CSC420 Assignment 1

Brendan Neal | 1001160236 | nealbre1

Imports and some helper functions

```
In [1]: import numpy as np
        import cv2 as cv
        from scipy import signal
        from scipy.ndimage.filters import gaussian_filter, laplace
        from matplotlib import pyplot as plot

        # Make the plot a certain size
        plot.rcParams["figure.figsize"] = [8, 6]

        # Shows an image, and saves it if a filename is given
        def display_image(img, file_name=None):

            flt_img = img.astype(float)
            img_max, img_min = np.max(flt_img), np.min(flt_img)

            norm_img = (((flt_img - img_min) / (img_max - img_min)) * 255).astype(np.uint8)

            if len(img.shape) == 2:
                plot.imshow(norm_img, cmap='gray')
            elif (len(img.shape) == 3):
                plot.imshow(cv.cvtColor(norm_img, cv.COLOR_BGR2RGB))

            plot.show()

            if file_name:
                cv.imwrite(file_name, norm_img)


        # Creates a normalized filter (divided by sum of absolute elements)
        def norm_filter(filt):
            return np.array(filt).astype(float) / np.absolute(filt).sum()
```

Question 1: Correlation Implementation

In [2]:
```
# So I don't have to type out quotes
valid, same, full = "valid", "same", "full"


def correlation(i, f, mode):
    """(numpy.array, numpy.array, str) -> numpy.array
    Correlates an image with a filter, according to some boundary rule
    :param i: a numpy-array representation of the image (grayscale or color)
    :param f: a numpy-array representation of a 2D filter
    :param mode: how to deal with image boundaries (valid, same, full)
    :return: i correlated with f
    """

    # Can only take 2D filters, 2D or 3D (RGB) images
    assert len(f.shape) == 2
    assert len(i.shape) in [2, 3]

    # If we are given a color image, execute for each color
    if len(i.shape) == 3:
        return np.stack(
            (
                correlation(i[:, :, color], f, mode)
                for color in range(i.shape[-1])
            ),
            axis=2
        )

    # Create the zero-padded array, cast as floats to allow for proper multiplication
    patch_h, patch_w = f.shape
    padded = corr_padding(i, patch_h, patch_w, mode).astype(float)

    # View it in terms of patches
    patches = patch_view(padded, patch_h, patch_w)

    # Multiply f element-wise with each patch, and sum up within patches
    # Convert back to the original data type
    return (patches[:, :] * f).sum(axis=(2, 3)).astype(i.dtype)


def patch_view(arr, patch_h, patch_w):
    """(numpy.array, int, int) -> numpy.array
    Creates a 4D, read-only view of some 2D numpy array as a 2D array of 2D patches.
    There are (arr_height x arr_width) patches and each patch is of size (patch_h, patch_w)

    :param arr: some 2D numpy array
    :param patch_h: desired height for patches
    :param patch_w: desired width for patches
    :return: 4D array of (patch_h x patch_w) patches
    """
    assert len(arr.shape) == 2

    # Numpy stride code examples (magic):
    # https://stackoverflow.com/questions/16774148/
    # https://github.com/keras-team/keras/issues/2983

    # New height and width are now going to be in terms of
    # number of overlapping patches we can fit into arr
    new_h, new_w = np.array(arr.shape) - \
                   np.array([patch_h, patch_w]) + \
                   1

    return np.lib.stride_tricks.as_strided(
        np.ascontiguousarray(arr),
        shape=(new_h, new_w, patch_h, patch_w),
        strides=arr.strides + arr.strides,
        writeable=False
    )


def corr_padding(arr, patch_h, patch_w, mode):
    """(numpy.array, int, int, str) -> numpy.array

    Creates a new 2D array that is zero-padded for correlation according to mode.
    valid = no padding
    same = padding thickness of floor(patch_dim/2)
    full = padding thickness of patch_dim - 1
    :param arr: some 2D numpy array
    :param patch_h: desired height for patches
    :param patch_w: desired width for patches
    :param mode: how to deal with image boundaries (valid, same, full)
    :return: padded 2D numpy array
    """

    assert mode in [valid, same, full]

    # Decide how much to pad in each direction (h = height, w = width)
    if mode == valid:
        pad_h, pad_w = 0, 0
    elif mode == same:
        pad_h, pad_w = patch_h // 2, patch_w // 2
    elif mode == full:
        pad_h, pad_w = patch_h - 1, patch_w - 1

    # Create the padded array
    return np.pad(
        arr, ((pad_h, pad_h), (pad_w, pad_w)),
        mode='constant', constant_values=0
    )
```

