# Summary CompVis

Adrian Willi

October 3, 2022

# Chapter 1

# Introduction to Image Processing

## 1.1 Basics

### 1.1.1 Grayscale images

Grayscale images are 2-dimensional numpy arrays. The values of the matrix represent the intensity of each pixel.
**Note:** our perceived brightness is mediated by our brain. It might not match the true intensity values in the image! Meaning that our brains are very good at ignoring intensity differences due to irrelevant things (eg. lighting, eg. shadows, etc.).

#### uint8 arithmetic

uint8 means simply one byte per pixel (values 0(black) - 255(white)). You need to be careful because uint8 arithmetic wraps around. When displaying uint8 grayscale images with `imshow` use always `vmin` and `vmax` parameters to set the colors that should map to black and white. Furthermore, we normally display grayscale images with a grayscale colormap. For **floating point images** we conventionally use a range from 0(black) to 1(white), unless specified otherwise.

```
print(np.array([0], dtype="uint8") - 1) # [255]

a = np.array([100], dtype="uint8")
print(a, a*2, a*3)   # [100] [200] [44]

plt.imshow(im,       # uint8 grayscale image
           vmin=0,   # value which maps to black
           vmax=255, # value which maps to white
           cmap="gray")

# floating point
im_float = im/255    # 0 stays 0; 255 becomes 1; is now of type float64
plt.imshow(im_float, cmap="gray", vmin=0, vmax=1)
```

#### Simple geometric transformations

```
plt.imshow(np.vstack((im_float, im_float)),
           cmap="gray", vmin=0, vmax=1) # stack images vertically; hstack for horizontally

plt.imshow(np.hstack((
                im_float,
                im_float * 2,   # changes contrast
                im_float + 0.5  # changes brightness
           )), cmap="gray", vmin=0, vmax=1)
```

### Handling color images

Color images are three-dimensional numpy array whose third dimension has length 3. These are called channels and correspond to red, green, blue intensities for each pixel. Sometimes there is an additional $\alpha$ channel which stands for the transparency. A pixel with the color red that is shown with full opacity would have a high $\alpha$ value.

```
# Visualize the red, green, blue channels as grayscale images
plt.imshow(np.hstack((
        im[:,:,0],   # red channel
        im[:,:,1],   # green channel
        im[:,:,2])), # blue channel
    vmin=0, vmax=255, cmap="gray")

# Display flags
# Russia
colors = ((255, 255, 255), (0, 0, 255), (255, 0, 0))
flag = np.vstack([np.full((200//3,300,3), c) for c in colors])
plt.imshow(flag)

# Switzerland
mask = np.zeros((200, 200))
mask[80:120, 40:160] = 1
mask[40:160, 80:120] = 2
flag = np.full((200, 200, 3), [255,0,0])
flag[mask==1] = [255, 255, 255]
flag[mask==2] = [255, 255, 255]
plt.imshow(flag)

# Japan
def hex2rgb(h):
    return [int(h[i:i+2], 16) for i in range(0, 6, 2)]

H, W = 200, 300
R = H * 3/5 / 2
Y, X = np.indices((H,W))
flag = np.ones((H, W, 3))
mask = (X-W/2)**2 + (Y-H/2)**2 < R**2
flag[mask] = [color / 255 for color in hex2rgb('bc002d')]
plt.imshow(flag, interpolation='bilinear')
```

### Handling video data

```
# Load video data
ims = []
for i in range(1,43):
    ims.append(skimage.io.imread(f"data/video_frames/{i:08d}.png"))
ims = np.array(ims)

# Isolate the moving person in the movie for each frame of the video
thresh = 25
len_frames = len(ims[:])
plt.figure(figsize=(20, 40))

for frame in range(len_frames):
    r = np.abs(ims[frame,:,:,0] - np.mean(ims[:,:,:,0], axis=0)) > thresh
    g = np.abs(ims[frame,:,:,1] - np.mean(ims[:,:,:,1], axis=0)) > thresh
    b = np.abs(ims[frame,:,:,2] - np.mean(ims[:,:,:,2], axis=0)) > thresh
```

```
diff = r | g | b
plt.subplot(int(np.ceil(len_frames/5)), 5, frame+1)
plt.imshow(diff, cmap="gray")
plt.tight_layout()
```

