CSC420 Assignment 1

Yatu Zhang 999585836

September 27, 2016

Problem 1

(a) The following is my implementation of the 2D convolution. As seen, the kernel is first modified so that it is always odd sized. This is so that we will always have a center point in our kernel. Then the image is zero-padded so that the output of the convolution is the same size as the input image. This implementation is done using looping. The two outer loops go through every valid image pixel and at every iteration, two inner loops perform the convolution at that exact pixel position by summing up the product of corresponding pixels in the kernel and image surrounding it.

```python
def oddifyFilter(kernel):
        #if row divisible by 2, add a row
        (row, col) = kernel.shape
        if(row%2==0):
                newRow = np.zeros((1,col))
                kernel = np.concatenate((kernel,newRow), axis=0)
                row = row + 1
        #if col divisible by 2
        if(col%2==0):
                newCol = np.zeros((row,1))
                kernel = np.concatenate((kernel,newCol), axis=1)

        return kernel

def padding(image, kernel):

        #rows & columns to padd before and after
        pad_vsize = kernel.shape[0]/2;
        pad_hsize = kernel.shape[1]/2;

        #npad is a tuple of the number of rows to pad (above and below);
        npad = ((pad_vsize, pad_vsize),(pad_hsize, pad_hsize))

        image = np.pad(image, pad_width=npad, mode='constant', constant_values=0)

        return image

def conv2d(image, kernel):
        #returns a matrix that is the same size as the input image.  Need to do
↪  padding
        #First save the size of the image before padding (this is for use in the
↪  convolution step)
        row_init, col_init = image.shape

        result = np.zeros((row_init, col_init))
        #make filter odd sized and padd the image
        kernel = oddifyFilter(kernel)
        image = padding(image, kernel)

        (row_kernel, col_kernel) = kernel.shape
        (row_im, col_im) = image.shape


        for i in range(row_init):
                for j in range(col_init):
                        for u in range(0,row_kernel):
                                for v in range(0,col_kernel):
                                        #NOTE: here the kernel index should start
↪ at 0.
                                        result[i,j] +=
↪ kernel[row_kernel-u,row_kernel-v] * image[i+row_kernel/2+u,j+col_kernel/2+v]

        return result
```

(b) It is certainly possible to implement the the convolution of a kernel and a patch of the image using a dot(inner) product of two 1D vectors. This is essentially simplifying the summation and multiplication operations in the two inner loops because these two operations between corresponding pixel and kernel pairs is essentially the same as taking the inner product of two vectors. First the kernel is flattened and flipped, then every patch formed by the moving window across the image has also been flattened to a 1D vector like the following and a dot product is taken between the two. The following function is the implementation.

```python
def conv2d_vec(image, kernel):
        #returns a matrix that is the same size as the input image.  Need to do
↪ padding
        #First save the size of the image before padding (this is for use in the
↪ convolution step)
        row_init, col_init = image.shape

        result = np.zeros((row_init, col_init))
        #make filter odd sized and padd the image
        kernel = oddifyFilter(kernel)
        image = padding(image, kernel)

        (row_kernel, col_kernel) = kernel.shape
        (row_im, col_im) = image.shape

        #Flatten the Kernel
        kernel = kernel.flatten()
        k_r = row_kernel/2
        k_c = col_kernel/2

        for i in range(row_init):
                for j in range(col_init):
                        #NOTE: here the kernel index should start at 0.
                        temp = image[i:row_kernel+i, :][:, j:col_kernel+j]
                print (i,j)
                result[i][j] = np.dot(temp.flatten(), np.fliplr(kernel))

        return result
```

(c) It is also possible to do the convolution using matrix multiplication. First, we have to rearrange the patches formed by a sliding window of the size of the filter into a column of 1D vectors, in which every vector in it is a flattened patch in the previous case. We then flatten the kernel and flip it. Lastly we take the matrix multiplication of the kernel and the matrix containing a column of 1D vectors and reshape it to the size of the original image. The following is the implementation.

```
def conv2d_matmul(image, kernel):
        #returns a matrix that is the same size as the input image.  Need to do
↪  padding
        #First save the size of the image before padding (this is for use in the
↪  convolution step)
        row_init, col_init = image.shape

        #make filter odd sized and padd the image
        kernel = oddifyFilter(kernel)
        image = padding(image, kernel)

        (row_kernel, col_kernel) = kernel.shape
        (row_im, col_im) = image.shape

        #Flatten the Kernel
        kernel = kernel.flatten()
        k_r = row_kernel/2
        k_c = col_kernel/2

        column_vector = np.empty((1,row_kernel*col_kernel))

        #This part is the implementation of rearranging the image into a column of
↪  vectors, needs to be optimized!!!!!
        for i in range(row_init):
                for j in range(col_init):
                        #NOTE: here the kernel index should start at 0.
                        temp = image[i:row_kernel+i, :][:,
↪  j:col_kernel+j].reshape(1,-1)
                        print column_vector.shape
                        column_vector = np.concatenate((column_vector, temp),
↪  axis=0)

        result = np.matmul(kernel, column_vector).reshape(row_init,col_init);

        return result
```

Problem 2

(a) For a $n \times n$ image convolving with a $m \times m$ filter, the cost of computing a convolution is m^2 operations per pixel. This is because multiplication between a image pixel and a value in the corresponding position on the kernel needs to be computed. Therefore, a total of m^2n^2 operations are needed for the entire image because this convolution operation needs to be performed on every image pixel.

For a separable filter, the cost is $2m$ operations per pixel and a total of $2mn^2$ for the image. This is because the size of each separable filter is only m and we compute the convolution using X and Y filters separately, therefore when we compute the convolution it only requires m multiplication operations per pixel for each filter. In total this sums to $2mn^2$ because convolution operation is performed for every pixel.

(b) Applying a Gaussian of $\sigma = 3$ and then applying another Gaussian with $\sigma = 8$ is equivalent to applying a Gaussion with $\sigma = 24$ This is best explained by the associative rule for linear filters (Gaussian is a linear filter). $G1 * (G2 * I) = (G1 * G2) * I$.