Question 2: Convolution via Correlation

Note that the $k \times k$ Gaussian filter $G_{k \times k}(\sigma_x, \sigma_y)$ is separable:

$$G(\sigma_x, \sigma_y)_{k \times k} = G_{k \times 1}(\sigma_y) * G_{1 \times k}(\sigma_x)$$

Therefore, for some image I:

$$I * G(\sigma_x, \sigma_y)_{k \times k} = I * G_{k \times 1}(\sigma_y) * G_{1 \times k}(\sigma_x)$$

An equivalent convolution of I can be obtained by convolving I with both filters sequentially.

The commutative property of convolution means that this can be done in any order.

Note that a Gaussian filter is symmetric, so any convolution with a Gaussian is strictly equivalent to a correlation (no filter-flipping needed).

For this question, consider $k = 15, \sigma_x = 3, \sigma_y = 5$ ($k = 15$ was chosen to cover 3 standard deviations, or ~99.7% of the $\sigma_y = 5$ gaussian).

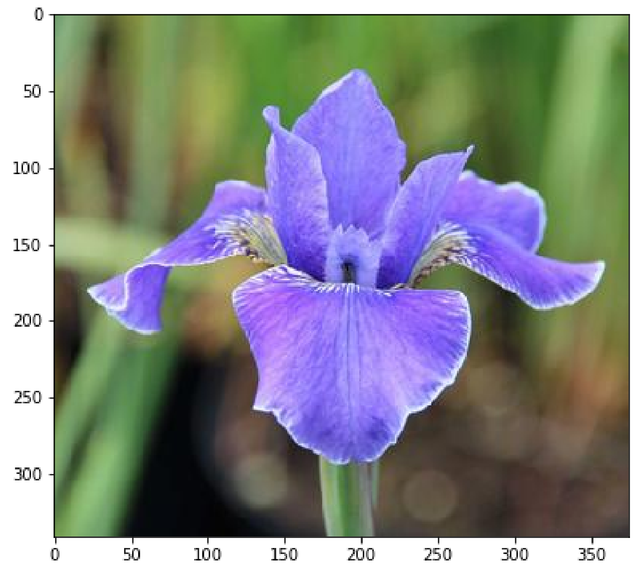
Obtaining $G(\sigma_x = 3)_{1 \times 15}$ and $G(\sigma_y = 5)_{15 \times 1}$ (using scipy's Gaussian function):

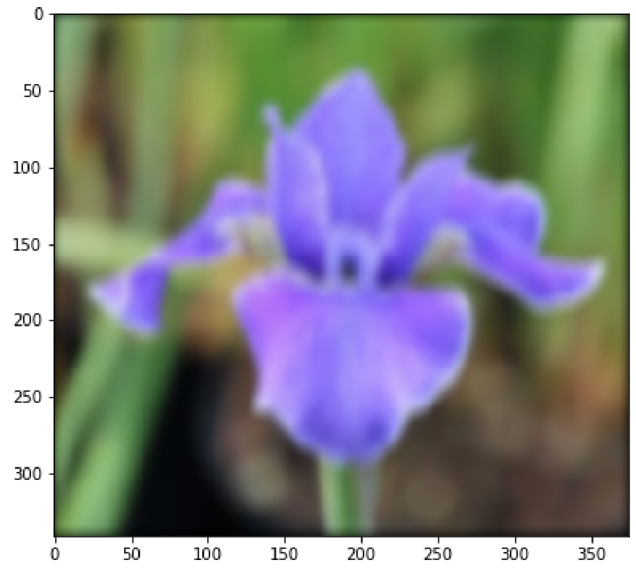
In [3]:
```
gauss_3x = norm_filter(signal.gaussian(15, std=3).reshape(1, 15))
gauss_5y = norm_filter(signal.gaussian(15, std=5).reshape(15, 1))

print(gauss_3x)
print(gauss_5y)
```

```
[[0.00884695 0.01821591 0.0335624  0.05533504 0.08163802 0.10777793
  0.12732458 0.13459835 0.12732458 0.10777793 0.08163802 0.05533504
  0.0335624  0.01821591 0.00884695]]
[[0.03453786]
 [0.04479319]
 [0.05581576]
 [0.06682359]
 [0.07686543]
 [0.08494944]
 [0.09020242]
 [0.09202463]
 [0.09020242]
 [0.08494944]
 [0.07686543]
 [0.06682359]
 [0.05581576]
 [0.04479319]
 [0.03453786]]
```

