ENGR110 T2 2019

Engineering Modeling and Design
Project 2 - Solar tracker. Image processing Part1.

Arthur Roberts

School of Engineering and Computer Science
Victoria University of Wellington

Project

Figure:

Solar tracker project involves tracking the sun. Sunflowers can do that. We are smarter.
It involves adjusting motors to keep the red circle (sun) in the middle of the screen. Quadrant 4, basically, only in 2 dimensions.

Some help working without RPI

There is small program which emulates working of RPI, **read.cpp**. You can find it zipped together with some images taken by RPI camera on **Assignments** course webpage.
Program takes pictures from image ***.ppm** files.
Functions **get_pixel()** and **set_pixel()** are same as in E101 library. Feel free to modify code to suit you.

Red ruby, anyone?

Sun is always red.

- Take the picture. Scan it going through all the pixels. Obtain color of each pixel (**get_pixel(row,column,rgb)**)
- For **each pixel**: if $red/blue > threshold$ and $red/green > threshold$ and $red > something$- mark this pixel as red.

Now we have bunch of pixels which are red. How can we decide if sun is there?

Feature extraction

It becomes **feature extraction**. It involves dimensionality reduction - instead of thousands of pixels we have one answer - where is the circle (if it is there).
Should filter out other objects which are not circles.

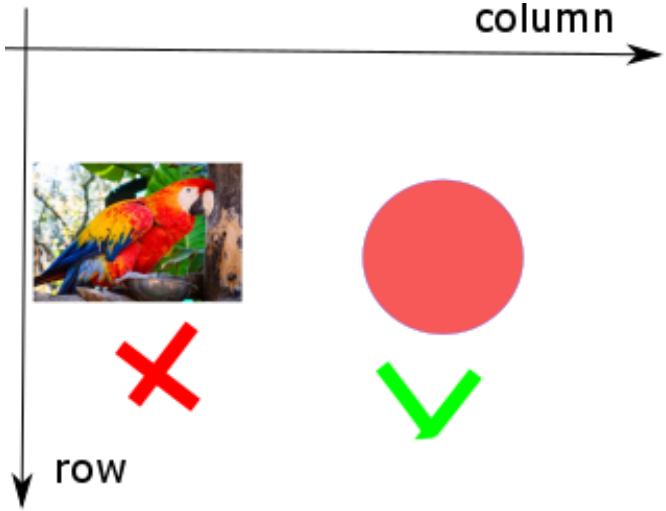


Figure:

Approach 1

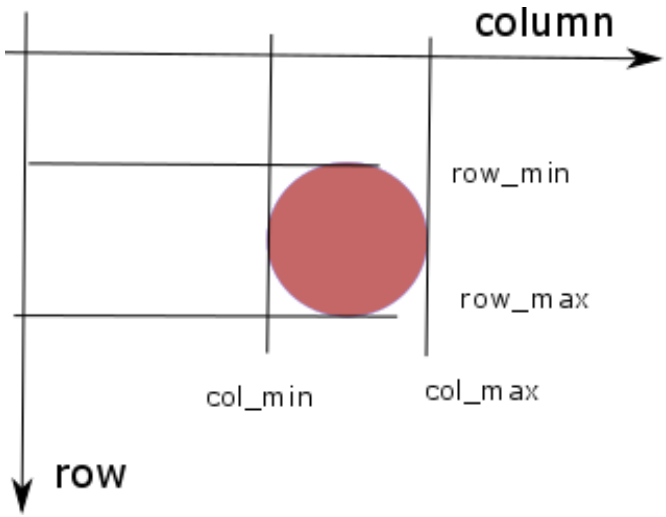


Figure:

- Determine leftmost column of all red pixels: *col_min*. Same for rightmost: *col_max*
- Repeat for columns.
- We can calculate center of the sun position by $\frac{col\_min+col\_max}{2}$ $\frac{row\_min+row\_max}{2}$
- Span of the sun disk as $dy = col\_max - col\_min$ and $dy = row\_max - row\_min$. If dx and dy are different - what it means?

