CSC420 A1

Aalan Mohammad

January 2021

1

a) Following is my implantation of the 2D gray scale convolution function in Python.

Assumptions made 1) inputs are in matrix form 2) the filter is odd sized square matrix 3) zero padding was used to get the same size result

```python
def convolution(img_matrix, fltr_matrix):

    # flip the filter vertically and horizontaly for convolution.
    fltr_matrix = fltr_matrix[::-1,::-1]
    fltr_shape = fltr_matrix.shape[0]
    padding = fltr_shape - 1

    # zero pad the original image to do the convolution.

    padded = np.zeros((img_matrix.shape[0] + padding, img_matrix.shape[1] + padding))
    # copy in the image matrix between the padding in the matrix
    for row in range(padding//2, padded.shape[0] - (padding//2)):
        for col in range(padding//2, padded.shape[1] - (padding//2)):
            padded[row,col] = img_matrix[row - (padding//2), col - (padding//2)]

    # output matrix same size as the input matrix
    result = np.zeros(img_matrix.shape)

    # let's fill the result!
    for row in range(result.shape[0]):
        for col in range(result.shape[1]):
            # get the padded image section
            img_section = padded[row:(row + fltr_shape),col:(col + fltr_shape)]
            # do the convolution calculation using the dot product by flattening the matrices
            convolution_result = img_section.flatten().dot(fltr_matrix.flatten())
            result[row, col] = convolution_result
    return result
```

Listing 1: Convolution

b) Assumptions made 1) inputs are in matrix form 2) filter is not flip, so I flipped for convolution 3) Rounding limit made to 15 digits, so any value more that 15 is treated as zero for verification

```python
def separable(fltr, image):
    """
    1) check to see if filter is separable
    2) perform a faster convolution with the given image

    NOTE: assuming that filter and image are in matrix form

    """
    #flip the filter
    new_fltr = fltr[::-1,::-1]
    # check to see if filter is separable
    U, S, V = np.linalg.svd(new_fltr)

    # Assuming that the 2nd element is close to zero, gives room for python rounding errors
    if (round(S[1], 15) != 0):
        return "Filter is not separable"

    # function is separable, continue
    sigma = S[0]
    # The vertical and horizontal filters from first part
    vertical = math.sqrt(sigma) * np.asmatrix(V[0])
    horizontal = math.sqrt(sigma) * np.asmatrix(U[:,0])

    # Using python built in function, first convolve image with
    # horizontal filter, then with the vertical filter.

    horizontal_output = ndimage.convolve(image, horizontal)

    return ndimage.convolve(horizontal_output, vertical.T)
```

Listing 2: Separable

c) The number of operations required for a 2D convolution is k^2 where k is the size of the square filter and k x l for a rectangular filter. A convolution with a separable filter is $2k$ for the square filter and $k + l$ for a rectangular filter.

So an example would be if we had an image with 500 pixels, and a filter that is 5 x 5 with $k = 5$, then the standard convolution is 500 x 5^2 = 12,500 operations. With a separable filter, it would be 500 x (2 x 5) = 5,000 operations, so we save double the operations.

2

a) Yes, it is possible to perform just one convolution instead of 2. The property that allows us to do that is associative property.

The current method is $(I \circledast f_1) \circledast f_2$ and this takes two convolutions. What we can do instead is combine the two filters and then perform a single convolution with the image. $I \circledast (f_1 \circledast f_2)$

To get one filter kernel we do:

$$f_3 = f_1 \circledast f_2 = \sum_{u=-k}^{k} \sum_{v=-k}^{k} f_1[i,j] f_2[i-u, j-v]$$

First, lets flip f_2 to convolve with f_1:

$$f_2 = \begin{bmatrix} h & g \\ f & e \end{bmatrix}$$

Now let's zero pad f_1 for convolution.

$$f_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & a & b & 0 \\ 0 & c & d & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$f_3 = \begin{bmatrix} e \cdot a & (f \cdot a + e \cdot b) & f \cdot b \\ (g \cdot a + e \cdot c) & f2 \cdot f1 & (h \cdot b + d \cdot f) \\ g \cdot c & (h \cdot c + d \cdot g) & d \cdot a \end{bmatrix}$$

$$I \circledast f_3$$

Since the two filters are small it will be a quick computation instead of the double convolution on the bigger image.

b) For the Laplachian of Gaussians, we have to find the second partial derivatives of the Gaussian Function.

