Assignment No 5

Pranav Phatak, EE19B105

March 24, 2021

Objective

The aim of this assignment is to obtain the solution to potential in a region by solving *Laplace's* equation in 2-dimensions.

Numerical Solution to Laplace's equation

Laplace's equation in 2-dimensions can be written in Cartesian coordinates as

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\partial^2 \phi}{\partial y^2} \tag{1}$$

An approximate solution for the above equation for a 2-dimensional grid of points would be

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

ie. the solution at any point is a sum of values at neighbouring points. The algorithm implemented makes use of the above equation to update ϕ over many iterations till it converges within an acceptable error.

The Potential Solution

Along with the update mentioned in Eqn.1 we also need to take care of the boundary conditions imposed on the system. The following code blocks update the value of ϕ over `Niter` iterations, along with the error profile :
1. The function to update voltage matrix

```
#function to update voltage matrix for every iteration
def new_phi(phi_matrix,phi_old):
    phi_matrix[1:-1,1:-1]=0.25*(phi_old[1:-1,0:-2]+ phi_old[1:-1,2:]+ phi_old[0:-2,
    return phi_matrix
```

2. The function which sets the boundary conditions for voltage matrix

```
#function to set the boundary conditions on voltage matrix for every iteration
def voltage_boundary_conditions(phi_matrix):
    phi_matrix[:,Nx-1]=phi_matrix[:,Nx-2] # Right Boundary
    phi_matrix[:,0]=phi_matrix[:,1] # Left Boundary
    phi_matrix[0,:]=phi_matrix[1,:] # Top Boundary
    phi_matrix[Ny-1,:]=0
    center = np.where(X**2+Y**2<(0.35)**2)
    phi_matrix[center]=1.0  #Make electrode voltage as 1V
    return phi_matrix
```

3. The function which finds error profile

```
#Initialize error matrix
error = np.zeros(Niter)
#Iterate and update the voltage matrix

for iteration in range(Niter):
    phi_old = phi_matrix.copy()
    phi_matrix = new_phi(phi_matrix,phi_old)  #Update matrix
    phi_matrix = voltage_boundary_conditions(phi_matrix)    #Set boundary condition
    error[iteration] = np.max(np.abs(phi_matrix-phi_old))    #Error between old and
    if(error[iteration] == 0):  #Break if error reaches steady state
        break
```

The initial potential configuration of the system can be visualised using the `pylab.contourf()` function :

```
#plot initial potential
plt.figure(3)
plt.title("Initial potential 2D contour")
plt.xlabel(r"X$\longrightarrow$")
plt.ylabel(r"Y$\longrightarrow$")
plt.contourf(X,Y,phi_matrix,cmap=plt.get_cmap('coolwarm'))
plt.colorbar()
plt.savefig("3.jpg")
plt.close()
```

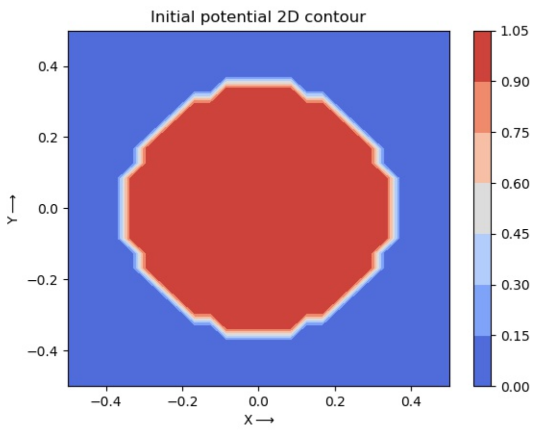

Figure 1: Plot of the initial potential configuration

Error in estimation and best fit

The error function appears to vary exponentially with *Niter*. We attempt to extract the exponent of this dependence. In particular, we attempt to fit a function of the form

$$y = Ae^{Bx}$$

Thus ,

$$logy = logA + Bx$$

$logA$ and B can be estimated using the least squares method. The following code block accomplishes the same :

```
def fit_exponential(y,Niter,iteration_start):
    log_error = np.log(error)[-iteration_start:]
    X = np.vstack([(np.arange(Niter)+1)[-iteration_start:],np.ones(log_error.shape)]
    log_error = np.reshape(log_error,(1,log_error.shape[0])).T
    return s.lstsq(X, log_error)[0]
```

