CSC420_Coding

January 29, 2024

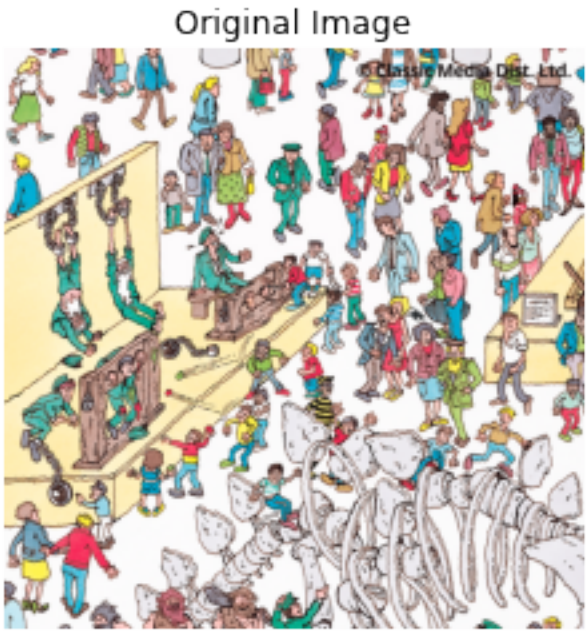
[ ]:
```
from __future__ import annotations
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.ndimage import correlate as corr
from scipy.ndimage import convolve as conv
from scipy.signal import correlate2d
import timeit
```

[ ]:
```
# The cross correlation function Question 1a
def cross_correlation_gray(image, kernel):
    '''
    image: A numpy.ndarray of size H x W
    kernel: A numpy.ndarray of size K x K where K is odd

    Perform 2D cross correlation on the image using the given filter

    return: A 2D numpy.ndarray of size H x W
    '''
    if len(image.shape) == 3:
        image_gray = np.dot(image, [0.2989, 0.5870, 0.1140])
    else:
        image_gray = image

    if len(kernel.shape) == 3:
        filter_gray = np.dot(kernel, [0.2989, 0.5870, 0.1140])
    else:
        filter_gray = kernel

    filter_gray_shape_x, filter_gray_shape_y = filter_gray.shape
    filter_gray_shape_x_floor = math.floor(filter_gray_shape_x / 2)
    filter_gray_shape_y_floor = math.floor(filter_gray_shape_y / 2)
    padded_image_gray = np.pad(image_gray, filter_gray_shape_x_floor,␣
 ↪mode='constant') # TODO: Need to generalize this for x and y borders

    final_image = []
```

```
    for i in range(padded_image_gray.shape[1]):
        final_image_row = []
        for j in range(padded_image_gray.shape[0]):
            padded_image_gray_section = padded_image_gray[j -␣
↪filter_gray_shape_x_floor : j + filter_gray_shape_x_floor + 1, i -␣
↪filter_gray_shape_y_floor : i + filter_gray_shape_y_floor + 1]

            if padded_image_gray_section.shape == filter_gray.shape:
                value_of_new_image = np.sum(filter_gray *␣
↪padded_image_gray_section)
            else:
                continue

            final_image_row.append(value_of_new_image)


        if len(final_image_row) != 0:
            final_image.append(final_image_row)

    return np.array(final_image).T
```

[ ]:
```
image = plt.imread('waldo.png')[...,:3]
plt.axis('off')
plt.title("Original Image")
plt.imshow(image, cmap='gray')
```

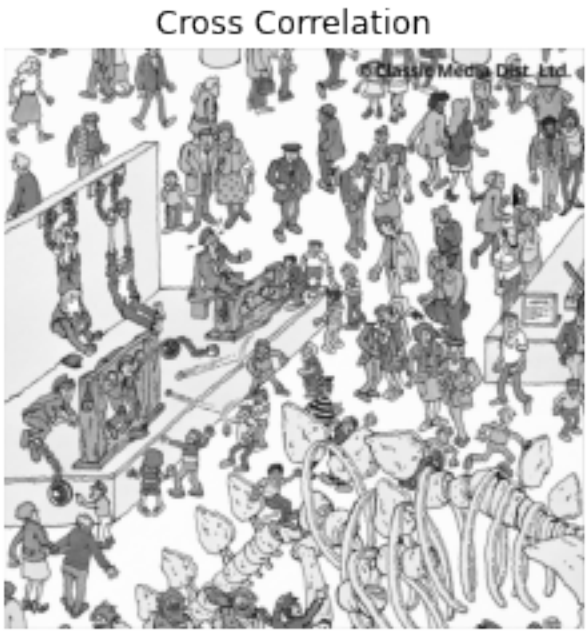
[ ]:
```
<matplotlib.image.AxesImage at 0x7f9143afdb50>
```



