CSC420 Assignment 2

By: Aalan Mohammad

Feb, 2021

Question 1:

a)

I used one function to perform the seam carving but I will be covering parts a) to d) separately.

Starting off, the function takes a colored image as input and the number of columns to be deleted, this is helpful for parts C and D later on. Returned the image with columns removed using seam carving.

a) Compute magnitude of gradients of an image

```
def seam_carving(image,columns_deleted):
    """
    perform seam carving algorithm on the given image
    """
    # Read image with RGB norm, as discussed on Discussion
    img = io.imread(image)

    height = img.shape[0]
    width = img.shape[1]

    red_scale = np.zeros((height,width))
    blue_scale = np.zeros((height,width))
    green_scale = np.zeros((height,width))

    for row in range(height):
        for col in range(width):
            # using norm of RGB vector
            red_scale[row][col] = img[row][col][0]
            blue_scale[row][col] = img[row][col][1]
            green_scale[row][col] = img[row][col][2]

    # (a) Compute Magnitude of gradients for each color scale
    red_x = ndimage.sobel(red_scale,axis=0)
    red_y = ndimage.sobel(red_scale,axis=1)
    blue_x = ndimage.sobel(blue_scale,axis=0)
    blue_y = ndimage.sobel(blue_scale,axis=1)
    green_x = ndimage.sobel(green_scale,axis=0)
    green_y = ndimage.sobel(green_scale,axis=1)

    magnitudes = np.zeros((height,width))

    for row in range(height):
        for col in range(width):
            # compute magnitude of gradients for each pixel for each color
            red = math.sqrt((red_x[row,col] ** 2) + (red_y[row,col] **2))
            blue = math.sqrt((blue_x[row,col] ** 2) + (blue_y[row,col] **2))
            green = math.sqrt((green_x[row,col] ** 2) + (green_y[row,col] **2))
            magnitudes[row,col] = math.sqrt((red ** 2) +  (blue ** 2) + (green ** 2))
    print("found magnitudes")
```

For this part, I took the TA's advice and square rooted the individual color channels instead of gray scaling the image as mentioned on the Quercus Discussion.

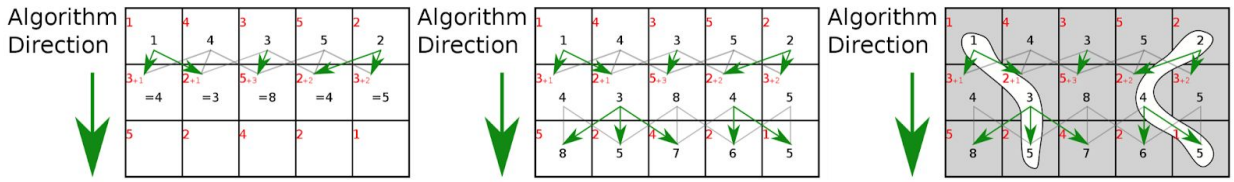
b) Now we have to find the connected path of pixels that has the smallest sum of gradients. For this part I used a helper function that takes the magnitude of gradients and returns the smallest path using dynamic programming. I also used a Python Pixel class that stores information for the dynamic programming part.

```
def smallest_path(magnitudes):
    """
    returns a list of valid path from top to bottom of smallest magnitudes
    using dynamic programming, Valid paths need to have neighbors from
    image.
    """
    final_path = []

    matrix = np.empty(magnitudes.shape, dtype=object)
    height = matrix.shape[0]
    width = matrix.shape[1]

    # fill matrix with Pixel objects containg information (valid, value,parent_neighbor)
    # Pixel class is below, containing information from the path algorithm
    # Valid means if a Pixels are marked and cannot be used again
    # value is the pixel magnitude and parent neighbor is the pixels parent in the path

    for row in range(matrix.shape[0]):
        for col in range(matrix.shape[1]):
            if row == 0:
                # current sum is the original magnitude and no parent neighbor
                matrix[row][col] = Pixel(magnitudes[row][col],-10)
            else:
                if col == 0:
                    # left boundary of the image, compare top and top right only
                    lst = [matrix[row-1][col].val,matrix[row-1][col+1].val]
                    matrix[row][col] = Pixel(magnitudes[row][col] + min(lst),lst.index(min(lst)))
                elif col == matrix.shape[1] - 1:
                    # right boundary of image, compare top and top left only
                    lst = [matrix[row-1][col-1].val,matrix[row-1][col].val]
                    matrix[row][col] = Pixel(magnitudes[row][col] + min(lst),lst.index(min(lst))-1)
                else:
                    # compare all three possible paths, pick the smallest
                    lst = [matrix[row-1][col-1].val,matrix[row-1][col].val,matrix[row-1][col+1].val]
                    matrix[row][col] = Pixel(magnitudes[row][col] + min(lst),lst.index(min(lst))-1)

    # now find the the smallest sum in the last row, and take that path
    # get last row indices based on lowest sum
    lowest, index = 9999999, 0
    for col in range(matrix[-1].shape[0]):
        if matrix[-1][col].val < lowest:
            index = col
            lowest = matrix[-1][col].val

    # now get the path starting from bottom to the top
    current_col = index
    for row in range(height - 1, -1, -1):
        # traverse from down to up
        final_path.append((row,current_col))
        current_col = current_col + matrix[row][current_col].neighbor
    return final_path
```

```
class Pixel:
    """A pixel class containing
    information on pixel validation for paths, parent neighbor for path,and magnitude intensity
    """

    def __init__(self, value, neighbor):
        self.val = value
        self.neighbor = neighbor
```