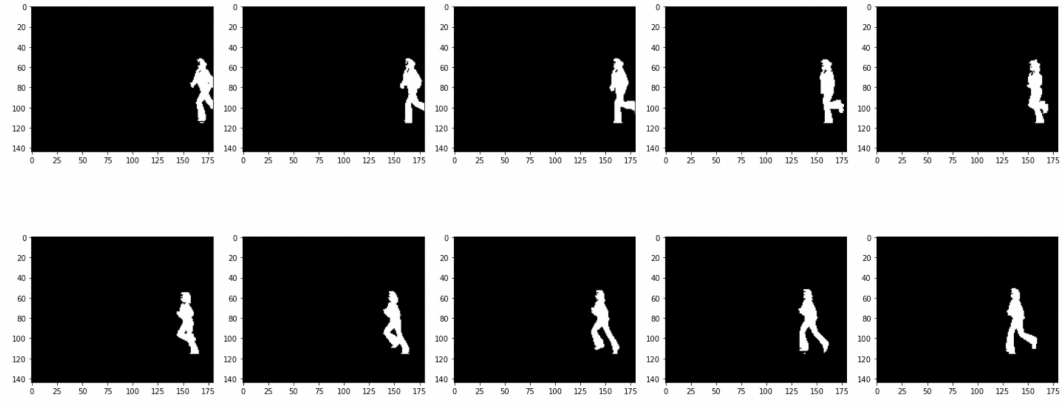

Figure 1.1: Isolated moving person frame wise

# Chapter 2

# Local Filtering and Edge Detection

## 2.1 Binarization and Connected Component Analysis

The goal is to find all coins in the picture. To achieve this a mask is created by using a certain threshold. Furthermore, the min size can be set in order to avoid that to small structures are recognized as a coin.

```
@widgets.interact(threshold = (0,1,0.01), minsize = (10,500))
def f(threshold=0.51, minsize=300):
    fig, ax = plt.subplots()
    ax.imshow(im, cmap="gray")
    mask = im > threshold
    labels = skimage.measure.label(mask)
    regions = skimage.measure.regionprops(labels)
    large_regions = [r for r in regions if r.area > minsize]
    for r in large_regions:
        (min_row, min_col, max_row, max_col) = r.bbox
        width = max_col - min_col
        height = max_row - min_row
        rect = patches.Rectangle((min_col,min_row),width,height,
                                 linewidth=1,edgecolor='b',facecolor='none')
        ax.add_patch(rect)
```

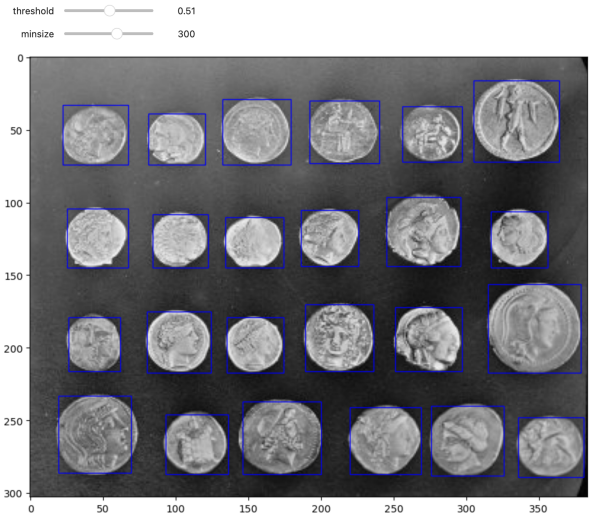


Figure 2.1: Find all coins using a threshold

## 2.2 Local filtering

### 2.2.1 Simple filtering in 1D: moving average

Replace every value with the average of the pixels in its neighborhood.

$b_{smooth}[i] = \frac{1}{2r+1} \sum_{j=i-r}^{i+r} b[j]$

### 2.2.2 Convolution

A convolution is a **weighted moving average**. The sequence of weights a[j] is called filter or convolution kernel. For example a bell-shaped kernel: $a = [1, 4, 6, 4, 1]/16$.

$(a * b)[i] = \sum_j a[j]b[i-j]$

#### 2D Convolution

Same as 1D, with one more index. Now the filter is a rectangle you slide around over a grid of numbers. Depending whether we want to keep the dimension of the input image we need to add some padding (mostly 0s) on the borders.

