Electrostatic Potential by the Relaxation Method

Physics 3300, Weber State University, Spring Semester, 2012

The goal of this project is to calculate the electrostatic potential (or "voltage") throughout a region that contains electrodes of various shapes, held at various fixed potentials. For simplicity we will take the region to be two-dimensional. For comparison, you may recall an introductory physics lab in which you *measured* the voltage throughout a two-dimensional region with various electrodes made of conducting paint. This project is the computational counterpart of that experiment.

Theory

The electrostatic potential, $V(x, y)$, is defined to be the function whose negative gradient is the electric field:

$$\vec{E} = -\nabla V, \qquad \text{that is,} \qquad E_x = -\frac{\partial V}{\partial x} \quad \text{and} \quad E_y = -\frac{\partial V}{\partial y}. \tag{1}$$

The electric field, in turn, obeys Gauss's law,

$$\nabla \cdot \vec{E} = \frac{\partial E_x}{\partial x} + \frac{\partial E_y}{\partial y} = \frac{\rho}{\epsilon_0}. \tag{2}$$

In the space between electrodes, the charge density ρ is zero. Then, if we substitute equation (1) into equation (2), we obtain simply

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \nabla^2 V = 0. \tag{3}$$

This is called *Laplace's equation.* It says that if the function $V(x, y)$ is concave-up along one direction, it has to compensate by being concave-down along the perpendicular direction.

To solve Laplace's equation numerically, we need to approximate second derivatives for non-infinitesimal dx and dy. First recall that the accurate way to estimate a *first* derivative at point x is to evaluate the function symmetrically, a little to either side of x:

$$\frac{\partial V}{\partial x} \approx \frac{V(x + \frac{1}{2}\delta) - V(x - \frac{1}{2}\delta)}{\delta}, \tag{4}$$

where δ is an abbreviation for dx (or dy, which will be the same as dx on our square grid). To evaluate the *second* derivative, you can just replace V in this formula with $\partial V/\partial x$. Then use equation (4) again to evaluate each of the two first derivatives in the numerator, noting (for instance) that a half step to either side of $x + \frac{1}{2}\delta$ takes us to $x + \delta$ on one side, and back to x on the other. When the smoke clears, you should find simply

$$\frac{\partial^2 V}{\partial x^2} \approx \frac{V(x+\delta) + V(x-\delta) - 2V(x)}{\delta^2}. \tag{5}$$

(Please derive this equation carefully in your lab report.)

Of course, a similar formula holds for $\partial^2 V/\partial y^2$. Plugging both results into Laplace's equation and canceling the factors of δ in the denominator, you can quickly show (in your lab report) that

$$V(x,y) = \frac{V(x+\delta,y) + V(x-\delta,y) + V(x,y+\delta) + V(x,y-\delta)}{4}. \tag{6}$$

In other words, the potential at any lattice site is simply the *average* of the potentials at the four adjacent sites.

Our goal, then, is to find a function $V(x,y)$ that satisfies equation (6) at every point on our grid, for some given boundary conditions (sites with fixed values of V) that may be geometrically complex.

A simple algorithm for doing this is to start with any set of V values whatsoever (perhaps zero everywhere except on the fixed electrodes), then go through and *set* the V value at each site equal to the average of its four neighbors. Of course, the neighbors will change as well, so this procedure doesn't produce an actual solution, but it should still get us *closer* to the solution. We then repeat the procedure over and over, getting closer to the actual solution with each iteration. This very simple algorithm is called the *relaxation method.* (The relaxation method can easily be adapted to a variety of boundary-value problems, in one or many dimensions, even for differential equations that are more complicated than Laplace's equation.)

Program Design

Use a square Canvas, broken up into smaller grid squares, to display the two-dimensional space of the simulation. Start with a grid of about 20×20 lattice sites, but plan to increase the resolution after you've tested everything. Use a two-dimensional array of type `double` to hold the electrostatic potential values at all the lattice sites. To keep track of arbitrary boundary conditions you can simply use a two-dimensional array of `boolean`s, set to true at each site where the potential is to be held fixed (i.e., each site that is part of an "electrode").

