OpenCV for Image Processing

Part-2

By,

R.Pratesh Kumar Reddy

MT2014519

Contents

General Image Filtering

- Filters are the most basic operations that can executed on images to extract information.
- A filter is any algorithm that starts with some image **I(x,y)** and computes a new image **I'(x,y)** by computing for each pixel location **(x,y)** in **I'** , some function of the pixels in **I** that are in some area around that point.
- The template that defines both this area's shape, as well as how the elements of that area are combined, is called a filter or a kernel.
- The size of the kernel is called 'support' of the kernel.
- We will mostly consider linear kernels. (??!)
- Filtering using linear kernels is also called as Convolution on image.
- Graphical representation of Kernel

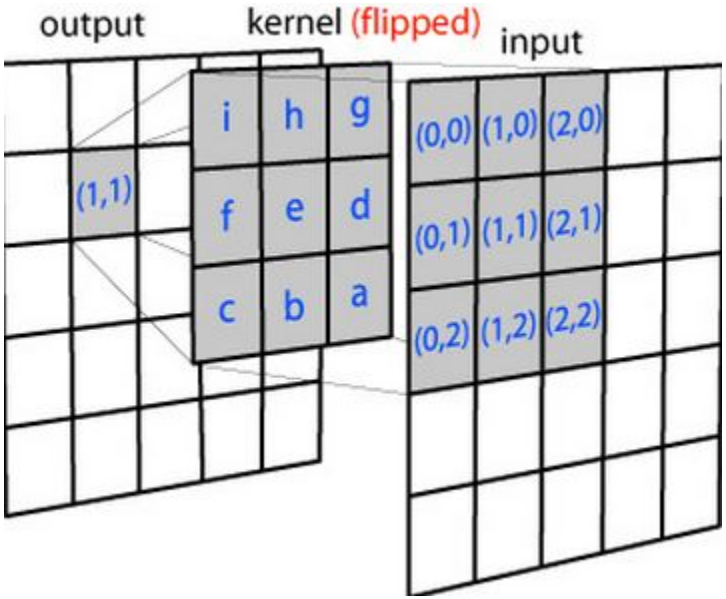
-1	0	1
-2	**0**	2
-1	0	1

$^{1}/_{273}$ x

1	4	7	4	1
4	16	26	16	4
7	26	**41**	26	7
4	16	26	16	4
1	4	7	4	1

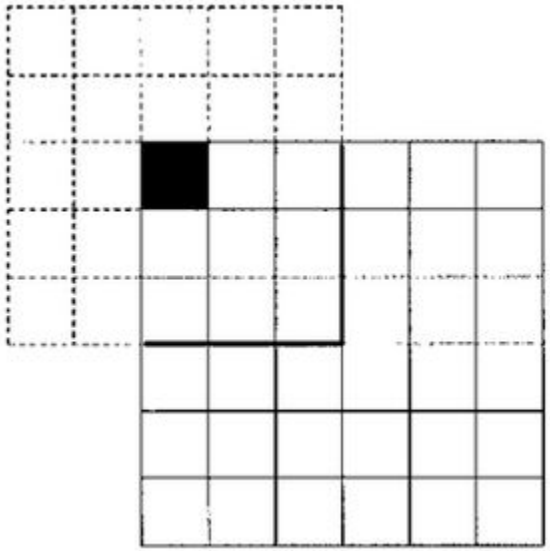
General Image Filtering

- **Anchor Points**: The bold underlined numbers in Kernels. This indicates how the kernel is to be aligned with the source image.



Border Interpolation

- Filtering the border pixels will require the pixels that are outside the image, which in reality does not exist.
- During filtering operations, OpenCV creates the output images of the same size as the input. (Unlike MATLAB)
- To achieve that result, OpenCV creates “virtual” pixels outside of the image at the borders.
- This is also called as **padding** of the pixels.
- Which means, most of the library functions you will use will create these virtual pixels for you.
- In that context, you will only need to tell the particular function how you would like those pixels created.
- To tell that, we use pre-defined macros called **borderTypes.**

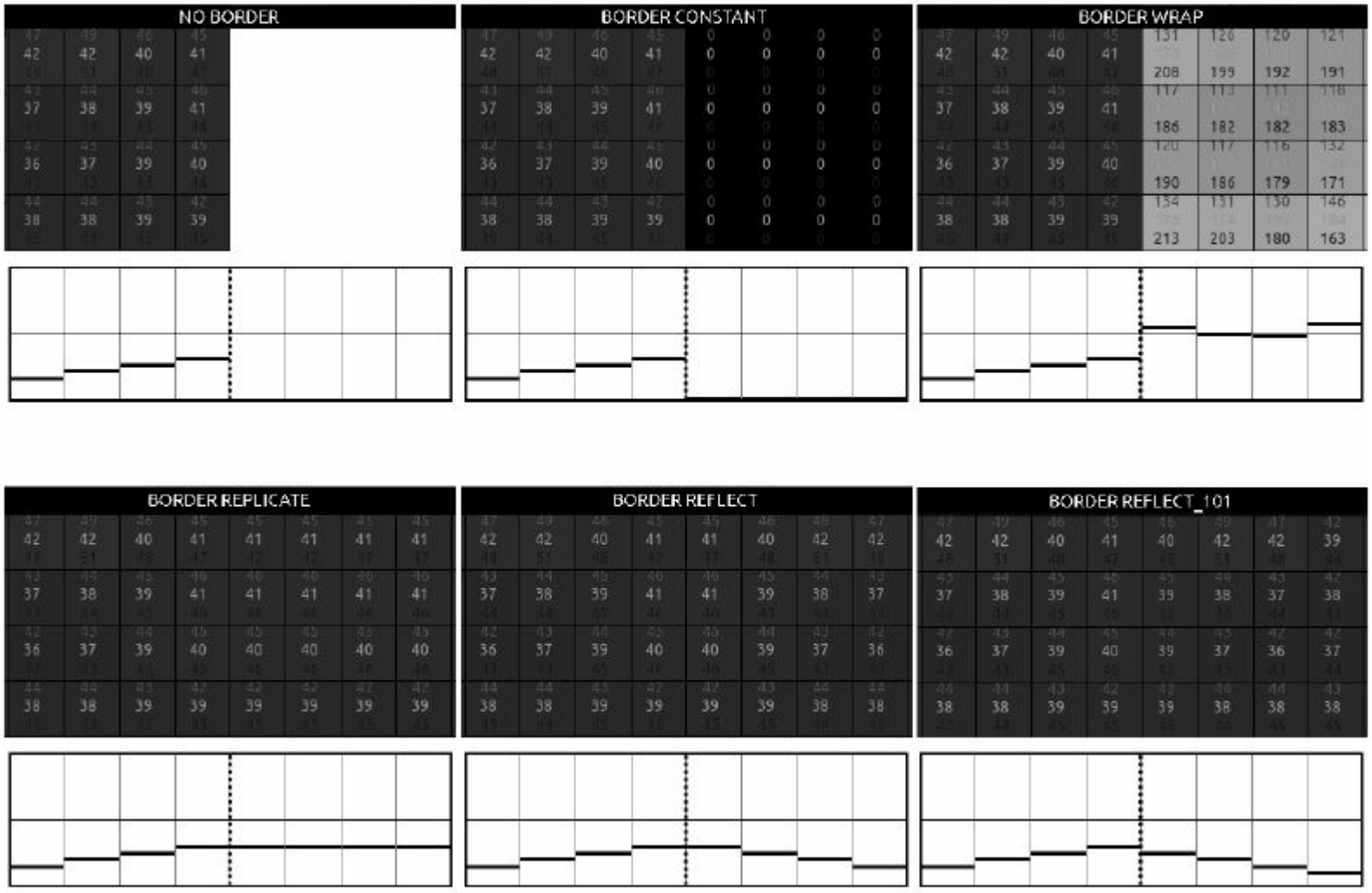


Border Interpolation

- Border Types: OpenCV supports five such border types. First image is just for comparison.

