Algebraic Multigrid

We're going to discuss algebraic multigrid, but first begin by discussing ordinary multigrid. Both of these deal with scale space, examining the image at multiple scales. This is important for segmentation, because an image segmentation is really a representation at a coarser scale than the pixel. To a first approximation, think that multigrid is like diffusion, and algebraic multigrid is like anisotropic diffusion. In both cases we build a scale space on top of them.

Multigrid

The basic idea behind multigrid comes from the following insight: if we smooth something, we can then subsample it without losing any information. In multigrid, we use this to solve PDEs. We start with some random initialization, in which our error is like white noise. Then we use an iterative solver, that has the effect of smoothing the error. After a while, the error is all low frequency. So we can subsample to a coarser scale, without losing information. We repeat, but everything is cheaper at the coarser scale. Eventually, we reach a small scale, where we can inexpensively get an exact solution. Then, knowing the solution at a coarse scale, we interpolate this to get an approximate solution at a fine scale. This new solution only has high frequency error, which we can get rid of pretty efficiently.

We're going to break this into two parts. First, we'll just consider the representation constructed by multigrid. Then we'll see how it is used to solve PDEs.

Gaussian Pyramid

The basic idea behind multigrid is also used in vision, and called the Gaussian pyramid. It was introduced by Burt and Adelson in the early 80s, but seems to have appeared in multigrid earlier.

As we discussed previously, we can't just shrink an image by sampling it. If we do that, we get aliasing. High frequencies have a big, random effect on the sampled image. So we first smooth, to get rid of high frequencies. Then when we sample, we don't lose any information. If we smooth with a Gaussian, and repeat this, we get the *Gaussian pyramid.*

This has been suggested for a bunch of vision applications:

Motion: We want to find the transformation that relates two images taken from different viewpoints. If we start at the top of the pyramid, it's easy to find the best solution, because there are so few. Then, as we move down the pyramid, we can assume that we have an approximate solution from the coarser level. We can take a Taylor series expansion around that solution, and analytically find the solution at the finer scale. This kind of approach is widely used.

Matching: Here we have an image and a template. We match them at a coarse scale, and then at the finer scale, we search near that scale. This is like motion, but a different application, and we may not be able to solve anything analytically.

Compression: An early suggestion was to use this pyramid for compression. This was through the *Laplacian* pyramid. This is like the G.P., but at each level we store the difference between the image, and the upsampled version of the coarser image. The idea is that this difference should be less correlated and lower energy, so easier to compress. Has the nice advantage that it makes progressive encoding easy. Some version of this is included in JPEG. The Laplacian pyramid isn't really directly used in compression, but the multiscale idea is one of the forerunners of wavelets, which are used in JPEG 2000.

Multigrid for solving PDEs

We'll look at one example of this, using the Poisson equation. I will be very sloppy about boundary conditions and discretization, and just try to give the intuitions.

$$\Delta u = f(x,y) \quad i.e. \quad -u_{xx} - u_{yy} = f(x,y)$$

If we write this discretely, for a grid we get:

$$f(x,y) = \left(\frac{1}{h^2}\right)\left(4u(x,y) - u(x+1,y) - u(x-1,y) - u(x,y+1) - u(x,y-1)\right)$$

This leads to an iterative algorithm, where we base one of these values on previous versions of the others.

$$z^{m+1} = \frac{1}{4}\left(h^2 f(x,y) + u(x+1,y) + u(x-1,y) + u(x,y+1) + u(x,y-1)\right)$$

Then we can set:

$$u^{m+1} = z^{m+1}$$

Note that at least it's easy to see that the right solution is a fixed point of this iteration. We can study convergence rates, and issues of local minima, but not here. This is called the Jacobi iteration. Other methods (Gauss-Seidel) are better, but this is simpler.

It may be a good idea to slow down the updates a bit, as:

$$u^{m+1} = u^m(1-w) + wz^{m+1}$$

It turns out that $w = 4/5$ is a good choice, and we get:

$$u^{m+1} = \frac{1}{5}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \otimes u^m + \frac{h^2}{5} f \quad \textit{where} \otimes \textit{indicates convolution}$$

That is, we are updating *u* by basically smoothing the old *u* with an averaging filter.

