Lab 9

Poisson's Equation: Iteration Methods

In three dimensions, Poisson's equation is given by

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\epsilon_0} \tag{9.1}$$

Poisson's equation is used to describe the electric potential in a region of space with charge density described by ρ:[1] You can solve the full 3D equation using the technique we teach you in this lab, but we won't make you do that here. Instead, we'll focus on geometries that are infinitely long in the z-dimension with a constant cross-section in the x-y plane. In these cases the z derivative goes to zero, and Poisson's equation reduces to

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\epsilon_0} \tag{9.2}$$

Note that by studying this equation we are also studying Laplace's equation (Poisson's equation with $\rho = 0$) and the steady state solutions to the diffusion equation in two dimensions ($\partial T/\partial t = 0$ in steady state).

Finite difference form

The first step in numerically solving Poisson's equation is to define a 2-D spatial grid. For simplicity, we'll use a rectangular grid where the x coordinate is represented by N_x values x_j equally spaced with step size h_x, and the y coordinate is represented by N_y values y_k equally spaced with step size h_y. This creates a rectangular grid with $N_x \times N_y$ grid points, just as we used in Lab 6 for the 2-d wave equation. We'll denote the potential on this grid using the notation $V(x_j, y_k) = V_{j,k}$.

The second step is to write down the finite-difference approximation to the second derivatives in Poisson's equation to obtain a grid-based version of Poisson's equation. In our notation, Poisson's equation is the represented by

$$\frac{V_{j+1,k} - 2V_{j,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} - 2V_{j,k} + V_{j,k-1}}{h_y^2} = -\frac{\rho_{j,k}}{\epsilon_0} \tag{9.3}$$

This set of equations can only be used at interior grid points because on the edges it reaches beyond the grid, but this is OK because the boundary conditions tell us what V is on the edges of the region.

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 138-150.

Equation (9.3) plus the boundary conditions represent a set of linear equations for the unknowns $V_{j,k}$, so we could imagine just doing a big linear solve to find V all at once. Because this sounds so simple, let's explore it a little to see why we are not going to pursue this idea. The number of unknowns $V_{j,k}$ is $N_x \times N_y$, which for a 100×100 grid is 10,000 unknowns. So to do the solve directly we would have to be working with a $10,000 \times 10,000$ matrix, requiring 800 megabytes of RAM to store the matrix. Doing this big solve is possible for 2-dimensional problems like this because computers with much more memory than this are common. However, for larger grids the matrices can quickly get out of hand. Furthermore, if you wanted to do a 3-dimensional solve for a $100 \times 100 \times 100$ grid, this would require $(10^4)^3 \times 8 = 8 \times 10^{12}$, or about 8 terabytes of memory. Computers like this are becoming possible, but this is still a tiny computational grid. So even though computers with large amounts of RAM are becoming common, people still use iteration methods like the ones we are about to describe.

Iteration method on a simple example

Consider solving this equation:

$$x = e^{-x} \tag{9.4}$$

One method to solve this equation is to make a guess for the solution, call it x_0, and then iterate on the equation like this:

$$x_{n+1} = e^{-x_n} \tag{9.5}$$

For large values of n, we find that the process converges to the exact solution $\bar{x} = 0.567$. Let's do a little analysis to see why it works. Let $\bar{x}$ be the exact solution of this equation and suppose that at the n^{th} iteration level we are close to the solution, only missing it by the small quantity δ_n like this: $x_n = \bar{x} + \delta_n$. Let's substitute this approximate solution into Eq. (9.5) and expand using a Taylor series. Recall that the general form for a Taylor's series is

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{9.6}$$

When we substitute our approximate solution into Eq. (9.5) and expand around the exact solution $\bar{x}$ we get

$$x_{n+1} = e^{-\bar{x}-\delta_n} \approx e^{-\bar{x}} - \delta_n e^{-\bar{x}} + \cdots \tag{9.7}$$

If we ignore the terms that are higher order in δ (represented by $\cdots$), then Eq. (9.7) shows that the error at the next iteration step is $\delta_{n+1} = -e^{-\bar{x}}\delta_n$. When we are close to the solution the error becomes smaller every iteration by the factor $-e^{-\bar{x}}$. Since $\bar{x}$ is positive, $e^{-\bar{x}}$ is less than 1, and the algorithm converges. When iteration works it is not a miracle—it is just a consequence of having this expansion technique result in an error multiplier that is less than 1 in magnitude.

