MIT EECS 6.815/6.865: Assignment 8:
Deconvolution and Poisson Image Editing

Due Wednesday Nov 6 at 9pm

1 Summary

- Deconvolve an image with a known kernel (non-blind deconvolution)
- Deblurring with gradient descent
- Deblurring with conjugate gradient descent
- Deblurring with noise regularization
- Poisson solver with gradient descent
- Poisson solver with conjugate gradient
- Make your own image composition

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 Deblurring

Our goal here is to reconstruct the sharp image given a blurry image due to hand-shake or incorrect focusing. For this we need to understand the formation of the blured image. A simple formation of a blurred image y from its sharp image x can be represented by the following:

$$y = x \otimes k \tag{1}$$

where $\otimes$ is the convolution operator, and k is called *point spread function*, or kernel in Pset 3. Recall that in Pset 3 we blur the image by convolving with a Gaussian kernel.

Non-blind deconvolution is the problem of recovering x from y, given the known k. To estimate the x, we will use a least square optimization approach. Formally, we estimate x by minimizing the difference between observed blurry image and the convolved x:

$$x^* = \text{argmin}\|y - x \otimes k\|^2 \tag{2}$$

As discussed in class, convolution is a linear operator, which means that we can write it with a matrix M such that $Mx = x \otimes k$. Then the deburring become minimzing the energy $Q(x) = \|Mx - y\|^2$.

Now the optimization is in the matrix form. The optimized x satisfies the following condition:

$$\frac{d}{dx}(Mx - y)^T(Mx - y) = 0 \tag{3}$$

This is equivalent to solving the following equation:

$$M^T Mx - M^T y = 0. \tag{4}$$

Equation 4 may look like solving another linear system of the same size as $Mx = y$. But the good news is, equation 4 is *determined*, since $M^T M$ is *positive definite* and $M^T y$ is in the image of $M^T M$. Now we are ready to solve equation 4.

2.1 Image dot product and convolution

We start by explaining basic elements in the algorithm. 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 descent dot product. Make sure you use bumpy array operations such as `np.sum` to make everything fast.

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

Note that the directly using `np.dot()` might not do what you want.

2.2 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. 4 that the gradient of the quadratic minimization energy 2 is the corresponding linear equation $M^T Mx - M^T y$. The direction of steepest descent is the opposite, which we denote as

$$r_i = M^T y - M^T M x_i$$

(the letter r comes from the fact that it is the *residual* of the equation, how much $M^T M x_i$ is different from $M^T y$)

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.

$$\begin{aligned}
\frac{d}{d\alpha} Q(x_{i+1}) &= \frac{d}{dx} Q \cdot \frac{dx_{i+1}}{d\alpha} && (5) \\
&= (M^T y - M^T M x_{i+1}) \cdot r_i && (6) \\
&= (M^T y - M^T M x_i - \alpha M^T M r_i) \cdot r_i && (7) \\
&= (r_i - \alpha M^T M x_i) \cdot r_i && (8)
\end{aligned}$$

and setting it to zero gives:

$$\alpha = \frac{r_i \cdot r_i}{r_i \cdot M^T M r_i} \qquad (9)$$

Implementation Write a function `deconvGradDescent(imblur, kernel, niter=10)` that implements gradient descent to solve for deconvolution.

Your iteration starts by initializing x with a black image. Then for each iteration, you want to compute the derivative direction and step size alpha.

The derivative is $r = M^T y - M^T M x = M^T(y - Mx)$. First writ a function `applyKernel(im, kernel)` that returns Mx (x is the image `im` here). You will have to replace the matrix-vector operations by their image implementation. Then write a function `computeResidual(kernel, x, y)` that returns $y - Mx$. Then write another function `applyConjugatedKernel(im, kernel)` that returns $M^T x$. With these three functions, you should be able to compute r_i for each iteration. Finally write `computeStepSize(r, kernel)` to compute the step size alpha. For this you have to write `dotIm( im1, im2)` that returns the inner product of the two images. With all these method, your `deconvGradDescent` should be obvious now.

One way to debug your code is to look at the energy Q(x). Q(x) should monotonically decrease with the iteration number.