You'll need to decide what range of potential values to work with. This is pretty arbitrary, so do whatever seems to make sense: 0 to 1, or 0 to 100, or -10 to 10, etc. Be sure to document your choice, both in your code and in your lab notebook.

To graphically represent the various potential values, create a one-dimensional array of a hundred or more colors, varying smoothly over some range. The simplest option is to vary the colors from black (to represent the lowest potential value) to white (to represent the highest), but feel free to get creative with hues. You'll need to use `new` once to create the array, then write a loop to create each separate `Color` object (either calling a `Color` constructor method with `new` or calling a static method such as `getHSBColor`). Whatever you do, be sure to explain it in your lab notebook.

Put some temporary data into your array of potentials, then write an appropriate `paint` method to color each square according to the potential at the corresponding site. Test your `paint` method thoroughly. How do you want to draw the electrodes themselves? Explain your decision in your lab report, then implement it and test.

Now that the graphics is working, write a method that executes one iteration of the relaxation algorithm. Here you're suddenly faced with an interesting choice: Should you update all the potential values in place, or store the new values in a temporary array until they've all been calculated? If you update them in place, you'll be using some *new* potential values on the right-hand side of equation (6). That actually makes the algorithm converge a little more rapidly, and it will allow us to implement another trick to speed things up even more. This variation of the relaxation method is called the *Gauss-Seidel algorithm.* (If you were to use all four old values on the right-hand side of equation (6), you would be implementing the *Jacobi algorithm.*)

You'll also need to decide how to treat the boundaries of the simulated region. The simplest choice is to assume that all the sites around the edges are electrodes held at $V = 0$. Also be sure, in executing the relaxation algorithm, not to alter the potential at any other sites that are labeled (through your boolean array) as electrodes.

Create a button that calls your relaxation method just once. (Don't create a thread to call it repeatedly.) Then write some temporary code to initialize your arrays (including the `boolean` array that labels the electrodes) to some nontrivial values, and test everything to make sure it works.

User Interface

By now you're probably eager to try out a variety of electrode shapes. You could, of course, write code to create various types of electrodes, but that quickly gets cumbersome. Instead, this is a good opportunity for you to learn to handle mouse events, so you can simply *draw* the electrodes.

Look up the `MouseListener` and `MouseMotionListener` interfaces in the Java API documention; both are found in the `java.awt.event` package. Also look up the `addMouseListener` and `addMouseMotionListener` methods of the `Component` class. You'll want to add one of each to your (extended) `Canvas`, and the simplest way to do it is to implement them within that class itself (rather than in a separate class or an anonymous inner class). You'll therefore put "`implements MouseListener, MouseMotionListener`" into your class declaration, and insert the lines "`addMouseListener(this);`" and "`addMouseMotionListener(this);`" into the portion of your constructor method that initializes the GUI.

The `SimplePaint` demonstration program provides a useful example of how to work with mouse events.

Whenever a mouse event occurs in your canvas, the appropriate `MouseListener` or `MouseMotionListener` method will be called and passed a parameter of type `MouseEvent`. You can then call the `getX` and `getY` methods of this object to determine the coordinates of the event in screen pixels, relative to the top-left corner of the canvas. From these coordinates, you can calculate the corresponding indices in your array of lattice sites. Think carefully about what you want to happen when you press the mouse button, drag to a different location, and release. Describe the intended behavior clearly in your lab report before you try to implement it in code. Presumably, pressing on a site that isn't already part of an electrode should turn it into one. What should happen if you press on a site, or drag across a site, that is already part of an electrode? Consider using a scrollbar to set the voltage of the electrode that you're drawing.

Besides using the mouse to draw electrodes, you can also use it to extract information. Add code to display the lattice coordinates and voltage at whatever site the mouse is currently over, using either a separate `Canvas` or a `Label` for the data readout. While you're at it, add a readout for the x and y components of the electric field (which you should calculate in a symmetrical way from the adjacent sites to each side). Although you may be tempted to use a high-resolution lattice measuring hundreds of sites across, it's probably best to keep the resolution to 100 across or less, with each site drawn *at least* four or five pixels wide.