P9.1 Write a program to solve the equation $x = e^{-x}$ by iteration and verify that it converges. Then try solving this same equation the other way round: $x = -\ln x$ and show that the algorithm doesn't converge. If you were to use the $\bar{x} + \delta$ analysis above for the logarithm equation, you'd find that the multiplier is bigger than one. Our time is short today, so we won't make you do this analysis now, but it is a good exercise if you want to look at it on your own time.

Iteration method on Poisson's equation

Well, what does this have to do with Poisson's equation? If we solve the finite-difference version of Poisson's equation (Eq. (9.3)) for $V_{j,k}$, we find

$$V_{j,k} = \left(\frac{V_{j+1,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} + V_{j,k-1}}{h_y^2} + \frac{\rho_{j,k}}{\epsilon_0} \right) \Bigg/ \left(\frac{2}{h_x^2} + \frac{2}{h_y^2} \right) \tag{9.8}$$

With the equation in this form we could just iterate over and over by doing the following.

1. Choose an initial guess for the interior values of $V_{j,k}$.
2. Use this initial guess to evaluate the right-hand side of Eq. (9.8)
3. Replace our initial guess for $V_{j,k}$ by this right-hand side, and then repeat.

If all goes well, then after many iterations the left and right sides of this equation will agree and we will have a solution. [2]

To see, let's notice that the iteration process indicated by Eq. (9.8) can be written in matrix form as

$$V_{n+1} = \mathbf{L} V_n + r \tag{9.9}$$

where $\mathbf{L}$ is the matrix which, when multiplied into the vector V_n, produces the $V_{j,k}$ part of the right-hand side of Eq. (9.8) and r is the part that depends on the charge density $\rho_{j,k}$. (Don't worry about what $\mathbf{L}$ actually looks like; we are just going to apply general matrix theory ideas to it.) As in the exponential-equation example given above, let $\bar{V}$ be the exact solution vector and let δ_n be the error vector at the n^{th} iteration. The iteration process on the error is, then,

$$\delta_{n+1} = \mathbf{L} \delta_n \tag{9.10}$$

Now think about the eigenvectors and eigenvalues of the matrix $\mathbf{L}$. If the matrix is well-behaved enough that its eigenvectors span the full solution vector space of size $N_x \times N_y$, then we can represent δ_n as a linear combination of these eigenvectors. This then invites us to think about what iteration does to each eigenvector.

Carl Friedrich Gauss (1777–1855, German)

[2] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 429-441.

The answer, of course, is that it just multiplies each eigenvector by its eigenvalue. Hence, for iteration to work we need all of the eigenvalues of the matrix **L** to have magnitudes less than 1.

So we can now restate the conditions for this approach to work: Iteration on Eq. (9.8) converges if all of the eigenvalues of the matrix **L** on the right-hand side of Eq. (9.8) are less than 1 in magnitude. This statement is a theorem which can be proved if you are really good at linear algebra, and the entire iteration procedure described by Eq. (9.9) is known as *Jacobi iteration*. Unfortunately, even though all of the eigenvalues have magnitudes less than 1 there are lots of them that have magnitudes very close to 1, so the iteration takes forever to converge (the error only goes down by a tiny amount each iteration).