So we have:

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

First Partial Derivatives:

$$\frac{\partial f(x,y)}{\partial x} = -\frac{x}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$\frac{\partial f(x,y)}{\partial y} = -\frac{y}{2\pi\sigma^4} e^{-\frac{x^2+y^4}{2\sigma^2}}$$

Second Partial derivatives:

$$\frac{\partial f(x,y)}{\partial x^2} = \frac{(x^2 - \sigma^2)}{2\pi\sigma^6} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$\frac{\partial f(x,y)}{\partial y^2} = \frac{(y^2 - \sigma^2)}{2\pi\sigma^6} e^{-\frac{x^2+y^4}{2\sigma^2}}$$

So know we have those two, and add them up and to get the Laplachian:

```

def laplachian(sigma):
    # size of the kernel will be 5 x 5 matrix
    h_x = np.zeros((5,5))
    # using the second partial derivative of the Gaussian with respect to x, we get:
    for row in range(5):
        for col in range(5):
            power = ((row - 2)**2 - (sigma ** 2)) * np.exp(-((row - 2) ** 2 + (col - 2) ** 2) / (2 * sigma ** 2))
            h_x[row,col] = power / (2 * np.pi * sigma ** 6)

    h_y = np.zeros((5,5))
    # using the second partial derivative of the Gaussian with respect to y, we get:
    for row in range(5):
        for col in range(5):
            power = ((col - 2)**2 - (sigma ** 2)) * np.exp(-((row - 2) ** 2 + (col - 2)** 2) / (2 * sigma ** 2))
            h_x[row,col] = power / (2 * np.pi * sigma ** 6)
    # adding the two to get the Laplacian of Gaussians
    return h_y + h_x
```

Listing 3: LoG with sigma

c)
Used the following to get the image:

```
def gaussian(img, sigma):
    # Read image as matrix
    img_matrix = io.imread(img, as_gray=True)
    # Using a 9 x 9 matrix
    gauss = np.zeros((9,9))
    for row in range(9):
        for col in range(9):
            power =  np.exp(-((row - 4) ** 2 + (col - 4) ** 2) / (2 * sigma ** 2))
            gauss[row,col] = power / (2 * np.pi * sigma ** 2)

    filtr = ndimage.convolve(img_matrix, gauss)
    io.imshow(filtr)
    io.show()
```

Listing 4: Gaussian

Sorry, the Image is on the next page.

d) The vertical derivative of the Gaussian is similar to the regular Gaussian differ in shape. So depending on the shape of the original Gaussian filter, it will decide if it is separable of not. If the Gaussian is **isotropic** like the ones we have seen in lecture, then the vertical derivative will also be **separable**. The symmetry of the filter will allow it to be separable. If the Gaussian is **anisotropic** then the derivative will not be separable as the shape of the filter will not make it possible to be split into two separate arrays.

Figure 1: Waldo.png with my 9 x 9 Gaussian filter and sigma = 1

The Laplachian of Gaussians is not a separable filter. The LoG is the sum of the second derivatives of the Gaussian, LoG $= \frac{d}{dx^2} \cdot G + \frac{d}{dy^2} \cdot G$ and that can not be separated into two 1D filters as the shape of adding both the second derivatives will make it impossible for it to be separable.

3

a)

The following is my implementation of the gradient magnitude. I used the Prewitt filter for the horizontal and vertical gradients.

```

def magnitude_gradient(img):
    img_matrix = io.imread(img,as_gray=True)

    # Using the Prewitt filter for the horizontal gradient
    horizontal = ndimage.convolve(img_matrix, np.array([[-1,0,1],[-1,0,1],[-1,0,1]]))

    # Using the Prewitt filter for the vertical gradient
    vertical = ndimage.convolve(img_matrix, np.array([[1,1,1],[0,0,0],[-1,-1,-1]]))
    final = np.zeros(img_matrix.shape)
    # square rooting the sum of squares of vertical and horizontals
    for row in range(final.shape[0]):
        for col in range(final.shape[1]):
            final[row,col] = math.sqrt(vertical[row,col] ** 2 + horizontal[row,col] ** 2)
    #io.imshow(final)
    #io.show()
```

```
    return final
```

Listing 5: magnitude gradient

Here are the outputs for the two images.

