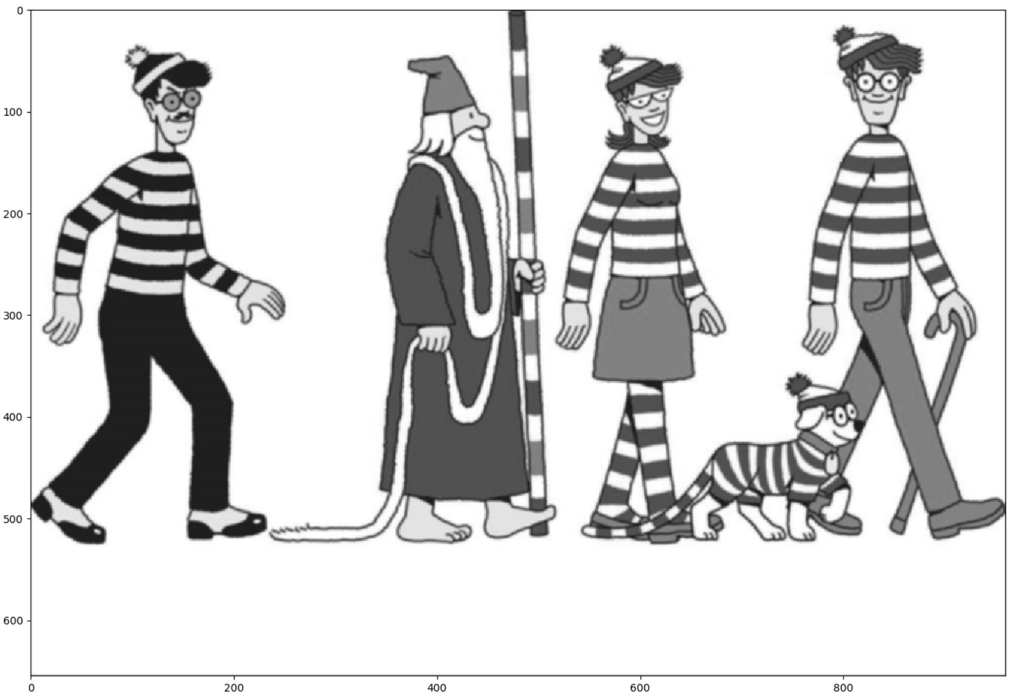
Question 1

Part A



Convolution function applied to waldo.png with a gaussian filter with std=1
* Code found in appendix

Part B

3D Convolution using a 3D matrix with center point = 1 and all other points 0. Result should be the original image.
* Code found in appendix

Question 2

Part A

Yes, it is possible to get the same result with only 1 convolution.

$(I * f_1) * f_2 = I * (f_1 * f_2)$ via. ASSOCIATIVITY

The filter to do this would then be g

$$I * (f_1 * f_2) = I * g \implies g = f_1 * f_2$$

Assuming the 2x2 filter uses the top left point (0,0), borders are zero-padded, and the output is the same size as the input f1 and f2, the resulting convolution yields

$$g = \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae & be + af \\ ce + ag & de + cf + bg + ah \end{bmatrix}$$

Where each matrix element is calculated using the convolution formula:

$$g(i,j) = \sum_{u=0}^{1} \sum_{v=0}^{1} f_2(u,v) \cdot f_1(i-u, j-v)$$

Part B

Since 99.7 percent of gaussian distributed data lie within 3 standard deviations of the mean, we want our gaussian filter to be at least 3 standard deviations away from the centre pixel. However, since we are dealing with a discrete normal distribution, the values exactly 3 standard deviations away from the mean are approximately 0. Therefore we can shrink the gaussian kernel to be of size 2*std on each side from the mean. This produces a filter size of 4*std + 1

* Code found in appendix

Part C

Convolution with gaussian with std=1

Convolution with gaussian with std=2

Part D

Isotropic Gaussian Distribution

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

$$\frac{\partial G(x,y)}{\partial y}=-\frac{1}{2\pi\sigma^4}ye^{-\frac{x^2+y^2}{2\sigma^2}}=\left(\frac{1}{\sigma^2\sqrt{2\pi}}e^{-\frac{x^2}{2\sigma^2}}\right)\left(-\frac{1}{\sigma^2\sqrt{2\pi}}ye^{-\frac{y^2}{2\sigma^2}}\right)$$