But Gauss and Seidel discovered that the process can be accelerated by making a very simple change in the process. Instead of only using old values of $V_{j,k}$ on the right-hand side of Eq. (9.8), they used values of $V_{j,k}$ as they became available during the iteration. (This means that the right side of Eq. (9.8) contains a mixture of V-values at the n and $n+1$ iteration levels.) This change, which is called *Gauss-Seidel iteration* is really simple to code; you just have a single array in which to store $V_{j,k}$ and you use new values as they become available. Here as a coded example of Guass-Seidel iteration for a rectangular region grounded on two sides, with the two other sides held at a potential:

Listing 9.1 (GaussSeidel.py)

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# Make the grid
xmin = 0
xmax = 2
Nx = 80
x,hx = np.linspace(xmin,xmax,Nx,retstep = True)
hx2 = hx**2
ymin = 0
ymax = 2
Ny = 40
y,hy = np.linspace(ymin,ymax,Ny,retstep = True)
hy2 = hy**2
X,Y = np.meshgrid(x,y,indexing='ij')

# Initialize potential
V = 0.5*np.ones_like(X)

# Enforce boundary conditions
V[:,0] = 0
V[:,-1] = 0
```

```
V[0,:] = 1
V[-1,:] = 1

# Allow possibility of charge distribution
rho = np.zeros_like(X)

# Iterate
denom = 2/hx2 + 2/hy2
fig = plt.figure(1)
for n in range(200):
    # make plots every few steps
    if n % 10 == 0:
        plt.clf()
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(X,Y,V)
        ax.set_zlim(-0.1, 2)
        plt.xlabel('x')
        plt.ylabel('y')
        plt.draw()
        plt.pause(0.1)