Loading and then convolving *iris.jpg* with $G(\sigma_x = 3)_{1 \times 15}$ then $G(\sigma_y = 5)_{15 \times 1}$ to obtain *q2-output.jpg*:

```
In [4]: iris = cv.imread("iris.jpg")

        conv_iris = correlation(
            correlation(iris, gauss_3x, same),
            gauss_5y, same
        )

        display_image(iris)
        display_image(conv_iris, file_name = "q2-output.jpg")
```





Question 3: Commutativity of Convolution vs. Correlation

Convolution is commutative

Definition of 2D convolution:

$$(F * I)(i, j) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} F(u, v) I(i - u, j - v)$$

Let:

$$U = i - u \implies u = i - U$$

$$V = j - v \implies v = j - V$$

Substituting:

$$(F * I)(i, j) = \sum_{U=\infty}^{-\infty} \sum_{V=\infty}^{-\infty} F(i - U, j - V) I(U, V)$$

Reversing the multiplication:

$$(F * I)(i, j) = \sum_{U=\infty}^{-\infty} \sum_{V=\infty}^{-\infty} I(U, V) F(i - U, j - V)$$

Switching summation start/end indices (sum is still over the same set of elements):

$$(F * I)(i, j) = \sum_{U=-\infty}^{\infty} \sum_{V=-\infty}^{\infty} I(U, V) F(i - U, j - V)$$

By the definition of convolution:

$$\sum_{U=-\infty}^{\infty} \sum_{V=-\infty}^{\infty} I(U, V) F(i - U, j - V) = (I * F)(i, j)$$

Thus:

$$(F * I)(i, j) = (I * F)(i, j)$$

Correlation is not commutative

The following code displays the correlation of two simple 3×3 matrices, in different orders.

The final results are clearly not equal, which means that correlation is not commutative.

It is interesting to note that the final matrices are x, y reflected versions of each other. This is because correlation is equivalent to a convolution using an x, y reflected filter.

In [5]:
```
a = np.array([
    [7, 5, 9],
    [3, 1, 6],
    [2, 4, 8]
])

b = np.array([
    [9, 8, 7],
    [5, 1, 2],
    [4, 6, 3]
])

print(correlation(a, b, same))
print(correlation(b, a, same))
```

```
[[ 38  94  74]
 [120 250 192]
 [ 41 107  85]]
[[ 85 107  41]
 [192 250 120]
 [ 74  94  38]]
```

Question 4: Separability of Gaussian Filter's Horizontal Derivative

The horizontal derivative G_x of the 2D Gaussian filter G is separable. Proof:

Definition of 2D Gaussian:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \cdot exp\left(-\frac{x^2 + y^2}{\sigma^2}\right)$$

$$= \frac{1}{2\pi\sigma^2} \cdot exp\left(\frac{-x^2}{\sigma^2}\right) \cdot exp\left(\frac{-y^2}{\sigma^2}\right)$$

Taking the horizontal (x) directional derivative, we can ignore every term that doesn't depend on x:

$$G_x(x, y) = \frac{d}{dx}\left(\frac{1}{2\pi\sigma^2} \cdot exp\left(\frac{-x^2}{\sigma^2}\right) \cdot exp\left(\frac{-y^2}{\sigma^2}\right)\right)$$

$$= \frac{1}{2\pi\sigma^2} \cdot exp\left(\frac{-y^2}{\sigma^2}\right) \cdot \frac{d}{dx}\left(exp\left(\frac{-x^2}{\sigma^2}\right)\right)$$

$$= \frac{1}{2\pi\sigma^2} \cdot exp\left(\frac{-y^2}{\sigma^2}\right) \cdot \frac{-2x}{\sigma^2} \cdot exp\left(\frac{-x^2}{\sigma^2}\right)$$

$$= \frac{1}{\pi\sigma^4} \cdot exp\left(\frac{-y^2}{\sigma^2}\right) \cdot \ - x \cdot exp\left(\frac{-x^2}{\sigma^2}\right)$$

Note that G_x can be expressed as the product of two 1D functions:

$$G_x(x, y) = A(x) \cdot B(y)$$

$$A(x) = \ - x \cdot exp\left(\frac{-x^2}{\sigma^2}\right)$$

$$B(y) = \frac{1}{\pi\sigma^4} \cdot exp\left(\frac{-y^2}{\sigma^2}\right)$$

Therefore, the horizontal derivative G_x of the Gaussian filter G is separable.

