Coding Assignment 3: Seam Carving Write-up

# 1. Reflection

Seam carving assignment facilitated me to relate and implement the dynamic programming concepts to the real word scenarios, where I developed a data-type that manipulates the aspect ratio of the image with the help of the several sub-modules. It helped me to understand with a practical hands-on experience that for certain scenarios in which sub-problems has repeated sub-sub-problem can be solved with a more efficient approach of dynamic programming than divide and conquer. The trade-off is between time and memory, where memorization complements recursion by storing the cost of the paths and path indices for backtracking. On implementation grounds, assignment 3 also gives me exposure to the set of tools available in python like scipy and skimage, which has excellent capability to handle large amount of data, here the data was pixel information of the input image.

I implemented seam carving by finding the energy of the image pixel with the help of dual energy gradient function and then calculating the minimum cost path, by taking pixels as the nodes and energy of pixels as the weight of taking corresponding path. Once the path costs were calculated and saved in a matrix, a path with minimum cost was selected. Finally, the pixels that are in the path of identified minimum cost path called as seam was removed from the image to create a new image. The above process was repeated a number of times to achieved the desired image aspect ratio.

Hence, Seam Carving assignment was a great learning experience.

# 2. Testing Output

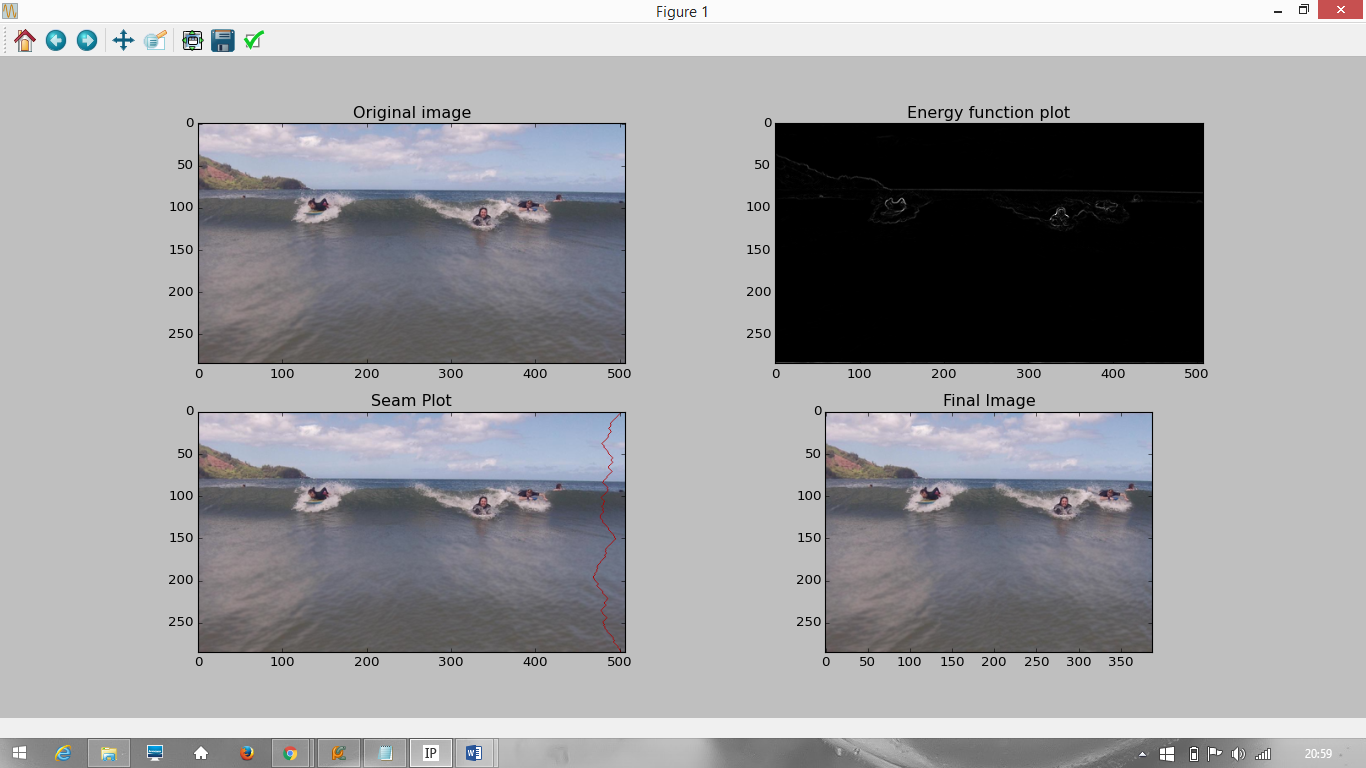
1. The below image shows the figure tray of scikit-image showing:

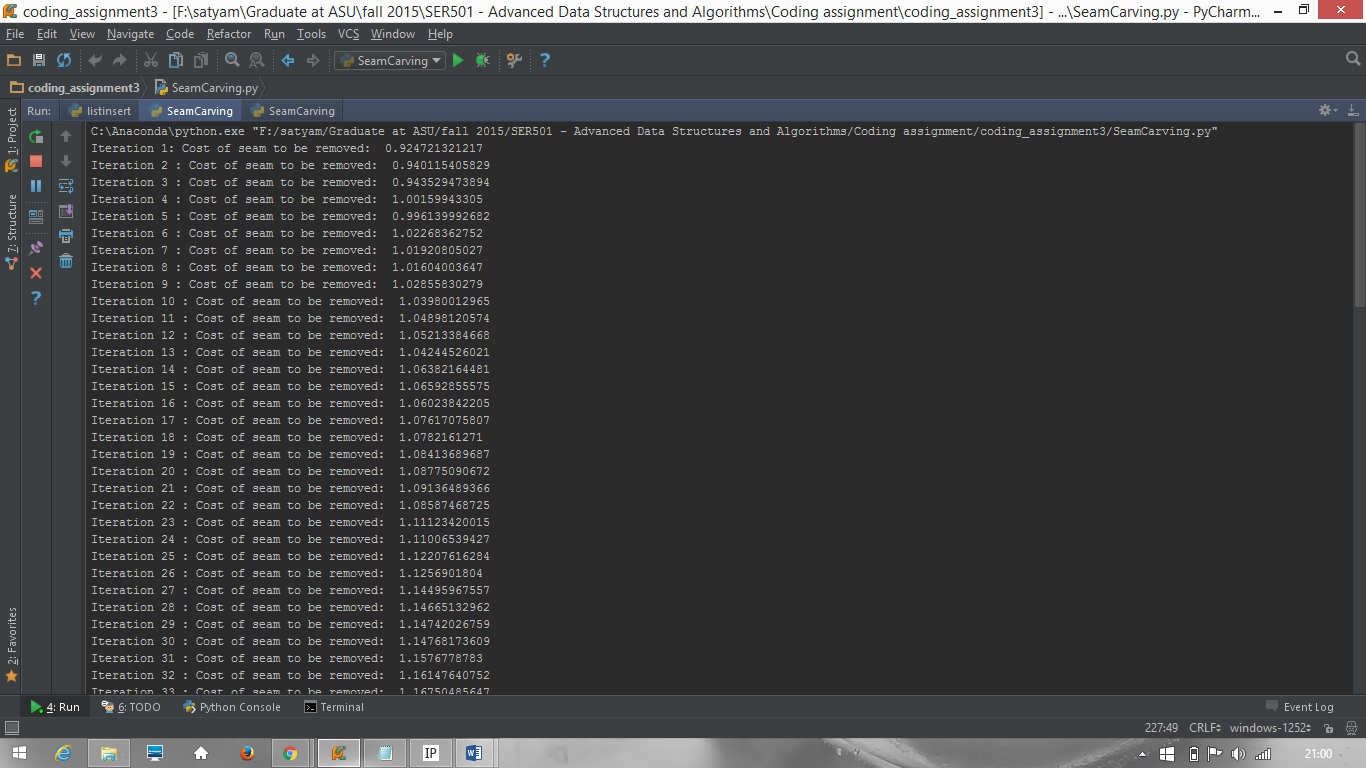
The input image,

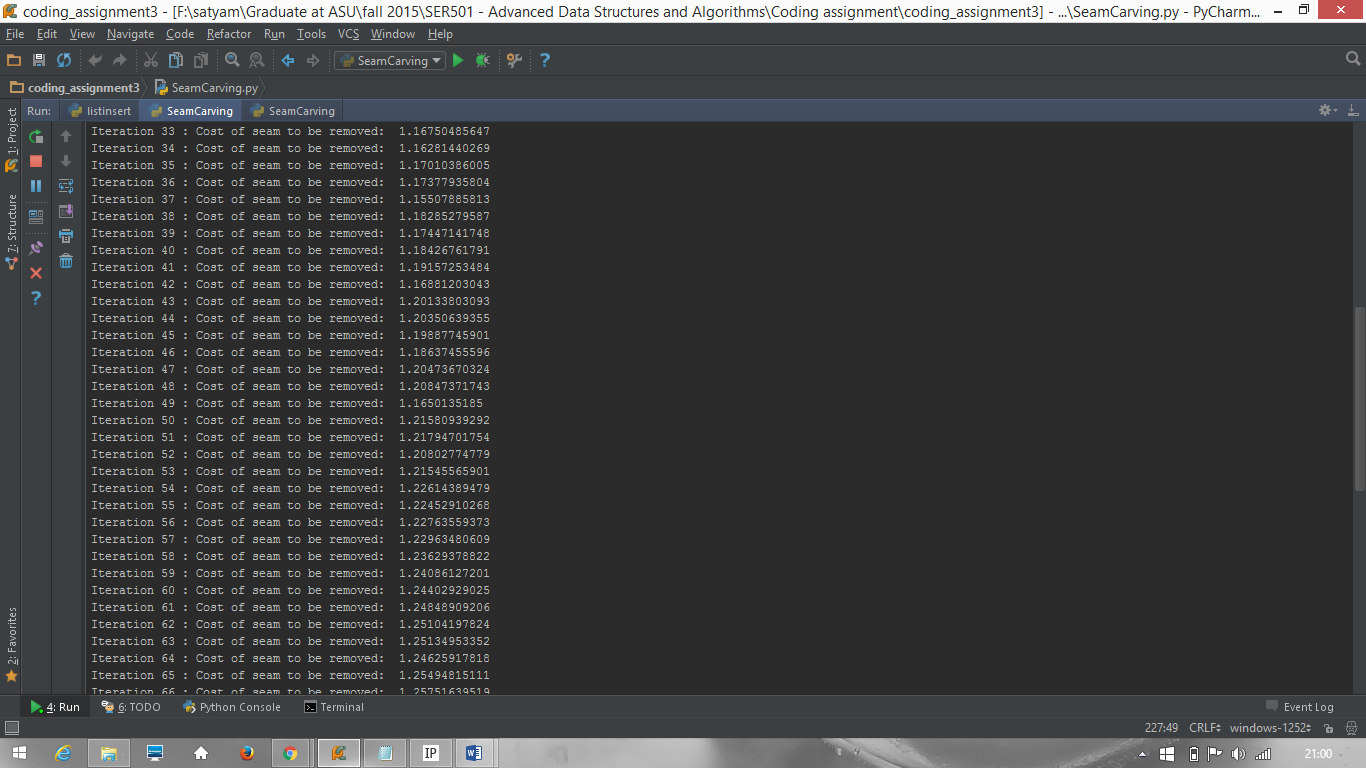