Border Interpolation

- Zoomed-in(at border) version of each border type:



Border Interpolation

- cv::BORDER_CONSTANT sets all of the pixels in the border region to some fixed value. This value can be cv::Scalar(0,0,0) where 3 values are the pixel intensities you would like to pad to each (R,G,B) channel.
- cv::BORDER_REFLECT_101 is similar to cv::BORDER_REFLECT except that the very edge pixel is not replicated in the former.
- cv::BORDER_REFLECT_101 is the default method followed by any method that requires this padding.
- So, there is one more alias for cv::BORDER_REFLECT_101 called cv::BORDER_DEFAULT.
- We can make our own padded borders using cv::copyMakeBorder()

```
void cv::copyMakeBorder(
  cv::InputArray    src,                    // Input Image
  cv::OutputArray   dst,                    // Result image
  int               top,                    // Top side padding (pixels)
  int               bottom,                 // Bottom side padding (pixels)
  int               left,                   // Left side padding (pixels)
  int               right,                  // Right side padding (pixels)
  int               borderType,             // Pixel extrapolation method
  const cv::Scalar& value = cv::Scalar()    // Used for constant borders
);
```

Using an Arbitrary Linear Filter

- For linear filters, OpenCV provided a seperate function to handle convolution **cv::filter2D().**

```
cv::filter2D(
  cv::InputArray  src,                    // Input Image
  cv::OutputArray dst,                    // Result image
  int             ddepth,                 // Pixel depth of output image (e.g., cv::U8)
  cv::InputArray  kernel,                 // Your own kernel
  cv::Point       anchor     = cv::Point(-1,-1),  // Location of anchor point
  double          delta = 0,                      // Offset before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

- This will allow you to take advantage of the internal optimizations of OpenCV.
- ddepth - if it is negative, it will be the same as src.depth(). The following combinations of src.depth() and ddepth are supported. This restriction is to avoid overflow.
  - src.depth() = CV_8U, ddepth = -1/CV_16S/CV_32F/CV_64F
  - src.depth() = CV_16U/CV_16S, ddepth = -1/CV_32F/CV_64F
  - src.depth() = CV_32F, ddepth = -1/CV_32F/CV_64F
  - src.depth() = CV_64F, ddepth = -1/CV_64F

Using an Arbitrary Linear Filter

- Delta is the offset you'll assign to each pixel after convolution.
- If you want to apply different kernels to different channels, split the image into separate color planes using **cv:split()** and process them individually.
- cv::split() will copy the three channels of a color image into three distinct cv::Mat instances.

```
// create vector of 3 images
std::vector<cv::Mat> planes;
// split 1 3-channel image into 3 1-channel images
cv::split(image1,planes);
```

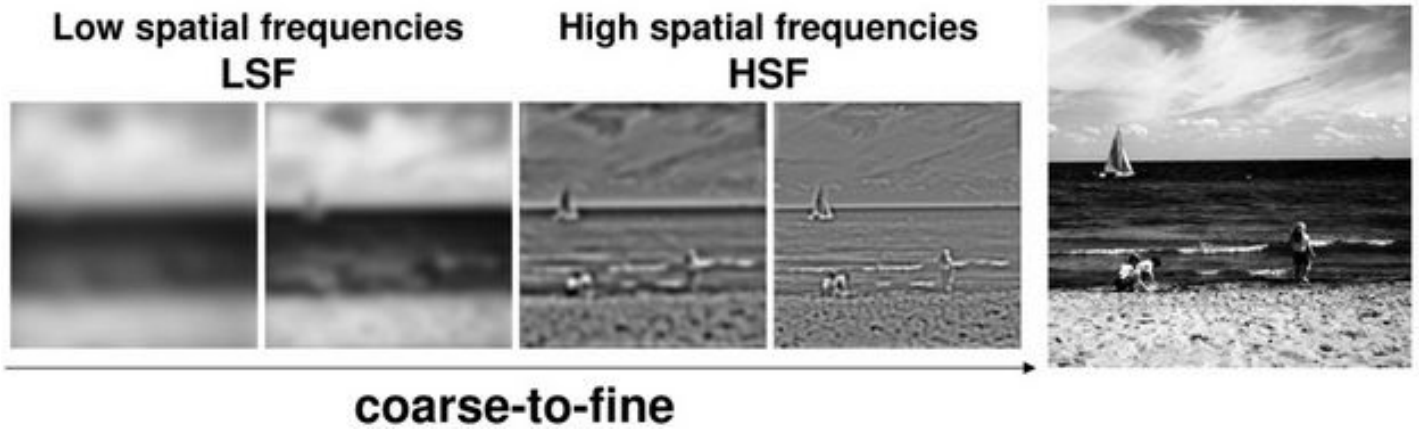
- Here, plane[0] will contain the blue channel (**REM: OpenCV stores in BGR order by default**) and so on...
- Once the seperate processing is done, we can merge them using **cv::merge().**

```
// merge the 3 1-channel images into 1 3-channel image
cv::merge(planes,result);
```

Image Blurring

Image Blurring

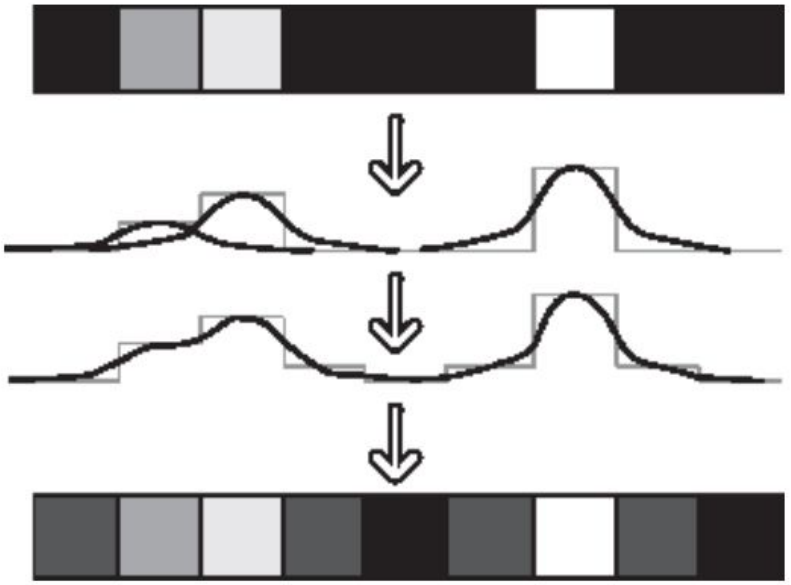
- Images have various “frequency components” along both of its axes’ directions.
- An image will have more higher frequencies if pixel intensity values in it change considerably in short pixel distances.
- For e.g. Edges have high frequencies. So a vertical edge creates high frequency components along the horizontal axis of the image and vice versa.



- Finely textured regions also have high frequencies.
- Whereas, slowly changing intensity values have low frequencies.

Image Blurring

- Image blurring is like a low pass filtering. Also called as **Image Smoothing.**
- Blurring will convert the abrupt changes in pixel intensities to gradual changes. Thus reduces the high-freq components in images.



- Blurring is also an important post processing step when you want to increase or decrease the size of an image. (why? .. will know later)

Image Blurring

- Image Blurring can be accomplished by replacing each pixel in the image with some sort of average of the pixels in a region around it.
- This a simple box kernel. This kernel deems every pixel equally important.

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

- A better kernel that gives more importance to pixels near the center is Gaussian Kernel

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

- We have to normalize the kernel by dividing them all with 256 for Gaussian Kernel.

Image Blurring- Box Filtering

- OpenCV also gives you the higher level function for simple box filtering blur().
- Each pixel in the output is the simple mean of all of the pixels in a window.

