

# Numerical Solutions To Poisson's/Laplace's Equations

## 1 Introduction

In this note we show two techniques that can be used to solve Laplace's equation given boundary conditions. The first attempts to solve it directly, through a technique called *relaxation*. The vanilla version of relaxation is both slow and inaccurate, but by combining it with multi-grid methods and iterative matrix solvers like conjugate gradient, we can speed up relaxation by orders of magnitude while also getting accurate solutions. The second technique uses Green's functions and the convolution theorem (plus, once again, an iterative matrix solver) to solve for the charge distribution on the boundaries given the boundary conditions. Poisson problems can be converted to Laplace problems by subtracting the potential from the fixed charges, and so our Laplace solvers will also work on Poisson's equation as well.

## 2 Relaxation

### 2.1 Simple Relaxation

For Laplace's equation, we have  $\nabla^2 V = 0$ . In one dimension, let us have three points,  $x_l$ ,  $x_m$ , and  $x_r$  (for left, middle, and right) separated by  $dx$  with potential  $V_l$ ,  $V_m$ , and  $V_r$ , then the first derivative on the left side of  $x_m$  is  $V'_l = (V_m - V_l)/dx$  and the derivative on the right side is  $V'_r = (V_r - V_m)/dx$ . The second derivative is the derivative of the first derivatives, so  $V'' = (V'_r - V'_l)/dx = (2V_m - V_r - V_l)/dx^2$ . If we are in a region without charge, this must be equal to zero, for  $V_m = (V_r + V_l)/2$ . This should not come as a complete surprise, since we already knew that for solutions to Laplace's equation, the potential at a point was the average in a shell around it. We can trivially extend this to higher dimensions since we just stack the second derivatives in each dimension.

For the simple case where we specify the potential as our boundary conditions, we can make a first pass at solving this by the *method of relaxation*. For every point in the interior, we replace the potential by the average of the potential in its neighboring points. Repeat until things quit changing. This has the benefit of being very easy to understand, but in practice it's quite slow to settle down. Note that we could also solve Poisson's equation, since in that case we have  $\nabla^2 V = -\rho/\epsilon_0$ . In three dimensions, this becomes  $(6V_m - V_l - V_r - V_u - V_d - V_f - V_b)/dx^2 = -q_{cell}/\epsilon_0 = -\rho dx^3/\epsilon_0$  or  $V_m = V_{ave} - \rho dx/6\epsilon_0$ . In two dimensions, we get a similar result with 6 changing to 4. If we have a point charge, we might also prefer the form with  $q_{cell}$ . The important thing to keep in mind when running relaxation is that the cells where boundary conditions are specified are *not* updated. We already know the potential there, and it had better not change!

## 2.2 Relaxation with Conjugate Gradient

We can do a bit better in our solution by noting that what we've really written down is a (sparse) matrix equation. Back to one-dimension, we could rewrite things as follows:

$$[-12 - 1][V_l V_m V_r]^T = -q_{cell}/\epsilon_0$$

This has to be true for each cell, though, so we can extend the matrix to be

$$[-1 \ 2 \ -1 \ 0 \ 0 \ \dots]$$

$$[0 \ -1 \ 2 \ -1 \ 0 \ \dots]$$

$$[0 \ 0 \ -1 \ 2 \ -1 \ \dots]$$

We can then harness the power of linear algebra (including any solving methods we wish to use) to solve our system. Note that this generalizes to higher dimensions, but the bookkeeping might be trickier to write down explicitly. Fortunately, we usually don't have to do that. Unfortunately, we can't actually solve the system as we have written it. You'll see that we have two fewer equations than unknowns, so this is not actually a closed system. We must supply *boundary conditions* to fill in the missing information. One natural thing to do is specify the potential on the edges. At the left edge, we would then have  $2V_m - V_l - V_r = 0$ , but we write down what  $V_l$  is in advance. This is often zero, but not always. In that case, our equation becomes  $2V_m - V_r = V_l$ . Essentially, we cut the first and last columns out of the matrix (reducing the number of unknowns by 2) and put the boundary conditions into the right hand side. In higher dimensions, we would need to specify what happens on the boundary surface. Of course, other possibilities exist as well. The root problem is that at the edge, we can't take our usual second derivative without referring to points off the edge, so instead we could estimate the second derivatives using only points we do actually have. Similarly, any point in the interior where the potential is specified can be handled the same way, where it isn't solved for and its value gets moved to the right hand side.