Energy of each pixel,

Seam plot of first seam found,

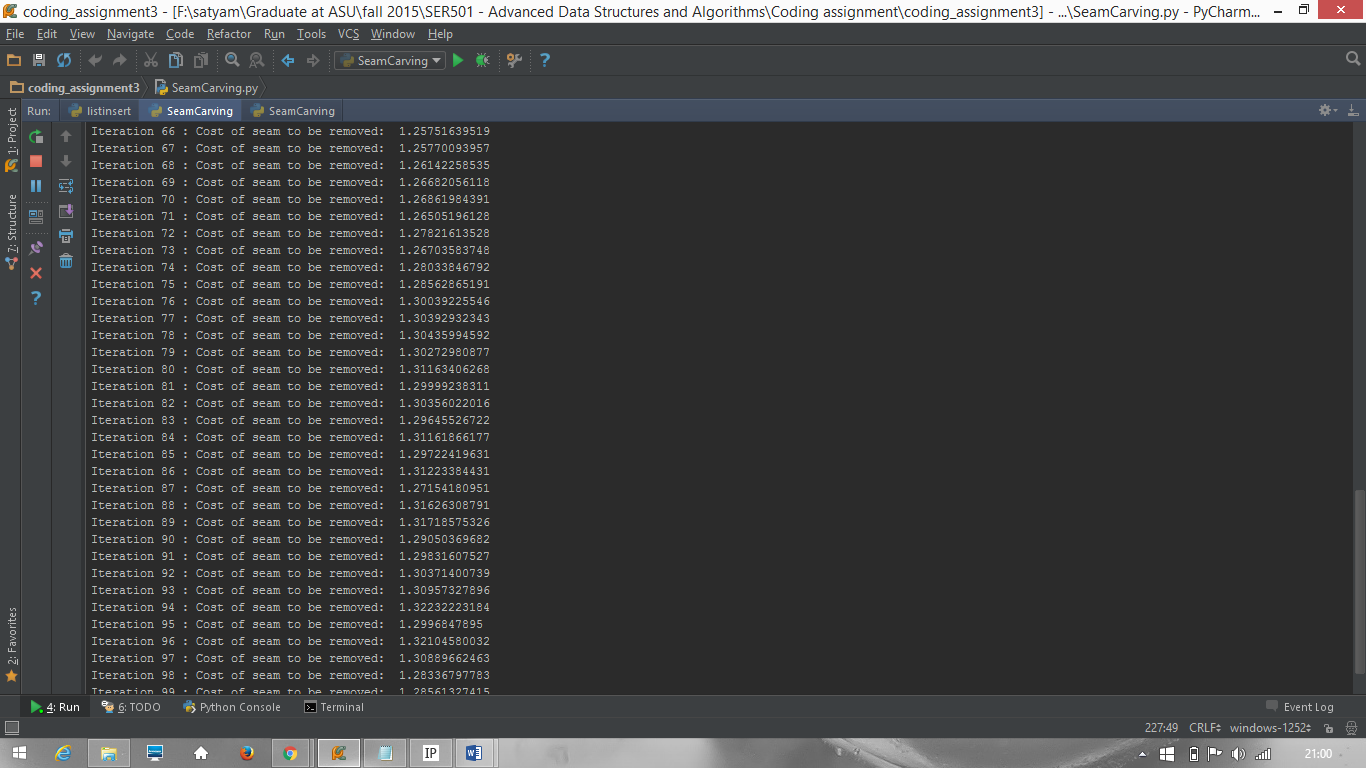
Final reduced image after a series of seam removal