(c) The following is the implementation of anisotropic Gaussian given σ_x and σ_y. Note that the size of the Gaussian filter is taken as 3σ each way from the centre in each dimension (3σ+3σ+1). For example for $\sigma_x = 15$ and $\sigma_y = 3$, the size of the filter will then be 91 by 19. The rule of 3σ is used because it is a rule of thumb that for a Gaussian almost 99.7% of the information are contained inside 3 standard deviations each way from the centre.

```python
def gaussian_2D(sigmax, sigmay):

        #HERE NEED TO MAKE USE OF THE 3 SIGMA RULE
         (M, N) = (6*sigmay, 6*sigmax)
         if(M%2==0):
                 M=M+1
         if(N%2==0):
                 N=N+1

         print(M,N)
         #initialize the gaussian filter to 0
         gaussian_filter = np.zeros((M,N))
         for i in range(M):
                 for j in range(N):
                         term1 =
  (((i-M/2)**2)+((j-N/2)**2))/((2.0*sigmax**2)+(2.0*sigmay**2))
                         term2 = 2.0*math.pi*sigmax*sigmay

                         gaussian_filter[i,j] = math.exp(-term1)/term2


         return gaussian_filter
```

(d) The following is the cat after applying a Gaussian with $\sigma_x = 15$ and $\sigma_y = 2$. The cat image is converted from RGB to Grayscale for better visualization of the effect since Gaussian does not actually depend on color. As seen, the effect of Gaussian is blurring the image. Since σ_x is much bigger than σ_y, the blur is applied more in the horizontal direction.

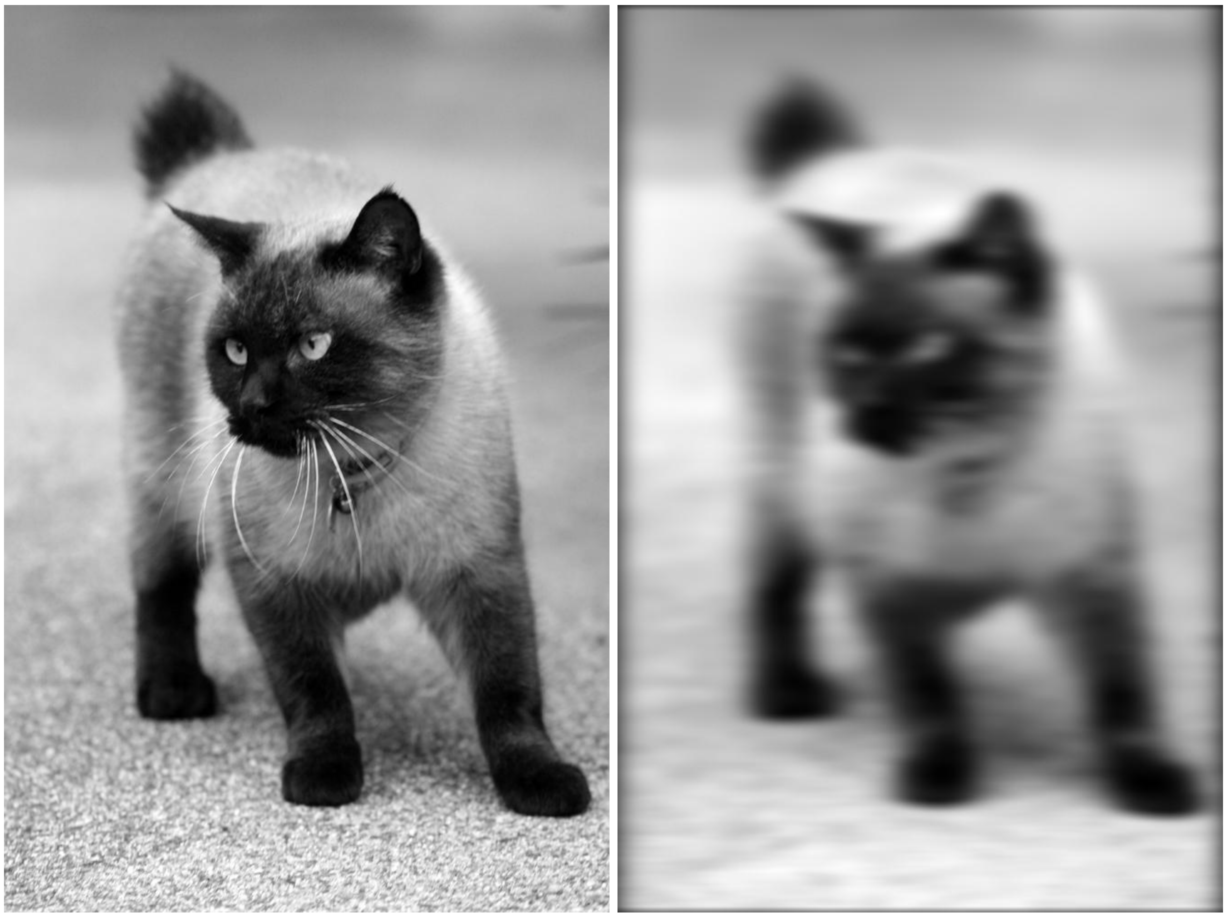
Figure 1: Comparison between images before and after Gaussion

(e) The 2D Gaussian is separable and can be separated into two different filters:

Horizontal:

$$F(x) = \frac{1}{\sigma_x\sqrt{2\pi}} e^{-x^2/2\sigma_x^2} \tag{1}$$

Vertical:

$$F(y) = \frac{1}{\sigma_y\sqrt{2\pi}} e^{-y^2/2\sigma_y^2} \tag{2}$$

By using the applying the seprated 1D Gaussian filters separately, we are able to reduce the computational cost drastically. For example, if our original 2D Gaussian has a size of M x N. Then for every pixel, we will need M x N computations. Depending on the total number of pixels, this could be very expensive. However, if separate the 2D Gaussian into 2 1D Gaussians, one with size M and the other with size N and apply them separately, then we will be able to reduce our number of computations to M + N per pixel. This is a significant improvement because we go from $O(MN)$ to $O(M+N)$ per pixel.

