OpenCV for Image Processing

Part-3

As part of Tech-Seminar,

Under

Prof.Dinesh Babu Jayagopi

By,

R.Pratesh Kumar Reddy

MT2014519

Contents

- General Image Transforms
- Geometric Image Transforms
  - Stretch and Shrink
  - Interpolation Methods
  - Image Pyramids - Gaussian and Laplacian
  - Affine Transforms
  - Perspective Transforms
  - Arbitrary Mapping

General Image Transforms

- All the filtering that we learnt in last sessions was in terms of convolution.
- But not all operations can be expressed as a little window scanning over the image doing one thing or another.
- The image transforms we will look at in this session have the purpose of converting an image into some entirely different representation.
- This different representation will usually still be an array of values, but those values might be **quite different in meaning.**
- For e.g. the array values need not represent the intensity values in the input image.
- E.g. Fourier transform will just represent it in terms of frequncy spectrum of image.
- E.g. Hough Line Transform will contain the list of lines.
- It can be divided into sub-sections like **geometrical transformations, frequency transformations** etc.,

GEOMETRIC IMAGE TRANSFORMS

Geometric Image Transforms

- They do not change the image content but deform the pixel grid and map this deformed grid to the destination image.
- To avoid forward projection artifacts, the mapping is done in the reverse order, from destination to the source. This is called "**reverse mapping**"
- That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding **"donor" pixel** in the source image and copy the pixel value:

$$\mathtt{dst}(x,y) = \mathtt{src}(f_x(x,y), f_y(x,y))$$

- Need to address two problems :

  ==> Extrapolation of non-existing pixels

  ==> Interpolation of pixel values

Geometric Image Transforms

==> Extrapolation of non-existing pixels :

fx(x,y)and fy(x,y) could obtain values that indicate a pixel outside the image boundary. It's similar to the filtering functions (described in the previous session) where we had to add more pixels to the boudaries.

So we can use the same border extrapolation methods as in the filtering functions.

In addition, it provides the method **BORDER_TRANSPARENT** .

==> Interpolation of pixel values

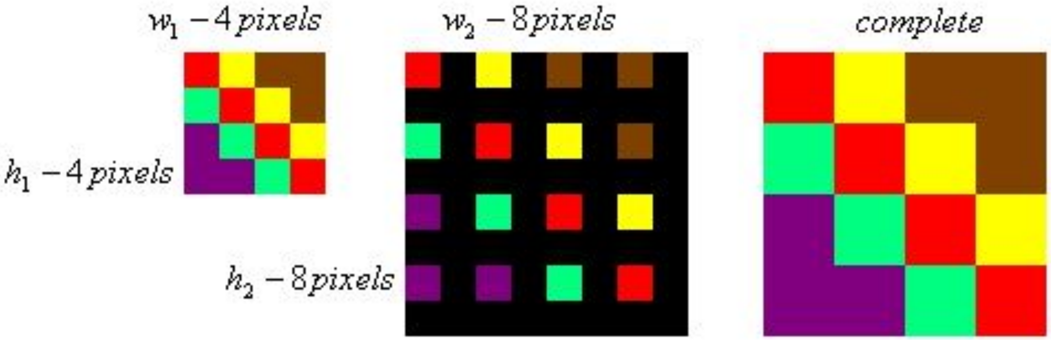
fx(x,y) and fy(x,y) can be floating-point numbers while pixel locations are only integers.

So,to retrieve pixel value at fractional coordinates we can just round to the nearest integer coordinates and the corresponding pixel can be used.

There are many methods for pixel interpolation.

Strech and Shrink

- This collectively is called image resizing.
- They are little less trivial than you might think.
- Because it involves pixels interpolation (for enlargement) or merging (for reduction).



- Function cv::resize() is used for image resizing.

```
void cv::resize(
  cv::InputArray  src,                              // Input Image
  cv::OutputArray dst,                              // Result image
  cv::Size        dsize,                            // New Size
  double          fx            = 0,                // x-rescale
  double          fy            = 0,                // y-rescale
  int             interpolation = cv::INTER_LINEAR // interpolation method
);
```

Strech and Shrink

- Absolute sizing : use dsize
- Relative sizing : use fx,fy
- Either dsize must be cv::Size(0,0) or fx and fy must both be zero.
- Interpolation Options

