Electrostatics with partial differential equations – A numerical example

28th July 2011

This text deals with numerical solutions of two-dimensional problems in electrostatics. We begin by formulating the problem as a partial differential equation, and then we solve the equation by Jacobi's method. We use a package called Cython to improve computational speed.

1 Introduction

So far in the Electromagnetism course, we have looked at Coulomb's law, which describes the force between charged particles:

$$\mathbf{F} = \frac{1}{4\pi\varepsilon_0}\frac{q_1 q_2}{|\mathbf{r}|^3}\mathbf{r}. \tag{1}$$

We have seen that Coulomb's law can be derived from the more general Gauss' law, which is the first of the four Maxwell equations. Gauss' law states that the total *electrical flux* through a closed surface has to be proportional to the electric charge contained within the surface. Gauss' law can be expressed in integral form as follows:

$$\oint_C \mathbf{E} = \frac{Q}{\varepsilon_0} = \frac{1}{\varepsilon_0}\int_V \rho\, dV \tag{2}$$

where C is the surface encapsulatin the volume V, and ρ is the charge density. By applying the Divergence Theorem, we can write equation (2) on *differential form* as

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0} \tag{3}$$

Furthermore, we have defined *electric potential* by the relation

$$\mathbf{E} \equiv -\nabla V. \tag{4}$$

Inserting this relation into (3), we get

$$\nabla \cdot \mathbf{E} = \nabla \cdot (-\nabla V) = -\nabla \cdot \nabla V = -\nabla^2 V = \frac{1}{\varepsilon_0}\rho \tag{5}$$

where

$$\nabla^2 \equiv \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{6}$$

is the Laplace operator in two dimensions.

Now we have an equation relating the electrical potential in a point in space to the charge density in that point. This is a *partial differential equation*, which becomes clear if we write it out as

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = -\frac{1}{\varepsilon_0}\rho(x,y) \tag{7}$$

An equation on this form is known as *Poisson's equation*. If we are able to solve this equation for a given charge distribution, we know what the potential is anywhere in space. By taking the gradient of the potential we can find the electric field.

2 Point charge

We already know some solutions to this equation. For a point charge, the electric field will point radially outward (for a positive charge) or inwards (negative charge). The electric potential will form circular equipotential surfaces around the charge (spherical in three dimensions, of course). The reference value for the potential is arbitrary, and we usually define $V(\infty) = 0$. It is straightforward to find the electric field and the potential for one or several point charges by using Coulomb's law (1).

3 Numerical solution of Poisson's equation

In previous courses, you have worked extensively with ordinary differential equations (ODE), i.e. equations for functions of *one* variable. You have learned how to solve these by computer, using for instance Euler's method, Euler-Cromer or Runge-Kutta. If we are given an ODE with initial conditions, it is pretty straightforward to solve it numerically – it is just a question of computer power.

A partial differential equation (PDE) requires a bit more care. The solution we are after is a scalar field $V(x, y)$, assigning a value to every point on a two-dimensional plane. When we solve this equation numerically, we divide the plane into discrete points (i, j) and compute V for these points.

3.1 Dissecting the mathematics

We remember, e.g. from Euler's method for ODE's, that we solve our differential equation in discrete steps, and use the function's value computed in the previous step to find the value in the next step. With this in mind, it is reasonable to expect that we use the function values in neighbouring points to compute the function value for a PDE as well – and you will see that this is indeed the case. In this example we utilise what is called *Jacobi's method* to solve the Poisson equation for different charge distributions. Briefly explained, the method works as follows: We think of the xy-plane, divided into points, as a matrix. Our goal is to find a matrix V containing the values of V for all points (i, j), corresponding to discrete x and y values. We begin by guessing a matrix $V^{(0)}$, and use our PDE to calculate new values in a matrix $V^{(1)}$ based on the values from $V^{(0)}$, and so on. ((0) og (1) refer to the number of iterations, and must not be confused with exponents!) Hopefully, this process converges towards a matrix V that is a solution to our equation.

But how can we use Poisson's equation to calculate these values? First we need to do a few clever tricks.

