Content

- Introduction to Robotic Vision & Image Processing
- The Digital Image
- Pixel Point Processing
- Pixel Group Processing
- Geometric Transformation
- Feature Extraction

Geometric Transformation

x, y

- Geometric transformation is used to reposition pixels within an image.
- We can move, spin, size and change the geometry of an image.
- Purpose: Correcting geometric distortions in an image
- Separated into two types:
  - Linear geometric operation:
    - Translation
    - Rotation
    - Scaling
  - Nonlinear geometric operation:
    - Warping

Geometric Transformation

$\rightarrow$ x, y

- Translation, rotation and scaling are quite straightforward – you have already learned about them in this course!

- Translation:
$$\begin{cases} x' = x + T_x \\ y' = y + T_y \end{cases}$$

- Rotation:
$$\begin{cases} x' = x\cos\theta + y\sin\theta \\ y' = -x\sin\theta + y\cos\theta \end{cases}$$

$\gtrless 1$

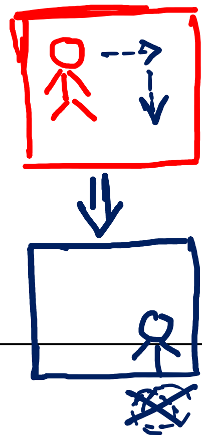
- Scaling:
$$\begin{cases} x' = S_x x \\ y' = S_y y \end{cases}$$

- Note that these operations work on the pixel location, NOT the pixel value!

x, y

MATLAB Codes

- Cameraman Example: Translation



```
I = imread('cameraman.tif');
figure, imshow(I)
title('Original')
```

```
I=double(I);
```

```
[m,n]=size(I);
```

Get the image size

x

y

```
Itrans = zeros(m,n);
TransX = 40;
TransY = 60; % positive means downwards, negative means upwards

for i = 1:m % rows from top to bottom
    for j = 1:n % columns from left to right
        itrans = round(i + TransY);
        jtrans = round(j + TransX);
        if (itrans > 0) && (itrans <= m) && (jtrans > 0) && (jtrans <= n)
            % The above line is to limit the size of output image
            Itrans(itrans, jtrans) = I(i,j);
        end
    end
end

Itrans = uint8(Itrans);
figure,imshow(Itrans)
title('Translated')
```

y

x

round

Rot.

19.2

19

int

MATLAB Codes

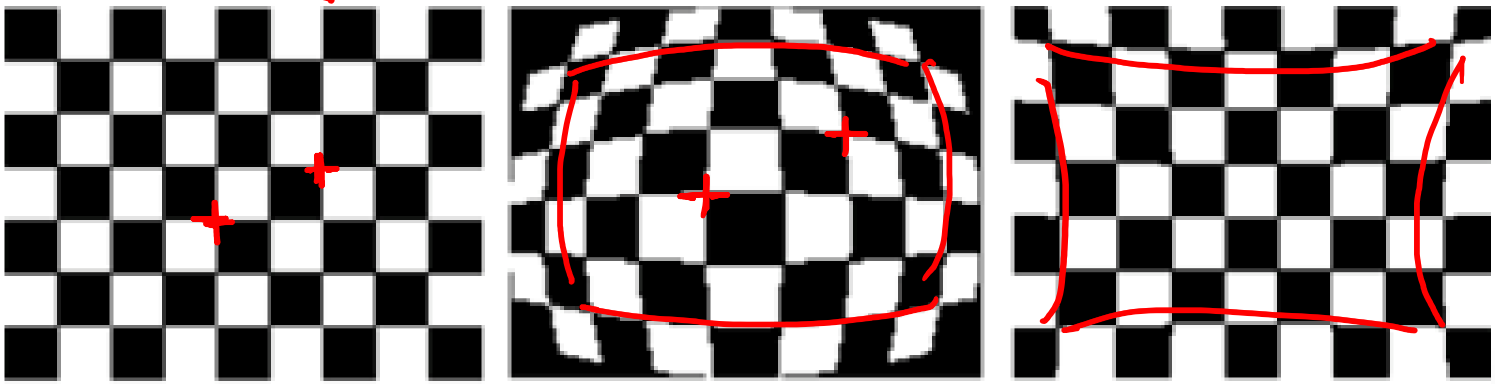
- The outcome of translation is as follows:

Original

Translated

Geometric Transformation

- Warping transformation can arbitrarily stretch and pull the image about defined points.



Original Barrel distortion Pincushion distortion

https://www.embedded-vision.com/platinum-members/bdti/embedded-vision-training/documents/pages/lens-distortion-correction

Geometric Transformation

- Looking back at the previous slides, the transformations can be combined as:

$$\begin{cases} x' = (x\cos\theta + y\sin\theta)S_x + T_x \\ \quad = (S_x\cos\theta)x + (S_x\sin\theta)y + T_x \\ \quad = a_2x + a_1y + a_0 \end{cases}$$

$$\begin{aligned} y' &= (-x\sin\theta + y\cos\theta)S_y + T_y \\ &= (-S_y\sin\theta)x + (S_y\cos\theta)y + T_y \\ &= b_2x + b_1y + b_0 \end{aligned}$$

- Warping are similar polynomial equations, albeit with higher order terms such as x^2, y^2, x^3, y^3 and so on.
  - The higher order, the more complex geometric warping it can be.
- Both the pincushion distortion and the barrel distortion can be removed by using 3rd order warping transformation:

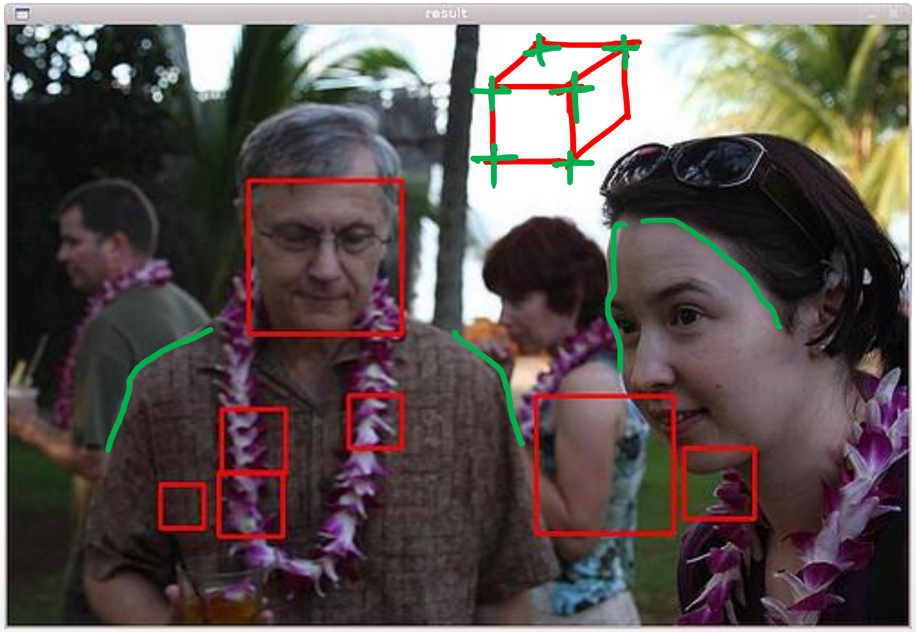
$$\begin{cases} x' = a_9x^3 + a_8y^3 + a_7x^2y + a_6xy^2 + a_5x^2 + a_4y^2 + a_3xy + a_2x + a_1y + a_0 \\ y' = b_9x^3 + b_8y^3 + b_7x^2y + b_6xy^2 + b_5x^2 + b_4y^2 + b_3xy + b_2x + b_1y + b_0 \end{cases}$$