$(a * b)[i, j] = \sum_{i',j'} a[i', j']b[i - i', j - j']$

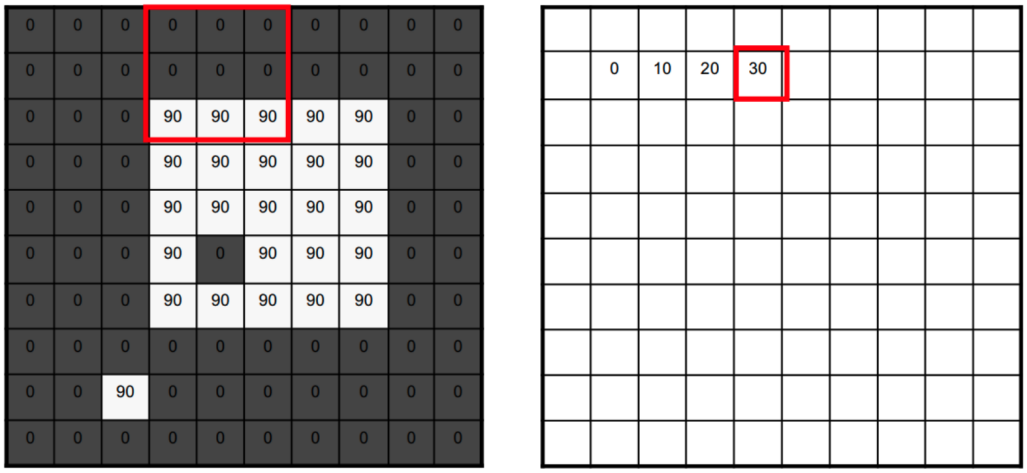


Figure 2.2: Example of a 2D convolution

#### Padding

When we don't use any padding then the output matrix will be smaller than the input matrix. In general, if the input matrix has $i$ rows and the kernel has $k$ rows, the output will have $i - k + 1$ rows and the same also applies to columns.
Often, we will use padding on the input image to get an output matrix that is the same size as the input. Then, there are different options to handle extra cells (see docs of the *mode* parameter of *scipy.ndimage.convolve*():

**'reflect' (*d c b a | a b c d | d c b a*)**

The input is extended by reflecting about the edge of the last pixel.

**'constant' (*k k k k | a b c d | k k k k*)**

The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

**'nearest' (*a a a a | a b c d | d d d d*)**

The input is extended by replicating the last pixel.

**'mirror' (*d c b | a b c d | c b a*)**

The input is extended by reflecting about the center of the last pixel.

**'wrap' (*a b c d | a b c d | a b c d*)**

The input is extended by wrapping around to the opposite edge.

Figure 2.3: Padding options for scipy.ndimage.convolve()

```
import scipy
a = np.array([   [3,3,2,1,0],
                 [0,0,1,3,1],
                 [3,1,2,2,3],
                 [2,0,0,2,2],
                 [2,0,0,0,1]])

k = np.array([   [0,1,2],
                 [2,2,0],
                 [0,1,2]])

scipy.ndimage.convolve(a, k[::-1,::-1], mode='constant', cval=0.0)
```

```
Mode =  reflect.        Mode =  nearest        Mode =  mirror
[[21 21 21 12  5]       [[21 21 21 12  5]      [[12 16 24 16 16]
 [14 12 12 17 17]        [14 12 12 17 17]       [14 12 12 17 17]
 [14 10 17 19 19]        [14 10 17 19 19]       [10 10 17 19 23]
 [15  9  6 14 20]        [15  9  6 14 20]       [11  9  6 14 16]
 [12  4  4  8 11]]       [12  4  4  8 11]]      [ 8  4  8 12 14]]

Mode =  wrap            Mode = constant, cval=0.0
[[ 8 14 17 13  8]       [[ 6 14 17 11  3]
 [16 12 12 17 23]        [14 12 12 17 11]
 [14 10 17 19 17]        [ 8 10 17 19 13]
 [15  9  6 14 22]        [11  9  6 14 12]
 [17 11  8  7 14]]       [ 6  4  4  6  4]]
```

### 2.2.3 Smoothing with box filter (moving average)

A *box filter* is a filter that just averages all values in the (square) neighborhood.

```
size = 3
kernel = np.full((size, size), 1/(size * size))
assert(np.isclose(np.sum(k), 1.0))
imf = scipy.ndimage.convolve(im, k[::-1,::-1], mode='constant', cval=0.0)
plt.imshow(imf, cmap='gray', vmin=0, vmax=1)
```

### 2.2.4 Smoothing with gaussian filters (weighted moving average)

Instead of considering all pixels in the neighborhood equally, it makes sense to implement a *weighted average* and give more weight to the central pixels. We can obtain that with a gaussian kernel.

