


浙江大学爱丁堡大学联合学院

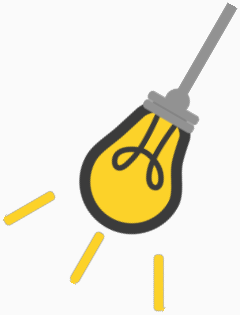
ZJU-UoE Institute

Lecture 5 - Edge detection

Nicola Romanò - nicola.romano@ed.ac.uk

Learning objectives

- Explain the use of image derivatives for edge detection
- Describe various edge detection algorithms
- Implement them in Python



Edge detection - introduction

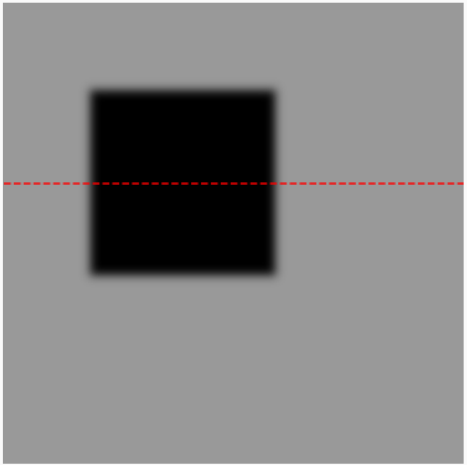
Edge detection problem

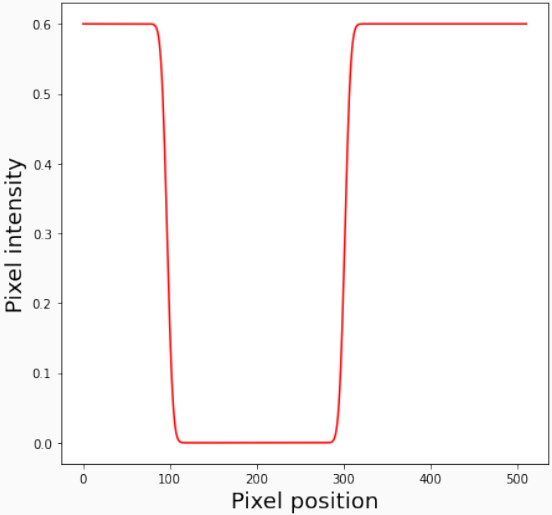
An edge is an area where the brightness of an image changes more or less gradually.

Detecting edges is useful e.g. to find objects in a scene, determine which pixels belong to which objects, and measure their properties.

Monarch butterfly - CC-BY-SA 2.0 Ted @ Flickr

How to detect an edge?

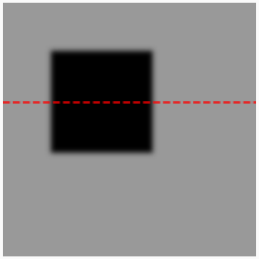


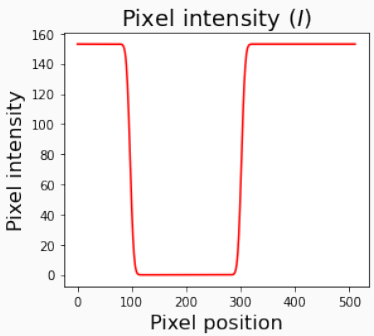


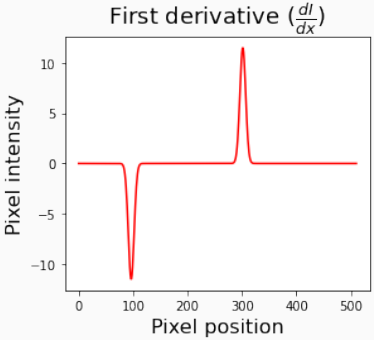
Consider the vertical edges. We can see a change in the intensity of pixels as we move from black to white and vice versa. **Can you think of a way to detect these edges?**

First derivatives approaches

We can use derivatives!







Edges will correspond to minima and maxima of the derivative of the intensity!