Content

- Introduction to Robotic Vision & Image Processing
- The Digital Image
- Pixel Point Processing
- Pixel Group Processing
- Geometric Transformation
- Feature Extraction

Feature Extraction

- For a robotic vision system to recognize objects, it needs to be able to extract certain features within an image, e.g.
  - Where is the edge?
  - Where is the corner?
  - Where is the blob?
  - What is the shape of the blob?
  - What is the size of the blob?

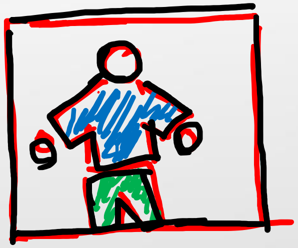


https://www.flickr.com/photos/mrsto/4045264431

Contiguous Region

Continuous

similar color



Edge Detection

- Let’s start by edge detection.
- Given the picture on the right:
  - How do you determine the edge?

Original

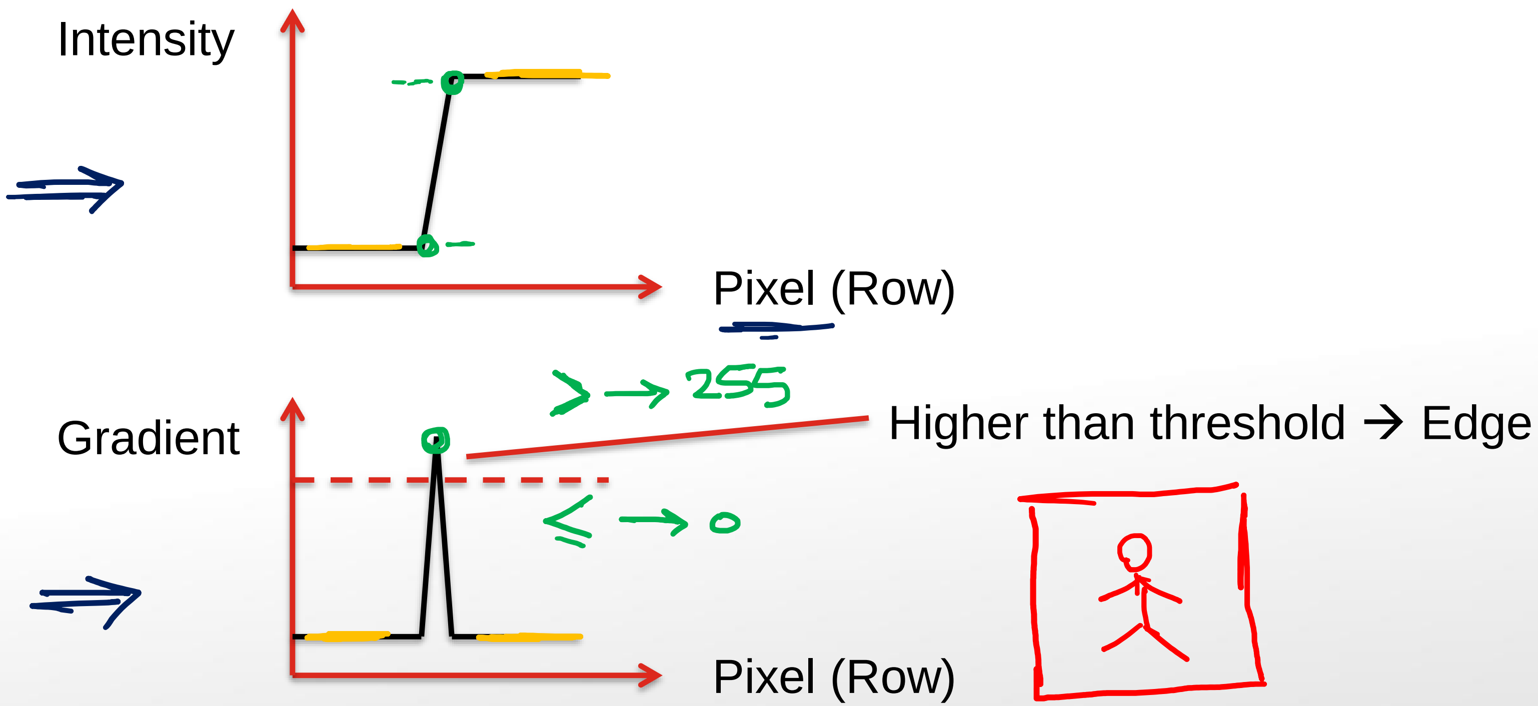
- One answer would be:
  - Edge is where the brightness level experiences a big change.

When we traverse from left to right, the insensitivity values are for instance …150, 150, 150, 150, 0, 0, 0, 0…

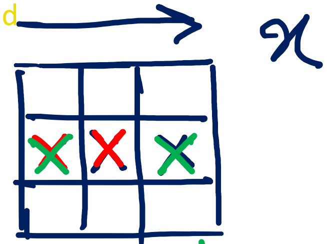
The edge is where 150 drops to 0.

Edge Detection

- How do we then detect where big changes occur?
- Use differentiation / gradient!



Edge Detection



- How do we calculate gradient? Mask
- The discrete approximation can be either:

$$\frac{\partial I(x,y)}{\partial x} \approx \frac{I(x+1,y)-I(x,y)}{1} \qquad \frac{\partial I(x,y)}{\partial x} \approx \frac{I(x,y)-I(x-1,y)}{1} \qquad \frac{\partial I(x,y)}{\partial x} \approx \frac{I(x+1,y)-I(x-1,y)}{2}$$

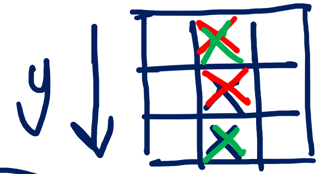
- These can be implemented using the convolution mask filter:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

(Scale by ½ ignored, since actual value not important)

- Note that the weights add up to zero.
  - If the filter passes through a region with constant brightness (no edges), the result will be zero.

