Homework #3: Image Resizing using Seam Carving

Assigned: Thursday, September 29
Due: Tuesday, October 11

In this assignment you are to implement and evaluate an algorithm for resizing an image that uses image content to do this with (hopefully) minimal noticeable distortions. The method is called **Seam Carving**. It is implemented in Photoshop as a feature called "content-aware scaling." First, read the paper "Seam Carving for Content-Aware Image Resizing" by S. Avidan and A. Shamir, with emphasis on Section 3. Write a `main.m` script file that reads a given color image file into MATLAB using `imread`, converts it to double so that pixels have values in the range [0..1], calls functions to implement seam carving, and then creates an output RGB image as a `.jpg` image file. Put all code, images, and answers to the questions given below in a *single* zip file called `<NetID username>-HW3.zip` and submit this one file to Moodle. Your code should implement the following steps:

1. **Compute the energy function**
   Implement a function `E = imenergy(I)` (in a file called `imenergy.m`) that computes the energy image `E` from an RGB image `I` as defined in Equation 1 in the paper. You can use the MATLAB function `Gmag = imgradient(J)` to compute the gradient magnitude matrix (of type `double`) of a grayscale image `J`. Because `I` is a color image, first convert `I` to double using `im2double`, and then to grayscale using `rgb2gray` before computing the gradient magnitude. `E` should be a double 2D array of the same size as `I`, with a floating-point value at each pixel representing the gradient magnitude.

   Your energy image should look like a gradient image. That is, the energy image should be very bright where the original image goes from very bright to very dark (or very dark to very bright) within a few pixels. And the energy image should be very dark where the original image has a region of nearly constant intensity. For example, see http://homepages.inf.ed.ac.uk/rbf/CVDICT/CVFIG3/img54.png (left is original image, right is gradient image). The edges of the patches on the dog are very bright because that's where the dog's fur goes from black to white or white to black. But within a white patch or within a black patch of the original image, the gradient image is dark. See the Wikipedia page on "seam carving" for another example.

2. **Compute the optimal horizontal seam**
   Implement a function `S = horizontal_seam(I)` that takes an image and finds the (one) optimal horizontal seam, returning a vector of length equal to the number of columns in `I` such that each entry in vector `S` is an integer-valued row number indicating which pixel in that column should be removed. For example, `S(10)=37` means that in the 10th column the pixel in row 37 is to be removed. The optimal seam can be found using dynamic programming to compute, left to right, the cumulative minimum energy array, `M`, as described in the paper. Starting from the minimum value in the rightmost column of `M`, `S` is computed during the backward pass from the rightmost column to the leftmost column of `M`. Adjacent entries in `S` must be at most one row apart so that the seam found is a path of 8-adjacent pixel coordinates. (Note: you can plot the points in `S` on top of image `I` using `imshow`, `hold on`, and `plot`. Plotting on top of the image will be used in creating the image `lastname.3.jpg` described below in the Experiments section.) See the Wikipedia

page on "seam carving" to see how dynamic programming works here (though the example shown is for a vertical seam). NOTE: It is hard to vectorize the dynamic programming algorithm in MATLAB so you are free to use loops.

To find the optimum seam, you need to keep track of two pieces of information at each pixel location <*r*, *c*>: 1) the cumulative energy of the best seam "so far" that includes the location <*r*, *c*>, and 2) a pointer to the location prior to <*r*, *c*>. Call the first piece of information *cumenergies* and the second one *pointers.* To store this information, you have several data structures to choose from:

1. **An M x N cell array, with a 1 x 2 matrix in each cell.** The two elements in each 1 x 2 matrix are *cumenergy* and *pointer*. A `cell array` has a grid representation like regular MATLAB matrices, except that each cell can store an arbitrary data structure. This is unlike regular MATLAB matrices where if one element is, say, `uint8`, all other elements in the matrix also have to be `uint8`. Because of MATLAB's somewhat clunky syntax rules, I wouldn't recommend this approach.
2. **An M x N x 2 matrix** where the first channel stores *cumenergies* and the second channel stores *pointers*. This choice is pretty natural except that you have to remember which channel stores what information, and that makes the code a little less readable.
3. **An M x N matrix for *cumenergies* and an M x N matrix for *pointers*.** This is probably the most readable option. I'd recommend this, but decide for yourself.

Note that to compute the optimal seam you need to know the *coordinates* of the min value rather than the min value itself. See Matlab documentation for the `min` function to help you decide how to implement this.

