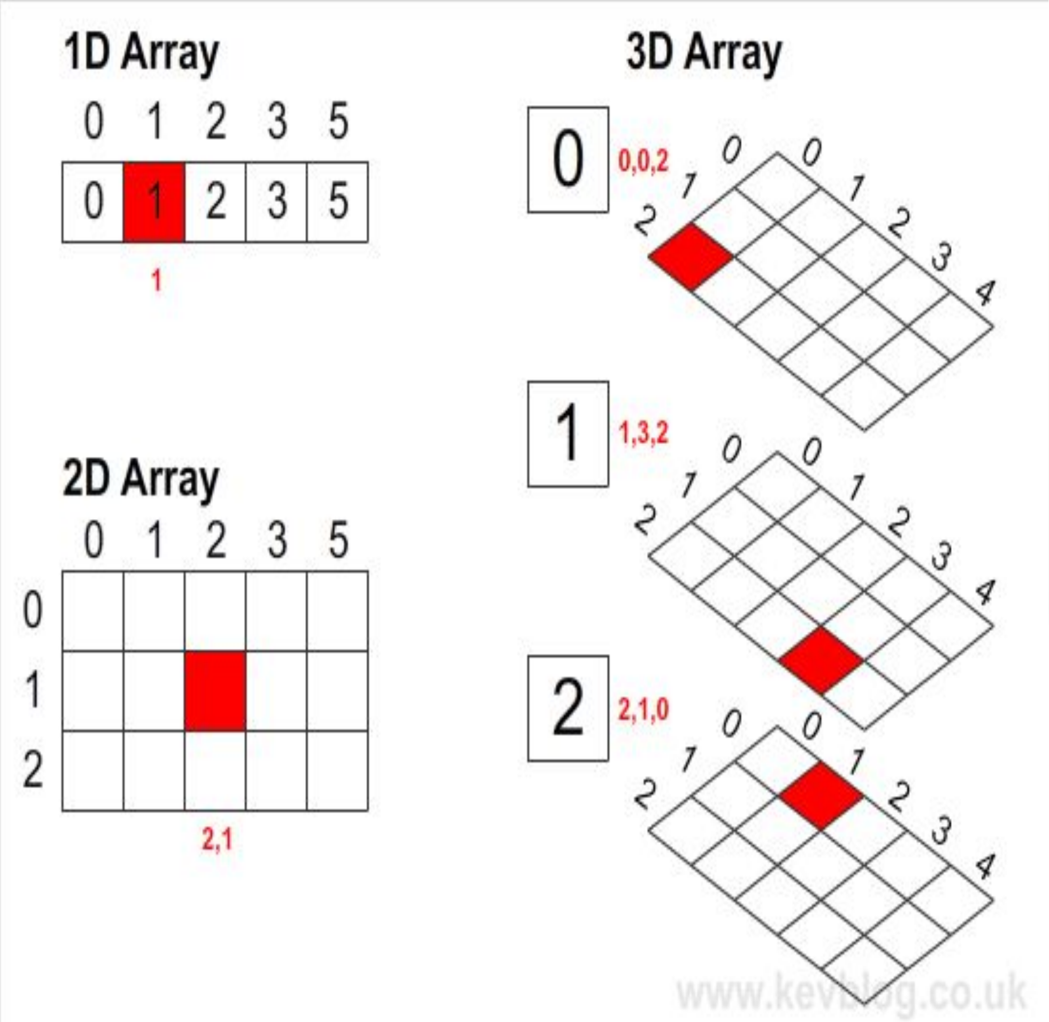


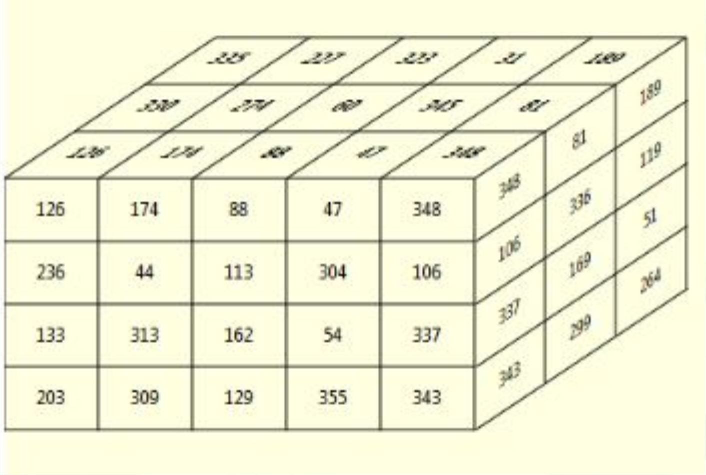
Fig. 1: OpenCV Image Data Array



Each element of the data in a cv::Mat can itself be either a single number, or multiple numbers. In the case of multiple numbers, this is what the library refers to as a multichannel array.

Large Array Types - cv::Mat





126	174	88	47	348
236	44	113	304	106
133	313	162	54	337
203	309	129	355	343

As a fact, an n-dimensional array and an (n-1)-dimensional multichannel array are actually very similar objects.

Large Array Types - cv::Mat

- So, an element of an array can be vector-valued.

Suppose we have a two-dimensional three-channel array of 32-bit floats.
So each element in array will consist of 3 values (because it has 3 channels)
And each such value is a 32-bit float.
So each element is of size = 32 * 3 bits = 12 Bytes

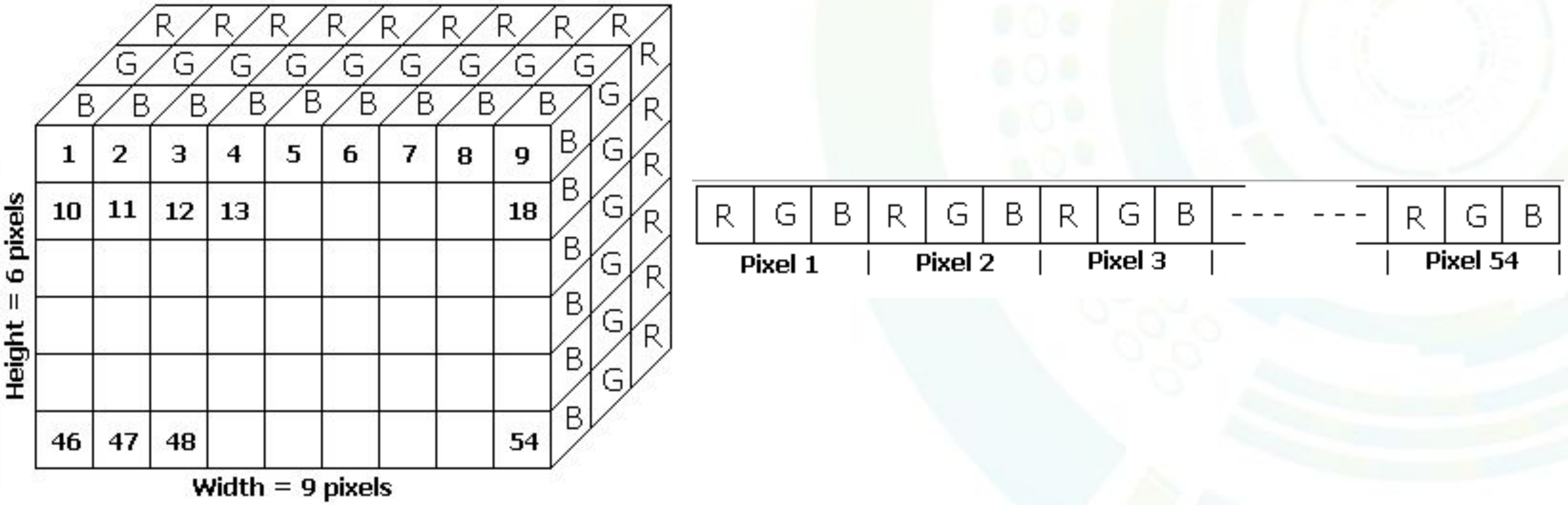


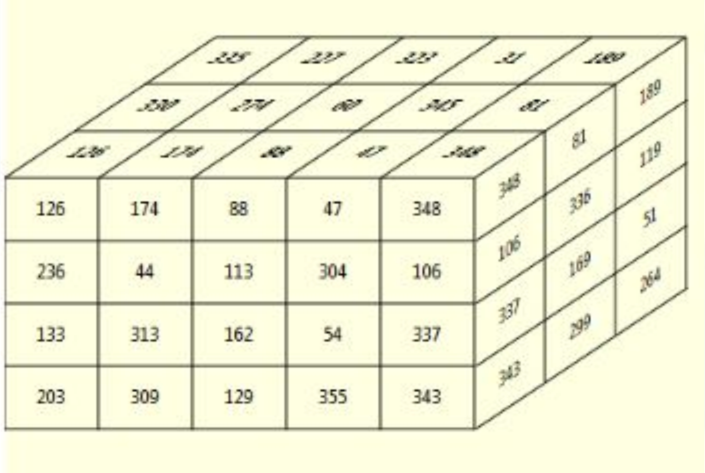
Fig. 1: OpenCV Image Data Array

Large Array Types - cv::Mat

- Class Mat has many members out of which **data, dims, flags, refcount , step[]** are important.
- **The individual elements are accessed by following method (internally)**

$$addr(M_{i_0,\ldots,i_{M.dims-1}}) = M.data + M.step[0] * i_0 + M.step[1] * i_1 + \ldots + M.step[M.dims-1] * i_{M.dims-1}$$

- For e.g. accessing 304 and accessing 119



126	174	88	47	348
236	44	113	304	106
133	313	162	54	337
203	309	129	355	343

Large Array Types - cv::Mat

- Creating an Array :
  - cv::Mat m;
    - An array created in this manner has no size and no data type.
    - Can be later asked to allocate data by using a member function such as create()

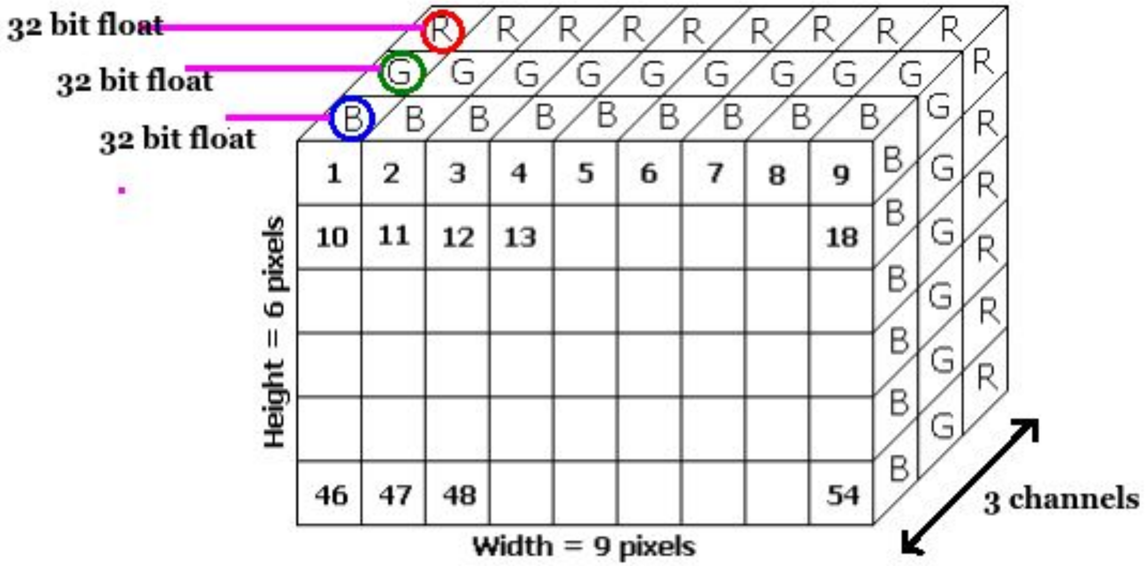
m.create(6, 9, CV_32FC3);
Configures the array to represent a two-dimensional 3-channel object.
It has 6 rows and 9 columns.
cv::F32C3 tells that each values in an element is 32-bit float value and each element has 3 channels.

m.setTo(cv::Scalar(1.0f, 0.0f, 1.0f))
0th channel is 1.0, 1st 0.0, 2nd 1.0

This is all equal to the following allocation during initialization
cv::Mat m(6, 9, CV_32FC3, cv::Scalar(1.0f, 0.0f, 1.0f));

Large Array Types - cv::Mat

- **m.create(6,9,CV_32FC3)**



OpenCV Image Data Array

m.create(6, 9, CV_32FC3)

Compare the above scenario with that of 3 dimesional single channel matrix i.e.

int sizes[3] = {6, 9, 3};

Mat M(3, sizes, CV_32F);

What changes do we observe in terms of storage and access ?