The following is a visualization of the decomposed 1D Gaussians from the previous question, with a horizontal Gaussian with $\sigma_x = 15$ and $\sigma_y = 2$

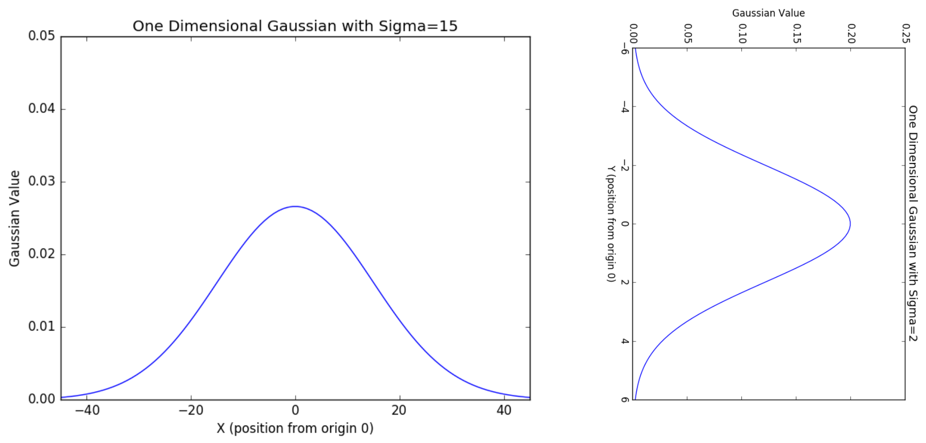

Figure 2: Visualization of Separated Gaussian

Problem 3

(a) The following are the magnitudes of the gradients computed for the template Waldo at three different scales($\sigma = 1$, $\sigma = 2$ and $\sigma = 3$). As we can see, as σ increases, the noise becomes less and less, and the edges detected are becoming thicker and thicker. This is because Gaussian has an effect of blurring and smoothing the image. A large Gaussian will spread the edge out more, which is thicker.

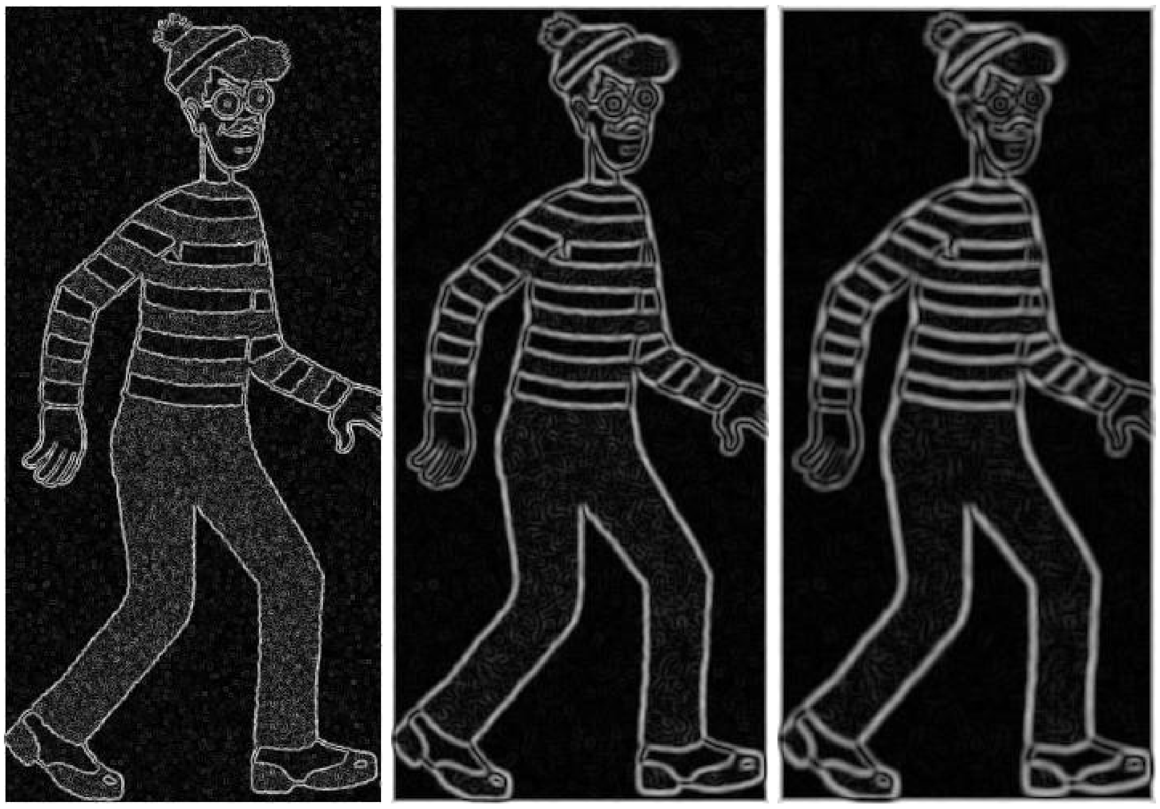

Figure 3: From Left to Rigth: The magnitude of gradients for Waldo at $\sigma = 1$, $\sigma = 2$ and $\sigma = 3$