    # Iterate the solution
    for j in range(1,len(x)-1):
        for k in range(1,len(y)-1):
            V[j,k] = ( (V[j+1,k] + V[j-1,k])/hx2
                      +(V[j,k+1] + V[j,k-1])/hy2
                      +rho[j,k]) / denom
```

P9.2 Paste the code above into a program, run it, and watch the solution iterate. Study the code, especially the part in the loop. When you understand the code, call a TA over and explain it to them to pass off this part.

Successive over-relaxation

Gauss-Seidel iteration is not the best we can do, however. To understand the next improvement let's go back to the exponential example

$$x_{n+1} = e^{-x_n} \tag{9.11}$$

and change the iteration procedure in the following non-intuitive way:

$$x_{n+1} = \omega e^{-x_n} + (1-\omega)x_n \tag{9.12}$$

where ω is a number which is yet to be determined.

P9.3 Verify quickly on paper that even though Eq. (9.12) looks quite different from Eq. (9.11), it is still solved by $x = e^{-x}$.

If we insert $x_n = \bar{x} + \delta_n$ into this new equation and expand as before, the error changes as we iterate according to the following

$$\delta_{n+1} = \left(-\omega e^{-\bar{x}} + 1 - \omega\right)\delta_n \tag{9.13}$$

Notice what would happen if we chose ω so that the factor in parentheses were zero: The equation says that we would find the correct answer in just one step! Of course, to choose ω this way we would have to know $\bar{x}$, but it is enough to know that this possibility exists at all. All we have to do then is numerically experiment with the value of ω and see if we can improve the convergence.

P9.4 Write a program that accepts a value of ω and runs the iteration in Eq. (9.12). Experiment with various values of ω until you find one that does the best job of accelerating the convergence of the iteration. You should find that the best ω is near 0.64, but it won't give convergence in one step. See if you can figure out why not.

Hint: Think about the approximations involved in obtaining Eq. (9.13). Specifically go back to the previous derivation and look for terms represented by dots.

As you can see from Eq. (9.13), this modified iteration procedure shifts the error multiplier to a value that converges better. So now we can see how to improve Gauss-Seidel: we just use an ω multiplier like this:

$$V_{n+1} = \omega\left(\mathbf{L}V_n + r\right) + (1-\omega)V_n \tag{9.14}$$

then play with ω until we achieve almost instantaneous convergence.

Sadly, this doesn't quite work. The problem is that in solving for $N_x \times N_y$ unknown values $V_{j,k}$ we don't have just one multiplier; we have one for each eigenvalue of the matrix. So if we shift one of the eigenvalues to zero, we might shift another one to a value with magnitude larger than 1 and the iteration will not converge at all. The best we can do is choose a value of ω that centers the entire range of eigenvalues symmetrically between -1 and 1.

Using an ω multiplier to shift the eigenvalues is called *Successive Over-Relaxation*, or SOR for short. Here it is written out so you can code it:

$$V_{j,k} = \omega\left(\frac{V_{j+1,k}+V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1}+V_{j,k-1}}{h_y^2} + \frac{\rho_{j,k}}{\epsilon_0}\right)\Bigg/\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}\right) + (1-\omega)V_{j,k} \tag{9.15}$$

And what value should we use for ω? The answer is that it depends on the values of N_x and N_y. In all cases ω should be between 1 and 2, with $\omega = 1.7$ being a typical value. Some wizards of linear algebra have shown that the best value of ω

when the computing region is rectangular and the boundary values of V are fixed (Dirichlet boundary conditions) is given by

$$\omega = \frac{2}{1+\sqrt{1-R^2}} \tag{9.16}$$

where

$$R = \frac{h_y^2 \cos(\pi/N_x) + h_x^2 \cos(\pi/N_y)}{h_x^2 + h_y^2}. \tag{9.17}$$

These formulas usually give a reasonable estimate of the best ω to use. Note, however, that this value of ω was found for the case of a cell-edge grid with the potential specified at the edges. If you use a cell-centered grid with ghost points, and especially if you change to derivative boundary conditions, this value of ω won't be quite right. But there is still a best value of ω somewhere near the value given in Eq. (9.16) and you can find it by numerical experimentation.

Finally, we come to the question of when to stop iterating. It is tempting just to watch the values of $V_{j,k}$ and quit when the values stabilize at some level, like this for instance: quit when $\epsilon = |V(j,k)_{n+1} - V(j,k)_n| < 10^{-6}$. You will see this error criterion sometimes used in books, but *do not use it.* We know of one person who published an incorrect result in a journal because this error criterion lied. *We don't want to quit when the algorithm has quit changing V; we want to quit when Poisson's equation is satisfied.* (Most of the time these are the same, but only looking at how V changes is a dangerous habit to acquire.) In addition, we want to use a relative (%) error criterion. This is easily done by setting a scale voltage V_{scale} which is on the order of the biggest voltage in the problem and then using for the error criterion

$$\epsilon = \left| \frac{\text{Lhs} - \text{Rhs}}{V_{\text{scale}}} \right| \tag{9.18}$$

where Lhs is the left-hand side of Eq. (9.8) and Rhs is its right-hand side. Because this equation is just an algebraic rearrangement of our finite-difference approximation to Poisson's equation, ϵ can only be small when Poisson's equation is satisfied. (Note the use of absolute value; can you explain why it is important to use it? Also note that this error is to be computed at all of the interior grid points. Be sure to find the maximum error on the grid so that you only quit when the solution has converged throughout the grid.)

And what value should we choose for the error criterion so that we know when to quit? Well, our finite-difference approximation to the derivatives in Poisson's equation is already in error by a relative amount of about $1/(12N^2)$, where N is the smaller of N_x and N_y. There is no point in driving ϵ below this estimate. For more details, and for other ways of improving the algorithm, see *Numerical Recipes*, Chapter 19.

P9.5 Starting with the Gauss-Seidel example above, implement all of the improvements described above to write a full successive over-relaxation routine. Note that you will need to replace the `for` loop on the variable `n`

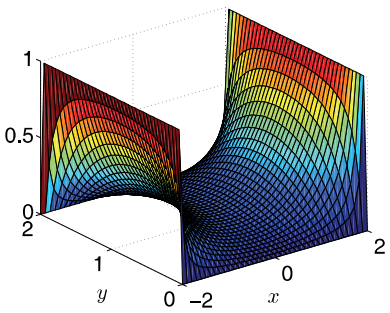

Figure 9.1 The electrostatic potential $V(x,y)$ with two sides grounded and two sides at constant potential.

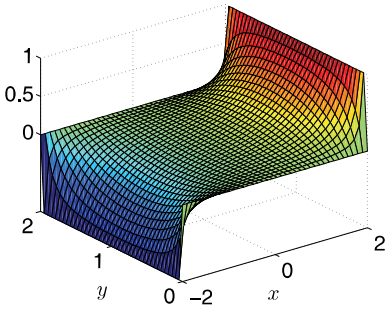

Figure 9.2 The potential $V(x,y)$ from Problem 9.6(a).

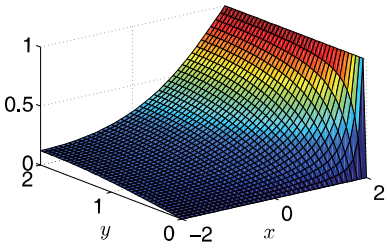

Figure 9.3 The potential $V(x,y)$ with zero-derivative boundary conditions on two sides (Problem 9.6(b).)

into a `while` loop that is based on the error criterion. Also move the plot command out of the loop so it only executes once, after the solution has converged. (So that you don't have to wait for a bunch of plots to be drawn.)

Run your code with $N_x = N_y = 30$ and several different values of ω. Note that the boundary conditions on $V(x,y)$ are $V(-L_x/2, y) = V(L_x/2, y) = 1$ and $V(x,0) = V(x,L_y) = 0$. Set the error criterion to 10^{-4}. Verify that the optimum value of ω given by Eq. (9.16) is the best one to use.

Some Different Geometries

P9.6 (a) Modify your code to model a rectangular pipe with $V(x=-2,y) = -V_0$, $V(x=2,y) = V_0$, and the $y=0$ and $y=L_y$ edges held at $V=0$.

(b) Modify your code from (a) so that the boundary condition at the $x = -L_x/2$ edge of the computation region is $\partial V/\partial x = 0$ and the boundary condition on the $y = L_y$ edge is $\partial V/\partial y = 0$. You can do this problem either by changing your grid and using ghost points or by using a quadratic extrapolation technique (see Eq. (2.10)). Both methods work fine, but note that you will need to enforce boundary conditions inside the main SOR loop now instead of just setting the values at the edges and then leaving them alone.

You may discover that the script runs slower on this problem. See if you can make it run a little faster by experimenting with the value of ω that you use. Again, changing the boundary conditions can change the eigenvalues of the operator. (Remember that Eq. (9.16) only works for cell-edge grids with fixed-value boundary conditions, so it only gives a ballpark suggestion for this problem.)

(c) Study electrostatic shielding by going back to the boundary conditions of Problem 9.6(a), while grounding some points in the interior of the full computation region to build an approximation to a grounded cage. Allow some holes in your cage so you can see how fields leak in. First make a rectangular cage similar to Fig. 9.4, but then you can try other geometries.

You will need to be creative about how you build your cage and about how you make SOR leave your cage points grounded as it iterates. One thing that won't work is to let SOR change all the potentials, then set the cage points back to $V=0$ before doing the next iteration. This takes forever to converge. It is much better to set them to zero and force SOR to never change them. An easy way to do this is to use a cell-edge grid with a *mask*. A mask is an array that you build that is the same size as `V`, initially defined to be full of ones like this