Since the function can be split into f(x)g(y) that means that it is a separable filter

Anisotropic Gaussian Distribution

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

$$\frac{\partial G(x,y)}{\partial y}=-\frac{1}{2\pi\sigma_x\sigma_y^3}ye^{-\left(\frac{x^2}{2\sigma_x^2}+\frac{y^2}{2\sigma_y^2}\right)}=\left(\frac{1}{\sigma_x\sqrt{2\pi}}e^{-\frac{x^2}{2\sigma_x^2}}\right)\left(-\frac{1}{\sigma_y^3\sqrt{2\pi}}ye^{-\frac{y^2}{2\sigma_y^2}}\right)$$

Since the function can be split into f(x)g(y) that means that it is a separable filter

Part E

Let us assume we are convolving a mxn image with a pxq kernel

Number of operations for 2D convolution:
* Considering only multiplication as an operation

$$(p\times q)\cdot(m\times n)$$

Here each pxq pixel in the kernel is multiplied with a corresponding pixel in the image. This is repeated for all mxn pixels in the image

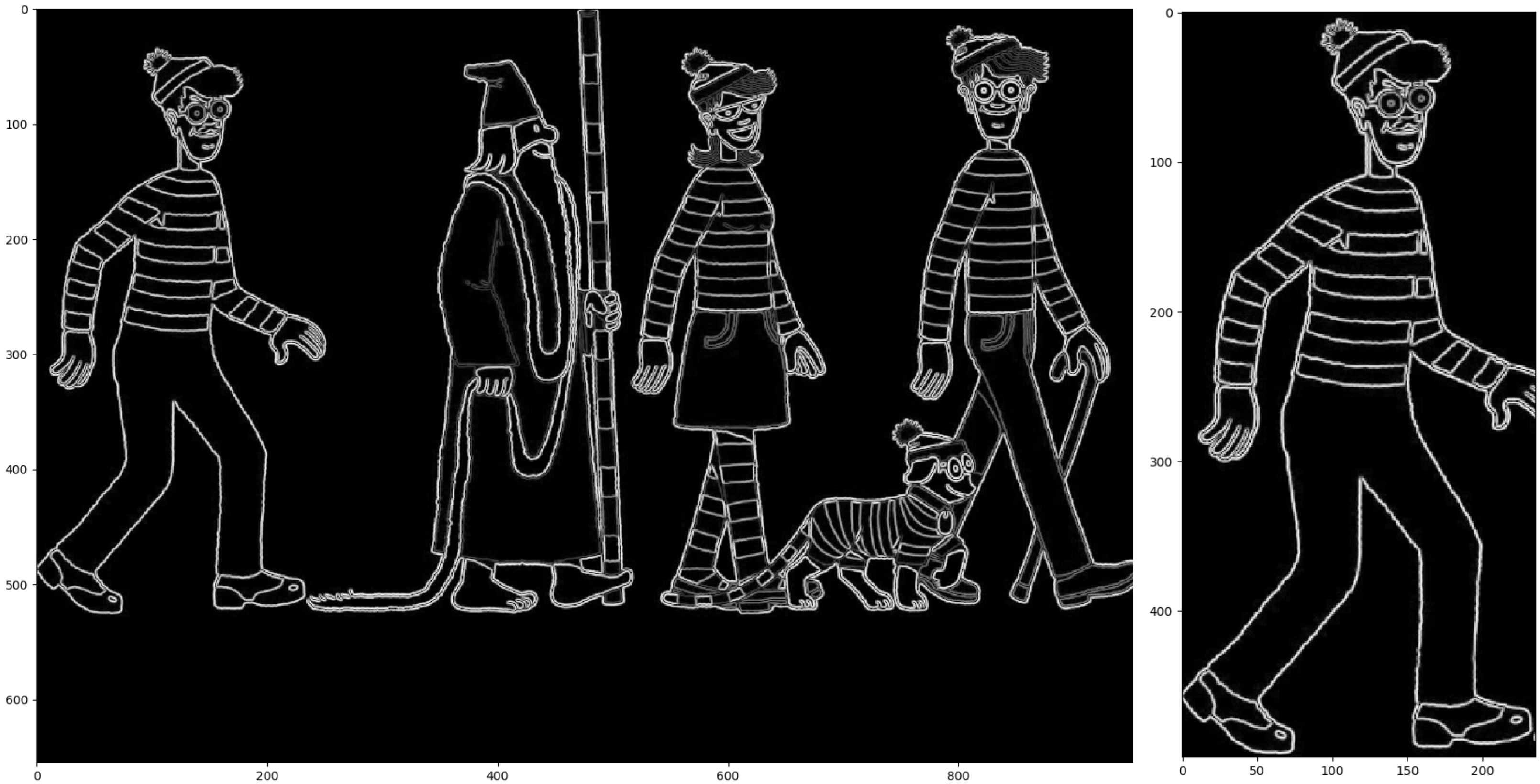
Number of operations for 2D convolution with separable filter:
* Considering only multiplication as an operation