[ ]:
```
image = np.dot(image, [0.2989, 0.5870, 0.1140])
filter_first = np.array([
    [0,     0.5,   0],
    [0.125, 0.5, 0.5],
    [0, 0.125, 0.125]
])
filter_waldo = plt.imread('template.png')[...,:3]
cross_correlated = cross_correlation_gray(image, filter_first)
plt.axis('off')
plt.title("Cross Correlation")
plt.imshow(cross_correlated, cmap='gray')
```

```
[[1.23375833 1.35713417 1.35713417 … 1.35525662 1.35504064 0.73808672]
 [1.72726167 1.8506375  1.8506375  … 1.84812858 1.85155961 1.23054329]
 [1.72726167 1.8506375  1.8506375  … 1.84737182 1.8497478  1.23009843]
 …
 [1.53830305 1.63234683 1.62235948 … 1.71345565 1.7181253  1.14361868]
 [1.5234731  1.62901786 1.62207256 … 1.71812001 1.71368231 1.14442074]
 [1.3077008  1.408773   1.41205756 … 1.49648898 1.48310633 1.02715633]]
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f9143b21d60>
```

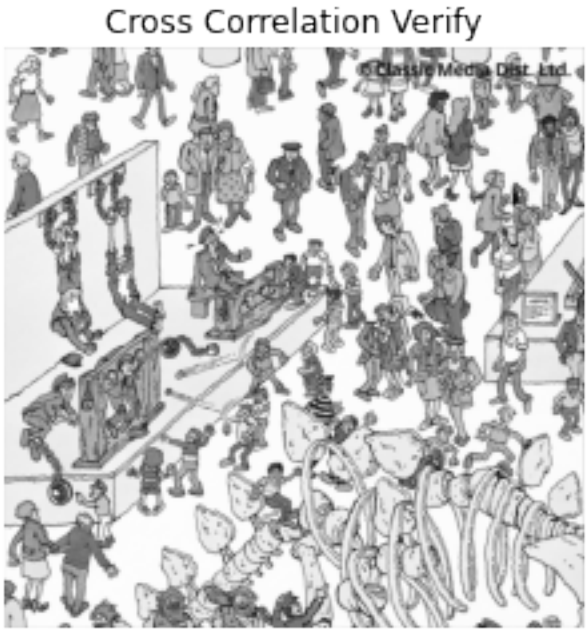


[ ]:
```
# Verify if answer is correct with built in function
verification = correlate2d(image, filter_first, mode='same', boundary='fill')
```

```
print(verification)
plt.axis('off')
plt.title("Cross Correlation Verify")
plt.imshow(verification, cmap='gray')
```

```
[[1.23375833 1.35713417 1.35713417 … 1.35525662 1.35504064 0.73808672]
 [1.72726167 1.8506375  1.8506375  … 1.84812858 1.85155961 1.23054329]
 [1.72726167 1.8506375  1.8506375  … 1.84737182 1.8497478  1.23009843]
 …
 [1.53830305 1.63234683 1.62235948 … 1.71345565 1.7181253  1.14361868]
 [1.5234731  1.62901786 1.62207256 … 1.71812001 1.71368231 1.14442074]
 [1.3077008  1.408773   1.41205756 … 1.49648898 1.48310633 1.02715633]]
```

[ ]: <matplotlib.image.AxesImage at 0x7f91540cb6d0>



[ ]:
```
# Function to verify if filter is separable Question 1b
def is_separable(kernel):
    '''
    filter_: A numpy.ndarray of size K x K where K is odd

    Verifies if a filter is separable or not by the property of a single value␣
 ↪being non-zero

    return: The two separated vectors that can form the kernel if kernel is␣
 ↪separable, else None
    '''
```

```
    U, S, V = np.linalg.svd(kernel)

    if np.sum(np.isclose(S, 0, 1e-15)) == len(S) - 1:
        U = U * np.sqrt(S[0])
        V = V * np.sqrt(S[0])
        return (True, U[:, 0], V[0, :])

    return (False, None, None)
```