What about something (not sun) red in the frame? Say, red bird is flying across.

Approach 2

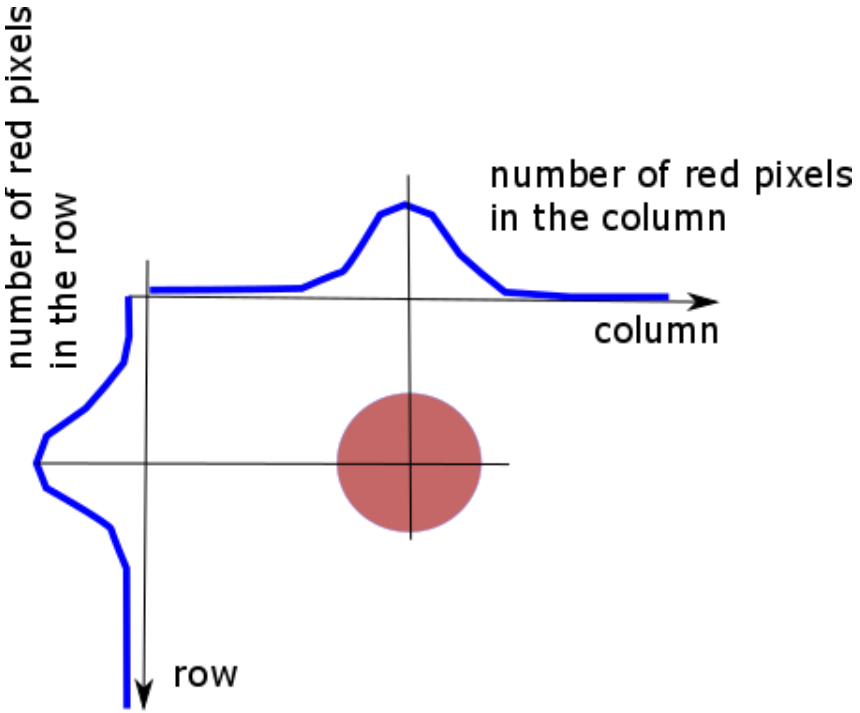


Figure:

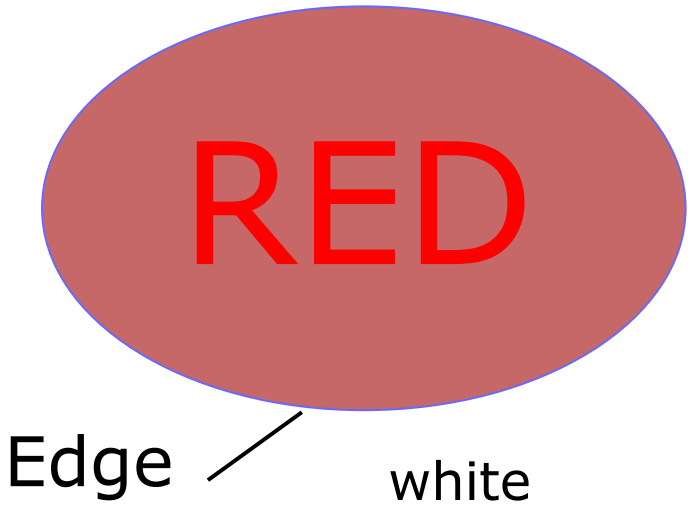
- For each row count how many red pixels are in the row
- Same for each column
- Find row with maximum number of red pixels in it
- Same for column
- Thus we find the center of most massive red blob
- But is it circle?

If bird is small compared with the sun - can be OK.

100% honest (and 200% hard) way

Detect the shape.

So we try to detect if transition from white to red is a circle. Transition is called the **edge** - the change in luminosity (or color).



We want to detect the change. What indicates the change? Derivative, of course.