Variance (sigma) determines the amount of smoothing. The larger sigma is the wider is the smoothed area. **Rule of thumb:** Set kernel size $K \approx 2\pi\sigma$ (also three times sigma). Due that Gaussian 2D filter is separable we can achieve the same by using two 1D Gaussian filters which are way faster. In 2-D, an isotropic Gaussian has the form:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

A discrete approximation to Gaussian function with $\sigma = 1.0$ looks like the following:

$\frac{1}{273}$

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 4 | 1 |
| 4 | 16 | 26 | 16 | 4 |
| 7 | 26 | 41 | 26 | 7 |
| 4 | 16 | 26 | 16 | 4 |
| 1 | 4 | 7 | 4 | 1 |

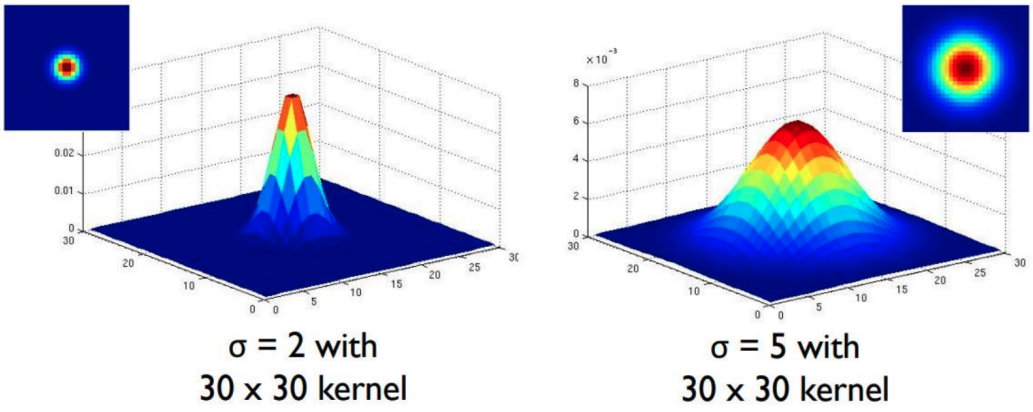


Figure 2.4: Example of Gaussian filter for different sigma

**Example of gaussian and box filter when adding some white pixels**

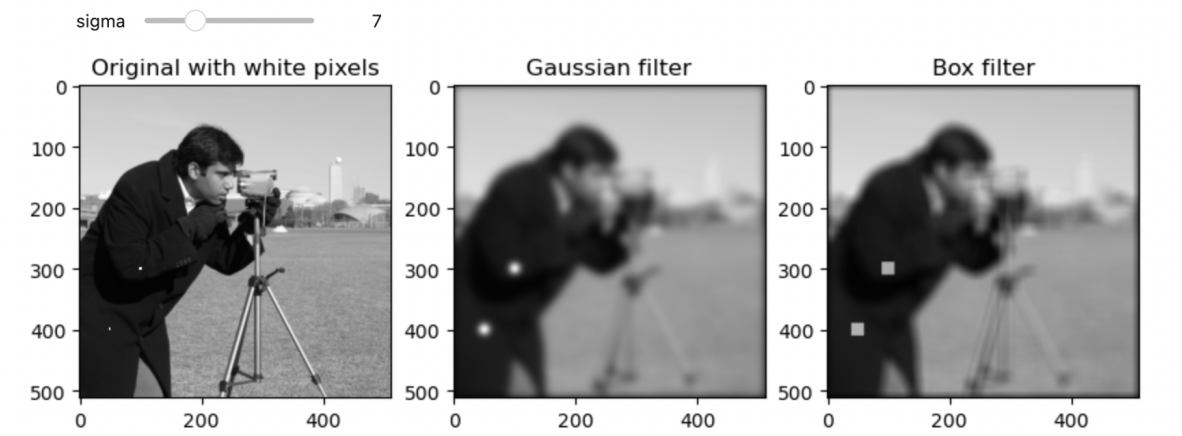


Figure 2.5: Example of gaussian and box filter when adding some white pixels

### 2.2.5 Sharpening

To sharpening an image we first smooth the image and then subtract it from the original image. With this we get the details. Then we just add this detail image to the original and obtain a sharpened image.