(b) The following is the implementation of the function that localizes Waldo. The exact location of the peak response is determined to be the coordinate (779, 618). First of all, edge detection is performed to both the image and the template by convolving their gradients respectively. This is because getting the edges instead of every pixel work better in terms of detecting Waldo because it's less subjective to variations in the images. The function then utilizes the normalized cross correlation to the edges of the Waldo template and the actual image. It then finds the indices of where the maximum response occurs by using the argmax function. The region of the maxium response signifies the highest correlation, therefore, Waldo should be found there. A rectangle is then contructed centered at the maximum response to box out Waldo.

```python
def findWaldo(image, kernel):
        k_r, k_c = kernel.shape

        gradient_filterX = gradient_2D(2, axis=0)
        gradient_filterY = gradient_2D(2, axis=1)
        toimage(kernel).show()
        g_Y = signal.convolve2d(kernel,gradient_filterY, boundary='fill',
  mode='same')
        g_X = signal.convolve2d(kernel,gradient_filterX, boundary='fill',
  mode='same')
        mag_kernel = np.sqrt(g_X**2+g_Y**2)
        #toimage(mag_kernel).show()

        g_im_Y = signal.convolve2d(image,gradient_filterY, boundary='fill',
  mode='same')
        g_im_X = signal.convolve2d(image,gradient_filterX, boundary='fill',
  mode='same')
        mag_im = np.sqrt(g_im_X**2+g_im_Y**2)
        #toimage(mag_im).show()

        corr = normxcorr2D(mag_im, mag_kernel)
        (max_x, max_y) = np.unravel_index(np.argmax(corr), corr.shape)

        plt.imshow(corr, cmap='rainbow')
        plt.show()

        box_coor1 = (max_x-k_r/2, max_y-k_c/2)
        box_coor2 = (max_x+k_r/2, max_y+k_c/2)

        fig,ax = plt.subplots(1)
        ax.imshow(image, cmap="Greys_r")
        rect =
  patches.Rectangle((max_y-k_c/2,max_x-k_r/2),k_c,k_r,linewidth=1,edgecolor='r',facecolor='none'
        ax.add_patch(rect)
        plt.show()
```

The gradient is visualized as the following. As seen, the red dot is the peak response in which Waldo is found.

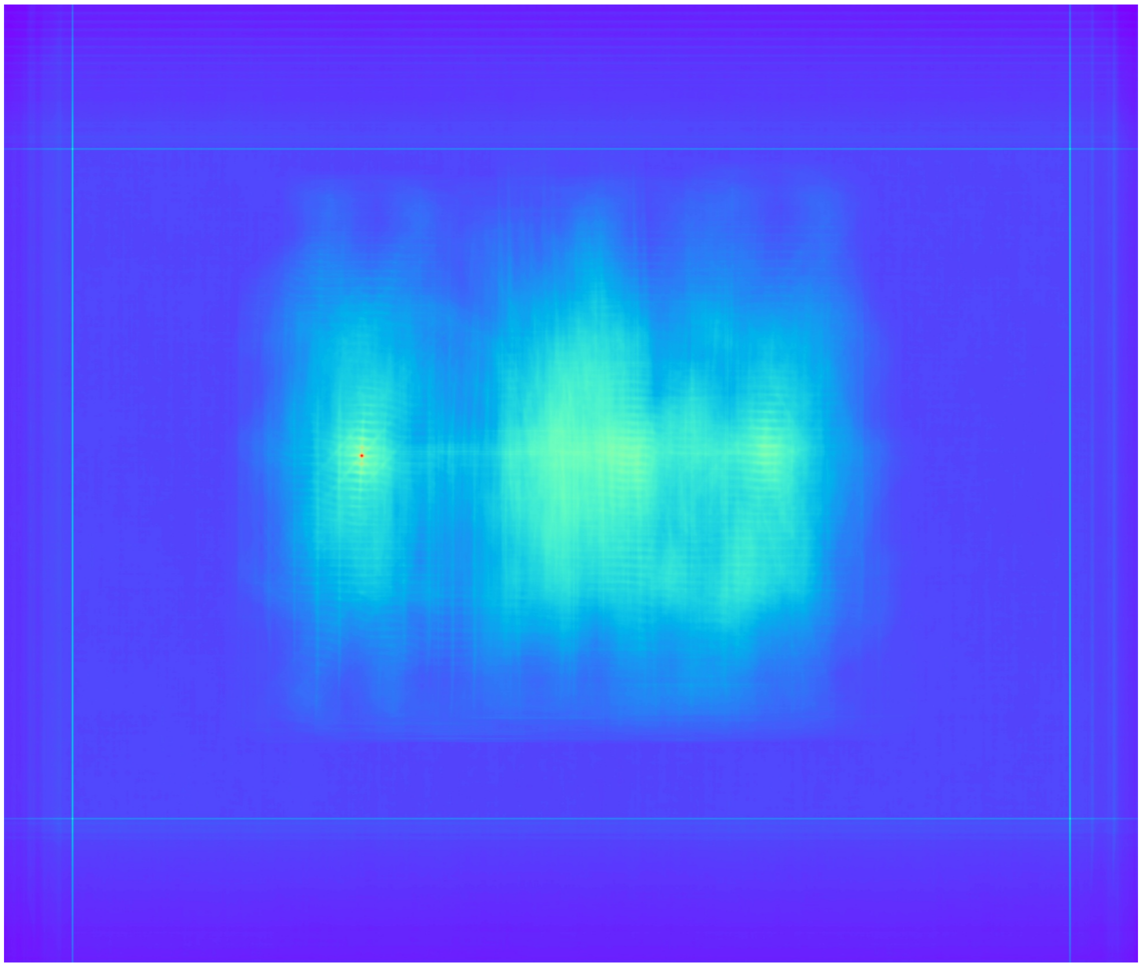

Figure 4: Visualization of Normalized Cross Correlation

A box was drawn centered at this peak response and we find Waldo.

