CIS581, Computer Vision
Project 4 Part A
Image Carving and Image Blending
Due November 29, 3:00pm

Instructions. This is a **group assignment**. All matlab functions should follow the names and arguments stated in the problems in order for them to run properly with the grading script. To submit the assignment, upload a zip file containing all your code and files via Canvas.

Turn-in

Before you submit, you need to register your group in canvas. Join the same empty group or create a new one. Each group have at most two people. Notice that the group will keep for the part B of project 4.

Then, zip your files into a folder named `GroupNumber_Project4_A.zip` and submit it via Canvas.

This should include:

- For Task 1, your .m files for 5 required Matlab functions you need to complete: (`carv.m`, `cumMinEngVer.m` and `cumMinEngHor.m`, `rmVerSeam.m` and `rmHorSeam.m`)
- For Task 2, your .m files for the 6 required Matlab functions (`seamlessCloningPoisson.m`, `maskImage.m`, `getIndexes.m`, `getCoefMatrix.m`, `getSolutionVect.m` and `reconstructImg.m`)
- any demo .m script(s) for reproducing/showing your results
- the input images you used
- the resulting resized image(s) and blended image(s) respectively.

- a PDF showing your results and describing any additional features of your implementation.
  - For Part 1, show the initial and final image(s), and a copy of the original image with seams colored in (like in the referenced paper, or notes covered in class).
  - For Part 2, show the source image, target image and final blending image.

The pdf file should be in the top level of the ZIP file, and other files should be in the subfolder `carving` and `blending`.

Overview

This project will have two focuses: Image resizing utilizing seam carving, and image blending.

Part 1 will focus on the concepts of image resizing utilizing the principals supporting minimum-energy seam carving and dynamic programming. The goal of this part of the project is to help you understand an important approach towards resizing images while attempting to preserve the integrity of important image information. The methodology closely follows that which is outlined in:

"*Seam Carving for Content-Aware Image Resizing*"; Avidan, S. & Shamir, A.; 2007

Part 2, on image blending, focuses on the gradient domain blending based on poisson equation. The goal of this part is to create a blended image that is partially cloned from the source image. You can learn the application of poisson equation and how to solve a sparse linear system. We will follow the technique described in the following paper, which is available on the course website:

"*Poisson Image Editing*", Perez, P.; Gangnet, M.; Blake, A. SIGGRAPH 2003

Task 1: Image Carving and Resizing

For this part of the project, you will be implementing image resizing utilizing scene carving. The structure of this part of the project is based heavily on the methods outlined by Avidan & Shamir in their paper: "*Seam Carving for Content-Aware Image Resizing*", which you are strongly encouraged to read prior to starting this part of the project (a link to the paper will be made available on the course wiki).

You are tasked with filling in 5 functions: `carv.m`, `cumMinEngVer.m` and `cumMinEngHor.m`, `rmVerSeam.m` and `rmHorSeam.m`.

We begin by computing the energy map, $e(I) = |\frac{\delta}{\delta x}I| + |\frac{\delta}{\delta y}I|$. We provide a function, `genEdgeMap.m` which computes this for you, in order to maintain consistency in the method by which the energy map is computed. Your task is to fill in the code that follows. For this project, we will be testing you strictly on shrinking an image (making it smaller), though the ideas applied towards enlarging an image are very similar to those of shrinking one.

1. Computing Cumulative Minimum Energy

You need to complete the code for the functions `cumMinEngVer.m` and `cumMinEngHor.m`, which correspond to computing the cumulative minimum energy over the vertical and horizontal seam directions, respectively.

The function `cumMinEngVer.m` has the following structure:

```
[My, Tby] = cumMinEngHor(e)
```

(INPUT) `e`: $n \times m$ matrix representing the energy map.
(OUTPUT) `My`: $n \times m$ matrix representing the cumulative minimum energy map along horizontal direction.
(OUTPUT) `Tby`: $n \times m$ matrix representing the backtrack table along horizontal direction.

You'll notice that Avidan & Shamir, as well as our class notes, define seams for an $n \times m$ image, I as follows:

- A vertical seam $\mathbf{s^x} = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n$, s.t $\forall i, |x(i) - x(i-1)| \leq 1$, where x is a mapping $x : [1, \ldots, n] \rightarrow [1, \ldots, m]$, i.e. the corresponding column for a seam pixel at row i along an 8-connected path from the top to the bottom of the image.