Now that we have written our problem as a matrix equation, other solvers like conjugate gradient can be used that will iteratively solve for the potential. Happily, conjugate gradient only requires that you be able to take a vector, and get the result of the matrix times that vector. Since we know that, at least in the interior, our matrix takes  $2N_{dim}$  times the value and subtracts the average of neighbors, we can just have a function that does that, without ever having to do the bookkeeping of writing down explicit matrix elements. To handle boundaries, since we know their values will be on the right hand side, we can still just do our usual operation if we artificially set the boundary cells to zero.

## 2.3 Multi-grid Relaxation

Conjugate gradient will converge much faster than relaxation. Since we're more-or-less doing the same thing we did for relaxation, this turns out to be a computational win. However, it is still rather slow. The reason is that for both

relaxation (not too hard to see) and conjugate gradient (less obviously, but still true) each time we pass through an iteration, the influence of a potential only spreads by one cell. If we discretize our region into say  $1000 \times 1000$  ( $\times 1000$ ) cells, we need to loop through a thousand times before we have any prayer of getting the correct answer (conjugate gradient will actually get there, but relaxation takes many times this to settle down). An alternative solution is to solve our problem at a lower resolution, say  $100 \times 100$  ( $\times 100$ ), which is 100 (2D) or 1000 (3D) times faster than the full resolution. If we then solve our original problem starting with the low-resolution solution (interpolated to higher resolution), the information only has to travel a little ways, since we really just need to “smooth out” the interpolation and so we get a good solution in just a few iterations rather than 1000. These techniques are called *multi-grid methods*. A multi-grid version of conjugate gradient can actually run quite quickly. If we change the resolution by a factor of two at each step, it is eight times cheaper every time we drop the resolution. So, the dominant work we have to do is just the few iterations at the highest resolution. In practice, we have taken the amount of work required to solve a large matrix equation from  $n^3$  (where  $n$  is the total number of cells) to a few times  $n$  - an enormous savings indeed!

I have coded a 2D version of this in `laplace_2d_multi.py`. You can set whatever boundary conditions you like fairly easily. For the case of potential held at zero on the edges, and a plate held at 1 with a hole in the middle, you can see the potential in Figure 1, the potential along the slice with the plate in Figure 2, and the charge along the slice in Figure 3. Note that the numerical oscillation that showed up in our separation of variables solution are gone!

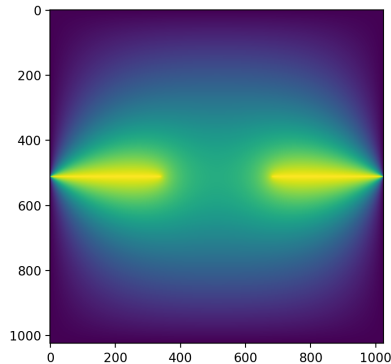


Figure 1: Potential everywhere in space when the edges are held at zero, a plate through the middle is held at 1, but the plate has a hole sliced into it.

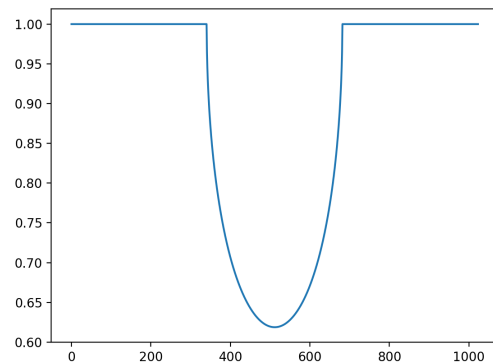


Figure 2: Potential along the slice that contains the plate. As expected, the potential is constant along the plate, but (not unexpectedly) drops in the gap. The numerical oscillations we saw in separation of variables are happily gone here, since the numerical solver isn't trying to express sums of sines/cosines.

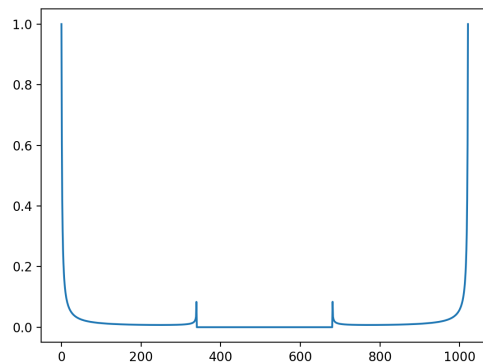


Figure 3: Same as above, but now for charge. The charge increases at the edges, as expected, but goes to zero in the interior, also as expected.

