CSCE 689 - Computational Photography

Programming Assignment 2

Deadline: Feb. 22nd

1 Overview

This project explores gradient-domain processing, a simple technique with a broad set of applications including blending, tone-mapping, and non-photorealistic rendering. This specific project explores seamless image compositing via "Poisson blending".

The primary goal of this assignment is to seamlessly blend an object or texture from a source image into a target image. The simplest method would be to just copy and paste the pixels from one image directly into the other (and this is exactly what the starter code does). Unfortunately, this will create very noticeable seams, even if the backgrounds are similar. How can we get rid of these seams without doing too much perceptual damage to the source region?

The insight is that people are more sensitive to gradients than absolute image intensities. So we can set up the problem as finding values for the output pixels that maximally preserve the gradient of the source region without changing any of the background pixels. Note that we are making a deliberate decision here to ignore the overall intensity! We will add an object into an image by reintegrating from (modified) gradients and forgetting whatever absolute intensity it started at.

Starter code (in MATLAB) along with the images can be downloaded from here. You are required to implement gradient domain image blending, described in Perez, et al.'s paper. You need to create at least two composites of your own in addition to the 7 provided test cases. One of the examples should show the limitation of the approach. The "getmask.m" file in the starter code can be used to draw masks on your images. Note that, for test cases 6 and 7, you need to implement the mixing gradients idea to produce a high quality result.

2 Simple 1D Example

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

2.1 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:

$$v = \arg\min_{v} \sum_{<i,j>\in M} (v_i - v_j)^2, \quad \text{with } v_i = t_i \text{ for all } i \in\sim M. \tag{1}$$

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.

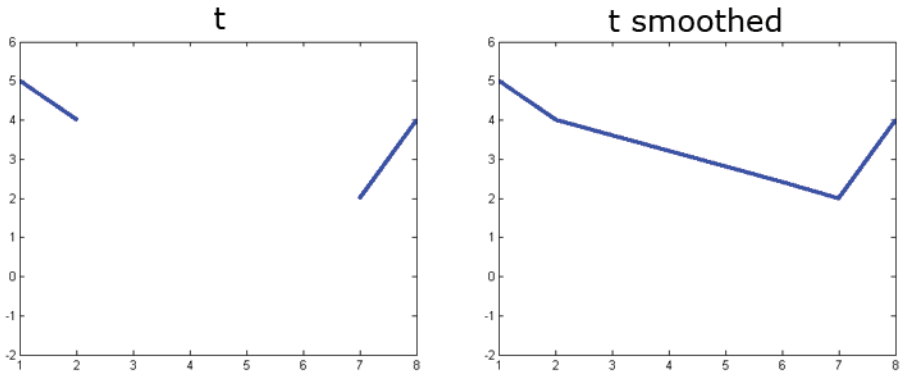

Figure 1: On the left, we show the target signal `t`. The goal is to fill in the samples 3 to 6 smoothly using the Poisson equation. On the right, we show the filled in result `t_smoothed`.

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:

$$\text{for all } i \in M, \quad v_i - v_{i-1} = 0 \quad \text{and} \quad v_{i+1} - v_i = 0, \tag{2}$$

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;
```

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];
```

[1] You could define neighborhood to be all surrounding pixels. In 2d, you would at least need to consider vertical and horizontal neighbors.

```
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.

2.2 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:

```
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;
```

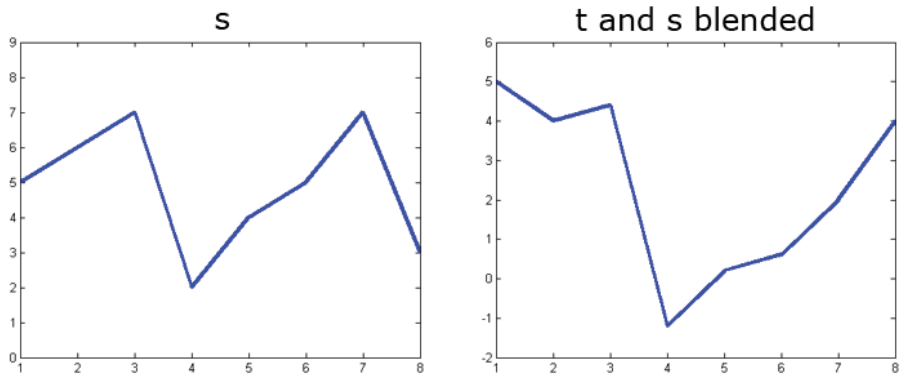

Figure 2: On the left, we show the source signal `s`. The goal is to blend this source signal with the target from Fig. 1. On the right, we show the blended result `t_and_s_blended`.

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. 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);
```

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.

3 Hints

- For color images, process each color channel independently (hint: matrix `A` won't 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've 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'll 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 intentionally true of the first 6 test cases), but the 7th test case will break this assumption.

- 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 in one operation. See `sparse(i,j,s,m,n,nzmax)`. 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`.

- PoissonBlend.m in the starter code contains some more suggestions.

4 Deliverables

Your project should be in a folder and submitted in zip format (called "`firstname_lastname.zip`") 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.

- A folder named "Results" containing the generated RGB images for each of the input glass plate images.

- A write up describing the algorithm used to implement the assignment. Please discuss any problem you faced when implementing the assignment or any decisions you had to make. For each result, you should show the input images, as well as the naive blending along with your blended result. In addition to the provided 7 images, include two test cases of your own. Please make sure that one them is a failure case and properly discuss it. *Make sure you write your name on top of the report.*

5 Acknowlegements

This project is derived from James Hays Computational Photography course with permission.