Large Array Types - cv::Mat

- **Channel Type Flags** :

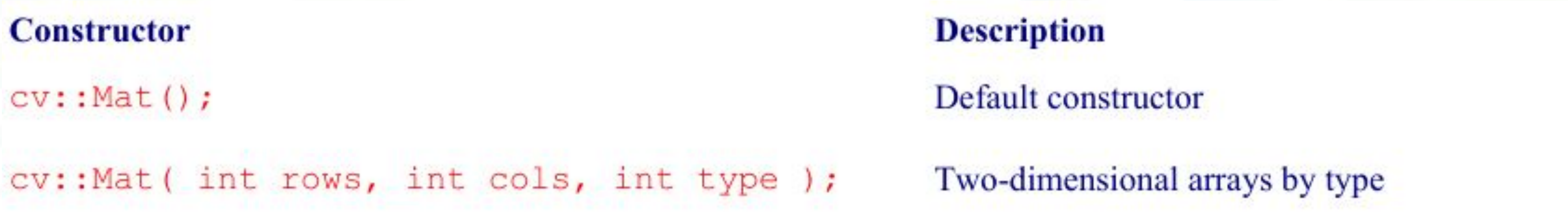
CV_{8U,16S,16U,32S,32F,64F}C{1,2,3,4}
The type of an array can be defined using above flags.
u for unsigned character (upto 255)

Arrays with more than 4 channels are allowed.
For that we have to use function CV_{8U,16S,16U,32S,32F,64F}C().
So if you want 7 channel 8-bit array, there is no macro for CV_8UC7, so we have to use m.create(3, 10, CV_8UC(7))

CV_8U - 8-bit unsigned integers (0..255) [b]
CV_8S - 8-bit signed integers (-128..127)
CV_16U - 16-bit unsigned integers (0..65535) [short int] [w]
CV_16S - 16-bit signed integers (-32768..32767) [short int] [s]
CV_32S - 32-bit signed integers (-2147483648..2147483647) [i]
CV_32F - 32-bit floating-point numbers (-FLT_MAX..FLT_MAX, INF, NAN) [f]
CV_64F - 64-bit floating-point numbers (-DBL_MAX..DBL_MAX, INF, NAN) [d]

Large Array Types - cv::Mat

- **Complete list of the CV::Mat constructors** : (Method 1)
  - Different ways we can create a cv::Mat
  - You will mostly use a small fraction of these most of the time :)



Constructor	Description
`cv::Mat();`	Default constructor
`cv::Mat( int rows, int cols, int type );`	Two-dimensional arrays by type

- E.g. cv::Mat M(100, 100, CV_32FC2);
- Creates a 100 x 100 2 channel matrix
- Alternative method

```
cv::Mat M;
M = cv::Mat(100,100,CV_32FC2);
```

- Alternative method 2

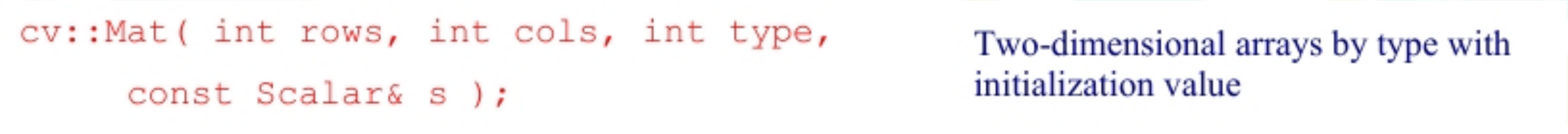
```
cv::Mat M;
M.create(100,100,CV_32FC2);
```

- Alternative method 3

```
cv::Mat M(50,50,CV_32UC3);
M.create(100,100,CV_32UC2); // Turns M into a new array.
                            // Old content will be deallocated only if shape
                               or type has changed
```

Large Array Types - cv::Mat

❖ **Complete list of the CV::Mat constructors : (Method 2)**



`cv::Mat( int rows, int cols, int type,`
`const Scalar& s );`
	Two-dimensional arrays by type with initialization value

- E.g. cv::Mat mat(50, 50, CV_8UC3, cv::Scalar(5,10,15));
- Creates a 50 x 50 3 channel matrix with 0th channel filled with all 5s, 1st channel filled with all 10s, 2nd channel filled with all 15s.
- Alternative method 1

  ```
  cv::Mat mat;
  mat = cv::Mat(50, 50, CV_8UC3, cv::Scalar(5,10,15));
  ```

- Alternative method 2

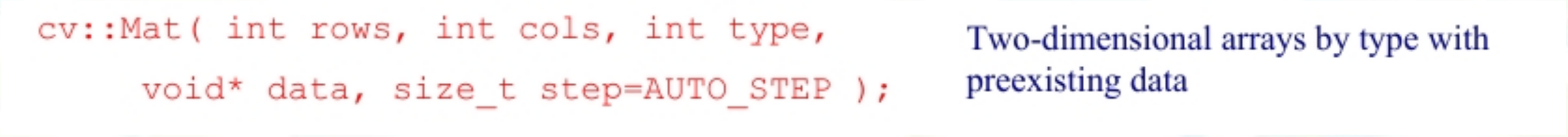
  ```
  cv::Mat mat;
  cv::Scalar S(5,10,15);
  mat = cv::Mat(50,50,CV_8UC3,S);
  ```

- Alternative method 3

  Just replace the last step with
  mat.create(50,50,CV_8UC3,S);

Large Array Types - cv::Mat

❖ **Complete list of the CV::Mat constructors : (Method 3)**



```
cv::Mat( int rows, int cols, int type,
     void* data, size_t step=AUTO_STEP );
```

Two-dimensional arrays by type with preexisting data

- Create a matrix from pre-existing array.
- Difference between matrix v/s array.
- ‘step’ is number of bytes each matrix row occupies. The value should include the padding bytes at the end of each row, if any. If the parameter is missing (set to AUTO_STEP), no padding is assumed and the actual step is calculated as channels*cols*elemSize().
- **Data does not get copied truly. Just creates a pointer to the existing data.**
- E.g. double m[2][2] = {{1.0, 2.0}, {3.0, 4.0}}

  cv::Mat M(2, 2, CV_32F,m);

  //creates an 2 x 2 (single channel) matrix M from a 2 x 2 array m.
- E.g int data[3][2] = {{111,9},{3,7},{2,4}};

  Mat A = Mat(2, 2, CV_32SC2, data);

  //creates an 2 x 2 (two channels) matrix M from a 3 x 2 array m.

  // A(0)(1)[0] will be 3 and A(0)(1)[1] will be 7

Large Array Types - cv::Mat

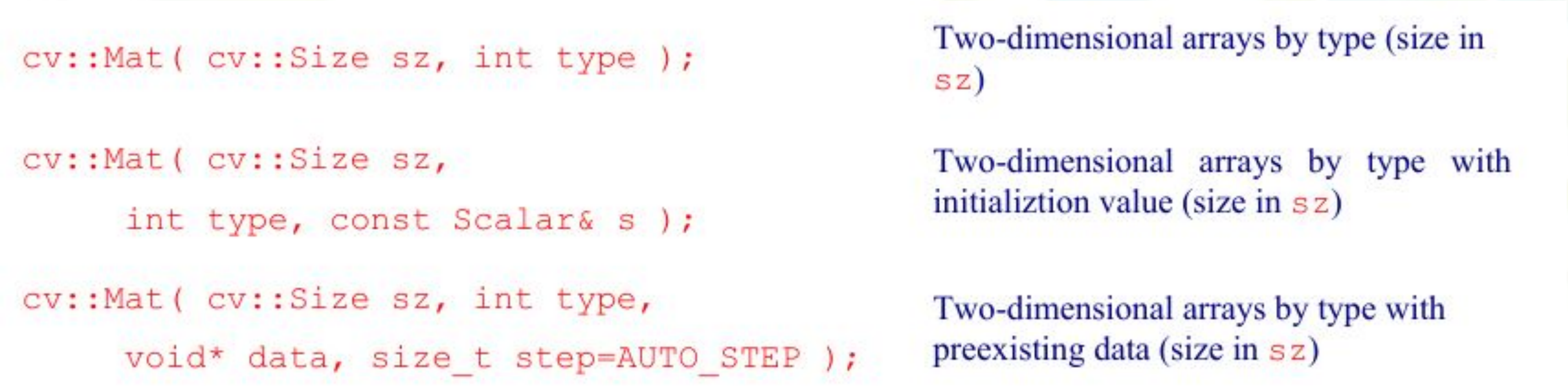
- **Complete list of the CV::Mat constructors** : (Method 3)

```
➢   E.g   unsigned char data1[3][2] = {{265,9},{3,7},{2,4}};
                Mat B = Mat(2, 2, CV_8UC2, data1);
             // same as previous except that the matrix is of unsigned integer and array
                // is of unsigned character.
       // MAKE SURE BOTH MATRIX AND ARRAY ARE OF SAME TYPE .
       // Here B(0)(0)[0] will be 10 and not 265 ‘coz of wrap around.

E.g  float data2 [6] = {125.0,9.0,3.0,7.3,2.4,4.4};
       Mat C = Mat(2, 2, CV_32FC2, data2);
     // create a 2 x 2 (two channel) matrix from a 6 x 1 array.
```

Large Array Types - cv::Mat

- **Complete list of the CV::Mat constructors : (Method 4)**



`cv::Mat( cv::Size sz, int type );`	Two-dimensional arrays by type (size in `sz`)

`cv::Mat( cv::Size sz,`
`int type, const Scalar& s );`
	Two-dimensional arrays by type with initializtion value (size in `sz`)

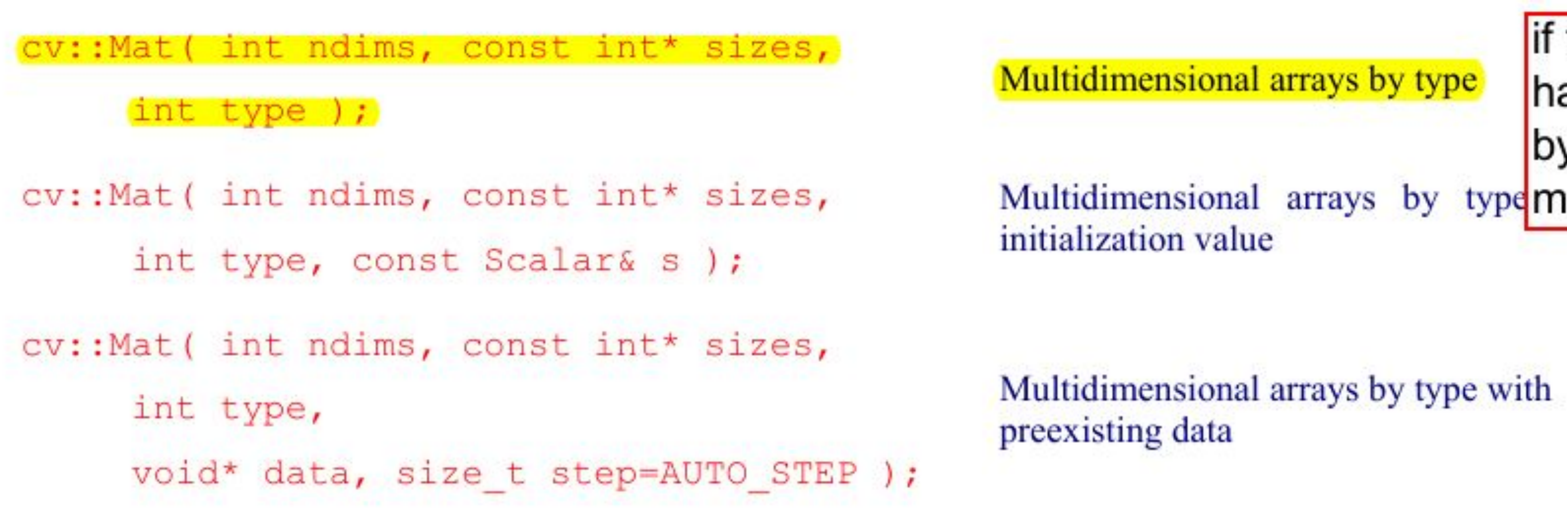
`cv::Mat( cv::Size sz, int type,`
`void* data, size_t step=AUTO_STEP );`
	Two-dimensional arrays by type with preexisting data (size in `sz`)

- **Same as previous methods, except that, now we are using ‘Size sz’ instead of directly specifying ‘rows and columns’**
- **Eg.,**

```
Size sz(10,10);
cv::Mat m(sz,CV_8UC3);
// Creates a 10 x 10 (3 channels) matrix
```

Large Array Types - cv::Mat

❖ Complete list of the CV::Mat constructors : (Method 5)



```
cv::Mat( int ndims, const int* sizes,
     int type );
```

Multidimensional arrays by type

```
cv::Mat( int ndims, const int* sizes,
     int type, const Scalar& s );
```

Multidimensional arrays by type initialization value

```
cv::Mat( int ndims, const int* sizes,
     int type,
     void* data, size_t step=AUTO_STEP );
```

Multidimensional arrays by type with preexisting data

if the argumts have int followed by sizes then it's multidim. array

- ➢ To create multidimensional arrays.
- ➢ The first argument ‘ndims’ specifies the number of dimensions.
- ➢ ‘sizes’ will contains the length of each dimension.
- ➢ E.g. int sizes[3] = {7, 8, 9};

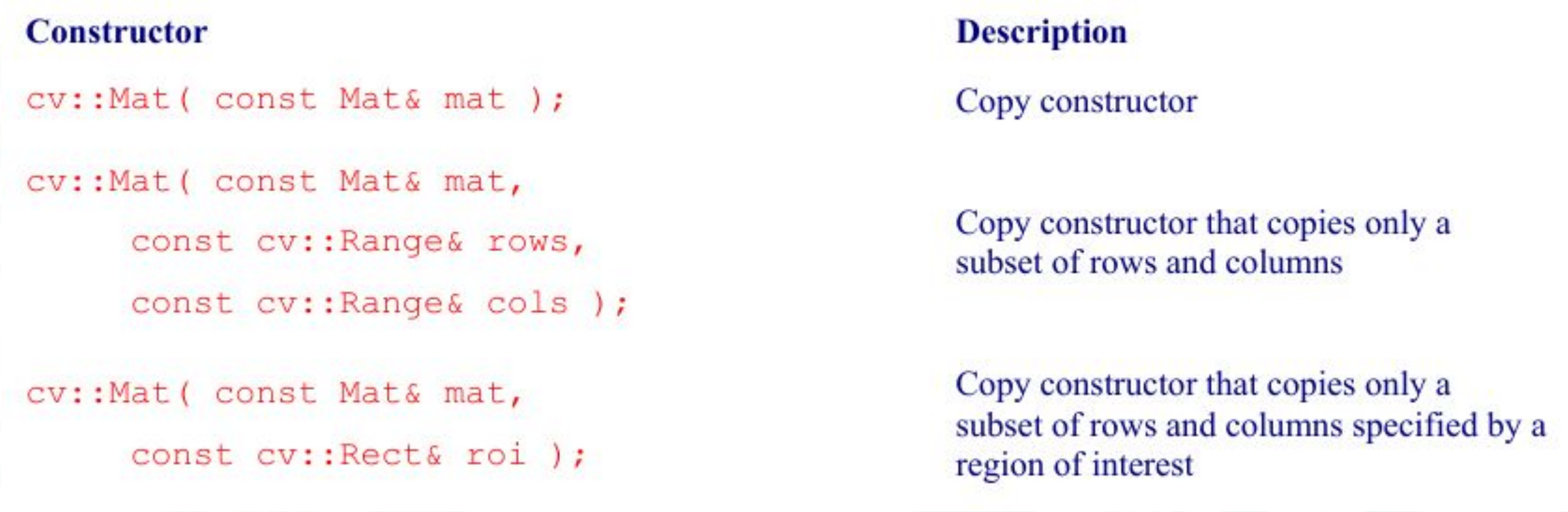
  Mat M(3, sizes, CV_8U, Scalar::all(0));

  // Creates a 3 dimensional array, where the size of each dimension is 7, 8

  // and 9 respectively. The array is filled entirely with 0.