  Note that for any vertical seam, there only exists one seam pixel per row.

- Similarly, a horizontal seam is defined as $\mathbf{s^y} = \{s_j^y\}_{j=1}^m = \{(y(j), j)\}_{j=1}^m$, s.t $\forall j, |y(j) - y(j-1)| \leq 1$, where y is a mapping $y : [1, \ldots, m] \rightarrow [1, \ldots, n]$. As with the vertical seam, there exists one seam pixel per column along an 8-connected seam path.

For a vertical seam, the seam cost is defined as $E(\mathbf{s^x}) = E(\mathbf{I_{s^x}}) = \sum_{i=1}^n e(\mathbf{I}(s_i))$, and for horizontal seam, it is $E(\mathbf{s^y}) = E(\mathbf{I_{s^y}}) = \sum_{j=1}^m e(\mathbf{I}(s_j))$. We seek to identify the optimal seam, $\mathbf{s}^*$, that minimizes the seam cost, i.e. $\mathbf{s}^* = \min_s E(\mathbf{s})$

To quote Avidan & Shamir:
The optimal seam can be found using dynamic programming. The first step is to traverse

the image from the second row to the last row and compute the cumulative minimum energy **M** for all possible connected seams for each entry (i, j):

$$M_x(i,j) = e(i,j) + min(M_x(i-1,j-1), M_x(i-1,j), M_x(i-1,j+1))$$

$$M_y(i,j) = e(i,j) + min(M_y(i-1,j-1), M_x(i,j-1), M_x(i+1,j-1))$$

At the end of this process, the minimum value of the last row in $\mathbf{M_x}$ will indicate the end of the minimal connected vertical seam. Note that you have to record a backtrack table along the way. Hence, in the second step we backtrack from this minimum entry on $\mathbf{M_x}$ to find the path of the optimal seam. Also, in order to maintain consistency, if the minimum value is found at multiple indices, choose the smaller index. The definition of $\mathbf{M_y}$ for horizontal seams is similar.

2. Removing a Seam

For this task, you will fill in the two functions, `rmVerSeam.m` and `rmHorSeam.m`, which remove vertical and horizontal seams respectively. This task should be fairly simple. You should identify the pixel from $\mathbf{M_x}$ or $\mathbf{M_y}$ from which you should begin backtracking in order to identify pixels for removal (**rmIdx**), and remove those pixels from the input image. You will receive two inputs to each function, the corresponding cumulative minimum energy map, and the image. Utilizing these, you should output an image with one (appropriate) seam removed.

`rmVerSeam.m` has the following structure:

```
[Iy, E] = rmHorSeam(I, My, Tby)
```

(INPUT) `I`: $n \times m \times 3$ matrix representing the input image.
(INPUT) `My`: $n \times m$ matrix representing the cumulative minimum energy map along horizontal direction.
(INPUT) `Tby`: $n \times m$ matrix representing the backtrack table along horizontal direction.
(OUTPUT) `Iy`: $(n-1) \times m \times 3$ matrix representing the image with the row removed.

(OUTPUT) `E`: the cost of seam removal.

3. Discrete Image Resizing

Now that you're able to handle finding seams of minimum energy, and seam removal, we shall now tackle resizing images when it may be required to remove more than one seam,

sequentially and potentially along different directions. When resizing an image from size $n \times m$ to $n' \times m'$ (we will assume $n' < n$ and $m' < m$), the sequence of removing vertical and/or horizontal seams is important.

For this task, fill in the function `carv.m`, making use of the method described below. Utilizing recursive calls of the functions completed in the previous 2 tasks, complete this function to output the resized image.

```
[Ic, T] = carv(I, nr, nc)
```

(INPUT) `I`: $n \times m \times 3$ matrix representing the input image.
(INPUT) `nr`: the numbers of rows to be removed from the image.
(INPUT) `nc`: the numbers of columns to be removed from the image.
(OUTPUT) `Ic`: $(n - nr) \times (m - nc) \times 3$ matrix representing the carved image.
(OUTPUT) `T`: $(nr + 1) \times (nc + 1)$ matrix representing the transport map.