[ ]:
```
# Check if given filter is separable
if is_separable(filter_first)[0]:
    print("The given filter in the pdf is separable")
else:
    print("The given filter in the pdf is not separable")
```

```
The given filter in the pdf is not separable
```

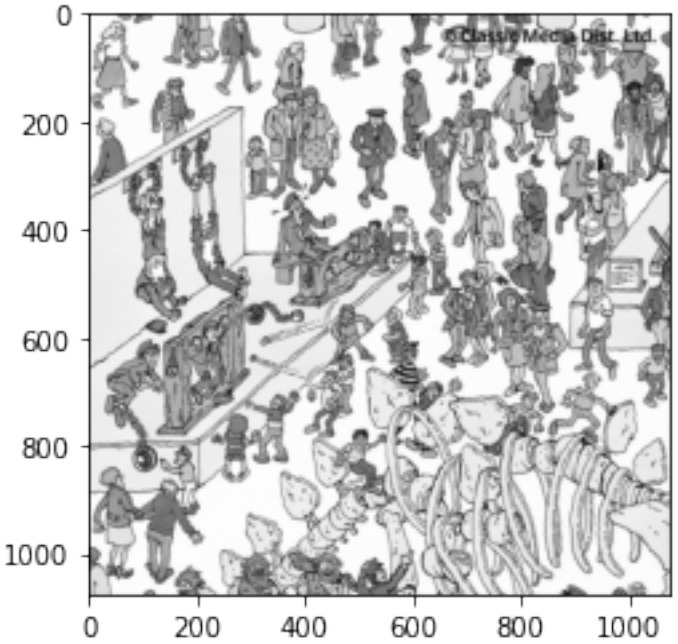
[ ]:
```
# Question 1c
def cross_correlation_gray_faster(image, kernel):
    '''
    image: A numpy.ndarray of size H x W x 3 or H x W x 1 (RGB or Grayscale)
    kernel: A numpy.ndarray of size K x K where K is odd

    Perform 2D cross correlation on the image using the given filter and takes
    advantage of filters that are separable for efficiency

    return: A 2D numpy.ndarray of size H x W
    '''
    if len(image.shape) == 3:
        image_gray = np.dot(image, [0.2989, 0.5870, 0.1140])
    else:
        image_gray = image

    if len(kernel.shape) == 3:
        filter_gray = np.dot(kernel, [0.2989, 0.5870, 0.1140])
    else:
        filter_gray = kernel

    if is_separable(filter_gray)[0] is None:
        return cross_correlation_gray(image_gray, filter_gray)

    _, horizontal, vertical = is_separable(filter_gray)

    filter_gray_shape_x, filter_gray_shape_y = filter_gray.shape
    filter_gray_shape_x_floor = math.floor(filter_gray_shape_x / 2)
    filter_gray_shape_y_floor = math.floor(filter_gray_shape_y / 2)
```

```
  padded_image_gray = np.pad(image_gray, filter_gray_shape_x_floor,␣
↪mode='constant')

  final_image_x = []

  for i in range(padded_image_gray.shape[1]):

      final_image_row = []
      for j in range(padded_image_gray.shape[0]):
          padded_image_gray_section = padded_image_gray[j -␣
↪filter_gray_shape_x_floor : j + filter_gray_shape_x_floor + 1, i]

          if padded_image_gray_section.shape[0] == filter_gray_shape_x:
              value_of_new_image = np.sum(horizontal *␣
↪padded_image_gray_section)
          else:
              continue

          final_image_row.append(value_of_new_image)

      if len(final_image_row) != 0:
          final_image_x.append(final_image_row)

  final_image_x = np.array(final_image_x)
  final_image = []
  for i in range(final_image_x.shape[0]):

      final_image_new_row = []
      for j in range(final_image_x.shape[1]):

          padded_image_gray_section = final_image_x[i -␣
↪filter_gray_shape_y_floor : i + filter_gray_shape_y_floor + 1, j]
          if padded_image_gray_section.shape[0] == filter_gray_shape_y:
              the_value_of_new_image = np.sum(vertical *␣
↪padded_image_gray_section)
          else:
              continue

          final_image_new_row.append(the_value_of_new_image)

      if len(final_image_new_row) != 0:
          final_image.append(final_image_new_row)

  return np.array(final_image).transpose()
```

