```
import numpy as np
import matplotlib.pyplot as plt
import time
from scipy.ndimage import correlate, convolve
from skimage import feature
from sklearn.preprocessing import normalize
```

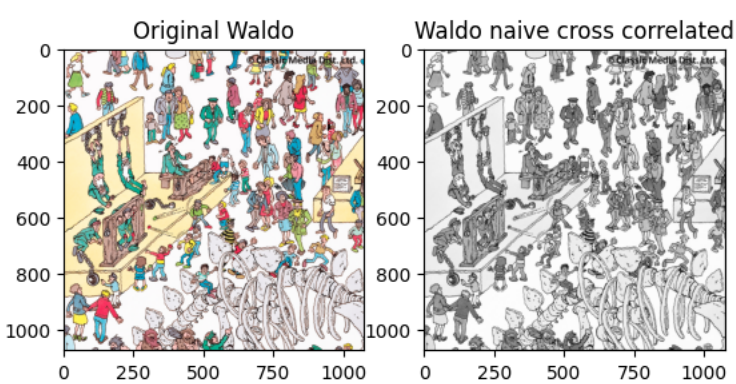
Setup and preparations

```
original_waldo = plt.imread("waldo.png")
original_template = plt.imread("template.png")

kernel = np.array([[0, 0.5, 0],
                [0.125, 0.5, 0.5],
                [0, 0.125, 0.125]])
# kernel = np.array([[0.25,1,0.25],[1,4,1],[0.25,1,0.25]])
separable_filter = np.array([[0.25,1,0.25],[1,4,1],[0.25,1,0.25]])
```

Q1 (a) Naive Cross Correlation

```
def naive_cross_corr(image: np.ndarray, kernel: np.ndarray):
    if len(image.shape) > 2:
        image = np.dot(image[...,:3], [0.299, 0.587, 0.114])

    v_padding = kernel.shape[0]//2
    h_padding = kernel.shape[1]//2
    padded_image = np.pad(image, [(v_padding, v_padding),(h_padding, h_padding)], mode='constant')

    result = np.zeros([image.shape[0], image.shape[1]])
    for i in range(v_padding, padded_image.shape[0] - v_padding):
        for j in range(h_padding, padded_image.shape[1] - h_padding):
            result[i-v_padding, j-h_padding] = np.sum(padded_image[i-v_padding:i+v_padding+1, j-h_padding:j+h_padding+1] * kernel
    return result
```

```
start_time = time.time()
naive_cross_corr_result = naive_cross_corr(original_waldo, kernel)
end_time = time.time()
naive_cross_corr_time = end_time - start_time

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(original_waldo)
axarr[0].set_title("Original Waldo")
axarr[1].imshow(naive_cross_corr_result, cmap='gray')
axarr[1].set_title("Waldo naive cross correlated")
plt.show()
```



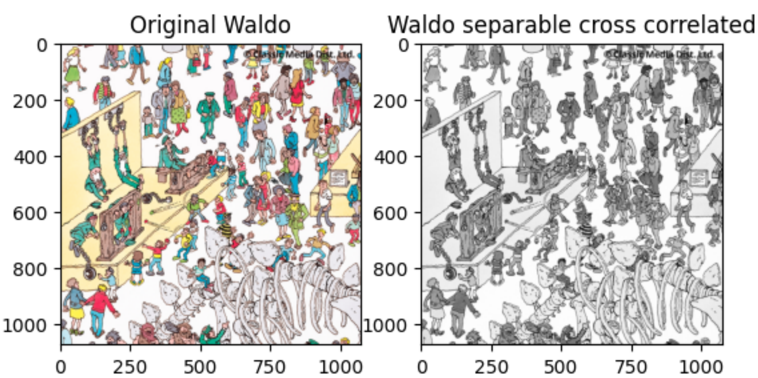
Q1 (b) Is filter separable?

```
def is_separable(kernel: np.ndarray):
    _,s,_ = np.linalg.svd(kernel)
    return s[0] > 1e-5 and all(x <= 1e-5 for x in s[1:])
```