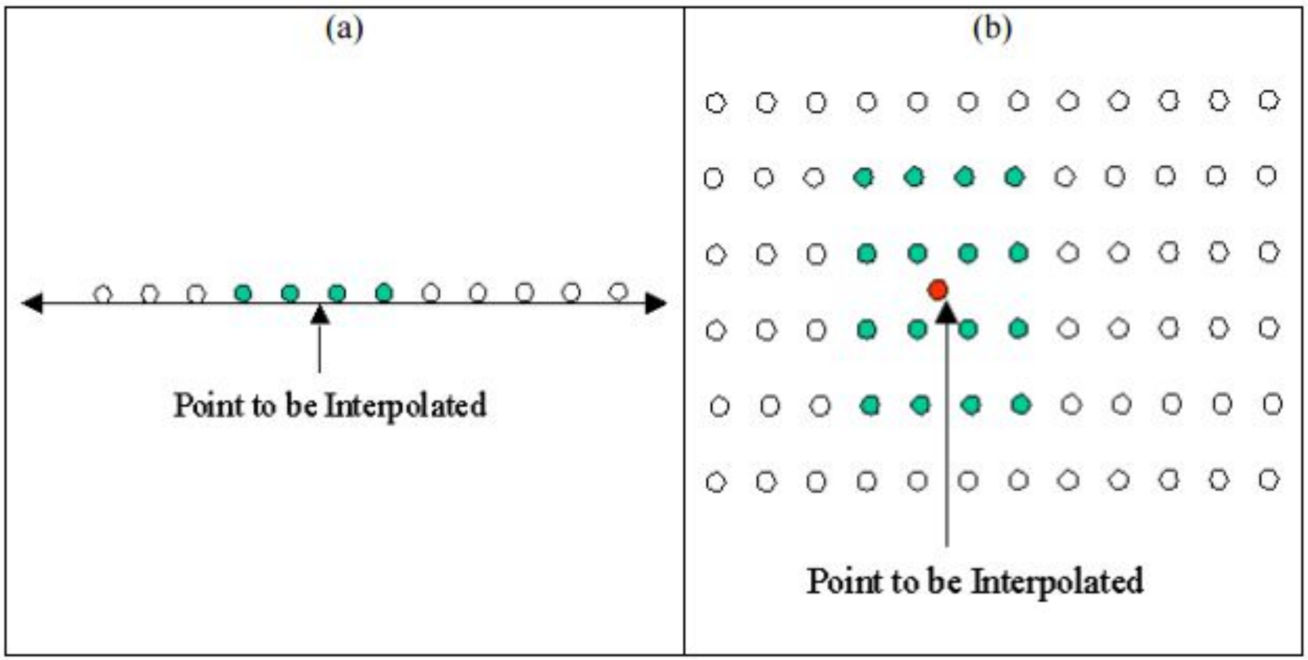
Interpolation	Meaning
cv::INTER_NEAREST	Nearest neighbor
cv::INTER_LINEAR	Bilinear
cv::INTER_AREA	Pixel area re-sampling
cv::INTER_CUBIC	Bicubic interpolation
cv::INTER_LANCZOS4	Lanczos interpolation over 8-by-8 neighborhood.

- type of dst is the same as of src.
- cv::Mat::resize() vs cv::resize()
- If you want to resize src so that it fits the pre-created dst

resize(src, dst, dst.size(), 0, 0, interpolation);

Interpolation methods

- Easy to visualise if if we split these methods into two dimensions.



- Nearest method does not really interpolate values, it just copies existing nearest value.

Interpolation methods

- Nearest Neighbour

$$u(s) = \begin{cases} 0 & |s| > 0.5 \\ 1 & |s| < 0.5 \end{cases}$$

- linear Interpolation

$$u(s) = \begin{cases} 0 & |s| > 1 \\ 1 - |s| & |s| < 1 \end{cases}$$

- Cubic Interpolation

$$u(s) = \begin{cases} 3/2|s|3 - 5/2|s|2 + 1 & 0 <= |s| < 1 \\ -1/2|s|3 + 5/2|s|2 - 4|s| + 2 & 1 <= |s| < 2 \\ 0 & 2 < |s| \end{cases}$$

- Linear vs Bilinear
- Cubic vs Bicubic
- Lanczos interpolation

Image Pyramids

- An image pyramid is a collection of images.
- Each image is successively downsampled from single original image until some desired stopping point is reached.
- Gaussian pyramid is used to downsample images
- Laplacian pyramid to reconstruct an upsampled image from an image lower in the pyramid.