Edge Detection



- The same concept applies for the vertical direction and we have:

$$\frac{\partial I(x,y)}{\partial y} \approx \frac{I(x,y+1)-I(x,y)}{1}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

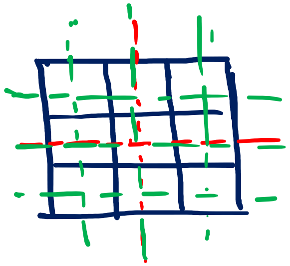
$$\frac{\partial I(x,y)}{\partial y} \approx \frac{I(x,y)-I(x,y-1)}{1}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\frac{\partial I(x,y)}{\partial x} \approx \frac{I(x,y+1)-I(x,y-1)}{2}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Edge Detection



- There are some of the more widely-used gradient filters:

- Prewitt:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

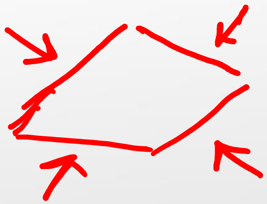
- Sobel:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

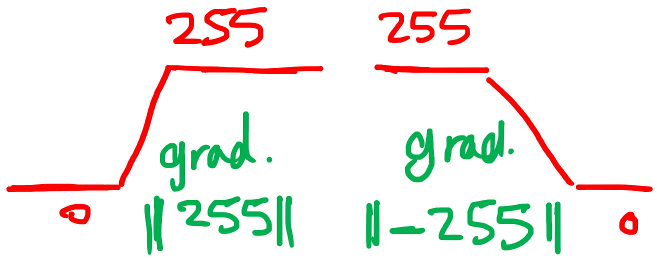
Unidirectional
$\underbrace{M_x \,\&\, M_y}$

- Laplacian (Omnidirectional):

$$M_{xy} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



Edge Detection



Matlab

Uint8

Negative

⇓

0

=

- Important Note 1:
- If the gradient is positive, there is no complication.
- However, if the gradient is negative (edge is still there, just that we come from high intensity region to low intensity region), MATLAB will clip the negative value to be zero.
- We should therefore take the modulus (absolute value) of the gradient value.

abs

- Important Note 2:
- When we use the Prewitt or Sobel filters, we have individual gradients for horizontal and vertical directions.
- To combine them to get both gradients, we can calculate:

$$\Rightarrow \quad \text{Gradient}_{\text{combined}} = \sqrt{\text{Gradient}_x^2 + \text{Gradient}_y^2}$$

MATLAB Codes

- Cameraman example (Sobel):

```
I = imread('cameraman.tif');
figure, imshow(I)
title('Original')
```

```
I=double(I);
```

```
[m,n]=size(I);
```

Get the image size

Get vertical edges

add → sharpening? thresholding?

```
% IsobelTemp = double(I);                 % INSTRUCTION: CHOOSE ONE
% IsobelTemp = double(Isharp);
IsobelTemp = double(Ithreshold);

Isobelx = zeros(m,n); % x because differentiation in x direction. The edge is vertical

for i = 2:m-1
    for j = 2:n-1
        Isobelx(i,j) = (-1*IsobelTemp(i-1,j-1)+0*IsobelTemp(i-1,j)+IsobelTemp(i-1,j+1)...
            -2*IsobelTemp(i,j-1)+0*IsobelTemp(i,j)+2*IsobelTemp(i,j+1)...
            -IsobelTemp(i+1,j-1)+0*IsobelTemp(i+1,j)+IsobelTemp(i+1,j+1));
    end
end

Isobelx = sqrt(Isobelx.^2); % getting absoluate value
Isobelx = uint8(Isobelx);
figure,imshow(Isobelx)
title('Sobel Vertical Edge Detection')
```

Good to pre-process the image first

MATLAB Codes

Get Horizontal edges

```
Isobely = zeros(m,n);   % y because differentiation in y direction. The edge is horizontal

for i = 2:m-1
    for j = 2:n-1
        Isobely(i,j) = (1*IsobelTemp(i-1,j-1)+2*IsobelTemp(i-1,j)+IsobelTemp(i-1,j+1)...
            +0*IsobelTemp(i,j-1)+0*IsobelTemp(i,j)+0*IsobelTemp(i,j+1)...
            -IsobelTemp(i+1,j-1)-2*IsobelTemp(i+1,j)-IsobelTemp(i+1,j+1));
    end
end

Isobely = sqrt(Isobely.^2); % getting absolute value
Isobely = uint8(Isobely);
figure,imshow(Isobely)
title('Sobel Horizontal Edge Detection')
```

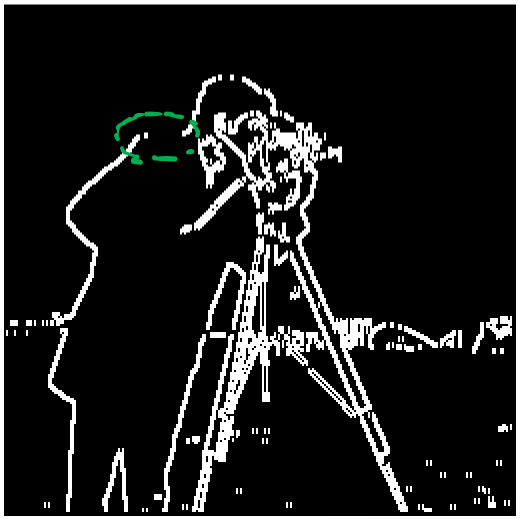
```
Isobelx = double(Isobelx);
Isobely = double(Isobely);
Isobelxy = sqrt(Isobelx.^2+Isobely.^2);

Isobelxy = uint8(Isobelxy);
figure,imshow(Isobelxy)
title('Sobel Combined Edge Detection')
```

Combine vertical and horizontal

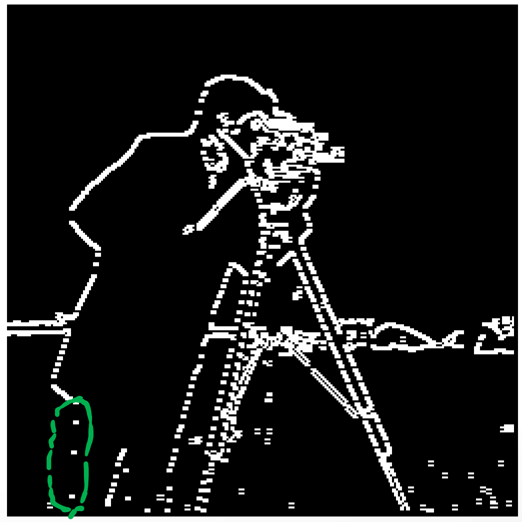
MATLAB Codes

- Sobel Results:

Sobel Vertical Edge Detection



Sobel Horizontal Edge Detection



Sobel Combined Edge Detection

- The edges are well-detected.