The definition of the derivative is, as we know,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{8}$$

It can equivalently be written as

$$f'(x) = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{9}$$

where h is in the last term instead. If h is a small number, then

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \text{ or } f'(x) \approx \frac{f(x) - f(x-h)}{h}. \tag{10}$$

The definition of second derivative is in a similar way

$$f''(x) = \lim_{h \to 0} \frac{f'(x+h) - f'(x)}{h} \tag{11}$$

By inserting the last expression (9) for $f'(x)$ into the expression (11) for $f''(x)$, we get

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{12}$$

For a function of two variables the procedure is the same, but we have to use partial derivatives. We write $\frac{\partial^2 V}{\partial x^2} \equiv V_{xx}$ to save some writing. We then get

$$V_{xx} \approx \frac{V(x+h,y) - 2V(x,y) + V(x-h,y)}{h^2} \tag{13}$$

$$V_{yy} \approx \frac{V(x,y+h) - 2V(x,y) + V(x,y-h)}{h^2} \tag{14}$$

By inserting this in equation (7) and solving for $V(x,y)$ (you should do this), we get

$$V(x,y) \approx \frac{1}{4}\left[V(x+h,y) + V(x-h,y) + V(x,y+h) + V(x,y-h)\right] + \frac{h^2}{4\varepsilon_0}\rho(x,y) \tag{15}$$

or, with discrete indices,

$$V_{i,j} = \frac{1}{4}\left[V_{i,j+1} + V_{i,j-1} + V_{i+1,j} + V_{i-1,j}\right] + \frac{h^2}{4\varepsilon_0}\rho_{i,j} \tag{16}$$

Keep in mind that indices in a matrix are reversed compared to what we are used to from coordinates – i describes which row we are in (y), and j describes which column (x).

This is the expression we need for Jacobi's method: By guessing an initial matrix $V^{(0)}$, we are able to find $V^{(1)}$ by using equation (16) and a computer loop.

But beware, there are some complications to this. If we look at equation (16), we see that we need to insert function values for all neighbouring points. This is fine as long as we are in the middle of the domain, but once we get to the edge we run into trouble. For instance, calculation of $V_{0,0}$ requires knowledge of $V_{-1,0}, V_{-1,-1}$ and $V_{0,-1}$, which are not included in our domain. Thus we do not have an expression for these boundary points, and we have to decide what values they should take prior to starting our algorithm. This is nothing unique to our method, but rather a mathematical consequence of working with a PDE. Our problem does not have a unique solution unless we specify the boundary vaules. This is why mathematical problems with PDE's are often called *boundary value problems*.

3.2 Code

Now that we have a machinery for solving our PDE numerically, it is time to put it to use. We begin by looking at a point charge as described above. Let us say we are looking at a square box with sides of length $L = 1$ (we use dimensionless units). We divide our box into $N = 21$ points and place our point charge in the middle of the box. We then get a charge density matrix ρ which is zero everywhere except in one point: $\rho_{11,11}$. Since we are dealing with discrete points, we can think of $(11, 11)$ as describing an area around the point, and $\rho_{11,11}$ as the charge density in this small (but finite) area.

Let us choose $\rho_{11,11} = 1$. We can always recalculate this to a realistic charge value later by multiplying the result by a factor. As mentioned, we have to guess an initial matrix $V^{(0)}$. A simple choice is the zero matrix. We can significantly lessen the time it takes to get convergence by making a clever guess – since the point charge problem is well-known we have a pretty good idea of what V is going to look like – but the point here is to let numerics do the job for us.

As for the boundary points, we will assume that the potential is zero along the edges. This is not unreasonable – remember that we usually define $V(\infty) = 0$, so if our domain is large then this is a good approximation.

We now have everything we need in order to write a program for solving the problem. We have chosen to write it in Python, example code below.

```
# point_charge.py - Iterative solution of 2-D PDE, electrostatics
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Set dimensions of the problem
L = 1.0
N = 21
ds = L/N

# Define arrays used for plotting
x = np.linspace(0,L,N)
y = np.copy(x)
X, Y = np.meshgrid(x,y)

# Make the charge density matrix
rho0 = 1.0
rho = np.zeros((N,N))
rho[int(round(N/2.0)),int(round(N/2.0))] = rho0

# Make the initial guess for solution matrix
V = np.zeros((N,N))


# Solver
iterations = 0
eps = 1e-8 # Convergence threshold
error = 1e4 # Large dummy error
while iterations < 1e4 and error > eps:
    V_temp = np.copy(V)
    error = 0 # we make this accumulate in the loop
    for j in range(2,N-1):
        for i in range(2,N-1):
            V[i,j] = 0.25*(V_temp[i+1,j] + V_temp[i-1,j] +
                    V_temp[i,j-1] + V_temp[i,j+1] + rho[i,j]*ds**2)
            error += abs(V[i,j]-V_temp[i,j])
    iterations += 1
    error /= float(N)
print "iterations =",iterations

# Plotting
matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'
CS = plt.contour(X,Y,V,30) # Make a contour plot
plt.clabel(CS, inline=1, fontsize=10)
plt.title('PDE solution of a point charge')
CB = plt.colorbar(CS, shrink=0.8, extend='both')
#plt.show()
```

