Assignment 2

Daniel Silva

1) To implement a Sobel edge detector, the first step was to write a convolution function. In the function I wrote, I first pad the input so that the gradient map returned by the function remains the same size as the original image.

```
# Pad image so the output remains the same dimensions as the input
img = np.pad(img, pad_width = (m // 2, n // 2), mode = 'constant', constant_values=0.0)
```

We then use the equation for the valid coordinates of a convolution to create an empty matrix for the output.

```
# Valid convolution coords
valid_y = img.shape[0] - m + 1
valid_x = img.shape[1] - n + 1

# Create empty image matrix
new_image = np.empty((valid_y, valid_x))
```

Finally, we loop through the input performing element wise multiplications of 3x3 areas of the image against the 3x3 kernel, adding each of the 9 resulting values together.

```
# Convolve kernel over image
for i in range(valid_y):
    for j in range(valid_x):

        # Compute dot product of of 3x3 area in image against kernel and add the 9 values to get convolution output
        new_image[i, j] = np.sum(img[i:i+m, j:j+n] * kernel)

return new_image
```

With this function now defined, calculating the x and y gradients is trivial. Below, I simply load in the grayscale image into one channel and create the x and y Sobel kernels. Passing the image and the Sobel kernels to the convolution function will return the x and y components.

```
# Load image as grayscale
img = pil.open(filename).convert('L')
img = np.array(img)

# Create sobel kernels
kx = np.array([1, 0, -1, 2, 0, -2, 1, 0, -1]).reshape((3,3))
ky = kx.transpose()

# Apply sobel kernels in x and y direction
Ix = conv(img, kx)
Iy = conv(img, ky)
```

Below are the resulting horizontal and vertical edges, respectively:

To compute the final edge response, we calculate the magnitude of each pixel and threshold the magnitude above a certain range. The equation for magnitude is $mag = \sqrt{Ix^2 + Iy^2}$.

```
# Combine x and y components
magnitude = np.sqrt(np.power(Ix, 2) + np.power(Iy, 2))

# Apply threshholding
final_edges = np.where(magnitude >= threshold, magnitude, 0)
```

The below image applies a threshold of 100 to get the final edge response.

2) My Harris Corner Detection function begins by loading a grayscale image into one channel, as well as creating a 3-channel copy. This 3-channel copy is made so that we can create blue markings on corners found in the image.

```
# Load image as grayscale
img = pil.open(filename).convert('L')

# Create 3 channel output image so we can mark corners in blue
new_image = np.array(img.convert('RGB'))

img = np.array(img)
height, width = img.shape[0:2]
```

Before computing the gradients of an image, I first applied a gaussian blur to the input. Note that this was not necessary, although I achieved better performance due to the blur removing some noise from the image. My gaussian blur function first creates a gaussian kernel of specified size, using the standard deviation of the image as the sigma value. Using the standard gaussian function, the kernel is instantiated and then convolved across the input image.

```
def gaussian_blur(img, kernel_size):
    'Blurs an image using a gausian kernel of given size'

    # Get standard deviation of image
    sigma = np.std(img)

    # Create gaussian kernel of specified shape
    kernel = np.empty(shape = (kernel_size, kernel_size))

    # Creating kernel with gaussian function
    for i in range(kernel_size):
        for j in range(kernel_size):

            # Gaussian function
            a = 2 * np.pi * (sigma ** 2)
            b = np.exp(-(j ** 2 + i ** 2) / (2 * sigma ** 2))
            kernel[i, j] = (1 / a) * b

    # Convolve kernel to blur image
    output = conv(img, kernel)

    return output
```

```
# Blur image with 3x3 gaussian kernel
img = gaussian_blur(img, 3)
```

I then create Sobel kernels and convolve them across the blurred image to get the x and y gradients.

```
# Create sobel kernels
kx = np.array([1, 0, -1, 2, 0, -2, 1, 0, -1]).reshape((3,3))
ky = kx.transpose()

# Apply sobel kernels in x and y direction
Ix = conv2(img, kx)
Iy = conv2(img, ky)
```

The following step was to calculate the pixel wise products of the gradient images to get I_{xx}, I_{yy}, and I_{xy}.

```
# Computer pixel wise products of image gradients
Ixx = Ix * Ix
Iyy = Iy * Iy
Ixy = Ix * Iy
```