MATLAB Codes

- Cameraman example (Laplace):

```
I = imread('cameraman.tif');
figure, imshow(I)
title('Original')
```

```
I=double(I);
```

```
[m,n]=size(I);
```

Get the image size

Get omnidirectional edges

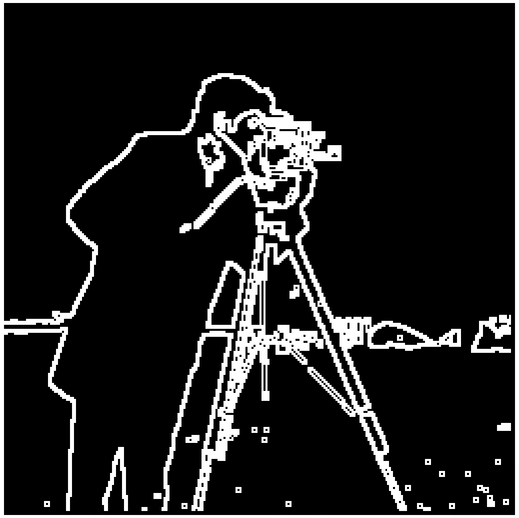
```
% IsobelTemp = double(I);                % INSTRUCTION: CHOOSE ONE
% IsobelTemp = double(Isharp);
IlaplaceTemp = double(Ithreshold);

Ilaplace = zeros(m,n); % x because differentiation in x direction. The edge is vertical

for i = 2:m-1
    for j = 2:n-1
        Ilaplace(i,j) = (-1)*IlaplaceTemp(i-1,j-1)-1*IlaplaceTemp(i-1,j)-1*IlaplaceTemp(i-1,j+1)...
            -1*IlaplaceTemp(i,j-1)+8*IlaplaceTemp(i,j)-1*IlaplaceTemp(i,j+1)...
            -IlaplaceTemp(i+1,j-1)-1*IlaplaceTemp(i+1,j)-1*IlaplaceTemp(i+1,j+1));
    end
end

Ilaplace = sqrt(Ilaplace.^2); % getting absoluate value
Ilaplace = uint8(Ilaplace);
figure,imshow(Ilaplace)
title('Laplacian Omnidirectional Edge Detection')
```

Good to pre-process the image first

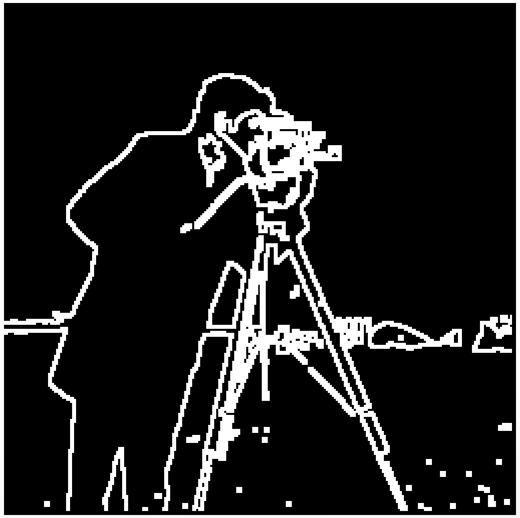
MATLAB Codes

- Laplacian Results:

Sobel Combined Edge Detection

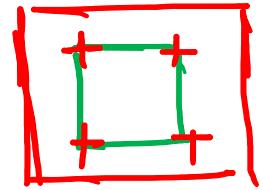


Laplacian Omnidirectional Edge Detection



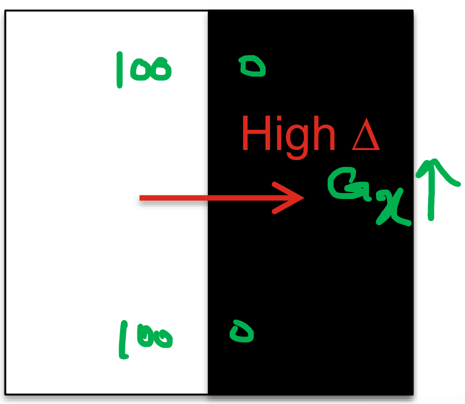
- The edges are well-detected.

Corner Detection

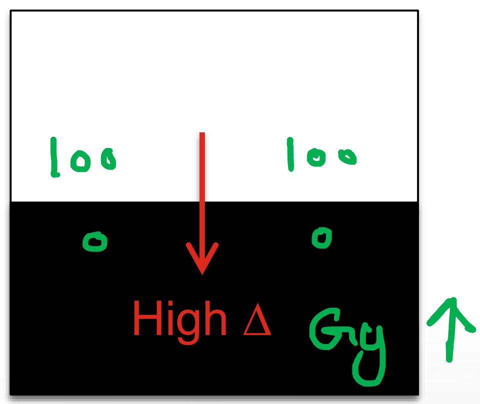


$G_c = \sqrt{G_x^2 + G_y^2}$

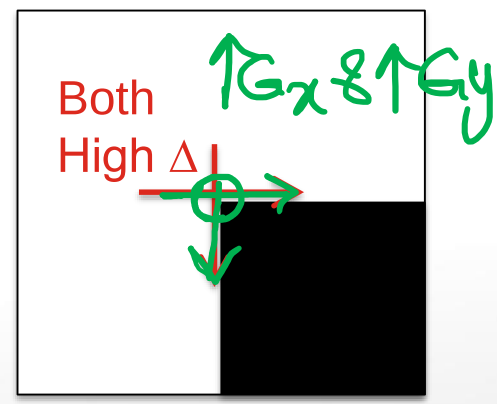
- Now that we already know the concept of edge detection, corner detection becomes intuitive.
- What is a corner? It is where BOTH vertical and horizontal gradients are high.



Not a corner



Not a corner



Corner!

$$\sqrt{100^2 + 0} \text{ or } \sqrt{0 + 100^2} < \sqrt{100^2 + 100^2}$$

Corner Detection

- For instance, these are the horizontal and vertical edges near a corner of a rectangle,

	43	44	45	46
36	0	0	0	0
37	0	1	1	0
38	0	3	3	0
39	0	4	4	0

	43	44	45	46
36	0	0	0	0
37	0	1	3	4
38	0	1	3	4
39	0	0	0	0

  - Whereby the convolution masks were: $M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ $M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

  - Note: "abs" has been used on the individual results.