Large Array Types - cv::Mat

- **Complete list of the CV::Mat constructors** : (Method 6)



Constructor	Description
`cv::Mat( const Mat& mat );`	Copy constructor

`cv::Mat( const Mat& mat,`
`const cv::Range& rows,`
`const cv::Range& cols );`
	Copy constructor that copies only a subset of rows and columns

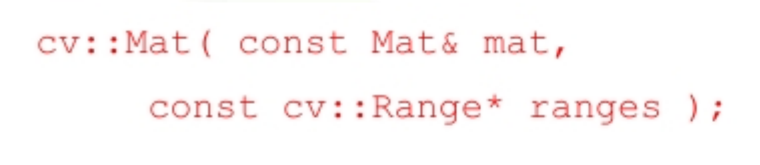
`cv::Mat( const Mat& mat,`
`const cv::Rect& roi );`
	Copy constructor that copies only a subset of rows and columns specified by a region of interest

- Constructors that copy data from other cv::Mat.
- Works only on a two-dimensional matrix.
- cv::Range[start,end] to specify the limits.
- cv::Rect to specify a rectangular sub-region.
- E.g.

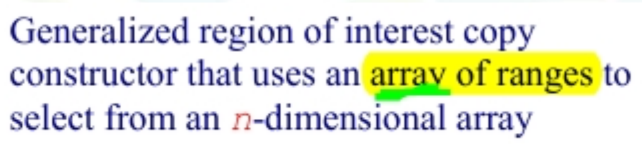
```
Mat M(10,10, CV_8U, Scalar::all(0));
cv::Range rows(3,7);
cv::Range cols(2,6);
Mat B(M,rows,cols);
// Creates a sub-matrix B out of matrix M
```

Large Array Types - cv::Mat

❖ **Complete list of the CV::Mat constructors** : (Method 7)



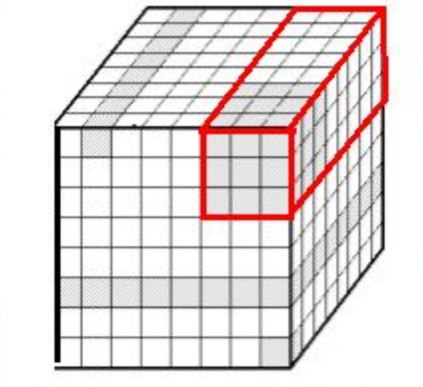
```
cv::Mat( const Mat& mat,
     const cv::Range* ranges );
```



Generalized region of interest copy constructor that uses an array of ranges to select from an *n*-dimensional array

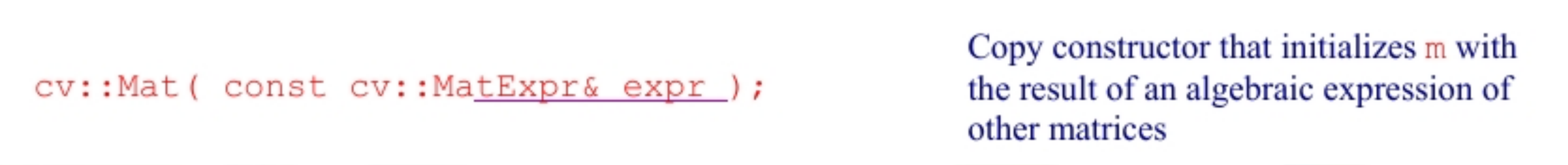
- These are also constructors that copy data from other cv::Mat.
- But works on any dimensional matrix.
- Ranges is not just a single range but array of ranges. Where each range is for each dimension
- E.g.

```
int size[3]={8,8,8};
Mat M(3,size, CV_8UC3, Scalar::all(0));
cv::Range rows(0,2);
cv::Range cols(6,8);
cv::Range dim3rd = Range::all();
cv::Range ranges[3]={rows,cols,dim3rd};
Mat B(M,ranges);
// Creates a sub-matrix B out of 3-dimensional matrix M.
```



Large Array Types - cv::Mat

❖ **Complete list of the CV::Mat constructors : (Method 8)**



```
cv::Mat( const cv::MatExpr& expr );
```

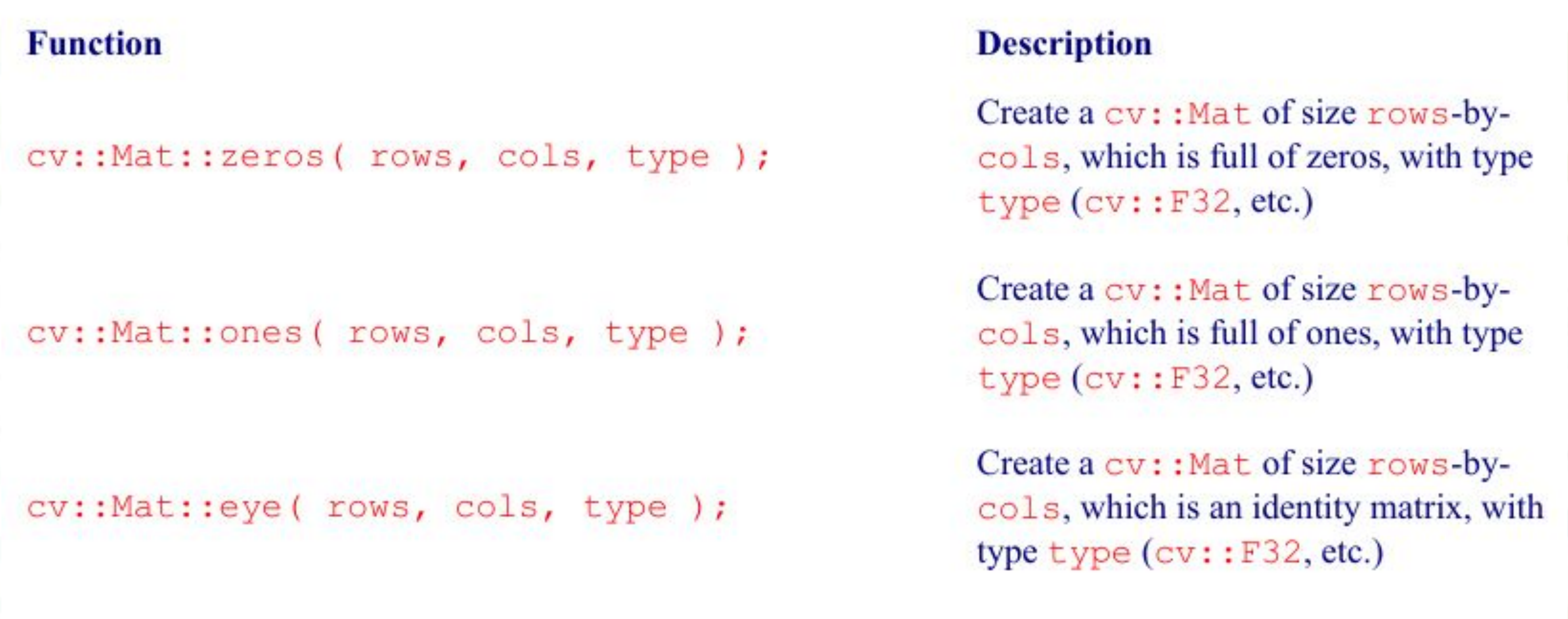
Copy constructor that initializes m with the result of an algebraic expression of other matrices

- ➢ **Create a matrix out of matrix expression**
- ➢ **The expression can contain 1 or many matrices.**
- ➢ **Examples of expressions are** A+B , A-s, A.mul(B), A*B, A.t() etc.,
- ➢ **E.g.**

```
Mat A(10,10, CV_8U, Scalar::all(0));
Mat B(10,10, CV_8U, Scalar::all(1));
Mat C(A+B);
// Creates a sub-matrix C=A+B
```

Large Array Types - cv::Mat

- **Complete list of the CV::Mat constructors** : (Method 9)



Function	Description
`cv::Mat::zeros( rows, cols, type );`	Create a `cv::Mat` of size `rows`-by-`cols`, which is full of zeros, with type `type` (`cv::F32`, etc.)
`cv::Mat::ones( rows, cols, type );`	Create a `cv::Mat` of size `rows`-by-`cols`, which is full of ones, with type `type` (`cv::F32`, etc.)
`cv::Mat::eye( rows, cols, type );`	Create a `cv::Mat` of size `rows`-by-`cols`, which is an identity matrix, with type `type` (`cv::F32`, etc.)

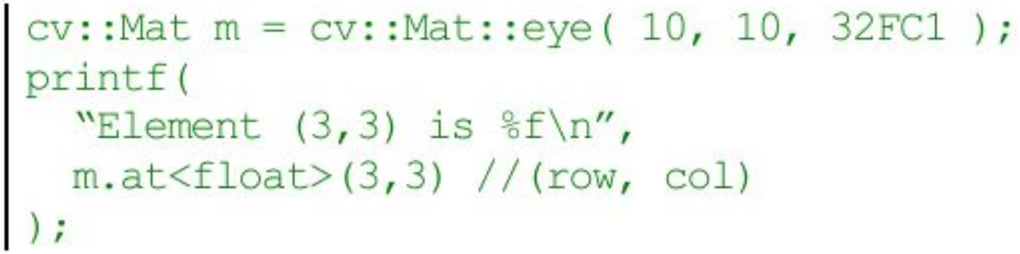
- Methods to create null matrix, matrix of 1s,identity matrix
- In the case of cv::Mat::eye() and cv::Mat::ones(), if the array created is multi-channel, only the first channel will be set 1.0 while the other channels will be 0.0.
- The following methods will do 'true copy':

Mat M2 = M1.clone();	Makes M2 a copy of M1
Mat M2; M1.copyTo(M2);	Makes M2 a copy of M1

Large Array Types - cv::Mat

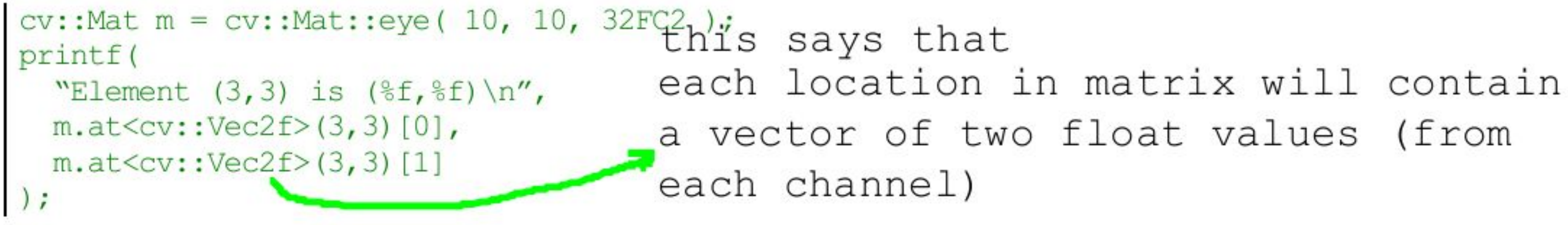
❖ Accessing the Matrix Elements:

- ➢ The basic means of direct access is the (template) member function at<>()[].
- ➢ <> will contain type of element that the matrix contains.
- ➢ () will contain the row,col number
- ➢ [] will contain the channel number



```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC1 );
printf(
  “Element (3,3) is %f\n”,
  m.at<float>(3,3) //(row, col)
);
```

For a multichannel array, the analogous example would look like this:



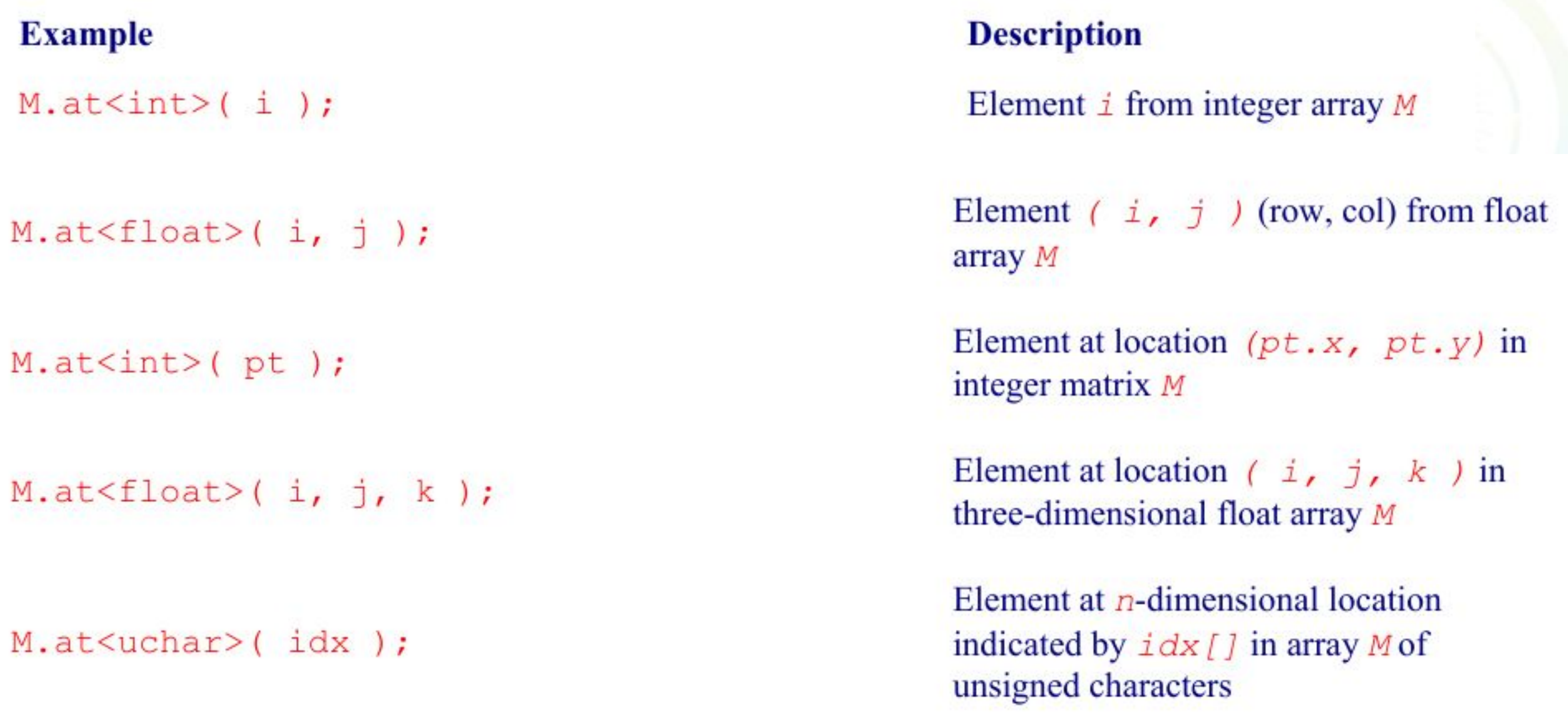
```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC2 );
printf(
  “Element (3,3) is (%f,%f)\n”,
  m.at<cv::Vec2f>(3,3)[0],
  m.at<cv::Vec2f>(3,3)[1]
);
```

this says that
each location in matrix will contain
a vector of two float values (from
each channel)

Large Array Types - cv::Mat

❖ **Accessing the Matrix Elements:**

➢ Various forms of at<> function.



Example	Description
`M.at<int>( i );`	Element *i* from integer array *M*
`M.at<float>( i, j );`	Element *(i, j)* (row, col) from float array *M*
`M.at<int>( pt );`	Element at location *(pt.x, pt.y)* in integer matrix *M*
`M.at<float>( i, j, k );`	Element at location *(i, j, k)* in three-dimensional float array *M*
`M.at<uchar>( idx );`	Element at *n*-dimensional location indicated by *idx[]* in array *M* of unsigned characters

Large Array Types - cv::Mat

- **Block Accessing the Matrix Elements:**

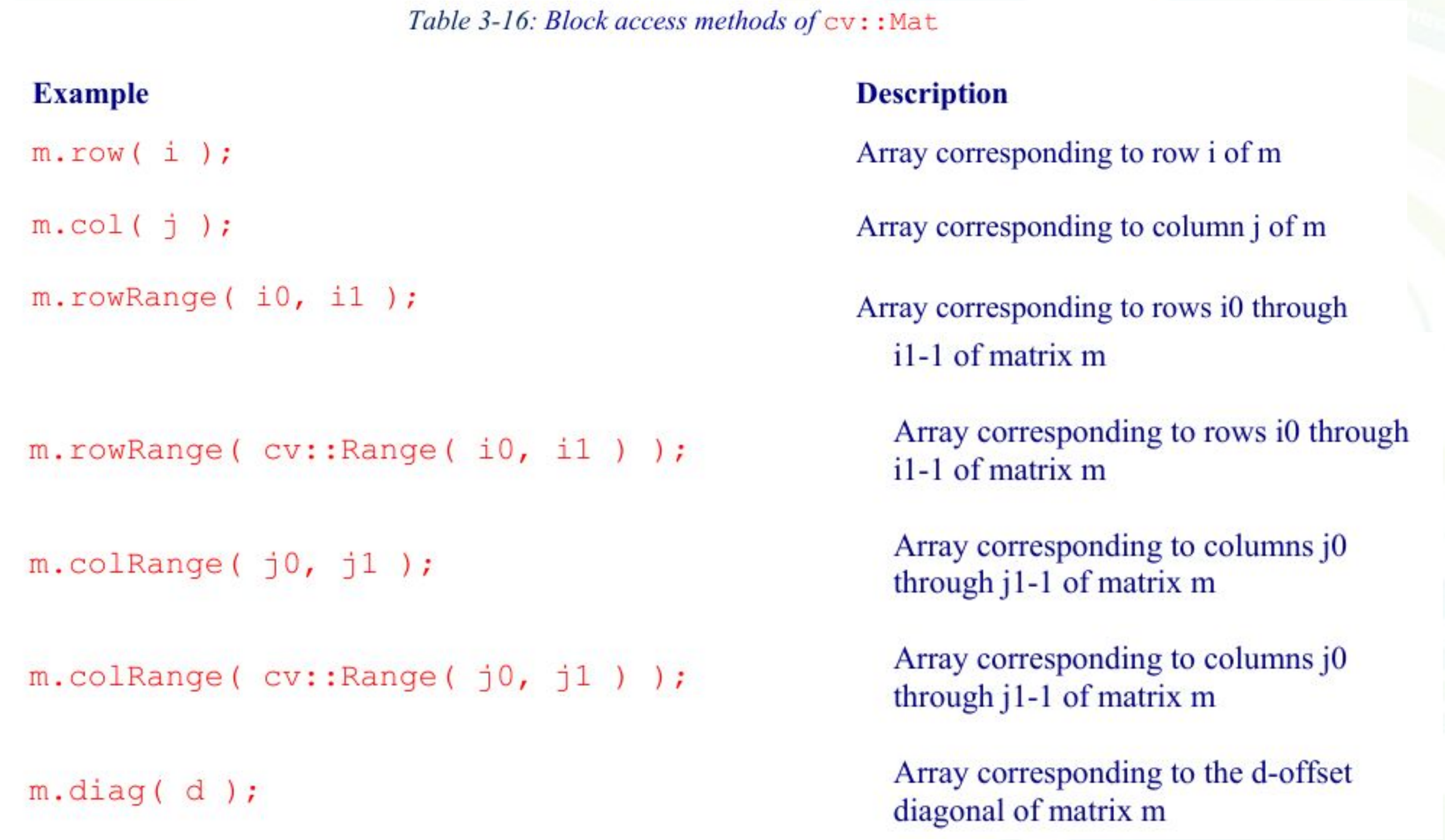


Table 3-16: Block access methods of `cv::Mat`

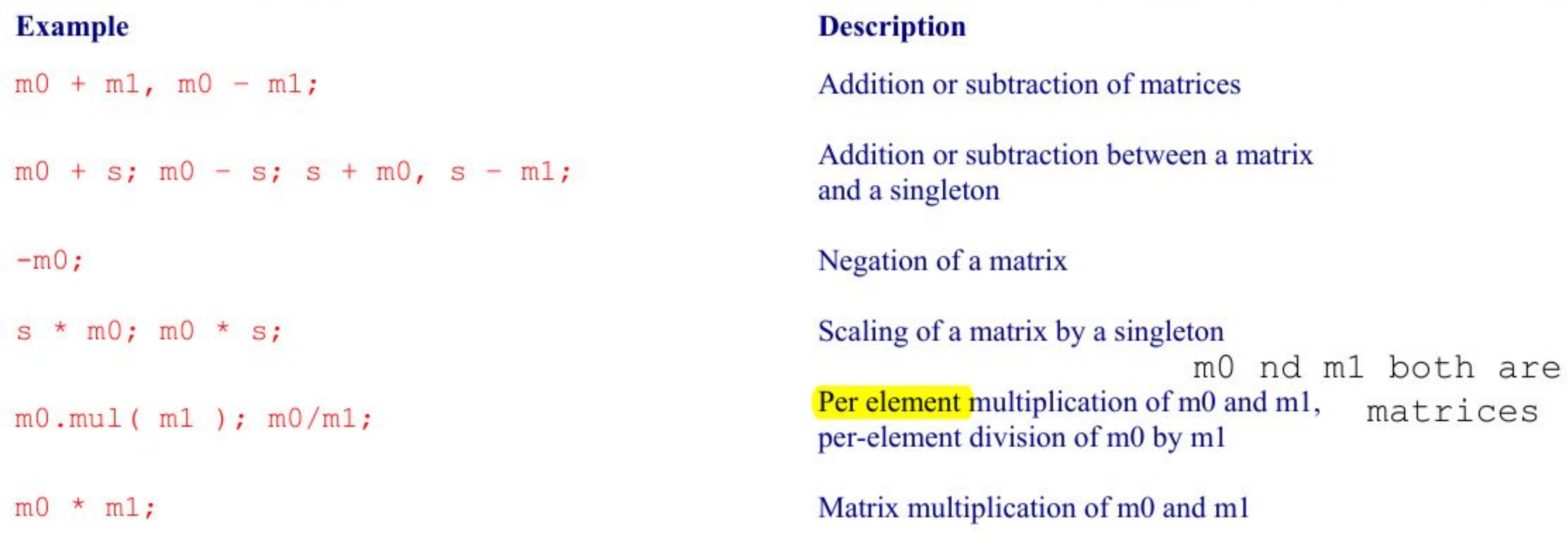
Example	Description
`m.row( i );`	Array corresponding to row i of m
`m.col( j );`	Array corresponding to column j of m
`m.rowRange( i0, i1 );`	Array corresponding to rows i0 through i1-1 of matrix m
`m.rowRange( cv::Range( i0, i1 ) );`	Array corresponding to rows i0 through i1-1 of matrix m
`m.colRange( j0, j1 );`	Array corresponding to columns j0 through j1-1 of matrix m
`m.colRange( cv::Range( j0, j1 ) );`	Array corresponding to columns j0 through j1-1 of matrix m
`m.diag( d );`	Array corresponding to the d-offset diagonal of matrix m

- **IMPORTANT : If a sub-matrix is modified then even the main matrix will be modified.**

Matrix Expressions

❖ Matrix Expressions:

- ➢ These all are of form cv::MatExpr.
- ➢ So if you find cv::MatExpr& exp in any function arguments, it means that it can be replaced with any of the following expressions.
- ➢ m2=m1 ,here, m2 would be another reference to the data in m1.
- ➢ But m2=m1+m0 means, it will be evaluated and the results will reside in a newly allocated data area. m2 will point to this new data.

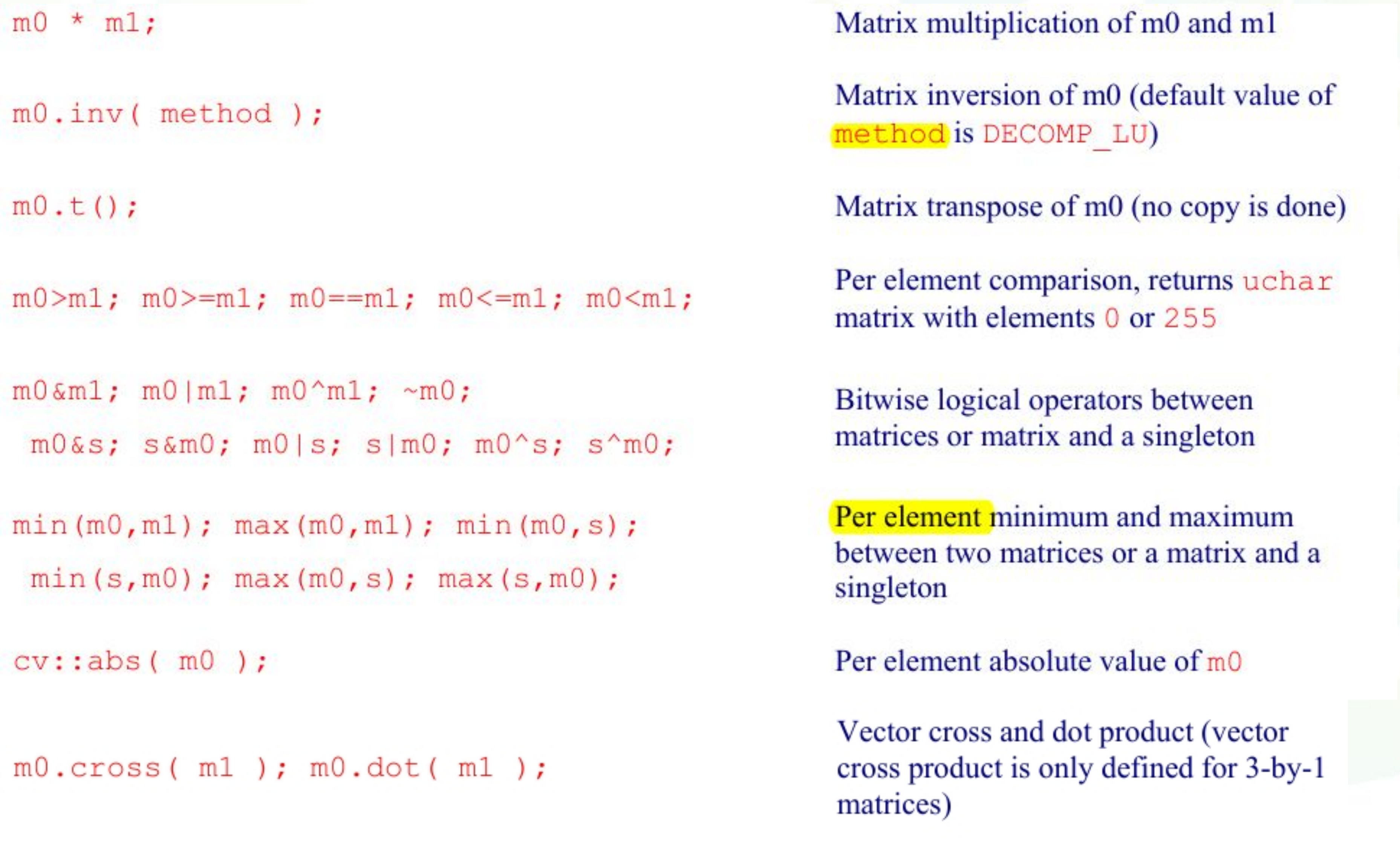


Example	Description
`m0 + m1, m0 - m1;`	Addition or subtraction of matrices
`m0 + s; m0 - s; s + m0, s - m1;`	Addition or subtraction between a matrix and a singleton
`-m0;`	Negation of a matrix
`s * m0; m0 * s;`	Scaling of a matrix by a singleton
`m0.mul( m1 ); m0/m1;`	Per element multiplication of m0 and m1, per-element division of m0 by m1
`m0 * m1;`	Matrix multiplication of m0 and m1