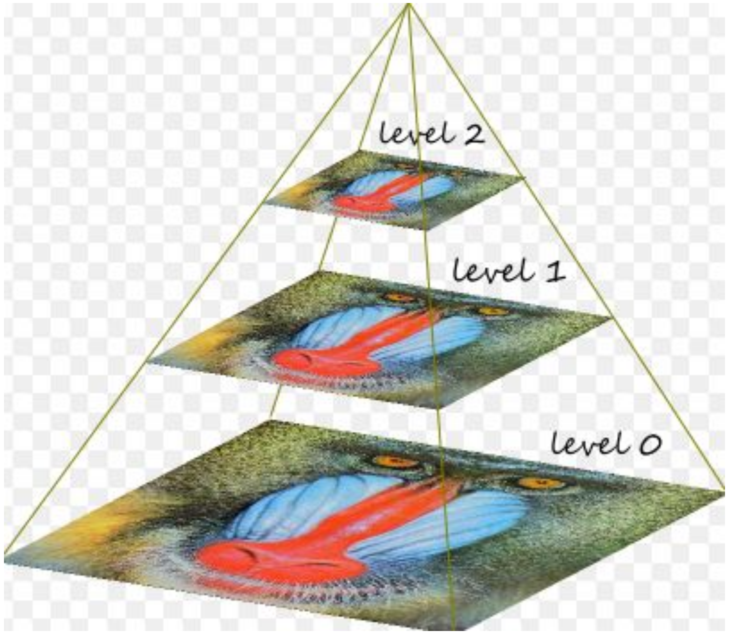


Image Pyramids - Gaussian

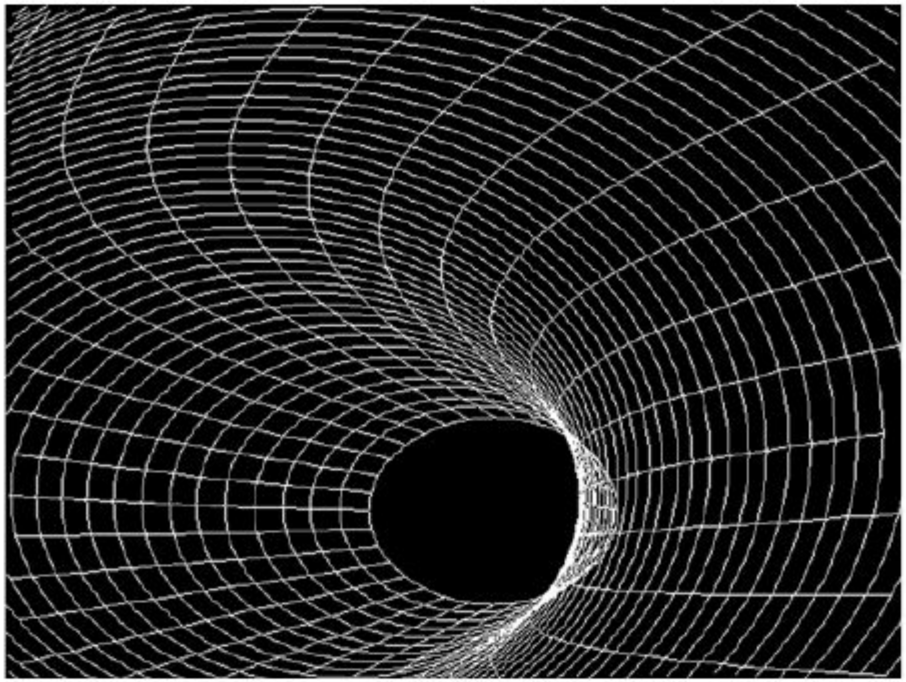
- cv::pyrDown() function is used.
- Blurring an image is the first step to reducing the size of images without effecting their appearance too much.
- Smaller images cannot handle high frequencies nicely.
- Can also explained in frequency domain.
- So, cv::pyrDown() first convolves image with a Gaussian kernel and then removing every even-numbered row and column.
- Usage:

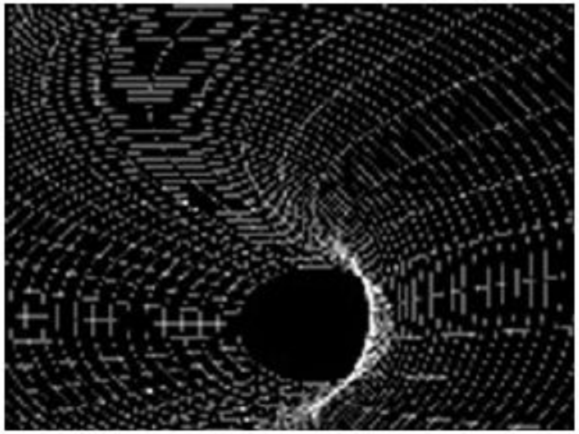
```
void cv::pyrDown(
  cv::InputArray  src,                         // Input Image
  cv::OutputArray dst,                         // Result image
  const cv::Size& dstsize = cv::Size()         // Output image size
);
```

- For it's default value, it scales down by a factor of 2.
- To use dstsize we have to obey some strict constraints. So keep it default.