- By using: $\text{combined edges} = \sqrt{(\text{horizontal edges})^2 + (\text{vertical edges})^2}$
  - We get:

	43	44	45	46
36	0	0	0	0
37	0	1.4142	3.1623	4
38	0	3.1623	4.2426	4
39	0	4	4	0

corner

Corner Detection

- There are some advanced algorithms for corner detection, e.g. Harris Corner Detection.
- This is beyond the scope of this course.
- Interested students can try to find out more on this.
- No assignments or exams will require the knowledge of these advanced algorithms.

issues of sobel :

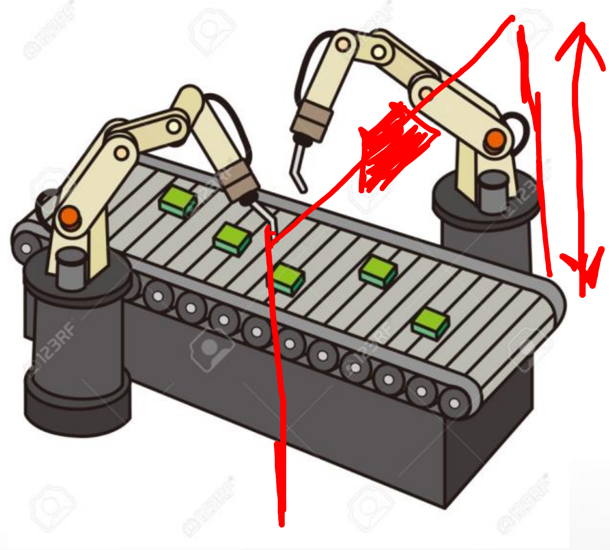
use four conv. mask (with only positive values) without abs

$$G1_x : \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad G2_x : \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix} \quad G3_y : \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad G4_y = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$$G_C = \sqrt{G1_x^2 + G2_x^2 + G3_y^2 + G4_y^2}$$

Industrial Setting

- From this point onwards, we will focus on industrial settings, where the parts are more deterministic:
  - Known shape
  - Known colour etc.

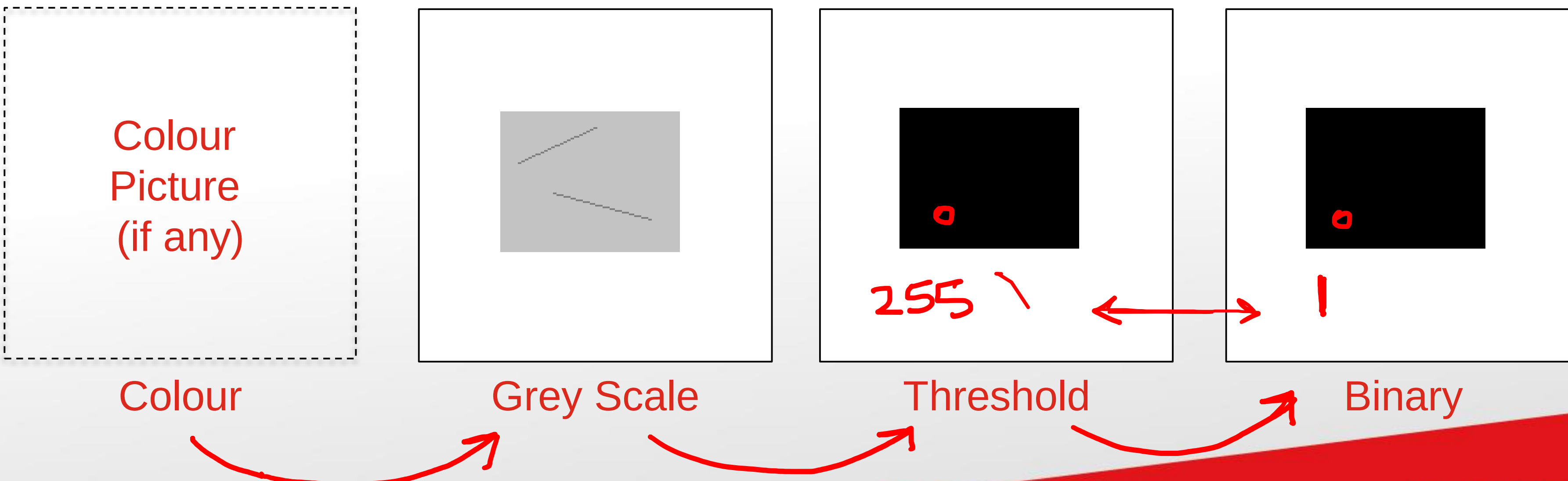


https://www.123rf.com/photo_27504443_stock-vector-belt-conveyor-and-industrial-robot.html

- The vision system's task is then to determine the correct part and also find out the position / orientation.
- This makes the problem simpler (as compared to robot vision in unstructured environment, such as self-driving car).

Black and White Image

- Earlier, we discussed about colour images.
- We also discussed about how to convert the colour images into grey scale images.
- Then, we learnt about thresholding, which makes all the pixels either 0 or 255.
- Now, we will take one more step – Convert all the pixels to just 0 or 1.



MATLAB Codes

- Import Colour Image:

```
I = imread('GreyRectangle.tif');
```

- Change to Grey Scale:

```
IRed = double(I(:,:,1));
IGreen = double(I(:,:,2));
IBlue = double(I(:,:,3));
IGrey = (IRed+IGreen+IBlue)/3;
I = uint8(IGrey);
```

MATLAB Codes

- Thresholding:

```
[m,n]=size(I);
```

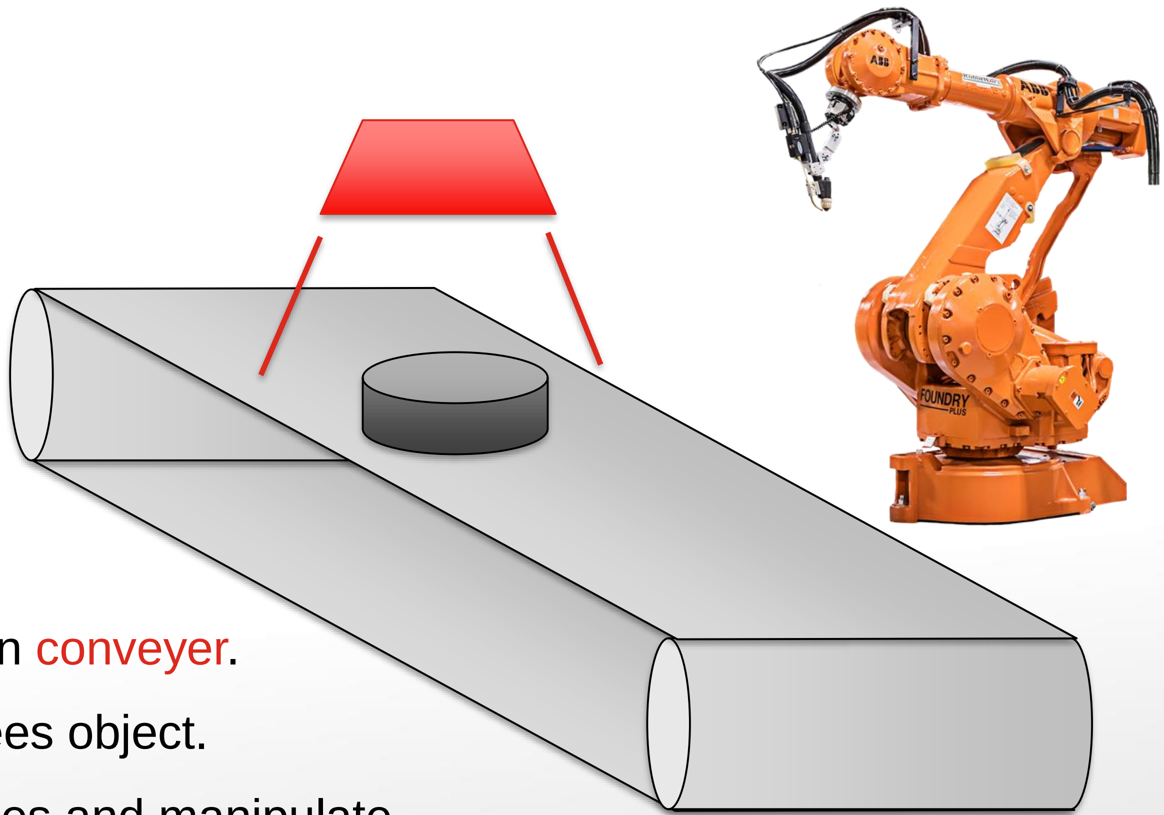
```
Ithreshold = zeros(m,n);

for i = 1:m
    for j = 1:n
        if I(i,j) > 220
            Ithreshold(i,j) = 255;
        else
            Ithreshold(i,j) = 0;
        end
    end
end

Ithreshold = uint8(Ithreshold);
figure,imshow(Ithreshold)
title('Threshold')
```