$$(p+q)\cdot(m\times n)$$

In this case the finer can be separated into two 1D kernels of size p and q. This means that each p and q pixels in the kernels are multiplied with corresponding pixels in the image. This is repeated for all mxn pixels in the image.

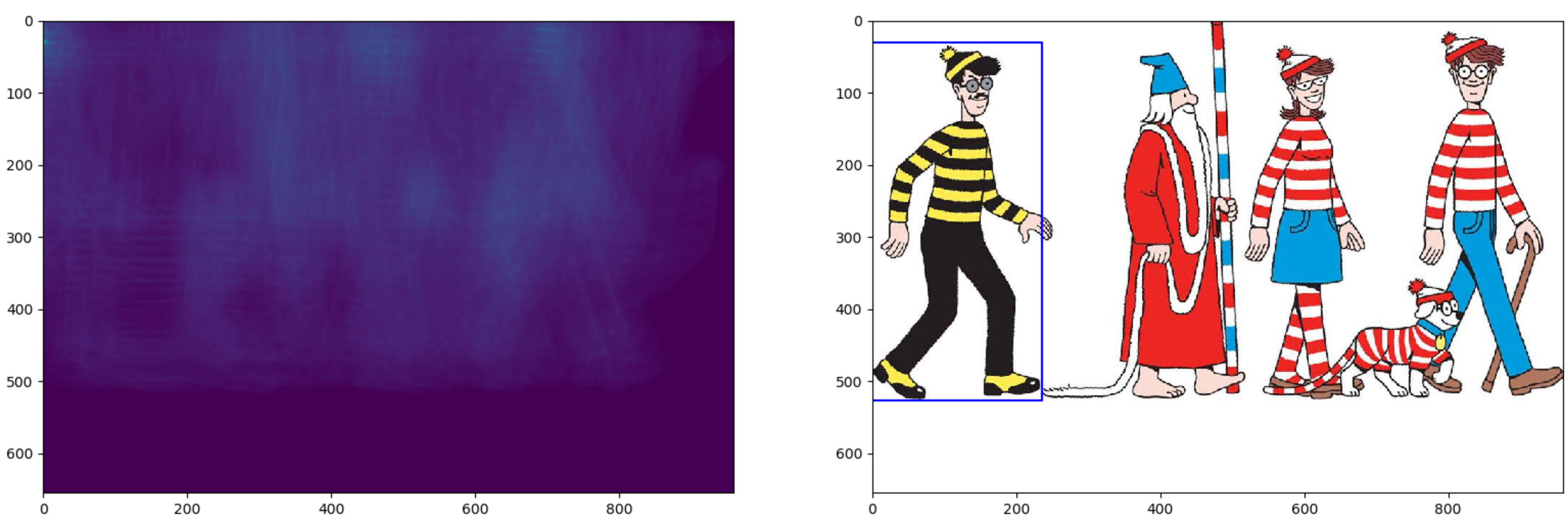
Question 3

Part A



gradient magnitude of waldo.png and template.png

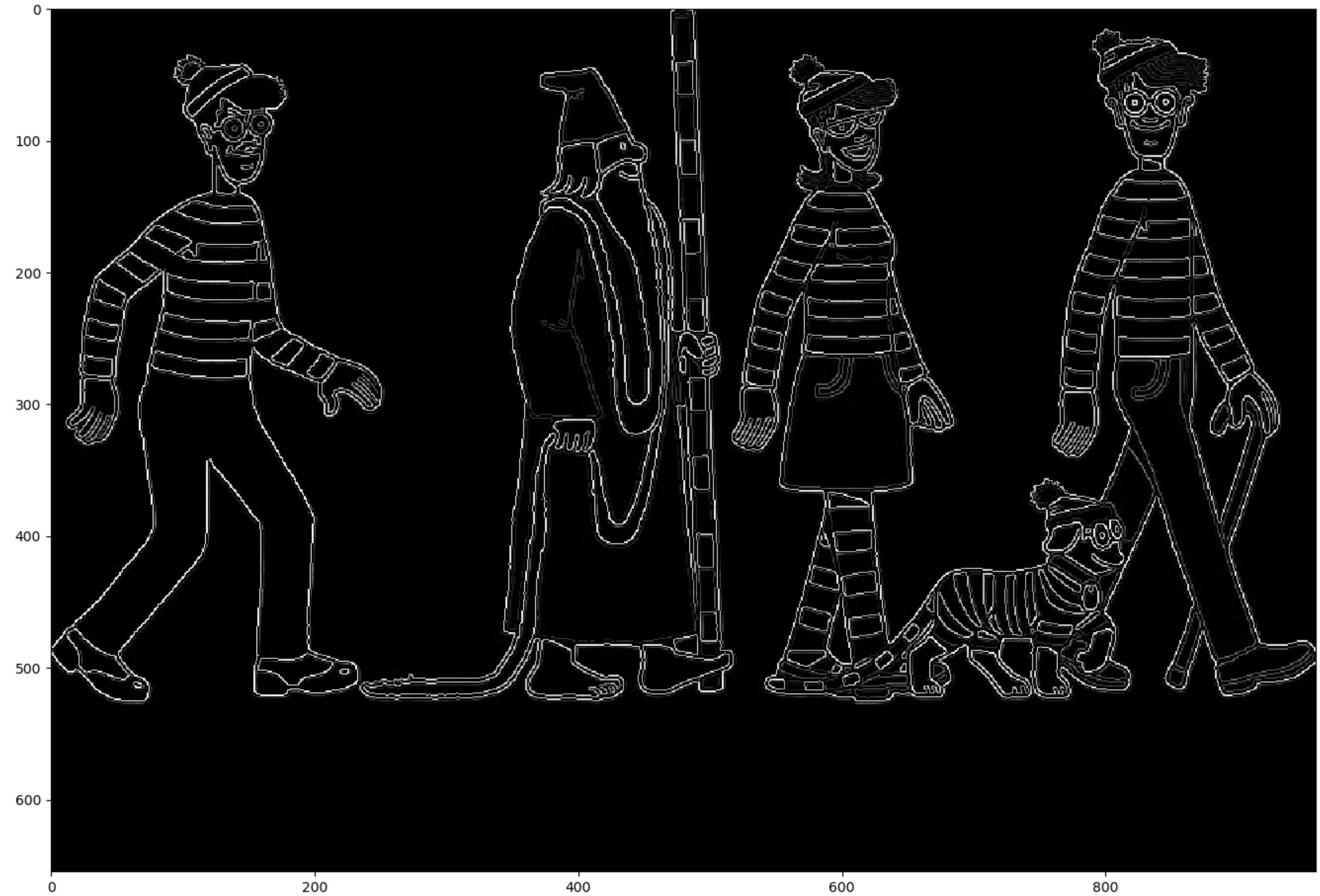
Part B



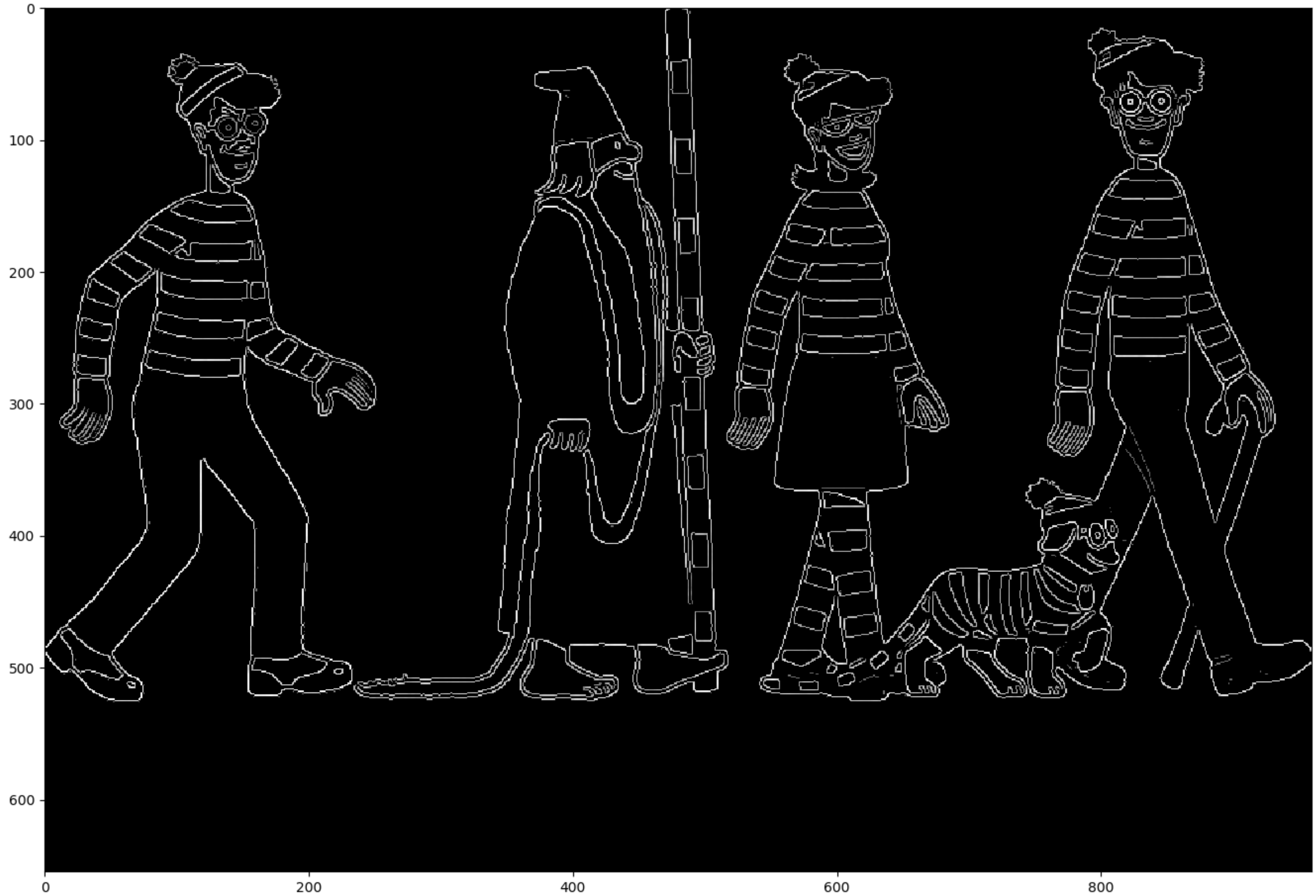
normalized cross-correlation is displayed on the left, and a localized template is shown on the right

Question 4

Part A



Implementation of the canny edge detector of waldo.png with no thresholding



