Week 5 assignment on Python: Laplace Equation

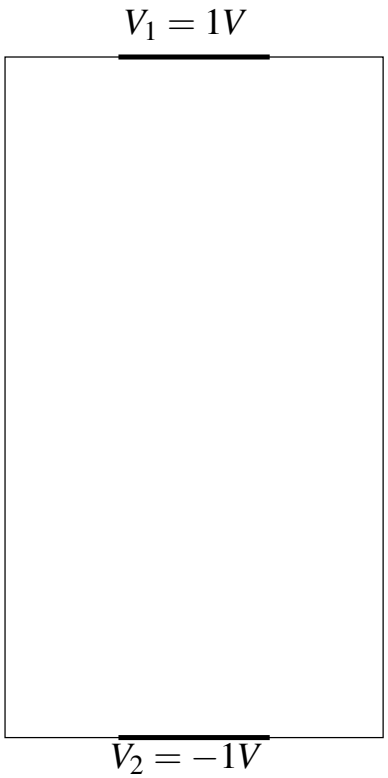
February 7, 2017

1 Introduction

Let us solve for the currents in a resistor. The currents depend on the shape of the resistor and we want to see if $R = \rho L/A$ works or not.

2 The Resistor Problem

A voltage $V_{AB} = 1$V is applied across the terminals of a resistor.



As a result, current flows. The current at each point can be described by a "current density" $\vec{j}$. This current density is related to the local Electric Field by the conductivity:

$$\vec{j} = \sigma \vec{E}$$

Now the Electric field is the gradient of the potential,

$$\vec{E} = -\nabla \phi$$

and continuity of charge yields

$$\nabla \cdot \vec{j} = -\frac{\partial \rho}{\partial t}$$

Combining these equations we obtain

$$\nabla \cdot (-\sigma \nabla \phi) = -\frac{\partial \rho}{\partial t}$$

Assuming that our resistor contains a material of constant conductivity, the equation becomes

$$\nabla^2 \phi = \frac{1}{\sigma} \frac{\partial \rho}{\partial t}$$

For DC currents, the right side is zero, and we obtain

$$\nabla^2 \phi = 0$$

3 Numerical Solution in 2-Dimensions

Laplace's equation is easily transformed into a difference equation. The equation can be written out in Cartesian coordinates

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

Assuming ϕ is available at points (x_i, y_j) we can write

$$\left.\frac{\partial \phi}{\partial x}\right|_{(x_i,y_j)} = \frac{\phi(x_{i+1/2}, y_j) - \phi(x_{i-1/2}, y_j)}{\Delta x}$$

and

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{(x_i,y_j)} = \frac{\phi(x_{i+1}, y_j) - 2\phi(x_i, y_j) + \phi(x_{i-1}, y_j)}{(\Delta x)^2}$$

Combining this with the corresponding equation for the y derivitives, we obtain

$$\phi_{i,j} = \frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}}{4} \tag{1}$$

Thus, if the solution holds, the potential at any point should be the average of its neighbours. This is a very general result and the above calculation is just a special case of it.

So the solution process is obvious. Guess anything you like for the solution. At each point, replace the potential by the average of its neighbours. Keep iterating till the solution converges (i.e., the maximum change in elements of ϕ is less than some tolerance).

But what do we do at the boundaries? At boundaries where the electrode is present, just put the value of potential itself. At boundaries where there is no electrode, the current should be tangential because charge can't leap out of the material into air. Since current is proportional to the Electric Field, what this means is the gradient of ϕ should be tangential. This is implemented by requiring that ϕ should not vary in the normal direction.

4 Code

Import the libraries

First import the libraries needed by python.