The following method is derived directly from the works of Avidan & Shamir:

We define the search for the optimal order as an optimization of the following objective function:

$$\min_{\mathbf{s^x},\mathbf{s^y},\alpha} \sum_{i=1}^{k} E(\alpha_i \mathbf{s_i^x} + (1 - \alpha_i)\mathbf{s_i^y})$$

where $k = r_t + c_t, r_t = (m - m'), c_t = (n - n')$ and α_i is used as a parameter that determine if at step i we remove a horizontal or vertical seam. $\alpha_i \in \{0, 1\}, \sum_{i=1}^{k} \alpha_i = r_t, \sum_{i=1}^{k}(1 - \alpha_i) = c_t$

We find the optimal order using a transport map $\mathbf{T}$ that specifies, for each desired target image size $n' \times m'$, the cost of the optimal sequence of horizontal and vertical seam removal operations. That is, entry $T(r, c)$ holds the minimal cost needed to obtain an image of size $n - r \times m - c$. We compute $\mathbf{T}$ using dynamic programming. Starting at $T(0, 0) = 0$ we fill each entry (r, c) choosing the best of two options - either removing a horizontal seam from an image of size $n - r \times m - c + 1$ or removing a vertical seam from an image of size $n - r + 1 \times m - c$:

$$T(r, c) = min(T(r - 1, c) + E(\mathbf{s^x}(\mathbf{I_{n-r+1 \times m-c}})), T(r, c - 1) + E(\mathbf{s^y}(\mathbf{I_{n-r \times m-c+1}})))$$

where $\mathbf{I_{n-r \times m-c}}$ denotes an image of size $n - r \times m - c$, $E(\mathbf{s^x}(\mathbf{I}))$ and $E(\mathbf{s^y}(\mathbf{I}))$ are the cost of the respective seam removal operation. We can store a simple binary map which indicates which of the two options was chosen in each step of the dynamic programming. Choosing a left neighbor corresponds to a vertical seam removal while choosing the top neighbor corresponds to a horizontal seam removal. Given a target size $n' \times m'$ where $n' = n - r_t$ and $m' = m - c_t$, we backtrack from $T(r_t, c_t)$ to $T(0, 0)$ and apply the corresponding removal operations. Note that in Matlab, indexes begin by 1 instead of 0, so in

practice you will backtrack from $T(r_t + 1, c_t + 1)$ to $T(1, 1)$. Also, for the sake of consistency, $T(r, c) = T(r - 1, c) + E(\mathbf{s^x}(\mathbf{I_{n-r+1 \times m-c}}))$, if $T(r - 1, c) + E(\mathbf{s^x}(\mathbf{I_{n-r+1 \times m-c}})) = T(r, c - 1) + E(\mathbf{s^y}(\mathbf{I_{n-r \times m-c+1}}))$.

Further notes:

In addition, in order to calculate $E(\mathbf{s^x}(\mathbf{I}))$ and $E(\mathbf{s^y}(\mathbf{I}))$, you have to record a table for $\mathbf{I_{n-r \times m-c}}$. More specifically, **TI** is the trace table and should be of size $(n - r_t + 1) \times (m - c_t + 1)$. $\mathbf{TI}\{\mathbf{1}, \mathbf{1}\}$ is the source image, and $\mathbf{TI}\{\mathbf{r + 1}, \mathbf{c + 1}\}$ is the image $\mathbf{I_{n-r \times m-c}}$, which is the source image removed r rows and c columns.

Also, due to MATLAB's use of 1-indexing, note that all indices referenced with 0-indexing in the method discussed above will be +1 in both rows and columns (e.g. $T(0, 0) \rightarrow T_{matlab}(1, 1)$).

Task 2: Gradient Domain Blending

For this part of the project, you will be implementing gradient domain blending. This method was described in Patrick, Michel and Andrew's paper: "*Poisson Image Editing*". It discuss the **importing gradients** in section 3, which you are strongly encouraged to read prior to starting this part of the project.

