Programming Assignment 3
Deadline: Mar. 2nd

1 Goal

The goal of this project is to perform image blending. Basically, given a source image and a mask, our goal is to seamlessy blend the masked areas of the source image into a target photograph. We explore two ways of performing this process. In the first task you will become familiar with pyramid blending. Then you'll implement Poisson blending in the second task.

2 Starter Code

Starter code (in both MATLAB and Python) along with the images can be downloaded from here. Note that, expected results for a few images are included in the "Results" folder.

3 Task 1

Here, you implement the pyramid blending approach that was discussed in the class. Given a source, target, and a mask, you have to follow the following steps to generate the blended image:

- Compute Laplacian pyramids L_S and L_T from the source and target images.

- Compute a Gaussian pyramid G_M from the mask.

- Use the the Gaussian pyramid to combine the source and target Laplacian pyramids as follows:

$$L_C(l) = G_M(l) \times L_S(l) + (1 - G_M(l)) \times L_T(l), \tag{1}$$

  where l is the layer index.

- Collapse the blended pyramid L_C to reconsruct the final blended image

You can test your implementation on image 1. You should also create one example of your own where pyramid blending produces a nice blended image. You can use the `GetMask` function to obtain a mask on your images.

3.1 Hints

- You can use MATLAB's `imresize` to build the Gaussian pyramid, but cannot use higher level functions like `impyramid`.

- The last level of your pyramid is going to be $2^{(n-1)}$ (n is the number of levels) times smaller than the original resolution in each dimension. Therefore, you have to make sure the size of your original image is divisible by this number. You can ensure that by padding the original image with, for example, zeros.

4 Task 2

Here, you are required to implement gradient domain image blending, described in Perez, et al.'s paper. Your implementation should run properly on all the examples (which means you need to handle the boundary conditions – see Sec. 4.3), however, you will only be able to produce high-quality results on images 2 to 6 (image 1 works better with pyramid blending, and images 7 and 8 require Poisson blending with mixing

gradients – see Extra Credit). To get the full point, you need to create one composite of your own. You can use the `GetMask` function to obtain a mask on your images.

The implementation of gradiant domain blending is not complicated, but it is easy to make a mistake. Below we provide a detailed description of the process for a simple 1D case and then extend it to 2D.

4.1 Simple 1D Example

Here, we are going to discuss a simple 1D case to get you started with implementing the 2D case.

4.1.1 Hole-Filling

Let's start with a simple case where instead of copying in new gradients we only want to fill in a missing region of an image and keep the gradients as smooth (close to zero) as possible. To simplify things further, let's start with a one dimensional signal instead of a two dimensional image.

Here is our signal `t` (see Fig. 1) and a mask `M` specifying which "pixels" are missing.

```
t = [5 4 0 0 0 0 2 4];
M = [0 0 1 1 1 1 0 0];
M = logical(M); % converting the mask from double to binary
```

We can formulate our objective as a least squares problem. Given the intensity values of `t`, we want to solve for new intensity values `v` under the mask `M` such that:

$$v = \arg\min_{v} \sum_{<i,j>\in M} (v_i - v_j)^2, \quad \text{with } v_i = t_i \text{ for all } i \in \ \sim M. \tag{2}$$

Here, i is a coordinate (1d or 2d) for each pixel under mask M. Each j is a neighbor of i. The summation guides the gradient (the local pixel differences) in all directions to be close to 0. Minimizing this equation could be called a Poisson fill.

For this example let's define neighborhood to be the pixel to your left.[1] The optimal pixel values can be obtained by setting the derivative of the above equation equal to zero. This gives us two equations at each pixel as follows:

$$\text{for all } i \in M, \quad v_i - v_{i-1} = 0 \quad \text{and} \quad v_{i+1} - v_i = 0, \tag{3}$$

where $v_j = t_j$ for all $j \in\sim M$. This produces the following system of equations:

```
v(1) - t(2) = 0; %left border
v(2) - v(1) = 0;
v(3) - v(2) = 0;
v(4) - v(3) = 0;
t(7) - v(4) = 0; %right border
```

Note that the coordinates do not directly correspond between `v` and `t`. For example, `v(1)`, the first unknown pixel, sits on top of `t(3)`. You could formulate it differently if you choose. Plugging in known values of `t` we get:

```
v(1) -      4 = 0;
v(2) - v(1) = 0;
v(3) - v(2) = 0;
v(4) - v(3) = 0;
   2 - v(4) = 0;
```