1D derivative

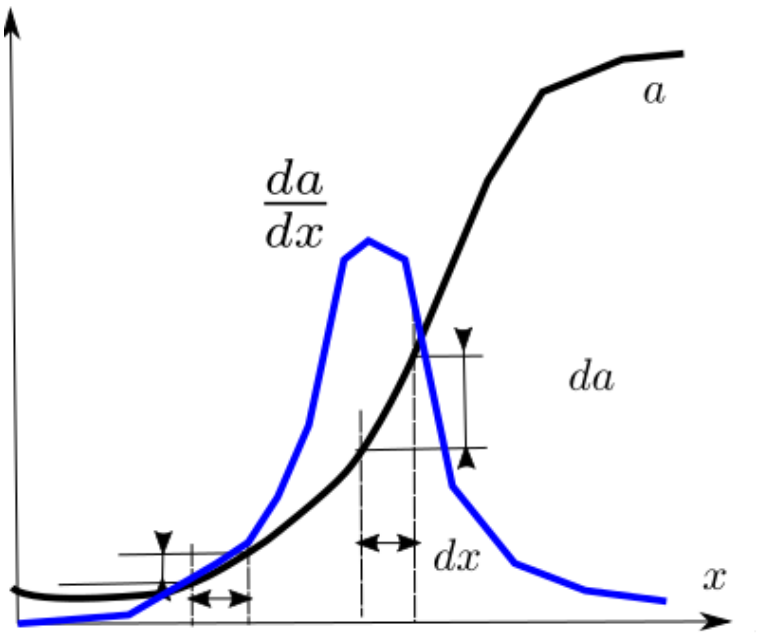


Figure: Derivative estimation: $\frac{da}{dx}$

Now we want to estimate the derivative for all values of the argument x as we don't know where change is happening.
If $\frac{da}{dx} > threshold$ - there is the edge at x.

1D derivative - more accurate

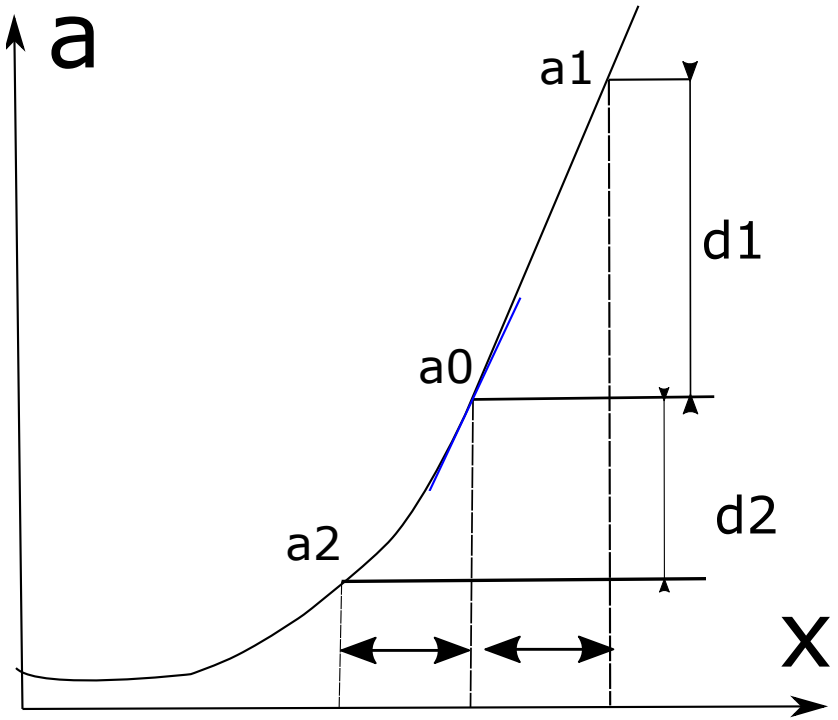


Figure:

- When estimating derivative to step to the right
- More accurate way is step to the right (a1), then to the left (a2)
- Then take average

$$\frac{da}{dx} = \frac{\frac{a1-a0}{1} + \frac{a0-a2}{1}}{2} = \frac{a1-a2}{2} \quad (1)$$

1D derivative - How it looks for arrays?