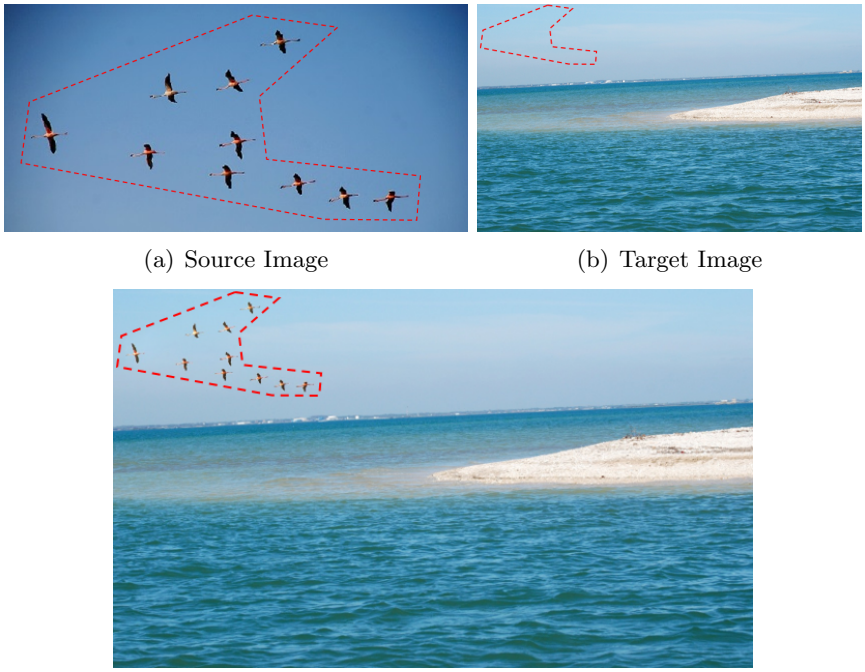
The goal is seamlessly blend two images together automatically given the blending region. As shown in the figure 1, the red line mark the blending region. To achieve this, you need to fill in 6 functions: `seamlessCloningPoisson.m`, `maskImage.m`, `getIndexes.m`, `getCoefMatrix.m`, `getSolutionVect.m` and `reconstructImg.m`.

Let's first define the image we're changing as the **target image**, the image we're cutting out and pasting as the **source image**, the pixels in target image that will be blended with source image as the **replacement pixels**.

The key idea of the gradient domain blending is to apply the gradient of the source image to the target image's replacement pixels, but keep other pixels. For continua image function, it can summarize as the following equation:

$$\min_f \iint_\Omega |\nabla f - \nabla g|^2, \quad \text{with} \quad f|_{\partial\Omega} = f^*|_{\partial\Omega}, \tag{1}$$

where f is the blending image function, f^* is the target image function, g is the source image function, Ω is the blending region, and $\partial\Omega$ is the boundary of blending region.



(a) Source Image (b) Target Image

(c) Result

Figure 1: Gradient Domain Blending

In the discrete pixel grid, the equation is as follows:

$$\min_{f|_\Omega} \sum_{p\in\Omega} \left(\left(|N_p| f_p - \sum_{q\in N_p} f_q \right) - \left(|N_p| g_p - \sum_{q\in N_p} g_q \right) \right)^2, \quad \text{with} \quad f|_{\partial\Omega} = f^*|_{\partial\Omega}, \tag{2}$$

where f is the blending image, f^* is the target image, g is the source image, N_p are the neighbouring pixels of pixel p, Ω are the replacement pixels, and $\partial\Omega$ are the boundary pixel of blending region.

If we convert the eq.2 to a linear system. For each $p \in \Omega$, we have

$$|N_p| f_p - \sum_{q\in N_p} f_q = |N_p| g_p - \sum_{q\in N_p} g_q, \tag{3}$$

Our task is to solve all f_p from the set of linear equations. If we form all f_p as a vector x. The set of eq.3 can be convert to a linear system $Ax = b$. Its solution would be the most agreeable solution of eq.2. Notice that, not all f_q is unknown parameter. Because it's possible that $q \in N_p$, but also $q \in \partial\Omega$. In this case, the $f_q = f_q^*$, which means it's known parameter. So you need move it to the right side of equation.

1. Align Images and Create Mask

Firstly, you need to align the source and target image. Use image editor to adjust the source image size and position, make sure that the region of the target image you want to replace is well aligned with the source image. Then, save the resized source image and its top left corner in the target image coordinate as the offset. For the following steps, the source image refers to the resized source image.

