CS 3220 Spring 2010
Homework 6

Problem 1: Condition number

Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix}.$$

(a) Compute the condition numbers $\mathrm{cond}_1(\mathbf{A})$ and $\mathrm{cond}_\infty(\mathbf{A})$ using the formula $\|\mathbf{A}\|\|\mathbf{A}^{-1}\|$, using the row-sum and column-sum interpretations of the norms.

(b) These condition numbers give us bounds on the magnification of relative error in the 1- and ∞-norms. For each norm, give a vector $\mathbf{x}$ and error $\Delta\mathbf{x}$ that achieve this bound exactly when we compute $\mathbf{Ax}$ from $\mathbf{Ax}$. (So that it feels like a perturbation, set $\|\Delta\mathbf{x}\|$ between 100 and 1000 times smaller than $\|\mathbf{x}\|$.)

Problem 2: Poisson Image Editing.

Suppose we want to replace an area in an image with a piece cut from another image, but hiding the boundary so that the seam won't be noticed. That is, we have a "background" image $b(x, y)$, a "foreground" image $f(x, y)$, and a binary mask $m(x, y)$ denoting a set of pixels where we'd like to replace the background

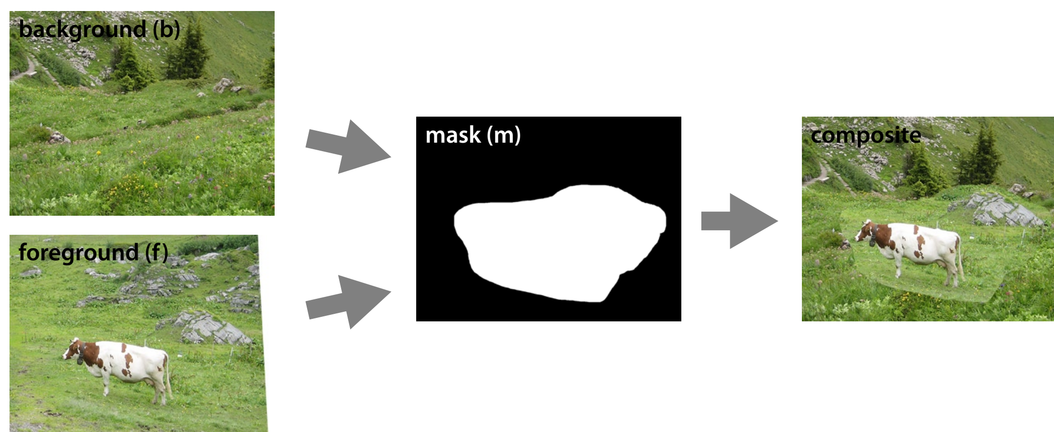

Figure 1: Simple cut-and-paste.

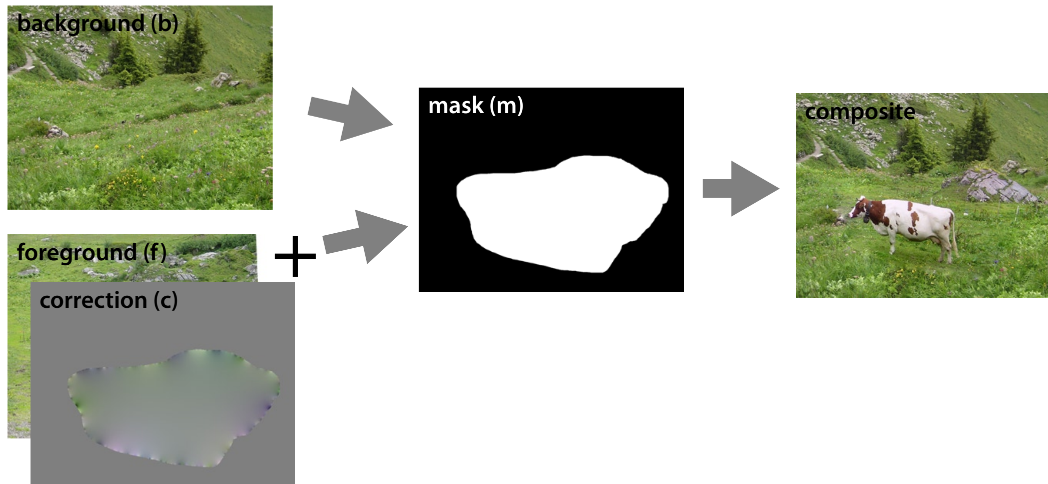

Figure 2: Cut-and-paste with correction image.

with the foreground. The basic patching operation is to define a new image

$$g(x,y) = \begin{cases} b(x,y) & \text{if } m(x,y) = 0 \\ f(x,y) & \text{if } m(x,y) = 1. \end{cases}$$

In Matlab we could express this using the logical indexing notation

```
g = b; g(mask) = f(mask);
```

For color images, think of doing this separately on three different matrices for the red, green, and blue color channels.

However, this simple replacement operation creates a patch with an obvious seam, if the colors don't happen to match well at the edges, as seen in Figure 1.