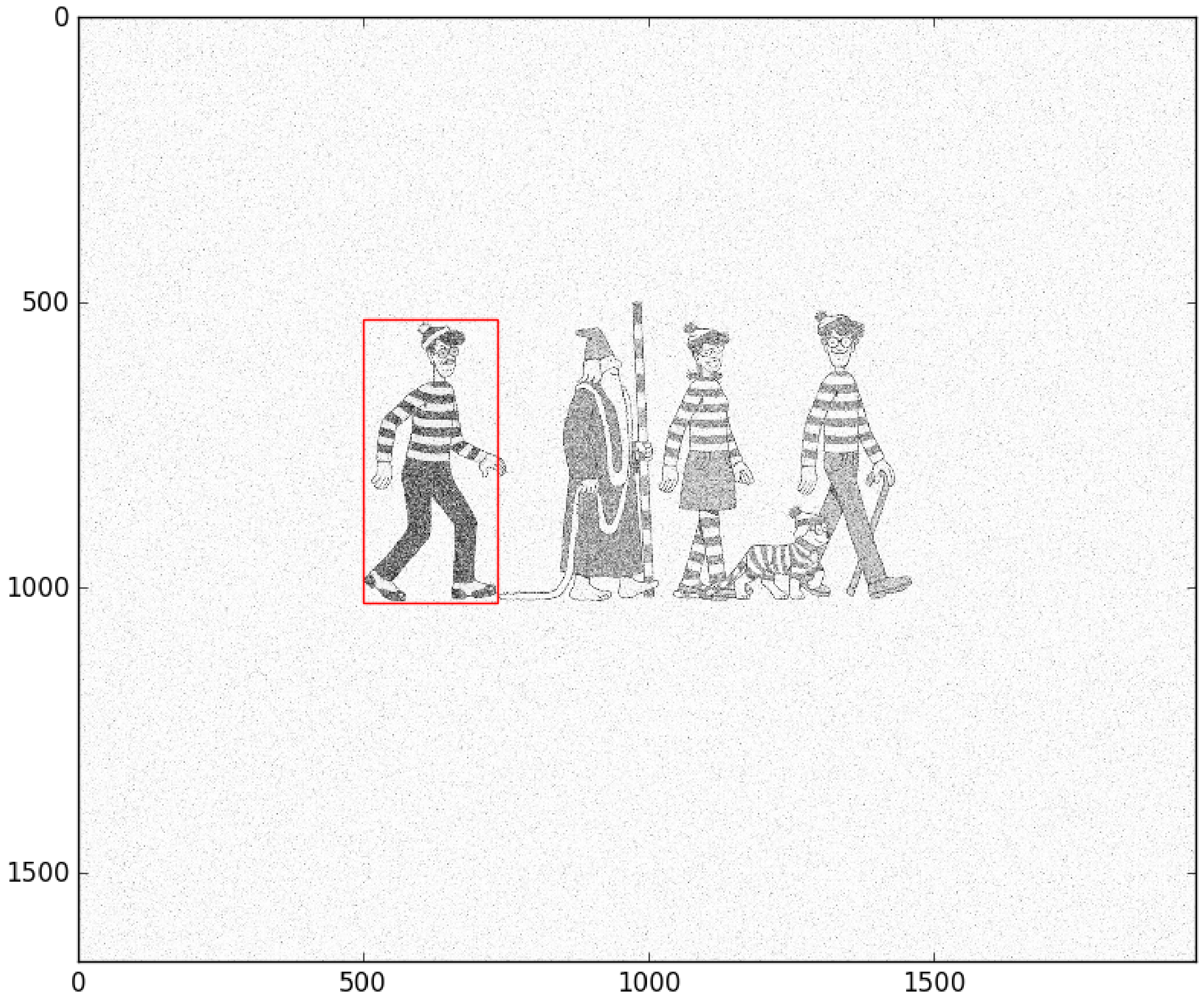

Figure 5: Found Waldo

Problem 4

(a) To start off, $\sigma = 1$, $lowthreshold = 10$ and $highthreshold = 20$ are picked as the parameters of the Canny filter. A low σ ensures that we are capturing thin edges. $lowthreshold = 10$ and $highthreshold = 20$ ensures we are capturing both the high and low contrast edges.

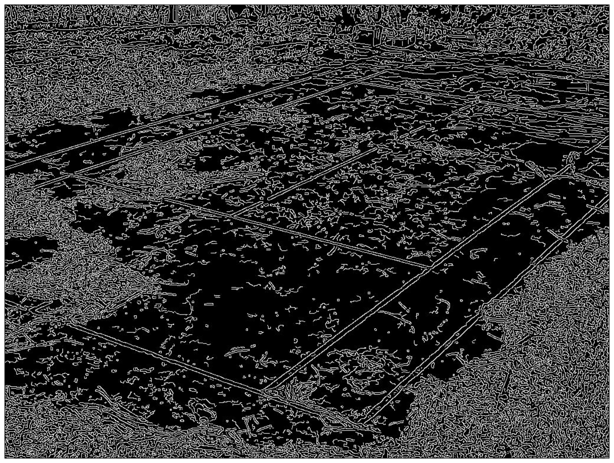

Figure 6: $\sigma = 1$, $lowthreshold = 10$, $highthreshold = 20$

Next, we can increase σ, in this case $\sigma = 3$ is used and everything else is kept the same. The increase of sigma will increase the effect of denoising and blurring, so the finer edges are discarded and only thicker edges(like the border of the tennis court) will be kept.

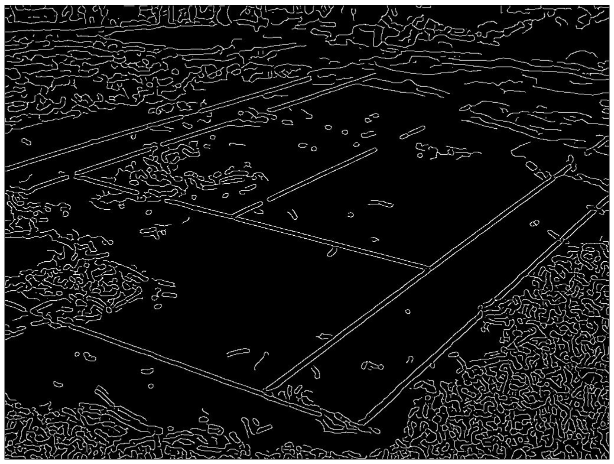

Figure 7: $\sigma = 3$, $lowthreshold = 10$, $highthreshold = 20$

Next, the low threshold is increased to 40 and high threshold is increased to 50 to remove some of the low contrast edges (any pixel with a gradient below 40 and some between 40 and 50). As can be observed from the image, some edges from the leaves and grass are now removed.