2 ⟨*setup* 2⟩≡

```
from pylab import *
import mpl_toolkits.mplot3d.axes3d as p3
```

Define the Parameters

These are the parameters that control the run. They have the following defaults, which should be corrected by the user via commandline arguments (see `sys.argv`)

```
Nx=25; // size along x
Ny=25; // size along y
Nbegin=8;// start of electrode on each side
Nend=17; // end of electrode on each side
Niter=1500; // number of iterations to perform
```

Allocate the potential array and initialize it

First initialize ϕ to zero. Note that the array should have `Ny` rows and `Nx` columns. Also set the top row elements corresponding to the positive electrode to unity. When you are done, ϕ should look like so:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & 0 \end{pmatrix}$$

Perform the iteration

During the iteration, we will use a second potential array, called `oldphi` to hold the values of the previous iteration. This is to keep track of how much the array changed during the current iteration.

You cannot create it by assignment:

```
oldphi=phi
```

The problem is that a python array is a pointer. So the assignment above merely creates a new pointer to the existing data. Changing `oldphi` will change `phi` as well! That is not what we want. We want a new memory area. The way to do this with an assignment operation is to use the `copy` member function:

```
oldphi=phi.copy()
```

To keep track of errors, allocate an error vector, `errors` with `Niter` elements.

Iterate `Niter` times and record the errors. The structure of your for loop should be as follows:

```
for k in range(Niter)
  save copy of phi
  update phi array
  assert boundaries
  errors[k]=(abs(phi-oldphi)).max();
#end
```

Updating the Potential

We first have to update the potential using Eq. 1. This is the central part of the algorithm. How can we do it? In C, it is just a nested pair of for loops:

```
for(i=1 ; i<Ny-1 ; i++ ){ // interior nodes
  for(j=1 ; j<Nx-1 ; j++ ){ // interior nodes
    phinew[i,j]=0.25*(phi[i,j-1]+phi[i,j+1]
      +phi[i-1,j]+phi[i+1,j]);
  }
}
```

This would result in $N_x N_y$ passes through the code and would reduce Python to a crawl. We have to do this same work in a single line of code.

Python subarrays

Note how python specifies sub arrays:

- To extract a sub array of ϕ say $\phi[2,3]$ to $\phi[5,8]$, we use

  ```
  phi[2:4,5:9]
  ```

  Note that the last element of our range is *not included in the subarray*.

- To extract the last ten rows and first eight columns of ϕ we use

  ```
  phi[-10:,0:8]
  ```

  Note that `phi[-10:-1,0:8]` would have got the last ten rows but would have left out the very last row.

With this information, we can now convert the for loops in C to vector operations in Python

Vectorizing For Loops

```
for(i=1 ; i<Ny-1 ; i++ ){ // interior nodes
  for(j=1 ; j<Nx-1 ; j++ ){ // interior nodes
    phinew[i,j]=0.25*(phi[i,j-1]+phi[i,j+1]
      +phi[i-1,j]+phi[i+1,j]);
  }
}
```

The way to "parallelize" this block of C code is to look at the slice of the matrix we work with. We are updating all the interior elements of `phi` that is `phi[1:-1,1:-1]`.

Now consider something rather strange: The matrix of left neighbours of this slice, i.e., the matrix created by `phi[i,j-1]`. Clearly it can be described as the matrix `phi[1:Ny-2,0:Nx-3]`, or equivalently, `phi[1:-1,1:-2]`. Note that the last row of the `phi` is `Ny-1` and not `Ny`.

Proof: The $(i,j)^{\text{th}}$ element of `phi[1:Ny-1,1:Nx-1]` is `phi[i+1,j+1]`. The $(i,j)^{\text{th}}$ element of `phi[1:Ny-1,0:Nx-2]` is `phi[i+1,j]`, which is the left neighbour of `phi(i+1,j+1)`.

Similarly construct the matrix of right neighbours, top neighbours and bottom neighbours. Once this is done, the C code above can be written as a single line:

```
phi[1:-1,1:-1]=0.25*(phi[1:-1,0:-2]
  + phi[1:-1,2:]
  + top neighbours
  + bottom neighbours);
```

This is a matrix equation, which means that the for loops are still there, but they are executed in C within the Python built in functions. Code in this line in Python.