- Convert to Binary:

```
Ibw = imbinarize(Ithreshold);
figure,imshow(Ibw);
title('Binary')
```

Blob Detection

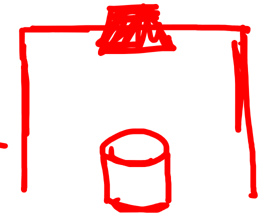
- Assume the following situation:



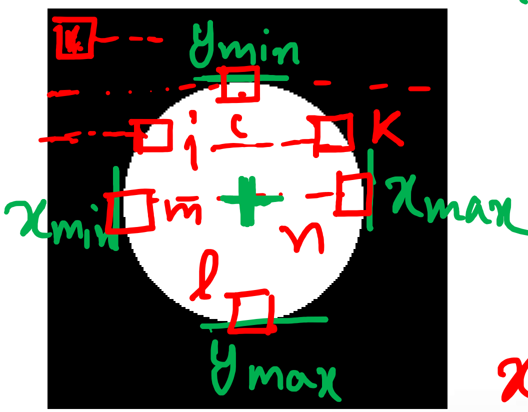
https://www.robots.com/articles/the-speed-of-abb-arc-welding-robots

- Object is on conveyer.
- Camera sees object.
- Robot comes and manipulate the object.

Blob Detection



- The image seen by the camera (within its field of view) would be as follows, after some thresholding operation and conversion to binary image:



x_{min} x_{max} y_{min} y_{max}

$\cancel{x_i}$ $\cancel{x_i}$

$\cancel{x_j}$ $\cancel{x_k}$

$\underline{x_m}$ $\underline{x_n}$ $\underline{y_i}$ $\underline{y_l}$

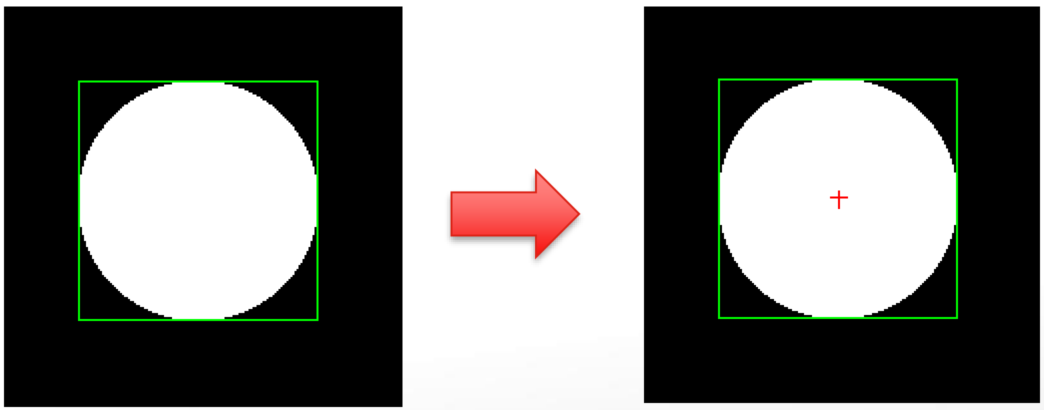
$$x_c = \frac{x_{min} + x_{max}}{2}$$

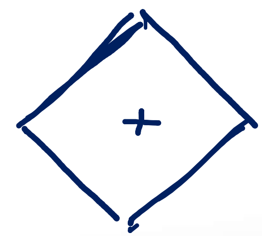
$$y_c = \frac{y_{min} + y_{max}}{2}$$

- With this image, we can now work on "blob detection".
  - Blob means a region or connected components;
  - A set of contiguous (adjacent) pixels of the same colour or value.

Blob Detection

- Where is the blob?
- First – Find the maximum or minimum pixel location in x and y axes.
- Next, calculate the middle point as average of max and min:





X from 39 to 158
Y from 38 to 157

Center = (98.5, 97.5)

- So now the vision system knows where the blob is, and the robot can now come and pick up the object.

Moment

COM

0 – moment ⇒ COM
1 – "
2 – "

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 \end{array}$$

- The p-q^th-moment of an image is:

$$\Rightarrow \quad M_{pq} = \sum_{(x,y)\in \text{Image}} x^p y^q I(x,y)$$

coordinates

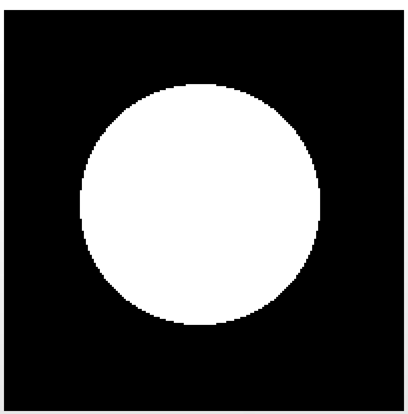
$$M_{00} = \sum I(x,y) = 1+1+0+0+1+0+0+1+1 = 5$$

- $p+q$ is the order of moment.
- The $p = q = 0$ moment of an image is:

$$M_{00} = M_{pq} = \sum_{(x,y)\in \text{Image}} I(x,y)$$

$$M_{02} = \sum y^2 I(x,y) = 0 \times (\text{-----}) + 1 \times (0+1+0) + 4 \times (0+1+1) = 9$$

which is simply the number of white pixels (or area) for a binary image.



Area = 11300

Verification: Radius was 60
Area is πr^2 = 11310
Similar as shape is not continuous circle

Moment and Blob Detection

- The moments can be used for finding the centroid (center of mass) as well!
- The formulae are:

$$X_C = \frac{M_{10}}{M_{00}} \qquad Y_C = \frac{M_{01}}{M_{00}}$$

- The answers given by this method is:

Center = (98.5, 97.5)

which is the same as using the min/max/average method.