This calculation can be written as a matrix:

$$da/dx_i = a_{i-1} * b_0 + a_i * b_1 + a_{i+1} * b_2 \tag{2}$$

, where $b_0 = -0.5, b_1 = 0$ and $b_2 = 0.5$, to be repeated for all is.
b is called a **kernel**.

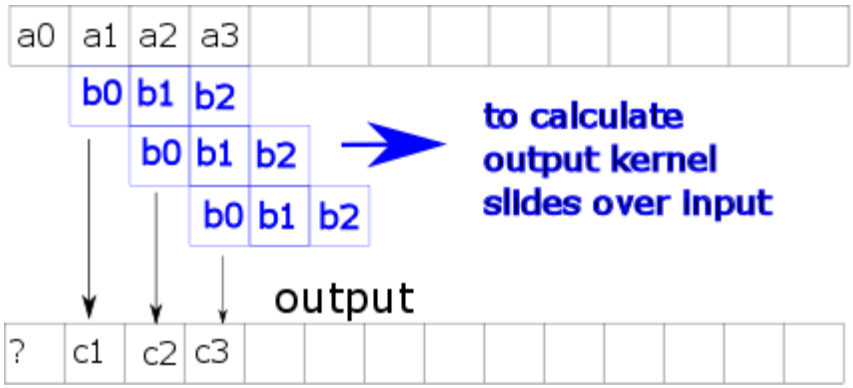


Figure: 3 elements kernel

To get the derivative:

$$\begin{aligned} b_0 &= 0.5 \\ b_1 &= 0.0 \\ b_2 &= -0.5 \end{aligned} \tag{3}$$

Notice that kernel can not be applied at the boundaries of the image . We are losing some pixel at the boundary.

2D derivative

Operation can be extended to 2D case.

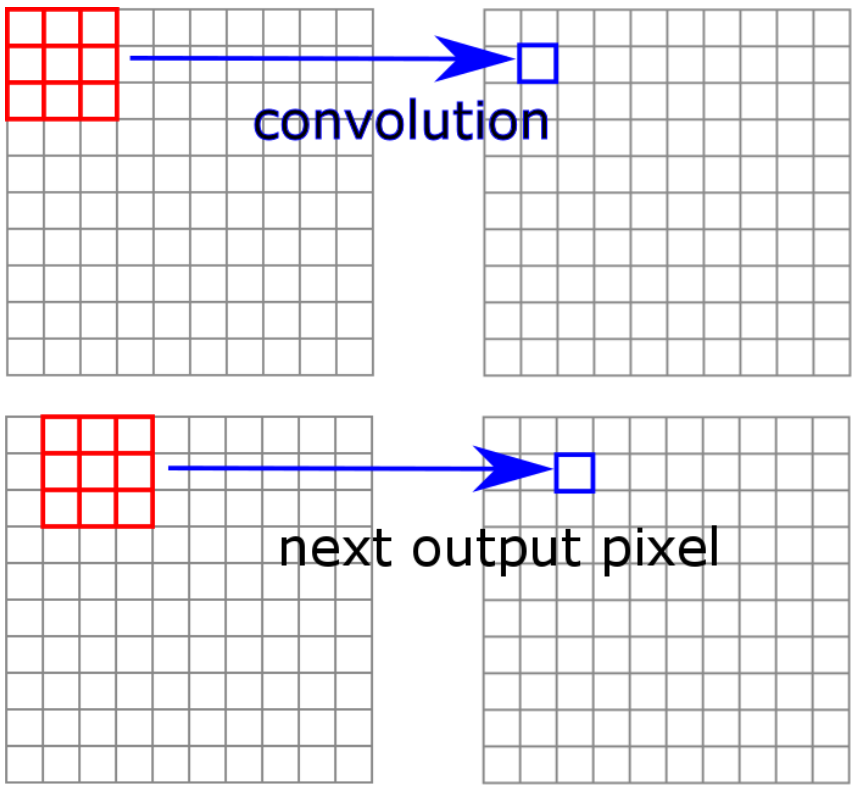


Figure: Kernel(red) moves over the input image

For each pixel in output image except boundaries:

- pick same location pixel in an input image
- apply kernel weights to neighbours (multiply)
- add all that together
- store result in output image pixel
- repeat for all (except boundaries) pixels of the output image

Calculating output pixel c[10][10]:

a[9][8]	a[9][9] b[0][0]	a[9][10] b[0][1]	a[9][11] b[0][2]	a[9][12]
a[10][8]	a[10][9] b[1][0]	a[10][10] b[1][1]	a[10][11] b[1][2]	a[10][12]
a[11][8]	a[11][9] b[2][0]	a[11][10] b[2][1]	a[11][11] b[2][2]	a[11][12]

Figure:

Calculation of the output follows the rules of matrix multiplication.

$$c[10][10] = a[9][9] * b[0][0] + a[9][10] * b[0][1] + a[9][11] * b[0][2] + a[10][9] * b[1][0] + a[10][10] * b[1][1] + a[10][11] * b[1][2] + a[11][9] * b[2][0] + a[11][10] * b[2][1] + a[11][11] * b[2][2] \quad (4)$$

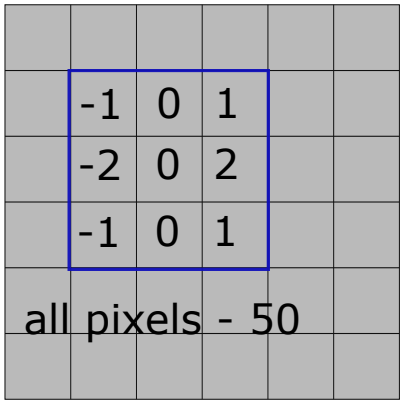
Kernels for edge detection

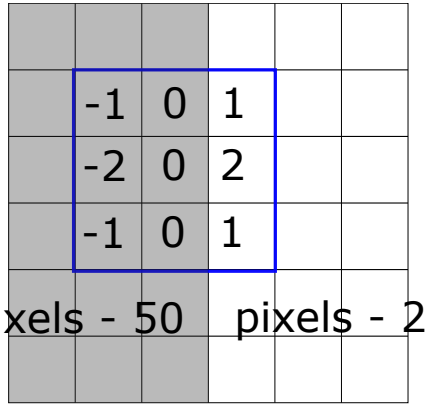
Popular **Sobel** edge detector:

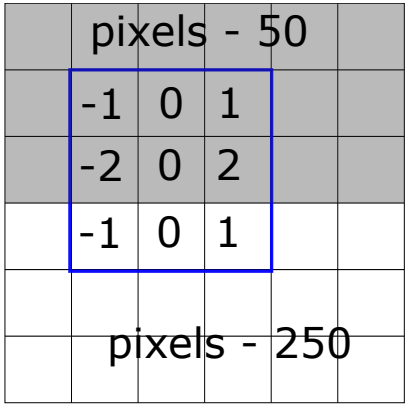
$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{pmatrix}$$

There are two kernels - one for detecting vertical edges and one for detecting horizontal edges.

How Sobel kernels work?







What is result:
-1*50 + 0*50 + 1*50+
-2*50 + 0*50 +2*50 +
-1*50 + 0*50 +2*50 =
0

What is result:
-1*50 + 0*50 + 1*250+
-2*50 + 0*50 +2*250 +
-1*50 + 0*50 +2*250 =
1050

What is result:
-1*50 + 0*50 + 1*50+
-2*50 + 0*50 +2*50 +
-1*250 + 0*50 +2*250 =
0

Vertical(ish) edges are detected. Perfectly horizontal edges are not detected at all.