[ ]:
```
# Gaussian Filters are separable
my_filter = gaussian_filter(3, 2)

if is_separable(my_filter)[0]:
    print("The given filter is separable")
else:
    print("The given filter is not separable")

image = plt.imread('waldo.png')[...,:3]
image = np.dot(image, [0.2989, 0.5870, 0.1140])
plt.imshow(correlate2d(image, my_filter, mode='same', boundary='fill'),␣
 ↪cmap='gray')
```

```
The given filter is separable
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f9143b35d00>
```



[ ]:
```
# Show runtime comparison
result_1 = timeit.timeit(lambda: cross_correlation_gray(image, my_filter),␣
 ↪number=1)
result_2 = timeit.timeit(lambda: cross_correlation_gray_faster(image,␣
 ↪my_filter), number=1)

print(f"The time for non separable filter {result_1}")
print("========")
print(f"The time for separable filter {result_2}")
```

```
print("========")
cross_correlation_gray_faster(image, my_filter)
```

```
The time for non separable filter 13.510282814000675
========
The time for separable filter 21.412632890998793
========
```

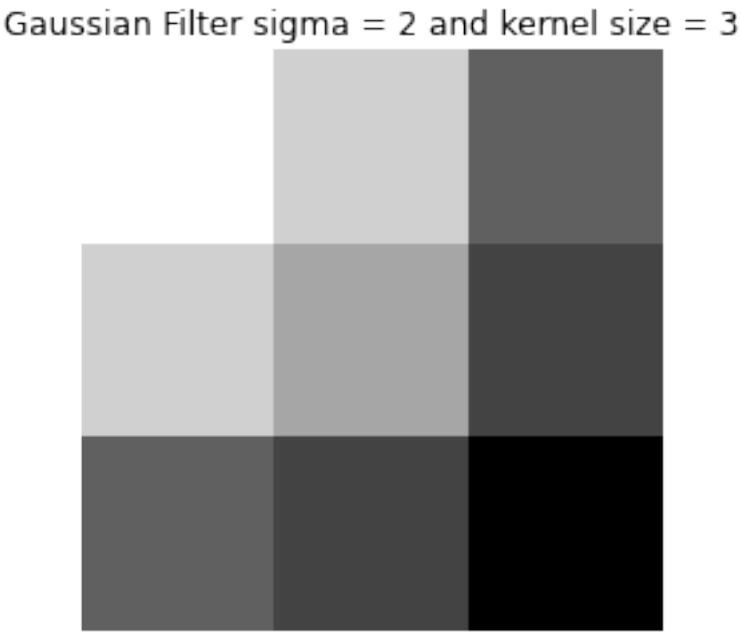
[ ]:
```
array([[0.08707344, 0.14555015, 0.14555015, …, 0.14527039, 0.14533587,
        0.11003558],
       [0.14555015, 0.24329862, 0.24329862, …, 0.24270378, 0.24297008,
        0.18391018],
       [0.14555015, 0.24329862, 0.24329862, …, 0.24245562, 0.242852  ,
        0.18378892],
       …,
       [0.12927702, 0.2154255 , 0.21383029, …, 0.22656648, 0.22566639,
        0.17072006],
       [0.12869593, 0.21476184, 0.21359814, …, 0.2265865 , 0.22566923,
        0.17036973],
       [0.09734543, 0.16235151, 0.16122982, …, 0.1718212 , 0.17095714,
        0.1288004 ]])
```