To get rid of the seam, while making the least noticeable changes possible to the foreground image, we can add a "correction image" $c(x,y)$ to the part of f that is inside the patch. The correction image should have the following properties: (a) it should cause $f + c$ and b to match well at the edges, and (b) it should nice and smooth. With appropriate notions of "matches well" and "smooth," we arrive at a linear system to be solved for the correction image c. Then we define

$$g(x,y) = \begin{cases} b(x,y) & \text{if } m(x,y) = 0 \\ f(x,y) + c(x,y) & \text{if } m(x,y) = 1. \end{cases}$$

The improved result can be seen in Figure 2.

We can make this into a linear system using the idea of a smooth image being one where each pixel's value is equal to the average of the values of the four neighboring

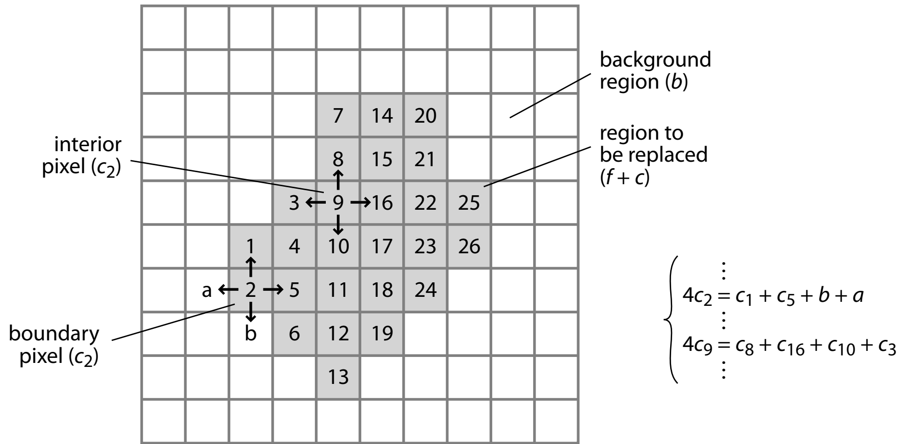

Figure 3: Constructing rows of the linear system.

pixels' values. The correction image should be smooth inside the patch (this is a statement about values of c), and it should match the values of $b - f$ at the edges (this is a statement about values of c and values of $b - f$). This way, when we add c to f, it will cause pixels at the edge of the patch to match b.

These constraints on c define a square linear system. For each interior pixel of the patch (interior meaning that the four neighbors are all inside the patch) we have

$$c(x, y) = \frac{c(x-1, y) + c(x+1, y) + c(x, y-1) + c(x, y+1)}{4}$$

and for boundary pixels the corresponding values of c are replaced with values of $b - f$. For instance, for a pixel whose right-hand neighbor is outside the patch:

$$c(x, y) = \frac{c(x-1, y) + c(x+1, y) + c(x, y-1) + (b(x, y+1) - f(x, y+1))}{4}$$

(I am using Matlab indexing conventions here, so the row is first, then the column.) Each pixel in the patch gives rise to one equation, and each pixel in the patch is also a variable in the system, so this is a square system. Figure 3 illustrates this for a different example.

To do a cut-and-paste operation, then, we follow these steps:

1. Construct the matrix system $\mathbf{Ac} = \mathbf{d}$
2. Solve the system to get the vector $\mathbf{c}$ containing the pixel values of the correction image.
3. Replace each pixel in the masked region of b with the sum of the corresponding pixels in f and c.

Write a function `result = poisson_paste(fg, bg, mask)` that achieves this for grayscale images "fg" and "bg." For color images you can just call this three times—but bonus points for setting it up to build the matrix once and solve three times. The homework is provided with a few test cases, together with a script to show how to read in the images and take apart the color channels, and it is fun to make your own too, with your own photos.

When you are done you'll have a tool very similar to the Healing Brush in Adobe Photoshop (they are not telling what they use exactly, but it sure acts similar to this method). This homework is a special case of the algorithm described in the SIGGRAPH 2003 paper "Poisson Image Editing" by Pérez, Gangnet, and Blake (http://portal.acm.org/citation.cfm?id=882269).

Hints. Remember that for boundary pixels, the values of b and f are constants, not variables in the system. To put the equations into a matrix problem you need the variables on the left and the constants on the right of the equals sign.

Also remember that pixels outside the masked region are not variables. Figuring out how to number the variables (i.e. how to set up correspondence between rows in the linear system and pixels in the masked region), and then how to find out the numbers of the neighboring pixels, is a tricky practical problem with various solutions. I found it handy to keep a matrix the same size as f, b, and m containing integers just like the numbers in the grid in Figure 3. You may find logical indexing to be helpful (see `doc logical` and the Matlab documentation under "Getting Started : Matrices and Arrays : More About Matrices and Arrays."

The matrix you will build is pretty big for a decent-sized patch, and solving it will start to get slow. The matrix is also sparse—the vast majority of entries are zero. When you know a matrix will be sparse, you can help Matlab help you by creating it using `sparse(n,n)` rather than `zeros(n,n)`. This way it will allocate memory proportional to the number of nonzero entries, not the total size of the matrix. Under the right circumstances (which these are) this can speed up the solution of systems drastically. For fun, try using the function `spy(A)` on your matrix to see what the pattern of nonzeros looks like.