m0 nd m1 both are matrices

Matrix Expressions

❖ **Matrix Expressions:**



`m0 * m1;`	Matrix multiplication of m0 and m1
`m0.inv( method );`	Matrix inversion of m0 (default value of `method` is `DECOMP_LU`)
`m0.t();`	Matrix transpose of m0 (no copy is done)
`m0>m1; m0>=m1; m0==m1; m0<=m1; m0<m1;`	Per element comparison, returns `uchar` matrix with elements `0` or `255`

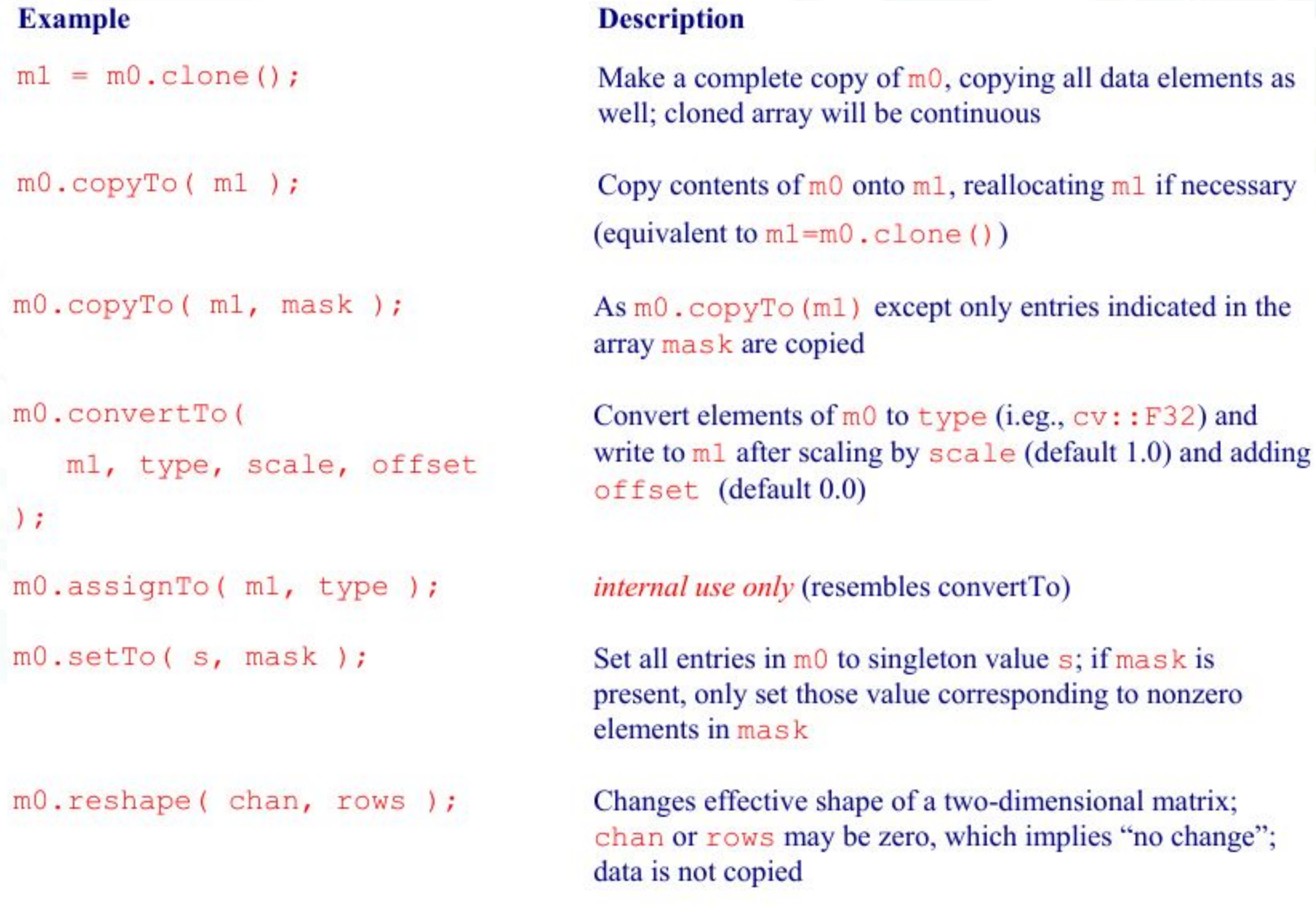
`m0&m1; m0|m1; m0^m1; ~m0;`
`m0&s; s&m0; m0|s; s|m0; m0^s; s^m0;`
	Bitwise logical operators between matrices or matrix and a singleton

`min(m0,m1); max(m0,m1); min(m0,s);`
`min(s,m0); max(m0,s); max(s,m0);`
	Per element minimum and maximum between two matrices or a matrix and a singleton

`cv::abs( m0 );`	Per element absolute value of `m0`
`m0.cross( m1 ); m0.dot( m1 );`	Vector cross and dot product (vector cross product is only defined for 3-by-1 matrices)

Matrix Expressions

- **More things array can do:**



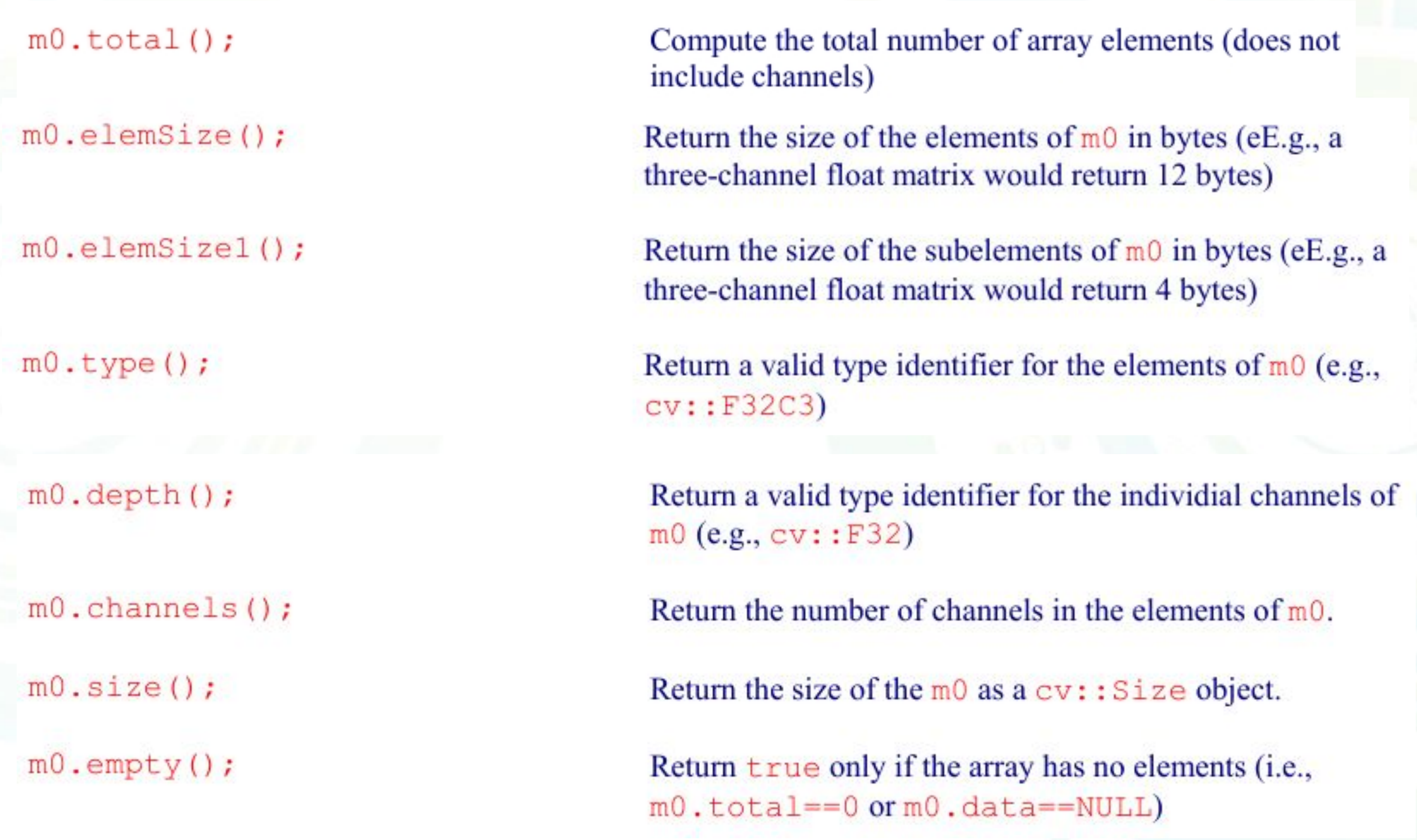
Example	Description
`m1 = m0.clone();`	Make a complete copy of `m0`, copying all data elements as well; cloned array will be continuous
`m0.copyTo( m1 );`	Copy contents of `m0` onto `m1`, reallocating `m1` if necessary (equivalent to `m1=m0.clone()`)
`m0.copyTo( m1, mask );`	As `m0.copyTo(m1)` except only entries indicated in the array `mask` are copied

`m0.convertTo(`
`m1, type, scale, offset`
`);`
	Convert elements of `m0` to `type` (i.eg., `cv::F32`) and write to `m1` after scaling by `scale` (default 1.0) and adding `offset` (default 0.0)

`m0.assignTo( m1, type );`	*internal use only* (resembles convertTo)
`m0.setTo( s, mask );`	Set all entries in `m0` to singleton value `s`; if `mask` is present, only set those value corresponding to nonzero elements in `mask`
`m0.reshape( chan, rows );`	Changes effective shape of a two-dimensional matrix; `chan` or `rows` may be zero, which implies “no change”; data is not copied