[ ]:
```
# Question 1d

# You can still take advantage of separability. Since the filter is flipped␣
 ↪horizontally and vertically
# the new filter may not be separable, but if it is, then we can take advantage␣
 ↪of separability because
# we essentially perform cross correlation on the flipped filter
def convolution_faster(image, kernel):
    '''
    image: A numpy.ndarray of size H x W x 3 or H x W x 1 (RGB or Grayscale)
    kernel: A numpy.ndarray of size K x K where K is odd

    Perform 2D convolution on the image using the given filter

    return: A 2D numpy.ndarray of size H x W
    '''
    if len(image.shape) == 3:
        image_gray = np.dot(image, [0.2989, 0.5870, 0.1140])
    else:
        image_gray = image

    if len(kernel.shape) == 3:
        filter_gray = np.dot(kernel, [0.2989, 0.5870, 0.1140])
    else:
        filter_gray = kernel
```

```
    if is_separable(filter_gray) is None:
        return cross_correlation_gray(image_gray, filter_gray)

    kernel_flipped = kernel[::-1, ::-1]
    return cross_correlation_gray_faster(image, kernel_flipped)
```

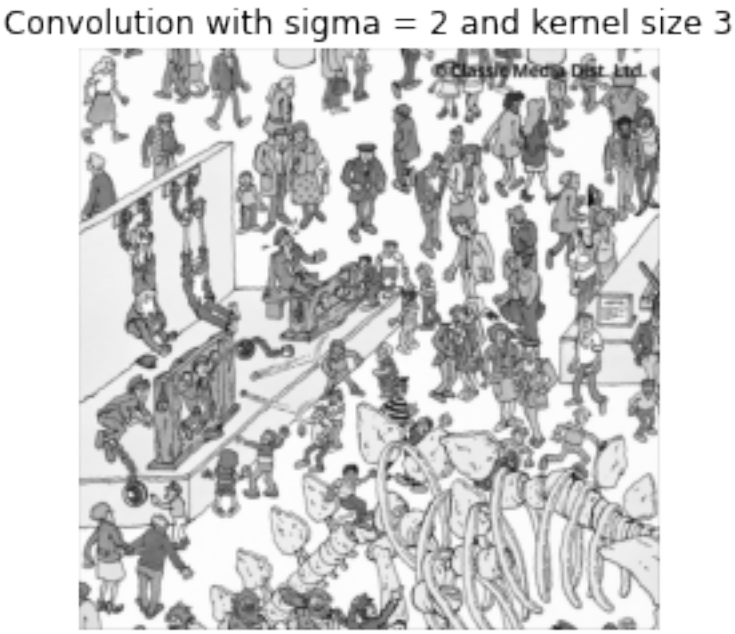
[ ]:
```
plt.axis('off')
plt.title("Gaussian Filter sigma = 2 and kernel size = 3")
plt.imshow(my_filter, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f913f5012e0>
```



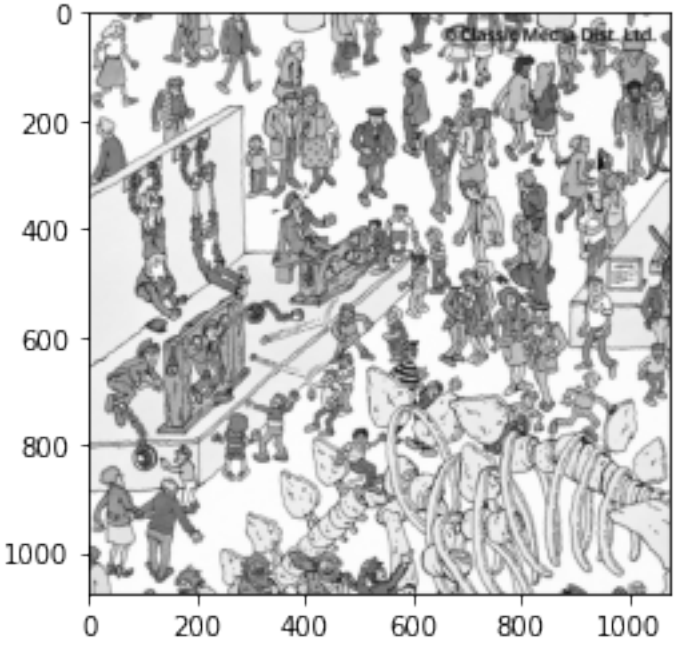
[ ]:
```
image_convoluted = convolution_faster(image, my_filter)
plt.axis('off')
plt.title("Convolution with sigma = 2 and kernel size 3")
plt.imshow(image_convoluted, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f913958f8e0>
```