Q1 (c) Separable Cross-Correlation

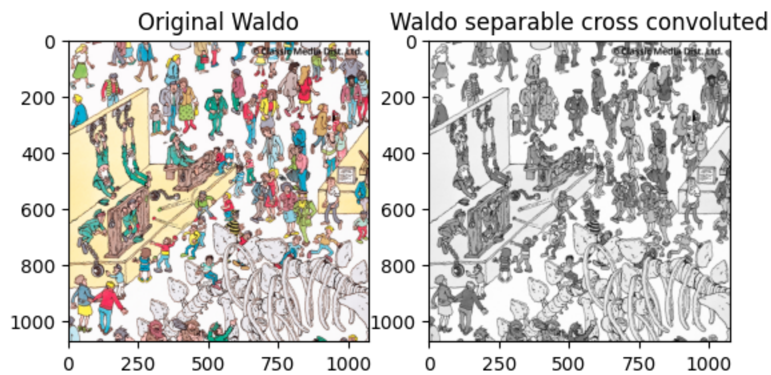
```
def separable_cross_corr(image: np.ndarray, kernel: np.ndarray):
    if len(image.shape) > 2:
        image = np.dot(image[...,:3], [0.299, 0.587, 0.114])

    if not is_separable(kernel):
        kernel = separable_filter

    u,s,v = np.linalg.svd(kernel)
    v_vector = (u[:,0] * np.sqrt(s[0])).reshape((u.shape[0],1))
    h_vector = (v[0,:] * np.sqrt(s[0])).reshape((1,v.shape[1]))

    cross_corr_intermediate = naive_cross_corr(image, h_vector)
    final_cross_corr = naive_cross_corr(cross_corr_intermediate, v_vector)

    return final_cross_corr

start_time = time.time()
separable_cross_corr_result = separable_cross_corr(original_waldo, kernel)
end_time = time.time()
separable_cross_corr_time = end_time - start_time

print(f"Naive Cross-corr time: {naive_cross_corr_time} sec")
print(f"Separable Cross-corr time: {separable_cross_corr_time} sec")

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(original_waldo)
axarr[0].set_title("Original Waldo")
axarr[1].imshow(separable_cross_corr_result, cmap='gray')
axarr[1].set_title("Waldo separable cross correlated")
plt.show()
```

```
Naive Cross-corr time: 6.534672260284424 sec
Separable Cross-corr time: 9.122668027877808 sec
```



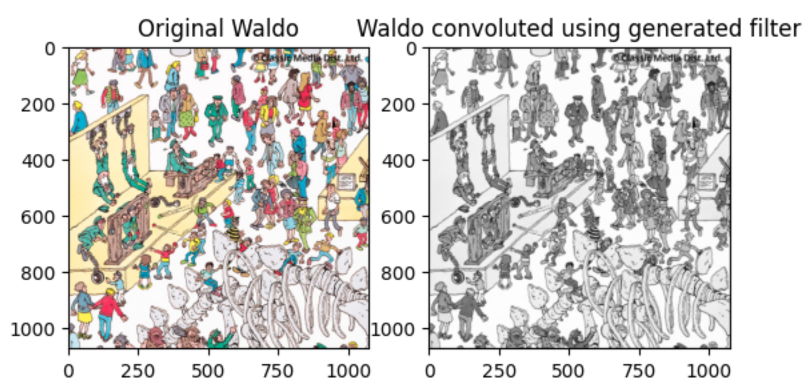
Q1 (d) Separable Convolution

```
def separable_cross_conv(image: np.ndarray, kernel: np.ndarray):
    if len(image.shape) > 2:
        image = np.dot(image[...,:3], [0.299, 0.587, 0.114])

    if not is_separable(kernel):
        kernel = separable_filter

    u,s,v = np.linalg.svd(np.flip(kernel))
    v_vector = (u[:,0] * np.sqrt(s[0])).reshape((u.shape[0],1))
    h_vector = (v[0,:] * np.sqrt(s[0])).reshape((1,v.shape[1]))
    cross_conv_intermediate = naive_cross_corr(image, h_vector)
    final_cros_conv = naive_cross_corr(cross_conv_intermediate, v_vector)

    return final_cros_conv

separable_cross_conv_result = separable_cross_conv(original_waldo, kernel)

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(original_waldo)
axarr[0].set_title("Original Waldo")
axarr[1].imshow(separable_cross_conv_result, cmap='gray')
axarr[1].set_title("Waldo separable cross convoluted")
plt.show()
```