---

[1]You could define neighborhood to be all surrounding pixels. In 2d, you would at least need to consider vertical and horizontal neighbors.
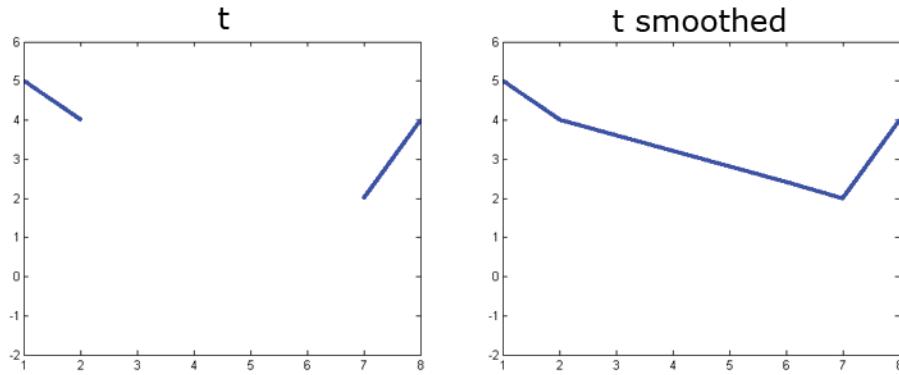
Figure 1: On the left, we show the target signal `t`. The goal is to fill in the samples 3 to 6 smoothly using the Poisson equation. On the right, we show the filled in result `t_smoothed`.

Now let's convert this to matrix form and have Matlab solve it for us

```
A = [ 1   0   0   0; ...
     -1   1   0   0; ...
      0  -1   1   0; ...
      0   0  -1   1; ...
      0   0   0  -1];

b = [4; 0; 0; 0; -2];

v = A\b; % "help mldivide" describes the '\' operator.
t_smoothed = zeros(size(t));
t_smoothed(~M) = t(~M);
t_smoothed( M) = v;
```

As it turns out, in the 1d case, the Poisson fill is simply a linear interpolation between the boundary values. But in 2d the Poisson fill exhibits more complexity.

4.1.2 Blending Source

Now instead of just doing a fill, let's try to seamlessly blend content from one 1d signal into another. We'll fill the missing values in `t` using the corresponding values in `s` (see Fig. 2):

```
s = [8 6 7 2 4 5 7 8];
```

Now our objective changes. Instead of trying to minimize the gradients, we want the gradients to match another set of gradients (those in `s`). Therefore, our set of equations changes as follows:

```
v(1) - t(2) = s(3) - s(2);
v(2) - v(1) = s(4) - s(3);
v(3) - v(2) = s(5) - s(4);
v(4) - v(3) = s(6) - s(5);
t(7) - v(4) = s(7) - s(6);
```

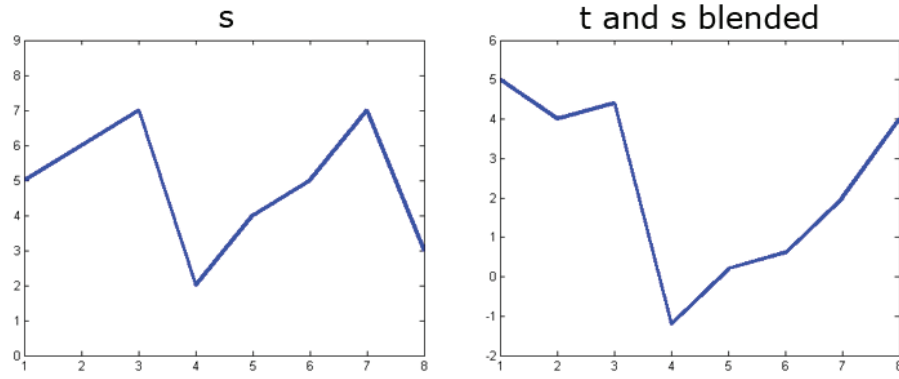After plugging in known values from `t` and `s` this becomes:

3

Figure 2: On the left, we show the source signal `s`. The goal is to blend this source signal with the target from Fig. 1. On the right, we show the blended result `t_and_s_blended`.

```
v(1) -      4 =    1;
v(2) - v(1) =   -5;
v(3) - v(2) =    2;
v(4) - v(3) =    1;
    2 - v(4) =    2;
```

Finally, in matrix form for MATLAB

```
A = [ 1   0   0   0;  ...
     -1   1   0   0;  ...
      0  -1   1   0;  ...
      0   0  -1   1;  ...
      0   0   0  -1];

b = [5; -5; 2; 1; 0];

v = A\b;
t_and_s_blended = zeros(size(t));
t_and_s_blended(~M) = t(~M);
t_and_s_blended( M) = v;
```

Notice that in our quest to preserve gradients without regard for intensity we might have gone too far. Our signal now has negative values. The same thing can happen in the image domain, so you'll want to watch for that and at the very least clamp values back to the valid range.