2.3 Conjugate gradient descent

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 $M^T M$). 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. For simplicity, we denote $M^T M$ as A, and $M^T y$ as b. Initialize x_0 and

$$r_0 = d_0 = b - Ax_0$$

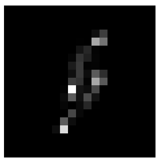
Then iterate:

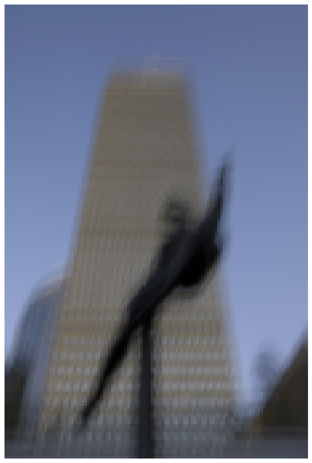
$$\alpha = \frac{r_i \cdot r_i}{d_i \cdot Ad_i}$$
$$x_{i+1} = x_i + \alpha d_i$$
$$r_{i+1} = r_i - \alpha Ad_i$$
$$\beta = \frac{r_{i+1} \cdot r_{i+1}}{r_i \cdot r_i}$$
$$d_{i+1} = r_{i+1} + \beta d_i$$

Implementation Write a function `deconvCG(imblur, kernel, niter=10)` that implements the above conjugate gradient algorithm to solve deconvolution. The specifications are the same as for gradient descent. You should reuse most of the methods written in the gradient descent. Write `computeGradientStepSize(r, d, kernel)` for alpha, and `computeConjugateDirectionStepSize(oldR, newR)` for beta.

Note how much faster the conjugate method converges. For debugging, you can check if Q(x) decrease monotonically and faster than conjugate gradient descent.

Real world kernel We provided an real world point spread function in our test case. We will convolve the sharp image with this this kernel for the input. Don't be frightened if your result has righting artifacts and is not satisfying. Deblurring is an ill-posed problem and still active research field.





2.4 Noise regularization

In the real world, the captured images contain the additive noise. One problem of the above methods is that they may boost up the noise. We consider the real world image formation as

$$y = Mx + n \tag{10}$$

where n is per-pixel additive noise. To avoid boosting up the noise, we add a regularization term in the energy term:

$$Q(x) = \|Mx - y\|^2 + \lambda x^T Lx \tag{11}$$

where λ control the relative importance of the regularization term, and L is *Laplacian*. The regularization term, $x^T Lx$, penalize the high frequency component, and favor a smoother solution for x. Again, we use matrix calculus to minimize this energy. With similar derivation, this is equivalent to solve the following linear system:

$$M^T Mx + \lambda Lx = M^T y \tag{12}$$

Using the same notation in the conjugate gradient descent, then the above system becomes the following:

$$(A + \lambda L)x = b \tag{13}$$

Now, solve the above equation with your conjugate gradient method. This should be straightforward because you only need to replace A in the conjugate gradient method with $A + \lambda L$.

Implementation Write `deconvCG_reg(imblur, kernel, lamb=0.05, niter=10)` that returns a debarred image without boosting up the noise. For this, write new sub functions that modify the methods written in the conjugate descent. Write `laplacianKernel()` that returns 3-by-3 laplacian kernel, and `def applyLaplacian(im)` that returns Lx. Write `applyAMatrix(im, kernel)` that returns Ax. With these, write `applyRegularizedOperator(im, kernel, lamb)` that returns $(A+\lambda L)x$. You have to write a modified method `computeGradientStepSize_reg(r, p, kernel, lamb)` to compute the alpha.

3 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.

3.1 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 :.

4 Poisson image editing

4.1 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

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

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

$$\min x^T Ax - 2x^T As + s^T As \tag{14}$$

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.

$$\frac{d}{dx}\left(x^T Ax - 2x^T As + s^T As\right) = 2Ax - 2As \tag{15}$$

which leads to the linear system

$$Ax = As \tag{16}$$

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.

Similar to deconvolution, 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.

4.2 Gradient descent

Write `poisson(bg, fg, mask, niter=200)` for poisson image composition. `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 the pixel in x outside the masked area by copying those in `bg`. See below for examples of x, and in particular x_0. You can do all 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.

In the test script, change the iteration number until convergence. You can test this function on the small inputs because it is slow to converge.

4.3 Conjugate gradient

Write `PoissonCG(bg, fg, mask, niter=200)` that returns the composited image with conjugate gradient method. Don't forget to mask the derivative direction.

4.4 Have fun

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

5 Extra credits

In deconvolution, our regurlarization term takes L_2 form. A better way is to use L_1 or $L_{0.8}$, so that the regularization term favor the solution with more sparsity. As a result, the output is sharper than L_2 norm. Implement the L_1 or $L_{0.8}$ with iterated reweighed method.

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

When computing the divergence, careful with centering (use a combination of forward and backward differences)

6 Submission

Turn in your images, python files, and make sure all your functions can be called from a module `a8.py`. Put everything into a zip file and upload to the submission site. Also, this time, you write `a8_readme.py`. Follow the instruction in the .py file. Upload everything except the data we provided.