Image Pyramids - Gaussian

- Aliasing artifacts—simple geometric resizing of an image down by a factor of 4





- The absence of aliasing artifacts in cv::pyrDown() ==>

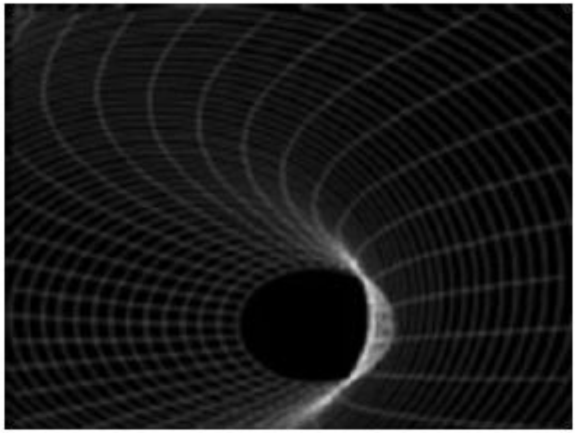


Image Pyramids - Gaussian

- cv::buildPyramid() to build pyramid directly upto desired levels.

```
void cv::buildPyramid(
  cv::InputArray           src,              // Input Image
  cv::OutputArrayOfArrays dst,              // Output Images from pyramid
  int                      maxlevel          // Number of pyramid levels
);
```

- Argument **dst** is of type cv::OutputArrayOfArrays like vector<cv::Mat>
- The first entry in dst will be identical to src. The second will be half and so on..

Image Pyramids - Laplacian

- Useful in converting an existing image to an image that is twice as large.
- It's **not the complete inverse process** of Gaussian Pyramids.
- Laplacian Process:

  -- In order to restore higher-resolution image, we require the information that was discarded by the downsampling

  -- The info. lost in Downsampling is high freqeuncy features like egdes,corners

  -- We use cv::pyrUp() function to just upsample.

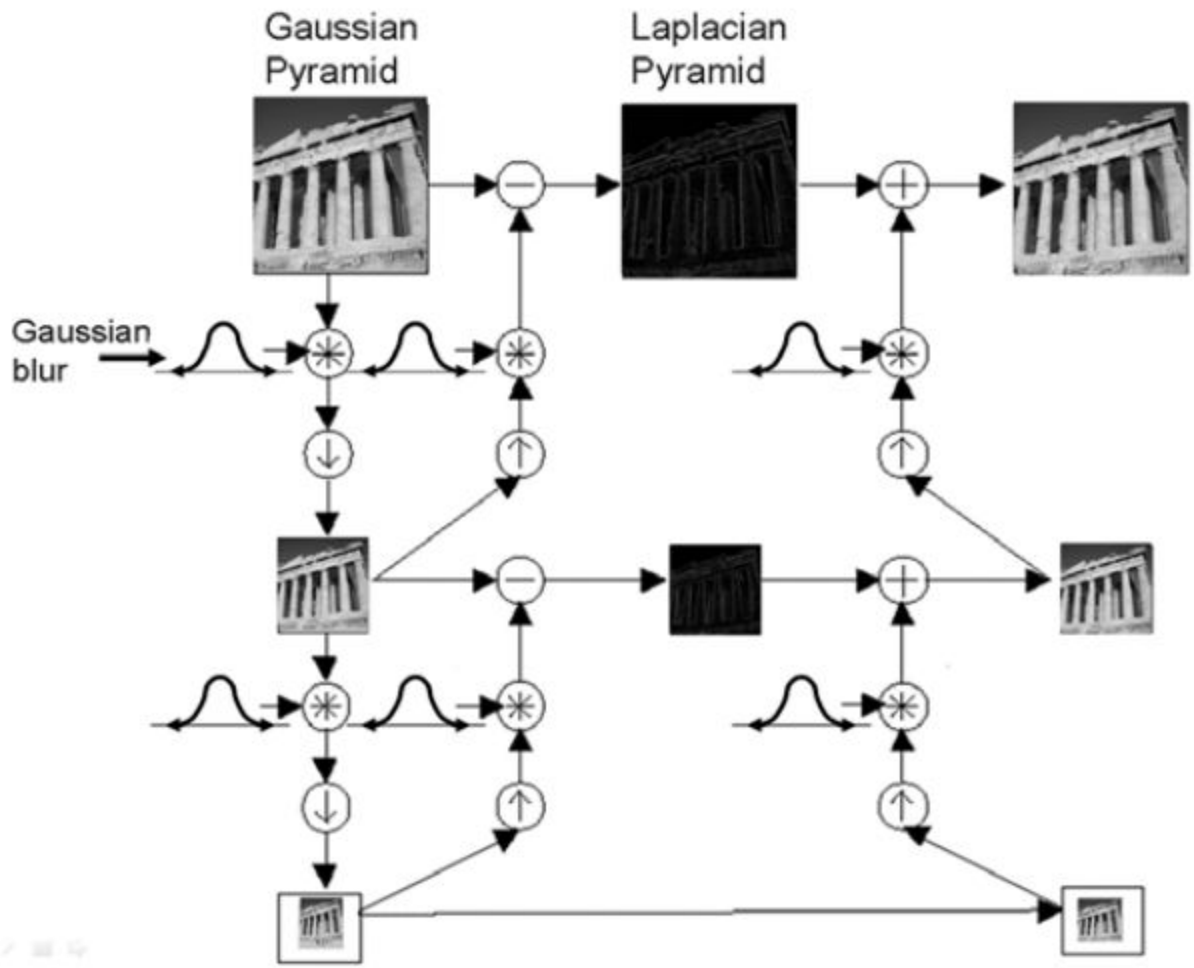
  -- cv::pyrUp() involves upsizing with blank pixels and then convolving with Gaussian to approx. those blank pixels.