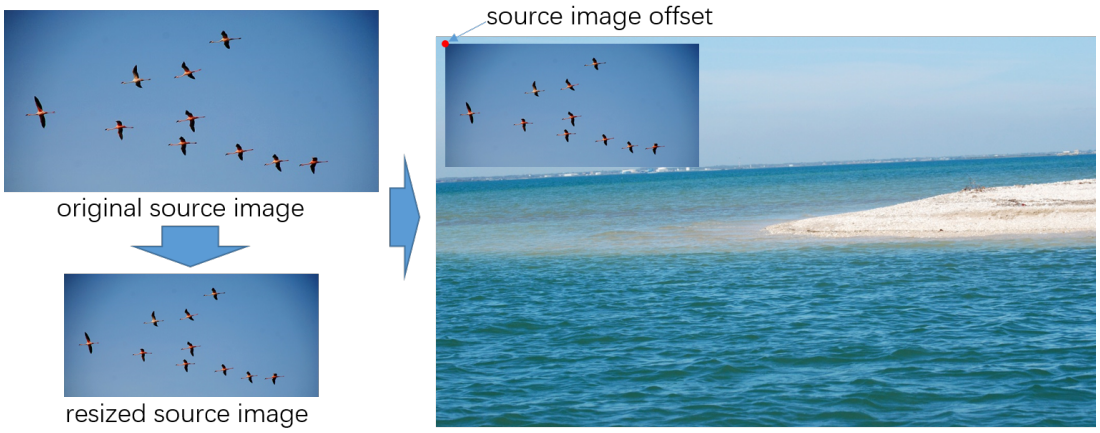


Figure 2: Align Image

Complete the following function to create the mask image - the logical matrix representing which pixels you want to replace in the source image. 1 means we are using the pixel, 0 means that the pixel will not be used. We recommend using Matlab's functions `imfreehand` and `createMask`.

```
[mask] = maskImage(img)
```

(INPUT) `img`: $h \times w \times 3$ matrix representing the source image.
(OUTPUT) `mask`: $h \times w$ logical matrix representing the replacement region.

2. Index Pixel

The replacement pixels' intensity are solved by the linear system $Ax = b$. But, not all the pixels in target image need to be computed. Only the pixels mask as 1 in the mask image will be used to blend. In order to reduce the number of calculations, you need to index the replacement pixels so that each element in x represents one replacement pixel. As shown in figure 3, the yellow locations are the replacement pixels (indexed from left to right).

Implement the following funtion:

`[indexes] = getIndexes(mask, targetH, targetW, offsetX, offsetY)`

(INPUT) `mask`: $h \times w$ logical matrix representing the replacement region.
(INPUT) `targetH`: target image height, h'.
(INPUT) `targetW`: target image width, w'.
(INPUT) `offsetX`: the x axis offset of source image regard of target image.
(INPUT) `offsetY`: the y axis offset of source image regard of target image.
(OUTPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel. 0 means not a replacement pixel.

Notice that the offset are regard of target image, which means the correspondence of the source image coordinate (x, y) in target image is $(x + offsetX, y + offsetY)$

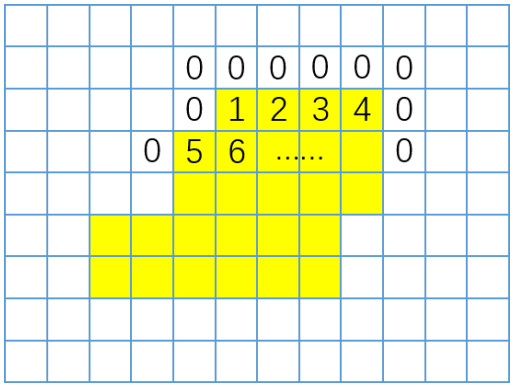


Figure 3: Index Pixels

3. Compute Coefficient Matrix

As we described in the past section, the replacement pixel intensities are solved using eq.3. In the function `getCoefMatrix`, you need to generate the coefficient matrix A in the linear system $Ax = b$. Notices that the coefficient matrix size is N by N, where N is the number of replacement pixels. To reduce the memory of this matrix, you need to use a sparse matrix. Check the Matlab doc (https://www.mathworks.com/help/matlab/sparse-matrices.html) for more details.

`[coefM] = getCoefMatrix(indexes)`

(INPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel.
(OUTPUT) `coefM`: $n \times n$ sparse matrix representing the coefficient matrix. n is the number of replacement pixels.

