Low-level feature extraction

Edge detection

Image Feature Extraction Techniques

Piotr Fulmański

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
University of Lodz

Derivative

Derivative and differencing

Derivative

The derivative of a function of a real variable measures the **sensitivity to change of the function value** (output value) with respect to a change in its argument (input value).

The derivative of a function f at a point x is defined by the limit.

The derivative of a function of a single variable at a chosen input value, when it exists, is the slope of the tangent line to the graph of the function at that point. The tangent line is the best linear approximation of the function near that input value. For this reason, the derivative is often described as the "instantaneous rate of change", the ratio of the instantaneous change in the dependent variable to that of the independent variable.

Derivatives are a fundamental tool of calculus. For example, the derivative of the position of a moving object with respect to time is the object's velocity: this measures **how quickly the position of the object changes when time advances**.

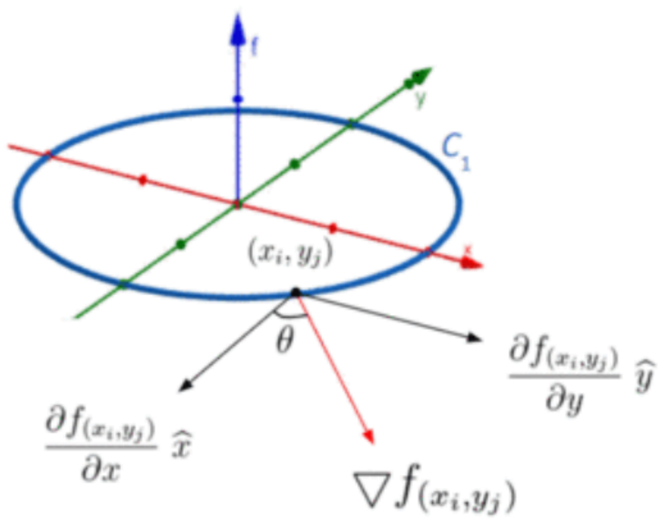
A function of a real variable $y = f(x)$ is differentiable at a point a of its domain, if its domain contains an open interval I containing a, and the following limit exists:

$$\lim_{h \to 0} \frac{f(a+h) - f(a)}{h}.$$

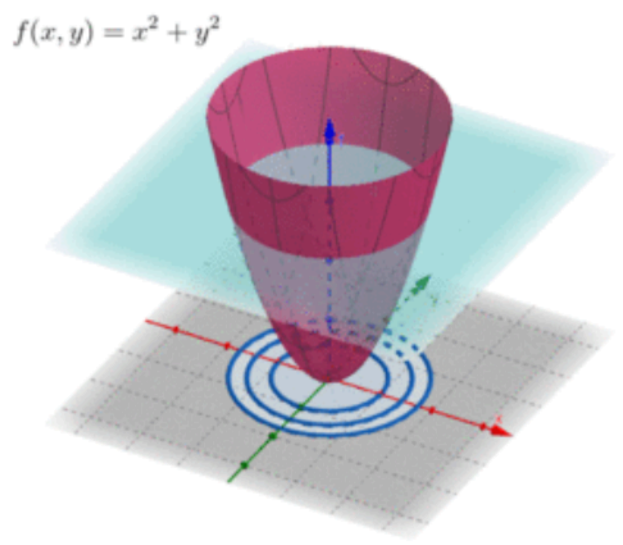
This limit is called the *derivative* of f at a, and denoted $f'(a)$.

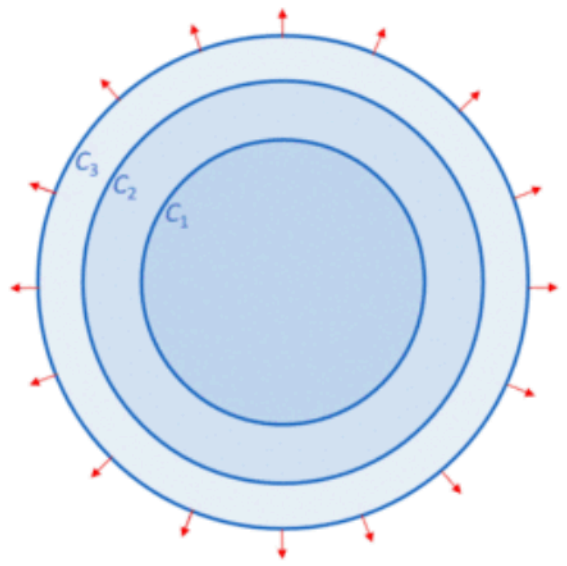
Derivative and differencing

Calculus of the gradient vector



f
y
C_1
(x_i, y_j)
θ
$\frac{\partial f_{(x_i,y_j)}}{\partial y}\,\widehat{y}$
$\frac{\partial f_{(x_i,y_j)}}{\partial x}\,\widehat{x}$
$\nabla f_{(x_i,y_j)}$





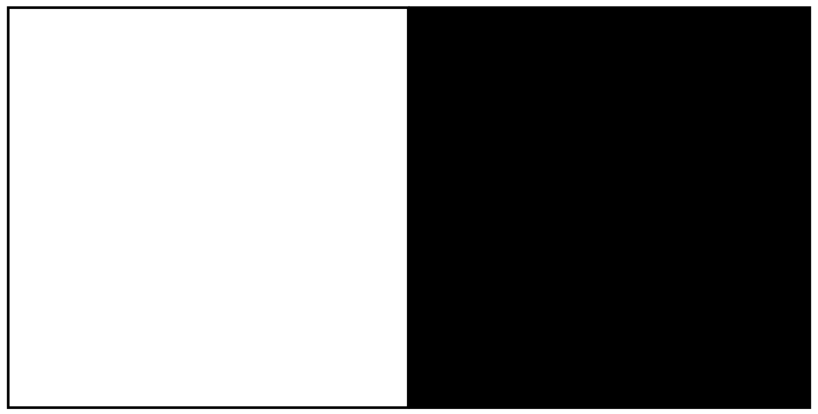
$$|\nabla f_{(x_i,y_j)}| = \sqrt{\frac{\partial f_{(x_i,y_j)}}{\partial x}^2 + \frac{\partial f_{(x_i,y_j)}}{\partial y}^2}$$