  -- Then take original image and subtract the above obtrained image.

  -- The resultant image will just contain high-freq. features

  -- This high-freq. features form the Laplacian Pyramid.
- **REMEMBER** : This doesnot require interpolation as we are not creating any new info.

Laplacian and Gaussian Pyramids



Affine and Perspective Transforms

- Both of them represent the relationship between two images.
- These are the transforms for planar areas.
- **Affine Transformation** that can be expressed in the form of a matrix multiplication (linear transformation) followed by a vector addition (translation).
- The two operations can be augumented into single 2 X 3 matrix M = [A : B]

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times 2} \quad B = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}_{2\times 1}$$

$$M = [A \quad B] = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}_{2\times 3}$$

- Then to transform a 2D vector X = [x ,y] by using A and B we can do it as

$$T = A \cdot \begin{bmatrix} x \\ y \end{bmatrix} + B$$

or

$$T = M \cdot [x, y, 1]^T$$

===>

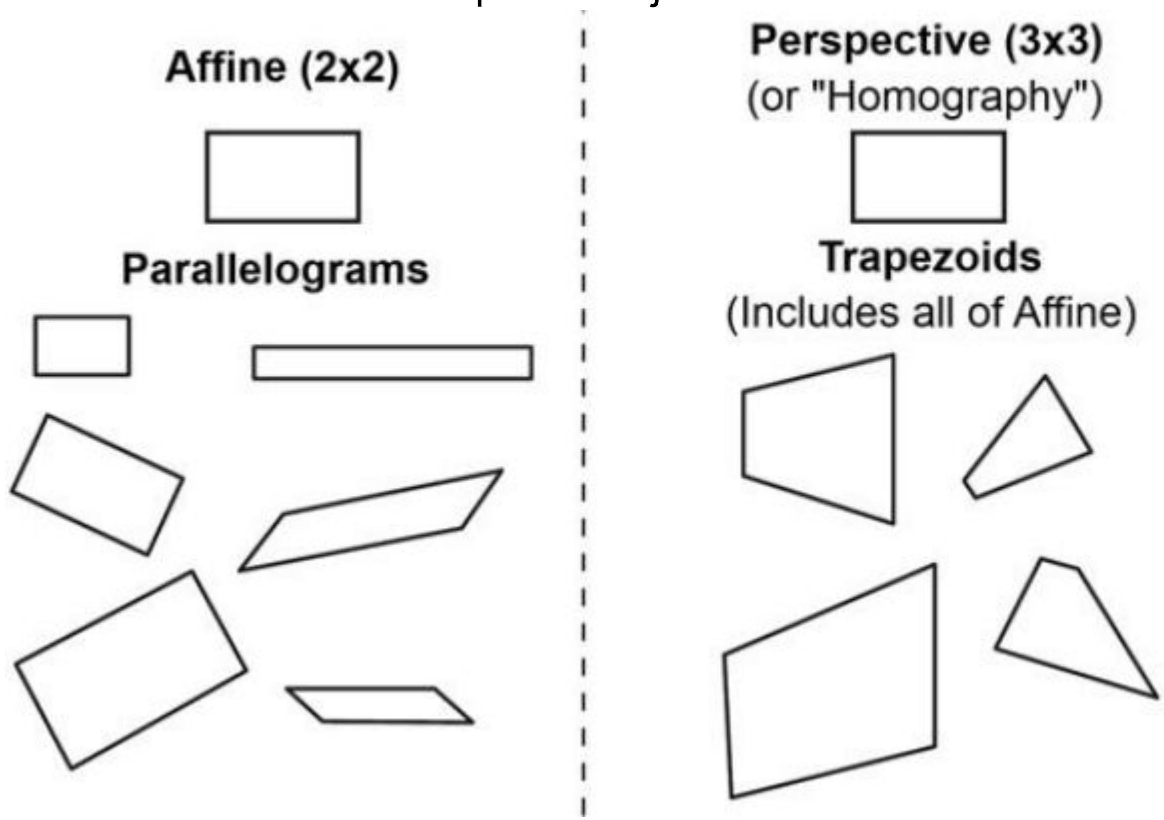
$$T = \begin{bmatrix} a_{00}x + a_{01}y + b_{00} \\ a_{10}x + a_{11}y + b_{10} \end{bmatrix}$$