Matrix Expressions

❖ **More things array can do:**

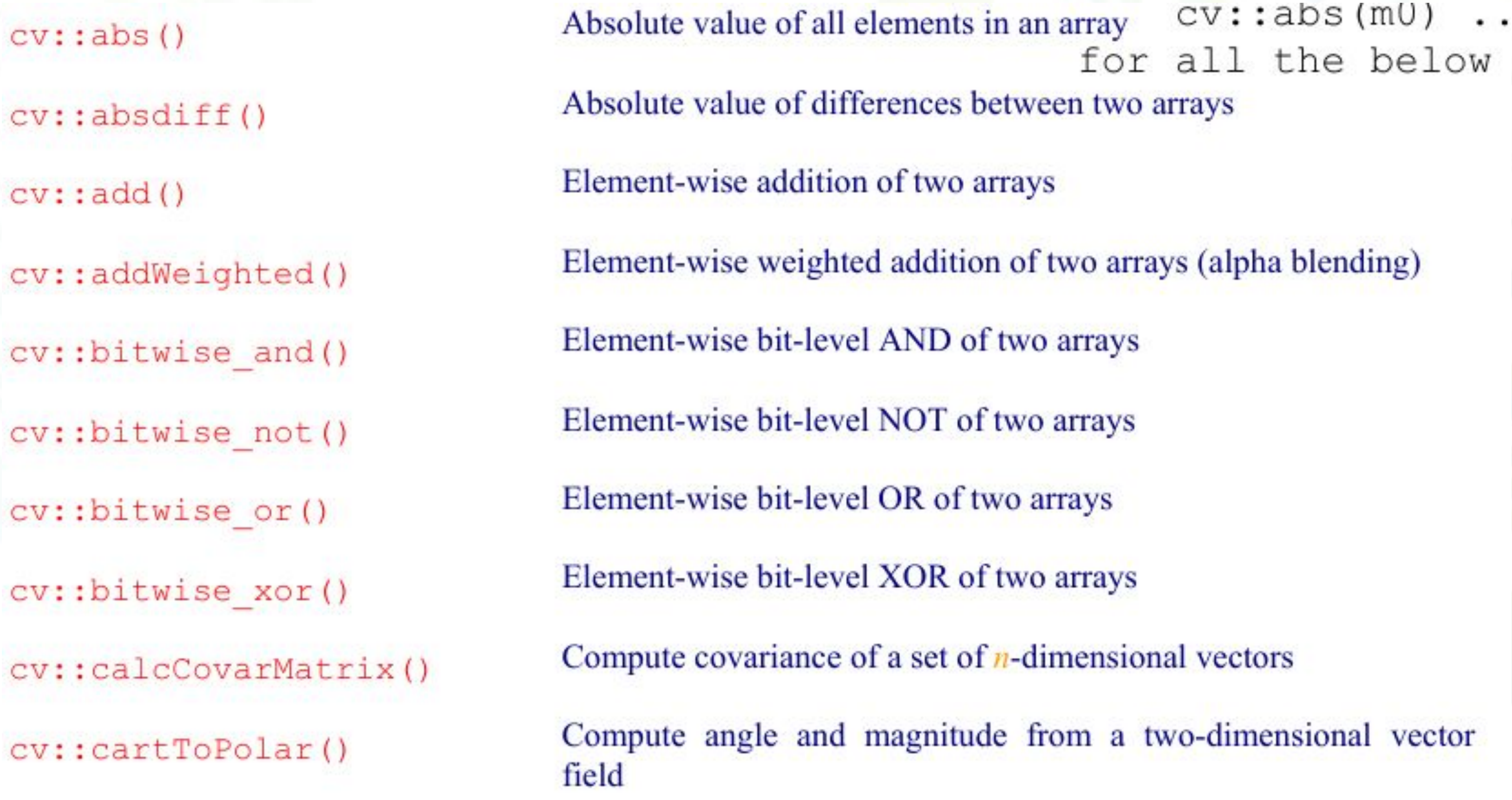


`m0.total();`	Compute the total number of array elements (does not include channels)
`m0.elemSize();`	Return the size of the elements of `m0` in bytes (eE.g., a three-channel float matrix would return 12 bytes)
`m0.elemSize1();`	Return the size of the subelements of `m0` in bytes (eE.g., a three-channel float matrix would return 4 bytes)
`m0.type();`	Return a valid type identifier for the elements of `m0` (e.g., `cv::F32C3`)
`m0.depth();`	Return a valid type identifier for the individial channels of `m0` (e.g., `cv::F32`)
`m0.channels();`	Return the number of channels in the elements of `m0`.
`m0.size();`	Return the size of the `m0` as a `cv::Size` object.
`m0.empty();`	Return `true` only if the array has no elements (i.e., `m0.total==0` or `m0.data==NULL`)

Array Operators

❖ Array Operators:

- These are not member functions.
- These are extra operations that are most naturally represented as “friend” functions that either take array types as arguments, have array types as return values, or both.
- So, they can't be used as m0.abs() but only as cv::abs(m0) .. They support mask and dst.
- Following is list of only the important array operators:

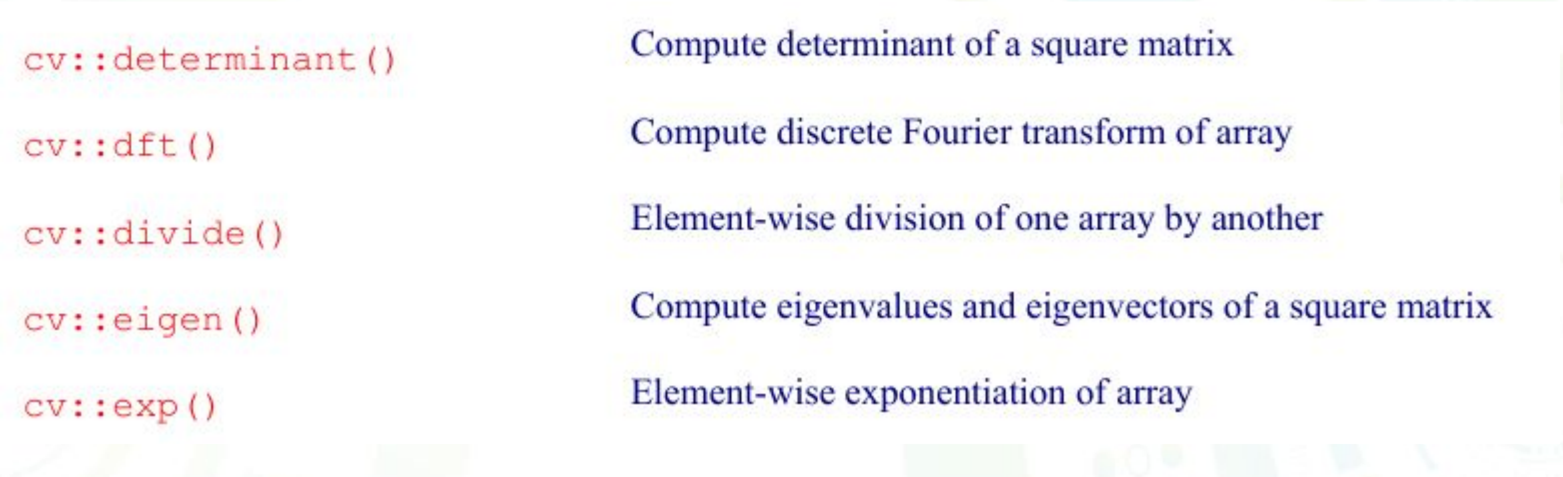


cv::abs(m0) .. for all the below

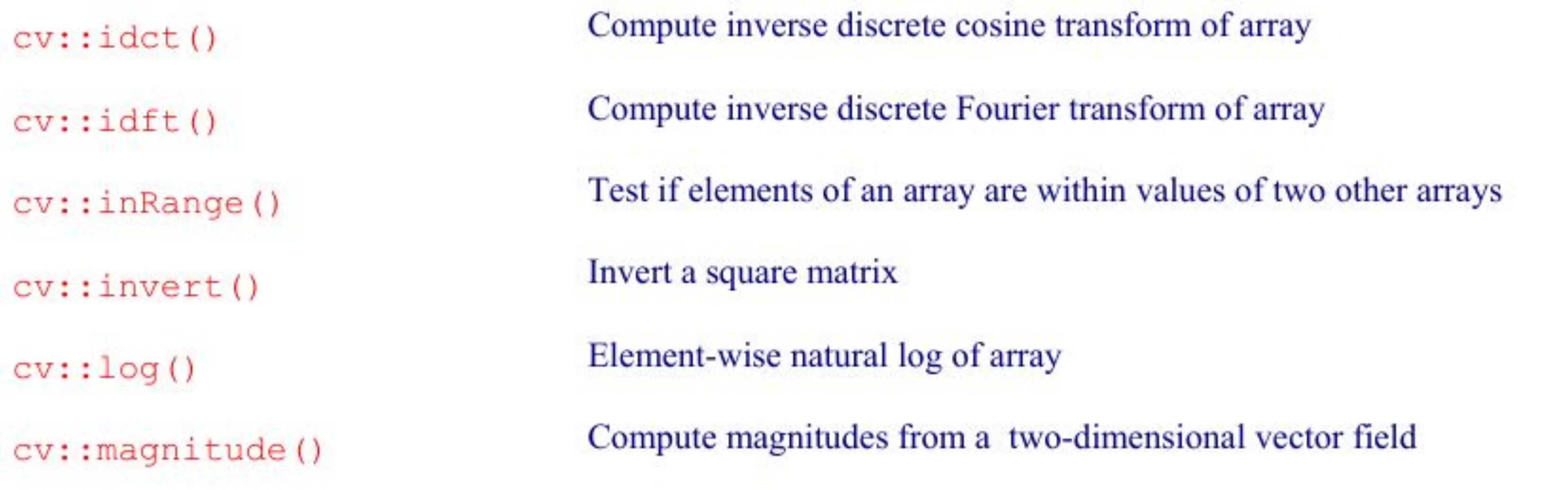
`cv::abs()`	Absolute value of all elements in an array
`cv::absdiff()`	Absolute value of differences between two arrays
`cv::add()`	Element-wise addition of two arrays
`cv::addWeighted()`	Element-wise weighted addition of two arrays (alpha blending)
`cv::bitwise_and()`	Element-wise bit-level AND of two arrays
`cv::bitwise_not()`	Element-wise bit-level NOT of two arrays
`cv::bitwise_or()`	Element-wise bit-level OR of two arrays
`cv::bitwise_xor()`	Element-wise bit-level XOR of two arrays
`cv::calcCovarMatrix()`	Compute covariance of a set of n-dimensional vectors
`cv::cartToPolar()`	Compute angle and magnitude from a two-dimensional vector field

Array Operators

❖ **Array Operators:**



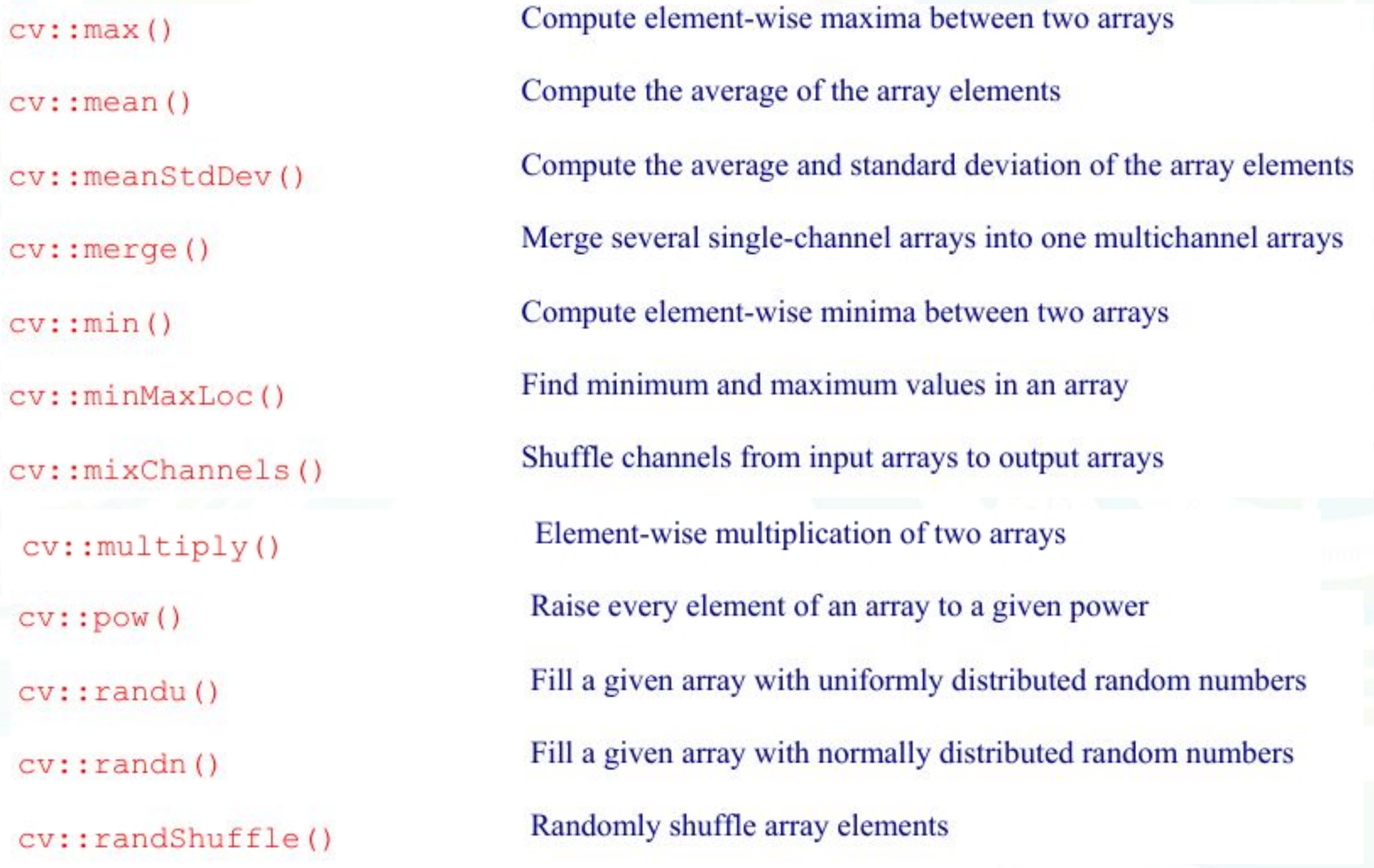
`cv::determinant()`	Compute determinant of a square matrix
`cv::dft()`	Compute discrete Fourier transform of array
`cv::divide()`	Element-wise division of one array by another
`cv::eigen()`	Compute eigenvalues and eigenvectors of a square matrix
`cv::exp()`	Element-wise exponentiation of array



`cv::idct()`	Compute inverse discrete cosine transform of array
`cv::idft()`	Compute inverse discrete Fourier transform of array
`cv::inRange()`	Test if elements of an array are within values of two other arrays
`cv::invert()`	Invert a square matrix
`cv::log()`	Element-wise natural log of array
`cv::magnitude()`	Compute magnitudes from a two-dimensional vector field