Boundary Conditions

At boundaries where the electrode is present, just put the value of electrode potential itself. At boundaries where there is no electrode, the gradient of ϕ should be tangential. This is implemented by requiring that ϕ should not vary in the normal direction.

The code to enforce the boundaries has to go in after the Poisson update. At the electrode, we do not have to do anything, since the edge row is not changed and the electrode potential is static. But at the other boundaries, we have to make the final row the same as the adjacent row.

For example, at the left boundary, we need to set the normal derivitive of potential to zero. This means that $\partial\phi/\partial x = 0$. This is achieved in C by

```
for( i=1 ; i<Ny-1 ; i++ )
  phi[i,0]=phi[i,1];
```

In Python, we do the same thing by the following vector command:

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

What does this line do? The i^{th} entry of this vector equation says

```
phi[i,0]=phi[i,1]
```

which is what we want. Similarly, for the right side we have

```
for( i=1 ; i<Ny-1 ; i++ )
  phi[i,Nx-1]=phi[i,Nx-2];
```

Convert it to Python code.

For the top and bottom rows, the normal derivitive condition translates to

```
for( i=1 ; i<Nbegin ; i++ ){
  phi[0,i]=phi[1,i];
  phi[Ny-1,i]=phi[Ny-2,i]
}
```

This is for the portion of the boundary to the left of the electrodes. Similar code handles the portion to the right. Convert all of these to Python. **Use Vectorized code!**

For such a simple boundary condition, these techniques are enough. Sometimes the boundary is complex, and identified by a condition. For example, suppose there was an electrode with potential V_0 in the middle of the region, whose shape is defined by

$$(x-x_0)^2+(y-y_0)^2=a^2$$

Then, the indices are not easily specified by hand. You must pull them out with a `where` command. Practice this, since I may give such a problem in the final exam.

Graph the results

First show how the errors are evolving. They should reduce but very slowly. Use a semi-log plot. You will find that a log-log plot gives a reasonably straight line upto about 500 iterations, but beyond that, you get into the exponential regime. This is for a 30 by 30 grid.

To see individual data points, plot every 50^{th} point.

Anytime we have something like an exponential, we should extract the exponent. However, note that it is an exponential decay only for larger iteration numbers. Let us try to fit the entire vector and only the portion beyond 500. Remember that if the fit is of the form