[ ]:
```
image_correlated = cross_correlation_gray_faster(image, my_filter)
plt.imshow(image_correlated, cmap='gray')
```

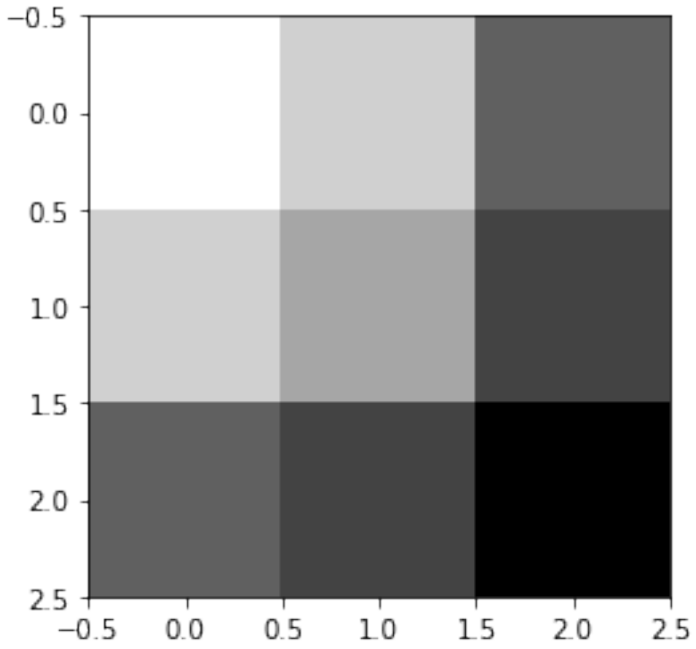
[ ]: <matplotlib.image.AxesImage at 0x7f9136a57970>



[ ]:
```
# Question 2 creating a gaussin filter and placing it on waldo.png
def gaussian_filter(size, sigma):
    '''
    size: an int determining the size of the filter
    sigma: an int determining the standard deviation of the filter

    Generates a numpy.ndarray of size K x K Gaussian filter of the given size␣
 ↪and standard deviation.
    '''
    kernel = np.fromfunction(lambda x, y: (1 / (2 * math.pi * sigma ** 2)) * np.
 ↪exp(-(x ** 2 + y ** 2) / (2 * sigma ** 2)), (size, size))
    return kernel
```

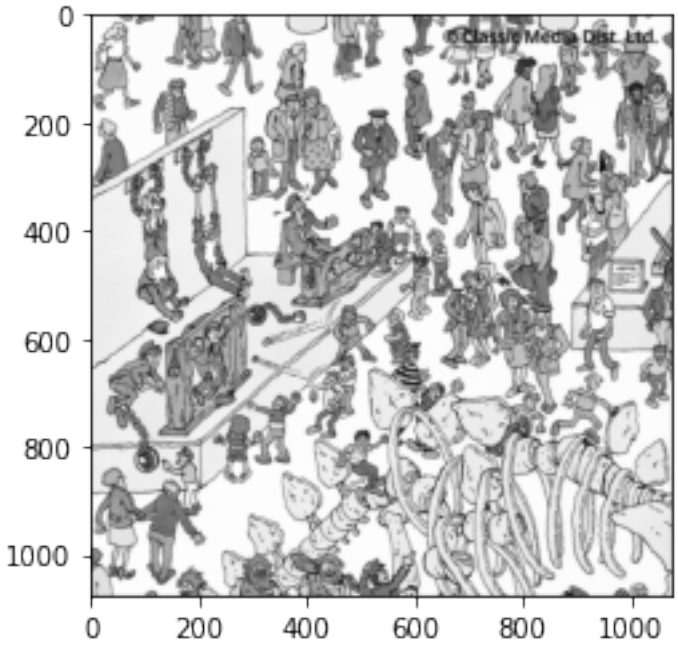
[ ]:
```
gaussian_3_2 = gaussian_filter(3, 2)
plt.imshow(gaussian_3_2, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f913f4f5f10>
```



[ ]:
```
image_convoluted = convolution_faster(image, gaussian_3_2)
plt.imshow(image_convoluted, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f914c534670>
```