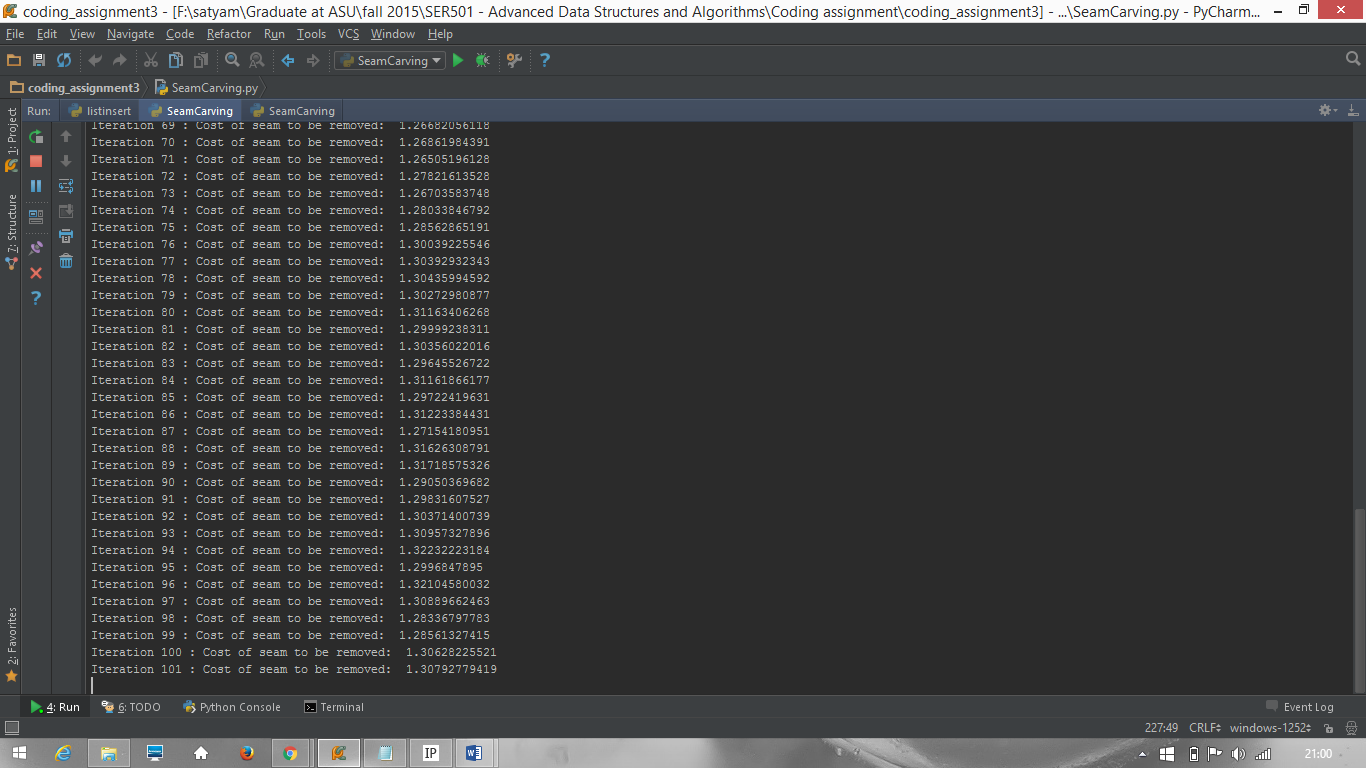
1. Below image shows the path cost of the seam selected for removal from iteration 1 to 33
2. Below image shows the path cost of the seam selected for removal from iteration 33 to 65