Image derivatives

With a 2D image, we can find the x and y derivatives of the image intensity $\left(\frac{\partial I}{\partial x} \text{and } \frac{\partial I}{\partial y}\right)$. We can combine the derivatives in a vector, called the **gradient**:

$$\nabla I = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]$$

Image derivatives

With a 2D image, we can find the x and y derivatives of the image intensity $\left(\frac{\partial I}{\partial x} \text{and } \frac{\partial I}{\partial y}\right)$. We can combine the derivatives in a vector, called the **gradient**:

$$\nabla I = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]$$

The gradient is a vector with:

- Direction perpendicular to the edge

$$\theta = \arctan\left(\frac{\partial I}{\partial x} \Big/ \frac{\partial I}{\partial y}\right)$$

Image derivatives

With a 2D image, we can find the x and y derivatives of the image intensity $\left(\frac{\partial I}{\partial x} \text{and } \frac{\partial I}{\partial y}\right)$. We can combine the derivatives in a vector, called the **gradient**:

$$\nabla I = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]$$

The gradient is a vector with:

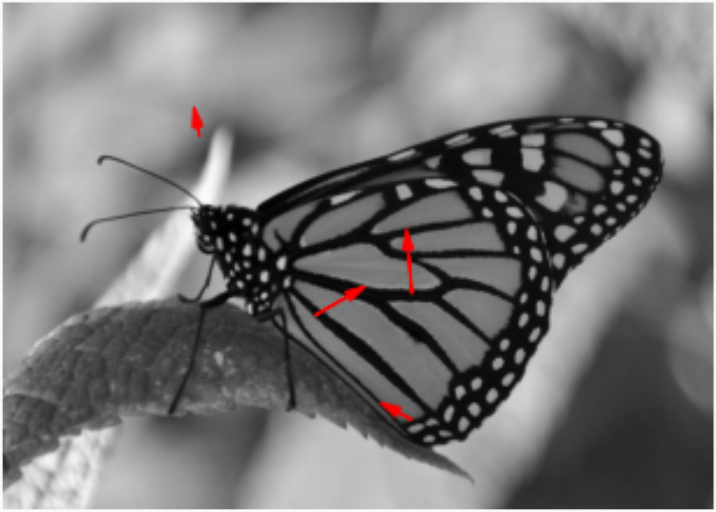
- Direction perpendicular to the edge

$$\theta = \arctan\left(\frac{\partial I}{\partial x} \,/\, \frac{\partial I}{\partial y}\right)$$

- Length (**gradient magnitude**) proportional to the intensity change

$$||\nabla I|| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

Example of gradient vectors



Calculating a discrete derivative

How to calculate a discrete derivative?

Remember the definition of a derivative for a continuous function $f(x)$:

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x}$$

Calculating a discrete derivative

How to calculate a discrete derivative?

Remember the definition of a derivative for a continuous function $f(x)$:

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x}$$

Our image is not continuous, as it is made up of discrete pixels so the minimum Δx value is 1 (a single pixel).

Calculating a discrete derivative

How to calculate a discrete derivative?

Remember the definition of a derivative for a continuous function $f(x)$:

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x}$$

Our image is not continuous, as it is made up of discrete pixels so the minimum Δx value is 1 (a single pixel).

The discrete derivative is given by:

$$\frac{dI}{dx} = \frac{I(x) - I(x-1)}{1} = I(x) - I(x-1)$$

Calculating the discrete derivative - variations

We can choose to calculate the discrete derivative in three different ways

- **Forward difference**: $I(x) - I(x+1)$
- **Backward difference**: $I(x) - I(x-1)$
- **Central difference**: $I(x+1) - I(x-1)$

Calculating the discrete derivative - variations

We can choose to calculate the discrete derivative in three different ways

- **Forward difference**: $I(x) - I(x+1)$
- **Backward difference**: $I(x) - I(x-1)$
- **Central difference**: $I(x+1) - I(x-1)$

These can be easily calculated using **convolution**!