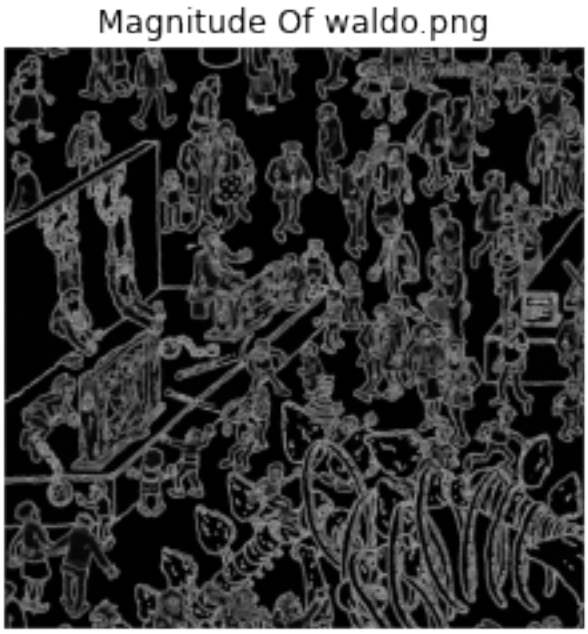


[ ]:
```python
# Question 3a
def gradient_magnitutde_direction(image):

    if len(image.shape) == 3:
        image_gray = np.dot(image, [0.2989, 0.5870, 0.1140])
    else:
        image_gray = image

    Sobel_x = np.array([[-1,0,1]])
    Sobel_y = np.array([[1,2,1]])
    Sobel_kernel_y = Sobel_x.T @ Sobel_y
    Sobel_kernel_x = Sobel_y.T @ Sobel_x


    edges_y = conv(image_gray, Sobel_kernel_y, mode='constant')
    edges_x = corr(image_gray, Sobel_kernel_x, mode='constant')

    magnitude = (edges_x**2 + edges_y**2)**(0.5)

    # Handle division by zero cases using numpy.where()
    direction = np.where(edges_x == 0, 0, np.arctan2(edges_y, edges_x))
    direction[direction < 0] += 2 * np.pi

    return (magnitude, direction)
```

[ ]:
```
image = plt.imread('waldo.png')[...,:3]
template = plt.imread('template.png')[...,:3]
magnitude_waldo, direction_waldo = gradient_magnitutde_direction(image)
magnitude_template, direction_template = gradient_magnitutde_direction(template)
```

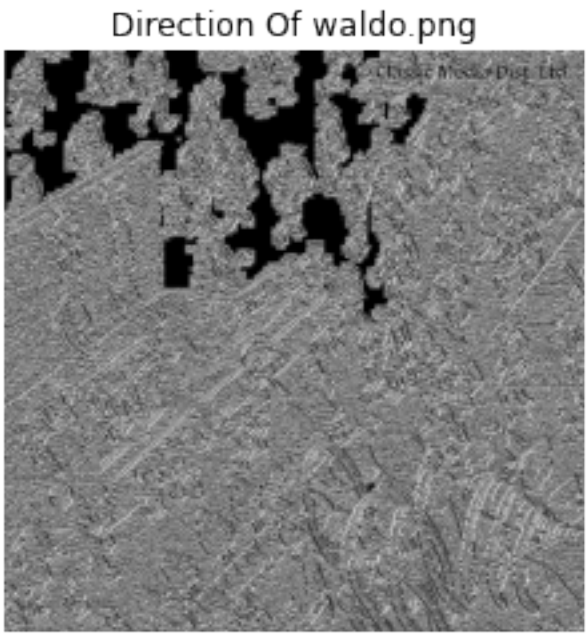
[ ]:
```
plt.axis('off')
plt.title("Magnitude Of waldo.png")
plt.imshow(magnitude_waldo, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f910585ad30>
```



[ ]:
```
plt.axis('off')
plt.title("Direction Of waldo.png")
plt.imshow(direction_waldo, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f91059cd850>
```



[ ]:
```
plt.axis('off')
plt.title("Magnitude Of template.png")
plt.imshow(magnitude_template, cmap='gray')
```