```
void cv::blur(
  cv::InputArray  src,                          // Input Image
  cv::OutputArray dst,                          // Result image
  cv::Size        ksize,                        // Kernel size
  cv::Point       anchor     = cv::Point(-1,-1),  // Location of anchor point
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

- In the case of multichannel images, each channel will be computed separately.
- Anchor can be used to specify how the kernel is aligned with the pixel being computed.
- Anchor is a point type. If its values are negative (like -1,-1), it means anchor is at the center of the image. If it's (0,0) it means the anchor is at top left corner so on..
- This is a normalised box filter.
- If you want to choose between normalized and un-normalised box filter then use **cv::boxFilter().**

Image Blurring- Median Filtering

- The median filter replaces each pixel by the median or “middle-valued” pixel (as opposed to the mean pixel).
- Useful in removing shot noise and salt-pepper noise.

- There is OpenCV function

cv::medianBlur()

```
void cv::medianBlur(
  cv::InputArray  src,          // Input Image
  cv::OutputArray dst,          // Result image
  cv::Size        ksize         // Kernel size
);
```

Image Blurring- Gaussian Blur

- One can create different sizes of the Gaussian kernel by using the OpenCV function **getGaussianKernel()**

  Mat **getGaussianKernel**(int **ksize**, double **sigma**, int **ktype**=CV_64F)

- OpenCV also gives you the higher level function GaussianBlur().

```
void cv::GaussianBlur(
  cv::InputArray  src,                    // Input Image
  cv::OutputArray dst,                    // Result image
  cv::Size        ksize,                  // Kernel size
  double          sigmaX,                 // Gaussian half-width in x-direction
  double          sigmaY      = 0.0,      // Gaussian half-width in y-direction
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

- ksize members must be positive and odd. Or, they can be zero's and then they are computed from sigma = 0.3*((ksize-1)*0.5 - 1) + 0.8 .
- sigmaX and sigmaY are the standard deviations in X and Y directions.
- If sigmaY is zero, it is set to be equal to sigmaX.
- If both the sigmas are zeroes then they are computed from ksize.width and height.

Image Blurring- Bilateral Filter

- Gaussian Blurring even smooths the edges which is not desired.
- The problem is with the typical motivation for Gaussian smoothing in a real life image.
- Here comes the Bilateral Filter which smooths everywhere except at edges.
- It's also does the weighted average of pixels but its weighted average has two components.
- the first one is normal weighting while the second component has 0 to 1 value based on the difference in intensity from the center pixel.
- Multiple iterations of bilateral filtering will turn the image into what appears to be a **watercolor painting** of the same scene.

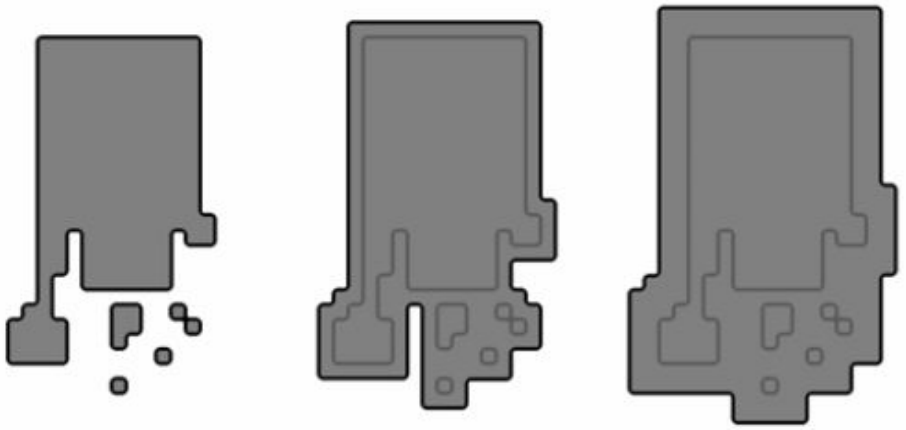
Image Blurring- Bilateral Filter

- Program to interactively blur an image using a Gaussian kernel of varying sizes (or supports)
- Lisitng 5.2 in the given source codes

Morphological Processes

Explanation

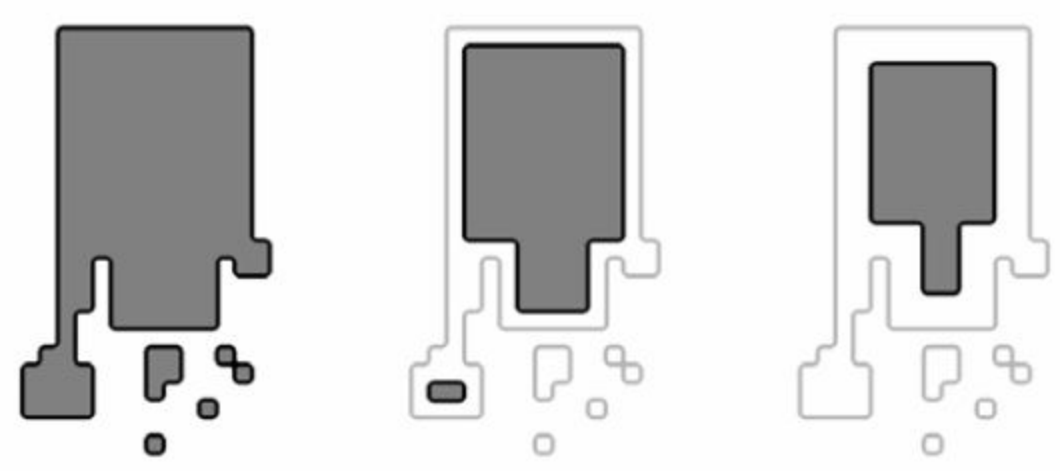
- Dilation and Erosion are the two basic morphological transformations.
- **Erosion** is defined as the **minimum** of all element-wise multiplications between kernel elements and pixels falling under the kernel. This will cause the black areas in the image to “encroach” into the white areas.



- They arise in a wide variety of contexts such as removing noise, isolating individual elements, and joining disparate elements in an image.
- The depth of erosion and dilation is decided by the size of the kernel.

Explanation

- **Dilation** is defined as the **maximum** of all element-wise multiplications between kernel elements and pixels falling under the kernel. This will cause the white areas in the image to “encroach” into the black areas.



- This is an example of a nonlinear filtering, so the kernel cannot be expressed as weighted sum.
- The kernel used in Morphological Processes is called as **Structuring Element.**
- This **SM** used can be a solid square, a cross or of ellipse shape.

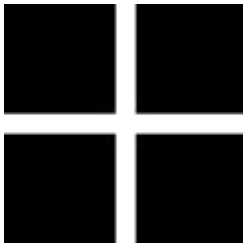
Making your own Str.Element

- Structuring elements can be created by **cv::getStructuringElement()** function.