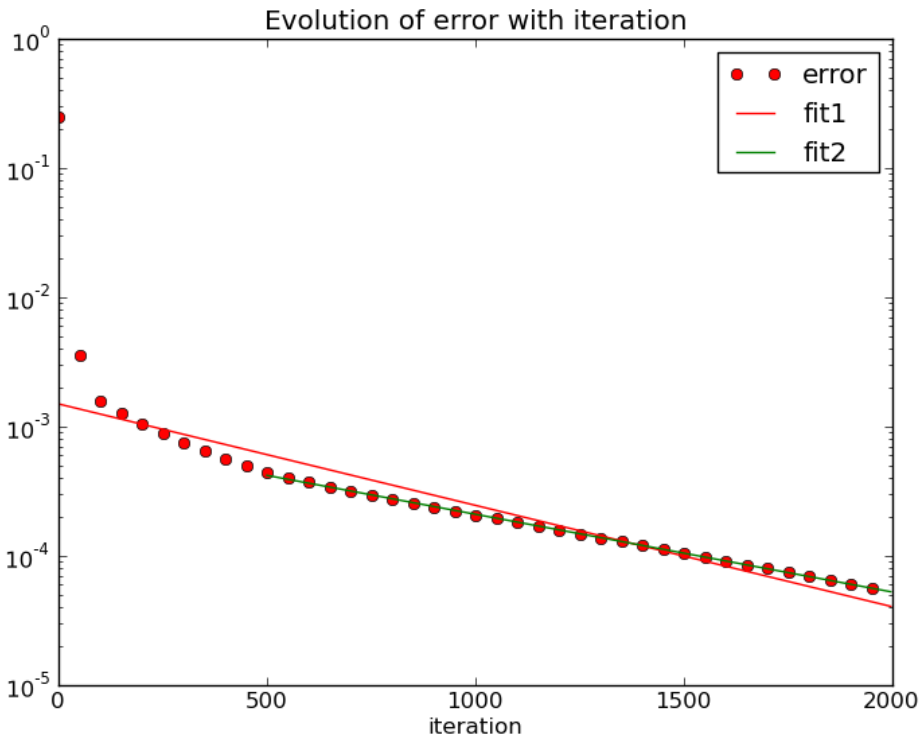
$$y=Ae^{Bx}$$

the thing to do is to take the log. Then we have

$$\log y=\log A+Bx$$

That is why it looks like a straight line in a semi-log plot.

Extract the above fit for the entire vector of errors **and for those error entries after the** 500^{th} **iteration**. Plot the fit in both cases on the error plot itself. use `legend` to label the curves (errors, fit1, fit2).



The red line shows the fit you get with all the data, while the green line uses only the later points. Clearly excluding the early iteration data gives a far better fit. This is an important lesson to learn. Fit the region where the fit works, not the entire data. A second lesson to learn is that the early iterations aren't convergence at all, even though the error is going down. Only when the errors start showing "asymptotic" behaviour, i.e., where we can predict what will happen if we go another thousand iterations, can we know where to stop.

Stopping Condition

So where should you stop? Or more precisely, what is the error? The maximum error scales as

$$\text{error} = Ae^{Bk}$$

where k is the iteration number. for our fit, For one choice of values, I got

$$\text{error}_k = 0.0014 \exp(-0.00226k)$$

Summing up the terms, we have

$$\begin{aligned} \text{Error} &= \sum_{k=N+1}^{\infty} \text{error}_k \\ &< \sum_{k=N+1}^{\infty} Ae^{Bk} \\ &\approx \int_{N+0.5}^{\infty} Ae^{Bk} dk \\ &= -\frac{A}{B} \exp(B(N+0.5)) \end{aligned}$$

Note that the inequality comes because the errors at any iteration could be either sign. So some of these errors *could* cancel. We look for a worst case result and sum up the absolute values. For our values, this expression evaluates to

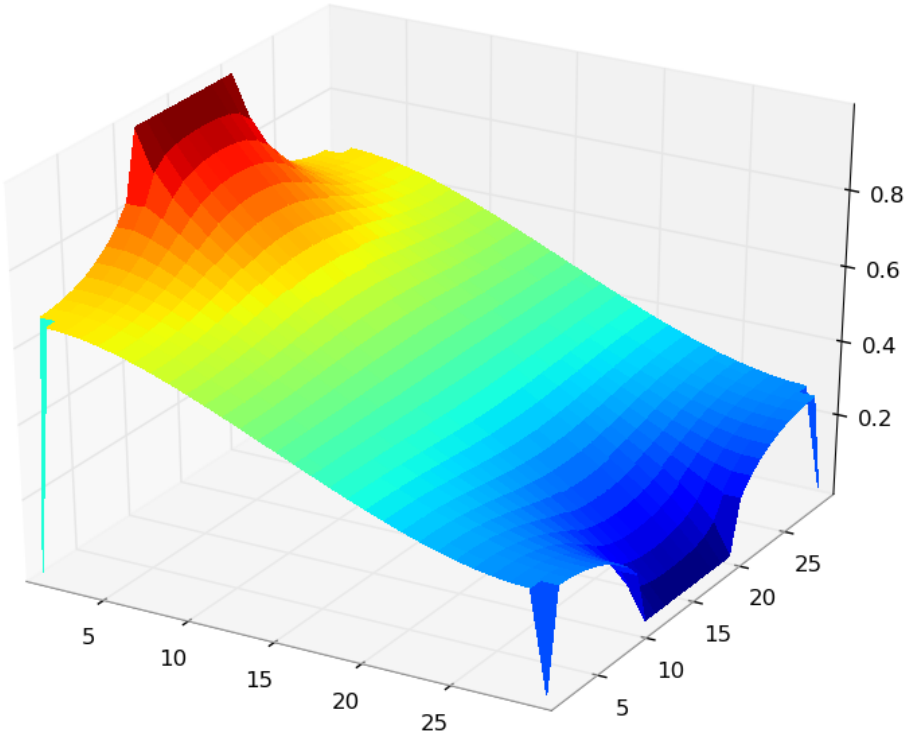
$$\text{Error} < 0.63 \times 0.03 = 0.02$$