Implementation of the canny edge detector of waldo.png with threshold = 0.4

Appendix: Code

q1.py

```
import numpy as np
from skimage import io
from convolution import *
from scipy import ndimage

def q1A():
    image = io.imread('images/waldo.png', as_gray=True)
    filter = np.array([
        [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],
    ])*(1/273)

    filtered_image = convolution(image, filter)
    #filtered_image = ndimage.convolve(image, filter, mode='constant', cval=0.0)

    io.imshow(filtered_image)
    io.show()

def q1B():
    image = io.imread('images/waldo.png')[:,:,:3]
    filter = np.array([
        [
            [0,0,0],
            [0,0,0],
            [0,0,0]
        ],
        [
            [0,0,0],
            [0,1,0],
            [0,0,0]
        ],
        [
            [0,0,0],
            [0,0,0],
            [0,0,0]
        ]
    ])

    # 3D Convolution
    # image: 3+ dimensions
    # filter: 3 dimensions
    res = convolution_3D(image, filter)

    # if we want to convolve a 2D filter with an RGB image
    # image: 3 dimensions (RGB)
    # filter: 2 dimensions
    # res = convolution_RGB_image(image, filter)

    io.imshow(res)
    io.show()

if __name__ == '__main__':
    q1A()
    q1B()
```