I also used a Pixel class seen in the code above, it is just for keeping the code clean and readable.

The dynamic algorithm was taken from Wikipedia:




The top row has nothing above it, so the energies are the same as the source image.

For each pixel in the rest of the rows, the energy is its own energy plus the minimal of the three energies above. Repeat until the bottom is reached.

For the lowest energies we have at the end, work back up the minimals to recover the seam with minimal energy.

c) Remove the pixels in the path from the image. This gives you a new image with one column less.

This part is just calling our helper, find the path and remove it.

```
print("removing " + str(columns_deleted) + " path(s)")
final_img = img.copy()
i = 0
while i < columns_deleted:
    img_path = smallest_path(magnitudes)
    for row in range(height):
        skip = 0
        for col in range(width - 1):
            if (row,col) in img_path or skip == 1:
                # shift to the left to fill in
                final_img[row,col] = final_img[row,col+1]
                magnitudes[row,col] = magnitudes[row,col+1]
                skip = 1
    final_img = np.delete(final_img, -1,1)
    magnitudes = np.delete(magnitudes, -1,1)
    width -= 1
    i += 1
plt.figure(1)
plt.axis("off")
plt.imshow(final_img)
plt.show()
```

This is the result from calling the function seam_carving(“tree.jpg”, 1)

You can see the path on the left, from the top to the bottom.

D) Now we remove few more paths, For this I removed 100 columns, lets compare the original and the 100 less columns:

Original

100 less columns

## Question 2:

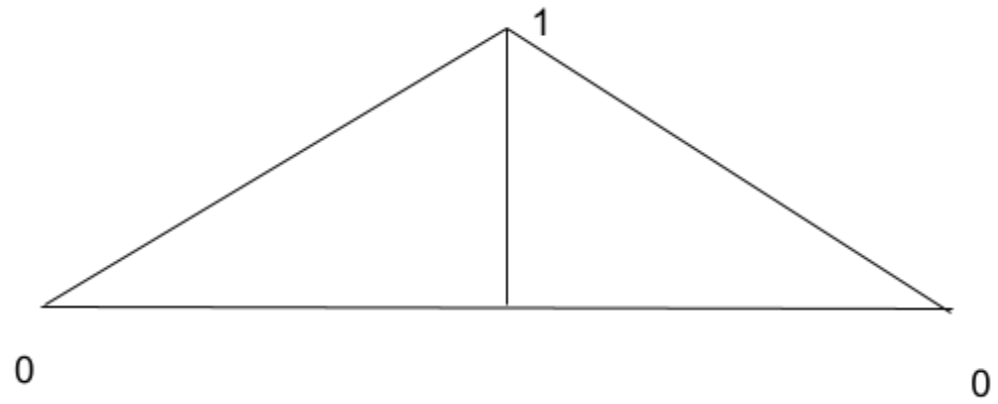
a)Write down the mathematical form of the convolution filter that performs up- scaling of a 1D signal by a factor d.