Affine and Perspective Transforms

- Any parallelogram ABCD in a plane can be mapped to any other parallelogram A′B′C′D′ by some affine transformation.
- We can use an Affine Transformation to express:
  - Rotations (linear transformation)
  - Translations (vector addition)
  - Scale operations (linear transformation)
- Can be defined uniquely by just **three vertices** of each the two parallelograms
- Think of drawing your image into a big rubber sheet and then deforming the sheet by pushing or pulling on the corners.
- Perspective Trasformation is expressed in the form of only matrix multiplication by 3 X 3 matrix.
- So it's just $T = M \cdot [x, y, 1]^T$

Affine and Perspective Transforms

- Perspective Trasform can visualised as computing the way in which a plane in three dimensions is perceived by a particular observer,who might not be looking straight on at that plane.
- It almost always refers to mapping between points on two image planes that correspond to the same location on a planar object in the real world.



Affine and Perspective Transforms

- IMP Difference : Affine can squash the shape but must keep the sides parallel; they can rotate it and/or scale it.

  Perspective transformations offer more flexibility; a perspective transform can turn a rectangle into a trapezoid or any general quadrilateral.

- So, affine transformations are a subset of perspective transformations.

Affine Transformation

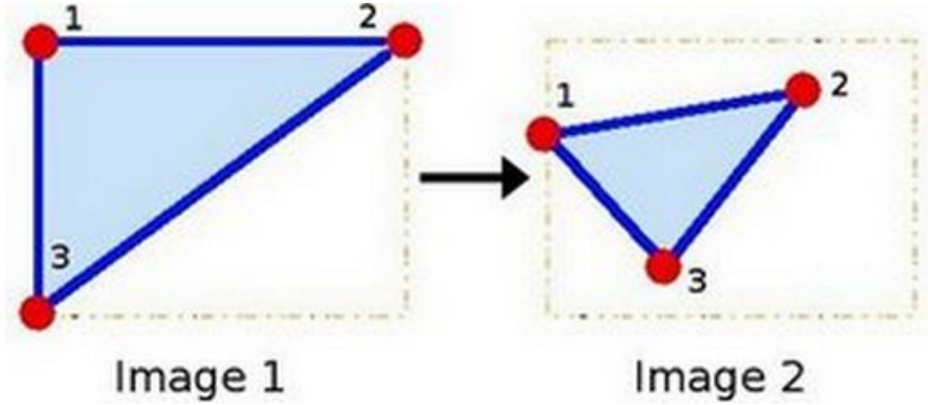
- There are two situations
  - We know both X and T and we also know that they are related. Then our job is to find M
  - We know M and X. To obtain T we only need to apply T = M * X. Our information for M may be explicit (i.e. have the 2-by-3 matrix) or it can come as a geometric relation between points.
- For case 1; we use **cv::warpAffine()**
- Usage :

```
void cv::warpAffine(
  cv::InputArray    src,                           // Input Image
  cv::OutputArray   dst,                           // Result image
  cv::InputArray    M,                             // 2-by-3 transformation matrix
  cv::Size          dsize,                         // Destination image size
  int               flags       = cv::INTER_LINEAR,    // Interpolation, and inverse
  int               borderMode  = cv::BORDER_CONSTANT, // Pixel extrapolation method
  const cv::Scalar& borderValue = cv::Scalar()  // Used for constant borders
);
```

- We get $dst(x, y) = src(M_{00}x + M_{01}y + M_{02}, M_{10}x + M_{11}y + M_{12})$
- Next flags are for interpolation and extrapolation

Affine Transformation

- For case 2 ; we can use relation b/w three points in both images



- Once we find the affine Tnsform b/w these 3 points then we can apply the same transform on all the points in the image.
- We use **cv::getAffineTransform()** to compute Affine Map Matrix

```
cv::Mat cv::getAffineTransform(                // Return 2-by-3 matrix
  const cv::Point2f* src,                      // Coordinates three of vertices
  const cv::Point2f* dst                       // Target coordinates three of vertices
);
```