Array Operators

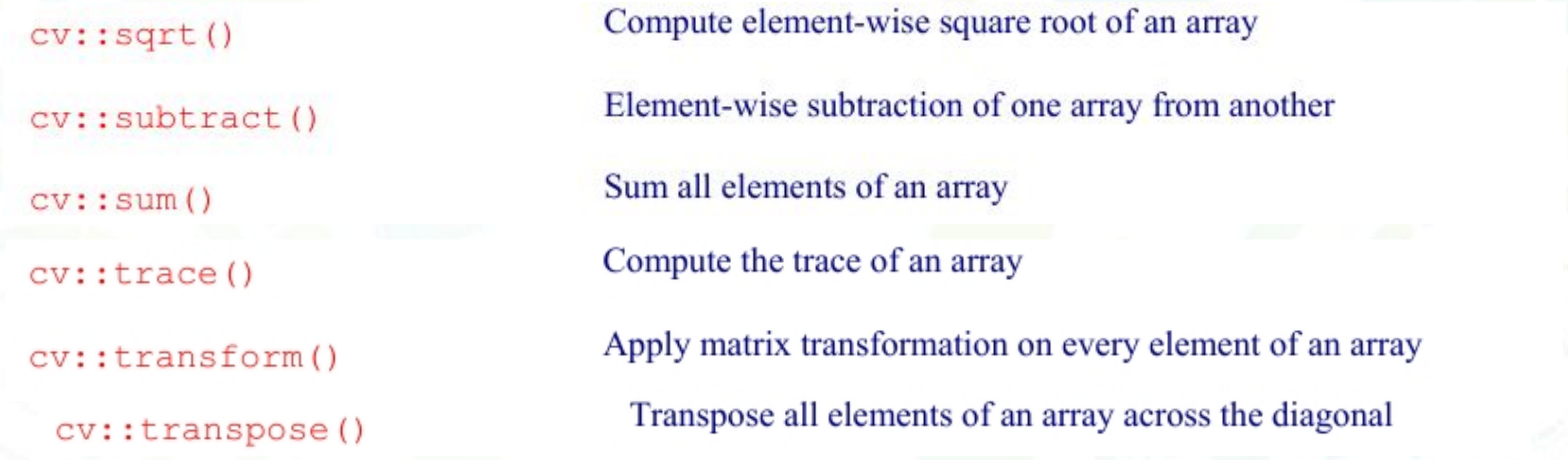
❖ **Array Operators:**



`cv::max()`	Compute element-wise maxima between two arrays
`cv::mean()`	Compute the average of the array elements
`cv::meanStdDev()`	Compute the average and standard deviation of the array elements
`cv::merge()`	Merge several single-channel arrays into one multichannel arrays
`cv::min()`	Compute element-wise minima between two arrays
`cv::minMaxLoc()`	Find minimum and maximum values in an array
`cv::mixChannels()`	Shuffle channels from input arrays to output arrays
`cv::multiply()`	Element-wise multiplication of two arrays
`cv::pow()`	Raise every element of an array to a given power
`cv::randu()`	Fill a given array with uniformly distributed random numbers
`cv::randn()`	Fill a given array with normally distributed random numbers
`cv::randShuffle()`	Randomly shuffle array elements

Array Operators

- **Array Operators:**



`cv::sqrt()`	Compute element-wise square root of an array
`cv::subtract()`	Element-wise subtraction of one array from another
`cv::sum()`	Sum all elements of an array
`cv::trace()`	Compute the trace of an array
`cv::transform()`	Apply matrix transformation on every element of an array
`cv::transpose()`	Transpose all elements of an array across the diagonal

➔ **REFER chapter 3 of “Learning OpenCV” by Bradski for explaination of each operator.**

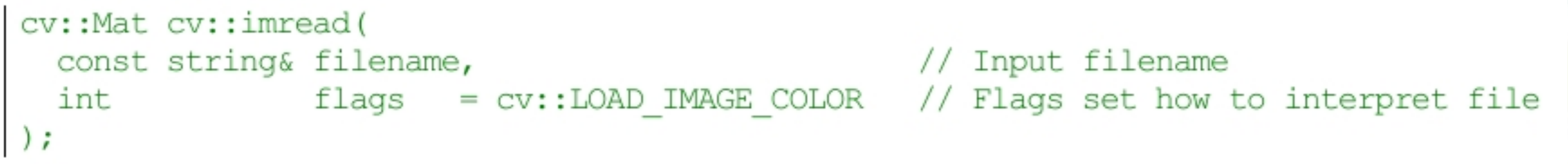
VERY IMP : Refer Pg. 79 of “Learning OpenCV” by Bradski to know about cv::cvtColor() and all of its Conversion codes

There are even operators like cv::cubeRoot() , cv::CV_Error(), cvFloor(), cvRound()

More functions realted to PCA, SVD, RNG are also there.

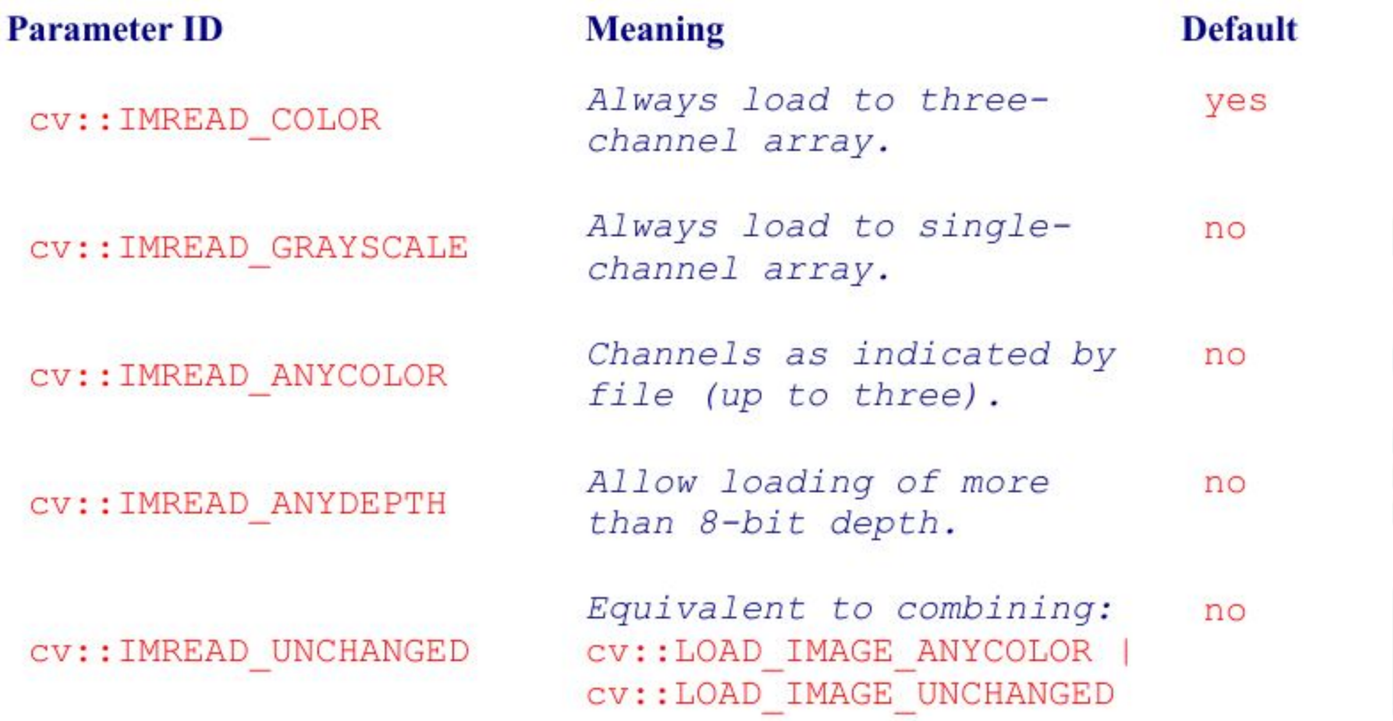
GUI Basics

- **Reading Files with cv::imread()**
  - ➢ Usage



```
cv::Mat cv::imread(
  const string& filename,                          // Input filename
  int           flags    = cv::LOAD_IMAGE_COLOR   // Flags set how to interpret file
);
```

Flags



Parameter ID	Meaning	Default
cv::IMREAD_COLOR	*Always load to three-channel array.*	yes
cv::IMREAD_GRAYSCALE	*Always load to single-channel array.*	no
cv::IMREAD_ANYCOLOR	*Channels as indicated by file (up to three).*	no
cv::IMREAD_ANYDEPTH	*Allow loading of more than 8-bit depth.*	no
cv::IMREAD_UNCHANGED	*Equivalent to combining:* cv::LOAD_IMAGE_ANYCOLOR | cv::LOAD_IMAGE_UNCHANGED	no

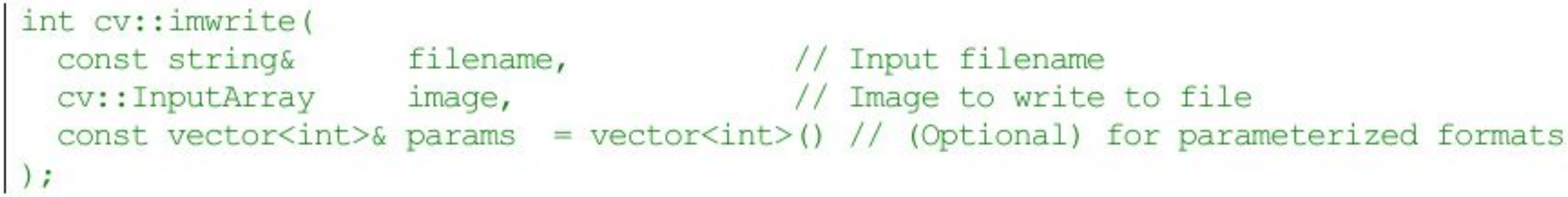
GUI Basics

- cv::imread() does not give a runtime error when it fails to load an image; it simply returns an empty cv::Mat (i.e., empty()==true).

In case of cv::IMREAD_COLOR, even if the image is actually grayscale in the file, the resulting image in memory will still have three channels, with all of the channels containing identical information.

Writing Files with cv::imwrite()

Usage

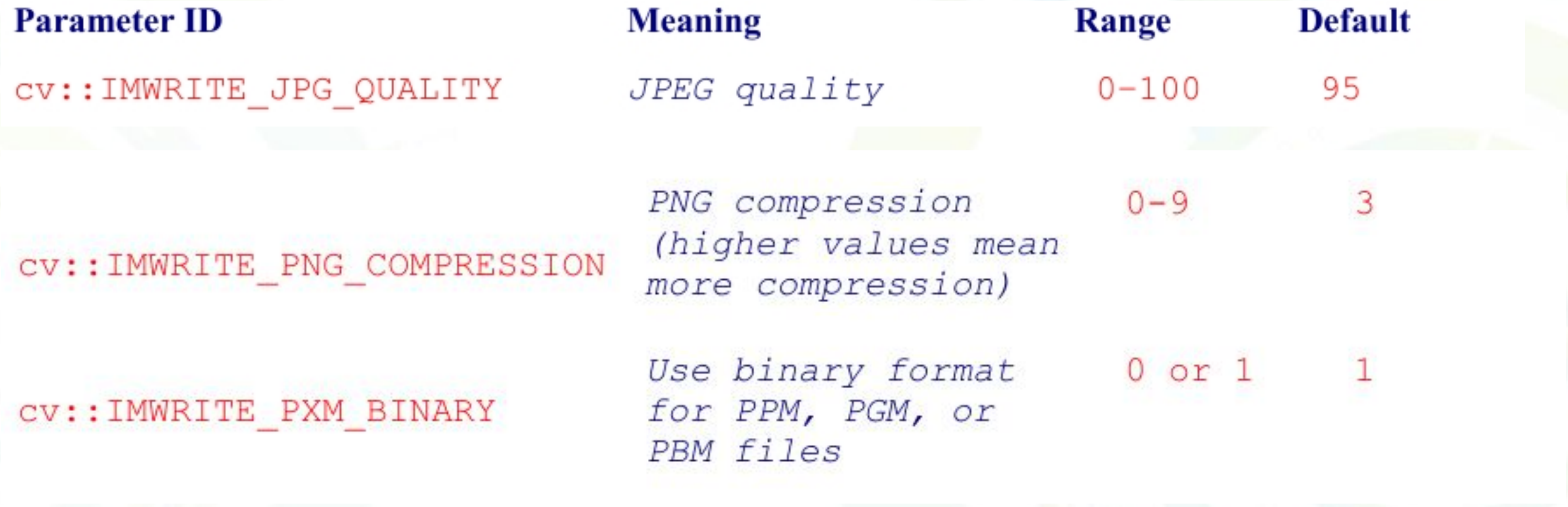


```
int cv::imwrite(
  const string&      filename,                // Input filename
  cv::InputArray     image,                   // Image to write to file
  const vector<int>& params  = vector<int>() // (Optional) for parameterized formats
);
```

The third argument expects a vector of integers. The vector consists of parameters that will be passed to particular file type being created.