```
cv::Mat cv::getStructuringElement(
  int       shape,                    // Element shape, e.g. cv::MORPH_RECT
  cv::Size  ksize,                    // Size of structuring element (should be odd)
  cv::Point anchor = cv::Point(-1,-1)  // Location of anchor point
);
```

- The argument shape controls the basic shape of the element . It takes three types of i/p

cv::MORPH_RECT **cv::MORPH_ELLIPSE** **cv::MORPH_CROSS**



- These above SMs are binary images and of type cv::Mat.
- You can simply pass any **cv::Mat** to the morphological operators **as a structuring element** if you need something more complicated than the basic shape-based elements created by cv::getStructuringElement()

Erosion and Dilation Functions

- OpenCV functions for erosion and dilation are cv::erode() and cv::dilate()

```
void cv::erode(
  cv::InputArray    src,                 // Input Image
  cv::OutputArray   dst,                 // Result image
  cv::InputArray    element,             // Structuring element, can be cv::Mat()
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  int               iterations = 1,                 // Number of times to apply
  int               borderType = cv::BORDER_DEFAULT // Border extrapolation to use

    const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
  );
```

- Can even pass an uninitialized array cv::Mat(), which will cause it to default to using a 3-by-3 kernel with the anchor at its center.

```
void cv::dilate(
  cv::InputArray    src,                 // Input Image
  cv::OutputArray   dst,                 // Result image
  cv::InputArray    element,             // Structuring element, can be cv::Mat()
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  int               iterations = 1,                 // Number of times to apply
  int               borderType = cv::BORDER_CONSTANT // Border extrapolation to use
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
);
```

General Morphology Function

- There are other morphological opertions like opening, closing, Tophat, Blackhat, Morphological Gradient.
- To do these operations we have a general Morphology Function **cv::morphologyEx()**

```
void cv::morphologyEx(
  cv::InputArray    src,                // Input Image
  cv::OutputArray   dst,                // Result image
  int               op,                 // Morphology operator to use e.g. cv::MOP_OPEN
  cv::InputArray    element,            // Structuring element, can be cv::Mat()
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  int               iterations = 1,                 // Number of times to apply
  int               borderType = cv::BORDER_DEFAULT // Border extrapolation to use

  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()

);
```

- **morphologyDefaulBorderValue()** gives +Inf for the erosion or -Inf for the dilation to outside border pixels. This means that the minimum (maximum) is effectively computed only over the pixels that are inside the image.

General Morphology Function

- **op** argument is used to select the desired Morph. operation

Value of operation	Morphological operator
cv::MOP_OPEN	Opening
cv::MOP_CLOSE	Closing
cv::MOP_GRADIENT	Morphological gradient
cv::MOP_TOPHAT	Top Hat
cv::MOP_BLACKHAT	Black Hat

- Know about each of these morphological operations from DIP by Gonzalez
- **Program to examine erosion and dilation of images. Lisiting 5.3 from source code**

Derivatives and Gradients

Concept

- In1D derivative, First order derivative is forward difference (look-ahead) implementation.

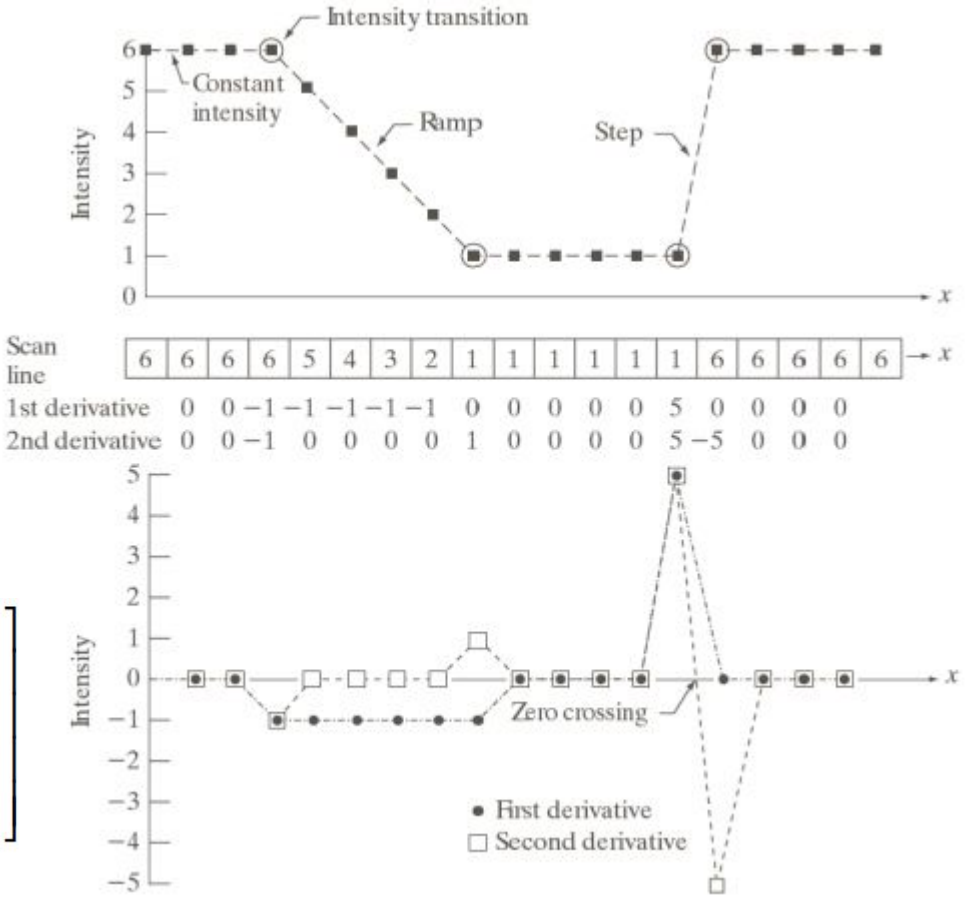
$$\frac{\partial f}{\partial x} = f(x+1) - f(x)$$

- Proceed furthur for the second order derivative.
- It's like we have a 1-D kernel like this

[1,-1]

- The gradient is nothing but the vector that contains derivatives in both directions.
- The gradient has magnitude and direction

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$



Evolution of Derivatives

- The simple gradient [1,-1] or [-1,1] has a problem : when you use it, you compute the gradient between two positions and not at one position. Infact, what you have computed is the gradient at the middle point of both positions.
- That's why we had a zero in the middle: [-1 0 1]

  Applying this one to pixels (x-1,y), (x,y) and (x+1,y) will clearly give you a gradient for the center pixel (x,y).
- But [-1,0,1] is sensitive to noise because it's dependent on only left and right pixels on the same row of image.
- That's why we decide not to apply it on a single row of pixels, but on 3 rows: this allows to get an average gradient on these 3 rows, that will soften possible noise:

```
[-1 0 1]
[-1 0 1]
[-1 0 1]
```

- But we want to give a little more weight to the center row so as to preserve the specificity of that very row :

```
[-1 0 1]
[-2 0 2]
[-1 0 1]
```

- This leads to sobel filter :

Sobel Derivative

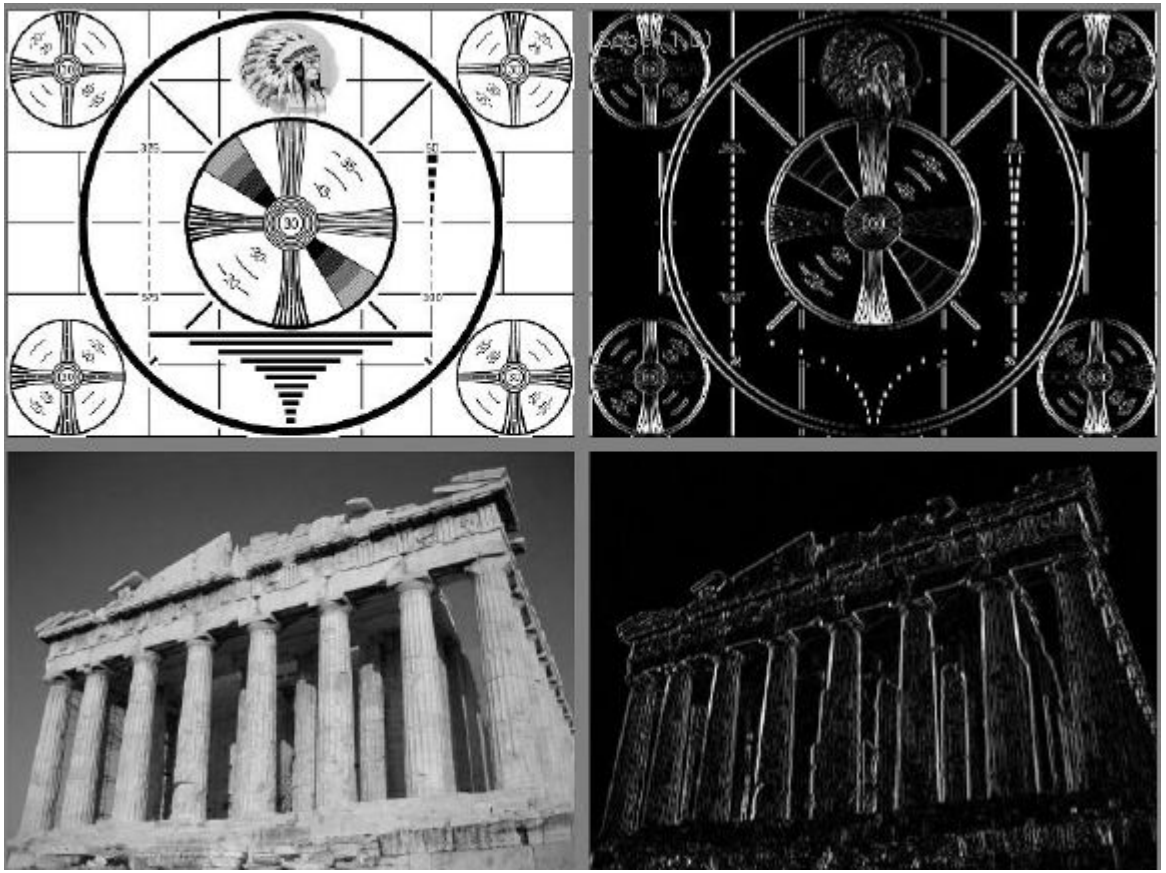
- **cv::Sobel()** is used for soble derivative

```
void cv::Sobel(
  cv::InputArray  src,                    // Input Image
  cv::OutputArray dst,                    // Result image
  int             ddepth,                 // Pixel depth of output image (e.g., cv::U8)
  int             xorder,                 // order of corresponding derivative in x
  int             yorder,                 // order of corresponding derivative in y
  cv::Size        ksize = 3,              // Kernel size
  double          scale = 1,              // Scale applied before assignment to dst
  double          delta = 0,              // Offset applied before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

- xorder and yorder are the orders of the derivative in resp. directions. Typically, you'll use 0, 1, or at most 2; a 0 value indicates no derivative in that direction. Both xorder and yorder cannot be zeros at a time.
- scale and delta can be useful when you want to actually visualize a derivative in an 8-bit image on the screen.

Sobel Derivative

- Results of Sobel operator when used to approximate a first derivative in the x-dimension



- Larger Sobel kernels are more robust to noise than the smaller ones.

Scharr Filter

- Smaller Soble operators are less accurate. Mainly if we want to find the image gradients.

  Overall gradient, G = sqrt(Dx^2 + Dy^2)

  Angle of gradient, Φ = arctan(Dy / Dx)

- Scharr Filter comes for the rescue.