Convolving these with a window function will complete the sum of squares calculation on the gradient images. The window function is applied using a variable sized kernel of shape (3x3), (5,5), or (7,7).

```
# Calculate sum of squares using window function
w = np.ones(shape=(window_size, window_size))
Sxx = conv2(Ixx, w)
Syy = conv2(Iyy, w)
Sxy = conv2(Ixy, w)
```

I then create a matrix to hold the Harris responses for each pixel and calculate an offset value. This offset value is used to ensure I calculate each Harris response for the center of each window.

```
# Array to hold harris responses
responses = np.zeros(shape=(height,width))

# Ensures calculated corner points are centered in window
offset = window_size // 2
```

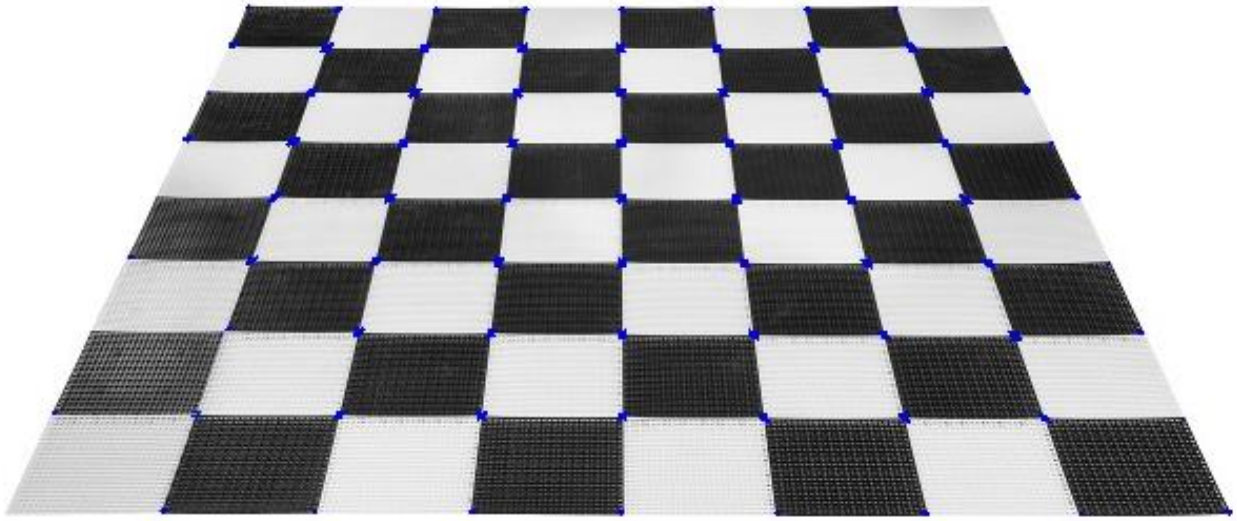
The next step was to loop through each pixel, computing the M matrix, trace of M, determinant of M, and finally the corner response value: $r = det - k * trace^2$.

```
# Compute the M matrix and response value for each pixel
for i in range(offset, height - offset):
    for j in range(offset, width - offset):

        # Create M matrix
        M = np.array([Sxx[i, j], Sxy[i, j], Sxy[i,j], Syy[i, j]]).reshape((2,2))

        # Calculate trace
        trace = M[0,0] + M[1,1]

        # Calculate determinant = ad - bc
        det = M[0,0] * M[1,1] - M[0, 1] * M[1, 0]

        # Calculate corner response values
        r = det - k * trace ** 2

        responses[i, j] = r
```