- **Forward difference**: $\begin{bmatrix} 1 & -1 \end{bmatrix}$
- **Backward difference**: $\begin{bmatrix} -1 & 1 \end{bmatrix}$
- **Central difference**: $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$

Discrete derivative - example

Let's calculate the discrete derivative of this 1D image using central difference

10	16	22	36	40	11	17	23	37	41

Discrete derivative - example

Let's calculate the discrete derivative of this 1D image using central difference

10	16	22	36	40	11	17	23	37	41

Convolving with the kernel

-1	0	1

We obtain the following

0	12	20	18	-25	-23	12	20	18	0

Derivatives of an image by convolution

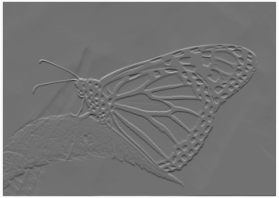
We can apply these convolution kernels to an image as well.

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

Original

Convolution with K_x

Convolution with K_y



Derivatives of an image by convolution

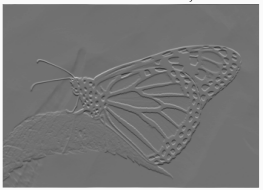
We can apply these convolution kernels to an image as well.

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

Original

Convolution with K_x

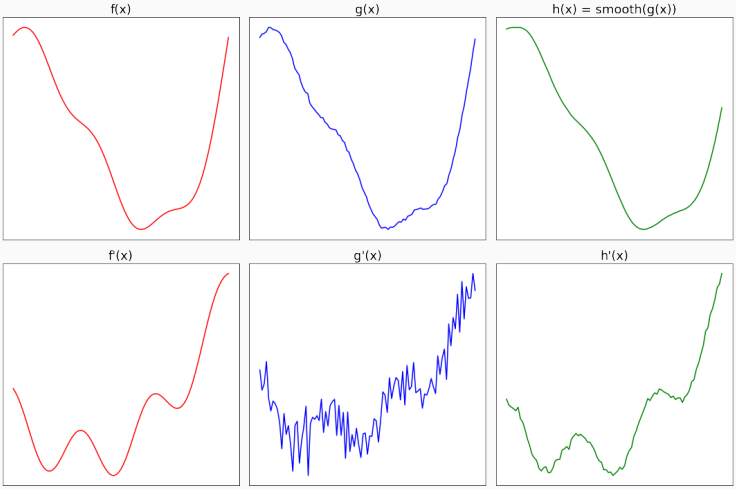
Convolution with K_y



Gradient magnitude

The problem with noise

Derivatives are very sensitive to noise. Smoothing the function beforehand helps



Smoothed 1st derivative kernels

We can smooth the derivative kernels by averaging nearby pixels.

These are called **Prewitt** kernels.

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

Alternatively the **Sobel** kernels can be used:

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Your turn!

Consider the following image. **What do you expect to obtain and why** after convolution with the Prewitt kernels?
Apply the convolution (you can easily do that by hand); **do the results match** your prediction?

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 50 & 50 & 50 & 50 & 50 & 50 & 50 & 50 \\ 100 & 100 & 100 & 100 & 100 & 100 & 100 & 100 \\ 100 & 100 & 100 & 100 & 100 & 100 & 100 & 100 \\ 50 & 50 & 50 & 50 & 50 & 50 & 50 & 50 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Edge detection - Prewitt and Sobel

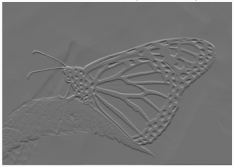
Having applied either the Prewitt or Sobel kernels to the image we can now detect edges.

Simply threshold the gradient magnitude of the image to define which pixels are edges.