The parameters A and B are obtained in two ways :
1. Fitting the entire error vector (*fit1*)
2. Fitting only the points beyond 500 iterations(*fit2*)

Using a log-log plot, and considering only every 50th point for the sake of better visualization, we obtain the following plot for the best fit and error on same scale :

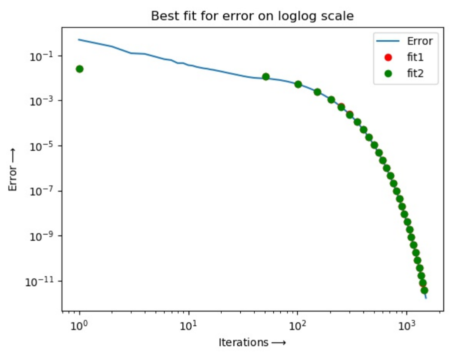

Figure 2: Log-log plot of Error and best fits vs number of iterations

Similarly, the following is the semilogy plot :

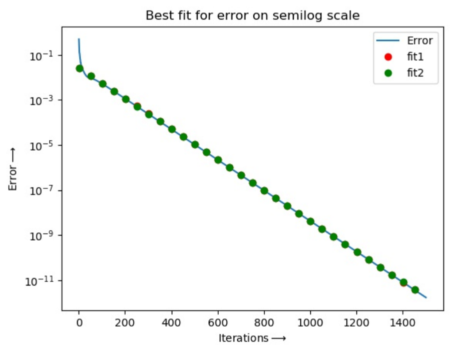

Figure 3: Semilogy plot of Error and best fits vs number of iterations

Notice that the plots almost exactly coincide, with differences hardly visible.

Stopping condition

The upper bound for the error estimated with each iteration is given by

$$\text{Error} = -\frac{A}{B}exp(B(N + 0.5))$$

This can be plotted with the number of iterations *Niter* . The following graph is obtained :

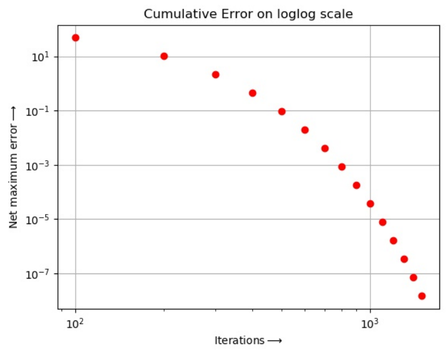

Figure 4: Cumulative Error vs iterations

Surface plot of potential

The contour plot of ϕ is obtained using the `pylab.contourf` function.

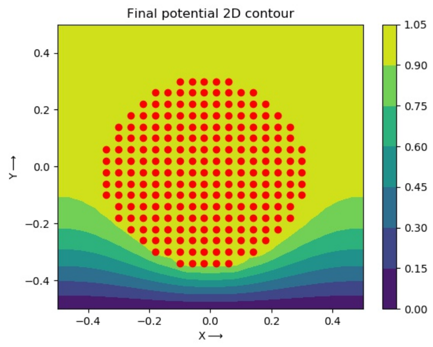

Figure 5: Contour plot of ϕ

After running the update for 1500 iterations, we obtain the following 3-D plot for the potential using the `plot_surface` function :

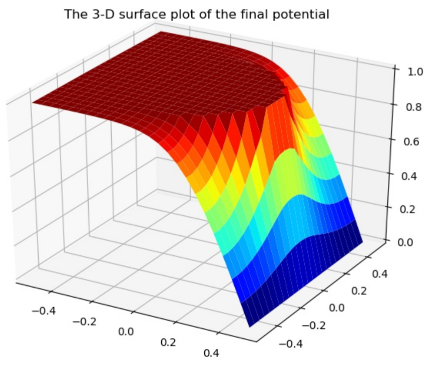

Figure 6: 3-D surface plot of ϕ after *1500* iterations

Vector plot of Currents

The currents in the system in Cartesian form can be expressed as :