If your try running this program, you will discover that it is a little slow. And we are running it with only 21×21 points – not a good resolution. So, we need to speed it up. There are several possible improvements that can be made, but perhaps the most obvious one is this:

Consider the (hopefully) well-known ODE algorithm Euler-Cromer's method (for second-order equations or higher). It differs from Euler's method in one way: When we have computed the value of the 1st order derivative in a step, this new value is used for computing the 0th order derivative – instead of the previous one as in Euler's method.

This simple idea can be translated to our PDE problem. The algorithm above runs through the points the same way we read text: It starts at the top left corner, and works its way horizontally until it hits the end, then continues from the left on the next row. This means that for all points above and to the left of the current point, we have updated values. If we tell the program to use these values instead of the old ones, we can, hopefully, get faster convergence. Implementing this simply means replacing `V_temp` with `V` in the calculation. The updated solver module is included below.

```
 # Solver - point_charge_improved.py
while iterations < 1e5 and error > eps:
    V_temp = np.copy(V)
    error = 0 # we make this accumulate in the loop
    for j in range(1,N-1):
        for i in range(1,N-1):
            V[i,j] = 0.25*(V[i+1,j] + V[i-1,j] +
                    V[i,j-1] + V[i,j+1] + rho[i,j]*ds**2)
            error += abs(V[i,j]-V_temp[i,j])
    iterations += 1
print "iterations =",iterations
```

But alas, this still is not very fast. There are other ways of improving the algorithm, but they are not really worth the trouble. *The main bottleneck here is Python.* Python is a *high-level language*, meaning that it is easy to write and understand, but comes with lots of built-in systems to protect the user from having to think of everything. These systems produce what is known as *overhead*, and this greatly slows down Python. For this reason, professional simulations are always done in *low-level languages* such as C, C++ or Fortran. These are much faster, but more difficult to use.

4 Introducing: Cython

But luckily, there are brilliant people out there, and a few of them have taken the trouble to write a Python package known as *Cython*. Cython provides a link between Python and C, making it possible to utilise some of C's fastness without actually writing C. The next portion of this text deals with implementing our code in Cython. This will give us a Python program capable of handling quite big problems.

4.1 Installing Cython

The first thing we need to do is install Cython. We assume you are using a Linux computer at UiO with Python installed. (If you are using your personal computer with Ubuntu, look further down.) Begin by pasting the following lines into your terminal and executing them, one at a time:

```
cd ~/
wget http://cython.org/release/Cython-0.14.1.tar.gz
tar xzvf Cython-0.14.1.tar.gz
cd Cython-0.14.1
python setup.py install
```

This will install Cython to a directory in your home folder. You should get an error message during the installation, since you don't have write access to the directory where Python is installed. This requires a little tweak to get around. Paste and execute the following lines:

```
cd ~/
echo "export PATH=~/Cython-0.14.1/bin:$PATH" >> .bashrc
echo "export PYTHONPATH=~/Cython-0.14.1:$PYTHONPATH" >> .bashrc
source .bashrc
```

This adds two lines to your terminal shell configuration and reloads the configuration file, to make bash tell Python that Cython is installed. And with that, we are good to go.

Personal Ubuntu computer: *The procedure is very similar, but since you do have write access you won't need the .bashrc tweak. You just have to make sure that you run the install command as super user. So instead of `python setup.py install`, you should run*

```
sudo python setup.py install
```

Some people have reportedly run into problems when trying to install Cython on Ubuntu with this approach. You may have to install the Python developer version, which you can get in apt by the command

```
sudo apt-get install python-dev
```