For X -direction

-3	0	3
-10	0	10
-3	0	3

For Y-direction

-3	-10	-3
0	0	0
3	10	3

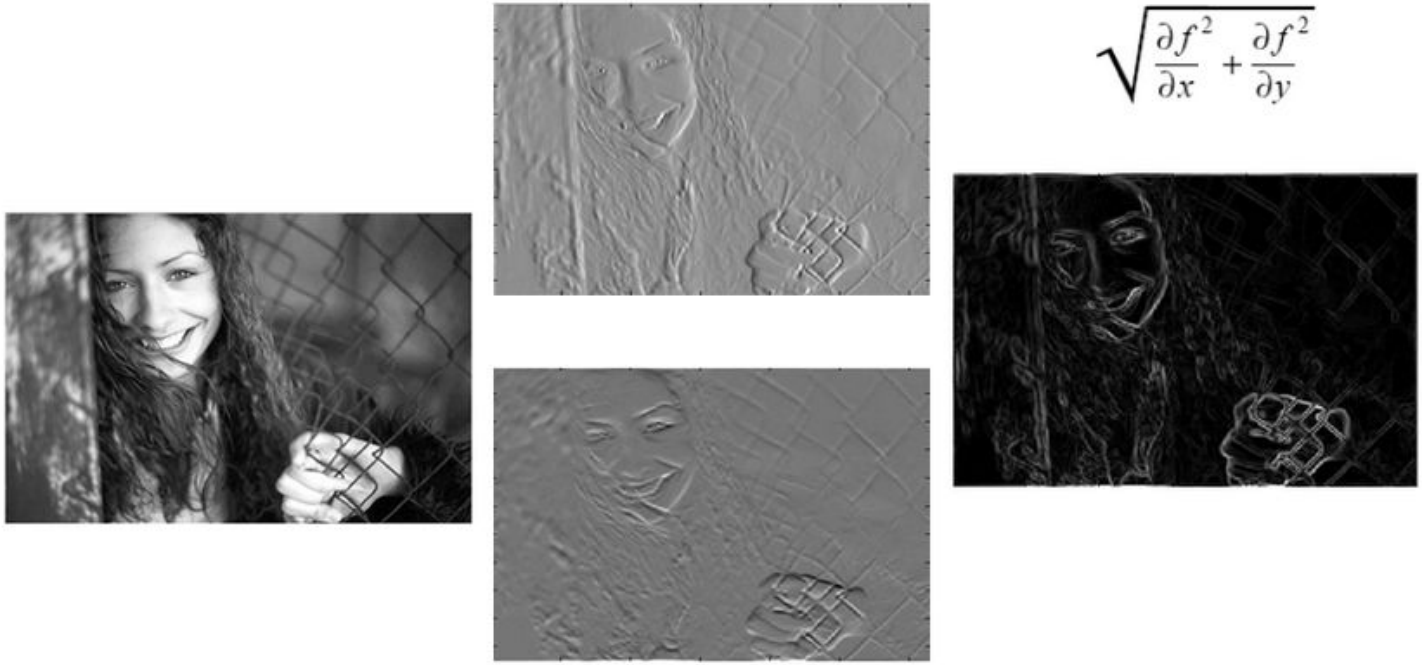
- We have cv::Scharr function

C++: void **Scharr**(InputArray **src**, OutputArray **dst**, int **ddepth**, int **dx**, int **dy**, double **scale**=1, double **delta**=0, int **borderType**=BORDER_DEFAULT)

- dx,dy are order of the derivatives in x and y direction.
- Program to detect edges in an image using the Scharr operator. **Listing 5.4** in codes.

Scharr Filter

- Results : Applying scharr in both directions and plotting the gradient's magnitude



- Why do the derivative images and gradient images look like the way they are ??
- But there is a lot of noise in the edge image. Since it's a 8-bit image
- We can use **cv::threshold()** function to get rid of this noise.

The Laplacian

- The laplacian is second order derivative operator.
- Infact, in Opencv's cv::Laplacian, it uses internally second-order Sobel derivative
- Usage

```
void cv::Laplacian(
  cv::InputArray  src,                  // Input Image
  cv::OutputArray dst,                  // Result image
  int             ddepth,               // Pixel depth of output image (e.g., cv::U8)
  cv::Size        ksize = 3,            // Kernel size
  double          scale = 1,            // Scale applied before assignment to dst
  double          delta = 0,            // Offset applied before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

- If ksize=1 then it uses the following kernel
- Otherwise, it uses the second-order sobel

0	1	0
1	-4	1
0	1	0

Canny Edge Detector

- Canny Edge detector is better in terms of detecting only the existent edges.
- Please refer

http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html

for steps involved in Canny edge detector.

- Opecv function is cv::Canny()

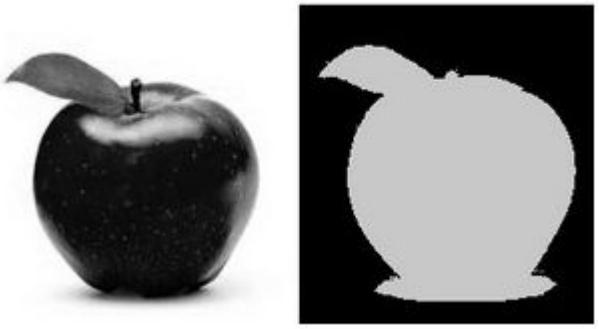
C++: void **Canny**(InputArray **image**, OutputArray **edges**, double **threshold1**, double **threshold2**, int **apertureSize**=3, bool **L2gradient**=false)

- edges – output edge map; it has the same size and type as image .
- threshold1 and 2 – first and second thresholds for the hysteresis procedure.
- L2gradient – a flag indicating the type of norm either 2nd or 1st.
- aperture is same as kernel size.

Threshold Operations

Thresholding

- It's the simplest segmentation process.
- Differentiate the pixels, we are interested in, from the rest.
- For this we perform a comparison of each pixel intensity value with respect to a threshold.
- Once we have our desired pixels we can set their intensity values to desired values so as to identify them.



- For e.g. in the above picture, the apple is segmented and its pixels are given gray (125) color values.

Normal OpenCV Thresholding

- OpenCV function cv::threshold() accomplishes these tasks