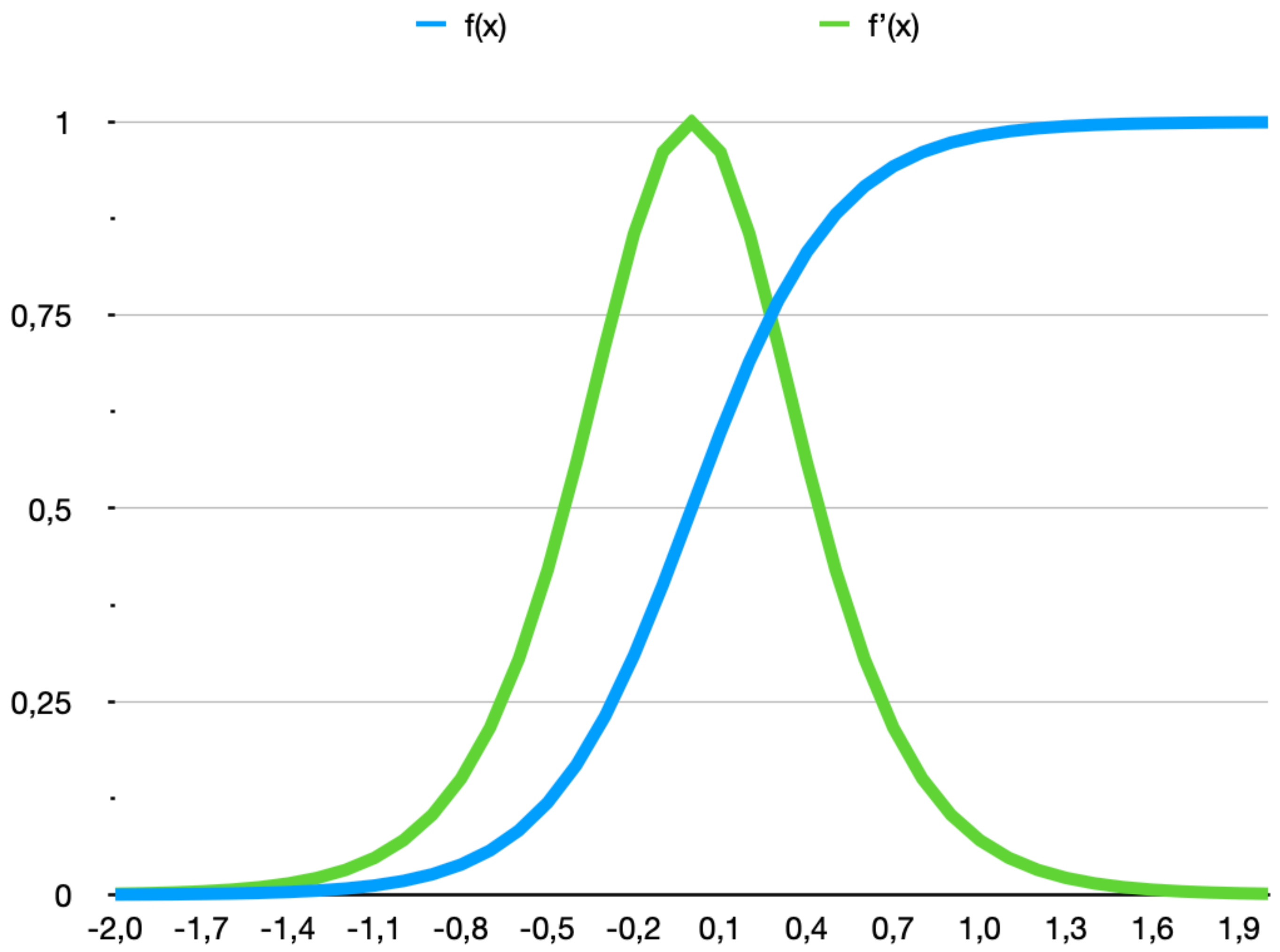
$$\theta = tan^{-1}\left(\frac{\partial f_{(x_i,y_j)}/\ \partial y}{\partial f_{(x_i,y_j)}/\ \partial x}\right)$$

Image source: [6]

Derivative and differencing

Derivative - idea of application in image edge detection





Derivative and differencing

Numerical differentiation - discrete gradient

Differentiation is the action of computing a derivative.

In numerical analysis, numerical differentiation describes algorithms for estimating the derivative.

Typically in numerical differentiation, **finite difference** is often used as an approximation of the derivative.

Three basic types of finite differences are commonly considered: forward, backward, and central.

Derivative and differencing

Numerical differentiation - discrete gradient

A **forward difference**, denoted $\Delta_h[f]$ of a function f is a function defined as:

$$\Delta_h[f](x) = f(x+h) - f(x)$$

Depending on the application, the spacing h may be variable or constant. When omitted, h is taken to be 1; that is,

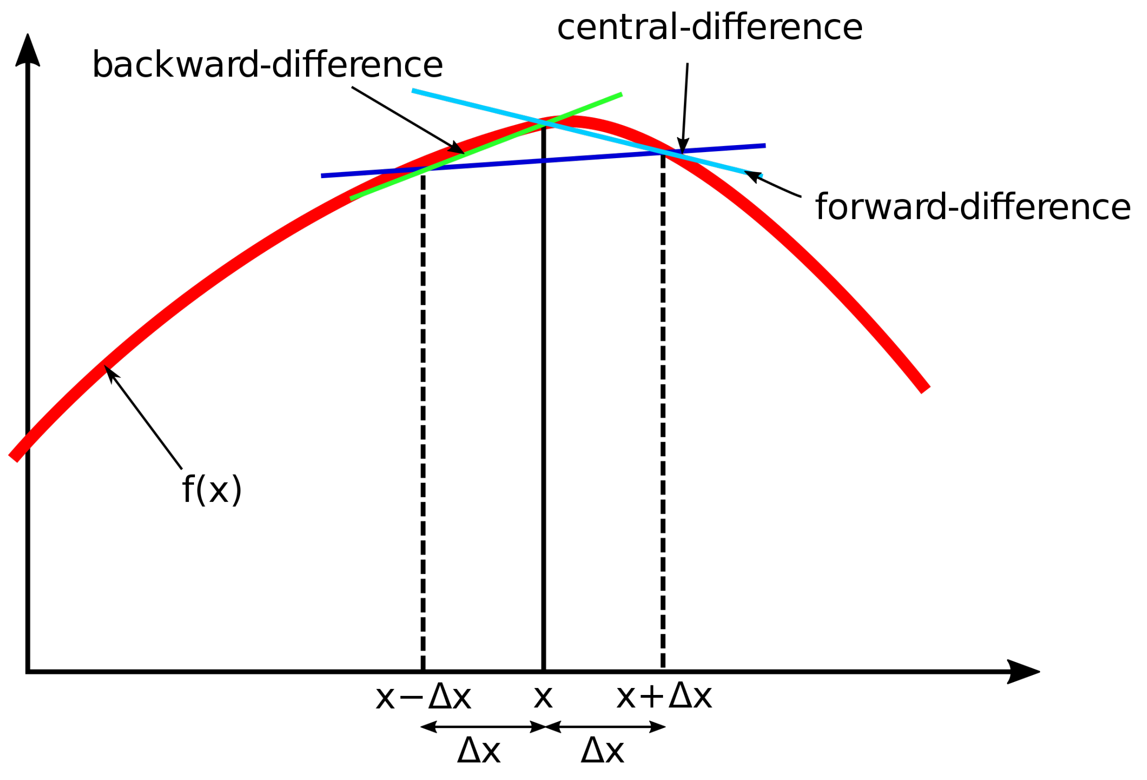
$$\Delta[f](x) = \Delta_1[f](x) = f(x+1) - f(x).$$

A **backward difference** uses the function values at x and $x - h$, instead of the values at $x + h$ and x:

$$\nabla_h[f](x) = f(x) - f(x-h) = \Delta_h[f](x-h).$$

Finally, the **central difference** is given by:

$$\delta_h[f](x) = f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right) = \Delta_h[f](x-\tfrac{h}{2})$$



Hence, the **differences divided by h approximates** the derivative when h is smal

Derivative and differencing

Numerical differentiation - discrete gradient

The **forward difference** divided by h approximates the derivative when h is small. The error in this approximation can be derived from Taylor's theorem. Assuming that f is twice differentiable, we have:

$$\frac{\Delta_h[f](x)}{h} - f'(x) = O(h) \to 0 \quad \text{as } h \to 0.$$

The same formula holds for the **backward difference**:

$$\frac{\nabla_h[f](x)}{h} - f'(x) = O(h) \to 0 \quad \text{as } h \to 0.$$

However, the **central** (also called centered) **difference** yields a more accurate approximation. If f is three times differentiable then:

$$\frac{\delta_h[f](x)}{h} - f'(x) = O\left(h^2\right).$$

Edge

Edge

Definition

Many approaches to image interpretation are based on edges, since **analysis based on edge detection is insensitive to change in the overall illumination level**.