q2.py

```
import numpy as np
from skimage import io
from scipy import ndimage

from gaussian import *

def q2B():
    G = gaussian_kernel(2)
    print(G)

def q2C():
    image = io.imread('images/waldo.png')[:,:,:3]
    G = gaussian_kernel(2)

    # Convolve each RGB dimension
    image[:,:,0] = ndimage.convolve(image[:,:,0], G, mode='constant', cval=0.0)
    image[:,:,1] = ndimage.convolve(image[:,:,1], G, mode='constant', cval=0.0)
    image[:,:,2] = ndimage.convolve(image[:,:,2], G, mode='constant', cval=0.0)

    io.imshow(image)
    io.show()

if __name__ == '__main__':
    q2B()
    q2C()
```

q3.py

```
import numpy as np
from skimage import io
from matplotlib import pyplot as plt
from gradient import *
from template_matching import *

def q3A():
    image = io.imread('images/waldo.png', as_gray=True)
    filter = io.imread('images/template.png', as_gray=True)

    G_image, G_x_image, G_y_image, theta_image = gradient(image)
    G_filter, G_x_filter, G_y_filter, theta_filter = gradient(filter)

    # PLOT BOTH IMAGE GRADIENTS
    fig, axis = plt.subplots(1, 2)      # nrows, ncols
    plt.subplot(1,2,1)                  # nrows, ncols, index
    plt.imshow(G_image, cmap='gray')

    plt.subplot(1,2,2)
    plt.imshow(G_filter, cmap='gray')

    plt.show()


def q3B():
    image = io.imread('images/waldo.png', as_gray=True)
    image_color = io.imread('images/waldo.png')
    filter = io.imread('images/template.png', as_gray=True)

    # GET IMAGE GRADIENTS
    G_image, G_x_image, G_y_image, theta_image = gradient(image)
    G_filter, G_x_filter, G_y_filter, theta_filter = gradient(filter)

    # LOCALIZE IMAGE
    similarity, corners = match_template(G_image, G_filter)

    # PLOT SIMILARITY
    fig, axis = plt.subplots(1, 2)      # nrows, ncols
    plt.subplot(1,2,1)                  # nrows, ncols, index
    plt.imshow(similarity)

    # PLOT BOX AROUND TEMPLATE
    plt.subplot(1,2,2)
    plt.plot(corners[:, 1], corners[:, 0], 'b')
    plt.imshow(image_color)

    plt.show()

if __name__ == '__main__':
    q3A()
    q3B()
```

q4.py

```
import numpy as np
from skimage import io
from matplotlib import pyplot as plt

from canny_edge_detector import *


def q4():
    image = io.imread('images/waldo.png', as_gray=True)

    CED = canny_edge_detector(image, std=1)

    # PLOT GRAYSCALE and CANNY EDGE DETECTED IMAGES
    fig, axis = plt.subplots(1, 2)      # nrows, ncols
    plt.subplot(1,2,1)                  # nrows, ncols, index
    plt.imshow(image, cmap='gray')

    plt.subplot(1,2,2)
    plt.imshow(CED, cmap='gray')

    plt.show()

if __name__ == '__main__':
    q4()
```

convolution.py

```
import numpy as np
from correlation import *

def convolution(image, filter):

    # flip filter on both its axis
    filter = np.flip(filter, 0)
    filter = np.flip(filter, 1)

    # filter applied in convolution is just the filter
    # flipped on both axis and applied in correlation
    return cross_correlation(image, filter)


# CONVOLUTION OF A RGB IMAGE AND A 2D FILTER
# image: 3 dimensions
# filter: 2 dimension
def convolution_RGB_image(image, filter):

    if image.ndim == 2:
        # GRAYSCALE
        filtered_image = convolution(image, filter)
    elif image.ndim == 3:
        # RGB
        filtered_image = np.empty_like(image)
        filtered_image[:,:,0] = convolution(image[:,:,0], filter)
        filtered_image[:,:,1] = convolution(image[:,:,1], filter)
        filtered_image[:,:,2] = convolution(image[:,:,2], filter)

    return filtered_image

# GENERAL 3D CONVOLUTION
# image: 3+ dimensions
# filter: 3 dimensions
def convolution_3D(image, filter):

    # flip filter on all its axis
    filter = np.flip(filter, 0)
    filter = np.flip(filter, 1)
    filter = np.flip(filter, 2)

    # filter applied in convolution is just the filter
    # flipped on both axis and applied in correlation
    return cross_correlation_3D(image, filter)
```