Another implementation detail is how to compute the min when you're at the top row or bottom row. For example, consider the following 3 x 4 image:

```
x   a   d   x
x   b   e   x
x   c   f   x
```

Say we're filling the grid from right to left. `b` depends on all three neighboring pixels, `d`, `e` and `f`, but at the top and bottom rows, you only have two neighboring pixels, e.g., `a` only depends on `d` and `e`, while `c` only depends on `e` and `f`. How do you code this? You have several options. Here's two:

1. **If-else statements.** This is probably the most obvious choice. Basically "if I'm at the top row, ignore the (non-existent) row above me; if I'm at the bottom row, ignore the (non-existent) row below me." It's a reasonable choice, but not super elegant.
2. **Pad *cumenergies* with infinities.** Before filling *cumenergies*, you add two rows of pixels all with value infinity (`Inf`), one above the top row and one below the bottom row, literally by just typing `Inf` in MATLAB, as shown below. That way you can always take the min of all three neighboring pixels without worrying about boundary cases (and since all pixels beside the `Inf` you've added are less than `Inf`, the min will never be `Inf`). But you need to remember that row #1 in the original image is now row #2 in *cumenergies*, etc.

```
Inf Inf Inf Inf
  x   a   d   x
  x   b   e   x
  x   c   f   x
Inf Inf Inf Inf
```

3. Remove 1 horizontal seam
Implement a function `J = remove_horizontal_seam(I, S)` that removes *one* horizontal seam from image `I` (of size *m* x *n*) to produce new image `J` that has size (*m*-1) x *n*.

4. Resize
Implement a function `J = shrnk(I, num_rows_removed, num_cols_removed)` that takes an input color image `I` and computes an output color image `J` that has `num_rows_removed` fewer rows than `I`, and `num_cols_removed` fewer columns than `I`. This should find one seam, remove it, find the next seam, remove it, etc. Implement this function using only the horizontal seam detector and remover by (a) removing all horizontal seams, (b) rotating the image 90° using `J = permute(I, [2 1 3])` (do not use `transpose`), (c) removing all vertical seams, and (d) un-rotating the image (using `J = permute(I, [2 1 3])`). Hence you do not need to implement a separate vertical seam removal function. Note: While your code's runtime will not affect your grade, the runtime for computing `shrnk` can be long; use built-in Matlab functions and the `:` operator instead of loops whenever possible to speed up your code.

Experiments

1. Using the test image union-terrace.jpg , run your `shrnk` function with the following values and create three (3) images called `lastname.1a.jpg` , `lastname.1b.jpg` and `lastname.1c.jpg` showing the results for:
   a. `num_rows_removed` = 0, `num_cols_removed` = 100
   b. `num_rows_removed` = 100, `num_cols_removed` = 0
   c. `num_rows_removed` = 100, `num_cols_removed` = 100

2. Create an image of the energy function output called `lastname.2a.jpg` and an image called `lastname.2b.jpg` of the cumulative minimum energy array `M` created *before* removing the first horizontal seam (your cumulative minimum energy array should be computed left to right across the image, with each pixel containing the cost of the optimal path from that pixel back to some pixel in the first column). That is, create the array `M` as described in the lecture notes, based on the Energy image you created from the input image. Be sure that `M` is the cumulative minimum energy computed from ***left to right***, starting from the first column and ending at the last column ***before removing the first horizontal seam***. So, the array `M` will be a 2D array the same size as the input image, with values at every pixel; values will increase from left to right across the image, so viewing the image you create should be darkest on the left and get lighter as you move right across the image. And don't put `M` on top of the original image; the image should just show the values in the cumulative minimum energy array. Use `imagesc` and `saveas` to create these images. Explain why these images look as they do. Put your comments in a file called `README.txt`

3. Create an image called `lastname.3.jpg` showing the original image together with the first selected horizontal seam and the first selected vertical seam overlaid. Explain why these are reasonable optimal seams. Put your explanation into the same `README.txt` file.

4. Find on the web or take a photo to use as an input image (that is not too big) and name it `lastname.4a.jpg` where this image shows an ***interesting successful result*** of shrinking (either horizontally, vertically, or both) an image by some non-trivial amount. Save your

result image as `lastname.4b.jpg` Hand in both images and describe why it seems to work well. Put your description in the same `README.txt` file as above.

5. Find on the web or take a photo to use as an input image (that is not too big) and name it `lastname.5a.jpg` where this image shows an ***interesting poor result*** of shrinking a different image (either horizontally, vertically, or both) by some non-trivial amount. Save your result image with the name `lastname.5b.jpg` Hand in both images and describe why it seems to work poorly. Put your description in the same `README.txt` file.

Extra Credit (up to 10%)

Implement one or more of the following or your own extension of any kind. Show and describe results. Hand in a separate `ExtraCredit` folder containing your code and images.

1. Write a function `J = expnd(I, num_rows_expanded, num_cols_expanded)` that expands the image by *adding* horizontal and vertical seams to create an output image that has `num_rows_expanded` more rows and `num_cols_expanded` more columns.

2. Implement one or more alternate energy functions and compare them experimentally.