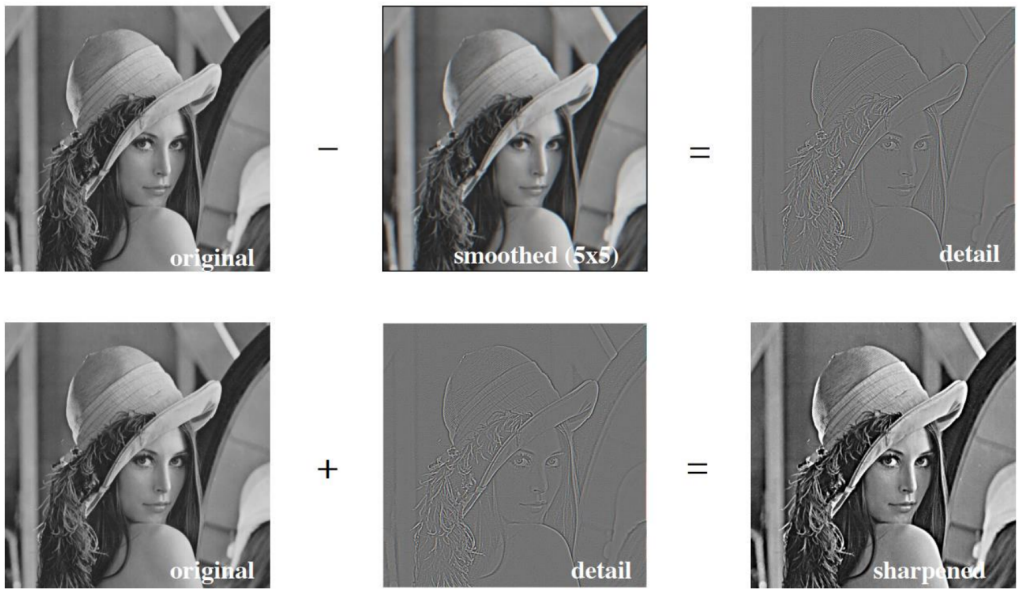


Figure 2.6: Process to obtain a sharpened image

## 2.3 Edge detection

Edges are important features when we look at images. There are many reasons for edges in images like: Reflectance change (appearance information, texture), Cast shadows, Change in surface orientation (shape) or Depth discontinuity (object boundary).

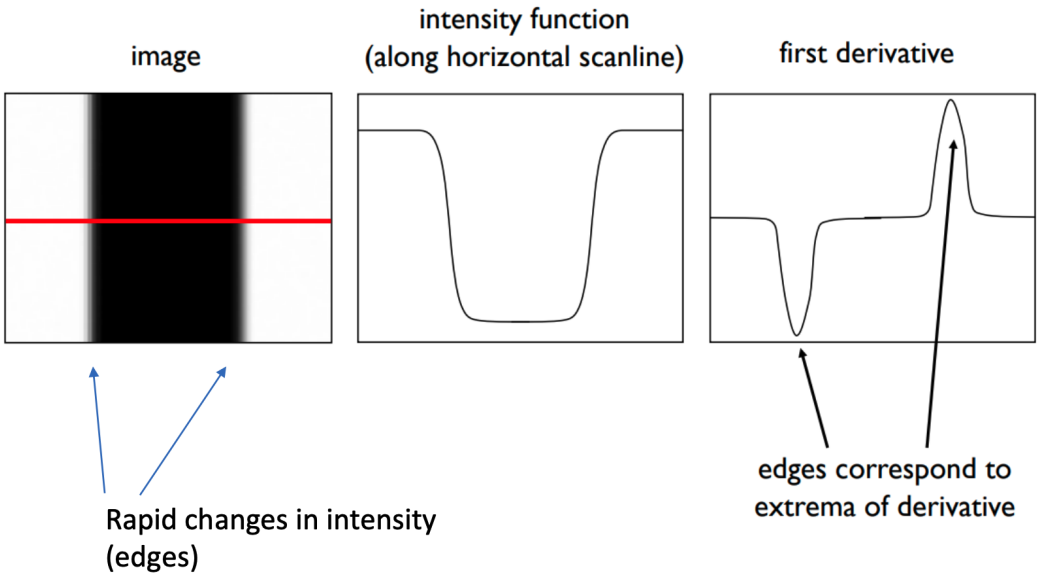


Figure 2.7: What is an edge in 1D?

### 2.3.1 Edges in 2D

In 2D, the derivative corresponds to the **gradient**. It points to the direction of most rapid increase of the intensity.

The gradient of an image: $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}]$

Gradient direction (orthogonal to the edge): $\theta = tan^{-1}(\frac{\partial f}{\partial x}/\frac{\partial f}{\partial y})$

Gradient magnitude (strength of the edge): $|\nabla f| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$

**Keep in mind:** Not all important edges have strong gradients. Not all strong gradients are important edges!

### 2.3.2 Computing the gradient on an image

We use finite differencing!