For this, we can take a look at linear interpolation, And we had is the linear filter would depend on the factor d.

It will have the form of:

$$h = [0, 1/d, ..., d - 1/d, 1, -(d-1)/d, .., -1/d, 0]$$

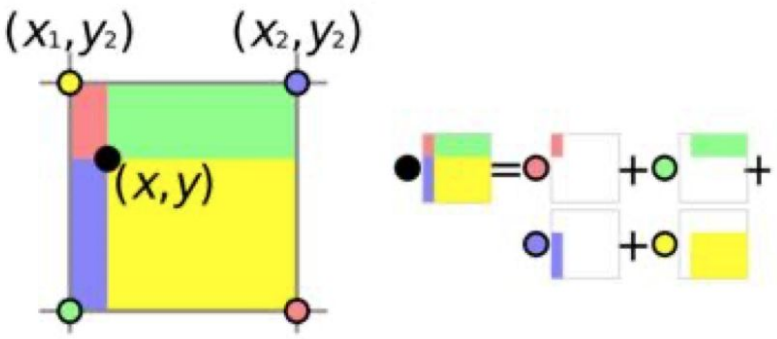
The graph of this filter is like a triangle




It has the center at 1 and then symmetrical slopes, on each side of the center.

b) Function that scales an image 3x its resolution. I used bilinear interpolation to perform the upsaclling.

Every 3rd pixel is copied to a matrix 3 times its size, and the rest of the values we perform Bilinear interpolation seen in lecture.


$(x_1, y_2)$
$(x_2, y_2)$
$(x, y)$
=
+
+
+

```
def upscalling(image):
    """
    Given an image, up scale 3 times its size.
    """
    # read RGB image
    img = io.imread(image)

    # make it 3 times bigger
    bigger = np.zeros((img.shape[0] * 3,img.shape[1] * 3,3))

    # copy values in every third pixel, copy the RGB values
    for row in range(bigger.shape[0]):
        for col in range(bigger.shape[1]):
            if (row % 3 == 0) and (col % 3 == 0):
                # copy here
                bigger[row,col] = img[row // 3, col // 3]

    # now perform bilinear interpolation
    for row in range(bigger.shape[0]):
        for col in range(bigger.shape[1]):
            if not((row % 3 == 0) and (col % 3 == 0)):
                # find x1, x2, y1, y2
                x1 = 3 * math.floor(col/3)
                x2 = x1 + 3
                y1 = 3 * math.ceil(row/3)
                y2 = y1 - 3
                x ,y = col, row
                width, height = bigger.shape[1], bigger.shape[0]
                # apply formula
                red_total,blue_total,green_total = 0,0,0
                if (x1 < width and y1 < height):
                    red_total += bigger[y1,x1][0] * abs(x2 - x) * abs(y2 - y)
                    blue_total += bigger[y1,x1][1] * abs(x2 - x) * abs(y2 - y)
                    red_total += bigger[y1,x1][2] * abs(x2 - x) * abs(y2 - y)
                if (x1 < width and y2 < height):
                    red_total += bigger[y2,x1][0] * abs(x2 - x) * abs(y - y1)
                    blue_total += bigger[y2,x1][1] * abs(x2 - x) * abs(y - y1)
                    green_total += bigger[y2,x1][2] * abs(x2 - x) * abs(y - y1)
                if (x2 < width and y1 < height):
                    red_total += bigger[y1,x2][0] * abs(x1 - x) * abs(y2 - y)
                    blue_total += bigger[y1,x2][1] * abs(x1 - x) * abs(y2 - y)
                    green_total += bigger[y1,x2][2] * abs(x1 - x) * abs(y2 - y)
                if (x2 < width and y2 < height):
                    red_total += bigger[y2,x2][0] * abs(x - x1) * abs(y - y1)
                    blue_total += bigger[y2,x2][1] * abs(x - x1) * abs(y - y1)
                    green_total += bigger[y2,x2][2] * abs(x - x1) * abs(y - y1)
                # now paste the RGB totals
                bigger[row,col][0] = int(1/9 * red_total)
                bigger[row,col][1] = int(1/9 * blue_total)
                bigger[row,col][2] = int(1/9 * green_total)
    return bigger
```