Edges are perhaps the low-level image features that are most obvious to human vision. They preserve significant features, so we can usually recognise what an image contains from its edge detected version.

Edge detection highlights image contrast. Detecting contrast, which is difference in intensity, can emphasise the boundaries of features within an image, since this is where image contrast occurs. **The boundary of an object is a step change in the intensity levels.**

To detect the edge position, you can use *first-order differentiation* since this emphasises *change* and gives no response when applied to signals that do not change.

Edge

Type of edges

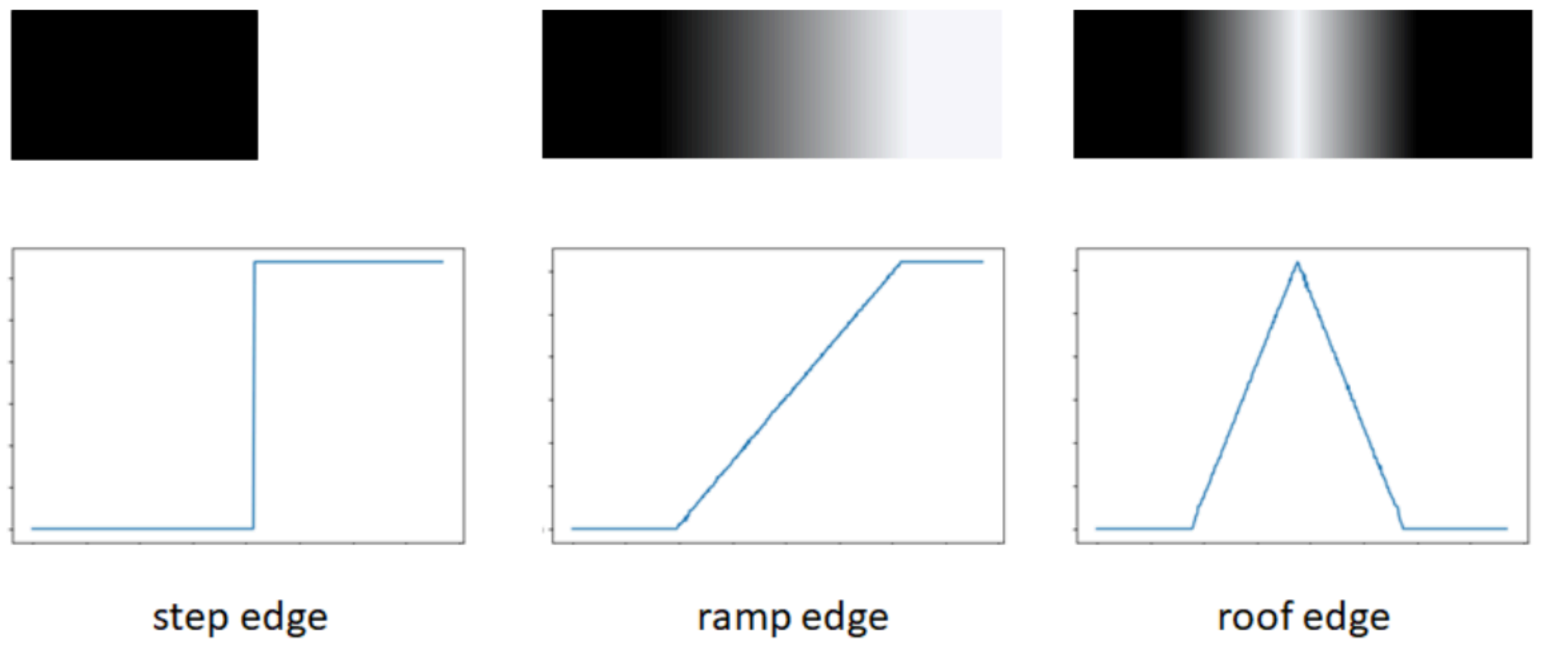


Image source: [6]

First-order differentiation

First-order differentiation

Basic idea

A change in intensity can be revealed by differencing adjacent points.

Differencing **horizontally adjacent points** will detect **vertical changes in intensity** and is often called a *horizontal edge detector* by virtue of its action. A horizontal operator will not show up horizontal changes in intensity since the difference is zero.

When applied to an image I of dimension n (width) and m (height) the action of the **horizontal edge detector forms the difference between two horizontally adjacent points**, as such **detecting the vertical edges**:

$$e_v = |I(x, y) - I(x + 1, y)|$$

for all $x = 1, \ldots, n - 1$ and $y = 1, \ldots, m$.

This method of differencing (when you consider point x and next point just after x which is $x + 1$) is called *forward differencing*.

First-order differentiation

Basic idea

Following the idea of detecting vertical edges you can easily get formula for detecting horizontal edges:

$$e_h = |\, I(x, y) - I(x, y+1) \,|\,.$$

Combining the horizontal and vertical detectors defines an operator e that can detect both vertical and horizontal edges together:

$$e = |\, I(x, y) - I(x+1, y) + I(x, y) - I(x, y+1) \,|$$

which gives:

$$e = |\, 2 \cdot I(x, y) - I(x+1, y) - I(x, y+1) \,|$$

From the last formula you can get the coefficients of a differencing template which can be convolved with an image to detect all the edge points:

2	-1
-1	0

As in the previous lectures, the reference point of a template (the position of the point you are computing a new value for) is marked with different color. Of course, the template shows only the weighting coefficients and not the modulus operation.

Roberts cross operator

Definition

The Roberts cross operator was one of the earliest edge detection operators (1963).

It implements a version of basic first-order edge detection and uses **two templates** which difference pixel values **in a diagonal manner**, as opposed to along the axes' directions proposed in previous part. The two templates are defined as:

$M^{-} =$

1	0
0	-1

$M^{+} =$

0	1
-1	0

What to do with values calculated based on these two templates? As the value of the edge at the point you can, for example:

- use the square root of the sum of values raised to the power of 2,
- use the sum of the absolute values of results of the two templates together,
- use the maximum of the absolute values delivered by application of these templates,
- use any other approach you think is reasonable.

Roberts cross operator

Definition

Roberts cross operator is less resistant to noise, but has narrower edges.

Prewitt edge detection operator

Prewitt edge detection operator

Definition

The Prewitt operator uses two 3×3 kernels to calculate approximations of the derivatives -- one for horizontal changes, and one for vertical:

$M_x =$

1	0	-1
1	0	-1
1	0	-1

$M_y =$

1	1	1
0	0	0
-1	-1	-1

The Prewitt kernels can be decomposed as the products of an (one-dimensional) **averaging** and a **differentiation** kernel, for example:

$$M_x = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}.$$

Prewitt edge detection operator

Edge magnitude and direction

Incorporating averaging within the edge detection process makes this filter less susceptible to the influence of noise.