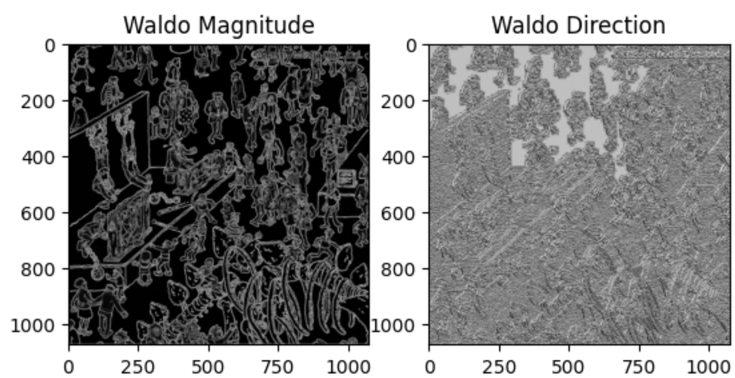
Q2 Implementation of Gaussian filter generator

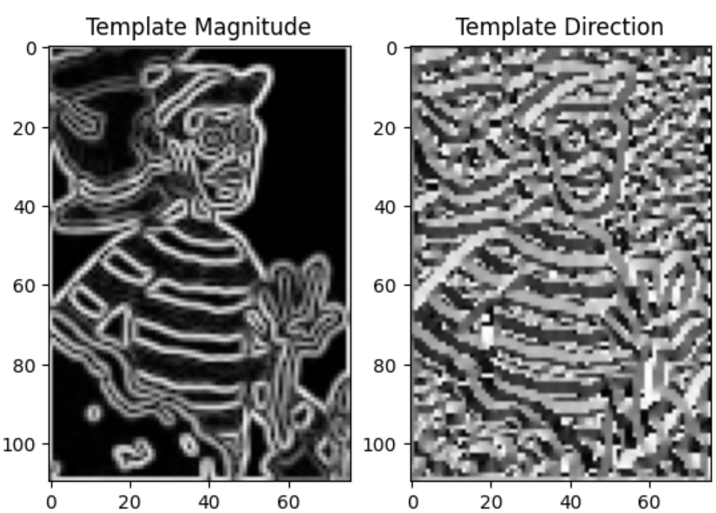
```
def generate_gaussian_filter(size: int, sigma: int):
    x, y = np.meshgrid(np.arange(-size//2 + 1, size//2 + 1), np.arange(-size//2 + 1, size//2 + 1))
    normal = 1 / (2 * np.pi * sigma**2)
    kernel = normal * np.exp(-(x**2 + y**2) / (2 * sigma**2))
    return kernel

new_waldo = convolve(np.dot(original_waldo[...,:3], [0.299, 0.587, 0.114]), generate_gaussian_filter(3, 2), mode='constant')

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(original_waldo)
axarr[0].set_title("Original Waldo")
axarr[1].imshow(new_waldo, cmap='gray')
axarr[1].set_title("Waldo convoluted using generated filter")
plt.show()
```



Q3 (a) Gradients computation

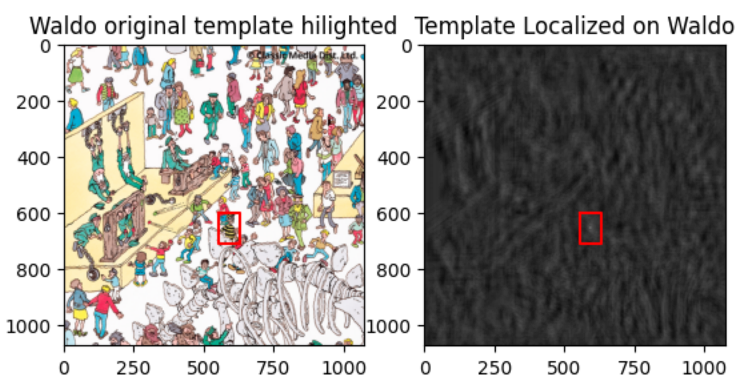
```
def gradient_computation(image: np.ndarray):
    if len(image.shape) > 2:
        image = np.dot(image[...,:3], [0.299, 0.587, 0.114])

    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]])

    gradient_x = correlate(image, sobel_x, mode='constant')
    gradient_y = correlate(image, sobel_y, mode='constant')

    magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
    direction = np.arctan2(gradient_y, gradient_x)

    return magnitude, direction

waldo_magnitude, waldo_direction = gradient_computation(original_waldo)
template_magnitude, template_direction = gradient_computation(original_template)

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(waldo_magnitude, cmap=plt.get_cmap('gray'))
axarr[0].set_title("Waldo Magnitude")
axarr[1].imshow(waldo_direction, cmap=plt.get_cmap('gray'))
axarr[1].set_title("Waldo Direction")
plt.show()

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(template_magnitude, cmap=plt.get_cmap('gray'))
axarr[0].set_title("Template Magnitude")
axarr[1].imshow(template_direction, cmap=plt.get_cmap('gray'))
axarr[1].set_title("Template Direction")
plt.show()
```