Central Moments & Inertia Matrix

0,0

centroid

- Moments were calculated based on the origin (0, 0).
- We can calculate the moments with respect to the centroid.
- This is called "Central Moment". $\Rightarrow$

$$\mu_{pq} = \sum_{(x,y)\in \text{Image}} (x - X_C)^p (y - Y_C)^q I(x,y)$$

- It can be shown that:

$$\mu_{00} = M_{00}$$ Area

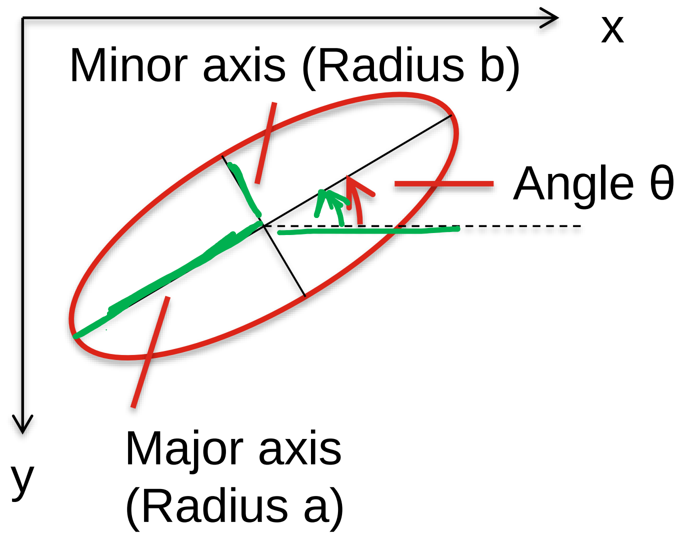
$$\mu_{01} = 0$$
$$\mu_{10} = 0$$
$$\mu_{11} = M_{11} - X_C M_{01} = M_{11} - Y_C M_{10}$$
$$\mu_{20} = M_{20} - X_C M_{10}$$
$$\mu_{02} = M_{02} - Y_C M_{01}$$

- The inertia matrix of a blob is then:

$$J = \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}$$

Inertia Matrix & Ellipse

- Assume we have an ellipse:



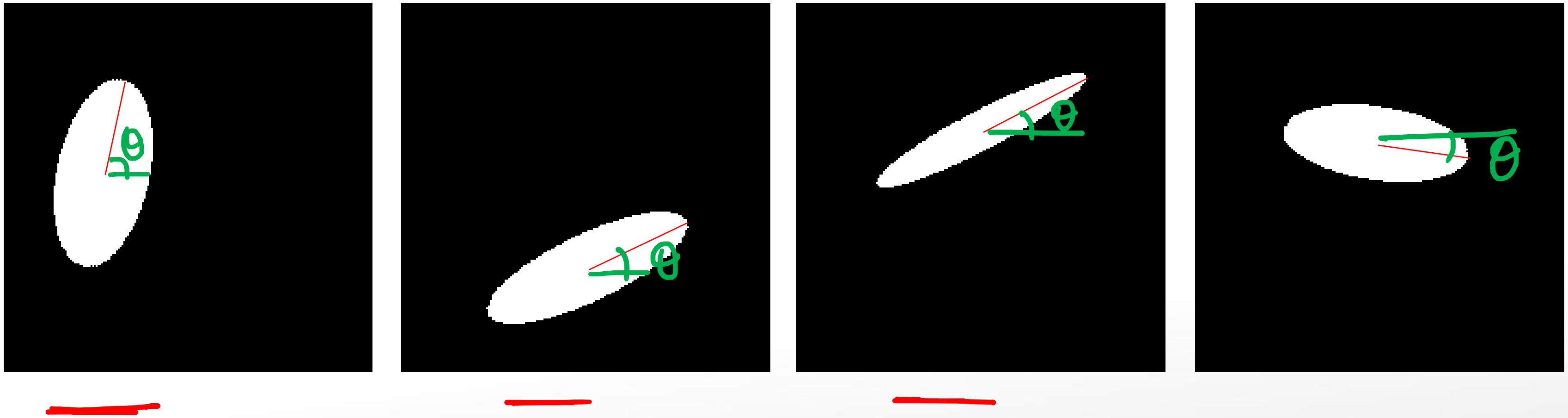
- All the ellipse parameters can be obtained from the inertia matrix / moments!

$$a = 2\sqrt{\frac{\lambda_1}{M_{00}}} \qquad b = 2\sqrt{\frac{\lambda_2}{M_{00}}} \qquad \theta = \arctan\left(\frac{V_y}{V_x}\right)$$

  - Where λ_i are the eigenvalues of J with $\lambda_1 > \lambda_2$.
  - And $V = \begin{bmatrix} V_x \\ V_y \end{bmatrix}$ is the eigenvector corresponding to the largest eigenvalue, λ_1.

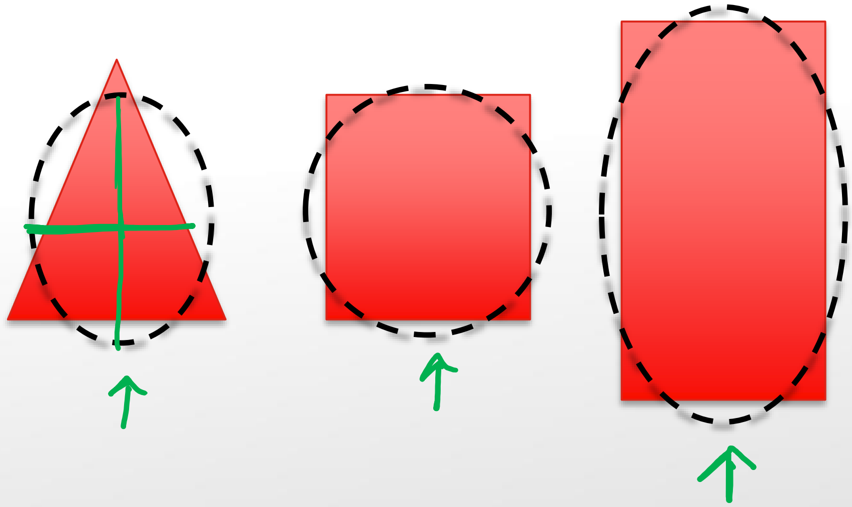
Inertia Matrix & Ellipse

- Example: These were the figures showing the major axes for different ellipse, using the inertia matrix.