Question 5: Computational Cost of Convolution / Correlation

Regardless of whether h is separable, the naive algorithm is to "drag" h across every pixel of I, and dot it with the corresponding patch of I. For a *non-separable* h, this algorithm is the best we can achieve.

Naive / Non-Separable

We note:

1) For valid, same and full correlation, there are $(n - \lfloor m/2 \rfloor)^2$, $(n-1)^2$, or n^2 patches of size $m \times m$ in I, respectively. Regardless, I has $O(n^2)$ patches.

2) Each time a $m \times m$ patch is dotted with h, it needs m^2 multiplications and $m^2 - 1$ additions, or $O(m^2)$ operations total.

This algorithm executes $O(m^2)$ operations $O(n^2)$ many times, thus incurring a total of $O(n^2) \times O(m^2) \in O(n^2 m^2)$ operations.

Optimized / Separable

If h is *separable*, then: $h_{m \times m} = g_{m \times 1} * j_{1 \times m}$, and by the associative rule of convolution: $I * h = I * (g * j) = (I * g) * j$.

We now note:

1) There are still $O(n^2)$ patches of size $1 \times m$ and $m \times 1$ in I.

2) Each patch only has m elements, which means only $O(m)$ operations are necessary to dot the patch.

To obtain $(I * g)$, the algorithm needs to first execute $O(m) * O(n^2) \in O(mn^2)$ operations.

Notice that $(I * g)$ still has $O(n^2)$ patches, so the same logic above applies: to obtain $(I * g) * j$, the algorithm must do $O(mn^2)$ operations again.

This optimization makes the algorithm run with $O(mn^2) + O(mn^2) = O(2mn^2) \in O(mn^2)$ operations. This is substantially cheaper than $O(m^2 n^2)$.

Question 6: Separable Sum of Two Separable Filters

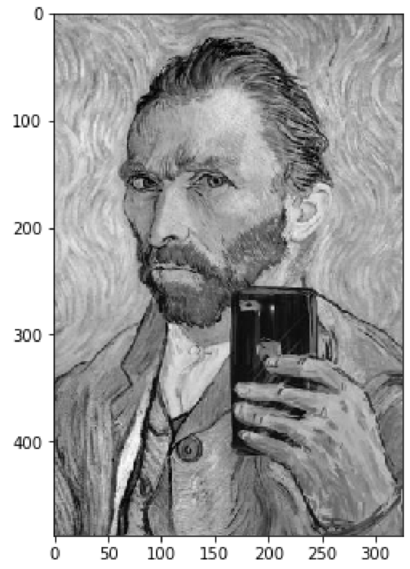
$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

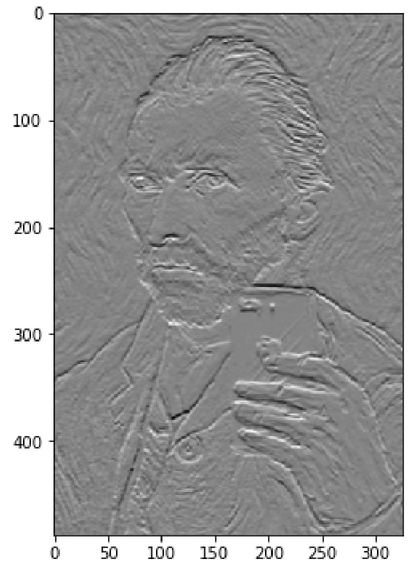
$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