Original image

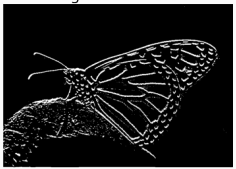
X derivative (Prewitt)

Y derivative (Prewitt)



Gradient magnitude

Magnitude > 0.04



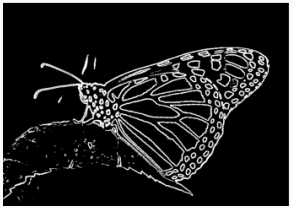
Edge detection in Scikit Image

```
from skimage.filters import prewitt, sobel
from skimage.io import imread

img = imread("butterfly.jpg")
im_prewitt = prewitt(img)
im_sobel = sobel(img)
```

Prewitt

Sobel



Edge detection in Scikit Image

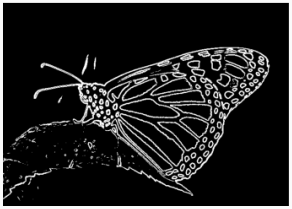
```
from skimage.filters import prewitt, sobel
from skimage.io import imread

img = imread("butterfly.jpg")
im_prewitt = prewitt(img)
im_sobel = sobel(img)

fig, ax = plt.subplots(2, 1, figsize=(5, 10))
ax[0].imshow(im_prewitt > 0.08, cmap="gray")
ax[0].set_title("Prewitt", fontsize=25)
ax[1].imshow(im_sobel > 0.08, cmap="gray")
ax[1].set_title("Sobel", fontsize=25)
for a in ax:
a.axis("off")
plt.show()
```

Prewitt

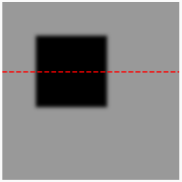
Sobel

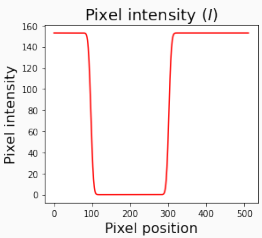


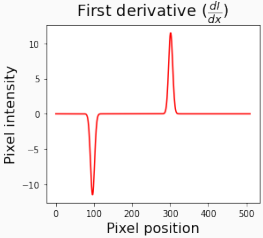
Second derivative approaches

Second derivative approaches

We can also use second derivatives to detect edges.

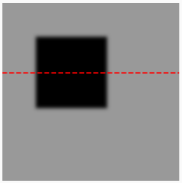


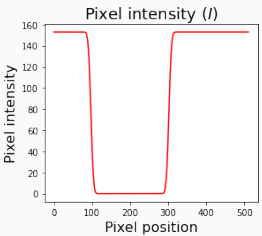




Second derivative approaches

We can also use second derivatives to detect edges. Edges correspond to **zero-crossings** of the second derivative.





The Laplacian

We can combine the second derivatives of the image using the **Laplacian** operator.

$$\nabla^2 I(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

The zero-crossing in the Laplacian are the **edges** of the image.

The Laplacian can be approximated by convolving with the kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Can you guess how we got to that?

Approximating the Laplacian

We calculated the first derivative of the image using the backward difference as:

$$\frac{\partial I}{\partial x} = I'_x = I_x - I_{x-1} \qquad \frac{\partial I}{\partial y} = I'_y = I_y - I_{y-1}$$

Approximating the Laplacian

We calculated the first derivative of the image using the backward difference as:

$$\frac{\partial I}{\partial x} = I'_x = I_x - I_{x-1} \qquad \frac{\partial I}{\partial y} = I'_y = I_y - I_{y-1}$$

Similarly, for the second derivative we can write:

$$\frac{\partial^2 I}{\partial x^2} = I''_x = I'_x - I'_{x-1} = [I_x - I_{x-1}] - [I_{x-1} - I_{x-2}] = I_x - 2I_{x-1} + I_{x-2}$$

Approximating the Laplacian

We calculated the first derivative of the image using the backward difference as:

$$\frac{\partial I}{\partial x} = I'_x = I_x - I_{x-1} \qquad \frac{\partial I}{\partial y} = I'_y = I_y - I_{y-1}$$