If you want to say a little bit more about edges, you can calculate edge magnitude and edge direction (gradient direction, so it is perpendicular to an edge) at point (x, y):

$$e_m(x, y) = \sqrt{M_x(x, y)^2 + M_y(x, y)^2},$$

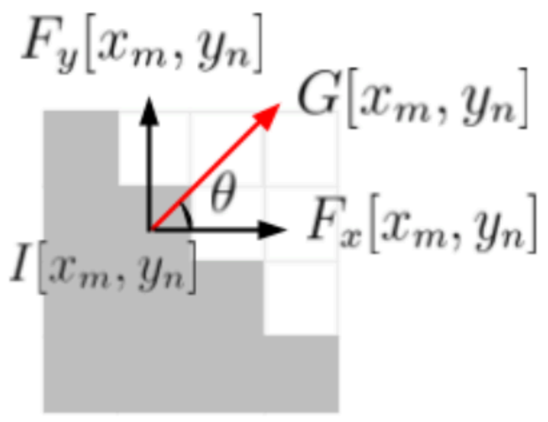
$$e_{dir}(x, y) = \tan^{-1}\left(\frac{M_y(x, y)}{M_x(x, y)}\right),$$

where M_x is the vertical template and M_y is the horizontal template.

The signs of M_x and M_y can be used to determine the appropriate quadrant for the edge direction or alternatively you can use a well known in low-level computer graphic function atan$2(y, x)$.

Prewitt edge detection operator

Edge magnitude and direction



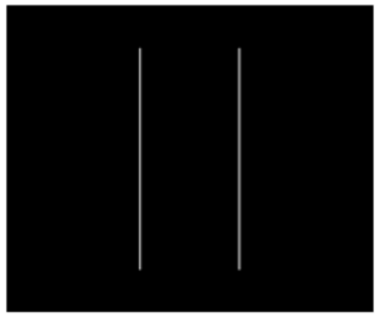
$$|G[x_m, y_n]| = \sqrt{F_x[x_m, y_n]^2 + F_y[x_m, y_n]^2}$$

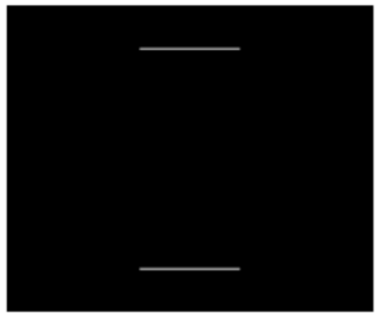
$$\theta = tan^{-1}(\frac{F_y[x_m, y_n]}{F_x[x_m, y_n]})$$

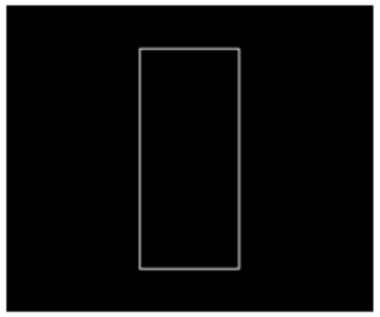
Image source: [6]

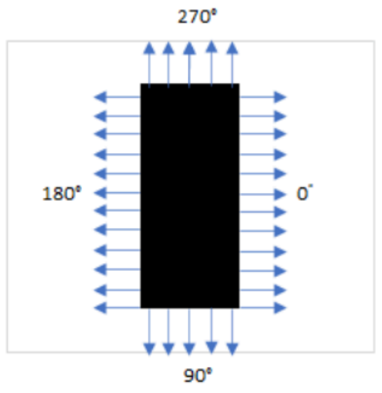
Prewitt edge detection operator

Edge magnitude and direction

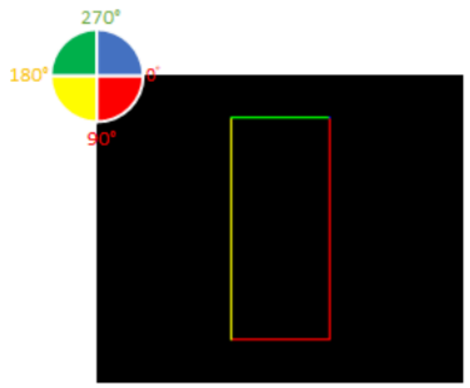

(b)


(c)


(d)



(e)



(f)

Image source: [6]

Prewitt edge detection operator

Edge magnitude and direction

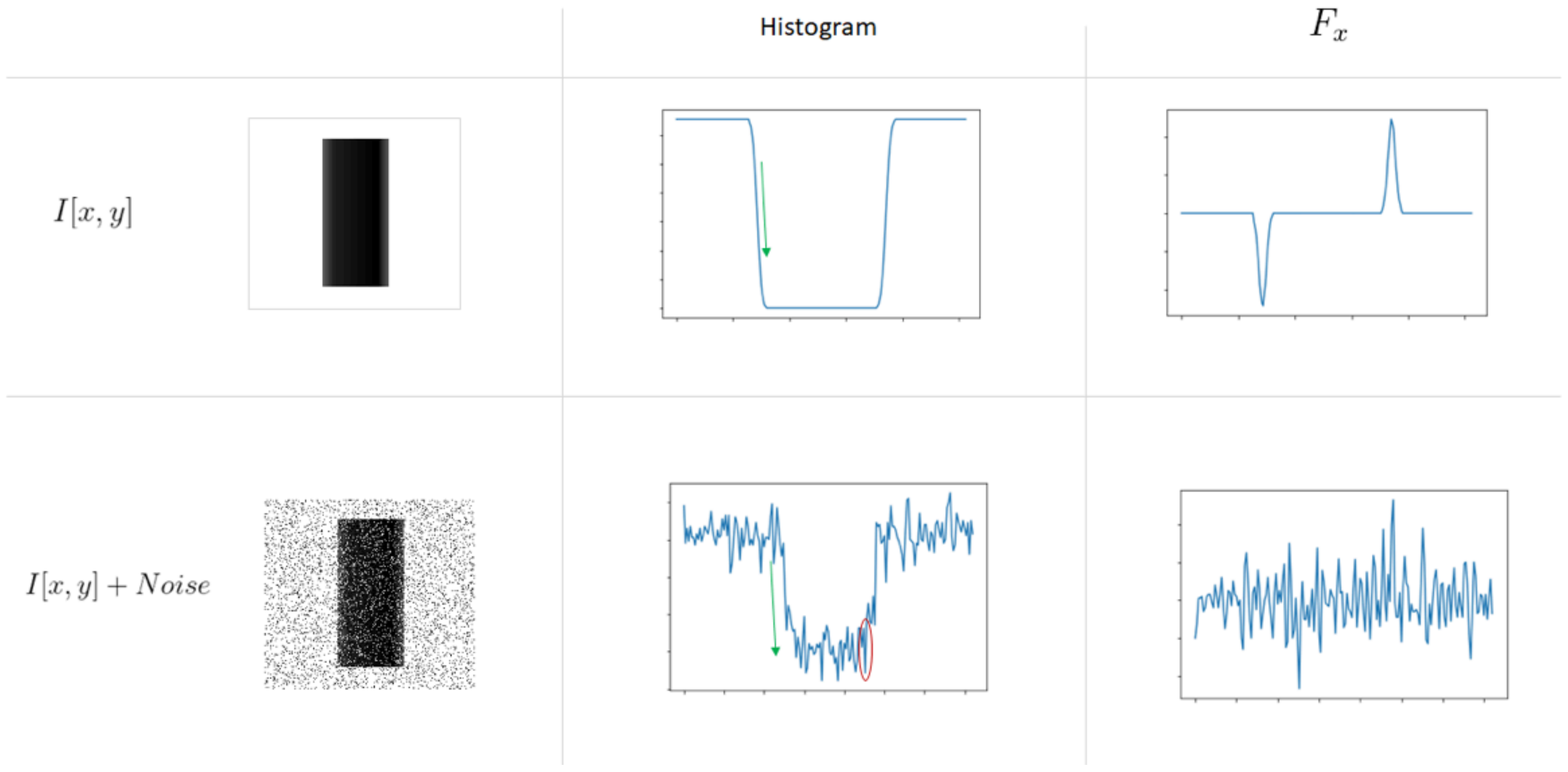


Image source: [6]

