MIT EECS 6.815/6.865: Assignment 11:

Poisson Image Editing

Due Wednesday May 2 at 9pm

1 Summary

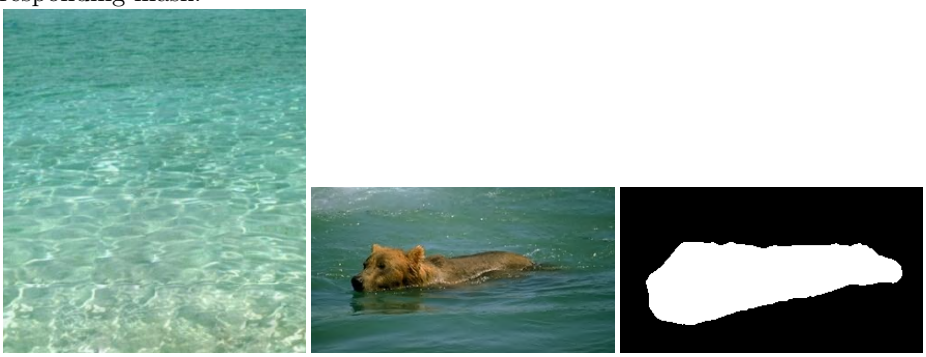
- dot product between two images
- Poisson solver with gradient descent
- Poisson solver with conjugate gradient
- Make your own montage

Don't be frightened by the amount of text and equations in this pdf. Your code itself should fairly short and simple, and it's not critical to understand all the derivations to eventually apply the final algorithms.

2 Compositing

Poisson image editing enables seamless compositing. The reference on the topic is Perez et al.'s article, "Poisson image editing," published at Siggraph 2003 and available at http://dl.acm.org/citation.cfm?id=882269

Given a background (target) and a new foreground (source), as well as a binary mask, we seek to replace the target region inside the mask by the content of the source. Below, we show a target (water), a source (bear), and the corresponding mask.



2.1 Naive compositing

Implement a function `naiveComposite(bg, fg, mask, y, x)` that simply copies the pixel values from an image `fg` into a target `bg` when `mask` is equal to 1. `fg`

is assumed to be smaller than bg and has the same size as `mask`. `y` and `x` specify where the upper left corner of `fg` goes in the source image `bg`.

You should be able to do it without a for loop, using multiplications between images and the mask or 1-mask, and using the slicing operator :. My result is below for `naiveComposite(bg, fg, mask, 50, 1)`

3 Poisson image editing

3.1 Math

Poisson equation The idea of Poisson image editing is to put values inside the masked region that seek to replicate the *gradient* of the source image, while respecting the value of the target image at the boundary of the mask. Since this is not perfectly possible, we seek to minimize the square of the difference between the gradient of the source and that of the result.

$$\min \int \int |\nabla f - \nabla s|^2$$

where f is the unknown output image, ∇ is the gradient operator, and s is the source.

For discrete images, the gradient can be computed using two convolutions by $[-1, 1]$ and $[-1, 1]^T$. If we encode all values of the image into a big vector x, since the gradient computation is linear, we can denote it with a matrix product Gx. Do not worry about the precise form of matrix G, we just need it for abstract derivations, and the final implementation will be simple in terms of image convolution. Our minimization then becomes

$$\min |Gx - Gs|^2$$

and we can express the above sum of squares as a dot product

$$\min (Gx - Gs)^T (Gx - Gs) = x^T (G^T G)x - 2x^T (G^T G)s + s^T (G^T G)s$$

If we define $A = G^T G$, we have a linear least square minimization

$$\min x^T Ax - 2x^T As + s^T As \tag{1}$$

and we will call the above energy $Q(x)$

Given that A is defined as $A = G^T G$, which is more or less the application of the gradient twice, it corresponds to the second derivative of the image, often called the *Laplacian* $\Delta = \frac{d^2}{dx^2} + \frac{d^2}{dy^2}$. In the discrete world of digital images, the Laplacian is the convolution by

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

We can minimize the above equation by setting its derivative with respect to the unknown image x as zero.

$$\frac{d}{dx}\left(x^T Ax - 2x^T As + s^T As\right) = 2Ax - 2As \tag{2}$$

which leads to the linear system

$$Ax = As \tag{3}$$

In conclusion, the problem of seamlessly pasting a source image into a region can be reduced to the solution of a big linear system $Ax = b$ where A is the convolution by the Laplacian operator and b is the Laplacian of the source image. In what follows, we will solve this system using gradient descent and conjugate gradient.

The beauty of this assignment is that we will never need to create the matrix A or represent our image as a 1D vector. We will keep the image representation we know and love, and all we need from A is the ability to apply it to a x that is an image. That is, we simply need to be able to convolve an image by the Laplacian kernel.

In what follows, we will expose the numerical solvers in terms of Matrix-vector notations $Ax = b$. But in your implementation, you will deal with images and convolution. For this, you first need to implement basic linear algebra operations for images.

3.2 Image dot product and convolution

The dot product between two vectors is the sum of the product of their coordinates. For example, $(x, y, z) \cdot (x', y', z') = xx' + yy' + zz'$ This is the same for images: take the sum of the pairwise products between all the values for all the pixels and channels of the two images.

Write a function `dotIm(im1, im2)` that returns the dot product between the two images. The output should be a single scalar, just like any decent dot product. Make sure you use bumpy array operations such as `sum` to make everything fast.

You also need to be able to apply the operator A to an image array. We have already done it for you in `a11Help.py` and implemented a function `Laplacian(im)`.