Note that the last per-iteration change was 0.0000474, which shows you that that is extermely misleading. Take a look at the error plot in detail. The error went from 6×10^{-4} to 1.5×10^{-4} in 500 iterations. As you know, in exponential decay, what matters is the "half life" or the time constant. For this error vector, the time constant is $1/B = 442$. That explains the difference, since $0.02/442 = 0.0000452$. The profile was changing very little every iteration, but it was continuously changing. So the cumulative error was still large.

A general comment on the algorithm: This method of solving Laplace's Equation is known to be one of the worst available. This is because of the very slow coefficient with which the error reduces.

Surface Plot of Potential

Next do a 3-D plot of the potential. For the default values I got the following plot (the boundary conditions were slightly different with $V_2 = 0$):



To create this plot, you need the following lines in your code:

```
fig1=figure(4)      # open a new figure
ax=p3.Axes3D(fig4) # Axes3D is the means to do a surface plot
x=arange(1,Nx+1)    # create x and y axes
y=arange(1,Ny+1)
X,Y=meshgrid(x,y)   # creates arrays out of x and y
title('The 3-D surface plot of the potential')
surf = ax.plot_surface(Y, X, phi, rstride=1, cstride=1, cmap=cm.jet,lin
```

`plot_surface` does the real job of plotting. You can play with the options and see what they do.

The current is primarily going from electrode to electrode, but it spreads in the middle.

Contour Plot of the Potential

Obtain a contour plot of the potential. For this look up the `contour` function

Vector Plot of Currents

Now obtain the currents. This requires computing the gradient. The actual value of σ does not matter to the shape of the current profile, so we set it to unity. Our equations are

$$\begin{aligned} j_x &= -\partial\phi/\partial x \\ j_y &= -\partial\phi/\partial y \end{aligned}$$

This numerically translates to

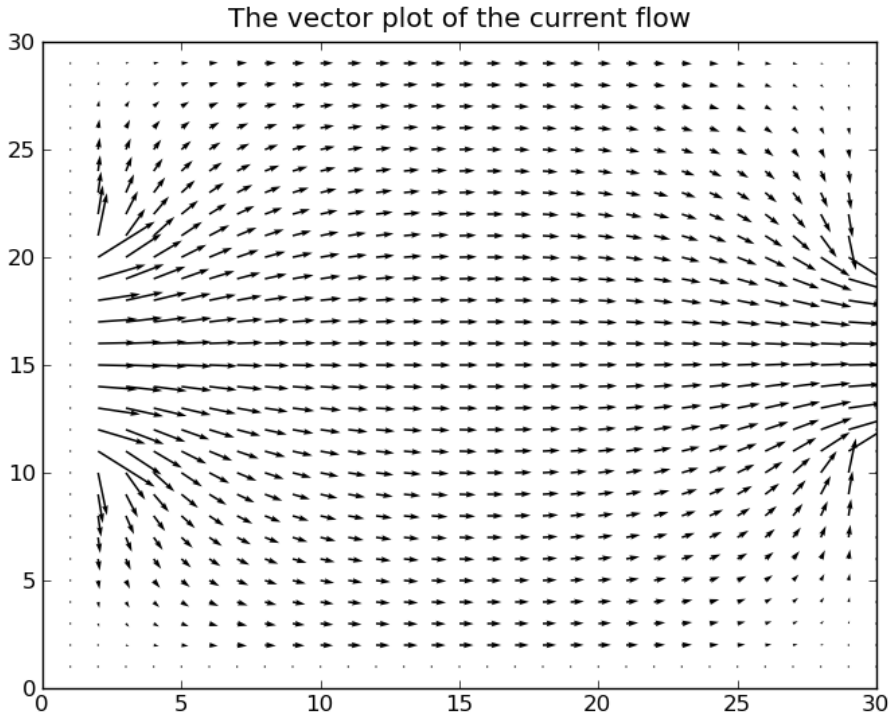
$$\begin{aligned} J_{x,ij} &= \frac{1}{2}\left(\phi_{i,j-1} - \phi_{i,j+1}\right) \\ J_{y,ij} &= \frac{1}{2}\left(\phi_{i-1,j} - \phi_{i+1,j}\right) \end{aligned}$$