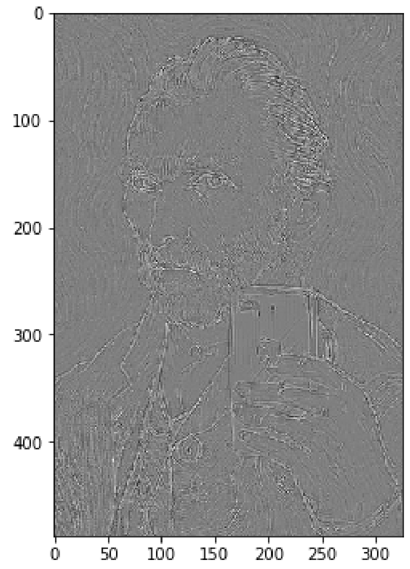
$$A + B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Question 7: Application of Gaussian Derivative + Laplacian to *portrait.jpg*

```
In [6]: portrait = cv.imread("portrait.jpg", cv.IMREAD_GRAYSCALE)

        # Apply derivative of gaussian (x-derivative) filter of sig_x, sig_y = 1,
        deriv_portrait = gaussian_filter(portrait.astype(float), sigma=[1, 1], order=[1, 0])

        # Apply laplacian
        laplace_portrait = laplace(portrait.astype(float))

        display_image(portrait)
        display_image(deriv_portrait, file_name = "q7-output-gauss-deriv.jpg")
        display_image(laplace_portrait, file_name = "q7-output-laplace.jpg")
```