correlation.py

```
import numpy as np
from boundary import *

# 2D CORRELATION ---------------------------------------------------
def correlation(image, filter, i, j):
    height, width = filter.shape
    k = int((height-1)/2)
    l = int((width-1)/2)

    # vectorize area around point i,j
    mask = image[i-k:i+k+1, j-l:j+l+1].flatten()
    filter = filter.flatten()

    res = np.dot(mask, filter)
    return res

def cross_correlation(image, filter):
    height, width = image.shape
    frame, col_pad, row_pad = zero_pad(image, filter)

    res = np.empty_like(frame)
    # Traverse all pixels of the image and calculate the correlation at each point
    for i in range(col_pad, col_pad+height):
        for j in range(row_pad, row_pad+width):
            res[i, j] = correlation(frame, filter, i, j)

    # Return the original size of the image
    return res[col_pad:col_pad+height, row_pad:row_pad+width]

# 3D CORRELATION ---------------------------------------------------
def correlation_3D(image, filter, i, j, k):
    height, width, depth = filter.shape
    p = int((height-1)/2)
    q = int((width-1)/2)
    r = int((depth-1)/2)

    # vectorize 3D shape around point i,j,k
    mask = image[i-p:i+p+1, j-q:j+q+1, k-r:k+r+1].flatten()
    filter = filter.flatten()

    res = np.dot(mask, filter)
    return res

def cross_correlation_3D(image, filter):
    height, width, depth = image.shape
    frame, col_pad, row_pad, depth_pad = zero_pad_3D(image, filter)

    res = np.empty_like(frame)
    # Traverse all elements of the 3D matrix and calculate the 3D correlation at each point
    for i in range(col_pad, col_pad+height):
        for j in range(row_pad, row_pad+width):
            for k in range(depth_pad, depth_pad+depth):
                res[i, j, k] = correlation_3D(frame, filter, i, j, k)

    # Return the original size of the image
    return res[col_pad:col_pad+height, row_pad:row_pad+width, depth_pad:depth_pad+depth]
```

boundary.py

```
import numpy as np

def crop_filter(filter):
    height, width = filter.shape
    # Crop filter to be odd x odd
    if height%2 == 0:
        height = height - 1
    if width%2 == 0:
        width = width - 1
    filter = filter[:height, :width]
    return filter

def zero_pad(image, filter):
    height, width = filter.shape
    j = int((height-1)/2)
    k = int((width-1)/2)

    pad_axis_1 = (j,j)      # Pad j pixels before and after axis 1
    pad_axis_2 = (k,k)
    padding = (pad_axis_1, pad_axis_2)

    frame = np.pad(image, padding, mode='constant', constant_values=0)
    return frame, j, k

def zero_pad_3D(image, filter):
    height, width, depth = filter.shape
    j = int((height-1)/2)
    k = int((width-1)/2)
    l = int((depth-1)/2)

    pad_axis_1 = (j,j)      # Pad j pixels before and after axis 1
    pad_axis_2 = (k,k)
    pad_axis_3 = (l,l)
    padding = (pad_axis_1, pad_axis_2, pad_axis_3)

    frame = np.pad(image, padding, mode='constant', constant_values=0)
    return frame, j, k, l

def zero_pad_extend(image, filter):
    height, width = filter.shape

    pad_axis_1 = (height,height)      # Pad j pixels before and after axis 1
    pad_axis_2 = (width,width)
    padding = (pad_axis_1, pad_axis_2)

    frame = np.pad(image, padding, mode='constant', constant_values=0)
    return frame, height, width
```