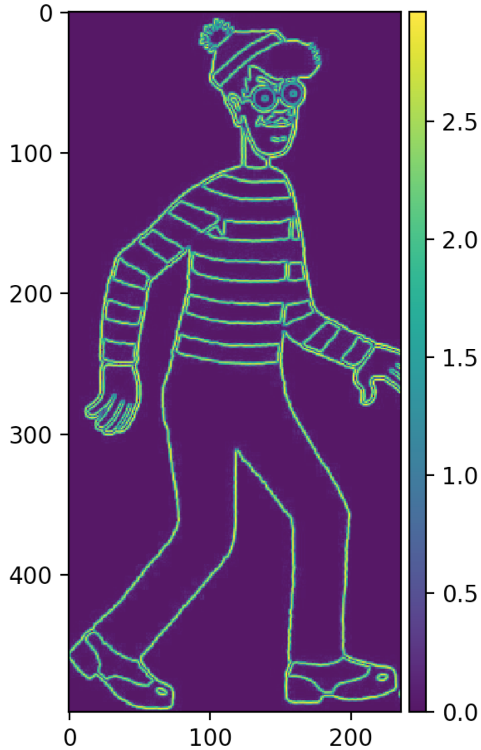

Figure 2: template.png with my gradient magnitude function

b)

For this part I used the template matching that was used in the second tutorial on the template and waldo images.

```

def grid_matching():
    # using function from part a to compute the gradient magnitude of the images
    img_gradient = magnitude_gradient("waldo.png")
    fltr_gradient = magnitude_gradient("template.png")
    #return img_gradient
    # using template matching to find result
    result = match_template(img_gradient, fltr_gradient)
    ij = np.unravel_index(np.argmax(result), result.shape)
    x,y = ij[::-1]
    fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=(8, 3))

```

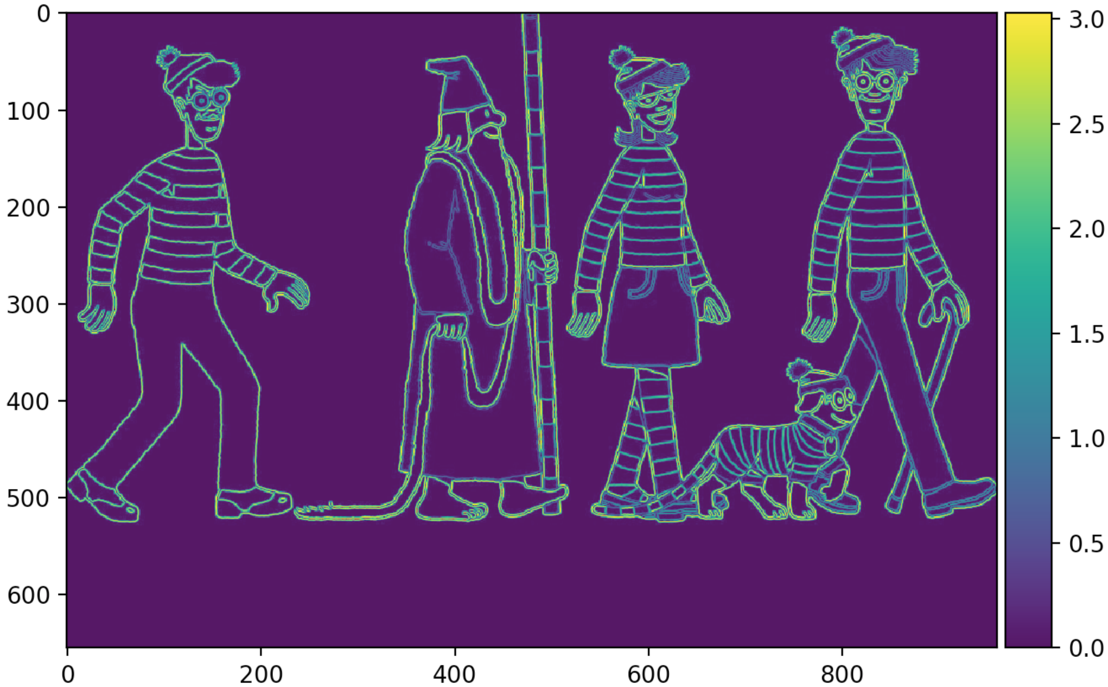

Figure 3: Waldo.png with my gradient magnitude function

```
    ax1.imshow(fltr_gradient)
    ax1.set_axis_off()
    ax1.set_title('template')

    ax2.imshow(img_gradient)
    ax2.set_axis_off()
    ax2.set_title('waldo')
    # highlight matched region
    xwaldo, ywaldo = fltr_gradient.shape
    rect = plt.Rectangle((x, y), ywaldo, xwaldo, edgecolor='r', facecolor='none')
    ax2.add_patch(rect)

    ax3.imshow(result)
    ax3.set_axis_off()
    ax3.set_title('`match_template`\nresult')
    # highlight matched region
    ax3.autoscale(False)
    ax3.plot(x, y, 'o', markeredgecolor='r', markerfacecolor='none', markersize=10)

    plt.show()
```