Q3 (b) Localize template on image

```
def localize(image: np.ndarray, template: np.ndarray):
    if len(image.shape) > 2:
        image = np.dot(image[...,:3], [0.299, 0.587, 0.114])
    if len(template.shape) > 2:
        template = np.dot(template[...,:3], [0.299, 0.587, 0.114])

    image_magnitude, _ = gradient_computation(image)
    template_magnitude, _ = gradient_computation(template)

    normalized_image_magitude = (image_magnitude-np.mean(image_magnitude))/(np.max(image_magnitude)-np.min(image_magnitude))
    normalized_template_magitude = (template_magnitude-np.mean(template_magnitude))/(np.max(template_magnitude)-np.min(template_
    normalized_cross_corr = correlate(normalized_image_magitude, normalized_template_magitude, mode='constant')

    return normalized_cross_corr


localization = localize(original_waldo, original_template)

max_point = np.argmax(localization)
image_width = original_waldo.shape[1]
template_height, template_width, _ = original_template.shape
max_point_x, max_point_y = max_point // image_width, max_point % image_width
bbox = np.array([[max_point_x - template_height // 2, max_point_y - template_width // 2],
                 [max_point_x + template_height // 2, max_point_y - template_width // 2],
                 [max_point_x + template_height // 2, max_point_y + template_width // 2],
                 [max_point_x - template_height // 2, max_point_y + template_width // 2],
                 [max_point_x - template_height // 2, max_point_y - template_width // 2]])

f, axarr = plt.subplots(1, 2)
axarr[0].imshow(original_waldo, cmap=plt.get_cmap('gray'))
axarr[0].set_title("Waldo original template hilighted")
axarr[0].plot(bbox[:, 1], bbox[:, 0], 'r')

axarr[1].imshow(localization, cmap=plt.get_cmap('gray'))
axarr[1].set_title("Template Localized on Waldo")
axarr[1].plot(bbox[:, 1], bbox[:, 0], 'r')
plt.show()
```



Q4 Canny edge detector

```python
def canny_edge_detector(image: np.ndarray):
    if len(image.shape) > 2:
        image = np.dot(image[...,:3], [0.299, 0.587, 0.114])

    kernel = generate_gaussian_filter(3, 2)
    correlated = correlate(image, kernel, mode='constant')
    gradient_magnitude, gradient_direction = gradient_computation(correlated)

    suppressed_magnitude = np.zeros(gradient_magnitude.shape)

    for i in range(1, gradient_magnitude.shape[0] - 1):
        for j in range(1, gradient_magnitude.shape[1] - 1):
            angle = abs(gradient_direction[i, j])

            if (0 <= angle < np.pi / 8) or (7 * np.pi / 8 <= angle <= np.pi):
                neighbors = [gradient_magnitude[i, j - 1], gradient_magnitude[i, j + 1]]
            elif np.pi / 8 <= angle < 3 * np.pi / 8:
                neighbors = [gradient_magnitude[i + 1, j - 1], gradient_magnitude[i - 1, j + 1]]
            elif 3 * np.pi / 8 <= angle < 5 * np.pi / 8:
                neighbors = [gradient_magnitude[i + 1, j], gradient_magnitude[i - 1, j]]
            elif 5 * np.pi / 8 <= angle < 7 * np.pi / 8:
                neighbors = [gradient_magnitude[i + 1, j + 1], gradient_magnitude[i - 1, j - 1]]

            if gradient_magnitude[i, j] >= max(neighbors):
                suppressed_magnitude[i, j] = gradient_magnitude[i, j]

    return suppressed_magnitude

canny_edges = canny_edge_detector(original_waldo)

plt.imshow(canny_edges, cmap=plt.get_cmap('gray'))
plt.title("Canny edge detection on waldo")
plt.show()
```