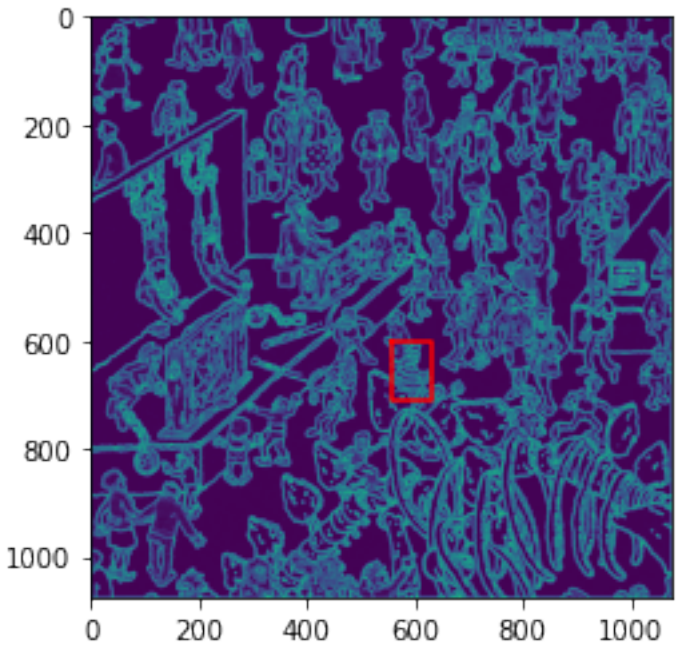
[ ]:
```
<matplotlib.image.AxesImage at 0x7f9105c71c70>
```

[ ]:
```
plt.axis('off')
plt.title("Direction Of template.png")
plt.imshow(direction_template, cmap='gray')
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f9105c45850>
```

[ ]:
```
# Question 3b
def localize(image, kernel):
    _, W = image.shape
    h, w = kernel.shape
    correlation = corr(image, kernel)
    max_point = np.argmax(correlation)
    m_h, m_w = max_point//W, max_point%W
    bbox = np.array([[m_h-h//2, m_w-w//2],[m_h-h//2, m_w+w//2],[m_h+h//2, m_w+w/
 ↪/2],[m_h+h//2, m_w-w//2],[m_h-h//2, m_w-w//2]])
    plt.imshow(image)
    plt.plot(bbox[:, 1], bbox[:, 0], 'r')
    plt.show()
```

[ ]:
```
localize(magnitude_waldo, magnitude_template)
```



[ ]:
```python
# Question 4
def canny_edge_detector(image):

    if len(image.shape) == 3:
        image_gray = np.dot(image, [0.2989, 0.5870, 0.1140])
    else:
        image_gray = image

    kernel = gaussian_filter(5, 1)
    image_gray = conv(image_gray, kernel)

    magnitude, direction = gradient_magnitutde_direction(image_gray)

    final = np.zeros(image_gray.shape)

    # Ranges of angles are acquired from https://towardsdatascience.com/
↪canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123
    # and chatGPT
    for i in range(1, magnitude.shape[0] - 1):
        for j in range(1, magnitude.shape[1] - 1):

            angle = direction[i, j]
            value = magnitude[i, j]

            if (0 <= angle < np.pi / 8) or (7 * np.pi / 8 <= angle < np.pi):
```

```
                neighbor_1 = magnitude[i, j + 1]
                neighbor_2 = magnitude[i, j - 1]

            elif (np.pi / 8 <= angle < 3 * np.pi / 8):
                neighbor_1 = magnitude[i - 1, j - 1]
                neighbor_2 = magnitude[i + 1, j + 1]

            elif (3 * np.pi / 8 <= angle < 5 * np.pi / 8):
                neighbor_1 = magnitude[i - 1, j]
                neighbor_2 = magnitude[i + 1, j]

            else:
                neighbor_1 = magnitude[i - 1, j + 1]
                neighbor_2 = magnitude[i + 1, j - 1]


        if value >= neighbor_1 and value >= neighbor_2:
            final[i, j] = value

    return final
```

[ ]:
```
canny = canny_edge_detector(image)
plt.imshow(canny, cmap='gray')
```

[ ]: <matplotlib.image.AxesImage at 0x7f9105940d30>