```
double cv::threshold(
  cv::InputArray    src,                    // Input Image
  cv::OutputArray   dst,                    // Result image
  double            thresh,                 // Threshold value
  double            maxValue,               // Max value for upward operations
  int               thresholdType           // Threshold type to use
);
```

- src – input array (single-channel, 8-bit or 32-bit floating point)
- dst – Destination image of the same size and the same type as src
- maxValue – Non-zero value assigned to the pixels for which the condition is satisfied

Types of Thresholding

- The argument 'thresholdType' will take macros that define the type of thresholding method.
- OpenCV supports five types of thresholding methods.
- They are
  - Threshold Binary
  - Threshold Binary, Inverted
  - Truncate
  - Threshold to Zero
  - Threshold to Zero, Inverted
- The macros are (in the same order) >>>

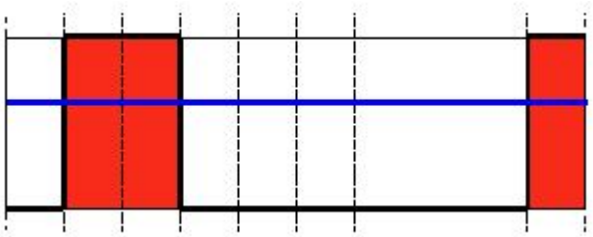
Threshold type
cv::THRESH_BINARY
cv::THRESH_BINARY_INV
cv::THRESH_TRUNC
cv::THRESH_TOZERO
cv::THRESH_TOZERO_INV

Types of Thresholding

- Threshold binary:
  - The operation is defiend as

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

  - So, if the intensity of the pixel src(x,y) is higher than thresh, then the new pixel intensity is set to a MaxVal. Otherwise, the pixels are set to 0.

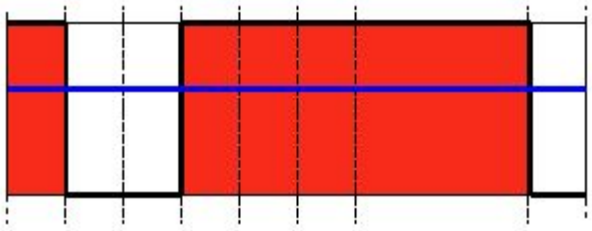


- Threshold Binary, Inverted:
  - operation can be expressed as

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxVal} & \text{otherwise} \end{cases}$$

Types of Thresholding

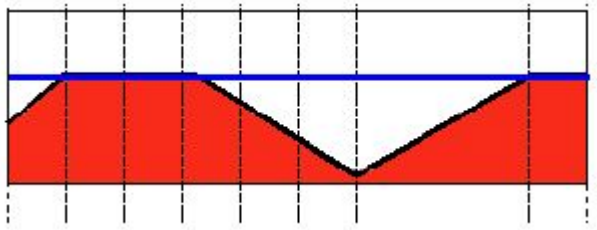
- If the intensity of the pixel src(x,y) is higher than thresh, then the new pixel intensity is set to a 0. Otherwise, it is set to MaxVal.



- Truncate:
  - operation can be expressed as

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

  - The maximum intensity value for the pixels is thresh, if src(x,y) is greater, then its value is truncated.

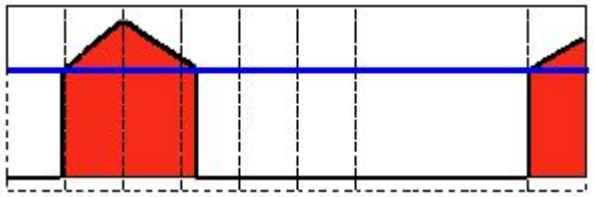


Types of Thresholding

- Threshold to Zero:
  - The operation is defiend as

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

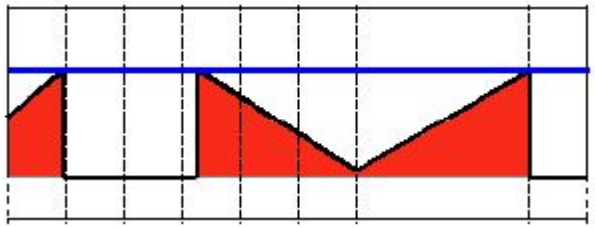
  - If src(x,y) is lower than thresh, the new pixel value will be set to 0.



- Threshold Binary, Inverted:
  - operation can be expressed as

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

  - If src(x,y) is greater than thresh, the new pixel value will be set to 0.



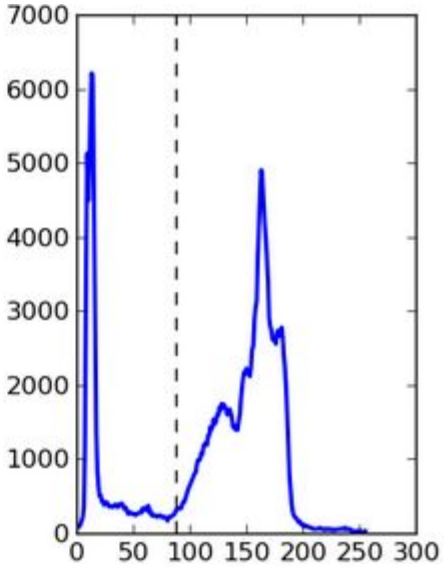
Thresholding Code

- threshold_op.cpp >>> code will be sent to you by email
- One of the main idea is that we don't want to add directly into three 8-bit arrays (with the idea of normalizing next) because the higher bits will overflow.
- Instead, we use equally weighted addition of the three color channels (cv::addWeighted());
- See the result of each type of threholding at the link:

  http://www.learnopencv.com/opencv-threshold-python-cpp/

Thresholding is related to histogram

- Based on **histogram** of the image we can threshold the image into different object clases
- For e.g the image is thresholded (segmented) into two classes ==> object and foreground.



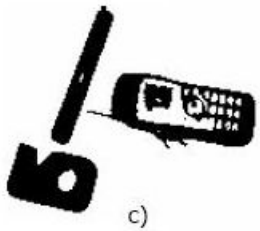
- The above histogram is called bi-modal histogram because it is divided into two classes of pixels.

Otsu's Algorithm

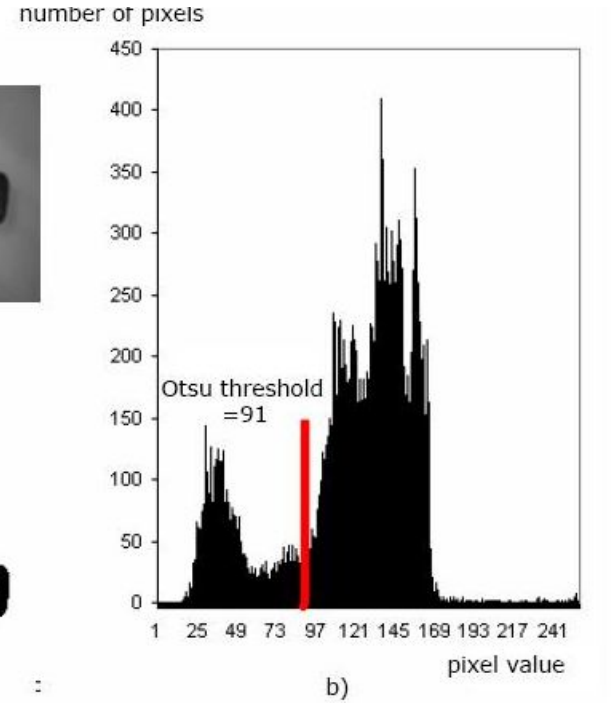
- Otsu's method uses this relationship of image thresholding with its histogram.
- The algorithm assumes that the image contains two classes of pixels following bi-modal histogram (foreground pixels and background pixels)
- It calculates the optimum threshold separating the two classes so that their combined spread (intra-class variance) is minimal.

$$\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$

a)



c)



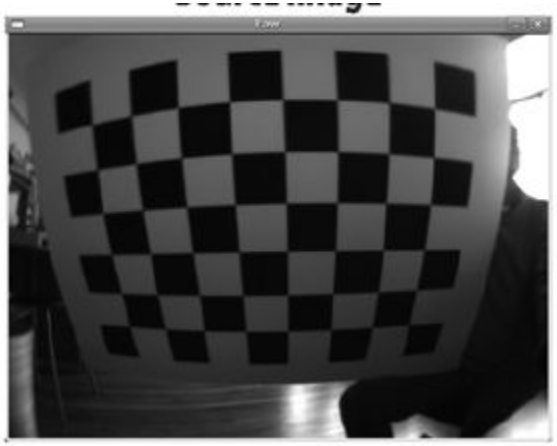
b)

Otsu’s Algorithm in OpenCV

- Have to use the same cv::threshold() function.
- But now the '**thresh**' argument will not be a number.
- It should be a special value **cv::THRESH_OTSU**.

Adaptive Thresholding

- In this, threshold level is itself variable (across the image)
- The adaptive threshold T(x, y) is set on a pixel-by-pixel basis.
- T(x,y) is computed by a weighted average of the b-by-b region around each pixel location minus a constant.
- The adaptive threshold technique is useful when there are strong illumination or reflectance gradients that you need to threshold relative to the general intensity gradient.



Adaptive Thresholding in OpenCV

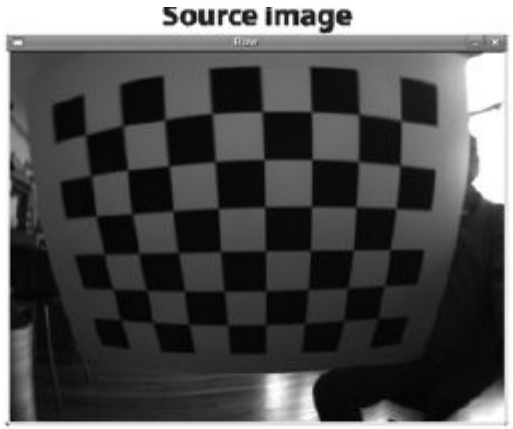
- This method is implemented in the cv::adaptiveThreshold() function.
- Usage:

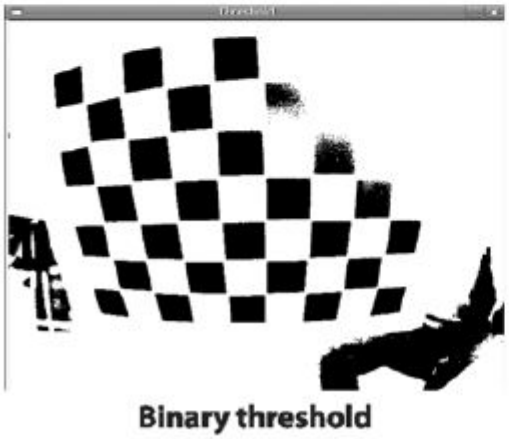
```
void cv::adaptiveThreshold(
  cv::InputArray    src,                  // Input Image
  cv::OutputArray   dst,                  // Result image
  double            maxValue,             // Max value for upward operations
  int               adaptiveMethod,       // Method to weight pixels in block by
  int               thresholdType         // Threshold type to use
  int               blockSize,            // Block size
  double            C                     // Constant to offset sum over block by