### 3 Green's Function/Fourier Techniques

An completely alternative way of attacking Laplace's equation is to ignore the interior and focus only on the boundary conditions using Green's functions. Assuming our standard setup (no charge in the interior), we know that any potential on the boundaries must come from charge on the boundaries. We can then try to work out the charge distribution that gives us the correct boundary

potential. Mathematically, we have

$$V(r) = \int_{BC} \frac{\rho(r')}{4\pi\epsilon_0|r-r'|} d^3r'$$

where the integral only goes over the boundary surfaces. This integral is a special form called a convolution, usually written as

$$h(x) = f(x) * g(x) \equiv \int f(x')g(x-x')dx'$$

By comparing the potential to the definition of the convolution, we see that

$$V(r) = \frac{1}{4\pi\epsilon_0} \frac{1}{r} * \rho(r)$$

If we knew how to solve  $V = \frac{1}{r} * \rho r$  on the boundaries, we'd be done. Note that  $\frac{1}{r}$  is the Green's function for Poisson's equation in three dimensions; in two dimensions, you would replace it with a natural log.

At first glance, solving this problem might seem nearly impossible, since we have to guess a solution, then do an unpleasant-looking integral to see how well we did, then update our guess. It turns out we have a trick up our sleeve, in the form of the *convolution theorem*. The convolution theorem states that  $H(k) = F(k)G(k)$  where  $H(k)$  is the Fourier transform of  $h(x)$ . In words, I can convolve two functions by inverse Fourier transforming the product of the Fourier transforms of the functions. Fast Fourier transforms (FFTs) can do this on a computer in order  $n \log(n)$  time, so convolutions can be carried out extremely quickly.

We now know we can *check* our guess very efficiently via a convolution. But how should we update our guesses? After a moment's thought, perhaps you can convince yourself that a Fourier transform can be treated as a matrix multiply. If we use the standard discrete definition of the Fourier transform, then the value of  $F(k)$  equals the dot product of  $\exp(-2\pi i k x / N)$  with  $f(x)$ . We do this for every value of  $k$ , and so the Fourier transform becomes a matrix multiply where the  $k$ th row is  $\exp(-2\pi i k x / N)$ . Since the Fourier transform is a matrix multiply, and rescaling the Fourier transform of  $\rho$  by the Fourier transform of  $1/x$  is a (diagonal) matrix multiply, then the whole convolution operator is a matrix multiply. This means the entire problem, once again, maps to solving a linear system  $V = C\rho$  where  $C$  is the convolution by  $1/r$ . We have already seen that conjugate gradient works in solving these sorts of systems, and we can apply it directly here. A fully worked (2D) example code is in `laplace_fft.py`.

## 4 Comparison and Discussion

Both (multi-grid) relaxation and Fourier-based techniques can be used to efficiently solve Laplace's/Poisson's equations for arbitrary boundary conditions. The path they take to get to the solutions look *very* different - relaxation has

an accurate solution grow in from the boundaries, while the Fourier technique gradually improves a guess to a global solution. The Fourier technique requires you to work out the Green's function for your problem first, but this is also a benefit - you can solve any system that obeys superposition by substituting in the correct Green's function in place of  $\frac{1}{x}$ . The Fourier solve also takes many fewer iterations to converge than relaxation, but each iteration is much slower since one must do two Fourier transforms. On my laptop, they run in comparable amounts of time with perhaps a slight edge towards multi-grid relaxation. Your mileage may vary, and may depend on factors like number of dimensions, accuracy requirements and the like. Both techniques can be adapted to solve Poisson's equation by subtracting the potential due to the fixed charge on the boundaries. More complicated situations, such as applying fixed charge to a conductor, may not be directly solvable, but can usually be broken down into the superposition of problems that, again, either solver could be used on<sup>1</sup>

While this has not been an exhaustive treatment of numerical solutions, I hope it gives you a flavor of the richness in numerical solutions to electrostatics, and by extension any of the many physics problems that reduce to Poisson's equation.

---

<sup>1</sup>In the case of charge on a conductor, we can solve the problem with our correct boundary conditions and the conductor held at zero potential, then solve the conductor held at fixed potential and the other boundaries held at zero potential. The sum of the first solution plus any constant times the second solution will match the potential on the boundary plus keep the potential on the conductor constant. We then take the Laplacian of the two solutions to find the charge on the conductor and scale the second solution to match the correct total charge.