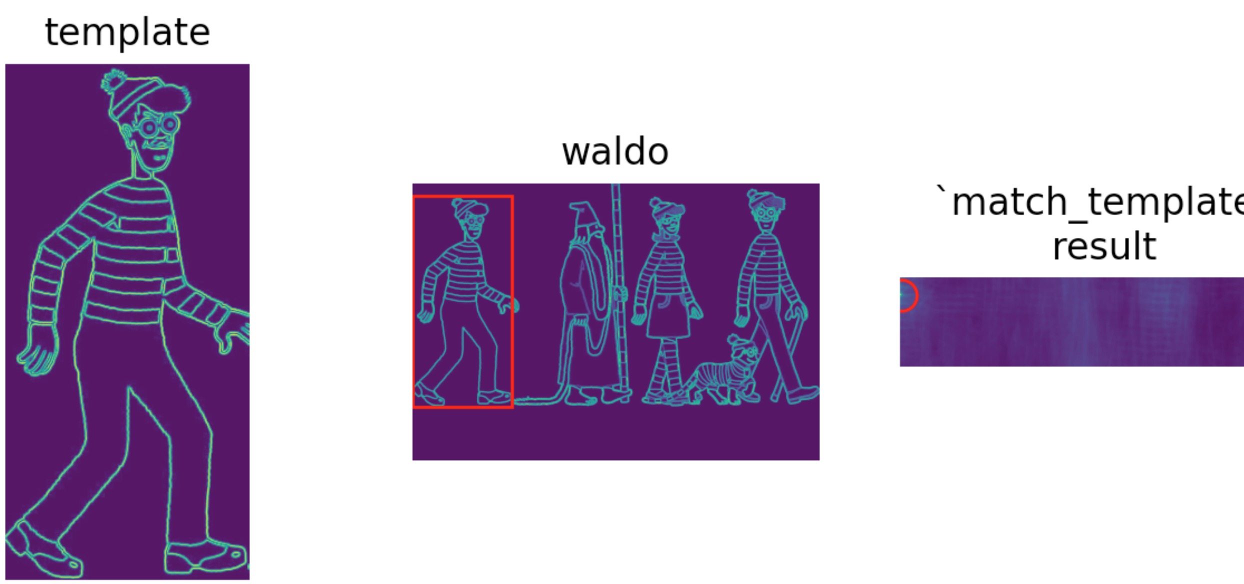

Figure 4: Template Matching

4

Here is my implementation of the Canny Edge detector. I first reduced the noise of the image with the gaussian filter, then used my gradient magnitude function from 3A and found the angles using the arctan function discussed in lecture. Finally performing non maximum suppression with the angles.

```python

def canny_edge(img):
    # read the image
    img_matrix = io.imread(img,as_gray=True)

    # Apply the Gaussian filter to reduce noise
    img_matrix = ndimage.gaussian_filter(img_matrix, sigma=1, order=0)

    # get the gradient magnitude from Q3A
    horizontal = ndimage.convolve(img_matrix, np.array([[-1,0,1],[-1,0,1],[-1,0,1]]))

    vertical = ndimage.convolve(img_matrix, np.array([[1,1,1],[0,0,0],[-1,-1,-1]]))
    gradient = np.zeros(img_matrix.shape)

    # square rooting the sum of squares of vertical and horizontals
    for row in range(gradient.shape[0]):
        for col in range(gradient.shape[1]):
            gradient[row,col] = math.sqrt(vertical[row,col] ** 2 + horizontal[row,col] ** 2)

    # get the angles for each pixel
    angles = np.zeros(img_matrix.shape)

    for row in range(angles.shape[0]):
        for col in range(angles.shape[1]):
            current_angle = np.arctan2(vertical[row,col],horizontal[row,col])
            # convert to degrees for simplicity
            current_angle = current_angle * 180 / np.pi
            if current_angle < 0:
                current_angle += 180
            angles[row, col] = current_angle
    # apply non maximum suppression
    non_max = np.zeros(img_matrix.shape)
    for row in range(1,non_max.shape[0]-1):
        for col in range(1,non_max.shape[1]-1):
            # Find the edge direction
            direction = 45 * round(angles[row,col] / 45)
            # save the neighbor edge strengths
            if (direction == 0) or (direction == 180):
                left = gradient[row, col-1]
                right = gradient[row, col+1]
            elif (direction == 45):
                left = gradient[row + 1, col - 1]
                right = gradient[row - 1, col + 1]
            elif (direction == 90):
                left = gradient[row - 1, col]
                right = gradient[row + 1, col]
            else:
                left = gradient[row - 1, col - 1]
                right = gradient[row + 1, col + 1]