How do we approximate $\frac{\partial f}{\partial x}(\rightarrow)$? convolve with

| -1 | 1 |
|---|---|

How do we approximate $\frac{\partial f}{\partial y}(\downarrow)$? convolve with

| -1 |
|---|
| 1 |

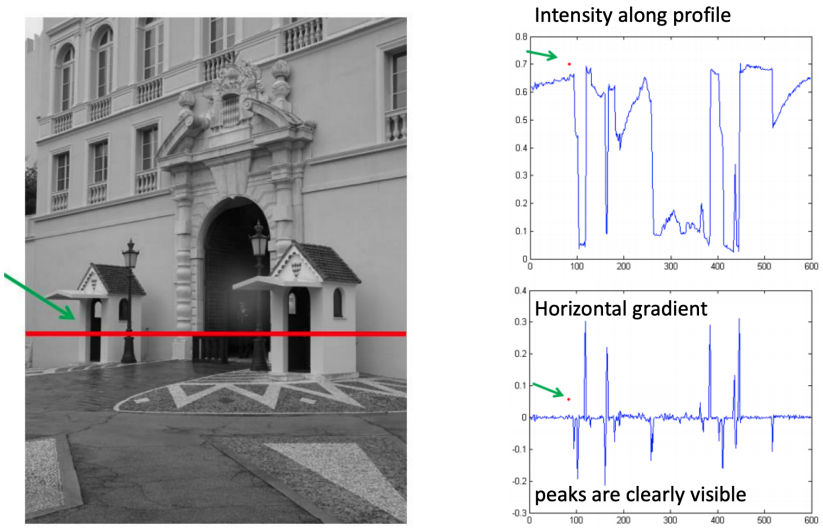


Figure 2.8: Example of finite differencing

We need to be careful with noise because smaller peaks are lost in the noise. Therefore, we want first to smooth the image and then to compute the gradient. The amount of smoothing depends on the value of sigma (width of the gaussian) and this leads then to different structures in the gradients.
**Larger values of sigma:** larger scale edges detected
**Smaller values of sigma:** finer details detected

**In practice**
To compute gradients, convolve with a derivative-of-gaussian filter. This is equivalent to smoothing with a gaussian (removes "high-frequency" components, values sum to one) and then taking the derivative (contain some negative values, values sum to 0, yield large responses at points with high contrasts like edges).

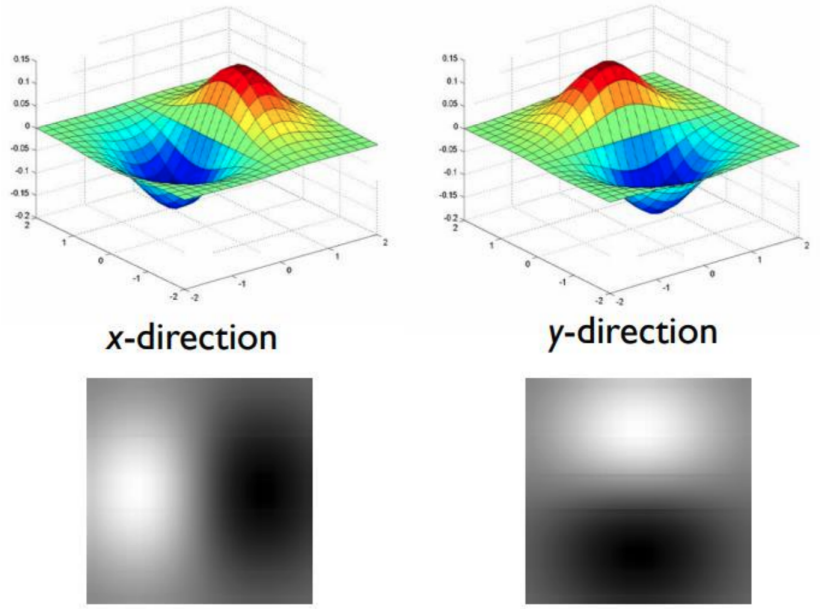


Figure 2.9: Derivative-of-gaussian filters

### 2.3.3 Sobel operator

Instead of using the simple finite differencing kernel, one often uses a 3x3 filter called the sobel operator.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \text{ and } \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