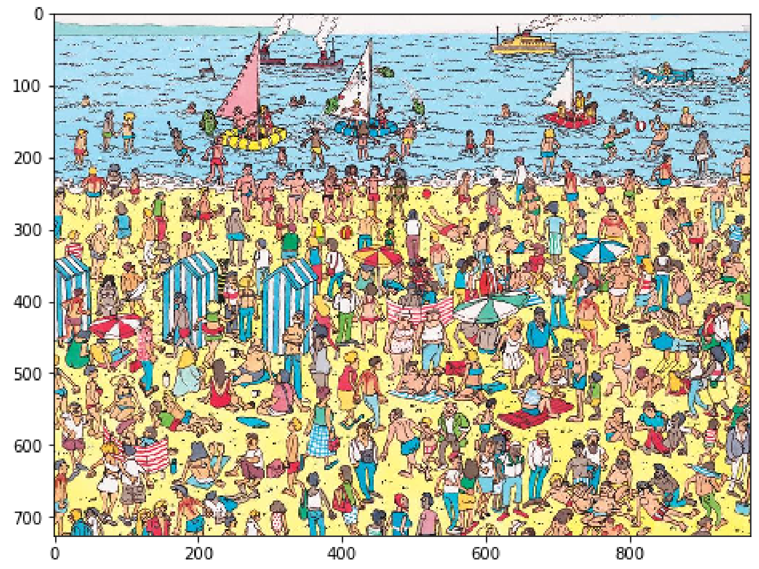


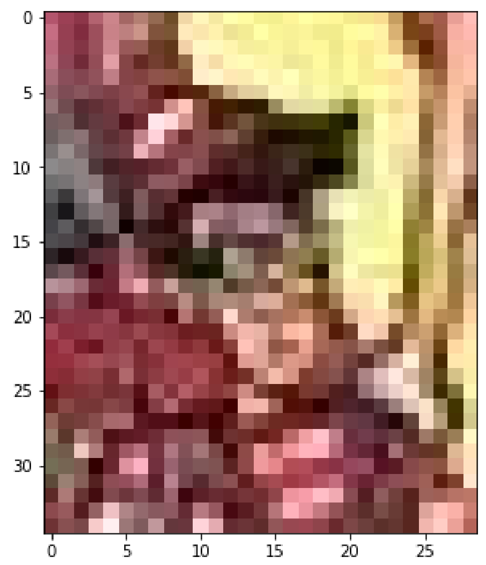


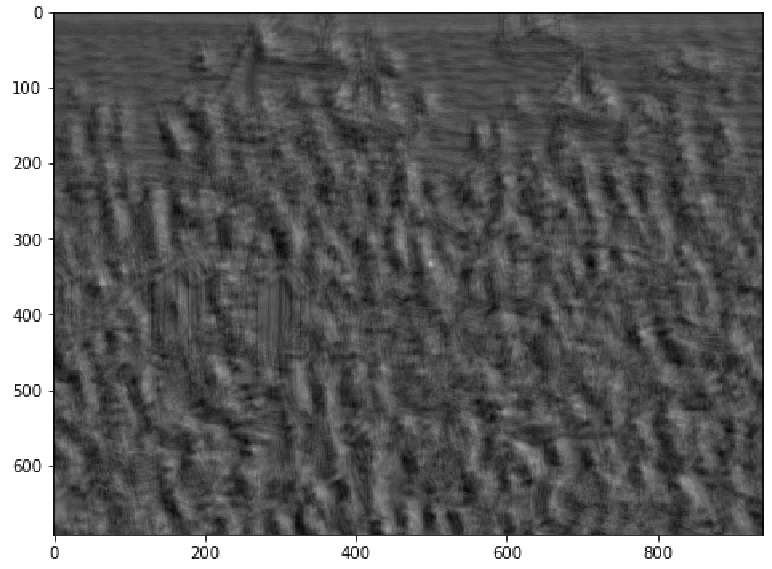


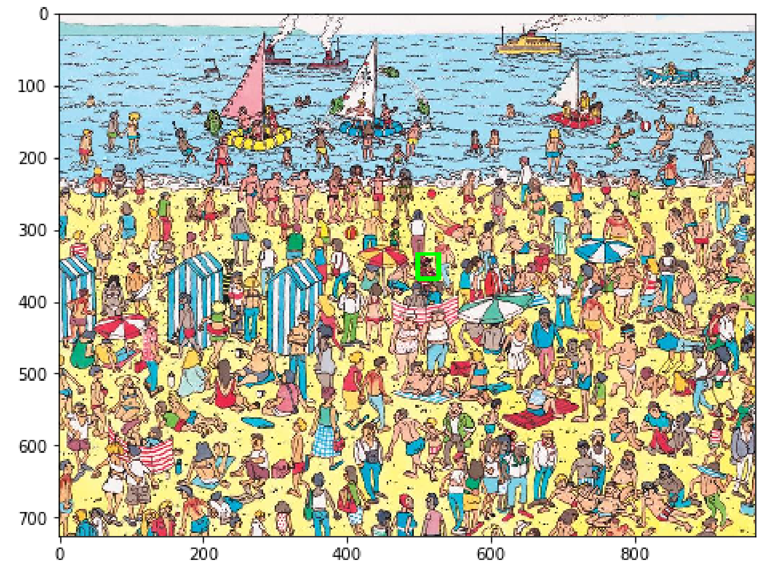
Question 8: Waldo Detection via Correlation

In [7]:
```
# Load and display images
waldo_beach = cv.imread("whereswaldo.jpg")
waldo = cv.imread("waldo.jpg")
display_image(waldo_beach)
display_image(waldo)

# Using OpenCV's template matching (correlation) method to find the most relevant points
beach_corr = cv.matchTemplate(waldo_beach, waldo, cv.TM_CCOEFF)
min_val, max_val, min_loc, max_loc = cv.minMaxLoc(beach_corr)
display_image(beach_corr, "q8-correlation-output.jpg")

# Figuring out the coordinates of the rectangle
waldo_height, waldo_width = waldo.shape[:2]
top_left = max_loc
bottom_right = (top_left[0] + waldo_width, top_left[1] + waldo_height)

# Draw a green rectangle on the beach image to indicate the location of waldo
waldo_found = np.copy(waldo_beach)
cv.rectangle(waldo_found, top_left, bottom_right, (0, 255, 0), 4)
display_image(waldo_found, "q8-rectangle-output.jpg")
```


0
100
200
300
400
500
600
700
0
200
400
600
800


0
5
10
15
20
25
30
0
5
10
15
20
25


0
100
200
300
400
500
600
0
200
400
600
800


0
100
200
300
400
500
600
700
0
200
400
600
800

Question 9 : Explanation of Canny Edge Detection

1) Remove / Reduce Image Noise

This is done to reduce the chances of false edges being detected. Edges are defined by extrema in the derivative(s) of the image. Noise may falsely inflate or deflate these derivatives, so noise reduction is imperative. This is accomplished by convolving the image with a Gaussian filter, thereby blurring it and removing the noise. The blurring is done in both the x and y directions to ensure that the respective directional derivatives are not influenced by noise.

2) Determine Image Gradient

In order to detect edges, the algorithm must take the directional derivatives I_x, I_y of the image. Edges are areas of rapid intensity change, and thus they appear as extrema in these derivatives. These directional derivatives can be used to compute the gradient and orientation of each edge pixel in the image:

Gradient: $\nabla I = \sqrt{I_x^2 + I_y^2}$

Orientation: $\theta = arctan\left(\frac{I_y}{I_x}\right)$

3) Reduce Width of Edges

At this point, the detected edges are probably fairly thick (i.e. many pixels wide). In order to produce clearly defined (one-pixel thick) edges, the algorithm must reduce the width of the detected edges. To accomplish this, *non-maxima suppression* is used - essentially, only the most intense edge pixels in a neighbourhood are kept. The gradients and orientations calculated in the previous step are used to accomplish this. If a pixel is a local maximum along its gradient's orientation, it is kept.