Prewitt edge detection operator

Code

```
import itertools

for x,y in itertools.product(range(0, width-1), range(0, height-1)):
  mX,mY = 0.0, 0.0

  for c in range(-1, 2):
    mX += float(inputImage[y + c, x - 1]) - float(inputImage[y + c, x + 1])
    mY += float(inputImage[y - 1, x + c]) - float(inputImage[y + 1, x + c])

  outputMagnitude[y,x] = math.sqrt(mX * mX + mY * mY)
  outputDirection[y,x] = math.atan2(mY, mX)
```

Sobel-Feldman edge detection operator

Sobel-Feldman edge detection operator

Definition

Technically, this operator, as all previous operators, is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function.

The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives -- one for horizontal changes, and one for vertical. Both are very similar to Prewitt operator but with stronger averaging component along x -axis and y-axis which result in stronger smoothing:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} +1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

Sobel-Feldman edge detection operator

Derive kernels

https://newbedev.com/sobel-filter-kernel-of-large-size

https://stackoverflow.com/questions/9567882/sobel-filter-kernel-of-large-size

Canny edge detection operator

Canny edge detection operator

Definition

In this case operator is rather a multistep algorithm.

It was formulated with three main objectives:

1. Detection of edge with **no false responses.**
2. The edge point detected from the operator should accurately **localize** on the **center of the edge**.
3. Single response to **eliminate multiple responses to a single edge**.

Canny edge detection operator

Definition

The first requirement aims to reduce the response to noise. This can be effected by optimal smoothing.

The second criterion aims for accuracy: edges are to be detected, in the right place. This can be achieved by a process of nonmaximum suppression (which is equivalent to peak detection). Nonmaximum suppression retains only those points at the top of a ridge of edge data, whilst suppressing all others. This results in thinning: the output of nonmaximum suppression is thin lines of edge points, in the right place.

The third constraint concerns location of a single-edge point in response to a change in brightness. This is because more than one edge can be denoted to be present, consistent with the output obtained by earlier edge operators.

Canny edge detection operator

Algorithm

The process of Canny edge detection algorithm can be broken down to the following steps:

1. Apply Gaussian filter to smooth the image in order to reduce the noise.

2. Use the Sobel operator (or any other you prefer) to find the intensity gradients of the image and its direction.

3. Apply nonmaximum suppression (or gradient magnitude thresholding or lower bound cut-off suppression or anything you think is a right tool) to get rid of spurious response to edge detection.

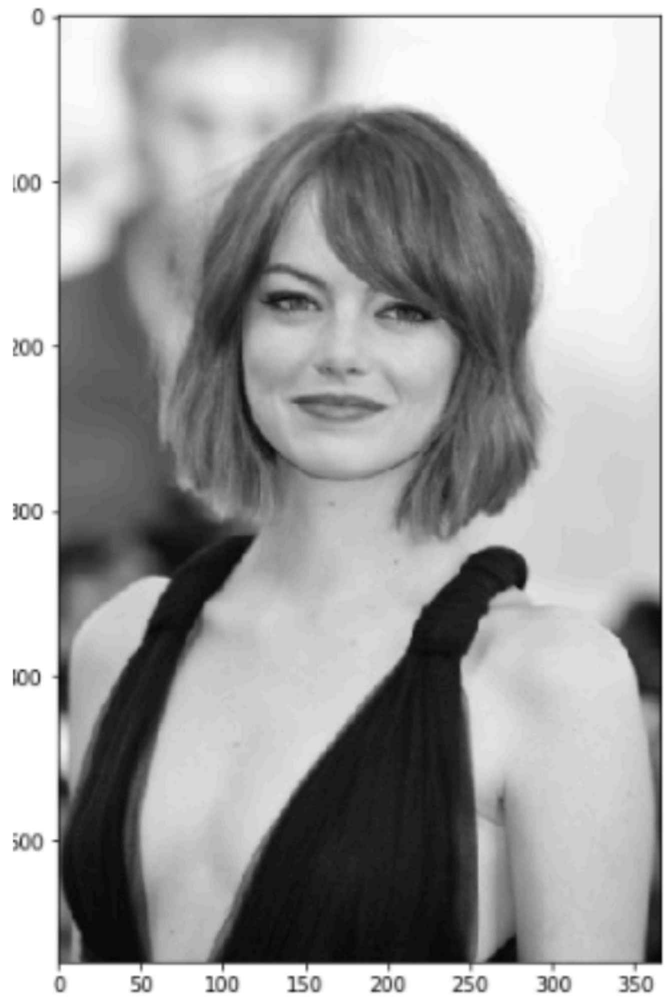
4. Apply double threshold to determine potential edges.

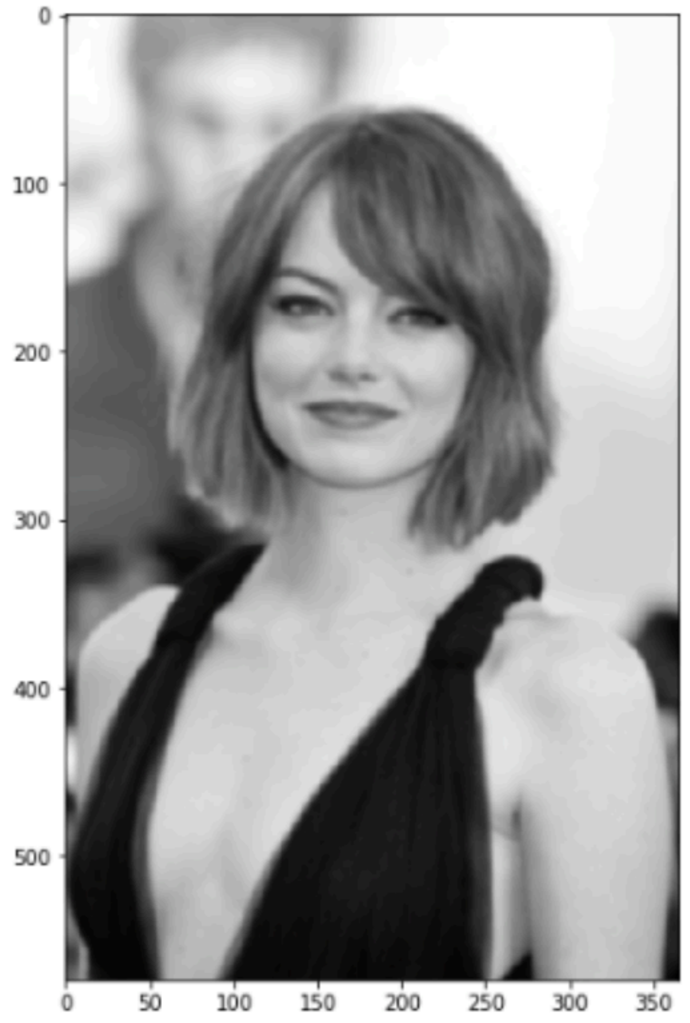
5. Threshold with hysteresis to suppress all the edges that are weak and not connected to strong edges.