In order to detect edges at all orientation, we will compute (an approximation of) both the horizontal and vertical components of the image gradients with the sobel operator; then, we take the square root of the sum of their squares (pythagoras' theorem) to obtain the modulo of the gradient regardless on the direction.

```
sobel_h = np.array([[-1, 0 +1],[-2, 0 +2],[-1, 0 +1]])
sobel_v = sobel_h.T

im_sobelh = scipy.ndimage.convolve(im, sobel_h[::-1,::-1], mode='constant', cval=0.0)
im_sobelv = scipy.ndimage.convolve(im, sobel_v[::-1,::-1], mode='constant', cval=0.0)
im_sobel = (im_sobelh ** 2 + im_sobelv ** 2) ** 0.5

fig, axs = plt.subplots(ncols=3)
axs[0].imshow(im_sobelh, vmin=-2, vmax=+2, cmap="bwr")
axs[1].imshow(im_sobelv, vmin=-2, vmax=+2, cmap="bwr")
axs[2].imshow(im_sobel,  vmin=0, vmax=+1, cmap="viridis")
```

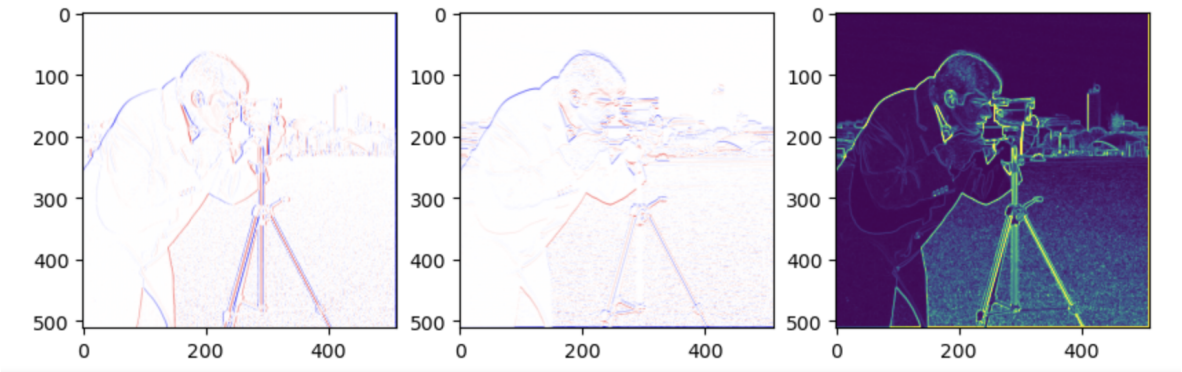


Figure 2.10: Edge detection using Sobel operator

## 2.4 Canny edge detection algorithm

- Approximate gradients $\frac{\partial f}{\partial x}$ $\frac{\partial f}{\partial y}$ along x and y by convolving with derivative-of-gaussian filters.
- Compute gradient magnitude as $|\nabla f| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$
- Make edges 1-pixel-wide (thinning): **non-maxima suppression** along perpendicular direction to edge, i.e. direction of gradient.
- Only keep strong edges: **hysteresis thresholding**

```
im = skimage.img_as_float(skimage.data.camera())

@widgets.interact(sigma=(1,21,1))
def f(sigma=3):
    im_edges = skimage.feature.canny(im, sigma=sigma)
    plt.imshow(im_edges, cmap="gray")
```