## Question 3

a)Implement a function to perform Harris corner detection. The function should take as input an image, and return corners.

My implementation takes in an image, Gaussian filter sigma value, threshold or R, and Non maximum suppression window size.

```
def Corner_Harris(image, gauss_sigma, threshold,window_size):
    """
    Given an image, perform the Corner Harris Detector, place circles on
    the image where the corners are
    """
    #Read the image
    img = io.imread(image,as_gray=True)

    # (1) Compute gradients Ix and Iy
    Ix = ndimage.sobel(img,axis=0)
    Iy = ndimage.sobel(img,axis=1)
    # (2) Compute Ixx, Iyy, Ixy
    Ixx = Ix * Ix
    Iyy = Iy * Iy
    Ixy = Ix * Iy
    # (3) Average Gaussian gives M
    g_Ixx = ndimage.gaussian_filter(Ixx, gauss_sigma)
    g_Iyy = ndimage.gaussian_filter(Iyy, gauss_sigma)
    g_Ixy = ndimage.gaussian_filter(Ixy, gauss_sigma)

    # (4) Compute R = det(M) - 0.04 * trace(M)^2 for each window image
    final_img = np.zeros(img.shape)
    for row in range(final_img.shape[0]):
        for col in range(final_img.shape[1]):
            first = g_Ixx[row][col]
            second = third = g_Ixy[row][col]
            last = g_Iyy[row][col]
            M = np.array([[first,second],[third, last]])
            determinant = np.linalg.det(M)
            trace = np.matrix.trace(M)
            final_img[row][col] = determinant - (0.04 * trace ** 2)
    # (5) Find points with large R > threshold

    for row in range(final_img.shape[0]):
        for col in range(final_img.shape[1]):
            if final_img[row][col] <= threshold:
                final_img[row][col] = 0


   # (6) perform Non maximum suppression
   corners = []
   for row in range(0,final_img.shape[0] - window_size,window_size):
       for col in range(0,final_img.shape[1] - window_size,window_size):
           patch = final_img[row:row+window_size,col:col+window_size]
           # find max index value
           biggest ,biggest_i, biggest_j = 0, 0, 0
           for i in range(window_size):
               for j in range(window_size):
                   if patch[i][j] > biggest:
                       biggest = patch[i][j]
                       biggest_i , biggest_j = row + i, col + j
           corners.append((biggest_i, biggest_j))

   plt.figure(1)
   plt.axis("off")
   plt.imshow(img,cmap='gray')
   for corner in corners:
       # print the corners with a red dot
       plt.plot(corner[1], corner[0], 'o', markeredgecolor='r', markerfacecolor='none', markersize=1)
   plt.show()
```

The algorithm follows all 6 steps from lecture including Non maximum suppression. The output is the original image with the corners.

b) I ran Corner_Harris(“building.jpg”, 1, 0.5, 20)

So gaussian filter had a sigma of 1, threshold of 0.5 and NMS window size of 20