Once the mouse interaction is working, you'll probably want to add some further GUI elements, such as a button to clear the electrodes and start over, and a button to run more than one step of the relaxation algorithm at a time. You'll also want a button to reset all the voltages to their starting values (presumably zero) except at the electrode locations, keeping the electrodes in place.

Convergence and Improving the Algorithm

The main weaknesses of your simulation at this point are that it takes a lot of relaxation steps to converge to the final solution, and (worse) it's hard to tell when it has actually converged. Let's address the second problem first.

Add some code to your relaxation algorithm to keep track, during each iteration step, of the maximum amount by which any particular site changes. Display this amount at the end of each step. Also display the number of iterations that you've done so far (since the last reset). Run the simulation for a simple electrode configuration. Observe the results, and make some notes in your lab report. For example, how many iterations does it take before the maximum change is less than some reasonably small amount? (Feel free to add a button to execute relaxation steps until the maximum change is less than your chosen small amount.)

To speed up the convergence, you can try to anticipate future iterations by "over-correcting" with each lattice site update. That is, calculate the *difference* between the average of the four neighbors (using the Gauss-Seidel choice of the new values for two of the neighbors) and the old value of the current site. Call this difference Δ. Then, instead of adding Δ to the old value in order to obtain the new, add Δ times a constant (call it ω) that's somewhat greater than 1. It's hard to determine the best value of this constant in advance, so create a scrollbar that lets you set it to anything between 1 and 2. This idea is called *successive over-relaxation*, often abbreviated *SOR*.

Again using some reasonably simple electrode configuration, first set $\omega = 1$ and check that you get the same behavior as before (with the Gauss-Seidel method). Then increase the value of ω and see what happens. What is the optimum value of ω for your electrode configuration? (Your lab report should include a table of data that documents your answer to this question. Also please include a printout, made from a screen capture, of the electrode configuration that you used for your tests.)

Visualizing the Results

So far you've been using a "density plot" to visualize the electrostatic potential field. Density plots are good and they're often enough, but sometimes it's preferable to use other types of plots. You could try to write Java code to produce other plot types, or you could look for a library of existing Java plotting code. Often, however, the best approach is to export your data into a more specialized computing environment for plotting. One such environment is Mathematica, which provides not only a `ListDensityPlot` function but also `ListContourPlot`, `ListPlot3D`, and `ListVectorPlot`.

To plot your data in Mathematica, you first need to write it to a file. Here's a code fragment that shows how to create a text file and write some data to it:

```
import java.io.*;   // (This goes at the top, of course)

PrintWriter out;
try {
    out = new PrintWriter("mydata.dat");
    out.println("Hello, world!");
    out.println(42 + "\t" + 3.14159);
    out.close();
catch (FileNotFoundException e) {}
```

Notice that the `PrintWriter` class allows you to use the same old `println` method that you already know and love. The "`\t`" is Java's code for a tab, which is often used to separate numbers in data files.

What you want to do is produce a data file in which each row has three numbers: the x and y coordinates of a lattice site, followed by the voltage at that site. Use tabs to separate the three numbers, and use a separate row for each lattice site. When you think it's working, open the data file in your text editor to check it.

Now launch Mathematica and use the following code (more or less) to navigate to your working directory and open your data file:

```
SetDirectory["~/java"]
vData = Import["mydata.dat"]
```

Then look up the documentation for Mathematica's `ListDensityPlot` function and give it a try. Also try `ListContourPlot` and `ListPlot3D`.

To plot the electric field, you can either calculate it in your Java program (and save the data in a separate file) or just calculate it directly in Mathematica. Either way, you'll have to do some Mathematica gymnastics to get it into the right format to use `ListVectorPlot`. Do your best to figure this out, and discuss it with your lab partner, but be sure to ask your instructor for help if you get stuck.

After you've succeeded in plotting both the voltage and the electric field, print images of both for at least three different electrode configurations that seem interesting to you. Then gather up all your notes, images, and Mathematica printouts (including your Mathematica code but *not* lengthy text listings of your data), and turn it all in as your lab report. Don't forget to email your source code, cleaned up and commented to your satisfaction.