4) Categorize / Discard Edges Using Thresholds

At this point, the remaining (non-zero) pixels are probably edges. However, some of them may be the result of noise or minor color variations. A thresholding system can be used to discard and categorize these pixels as follows:

Define two threshold values L (low), H (high), and let G be the gradient of some pixel:

$G < L \implies$ discard this pixel (*intuition: if the gradient is very small, the pixel is probably the result of noise, actual edges should be far more defined*)

$L \leq G < H \implies$ categorize this pixel as a "weak-edge" (*intuition: can't say for sure whether this pixel is actually the result of an edge*)

$H \leq G \implies$ categorize this pixel as "strong-edge" (*intuition: if the gradient is very large, the pixel probably represents an actual edge*)

5) Connect Edges Together (Hysteresis)

At this point, most of the non-edge pixels have been discarded. However, the detected edges may not be continuous. In order to increase the continuity of the edges, the algorithm extends edges between the remaining pixels.

The strong-edge pixels identified in the previous step are used as starting points for the edges, as these have the highest chance of belonging to an actual edge. Weak-edge pixels are only kept if they are next to a strong-edge pixel (vertically, horizontally or diagonally). This is because neighbouring weak and strong edges are likely to be part of the same edge.

Weak-edge pixels that do not have a strong-edge neighbour are eliminated. It is unlikely that these isolated weak-edge pixels belong to an actual edge.

Question 10: Explanation of Laplacian Zero-Crossings

Edges are areas of *rapid* change in the image intensity function. Therefore, they should correspond to the extrema of the image's first derivatives.

The 2D Laplacian operator ∇^2 is defined as follows (for some 2D function f, with second-order x, y derivatives of f_{x^2}, f_{y^2}):

$$\nabla^2 f = f_{x^2} + f_{y^2}$$

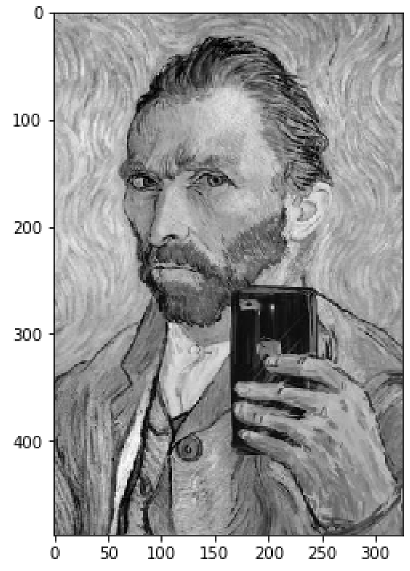
Note that some zero crossings of the Laplacian are resultant from both second-order derivatives being zero:

$$f_{x^2} = 0 \wedge f_{y^2} = 0 \implies \nabla^2 f = 0$$

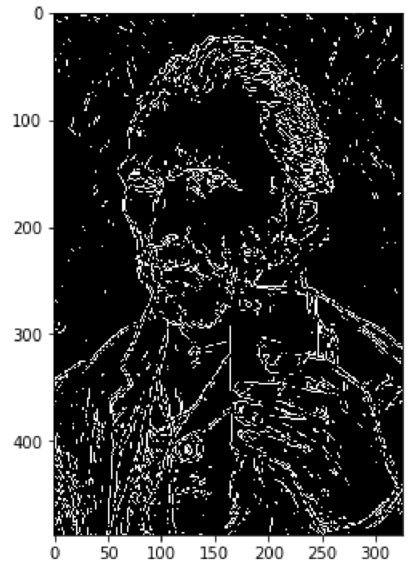
When either of these second-order derivatives are equal to zero, the respective integral (the first-derivative) must be at an extrema. If the first-derivative is at an extrema, the image intensity must be changing very rapidly. Therefore, this point (a zero-crossing of the Laplacian) is an edge.

Question 11: Canny Edge Detection on *portrait.jpg* (OpenCV)

In [8]:
```
display_image(portrait)

# Best thresholds so far: lower=200, upper=230, sobel size = 15
canny_portrait = cv.Canny(portrait, 200, 230, 15)
display_image(canny_portrait, file_name = "q11-output.jpg") # PDF doesn't do this justice, please see attached fil
e
```



0
100
200
300
400
0
50
100
150
200
250
300



0
100
200
300
400
0
50
100
150
200
250
300