gaussian.py

```
import numpy as np

def gaussian_distribution(std, x, y):
    term1 = 1/(2*np.pi*(std**2))
    term2 = np.exp(-(x**2 + y**2)/(2*(std**2)))
    return term1*term2

def gaussian_kernel(std):
    # The normal distribution is effectively zero at
    # 3 standard deviations away from the mean
    # Therefore given std we choose a kernel size of 4*std + 1

    # kernel size
    k = int(4*std + 1)  # make sure is an integer number

    # kernel sizes must be odd
    if k%2 == 0:
        k = k + 1

    # index of mean
    mu = (k-1)/2

    G = np.zeros((k,k))
    for x in range(0,k):
        for y in range(0,k):
            G[x, y] = gaussian_distribution(std, x-mu, y-mu)

    return G/G.sum()
```

gradient.py

```
import numpy as np
from scipy import ndimage

def gradient(image):

    sobel_x = np.array([
        [-1, 0, +1],
        [-2, 0, +2],
        [-1, 0, +1]
    ])
    sobel_y = np.array([
        [-1, -2, -1],
        [0, 0, 0],
        [+1, +2, +1]
    ])

    G_x = ndimage.correlate(image, sobel_x, mode='nearest')
    G_y = ndimage.correlate(image, sobel_y, mode='nearest')
    G = np.sqrt(G_x**2 + G_y**2)

    tan_theta = G_y/G_x             #ignore /0 warnings
    theta = np.arctan(tan_theta)

    # NORMALIZE THE MATRIX TO GRAYSCALE (0-1)
    G_max = G.max()
    G = G/G_max

    return G, G_x, G_y, theta
```

template_matching.py

```
import numpy as np
from boundary import *
from skimage import io

def normalized_correlation(image, filter, i, j):
    height, width = filter.shape

    mask = image[i:i+height, j:j+width].flatten()
    filter = filter.flatten()

    m_dot_f = np.dot(mask, filter)
    norm_m = np.linalg.norm(mask)
    norm_f = np.linalg.norm(filter)

    if (norm_m * norm_f)==0:
        res = 0
    else:
        res = m_dot_f/(norm_m * norm_f)
    return res

def normalized_cross_correlation(image, filter):
    height, width = image.shape
    frame, col_pad, row_pad = zero_pad_extend(image, filter)

    res = np.empty_like(frame)
    for i in range(col_pad, col_pad+height):
        for j in range(row_pad, row_pad+width):
            res[i, j] = normalized_correlation(frame, filter, i, j)

    # Return the original size of the image
    return res[col_pad:col_pad+height, row_pad:row_pad+width]

def match_template(image, filter):

    similarity = normalized_cross_correlation(image, filter)

    max_index = similarity.argmax()
    i, j = np.unravel_index(max_index, similarity.shape)

    height, width = filter.shape
    corners = np.array([
        [i, j],
        [i+height-1, j],
        [i+height-1, j+width-1],
        [i, j+width-1],
        [i, j]
    ])

    return similarity, corners
```

canny_edge_detector.py

```
import numpy as np
from skimage.filters import *
from gradient import *

def round_direction(theta):
    # Convert to degrees b/c easier to understand
    angle = (theta/np.pi)*180

    # Round each angle to 45 degrees
    angle = np.round(angle/45.0) * 45
    return angle

def canny_edge_detector(image, std):
    # APPLY GAUSSIAN SMOOTHING
    image = gaussian(image, sigma=std)

    # FIND IMAGE GRADIENT
    G, G_x, G_y, theta = gradient(image)
    angle = round_direction(theta)

    # Pad one extra pixel to avoid out of index error
    padding = (1,)
    G = np.pad(G, padding, mode='constant', constant_values=0)
    height, width = G.shape

    # NON MAXIMUM SUPRESSION
    for i in range(1, height-1):
        for j in range(1, width-1):
            dir = angle[i-1, j-1]
            if dir == 0:
                if not G[i, j] > max(G[i, j+1], G[i, j-1]):
                    G[i, j] = 0
            elif dir == 45:
                if not G[i, j] > max(G[i+1, j+1], G[i-1, j-1]):
                    G[i, j] = 0
            elif dir == -45:
                if not G[i, j] > max(G[i-1, j+1], G[i+1, j-1]):
                    G[i, j] = 0
            elif dir == 90 or dir == -90:
                if not G[i, j] > max(G[i+1, j], G[i-1, j]):
                    G[i, j] = 0

    # HYSTERISIS THRESHOIDING
    # applied to only minVal threshold
    minVal = 0.5
    G[G < minVal] = 0

    return G[1:-1, 1:-1]    # Return original sized image
```