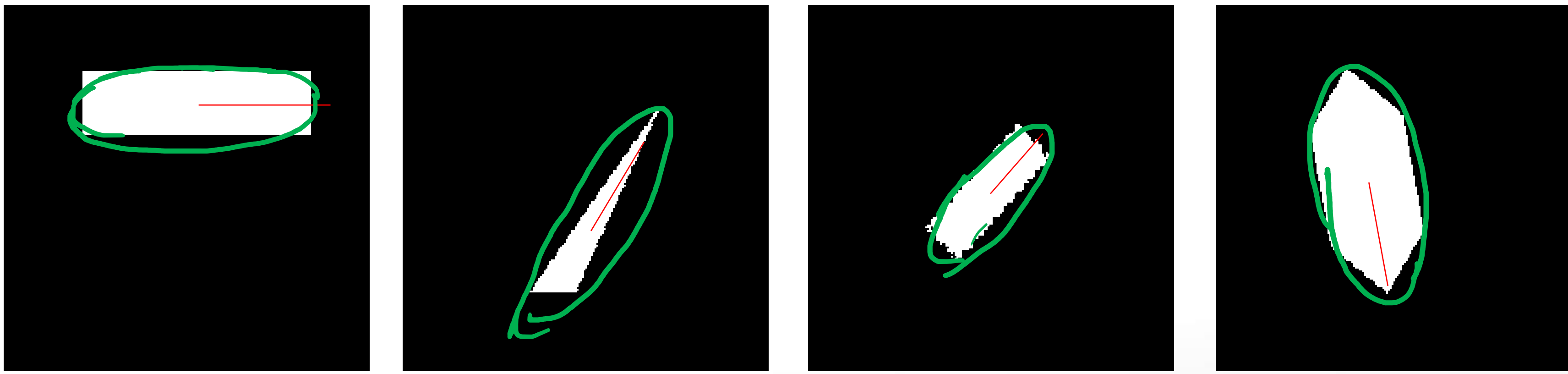
Inertia Matrix & Other Shapes

- Why is ellipse useful?
- For all other shapes, we can always fit an "equivalent ellipse" to the shapes.
- The equivalent ellipse is centred at the object's centre of gravity, and has the same moment of inertia.
- Therefore, we can use the exact same formula (or code) to find out the information about the object.



Inertia Matrix & Other Shapes

- Example: These were the figures showing the major axes for equivalent ellipse, which are fitted to the different shapes.



- So at this stage, the vision system already determined the position (centroid) and orientation (angle) of the object.
  - Robot can come and pick up the object!

Shape Recognition

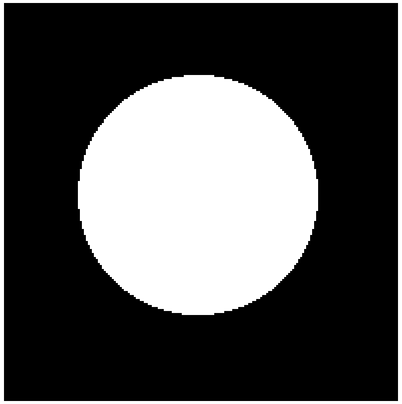
- There are many ways to determine the shape of an object.
- Method 1: Perform corner detection, and count the corners.
  - Note: Need to beware if algorithm gives a "group of pixels" at the corner. This should be treated as one!
- Method 2: Compute "Circularity"
  - Find perimeter p of shape (e.g. edge detection then sum up all "1" pixels).
  - Find area of shape (i.e. M_{00})
  - Then:

$$Circularity = \frac{4\pi M_{00}}{p^2}$$

- This would be 1 for a circle; π/4 for square; 0 for long line etc.

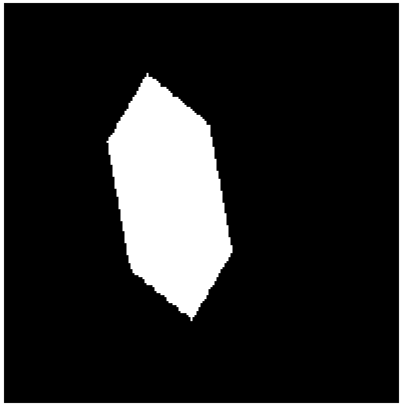
Shape Recognition

- Examples of Circularity:

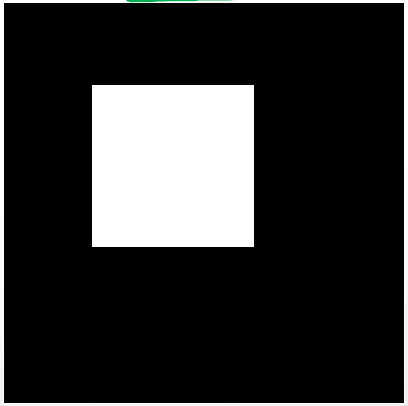


1.2578

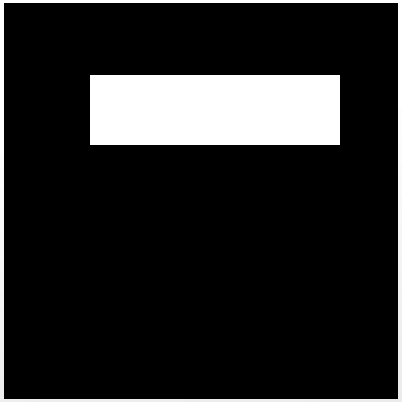
0.7426



0.8940



0.8052
(≈ π/4 i.e.
0.7854)

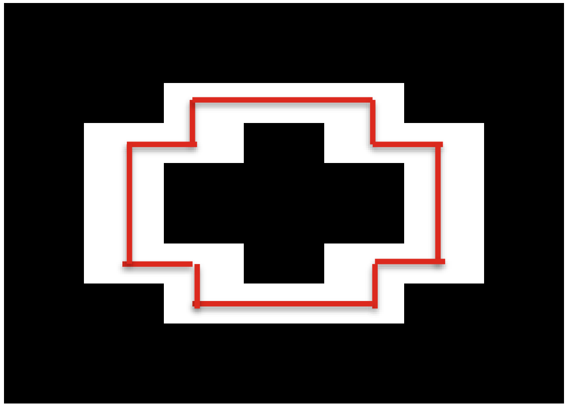


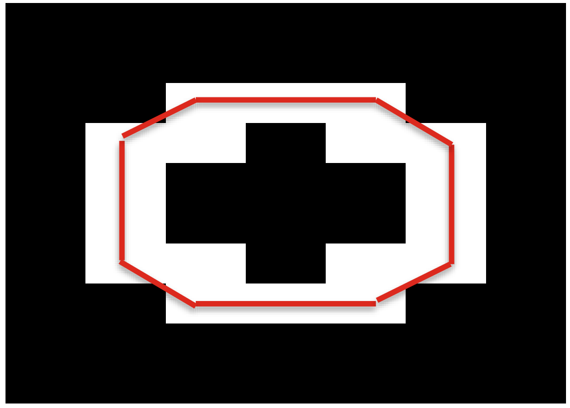
0.5506

0.3131

Shape Recognition

- Note: The result in the examples do not match the theoretical results. The circularity for circle is not exactly 1, and for square it is not exactly π/4.
- This is due to the approximation when finding perimeter.
- For instance, this is the edge of a very pixelated circle:
  - The code used in previous example sums up all the “1”, meaning that the perimeter is as shown on the left:





  - A better approximation would be to consider the diagonal components too, as shown on the right.
  - Obviously, the coding will be more involved.

Thank you!

Have a good evening.