Affine Transformation

- Another useful function is **cv::getRotationMatrix2D()**
- Calculates map matrix for a **rotation around some arbitrary point**, combined with an optional rescaling.
- It's just one possible kind of affine transformation; but a very frequent transformation.

```
cv::Mat cv::getRotationMatrix2D(                   // Return 2-by-3 matrix
  cv::Point2f  center                              // Center of rotation
  double       angle,                              // Angle of rotation
  double       scale                               // Rescale after rotation
);
```

- The center argument is the center point of the rotation.
- Combining the previous three OpenCV functions an image can be rotated, scaled, and warped.
- cv::invertAffineTransform() supplies the inverse affine transform matrix (**not image**)

```
void cv::invertAffineTransform(
  cv::InputArray  M,                               // Input 2-by-3 matrix
  cv::OutputArray iM                               // Output also a 2-by-3 matrix
);
```

Perspective Transformation

- cv::warpPerspective() does the job here

```
void cv::warpPerspective(
  cv::InputArray    src,                               // Input Image
  cv::OutputArray   dst,                               // Result image
  cv::InputArray    M,                                 // 3-by-3 transformation matrix
  cv::Size          dsize,                             // Destination image size
  int               flags       = cv::INTER_LINEAR,    // Interpolation, and inverse
  int               borderMode  = cv::BORDER_CONSTANT, // Pixel extrapolation method
  const cv::Scalar& borderValue = cv::Scalar()  // Used for constant borders
);
```

- Same as cv::warpAffine() with distinction that the map matrix must now be **3-by-3**
- cv::getPerspectiveTransform() for Computing the perspective map matrix.

```
cv::Mat cv::getPerspectiveTransform(           // Return 3-by-3 matrix
  const cv::Point2f* src,                      // Coordinates of four vertices
  const cv::Point2f* dst                       // Target coordinates of four vertices
);
```

- We need four vertices here, since it can form any quadrilateral

Perspective Transformation

- cv::perspectiveTransform() performs perspective transformations on lists of corresponding points.
- Usage

```
void cv::perspectiveTransform(
  cv::InputArray  src,                    // Input N-by-1 array (2 or 3 channele)
  cv::OutputArray dst,                    // Output N-by-1 array (2 or 3 channels)
  cv::InputArray  mtx                     // Transfor matrux (3-by-3 or 4-by-4)
);
```

- src and dst are lists of points

Arbitrary mappings

- cv::remap() to accomplish a arbitrary mapping.
- Usage

```
void cv::remap(
  cv::InputArray    src,                        // Input image
  cv::OutputArray   dst,                        // Output image
  cv::InputArray    map1,                       // target x-location for src pixels
  cv::InputArray    map2,                       // target y-location for src pixels
  int               interpolation = cv::INTER_LINEAR,    // Interpolation, and inverse
  int               borderMode    = cv::BORDER_CONSTANT, // Pixel extrapolation method
  const cv::Scalar& borderValue   = cv::Scalar()         // Used for constant borders
);
```

- map1 and map2, indicate where any particular pixel is to be relocated from src --> dst.
- Code remapping.cpp

If you proceed furthur ...