$$J_x = -\frac{\partial \phi}{\partial x}$$

$$J_y = -\frac{\partial \phi}{\partial y}$$

Numerically, this can be expressed as

$$J_{x,ij} = \frac{\phi_{i,j+1} - \phi_{i,j-1}}{2}$$

$$J_{y,ij} = \frac{\phi_{i+1,j} - \phi_{i-1,j}}{2}$$

The following code block evaluates the currents J_x and J_y and plots them:

```
 Jx,Jy = (1/2*(phi_matrix[1:-1,0:-2]-phi_matrix[1:-1,2:]),1/2*(phi_matrix[:-2,1:-1]
plt.figure(7)
plt.title("Vector plot of current flow")
plt.quiver(Y[1:-1,1:-1],-X[1:-1,1:-1],-Jx[:,::-1],-Jy)
x_electrode,y_electrode=np.where(X**2+Y**2<(0.35)**2)   #Points with electrode
plt.plot((x_electrode-Nx/2)/Nx,(y_electrode-Ny/2)/Ny,'ro')  #Mark those points as re
plt.savefig("7.jpg")
plt.close()
```

The current density vector is now plotted using the `quiver` function. The following plot is obtained :

From the current density plot, we notice that hardly any current flows through the top part of the wire. With a little thought, we observe that the lower surface being grounded, the easiest way for charge carriers to flow from the electrode would be directly through the lower half of the wire, thus avoiding a longer, more resistive path through the top half of the wire.

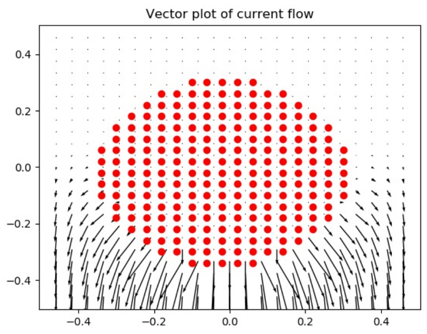

Figure 7: Current density profile

Temperature variation along the wire

Most of the current is in the narrow region at the bottom. That is what will get strongly heated. The equation for temperature is:

$$\nabla \cdot (\kappa \nabla T) = q = \frac{|J|^2}{\sigma} \tag{2}$$

For simplicity, we will assume κ and σ to be equal to 1 while finding the temperature values. The code for that is:

```
#laplaces equation
def new_temperature(temp,tempold,Jx,Jy):
    temp[1:-1,1:-1]=0.25*(tempold[1:-1,0:-2]+ tempold[1:-1,2:]+ tempold[0:-2,1:-1] +
    return temp
```

We shall now plot these values in a 3D plot, the code for the same is:

```
#initialize temperature matrix
temp=300 * np.ones((Nx,Ny),dtype = float)

#Iterate and update the temperature matrix
for k in range(Niter):
    tempold = temp.copy()
    temp = new_temperature(temp,tempold,Jx,Jy)   #Update step
    temp = temperature_boundary_conditions(temp)    #Set boundary condition after ev

#plotting 3d contour of final temp
fig10=plt.figure(8)
ax=p3.Axes3D(fig10)
plt.title('The 3-D surface plot of the temperature')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Temperature')
ax.plot_surface(X, Y, temp , rstride=1, cstride=1, cmap=plt.cm.jet,linewidth=0, anti
plt.savefig("8.jpg")
plt.close()
```

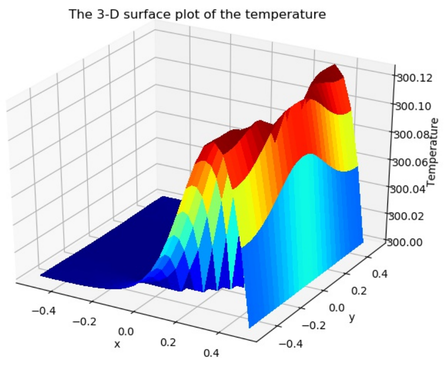

Figure 8: 3D temperature plot

Conclusion

We got to learn how to solve problems by vectorization instead of using nested for loops.
Using a finite differentiation approximation, we have found a solution to Laplace's equation for a given system. The error is seen to decay at a highly gradual pace. Thus the chosen method of solving Laplace's equation is inefficient. On analysing the quiver plot of the currents, it was noticed that the current was mostly restricted to the bottom of the wire, and was perpendicular to the surface of the electrode and the conductor.