```

```
            # compare edge strengths of current pixel with neighbors in gradient direction
            if (left < gradient[row,col] and right < gradient[row,col]):
                non_max[row,col] = gradient[row,col]
            else:
                non_max[row,col] = 0

    io.imshow(non_max)
    io.show()
```

Listing 6: Canny Edge detector

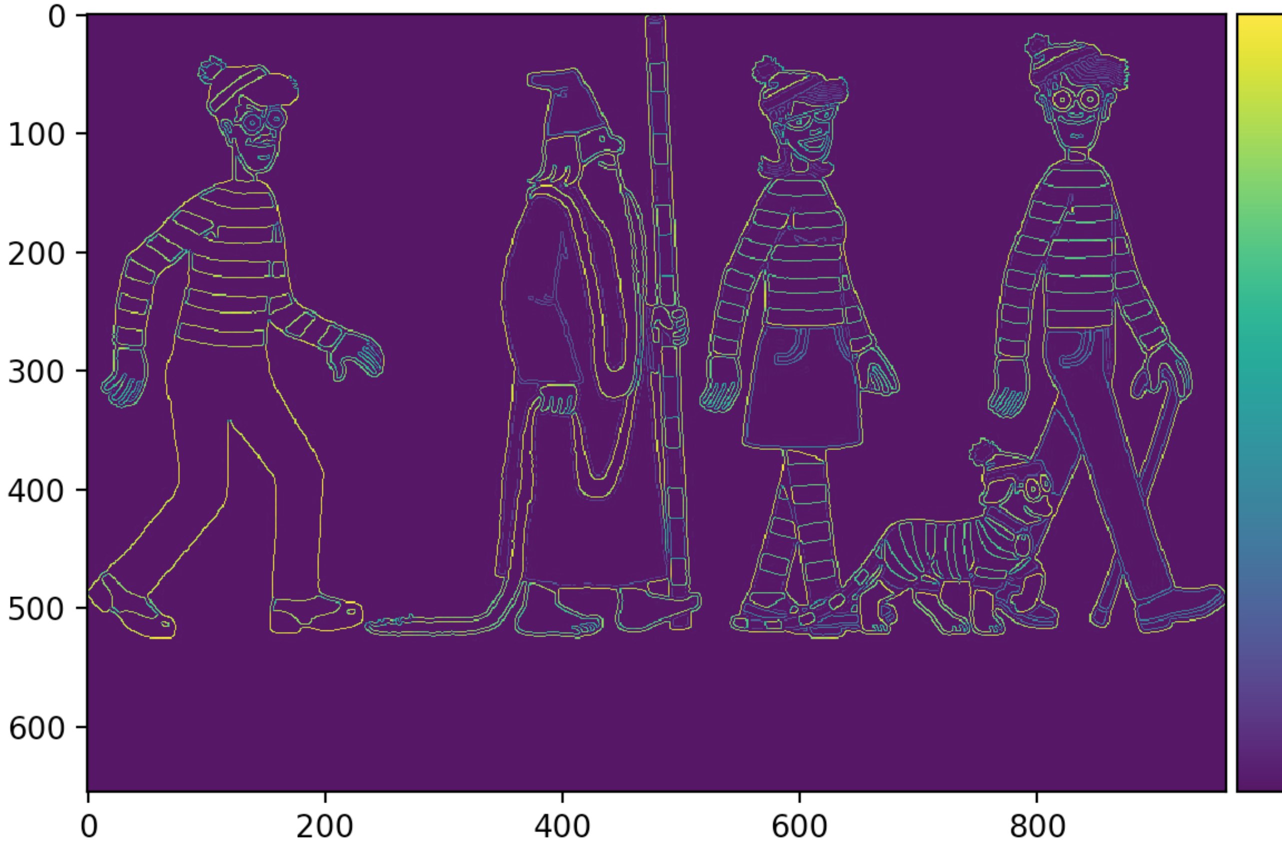

Figure 5: Canny Edge on waldo.png