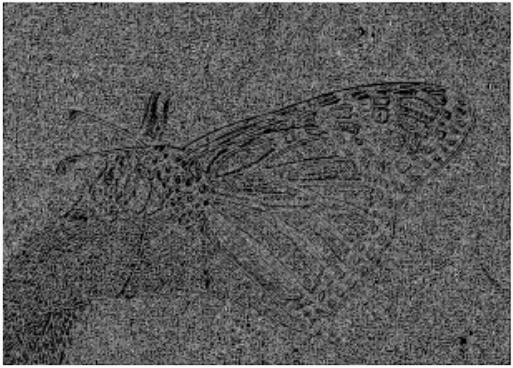
Similarly, for the second derivative we can write:

$$\frac{\partial^2 I}{\partial x^2} = I''_x = I'_x - I'_{x-1} = [I_x - I_{x-1}] - [I_{x-1} - I_{x-2}] = I_x - 2I_{x-1} + I_{x-2}$$

Applying the same reasoning to the y derivative, we obtain these kernels:

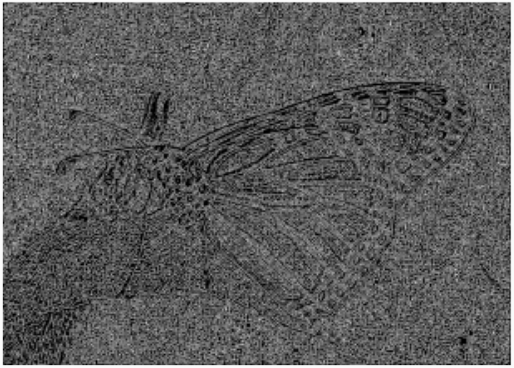
$$\begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \text{ which sum to } \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The Laplacian is very sensitive to noise



Zero crossing of the Laplacian of our butterfly image

The Laplacian is very sensitive to noise



Zero crossing of the Laplacian of our butterfly image

Challenge: can you try and reproduce this image?
Hint: use `skimage.filters.edges.convolve` *to apply the Laplacian kernel, then look at the sign of the result using the* `np.sign` *function... and think how you can define a zero-crossing!*

Laplacian vs Sobel/Prewitt operators

- Both operators are sensitive to noise (Laplacian is more sensitive).
- The Laplacian uses a single kernel, while the Sobel and Prewitt operators use two kernels.
- This means that the Laplacian loses orientation information.
- The Laplacian is an **isotropic filter**, meaning that it is invariant to the direction of the gradient (i.e. it performs well in all edge directions).
- Generally, Laplacian gives better edge localization

Smoothing the Laplacian

When using the Laplacian, most often we first smooth the image using a Gaussian filter, then convolve with the Laplacian kernel.

This can be done in one step since:

$$\nabla^2(G_\sigma * I) = \nabla^2(G_\sigma) * I$$

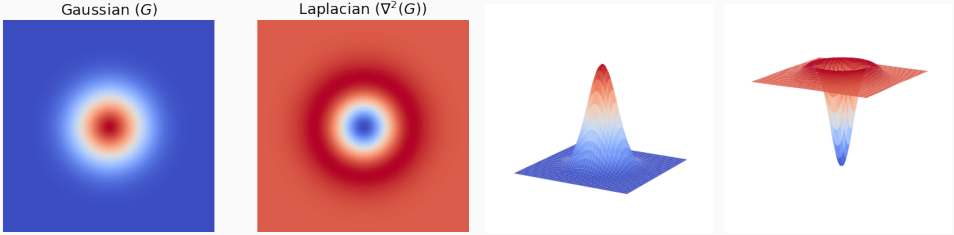
Smoothing the Laplacian

When using the Laplacian, most often we first smooth the image using a Gaussian filter, then convolve with the Laplacian kernel.

This can be done in one step since:

$$\nabla^2(G_\sigma * I) = \nabla^2(G_\sigma) * I$$

This is called the Laplacian of Gaussian (**LoG**) filter.



Approximation of the LoG

The Laplacian of a Gaussian function is given by:

$$LoG(x, y) = -\frac{1}{\pi\sigma^4}[1 - \frac{x^2 + y^2}{2\sigma^2}] \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

(no need to remember this formula, I certainly don't!)