4. Compute Solution Vector

Implement the following function to generate the solution vector b in the linear system $Ax = b$.

`[solVector] = getSolutionVect(indexes, source, target, offsetX, offsetY)`

(INPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel.
(INPUT) `source`: $h \times w$ matrix representing one channel of source image.
(INPUT) `target`: $h' \times w'$ matrix representing one channel of target image.
(INPUT) `offsetX`: the x axis offset of source image regard of target image.
(INPUT) `offsetY`: the y axis offset of source image regard of target image.
(OUTPUT) `solVector`: $1 \times n$ vector representing the solution vector.

5. Reconstruct Image

After you solve the linear system $Ax = b$ and compute the vector x, you need to change the solution to the replacement pixels' intensity and reconstruct the blending image.

Implement the function:

`[resultImg] = reconstructImg(indexes, red, green, blue, targetImg)`

(INPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel.
(INPUT) `red`: $1 \times n$ vector representing the red channel intensity of replacement pixel.
(INPUT) `green`: $1 \times n$ vector representing the green channel intensity of replacement pixel.
(INPUT) `blue`: $1 \times n$ vector representing the blue channel intensity of replacement pixel.
(INPUT) `targetImg`: $h' \times w' \times 3$ matrix representing target image.
(OUTPUT) `resultImg`: $h' \times w' \times 3$ matrix representing blending image.

6. Wrapper Function

After you complete all the tool functions, you will need to implement the wrapper function and the demo script. In the function `seamlessCloningPoisson.m`, call `getIndexes.m`, `getCoefMatrix.m`, `getSolutionVect.m` and `reconstructImg.m` in this function, and also solve the linear systems. The Matlab function `mldivide` is recommended. You can check the Matlab doc (https://www.mathworks.com/help/matlab/ref/mldivide.html) for more details.

Implement the function:

`[resultImg] = seamlessCloningPoisson(sourceImg, targetImg, mask, offsetX, offsetY)`

(INPUT) `sourceImg`: $h \times w \times 3$ matrix representing source image.
(INPUT) `targetImg`: $h' \times w' \times 3$ matrix representing target image.
(INPUT) `mask`: $h \times w$ logical matrix representing the replacement region.
(INPUT) `offsetX`: the x axis offset of source image regard of target image.
(INPUT) `offsetY`: the y axis offset of source image regard of target image.
(OUTPUT) `resultImg`: $h' \times w' \times 3$ matrix representing blending image.

Finally, write a script to generate your blended image using `seamlessCloningPoisson.m` and `maskImage.m`.

FAQ

Q: For part 1 task 3 the instructions and the paper say that once the DP table T has been computed one then backtracks from T(r,c) to T(0,0). First, to originally compute T the algorithm must compute TI which has I with various seems removed which means TI(r,c) contains the image we are trying to compute which means no more computation is required. Also, intuitively, backtracking form T(r,c) would correspond to starting with the image we are trying to compute and adding seams back in to get T(0,0) which is the input image. So even if you ignore the fact that the target image has already been computed wouldn't you want to start with T(0,0) and "forward-track" to the desired image?

A: Sorry for the confusion. There's an issue about this algorithm. The key of this problem is that whether or not you re-compute the energy map after each seam removal. (The paper doesn't explain this explicitly.)

If you recompute it, which is the case in our project requirement, then you don't have to backtrack (because the target image has been already stored in TI). Or you could backtrack to get the order of horizontal and vertical removals.

In another case, you don't re-compute the energy map after each seam removal. In this situation, you keep the energy map only (say, TE). Therefore, you have to backtrack to get the order of horizontal and vertical removals, and do it on the input image accordingly.

Q: In function rmHorSeam, we need to output the cost E, I want to confirm that E is the summation of e rather than M, right?

A: Yes, it is the cumulative energy along the optimal seam.

Q: Question about coloring the seams I am wondering how can we color the seams in the original image?

A: You could use rmIdx together with the original image to color the seam. For example, suppose I(i, j) is the pixel to remove, let I(i, j, 1) = I(i, j, 1) / 2+ 128. (color with red)

Q: In rmVerSeam.m and rmHorSeam.m, what if we have multiple same minimum cumulative energy values?

A: This is specified on Task 1: "Also, in order to maintain consistency, whenever there are more than one minimum values, choose the one with smaller index."