So, let's look at what happens to the error with this iteration. We define the error at iteration m to be:

$$v^m = u - u^m$$

Then we can show:

$$v^{m+1}(x,y) = \frac{1}{5}\left[v^m(x-1,y) + v^m(x+1,y) + v^m(x,y) + v^m(x,y-1) + v^m(x,y+1)\right]$$

Substituting, the left side equals:

$$v^{m+1}(x,y) = u(x,y) - u^{m+1}(x,y)$$

while the right side is:

$$\frac{1}{5}\left[v^m(x-1,y) + v^m(x+1,y) + v^m(x,y) + v^m(x,y-1) + v^m(x,y+1)\right]$$
$$= \frac{1}{5}\left[\begin{array}{l} u(x-1,y) - u^m(x-1,y) + u(x+1,y) - u^m(x+1,y) + u(x-1,y) - u^m(x-1,y) + u(x,y) - u^m(x,y) \\ + u(x,y-1) - u^m(x,y-1) + u(x,y+1) - u^m(x,y+1) \end{array}\right]$$

We have

$$u^{m+1}(x,y) = \frac{1}{5}\left[u^m(x-1,y) + u^m(x+1,y) + u^m(x,y) + u^m(x,y-1) + u^m(x,y+1)\right] - f(x,y)\frac{h^2}{4}$$

So, canceling some terms, we just need to show that:

$$u\ (x,y) = \frac{1}{5}\left[u\ (x-1,y) + u\ (x+1,y) + u\ (x,y) + u\ (x,y-1) + u\ (x,y+1)\right] - f(x,y)\frac{h^2}{4}$$

This is just the discrete statement of the Poisson equation.

So, the point is that this iteration smooths the error in our original estimate. Intuitively, this makes sense, since at every iteration we are estimating a point by averaging the neighbors, which averages the error. We are smoothing with something like a low pass

filter, so what happens is that the high frequency components of the noise disappear quickly, but low frequency noise is slow to disappear. So when high frequency noise is mostly gone, we downsample, and get a smaller problem, where the noise is high frequency again. We recursively repeat this process, and get an exact solution to the coarser problem. Then we upsample. When we upsample a noise free solution, we may get noise, but only in high frequency, so smoothing quickly gets rid of that.

Of course, we need to fill in some details here:

- First, we don't actually downsample u^m because if f has significant high frequency components, so will u^m. Instead, we formulate an iteration on the *defect*. If we write the Poisson equation as $Lu=f$, where L is a linear operator, we define the defect, $d = f - Lu$. The defect can be guaranteed to get smooth as we iterate.
- To downsample (shrink) the image, we just select every other pixel in each row (different options are possible). To upsample, we interpolate the missing pixels.
- If we are operating at many scales, there are many options about when to move up or down in scale.
- Overall, this can run in linear time, which is optimal.

Algebraic Multigrid

Now I'm just going to give some quick intuitions about how these ideas are extended in algebraic multigrid. This is a method that is useful when our iterative method performs some type of anisotropic smoothing. This can occur because of some irregularities or asymmetries in the PDE we are solving. For example, if we have a PDE like:

$$-u_{xx} - \frac{u_{yy}}{1000} = f(x,y)$$

This causes an asymmetry in the extent to which horizontal neighbors and vertical neighbors wind up influencing the solution at a point. Or, noting that the Poisson equation is the equation describing electromagnetic fields, you can imagine that if a small barrier blocks part of this field, this creates an anisotropy in the solution as well. In these cases, the proper iterative solution method winds up performing anisotropic smoothing. These asymmetries can occur many other ways, and in fact AMG can be applied to problems like sparse linear equations, where we may not even have a problem on a grid, but we won't worry about that.

As an example, for the above equation, we wind up with an iteration that averages neighbors in the horizontal direction, but not the vertical one. This gives us bands in the solution that are independently averaged, and so smooth in one direction but not the other. So after iterations, we have significant high frequency components in the error, and if we subsample, we will alias. This can be seen with an example where there is a small square that is different from the rest of the error, and subsampling can lose it.

More generally, we can start with our original grid and form weighted links between neighboring pixels, and then smooth by taking weighted averages according to these

weights (We can also formulate this more generally, for arbitrary weighted graphs). This is essentially non-linear diffusion. After smoothing, each pixel will have a similar intensity to the pixels that it is strongly connected to, or at least an intensity that can be linearly reconstructed from these. Then we subsample the graph so that every node that is deleted is strongly connected to some that remain, so we can reconstruct it. This algorithm has the same structure as regular multigrid, but the subsampling (*coarsening)* depends on the weights.

In the above example, we wind up sampling in one direction, but not subsampling in the other. This leads to a slower sampling rate, maybe ½ instead of ¼. But AMG still has the same basic structure of MG, smooth to get rid of error at one level, scale, then subsample and get rid of error at the next scale.

In light of this, we can say that what Sharon et al. are doing is perhaps anisotropic diffusion, with sampling for efficiency.