- Know about Hough Trasforms

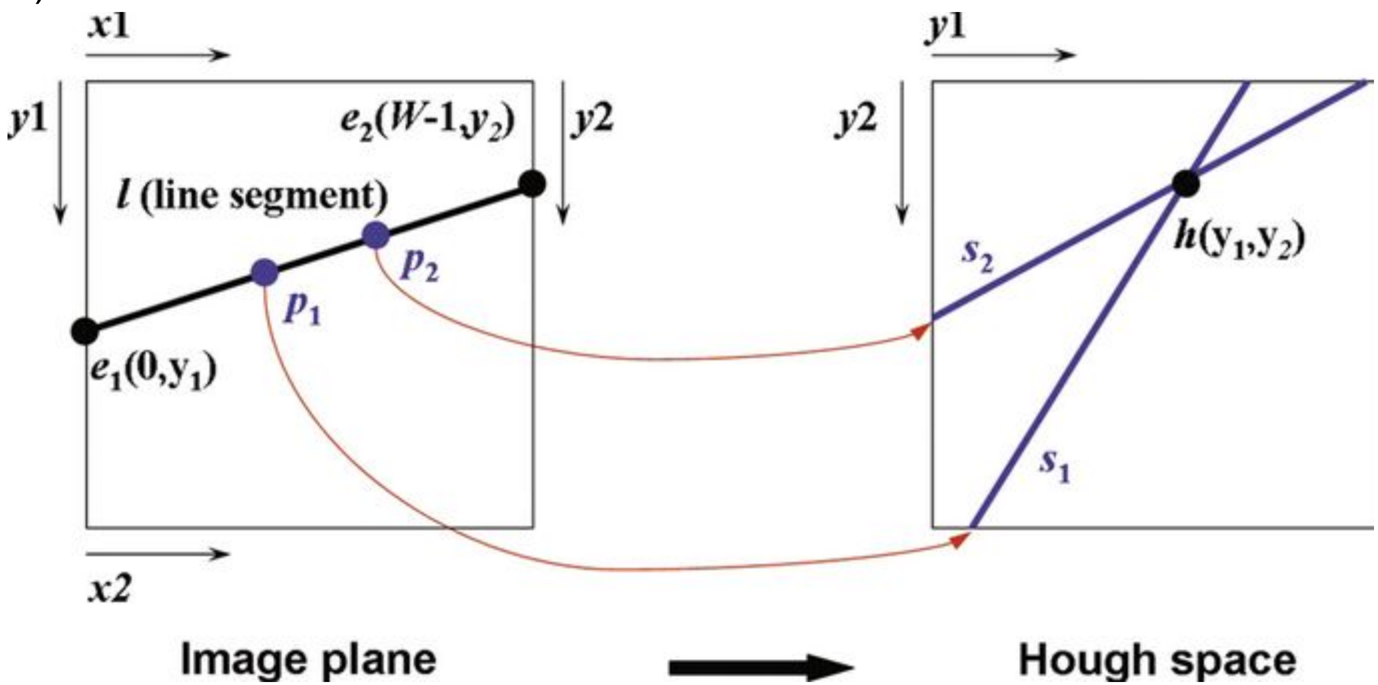
  Histogram Equalization

  Image segmentation methods

  Frequency Transforms

Hough Transform

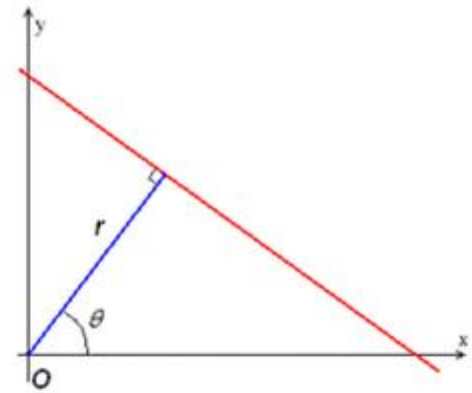
- Useful to find the dominent line segments in a binary image.
- Every point (or pixel) in binary image plane forms a line in parametric space (Hough Space)

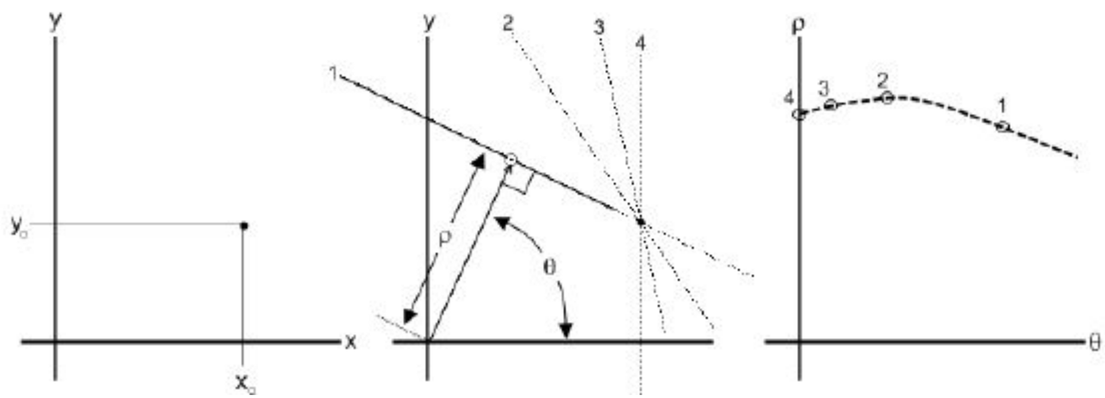


- So, an intersection in Hough Space represents a line joining the two points in Image plane.

Hough Transform

- The earlier point <---> Line transforms were based on normal line equation y=mx+b
- But for the vertical lines, this slope-intercept form will give rise to unbounded values of the slope parameter m.
- To avoid the above problem, $r = x\cos\theta + y\sin\theta$ form of line equation is used.
- Procedure :
  - For each data point, a number of lines are plotted going through it, all at different angles.
  - F





THANK YOU