GUI Basics

Some of the pre-defined vector aliases are



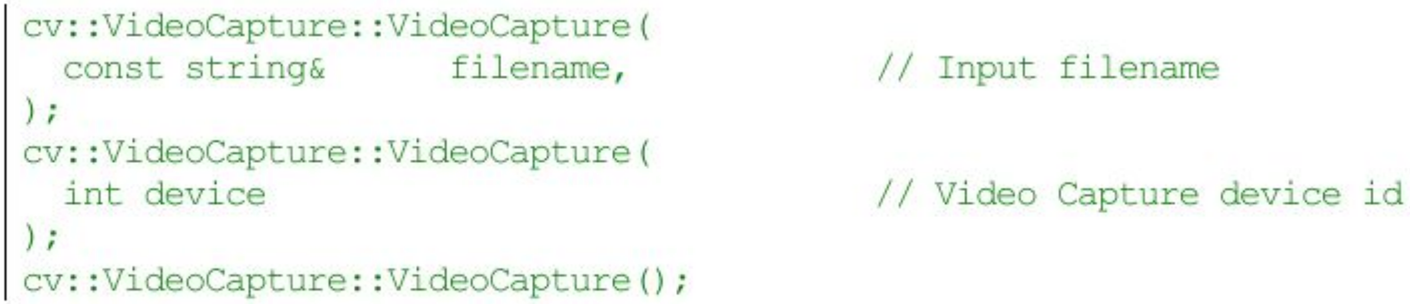
Parameter ID	Meaning	Range	Default
cv::IMWRITE_JPG_QUALITY	*JPEG quality*	0-100	95
cv::IMWRITE_PNG_COMPRESSION	*PNG compression (higher values mean more compression)*	0-9	3
cv::IMWRITE_PXM_BINARY	*Use binary format for PPM, PGM, or PBM files*	0 or 1	1

The return value will be 1 if the save was successful and should be 0 if the save was not.
imwrite() relies heavily on the codecs available in your OS.
OpenCV comes with some in-built Codecs (JPG,PNG and TIFF)

GUI Basics

❖ **Reading Video with the cv::VideoCapture Object:**

- ➢ Object contains the information needed for reading frames from a camera or video file.
- ➢ Usage



```
cv::VideoCapture::VideoCapture(
  const string&      filename,             // Input filename
);
cv::VideoCapture::VideoCapture(
  int device                               // Video Capture device id
);
cv::VideoCapture::VideoCapture();
```

Check the success or failure of the operation by using “ cv::VideoCapture::isOpen()”. Returns true if device/file is opened succesfully.

In the second method, device argument is sum of domain and identifier

device = domain + identifier

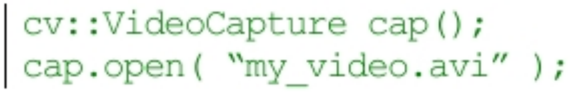
For. e.g to open a fire-wire camera attached to your computer, you have to give device =300, because domain for firewire is 300 and identifier for default device is 0. So device id = 300+0.

Similarly , if we have only one camera then, device = 0 opens it.

There are many typedefs for the domain numbers like

cv::CAP_ANY → 0 , cv::CAP_FIREWIRE → 200, cv::CAP_IEEE1394 → 300, cv::CAP_OPENNI → 900 , cv::CAP_ANDROID → 1000 so on.

For the third method, a dummy object will be created. Then we can use this object to open the source.



```
cv::VideoCapture cap();
cap.open( "my_video.avi" );
```

GUI Basics

- Read the frames:

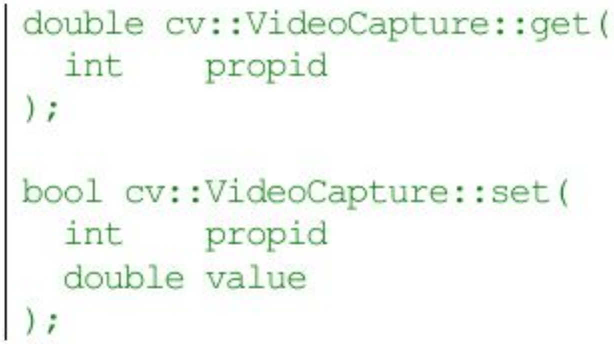
  Can read the Frames with cv::VideoCapture::read()
  E.g→ Mat M;
  cap.read(M);
  Returns TRUE or FALSE

  Or can alse read with cv::VideoCapture::operator>>()
  E.g → Mat M;
  cap >> M;
  Check the completion, by checking whether the array is empty or not.

  KNOW ABOUT grab() and retreive() methods .Useful for stereo-imaging.

Get and set the properties:

Video files/Camera stream also have meta-data like no.of frames , width of frames, video length etc.;

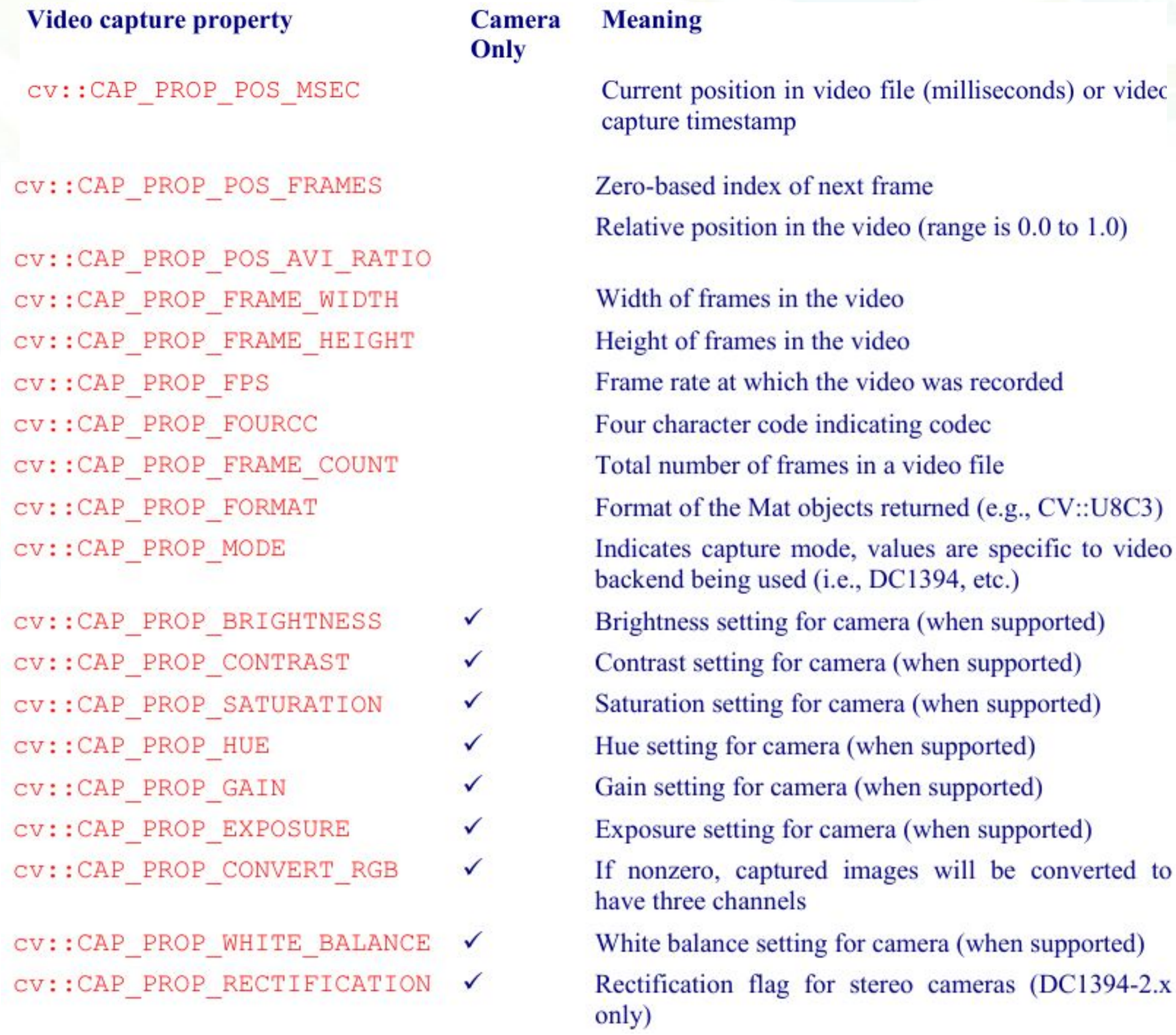


```
double cv::VideoCapture::get(
   int     propid
);

bool cv::VideoCapture::set(
   int     propid
   double  value
);
```

GUI Basics

- Prop.ids



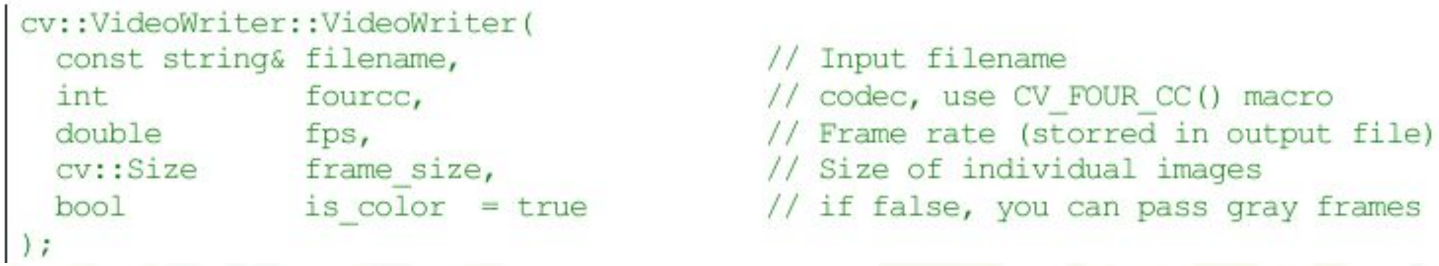
Video capture property	Camera Only	Meaning
cv::CAP_PROP_POS_MSEC		Current position in video file (milliseconds) or videc capture timestamp
cv::CAP_PROP_POS_FRAMES		Zero-based index of next frame
cv::CAP_PROP_POS_AVI_RATIO		Relative position in the video (range is 0.0 to 1.0)
cv::CAP_PROP_FRAME_WIDTH		Width of frames in the video
cv::CAP_PROP_FRAME_HEIGHT		Height of frames in the video
cv::CAP_PROP_FPS		Frame rate at which the video was recorded
cv::CAP_PROP_FOURCC		Four character code indicating codec
cv::CAP_PROP_FRAME_COUNT		Total number of frames in a video file
cv::CAP_PROP_FORMAT		Format of the Mat objects returned (e.g., CV::U8C3)
cv::CAP_PROP_MODE		Indicates capture mode, values are specific to video backend being used (i.e., DC1394, etc.)
cv::CAP_PROP_BRIGHTNESS	✓	Brightness setting for camera (when supported)
cv::CAP_PROP_CONTRAST	✓	Contrast setting for camera (when supported)
cv::CAP_PROP_SATURATION	✓	Saturation setting for camera (when supported)
cv::CAP_PROP_HUE	✓	Hue setting for camera (when supported)
cv::CAP_PROP_GAIN	✓	Gain setting for camera (when supported)
cv::CAP_PROP_EXPOSURE	✓	Exposure setting for camera (when supported)
cv::CAP_PROP_CONVERT_RGB	✓	If nonzero, captured images will be converted to have three channels
cv::CAP_PROP_WHITE_BALANCE	✓	White balance setting for camera (when supported)
cv::CAP_PROP_RECTIFICATION	✓	Rectification flag for stereo cameras (DC1394-2.x only)

GUI Basics

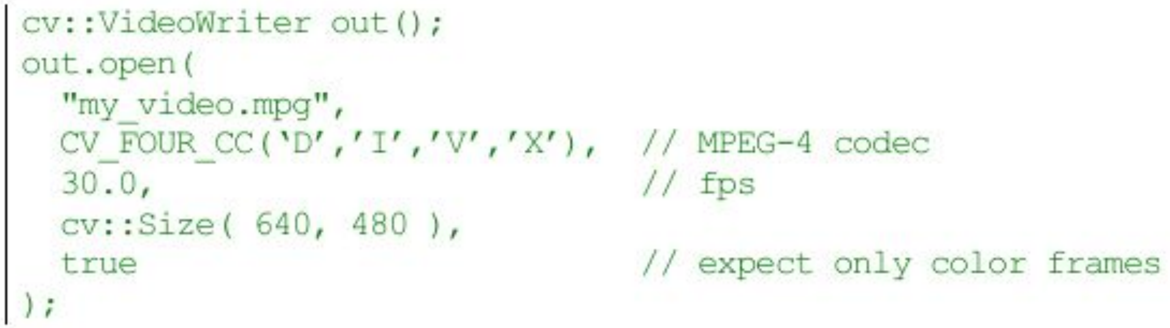
- All of these values are returned as type double, except for the case of FOURCC where we have to type cast to string to able to read.
- Can set all these properties, but the programmer should decide which property will make sense.

GUI Basics

❖ **Writing Video with the cv::VideoWriter Object :**



```
cv::VideoWriter::VideoWriter(
  const string& filename,              // Input filename
  int           fourcc,                // codec, use CV_FOUR_CC() macro
  double        fps,                   // Frame rate (stored in output file)
  cv::Size      frame_size,            // Size of individual images
  bool          is_color  = true       // if false, you can pass gray frames
);
```



```
cv::VideoWriter out();
out.open(
  "my_video.mpg",
  CV_FOUR_CC('D','I','V','X'),  // MPEG-4 codec
  30.0,                         // fps
  cv::Size( 640, 480 ),
  true                          // expect only color frames
);
```

- cv::VideoWriter::isOpened() method, which will return true if you are good to go.
- Write Frames with cv::VideoWriter::write() or with cv::VideoWriter::operator<<()
- You should supply the same type images that you mentioned in the object constructor.

GUI Basics

- **Other Important GUI functions :**
  - namedWindow
  - imshow
  - waitkey()
  - createTrackbar
  - mouseTrackbar functions

References

- Any function and its description can be found at OpenCV reference manual:
  - http://docs.opencv.org/opencv2refman.pdf
- Many OpenCV Tutorials:
  - http://docs.opencv.org/doc/tutorials/tutorials.html
- Learning OpenCV by Gary Bradski 2nd Edition - Pre-release version
- Practical OpenCV by Brahmbhatt
- OpenCV : Computer Vision Application programming Cookbook by Robert Laganiere

- Resources and ebooks are available at my google drive link

  https://drive.google.com/folderview?id=0B3zLbNLqKZ-afk5Jc2pjcEZLRDBGT1d6R2xlTW9MVFFzYXg3aXA4NG5DWHhsSlVwTGw5OVk&usp=sharing

THANK YOU