Canny edge detection operator

Algorithm, step 1 [1]

Apply Gaussian filter to smooth the image in order to reduce the noise.

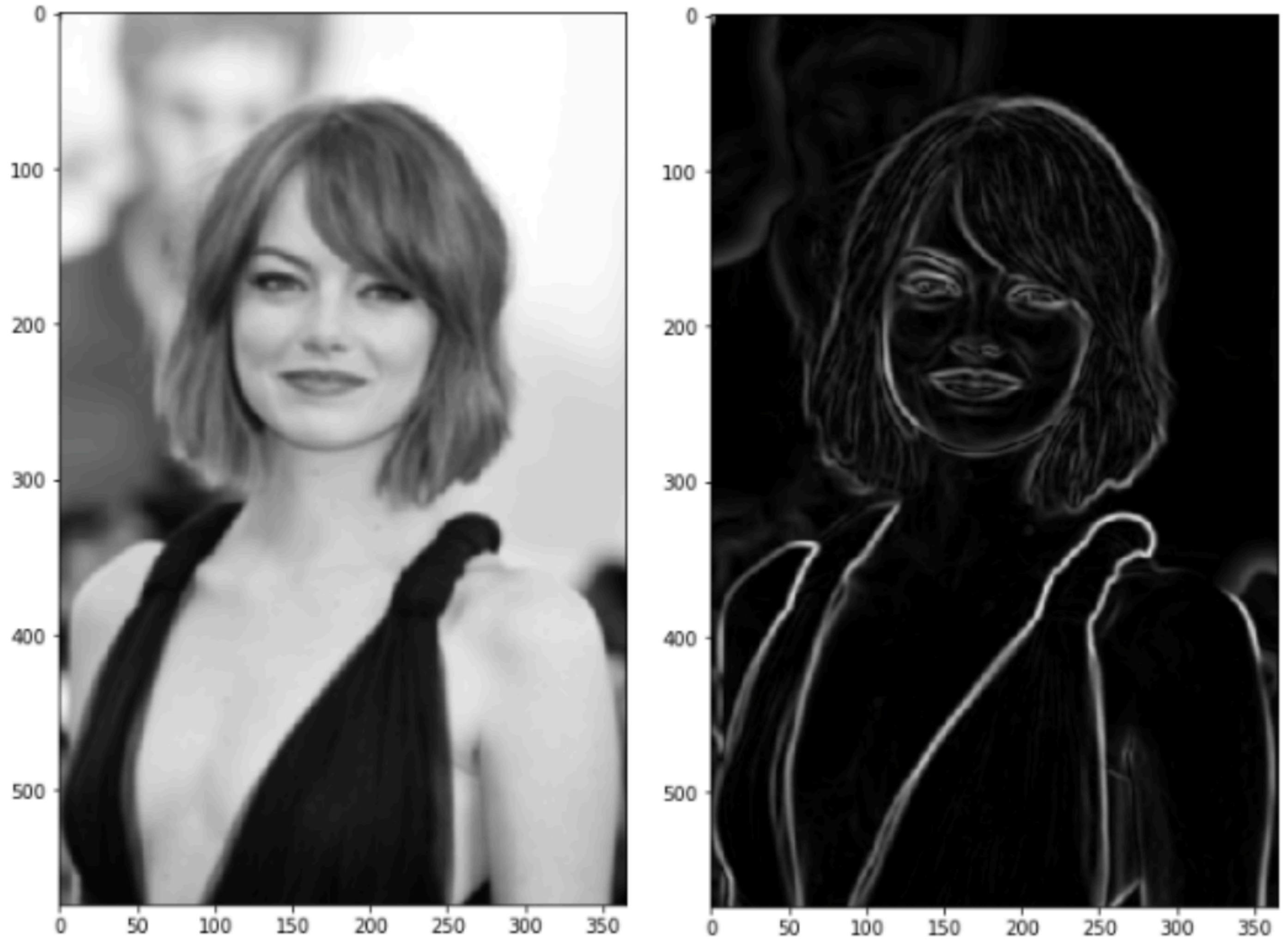




Canny edge detection operator

Algorithm, step 2 [1]

Use the Sobel operator (or any other you prefer) to find the intensity gradients of the image and its direction.



The result obtained in this step has two drawbacks:

1. Some of the edges are thick and others are thin. Non-Max Suppression step will help you mitigate the thick ones.

2. The gradient intensity level is between 0 and 255 which is not uniform. The edges on the final result should have the same intensity of value 255.