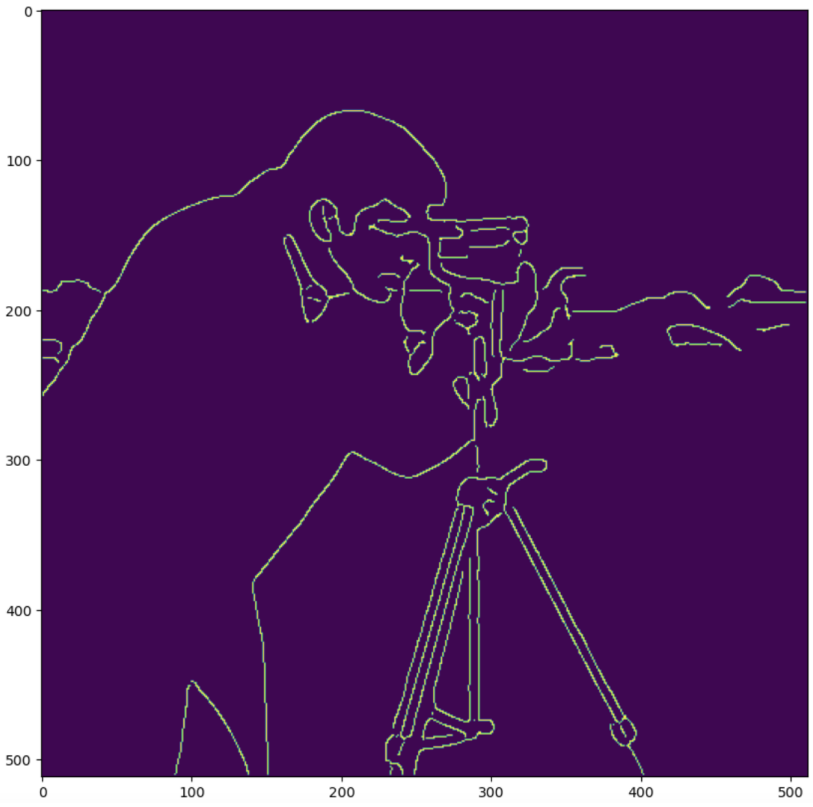


Figure 2.11: Example of canny edge detector

skimage.feature. **canny** (*image*, *sigma=1.0*, *low_threshold=None*, *high_threshold=None*, *mask=None*, *use_quantiles=False*) [source]

Edge filter an image using the Canny algorithm.

**Parameters:**

**image : 2D array**

Grayscale input image to detect edges on; can be of any dtype.

**sigma : float**

Standard deviation of the Gaussian filter.

**low_threshold : float**

Lower bound for hysteresis thresholding (linking edges). If None, low_threshold is set to 10% of dtype's max.

**high_threshold : float**

Upper bound for hysteresis thresholding (linking edges). If None, high_threshold is set to 20% of dtype's max.

**mask : array, dtype=bool, optional**

Mask to limit the application of Canny to a certain area.

**use_quantiles : bool, optional**

If True then treat low_threshold and high_threshold as quantiles of the edge magnitude image, rather than absolute edge magnitude values. If True then the thresholds must be in the range [0, 1].

**Returns:**

**output : 2D array (image)**

The binary edge map.

Figure 2.12: Documentation of Canny edge detector

### 2.4.1 Non-maxima suppression

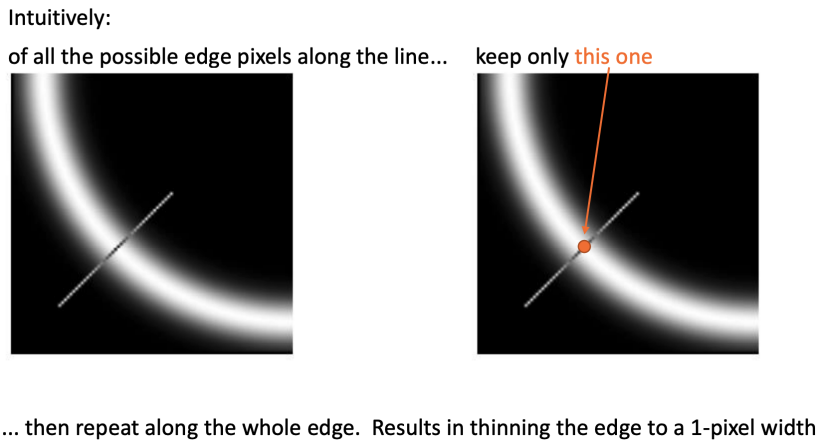


Figure 2.13: Non-maxima suppression

### 2.4.2 Hysteresis thresholding

Use a high threshold to start curves and a low threshold to continue them

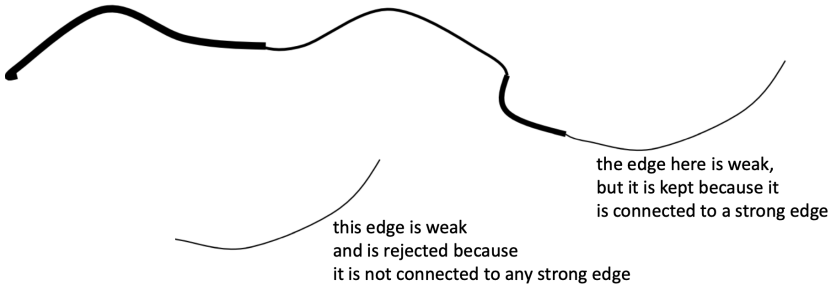


Figure 2.14: Hysteresis thresholding

# Bibliography