```
mask = np.ones_like(V)
```

Then you go through and set the elements of `mask` that you don't want SOR to change to have a value of zero. (We'll let you figure out the logic to do this for the cage.) Once you have your mask built, you add an `if` statement to our code so that the SOR stuff inside the `j` and `k` for loops only changes a given point and updates the error if `mask(j,k)` is one. This logic assumes you have to set the values of V for these points before the for loop execute, just like the boundary conditions. Using this technique you can calculate the potential for quite complicated shapes just by changing the mask array.

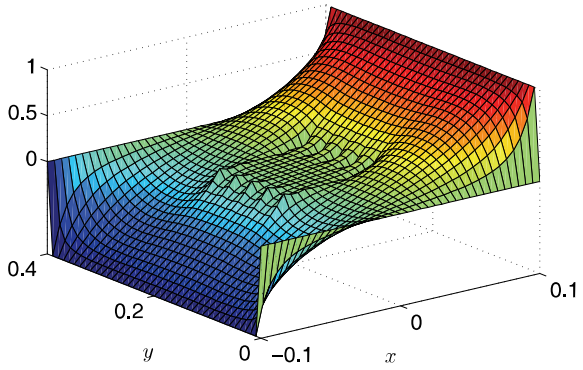

Figure 9.4 The potential $V(x, y)$ for an electrostatic "cage" formed by grounding some interior points. (Problem 9.6(c).)