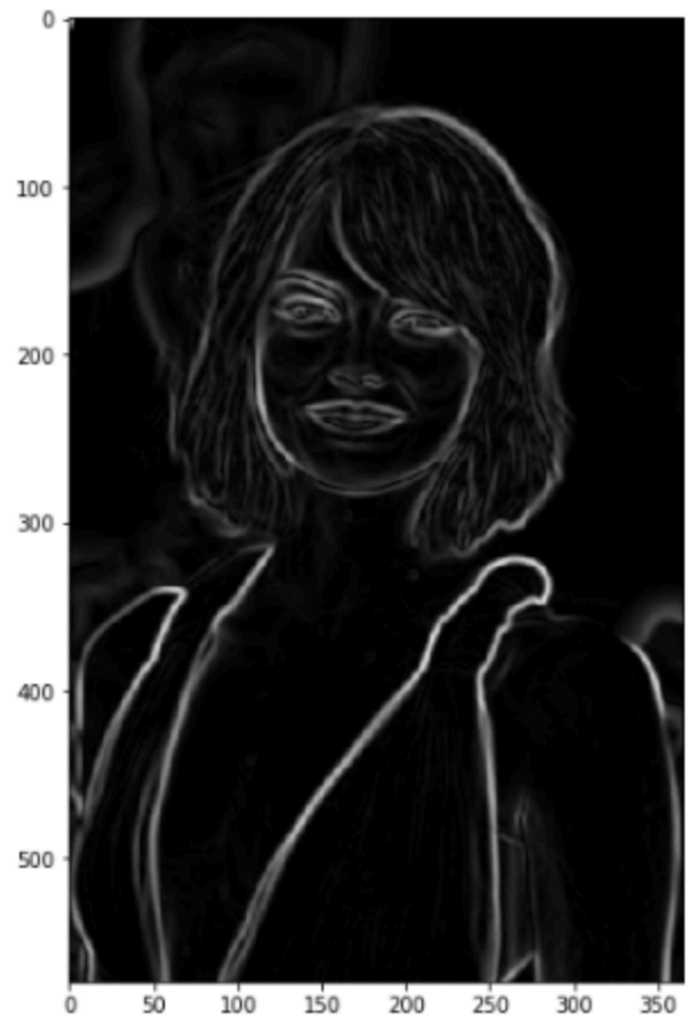
Canny edge detection operator

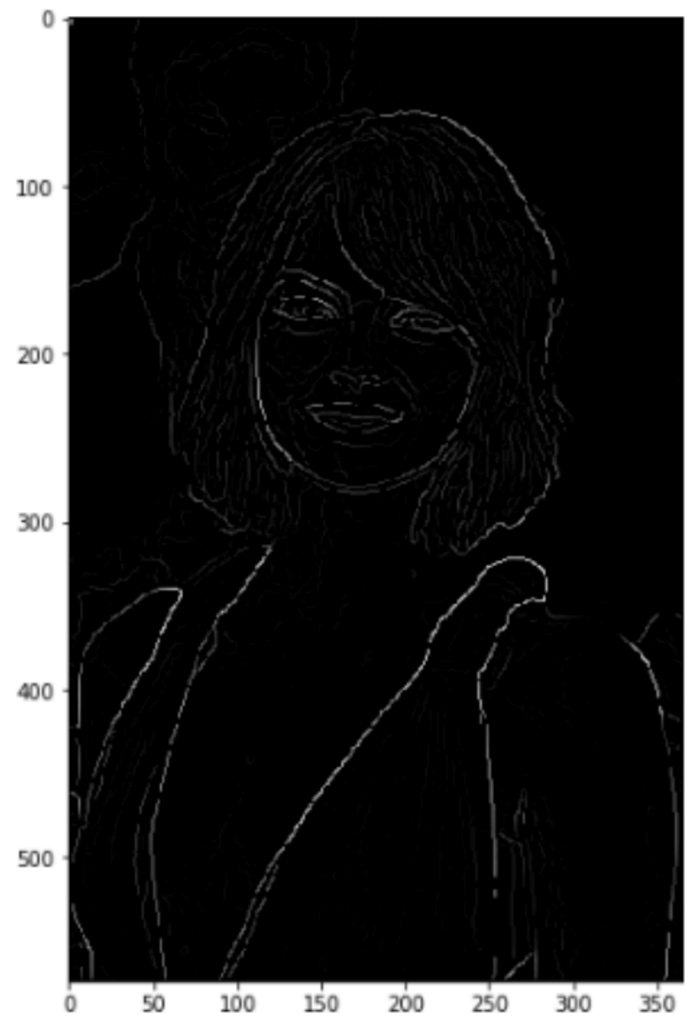
Algorithm, step 3 [1]

Perform non-maximum suppression to thin out the edges.

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a **local maximum in its neighborhood in the direction of gradient**.

In short, the result you get is a binary image with "thin edges".





The result is the same image with thinner edges. You can however still notice some variation regarding the edges' intensity: some pixels seem to be brighter than others, and you will try to cover this with the two final steps

Canny edge detection operator

Algorithm, step 4 [1]

Apply double threshold to determine potential edges.

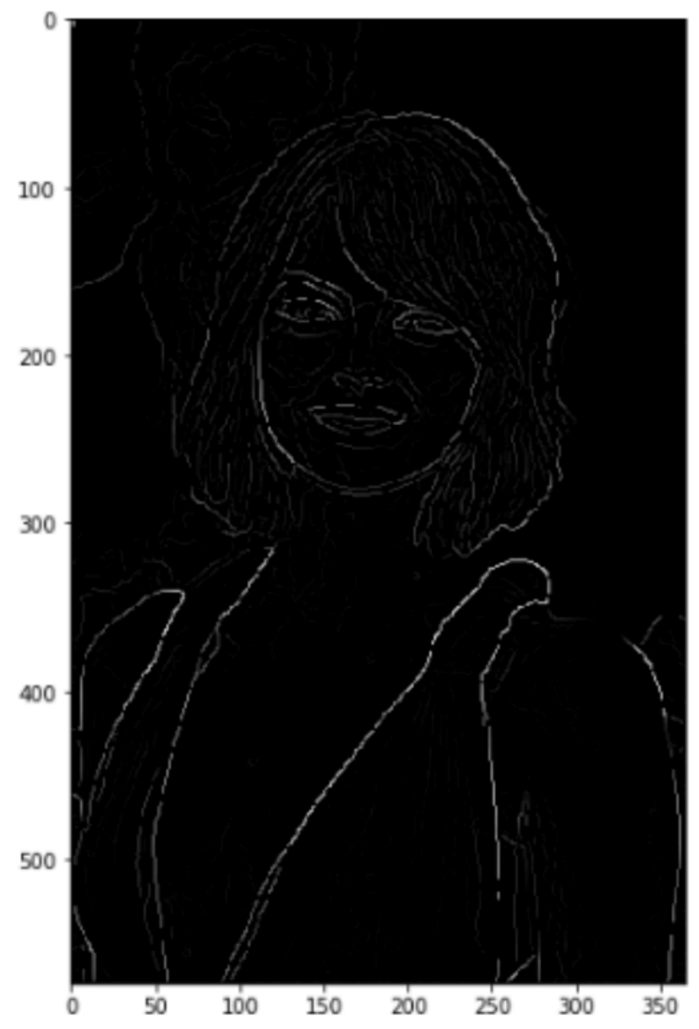
The double threshold step aims at identifying 3 kinds of pixels: strong, weak, and non-relevant:

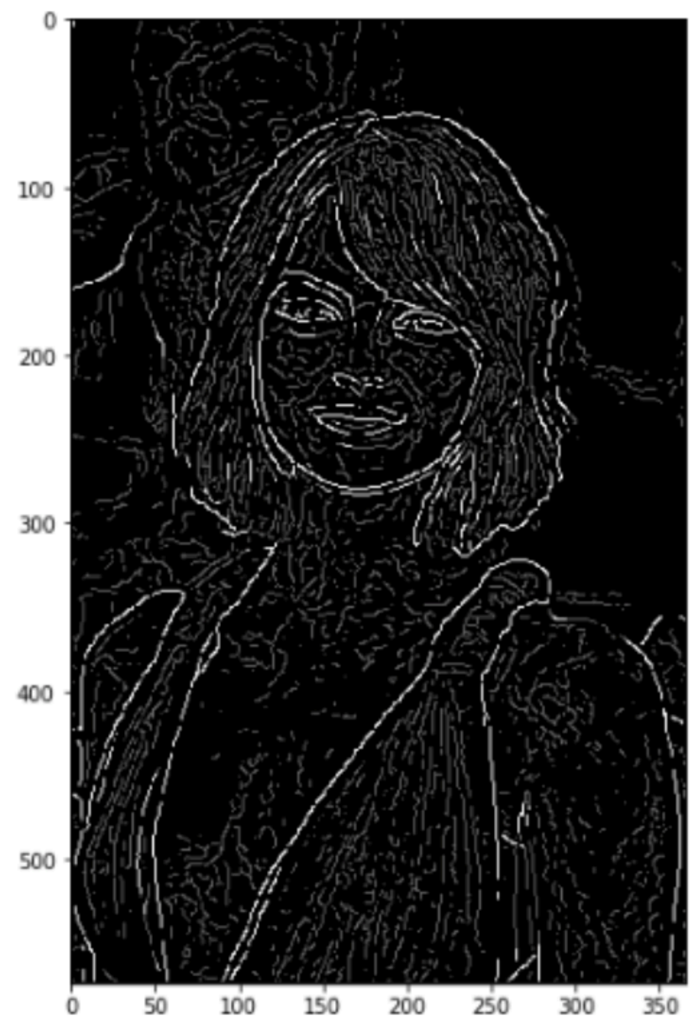
- Strong pixels are pixels that have an intensity so high that we are sure they contribute to the final edge.
- Weak pixels are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as non-relevant for the edge detection.
- Other pixels are considered as non-relevant for the edge.