Horizontal and vertical edges example

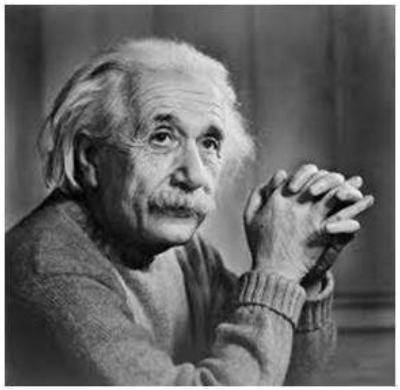


Figure: Original image

Figure: Vertical edges

Figure: Horizontal edges

image from:https://www.tutorialspoint.com/dip/sobel_operator.htm

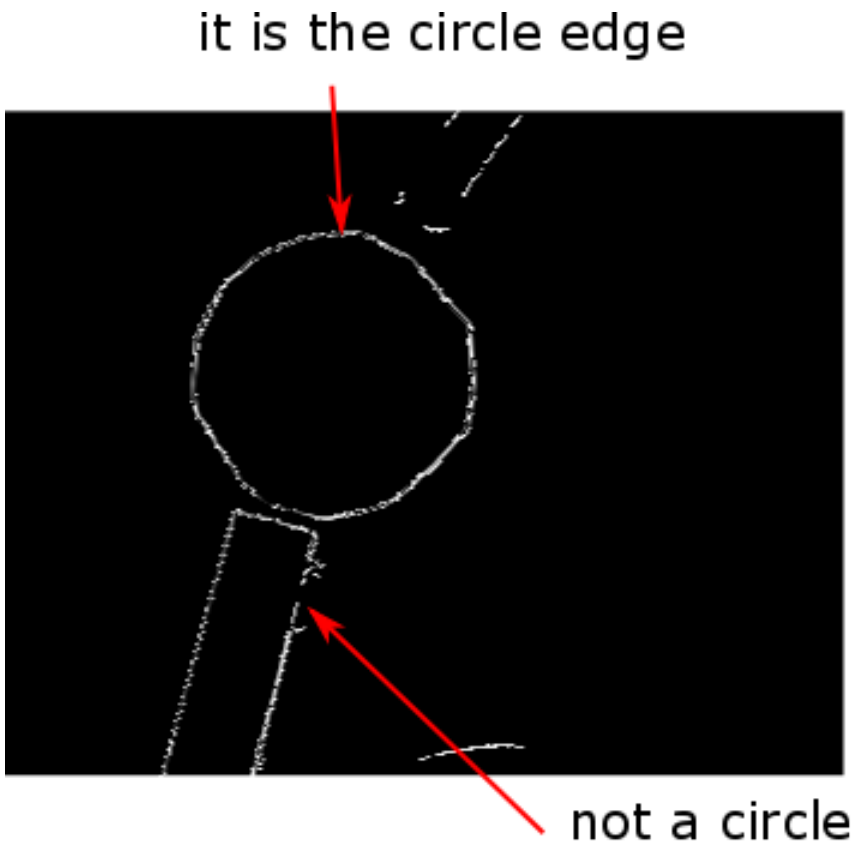
Sobel kernels calculate what is called **partial derivatives** - derivative in certain direction.

How to unite vertical and horizontal edges?

- Run convolution of input images with two kernels.
- It produces two images: one with vertical edges (I_x) and another with horizontal edges I_y
- Combine both images into one applying $\sqrt{((I_x)^2 + (I_y)^2}$ to each pixel, where I_x is the image with horizontal edges and I_y is the image with vertical edges.

Global algorithm (Hough transform)

In simple form:



How can we decide which (if any) pixels form the circle?
More on it next time. Hint - it is called **Hough transform**.

Questions?