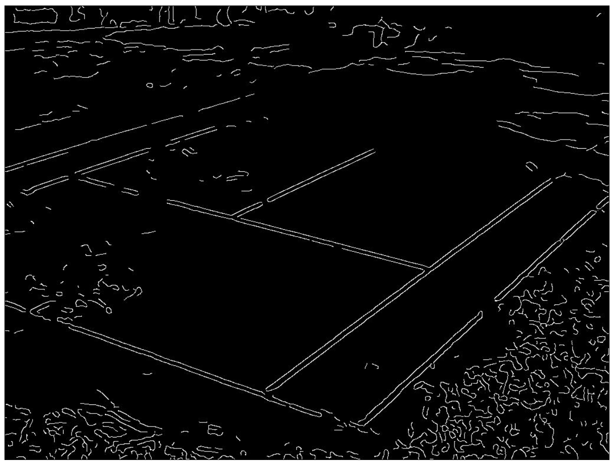

Figure 8: $\sigma = 3$, $lowthreshold = 40$, $highthreshold = 50$

If we continue to increase the threshold, we can see that more and more edges are removed. But at this point, we also see that the some of the lines of the tennis court start to disappear because they are below the 50 threshold, which is not desirable. Other pixels that are above the threshold still remain.

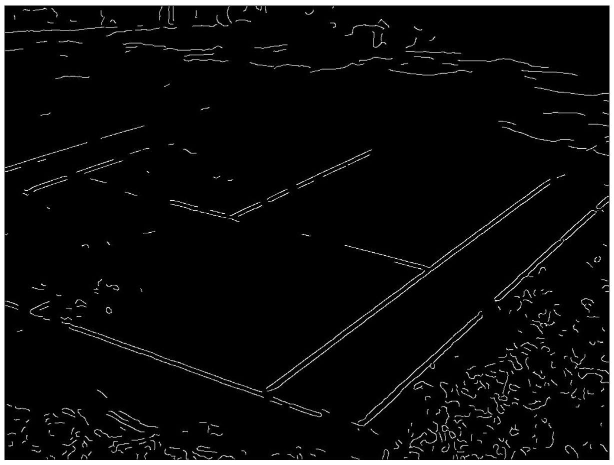

Figure 9: $\sigma = 3$, $lowthreshold = 50$, $highthreshold = 60$

(b) From the above experiment, a strategy can be developed to capture the tennis court sidelines. First of all, since the border of the tennis court is thick and the edges of leaves and grass is thin, we can choose a σ that is large enough so that it will only keep the thick edges. Secondly, we can remove the low contrast edges in which the gradient isn't particularly strong by introducing a relatively large number for the low threshold. We can also increase the value of the high threshold because the gradient of the tennis court border is relatively big. However, this will not remove the edges that have equal or higher gradient values, such as those of the leaves on the bottom right corner of **Figure 6**. In short, **Figure 6** looks like the best we can do.

Problem 5

(a) The following is my code that implements this algorithm.

First of all, the the gradient at every pixel of the image is computed. Then, a graph is created with each pixel's gradient as a node. An edge is then created between every pixel gradient and the 3 pixel gradients immediately below it to form a weighted DAG (Directed Acyclic Graph), in which the weights are the gradient values at each pixel position. This is so that the paths in our graph from top to bottom will only contain one pixel at each row. Then, the Dijkstra's Algorithm is performed between every point from the top row to every point from the bottom and the shortest path that has the lowest sum of gradient is found. Then the pixels along the shortest path are removed and the image is reshaped to correspond to 1 less column. This procedure removes one seam from the image. To remove N seams, we will just have to repeat this procedure N times.

```python
def buildGraph(image):
        #This builds a graph from a given image
        (M, N) = image.shape
        im_flattened = image.flatten()
        G = nx.DiGraph()
        G.add_nodes_from(range(0,M*N))


        for i in range(M):
                for j in range(N):
                        #handles corner cases
                        if i==0 and j==0:
                                G.add_edge(i*N+j, (i+1)*N+j,weight=im_flattened[(i+1)*N+j])
                                G.add_edge(i*N+j, (i+1)*N+j+1,weight=im_flattened[(i+1)*N+j+1])
                        elif i==0 and j==N-1:
                                G.add_edge(i*N+j, (i+1)*N+j,weight=im_flattened[(i+1)*N+j])
                                G.add_edge(i*N+j, (i+1)*N+j-1,weight=im_flattened[(i+1)*N+j-1])
                        elif i==0 and j!=0 and j!=N-1:
                                G.add_edge(i*N+j, (i+1)*N+j, weight=im_flattened[(i+1)*N+j])
                                G.add_edge(i*N+j, (i+1)*N+j+1, weight=im_flattened[(i+1)*N+j+1])
                                G.add_edge(i*N+j, (i+1)*N+j-1, weight=im_flattened[(i+1)*N+j-1])
                        elif i==M-1:
                                pass
                        elif j==0 and i!=0 and i!=M-1:
                                G.add_edge(i*N+j, (i+1)*N+j,weight=im_flattened[(i+1)*N+j])
                                G.add_edge(i*N+j, (i+1)*N+j+1,weight=im_flattened[(i+1)*N+j+1])
                        elif j==N-1 and i!=0 and i!=M-1:
                                G.add_edge(i*N+j, (i+1)*N+j, weight=im_flattened[(i+1)*N+j])
                                G.add_edge(i*N+j, (i+1)*N+j-1, weight=im_flattened[(i+1)*N+j-1])
                        else:
                                G.add_edge(i*N+j, (i+1)*N+j, weight=im_flattened[(i+1)*N+j])
                                G.add_edge(i*N+j, (i+1)*N+j+1, weight=im_flattened[(i+1)*N+j+1])
                                G.add_edge(i*N+j, (i+1)*N+j-1, weight=im_flattened[(i+1)*N+j-1])
        return G

def SeamCarving(image, gradient, num_seams):

        M, N = image.shape
        im_flattened = image.flatten()
        gradsums = []
        # Repeat num_seams times.
        for i in range(num_seams):
                print i
                last_row = (M-1)*N
                G = buildGraph(gradient)
                for j in range(N):
                        for k in range(max(0,j+1-M), min(N, j+M)):
                                if(M>N or k<M):
                                        gradsums.append((i, last_row+k, nx.dijkstra_path_length(G, j, last_row+k)))

                gradsums.sort(key=lambda x: x[2])
                #Get index and weight
                x,y,value = gradsums[0]
                gradsums = []
                shortest_path = nx.dijkstra_path(G, x, y)
                #Delete the paths
                #reduce column by one
                N = N - 1
                im_flattened = np.delete(im_flattened, shortest_path)
                gradient = np.delete(gradient, shortest_path).reshape(M, N)
                print gradient.shape
        return im_flattened.reshape(M,N)
```

The following are some of my my attempts to perform Seam Carving on a set of images. Due to the computational complexity of the algorithm, only images of 100 x 150 pixels are used and only 10 seams are removed for each image. The first column contains the original image, whileas the second image is the horizontally compresssed image.