Approximation of the LoG

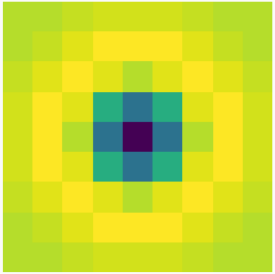
The Laplacian of a Gaussian function is given by:

$$LoG(x, y) = -\frac{1}{\pi\sigma^4}[1 - \frac{x^2 + y^2}{2\sigma^2}] \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

(no need to remember this formula, I certainly don't!)

This can be approximated by a LoG kernel. For example, for $\sigma = 1.4$ we can approximate LoG by:

$$\begin{bmatrix} 0 & 0 & 1 & 2 & 2 & 2 & 1 & 0 & 0 \\ 0 & 1 & 3 & 5 & 5 & 5 & 3 & 1 & 0 \\ 1 & 3 & 5 & 3 & 0 & 3 & 5 & 3 & 1 \\ 2 & 5 & 3 & -12 & -23 & -12 & 3 & 5 & 2 \\ 2 & 5 & 0 & -23 & -40 & -23 & 0 & 5 & 2 \\ 2 & 5 & 3 & -12 & -23 & -12 & 3 & 5 & 2 \\ 1 & 3 & 5 & 3 & 0 & 3 & 5 & 3 & 1 \\ 0 & 1 & 3 & 5 & 5 & 5 & 3 & 1 & 0 \\ 0 & 0 & 1 & 2 & 2 & 2 & 1 & 0 & 0 \end{bmatrix}$$



Approximation of the LoG

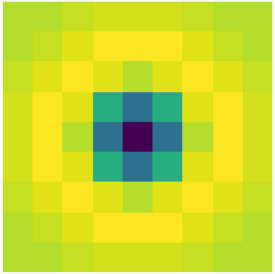
The Laplacian of a Gaussian function is given by:

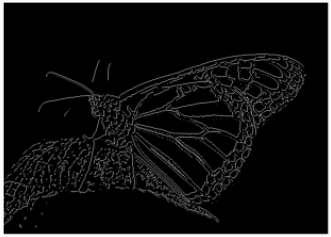
$$LoG(x, y) = -\frac{1}{\pi\sigma^4}[1 - \frac{x^2 + y^2}{2\sigma^2}] \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

(no need to remember this formula, I certainly don't!)

This can be approximated by a LoG kernel. For example, for $\sigma = 1.4$ we can approximate LoG by:

$$\begin{bmatrix} 0 & 0 & 1 & 2 & 2 & 2 & 1 & 0 & 0 \\ 0 & 1 & 3 & 5 & 5 & 5 & 3 & 1 & 0 \\ 1 & 3 & 5 & 3 & 0 & 3 & 5 & 3 & 1 \\ 2 & 5 & 3 & -12 & -23 & -12 & 3 & 5 & 2 \\ 2 & 5 & 0 & -23 & -40 & -23 & 0 & 5 & 2 \\ 2 & 5 & 3 & -12 & -23 & -12 & 3 & 5 & 2 \\ 1 & 3 & 5 & 3 & 0 & 3 & 5 & 3 & 1 \\ 0 & 1 & 3 & 5 & 5 & 5 & 3 & 1 & 0 \\ 0 & 0 & 1 & 2 & 2 & 2 & 1 & 0 & 0 \end{bmatrix}$$





Canny edge detection

The Canny edge detector

The Canny edge detector is a more advanced algorithm to detect edges.

It involves five steps

1. Apply a Gaussian filter to the image to smooth out noise
2. Calculate the gradient magnitude (e.g. using a Sobel filter)
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysteresis

The Canny edge detector - step 1 and 2 - smoothing and gradient

We start by convolving the image with a Gaussian kernel to smooth noise.

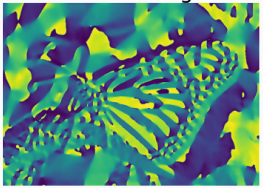
We then calculate the gradient magnitude and angle using the Sobel kernels.