Now you can see what the double thresholds holds for:

- High threshold is used to identify the strong pixels (intensity higher than the high threshold)
- Low threshold is used to identify the non-relevant pixels (intensity lower than the low threshold)
- All pixels having intensity between both thresholds are flagged as weak and the hysteresis mechanism (next step) will help us identify the ones that could be considered as strong and the ones that are considered as non-relevant.

The result of this step is an image with only 2 pixel intensity values (strong and weak).



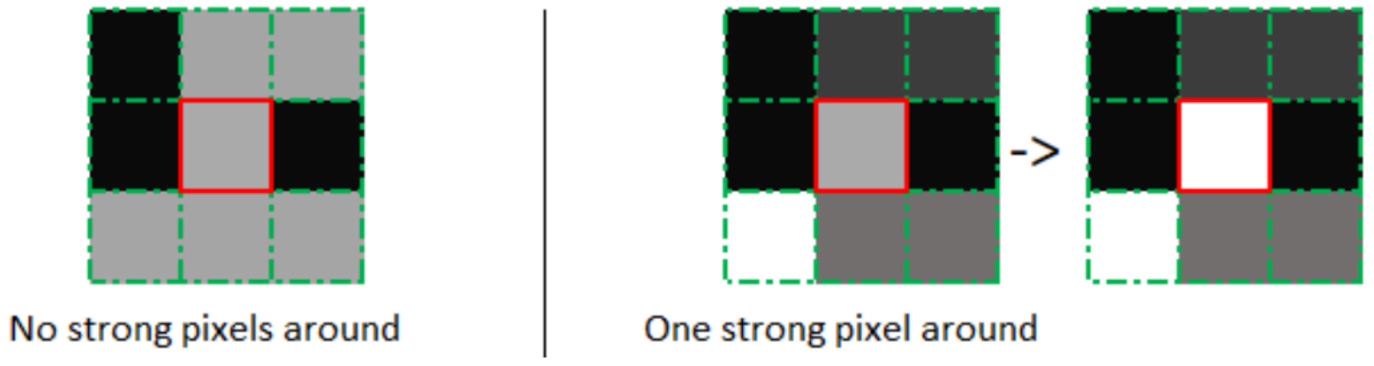


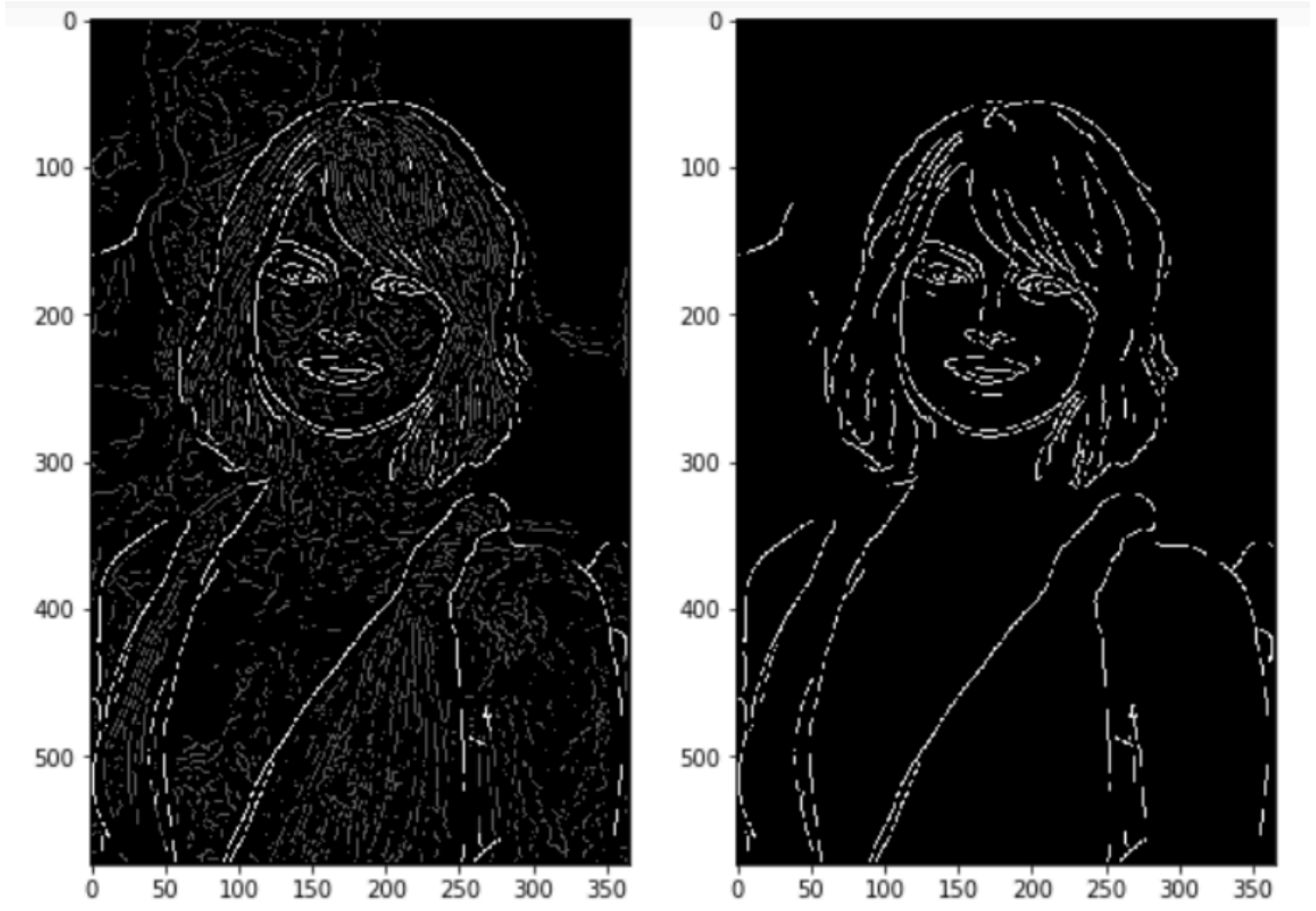
Canny edge detection operator

Algorithm, step 5 [1]

Threshold with hysteresis to suppress all the edges that are weak and not connected to strong edges.

Based on the threshold results, the hysteresis consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one.





Bibliography

Bibliography

1. Sofiane Sahir, *Canny Edge Detection Step by Step in Python — Computer Vision*, Jan 25, 2019, https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

2. Canny Edge Detection, https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html

3. CANNY EDGE DETECTION, http://justin-liang.com/tutorials/canny/

4. Zanurz się w Canny Edge Detection przy użyciu OpenCV-Python, https://ichi.pro/pl/zanurz-sie-w-canny-edge-detection-przy-uzyciu-opencv-python-129958935503260

5. Abhisek Jana, Implement Canny edge detector using Python from scratch, May 20, 2019, http://www.adeveloperdiary.com/data-science/computer-vision/implement-canny-edge-detector-using-python-from-scratch/

6. Image derivative, https://towardsdatascience.com/image-derivative-8a07a4118550