4.2 Implementing our code in Cython

What we are looking for is increased speed, but we do not want to write C, or having to make extensive modifications to our Python program. So we consider the program, and realize that almost all runtime is spent in the `while` loop. If we can optimize the loop to make it run with C, then we should get much more speed. Below is the modified Python program to this effect. The changes are highlighted.

```
# point_charge.pyx - Iterative solution of 2-D PDE, electrostatics
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# Cython-specific imports
cimport numpy as np
cimport cython
from libc.math cimport sqrt
ctypedef np.float_t FTYPE_t

@cython.boundscheck(False)
def main():
    # Declare variables
    cdef int N, iterations, i, j
    cdef double L, ds, rho0, eps, error
    cdef np.ndarray[FTYPE_t] x, y
    cdef np.ndarray[FTYPE_t, ndim=2] X, Y, rho, V, V_temp

    # Set dimensions of the problem
    L = 1.0
    N = 101
    ds = L/N

    # Define arrays used for plotting
    x = np.linspace(0,L,N)
    y = np.copy(x)
    X, Y = np.meshgrid(x,y)

    # Make the charge density matrix
    rho0 = 1.0
    rho = np.zeros((N,N))
    rho[int(round(N/2.0)),int(round(N/2.0))] = rho0

    # Make the initial guess for solution matrix
    V = np.zeros((N,N))


    # Solver
    iterations = 0
    eps = 10**(-8.0) # Convergence threshold
    error = 10**(4.0) # Large dummy error
    while iterations < 1e6 and error > eps:
        V_temp = np.copy(V)
        error = 0 # we make this accumulate in the loop
        for j in range(1,N-1):
            for i in range(1,N-1):
                V[i,j] = 0.25*(V[i+1,j] + V[i-1,j] +
                        V[i,j-1] + V[i,j+1] + rho[i,j]*ds**2)
                error += sqrt((V[i,j]-V_temp[i,j])*(V[i,j]-V_temp[i,j]))
        iterations += 1
        error /= float(N)
    print "iterations =",iterations

    # Plotting
    matplotlib.rcParams['xtick.direction'] = 'out'
    matplotlib.rcParams['ytick.direction'] = 'out'
    plt.title('Point Charge')
    CS = plt.contour(X,Y,V,30)
    CB = plt.colorbar(CS, shrink=0.8, extend='both')
    plt.clabel(CS, inline=1, fontsize=10)
    plt.show()
main()
```

So, what has changed? Starting from the top, we see that we have some additional import statements. `cimport` is the Cython import statement, and we need numpy and, of course, Cython, to be imported and understood by the C compiler. The line `from libc.math cimport sqrt` imports C's square root function from the C library – since this function is used in every iteration, it's a good idea to use a fast implementation of it. If, later, you wish to experiment with Cython for other applications, then you may need to explore other functions to be imported this way (e.g. the exponential function `exp`).

Directly below is the line `ctypedef np.float_t FTYPE_t`. This specifies to Cython that we are going to be defining arrays of floats, not integers or strings.

The next line is a little cryptic: `@cython.boundscheck(False)`. This is the magical command. It tells Cython not to bother checking our code for errors (like trying to index lists beyond their length). With Python, such an error produces an error like `Index out of bounds`. With C, all you get is `Segmentation fault`. C knows that something went wrong, but it has not bothered to find out what. It does not help us with debugging, but when we have a working script it greatly reduces the *overhead* discussed above.

The observant reader will notice that we have put our program inside a function `main()`, which is then called at the end of the script. This is simply a workaround neccessary to use Cython. The program works exactly the same.

The last changes are the four lines of `cdef int N, ...`. This is essential! Here we list all variables used in our program, and specify what kinds of objects they are going to be. The first line are the *integers* (notice the counter variables i and j, they are integers too). *Double* is C-language for *float*. `np.ndarray` specifies an array, and here we use the `FTYPE_t` definition from above to specify float. The last line defines matrices, distinguished by `ndim=2`.

4.3 Running Cython

To use our Cython program, it needs to be *compiled*. Begin by saving your Cython script (you may just copy-paste the one above) as a `.pyx` file (as opposed to `.py`!). Then, open an empty document named `setup.py` and paste the following code into it:

```
# setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

numpy = "/local/lib/python2.5/site-packages/numpy/core/include/"

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("point_charge_lib", ["point_charge.pyx"],
                             include_dirs = [numpy])]
)
```

You should replace the file name (“point_charge.pyx”) with the name of your Cython program. The library name (“point_charge_lib”) is arbitrary – you can call it whatever you want – but it should be descriptive. (For the record: The name “setup.py” is arbitrary too, and if your make multiple Cython scripts you might want to have designated setup files – “setup_point_charge.py”, etc.)