1. Below image shows the path cost of the seam selected for removal from iteration 66 to 98



1. Below image shows the path cost of the seam selected for removal from iteration 98 to 101

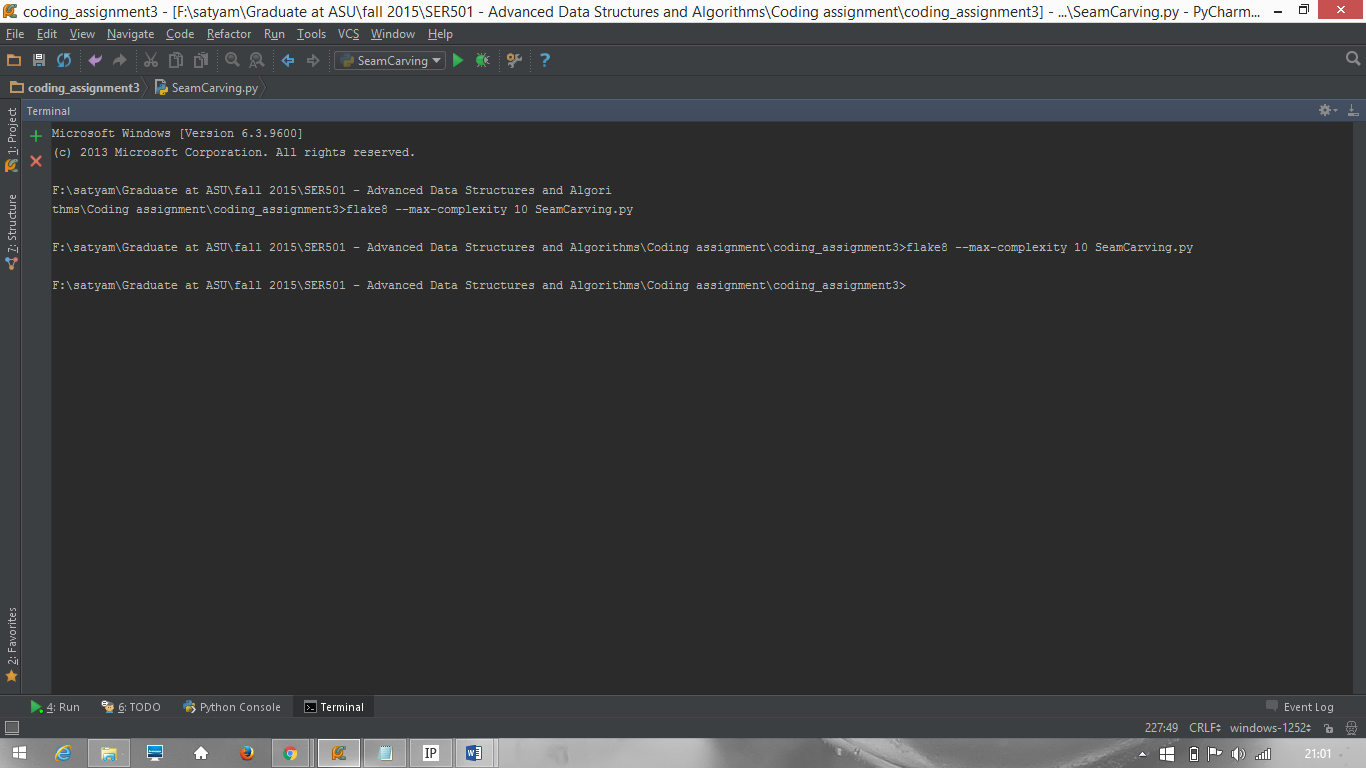


# 3. Static Analysis / Compilation Output

For static analysis, I have used Flake8.

Below is the Flake8 output which shows that

* All the functions written have code complexity less than 10 and
* The code is free of warnings



# 4. Source Code

This section shows the formatted code to achieve the seam carving functionality.

Each function includes a doc-string explaining the input the function takes, the output of the function and a short description of the working of the function.

**import** pylab

**import** math

**from** skimage **import** img\_as\_float

**class** SeamCarving:

**def** dual\_gradient\_energy(self, img\_path):

*"""*

*Input: Path of the image*

*Output:*

*a W x H array of floats, the energy at each pixel in input img.*

*W is width of the image*

*H is height of the image*

*Working:*

*Slices the image into its R, G and B components.*

*Calculates energy of each pixel by dual gradient energy function.*

*"""*

img = pylab.imread(img\_path)

      img = img\_as\_float(img)

      height, width = img.shape[:2]

      r = img[:, :, 0]

      g = img[:, :, 1]

      b = img[:, :, 2]

      energy = [[-1 **for** i **in** range(width)] **for** j **in** range(height)]

**for** i **in** range(height):

**for** j **in** range(width):

**if** i == 0:

                  ry = r[i+1][j] - r[height-1][j]

                  gy = g[i+1][j] - g[height-1][j]

                  by = b[i+1][j] - b[height-1][j]

**elif** i == height - 1:

                  ry = r[0][j] - r[i-1][j]

                  gy = g[0][j] - g[i-1][j]

                  by = b[0][j] - b[i-1][j]

**else**:

                  ry = r[i+1][j] - r[i-1][j]

                  gy = g[i+1][j] - g[i-1][j]

                  by = b[i+1][j] - b[i-1][j]

**if** j == 0:

                  rx = r[i][j+1] - r[i][width-1]

                  gx = g[i][j+1] - g[i][width-1]

                  bx = b[i][j+1] - b[i][width-1]

**elif** j == width - 1:

                  rx = r[i][0] - r[i][j-1]

                  gx = g[i][0] - g[i][j-1]

                  bx = b[i][0] - b[i][j-1]