4.2 Extension to 2D

When working with images, the basic idea is the same as above, except that each pixel has at least two neighbors (left and top) and possibly four neighbors. Either formulation will work. For example, in a 2d image using a 4-connected neighborhood, our equations above imply that for a single pixel in `v`, at coordinate `(i,j)` which is fully under the mask you would have the following equations:

```
v(i,j) - v(i-1, j) = s(i,j) - s(i-1, j);
v(i,j) - v(i+1, j) = s(i,j) - s(i+1, j);
v(i,j) - v(i, j-1) = s(i,j) - s(i, j-1);
v(i,j) - v(i, j+1) = s(i,j) - s(i, j+1);
```

4

In this case we have many equations for each unknown. It may be simpler to combine these equations such that there is one equation for each pixel, as this can make the mapping between rows in your matrix `A` and pixels in your images easier. Adding the four equations above we get:

```
4*v(i,j) - v(i-1, j) - v(i+1, j) - v(i, j-1) - v(i, j+1) =
4*s(i,j) - s(i-1, j) - s(i+1, j) - s(i, j-1) - s(i, j+1);
```

where everything on the right hand side is known. This formulation is similar to equation 8 in Pérez, et al.'s paper. You can read the paper, especially the "Discrete Poisson Solver", if you want more guidance.

4.3 Hints

These hints are specific to the Poisson blending (Task 2) part of this assignment.

- For color images, process each color channel independently (hint: matrix `A` will not change, so don't go through the computational expense of rebuilding it for each color channel).

- The linear system of equations (and thus the matrix `A`) becomes enormous. But `A` is also very sparse because each equation only relates a pixel to some number of its immediate neighbors.

- `A` needs at least as many rows and columns as there are pixels in the masked region (or more, depending on how you have set up your system of equations. You may have several equations for each pixel, or you may have equations for already known pixels just for implementation convenience). If the mask covers 100,000 pixels, this implies a matrix with at least 100,000,000,000 entries. Don't try that. Instead, use the sparse command in MATLAB to build sparse matrices for which all undefined elements are assumed to be 0. A naive implementation will run slowly because indexing into a sparse matrix requires traversing a linked list.

- You will need to keep track of the relationship between coordinates in matrix `A` and image coordinates. `sub2ind` and `ind2sub` might be helpful (although they are slow, so you might want to do the transformation yourself). You might need a dedicated data structure to keep track of the mapping between rows and columns of `A` and pixels in `s` and `t`.

- Not every pixel has left, right, top, and bottom neighbors. Handling these boundary conditions might get slightly messy. You can start by assuming that all masked pixels have 4 neighbors (this is true for the majority of cases), but your code should properly handle the boundary conditions (images 1 and 8).

- Your algorithm can be made significantly faster by finding all the `A` matrix values and coordinates ahead of time and then constructing the sparse matrix (using `sparse(i,j,s,m,n,nzmax)` in MATLAB) in one operation. This should speed up blending from minutes to seconds.

- Helpful MATLAB commands that may help you speed up your algorithm: `sparse`, `speye`, `find`, `sort`, `diff`, `cat`, and `spy`.

5 Extra Credit

Mixing Gradients – Mixing Gradients to allow transparent images or images with holes. Instead of trying to adhere to the gradients of the source image, at each masked pixel use the largest gradient available in either the source or target. You can test your implementation on images 7 and 8.

6 Write Up

Include all the results in your report. For each result, you should show the input images, as well as the naive blending along with your blended result. Describe how you implemented the assignment. Discuss any problem you faced when implementing the assignment or any decisions you had to make.

7 Graduate Credit

Graduate students have to do the extra credit.

8 Deliverables

Your entire project should be in a folder called "`firstname_lastname`". This folder should be zipped up and submitted through e-campus. Inside the folder, you should have the followings:

- A folder named "Code" containing all the codes for this assignment. Please include a README file to explain what each file does if you add any other files to the starter code.

- A report in the pdf format. **Make sure you write your name on top of the report. Also make sure the pdf file is under 5 MB.**

  Make sure you exclude all the results and original images from your submission.

9 Checklist

Make sure you can check all the items below before submitting your assignment. You will lose 5 points for each item that cannot be checked.

- ☐ The root folder is called "`firstname_lastname`". Note _ between first and last name.

- ☐ Inside the root folder, there is a folder called "Code" that contains your source code. Also make sure the report is in the root folder.

- ☐ The folders "Images" and "Results" are not included (you only submit your codes and a report).

- ☐ Name written on top of the report.

- ☐ The report is in pdf format and the file is under 5 MB.

10 Ruberic

Total credit: [150 points]

[40 points] - Implement pyramid blending
[10 points] - Include one example of your own
[50 points] - Implement Poisson blending
[10 points] - Implement the approach efficiently using sparse matrix
[10 points] - Handle the boundary condition
[10 points] - Include one example of your own
[20 points] - Write up

Extra credit: [10 points]

[10 points] - Mixing gradients

11 Acknowlegements

Poisson blending part of the project is derived from James Hays Computational Photography course with permission.