);
```

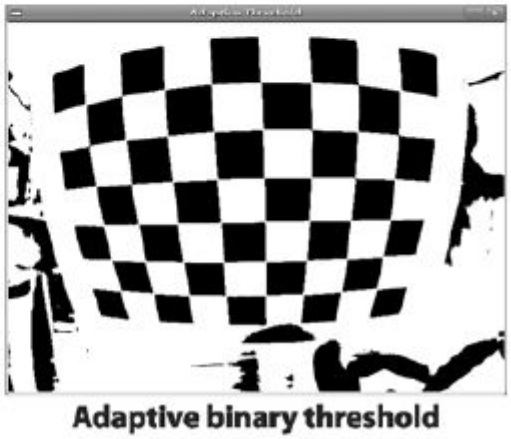
- It allows for two different adaptive threshold types depending on the settings of **adaptiveMethod**.
- If it's set to cv::ADAPTIVE_THRESH_MEAN_C, then all pixels in the area are weighted equally.
- If it is set to cv::ADAPTIVE_THRESH_GAUSSIAN_C, then the pixels in the region around (x,y) are weighted according to a Gaussian function of their distance from that center point.

Thresholding for noise removal

- Combining threholding and canny edge derrtction .. code from bhramm bhatt







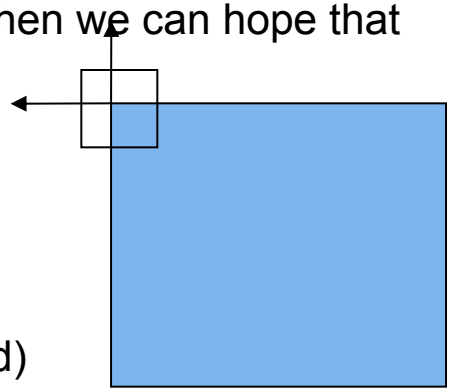
Thresholding for noise removal

- Combining thresholding and Scharr Operator.. Listing 5-5 from source codes.
- One can threshold it by a number between 0 and 255 to remove the noise.

Corners

Corner Detection

- Corners—not edges—are the points that contain enough information to be picked out from one frame to the next. (if you think of object tracking in a video)
- If strong derivatives are observed in two orthogonal directions then we can hope that this point is more likely to be a corner.



- OpenCV uses two methods for corner detection

  -- Harris' Method

  -- Minimum Eigen Value Method (Shi and Tomasi Method)

- Both the methods are based on "autocorrelation matrix of the second derivative images".
- Second order derivative (SOD) image :: Obtained by using Sobel Operators on main image for two times.
- Autocorrelation matrix of this SOD image :: Obtained by taking a small window of SOD image and using the following formula -

$$M(x,y)=\begin{bmatrix} \sum_{-K\le i,j\le K} w_{i,j} I_x^2(x+i,y+j) & \sum_{-K\le i,j\le K} w_{i,j} I_x(x+i,y+j) I_y(x+i,y+j) \\ \sum_{-K\le i,j\le K} w_{i,j} I_x(x+i,y+j) I_y(x+i,y+j) & \sum_{-K\le i,j\le K} w_{i,j} I_y^2(x+i,y+j) \end{bmatrix}$$

Corner Detection

- **Remember** : The kernel size of sobel operator used to obtain SOD image can be different from the kernel size used to get the autocorrelation matrix using SOD image.
- After this step, the methods diverge in their approach of calculating the **corner-quality** (how strong the corner is) ::

  --- Harris Method uses this formula to get corner-quality for each pixel

  $$\mathtt{dst}(x,y) = \det M^{(x,y)} - k \cdot \left(\mathrm{tr} M^{(x,y)}\right)^2$$

  --- Min.Eig.Val method finds eigenvalues of M at each pixel and then gets corner-quality by

  $$\mathtt{dst}(x,y) = \text{ min (eval1,eval2)}$$

  * eval1,eval2 are eigen values

- Both the methods are based on the observation that " if these two eigenvalues are low, we are in a relatively homogenous region. If one eigenvalue is high and the other is low, we must be on an edge. If both eigenvalues are high, then we are at a corner location."

Corner Detection in OpenCV

- OpenCV function **cv::goodFeaturesToTrack()** implements both of these corner detectors.
- Usage:

C++: void **goodFeaturesToTrack**(InputArray **image**, OutputArray **corners**, int **maxCorners**, double **qualityLevel**, double **minDistance**, InputArray **mask**=noArray(), int **blockSize**=3, bool **useHarrisDetector**=false, double **k**=0.04)

- image – Input 8-bit or floating-point 32-bit, **single-channel** image.

  corners – Output **vector** of detected corners.

  maxCorners – Maximum number of corners to return. If there are more corners than are found, the strongest of them is returned.

  qualityLevel - Parameter characterizing the **minimal accepted quality** of image corners.

  minDistance – Minimum possible Euclidean distance between the returned corners.

  mask - is a usual image, interpreted as Boolean values, indicating which points should and which points should not be considered as possible corners

Corner Detection in OpenCV

blockSize – Size of an average block **for computing a derivative covariation** matrix over each pixel neighborhood

useHarrisDetector – TRUE if you want to use a Harris detector, FALSE if you want to use Min.Eig.Val detector.

k – Free parameter of the Harris detector. useful only when

--- More about **qualityLevel : (always < 1)**

qualityLevel is useful to filter out weaker corners or fake corners.

Out of the entire **dst image** , we findout the highest corner-quality. let it be some 1500.

Assume that **qualityLevel = 0.01**

Then we multiply these two to get 1500 * 0.01 = 15

This 15 acts as the threshold corner-quality.

All the pixels in dst image that have corner-qualty < 15 are rejected.

Corner Detection in OpenCV

The rejection of corners doesn't jusst stop there.

--- The function will now use **minDistance** parameter.

For each corner location, it checks if there is any other corner that lies within this **minDistance.** Then only the corner with more corner-quality will survive. In particular, the **minDistance** guarantees that no two returned points are within the indicated distance**.**

- **Listing 5.6 in Source Code** - Program to detect corners in an image and number of corners are controlled by slider.
- More details at

  http://docs.opencv.org/modules/imgproc/doc/feature_detection.html#goodfeaturestotrack

  and in book

  "Learning OpenCV by Bradski" first edition (pages 317-319)

THE END