Original image

Gaussian smoothing

Gradient magnitude

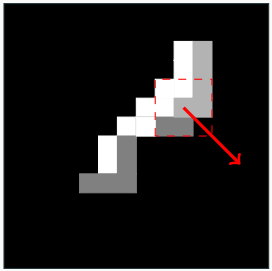
Gradient angle

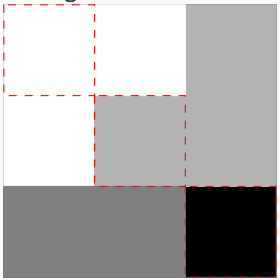


The Canny edge detector - step 3 - non-maximum suppression

Non-maximum suppression allows us to thin the edges by only keeping the pixels with the largest gradient magnitude in the edge.

For each pixel, we take the neighbouring pixels in the direction of the gradients, and we keep only the pixels with the largest gradient magnitude.

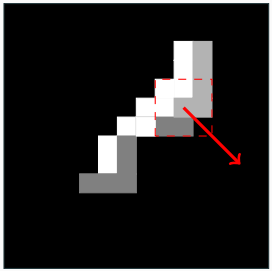


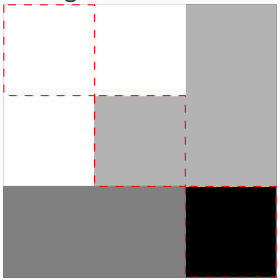


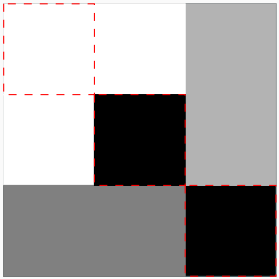
The Canny edge detector - step 3 - non-maximum suppression

Non-maximum suppression allows us to thin the edges by only keeping the pixels with the largest gradient magnitude in the edge.

For each pixel, we take the neighbouring pixels in the direction of the gradients, and we keep only the pixels with the largest gradient magnitude.







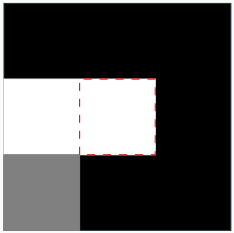
The Canny edge detector - step 4 - double thresholding

The next step is to set two arbitrary thresholds, one for the weak edges and one for the strong edges.

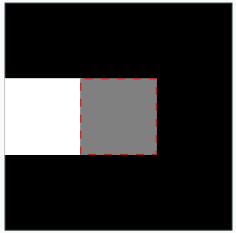
- **Strong edges** are those with gradient magnitude above the high threshold.
- **Weak edges** are those with gradient magnitude between the low and high threshold.
- Edges with gradient magnitude below the low threshold are suppressed (set to 0).

The Canny edge detector - step 5 - hysteresis

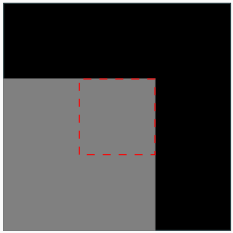
We need to decide what to do with weak edges. We keep those weak edges that are near a strong edge, and discard the others.



Strong edge, keep



Weak edge near strong, mark as strong



Weak edge, remove

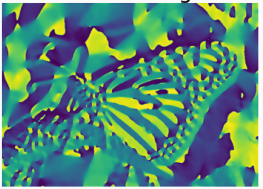
The Canny edge detector - The final result!

Original image

Gaussian smoothing

Gradient magnitude

Gradient angle



Non-max suppression

Double thresholding

Hysteresis

This is implemented in the skimage.feature.canny function. Try it by yourself and change the parameters to see what happens!

Want to read more? The original 1986 paper from Canny is attached (lots of maths in there!).

Summary

- Edge detection is a valuable tool for image processing.
- We have covered some of the most common algorithms for edge detection, which can be used as an early step in more complex analysis pipelines.
  - Prewitt and Sobel, based on the first derivative of the image.
  - The Laplacian of Gaussian (LoG) operator, used in second derivative edge detectors.
  - The Canny edge detector, still based on first derivatives, but with a more complex approach.
- In the next lecture we will look at other algorithms to detect specific features in images.

You can test the algorithms we have seen today here:

`https://apps.nicolaromano.net/EdgeDetection/`