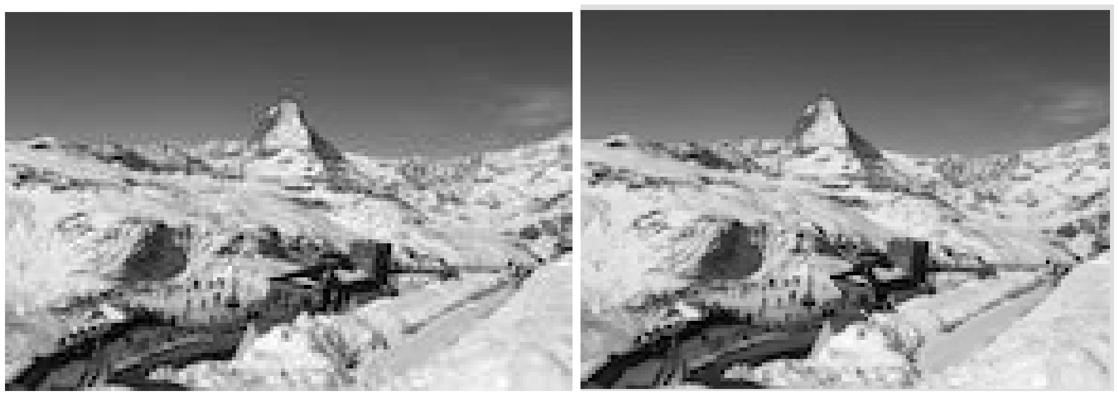

Figure 10: Seam Carving Examples: The Matterhorn

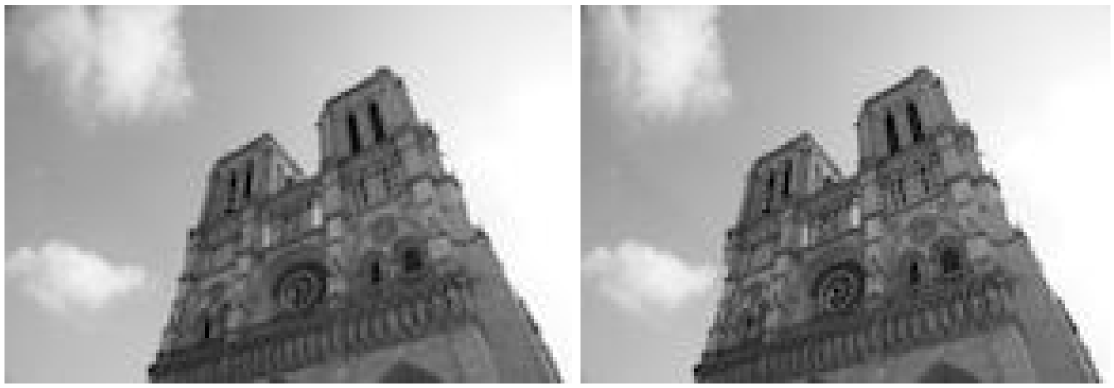

Figure 11: Seam Carving Examples: Arc De Triomphe

(b) The following is my attempt in segmenting the cattle. The graph is constructed based on the previous question but with a slight change, that is, instead of only allowing moving down, left-down and right-down, movements are allowed in all 8 directions. So a node can have 8 edges that connect to all its adjacent nodes. Different numbers of segmentation points are picked to see how the segmentation effect changes.

```python
def buildGraph_Segmentation(image):
        #This builds a graph from a given
        (M, N) = image.shape
        im_flattened = image.flatten()
        G = nx.DiGraph()
        G.add_nodes_from(range(0,M*N))


        for i in range(1, M-1):
                for j in range(1, N-1):
                        #Don't really care about corners here
                        #Down
                        G.add_edge(i*N+j, (i+1)*N+j,weight=im_flattened[(i+1)*N+j])
                        #Up
                        G.add_edge(i*N+j, (i-1)*N+j,weight=im_flattened[(i-1)*N+j])
                        #Left
                        G.add_edge(i*N+j, i*N+j+1,weight=im_flattened[i*N+j+1])
                        #Right
                        G.add_edge(i*N+j, i*N+j-1,weight=im_flattened[i*N+j-1])
                        #Down right
                        G.add_edge(i*N+j, (i+1)*N+j+1,weight=im_flattened[(i+1)*N+j+1])
                        #Up right
                        G.add_edge(i*N+j, (i-1)*N+j+1,weight=im_flattened[(i-1)*N+j+1])
                        #Down left
                        G.add_edge(i*N+j, (i+1)*N+j-1,weight=im_flattened[(i+1)*N+j-1])
                        #Up left
```

```
                                G.add_edge(i*N+j,
(i-1)*N+j-1,weight=im_flattened[(i-1)*N+j-1])
        return G

        G = buildGraph_Segmentation(mag)
        (M,N) = mag.shape
        plt.imshow(cattle, cmap='Greys_r')
        num_points = 8
        x = plt.ginput(num_points)
        plt.show()
        implot = plt.imshow(cattle, cmap='Greys_r')
        for i in xrange(1,num_points,1):
                x1, y1 = x[i-1]
                x2, y2 = x[i]
                point1 = int(y1*N+x1)
                point2 = int(y2*N+x2)
              (row, col) = np.unravel_index(nx.dijkstra_path(G, point1,
point2), (M,N))
              plt.scatter(col, row, c='red', s=3)
          x1, y1 = x[-1]
          x2, y2 = x[0]
          point1 = int(y1*N+x1)
          point2 = int(y2*N+x2)
        (row, col) = np.unravel_index(nx.dijkstra_path(G, point1, point2), (M,N))
        plt.scatter(col, row, c='red', s=3)
        plt.show()
```