**else**:

                  rx = r[i][j+1] - r[i][j-1]

                  gx = g[i][j+1] - g[i][j-1]

                  bx = b[i][j+1] - b[i][j-1]

              delta\_x = math.pow(rx, 2) + math.pow(gx, 2) + math.pow(bx, 2)

              delta\_y = math.pow(ry, 2) + math.pow(gy, 2) + math.pow(by, 2)

              energy[i][j] = delta\_x + delta\_y

**return** energy

**def** find\_seam(self, energy):

*"""*

*Input: Array of energy of each pixel*

*Output:*

*1. A List having column numbers for each row of the image,*

*depicting the seam with lowest cost.*

*2. Cost of the seam found with lowest cost.*

*Working:*

*Performs memoization by saving the solution of sub-problems in the*

*form of matrices cost and path.*

*Cost matrix helps us to find minimum cost at iteration without*

*recomputing it.*

*Path matrix helps us to find the seam co-ordinates without*

*recomputing it.*

*"""*

energy = img\_as\_float(energy)

      height, width = energy.shape

      cost = [[0 **for** i **in** range(width)] **for** j **in** range(height)]

      path = [[-1 **for** i **in** range(width)] **for** j **in** range(height)]

**for** i **in** range(width):

          cost[height-1][i] = energy[height-1][i]

**for** i **in** range(height-2, -1, -1):

**for** j **in** range(width):

**if** j == 0:

                  adj = [cost[i+1][j], cost[i+1][j+1]]

                  cost[i][j] = energy[i][j] + min(adj)

                  path[i][j] = j + adj.index(min(adj))

**elif** j == width-1:

                  adj = [cost[i+1][j-1], cost[i+1][j]]

                  cost[i][j] = energy[i][j] + min(adj)

                  path[i][j] = j + adj.index(min(adj)) - 1

**else**:

                  adj = [cost[i+1][j-1], cost[i+1][j], cost[i+1][j+1]]

                  cost[i][j] = energy[i][j] + min(adj)

                  path[i][j] = j + adj.index(min(adj)) - 1

      min\_cost = cost[0][0]

      start\_index = 0

**for** i **in** range(width):

**if** cost[0][i] < min\_cost:

              min\_cost = cost[0][i]

              start\_index = i

      seam = [start\_index]

      next\_index = start\_index

**for** i **in** range(height):

          seam = seam + [path[i][next\_index]]

          next\_index = path[i][next\_index]

**return** seam, min\_cost

**def** plot\_seam(self, image, seam):

*"""*

*Input:*

*1. image: Original image in the first instance and then*

*reduced image in subsequent iteration*

*2. seam: list having column numbers for each row of the image,*

*depicting the seam to be removed.*

*Output:*

*Input image with the seam drawn, showing the seam visualization*

*Working:*

*Replaces the RGB values of the seam pixel co-ordinates identified by*

*list - seam with (0.7, 0, 0) which RGB value for red*

*"""*

seam\_plot = pylab.imread(image)

      seam\_plot = img\_as\_float(seam\_plot)

      height, width = seam\_plot.shape[0:2]

**for** i **in** range(height):

**for** j **in** range(width):

**if** seam[i] == j:

                  seam\_plot[i][j][0] = 0.7

                  seam\_plot[i][j][1] = 0

                  seam\_plot[i][j][2] = 0

      pylab.imsave(**"SeamPlot"**, seam\_plot)

**return** seam\_plot

**def** remove\_seam(self, img, seam):

*"""*

*Input:*

*1. img: Original image in the first instance and then*

*reduced image in subsequent iteration*

*2. seam: list having column numbers for each row of the image,*

*depicting the seam to be removed.*

*Output:*

*NewImage.png: New image saved having the seam pixels removed*

*new\_img: New image having the seam pixels removed from input image*

*Working:*

*We copy the non-seam pixels from img to a new image.*

*We adopt this method since nd array is immutable.*

*"""*

img = pylab.imread(img)

      img = img\_as\_float(img)

      height = img.shape[0]

      width = img.shape[1]

      new\_img = [[[0 **for** k **in** range(3)] **for** i **in** range(width-1)]