Now we are ready to compile. This is done by one line (you obviously have to be in the directory where you saved your files):

```
python setup.py build_ext --inplace
```

With this, Cython translates our Python-esque code into C. Our program can now be run by the command

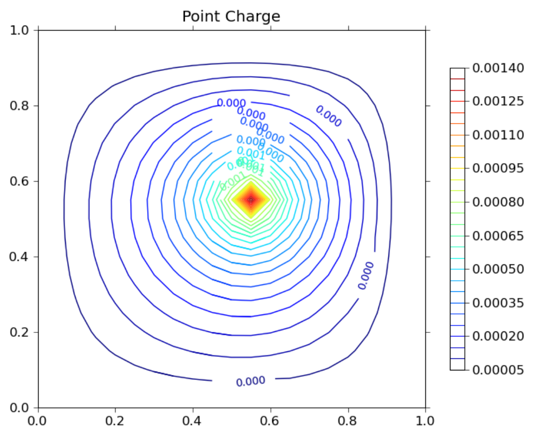
```
python -c "import point_charge_lib"
```

where “point_charge_lib” is your library name from above.

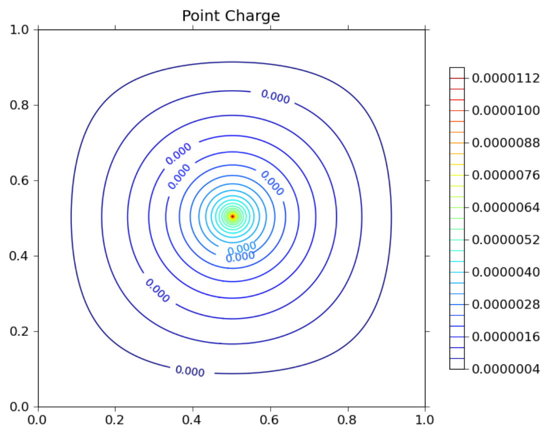
And that’s it! We are now running C without running C. (Or at least without having to think about it.)

5 Analyzing our data

Now that we (finally) have our full machinery up and running, it is time to do some physics. If you have tried to run the Cython program for the point charge, you should get this plot:

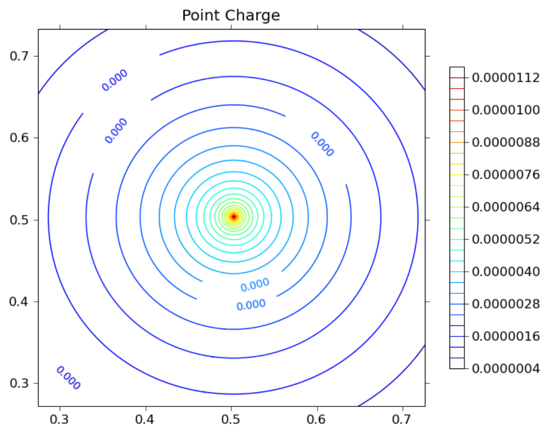


The first thing to notice is the low resolution. The edges of the contour are jagged, and the red area close to the charge is diamond-shaped. Not good. By changing the value of N in our Cython script to $N = 201$ (remember to compile again!), this problem is amended (a resolution as large as this is the reason we are running Cython, as Python would take forever):



The second thing to notice, which is apparent in both plots, is that the contour lines close to the edges are not circular. This is due to a problem with our setup: The point charge problem has circular symmetry. We are using a square box, and have defined the potential to be zero around the edges. **The results close to the edges are unphysical.** Well, not unphysical per se, but they correspond to having the charge inside a grounded box – not in free space, as we wanted. To get it right, we should have used a circular domain.

But the problem only exists close to the edges. Far from the edges, the solution does not depend heavily on the boundary values, and should be usable. We can easily examine this by zooming the plot a bit. To zoom in matplotlib, click the button looking like a cross – this is the pan/zoom utility. Place your mouse pointer on the red dot in the centre, click and hold your right mouse button and drag the pointer upwards and to the right. Zooming beyond the outmost two contour lines, it is looking a lot better:



5.1 Other applications

The point charge is just one example of an electrostatic problem. Another commonly encountered problem is the *capacitor*: Two plates with opposite charge placed close together. Our code is easily modified to study other problems – all we need to do is edit the charge density matrix. Here is a suggestion for modelling the capacitor (to replace the line `rho[int(round(N/2.0)),int(round(N/2.0))] = rho0`):

```
for j in range(round(N/2.0)-int(N/20.0),round(N/2.0)+int(N/20.0)):
        rho[round(N/2.0)-int(N/30.0),j] = rho0
        rho[round(N/2.0)+int(N/30.0),j] = -rho0
```

You should play around with other setups, e.g. a dipole, multiple poles, random charge distributions, etc.