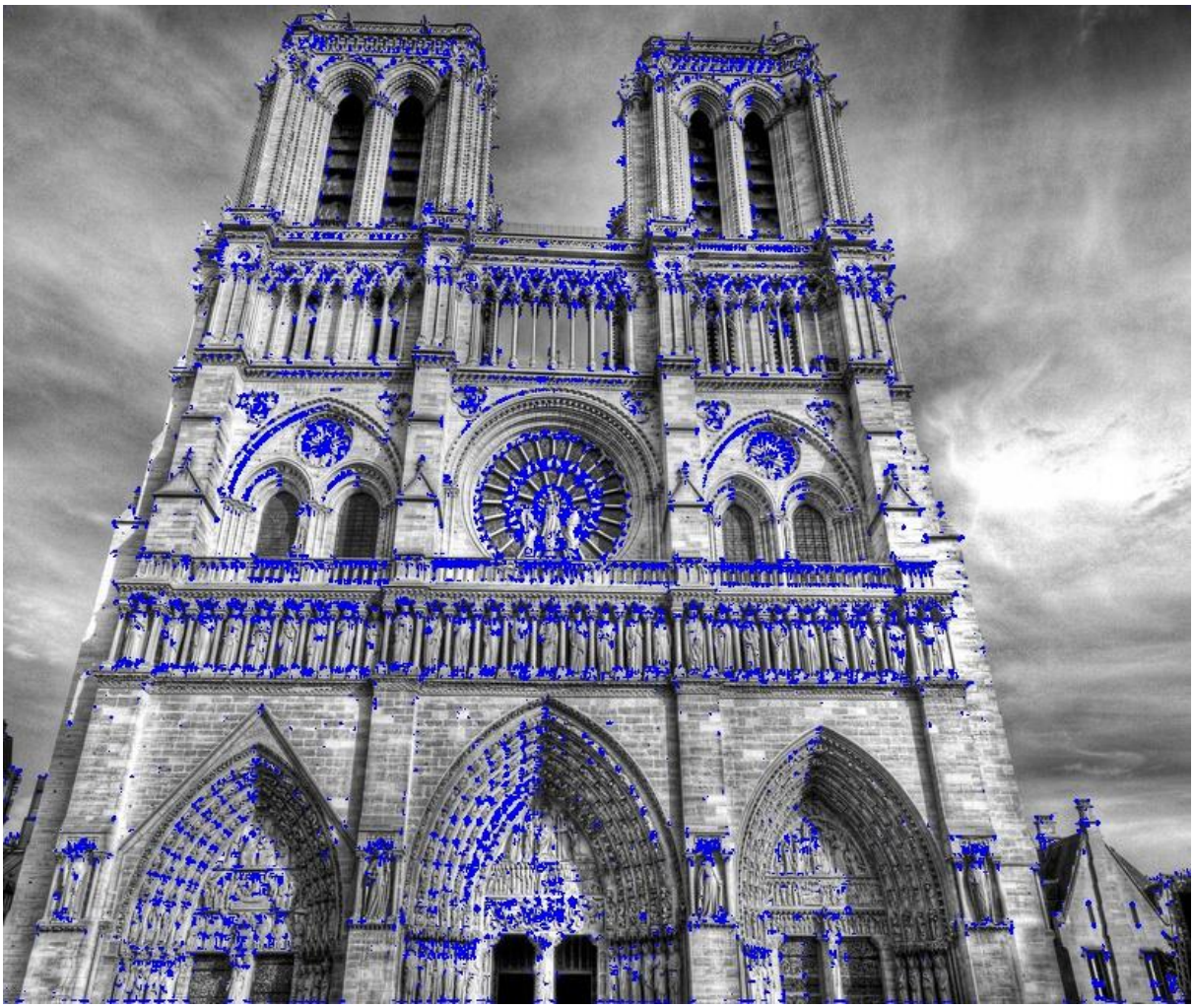
The calculated R values are finally thresholded by a given percentage of the largest response in the response matrix.

```
# Threshold R values
new_image[responses > threshold * responses.max()] = [0,0,255]
```

The below image was created using the following parameters: k = 0.4, a window size of (3x3), and a threshold of 1% of the largest response value.



The below image was created using the following parameters: k = 0.6, a window size of (3x3), and a threshold of 0.5% of the largest response value.



Regarding the effects of these parameters, a smaller k value increases the overall response values and seems to result in more corners detected, although this comes with a higher number of false positives. Increasing the value of k to 0.6 decreases the number of overall corners detected, but results in a much higher precision value. Using a larger window size increases the number of large response values at corners, due to us simply looking at a larger region around the corner as the window is slide across the image. This simply means the window will contain corners more frequently than with a smaller one, hence the greater number of large responses. Lastly, the threshold simply defines how strong the response values must be for the algorithm to be confident enough to label them as corners. In practice, I found that different images resulted in various ranges of r response values. Therefore, I found that using a percentage of the largest response value as a threshold resulted in improved and more steady performance.

All code for this assignment is present in code/Assignment2.ipynb. Some further testing and visualization of these different parameters is also present.