Create the arrays `Jx`, `Jy`. Then call the `quiver` command:

```
quiver(y,x,Jy.transpose(),Jx.transpose())
```

What this does is to create the vector plot in the desired direction. Python tries to be helpful and tries to create the plot with x horizontal and y vertical. So this plot will show the current going from left to right, as seen in the figure.

Plot the current density using **quiver**. Here is what I got:



The current quickly spreads out to fill the entire cross-section and then flows in the direction of the negative electrode. There is a corresponding gathering of current at the negative electrode since current can only enter or leave via the electrodes.

Resistance

To study the resistance of the object, calculate the total current injected. This is got by summing the currents at the positive and negative terminals. If the simulation is working correctly, you should have as much current flowing out of the positive terminal as is flowing into the negative terminal. This is a second, independent test of the accuracy of the simulation. For my simulation, I got after 1500 iterations on a 25 by 25 grid

```
Niter=2000: Iavg=0.64506, Idiff=0.13220
```

In the above line, `Iavg` is the average of the entering and leaving currents, while `Idiff` is the difference between the entering and the leaving currents. If the code has converged, `Idiff` should have gone to zero. After 1500 iterations, we still have a significant `Idiff`. What this says is that we have a very poor quality algorithm! We actually need even more iterations to get the error down. In fact, the error is greater than the 0.02 that was predicted by analysing the error trace.

The lesson here is that error analysis is a chancy business. Despite our best efforts we can end up wrongly estimating the error in the simulation.

Comparison with Theory

The problem you have simulated is a "very difficult" problem. If the electrodes covered the entire top and bottom surfaces, we can then solve the problem easily, and the answer is:

$$\phi = V_0 \frac{L-y}{L} \tag{2}$$

This is got from solving Laplace's equation with

$$\begin{aligned}
\phi(0,x) &= V_0 \\
\phi(L_y,x) &= 0 \\
\frac{\partial\phi}{\partial y}(y,0) &= 0 \\
\frac{\partial\phi}{\partial y}(y,L_x) &= 0
\end{aligned}$$

If you assume

$$\phi = X(x)Y(y)$$

we obtain

$$\begin{aligned}
X'' + k^2 X &= 0 \\
Y'' - k^2 Y &= 0
\end{aligned}$$

where k is not known. Solving we find

$$\begin{aligned}
k &= 0 \\
Y(y) &= Ay + B \\
X(x) &= \text{Constant}
\end{aligned}$$

which gives the above solution.

But when the electrode covers only a part of the top and bottom surfaces, the boundary condition becomes what is called a "mixed" boundary condition, and there are no nice analytic answers.

Verification

At this point you should have a vectorized code or it will take hours to run the remaining runs.

- **Turn your solver into a function, with the electrode size, N_x and N_y as arguments.**

Once you have done that, call the function for the case of the electrode covering the entire top and bottom surfaces. Obtain the potential and compare it to Eq. (2). Determine the difference by computing

$$\varepsilon_2 = \frac{1}{N_x N_y} \sum_{i,j} \left(\phi_{ij} - V_0 \frac{L - y_i}{L} \right)^2$$

which is the least squares deviation. I have divided by $N_x N_y$ since that would make the ε_2 ideally independent of the number of points in my simulation.

Now vary N_x and N_y both, keeping the ratio fixed. How does ε_2 change? Does it approach zero? Explain deviations.