**for** j **in** range(height)]

      new\_img = img\_as\_float(new\_img)

      y = 0

**for** i **in** range(height):

**for** j **in** range(width):

**if** j != seam[i]:

                  new\_img[i][y][0] = img[i][j][0]

                  new\_img[i][y][1] = img[i][j][1]

                  new\_img[i][y][2] = img[i][j][2]

                  y = (y + 1) % (width - 1)

      pylab.imsave(**'NewImage.png'**, new\_img)

**return** new\_img

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

  sc = SeamCarving()

  img = pylab.imread(**"HJoceanSmall.png"**)

  img = img\_as\_float(img)

  pylab.figure()

  pylab.gray()

*# Plotting the input image*

pylab.subplot(2, 2, 1)

  pylab.imshow(img)

  pylab.title(**'Original image'**)

*# Plotting the energy function of the image*

energy\_image = sc.dual\_gradient\_energy(**'HJoceanSmall.png'**)

  pylab.subplot(2, 2, 2)

  pylab.imshow(energy\_image)

  pylab.title(**'Energy function plot'**)

  seam, mininum\_cost = sc.find\_seam(energy\_image)

**print "Iteration 1: Cost of seam to be removed: "**, mininum\_cost

*# Plotting the seam found on the input image*

seam\_plot = sc.plot\_seam(**'HJoceanSmall.png'**, seam)

  pylab.subplot(2, 2, 3)

  pylab.imshow(seam\_plot)

  pylab.title(**'Seam Plot'**)

  new\_image = sc.remove\_seam(**'HJoceanSmall.png'**, seam)

*# Iterating the entire process 100 times to see significant reduction.*

**for** i **in** range(100):

      energy\_image = sc.dual\_gradient\_energy(**'NewImage.png'**)

      seam, mininum\_cost = sc.find\_seam(energy\_image)

**print "Iteration"**, (i+2), **": Cost of seam to be removed: "**,\

          mininum\_cost

      seam\_plot = sc.plot\_seam(**'SeamPlot.png'**, seam)

      new\_image = sc.remove\_seam(**'NewImage.png'**, seam)

*# Plotting the final image after computation*

pylab.subplot(2, 2, 4)

  pylab.imshow(new\_image)

  pylab.title(**'Final Image'**)

  pylab.show()

Revised rubric for coding assignments.

This is a 5-point rubric for coding projects. Graders should refrain from using fractional points (they are a pain to defend), choose the one one number that best reflects the assignment. For assignments with multiple parts, choose the lowest scoring part.

This rubric is based on the idea that students submit PDF write-ups with their coding assignment. Write-ups *must* be PDF’s with the source code so that graders can quickly view them annotate them using blackboard. The rubric does not address specific learning objectives — the assumption is that by completing the assignment the student has implicitly demonstrated some set objectives in addition to coding.

0 points — Student does not submit **all** parts of the assignment, meaning *both* a **PDF** writeup (all sections) that includes source code and output of testing, as well as a .**zip** file with source code.

2 points — The code does not run or does not *appear* to be able to run. The code it much longer than it should be, or does not appear to follow the scaffolding provided. The grader can but **does not have to verify that it does not run**, it is the student’s responsibility to provide a writeup that is sufficiently convincing. Student may not appeal by coming after the fact and showing that code runs on their machine.   
When grading, the grader should indicate portions of the code by annotating the writeup that are suspicious.

3 points — The code runs or looks like it would run, but the student has not shown via their writeup that it produces the correct result on reasonable inputs. Or, the student has implemented algorithms using approaches other than the ones indicated in the assignment, or the implementation has the wrong asymptotic complexity or that demonstrates a lack of understanding of the assignments objectives. The grader can, but **does not have to run the code to verify correctness** — it is the student’s responsibility to make a convincing case that the output and the algorithm is correct.   
When grading, the grader should indicate by annotating the write-up where results

4 points — The code runs or appears to run correctly, but has readability or style issues. The student has not demonstrated that their code has passed style guidelines, or the student's implementation appears to be unnecessarily complex (even though it looks like it works).   
When grading, the grader should indicate the style problems.

5 points — No issues that we can spot.