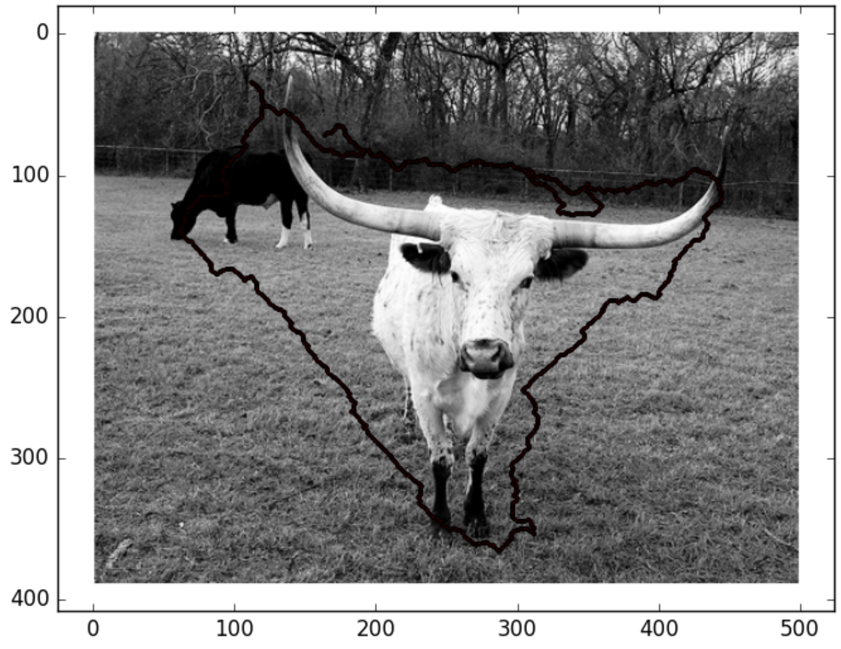

Figure 12: Segmentation Examples: 4 ginput Points

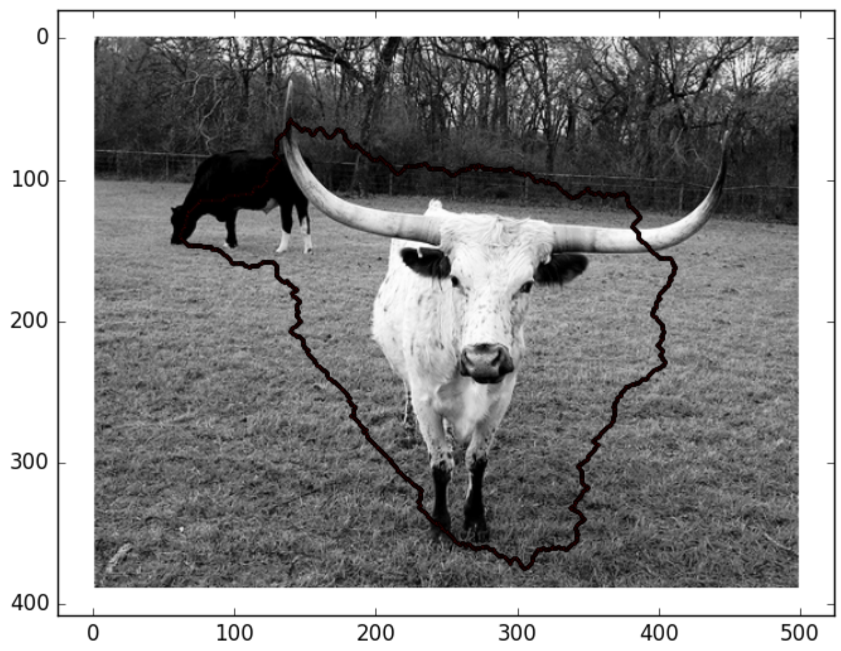

Figure 13: Segmentation Examples: 6 ginput Points

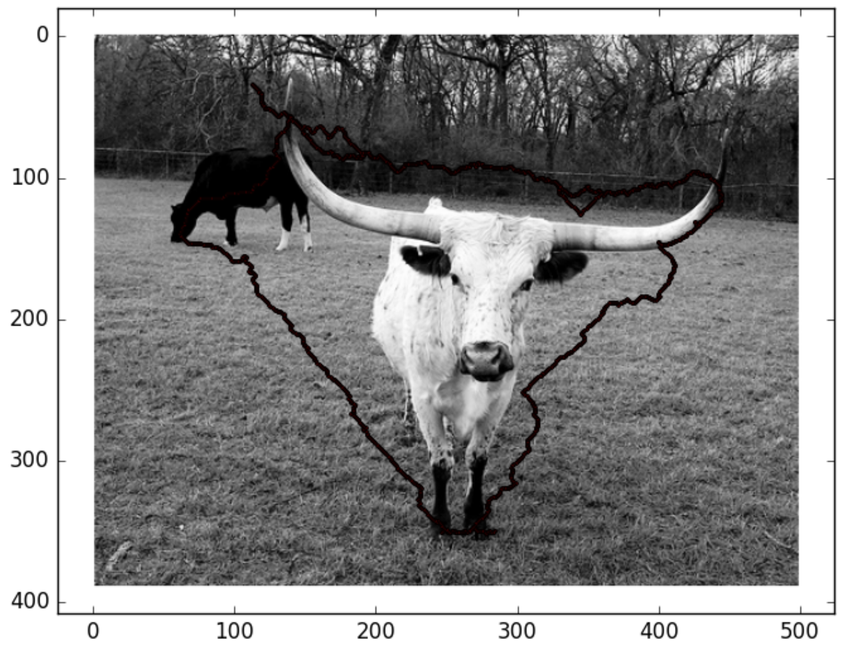

Figure 14: Segmentation Examples: 8 ginput Points