In what follows, each time you see a dot product, this will mean your function `dot`, and each time you see a multiplication by matrix A, it means using `Laplacian`. Don't be confused by the fact that we consider both the gradient of images and the gradient of a big quadratic optimization.

3.3 Gradient descent

We first implement a slow gradient descent solution. It is an iterative algorithm that refines an estimate x_i in the direction of steepest descent. We showed in Eq. 2 that the gradient of the quadratic minimization energy 1 is the corresponding linear equation $Ax_i - b$. The direction of steepest descent is the opposite, which we denote as

$$r_i = b - Ax_i$$

(the letter r comes from the fact that it is the *residual* of the equation, how much Ax_i is different from b)

We will update x_i by stepping in direction r_i by an amount α:

$$x_{i+1} = x_i + \alpha r_i$$

To compute the optimal alpha, we solve the minimization problem along the 1D line defined by r_i. That is, we want the gradient of the quadratic energy to be orthogonal to the line.

$$\begin{aligned}
\frac{d}{d\alpha}Q(x_{i+1}) &= \frac{d}{dx}Q \cdot \frac{dx_{i+1}}{d\alpha} && (4)\\
&= (b - Ax_{i+1}) \cdot r_i && (5)\\
&= (b - Ax_i - \alpha A r_i) \cdot r_i && (6)\\
&= (r_i - \alpha A x_i) \cdot r_i && (7)
\end{aligned}$$

and setting it to zero gives:

$$\alpha = \frac{r_i \cdot r_i}{r_i \cdot Ar_i} \quad (8)$$

Implementation Write a function `Poisson(bg, fg, mask, niter=200)` that implements gradient descent to solve for Poisson image compositing. `bg`, `fg`, and `mask` are assumed to have the same size. Your code will be a more or less direct implementation of the above math. In a nutshell, first compute the constant image b by applying the Laplacian operator to the image `fg`. Initialize x with a black image and iterate: compute the residual r by applying the Laplacian to your current estimate and subtract it from b. Then solve for alpha using the dot product you have implemented for images. Finally, update your estimate of x.

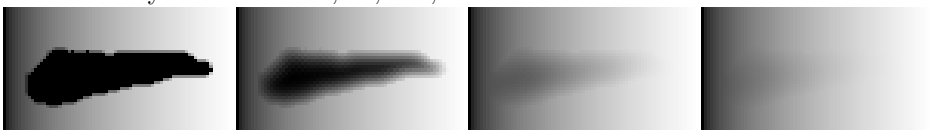
Besides replacing the matrix-vector operations by their image implementation, the other special thing you need is to make sure that pixels outside the

mask have values from `bg` and are not updated. For this, first initialize these pixel in x by copying those in `bg`. You can do this easily with array subtractions and multiplications. Next, you need to multiply your residual by the mask as soon as you compute it. This way, your updates will only consider pixels inside the mask.

We provide you with the wrapper function `PoissonComposite` which formats the data, and in particular makes sure that all the images passed to your `Poisson` function have the same size. The wrapper is itself called by the provided function `test`.

Test this function on the small inputs because it is slow to converge. We highly recommend the provided `testRamp` which seeks to paste a flat source into a background composed of a greyscale ramp.

Below are my results after 0, 10, 100, and 200 iterations



Note that the algorithm has not fully converged yet: the true result should the original ramp.

Turn in your results for 50 and 250 iterations for the ramp input.

3.4 Conjugate gradient

We can greatly speed up the convergence of gradient descent with minor modifications, using the *conjugate gradient* method. At a high level, a major problem with gradient descent is that it tends to have an erratic behavior and follow a zig-zag pattern. The conjugate gradient algorithm seeks to make an update as orthogonal as possible to the previous ones (for some special notion of orthogonality, which depends on the matrix A). Deriving it is a lot of work and we refer you to the excellent introduction by Jonathan Schewchuck *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, available at `http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf`

Instead of moving in the negative gradient direction r_i, the conjugate gradient uses a modified direction d_i, which is obtained through a process similar to Gram-Schmidt orthogonalization, where we subtract the part of r_{i+1} that is not A-orthogonal to the previous d_i. You do not need to worry about the mathematical justification, and can simply implement the recipe.

Initialize x_0 and

$$r_0 = d_0 = b - Ax_0$$

Then iterate:

$$\alpha = \frac{r_i \cdot r_i}{d_i \cdot Ad_i}$$
$$x_{i+1} = x_i + \alpha d_i$$
$$r_{i+1} = r_i - \alpha A d_i$$
$$\beta = \frac{r_{i+1} \cdot r_{i+1}}{r_i \cdot r_i}$$
$$d_{i+1} = r_{i+1} + \beta d_i$$

Implementation Write a function `PoissonCG(bg, fg, mask, niter)` that implements the above conjugate gradient algorithm to solve Poisson image editing. The specifications are the same as for gradient descent. Similarly, x_0, r_i and d_i need to be masked appropriately.

Here is my result when using the log domain version of compositing:

Note how much faster the conjugate method converges.

3.5 Have fun

After all this math, time to have fun! Create your own composite. Turn in your three inputs and the result.

4 Extra credits

Implements some of the extensions described in Perez et al.'s article.

5 Submission

Turn in your images, python files, and make sure all your functions can be called from a module `a11.py`. Put everything into a zip file and upload to Stellar.

Include a `README.txt` containing the answer to the following questions:

- How long did the assignment take?

- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented
- Collaboration acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?

Images:

- result of gradient descent after 50 and 250 iterations for the ramp.
- bear composite
- your composite (4 images total: 3 sources and one output).