Assignment 5: Laplace Equation and the Resistor Problem

J.Phani Jayanth - EE19B026

March 18, 2021

Abstract

This week's Python Assignment involves the following:

- Using Laplace equation to solve for Electric Potential and Currents in a resistor of a specific shape
- Observing how the errors evolve and fitting (linear approximation) the error data
- Plotting the Surface Plot and Contour Plot of the Electric Potential
- Plotting the Vector Plot of Current flow

Introduction

We aim to calculate the Electric potential using the Laplace equation:

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

where ϕ is the Electric potential

Laplaceś equation is easily transformed into a difference equation. In Cartesian coordinates, it can be written as:

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

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

$$\frac{\partial\phi}{\partial x}@(x_i, y_i) = \frac{\phi(x_{i+1/2}, y_j) - \phi(x_{i-1/2}, y_j)}{\Delta x}$$

and

$$\frac{\partial^2\phi}{\partial x^2}@(x_i, y_i) = \frac{\phi(x_{i+1}, y_j) - 2\phi(x_i, y_i) + \phi(x_{i-1}, y_j)}{(\Delta x)^2}$$

Combining this with the corresponding equation for the y derivatives, we obtain:

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

This implies that, the potential at any point is the average of its neighbours. So, we guess a solution and then iterate this further, till it converges with a certain tolerance and thereby obtain an accurate solution.

A wire is soldered to the middle of a copper plate and is held at 1 Volt. One side of the plate is grounded, while the remaining are floating. We aim to solve this model with our python code.

Defining parameters and Initializing the Potential array

Parameters:

The parameters that control the run are Nx (size along X-axis), Ny (size along Y-axis), Radius of Electrode (Central Lead) and the number of iterations. The default values are as follows:

```
Nx=25   #size along x
Ny=25 #size along y
radius=8  #radius of central lead
Niter=1500  #number of iterations to perform
```

The user can also provide the above parameters. This is done through the code as follows:

```
import sys
```

```
elif (len(sys.argv))==5:
  try:
    Nx = int(sys.argv[1])   #size along x
    Ny = int(sys.argv[2])   #size along y
    radius = float(sys.argv[3])   #radius of central lead
    Niter = int(sys.argv[4])      #number of iterations to perform
  except Exception:
    print(f'ERROR! Wrong Usage!!\nCorrect usage: python3 {sys.argv[0]} Nx Ny
   radius Niter'/n)
    print('Nx= size along X\nNy= size along Y\nradius = Radius of central lead\
   nNiter= Number of iterations to perform')
    exit()
```

Initializing the Potential array:

We initialized the potential at all points to zero before proceeding to obtaining the solution. Then we apply the boundary conditions at the boundaries of the electrode and also at the boundaries of the copper plate. The initialized array has Ny rows and Nx columns. The code for this is as follows:

```
#Allocating the potential array and initializing it:
phi= np.zeros([Ny,Nx])
```

To assign a value of 1V to the potential at points in the electrode (central lead), we identify the points inside the circular region of specified radius and allocate to these points, a value of 1V. This is done as follows:

```
#To find central circular region:
x= np.arange(-(Nx-1)/2,(Nx-1)/2+1,1)
y= np.arange((Ny-1)/2,-(Ny-1)/2-1,-1)
Y,X = np.meshgrid(y,x)
#coordinates inside the circle
ii= np.where(X*X +Y*Y <= radius*radius )
phi[ii]=1.0  #Allocating phi for electrode region
```

A contour plot of this potential is then plotted (Figure 1). the red dots represent the points in the central lead, at a potential of 1V.

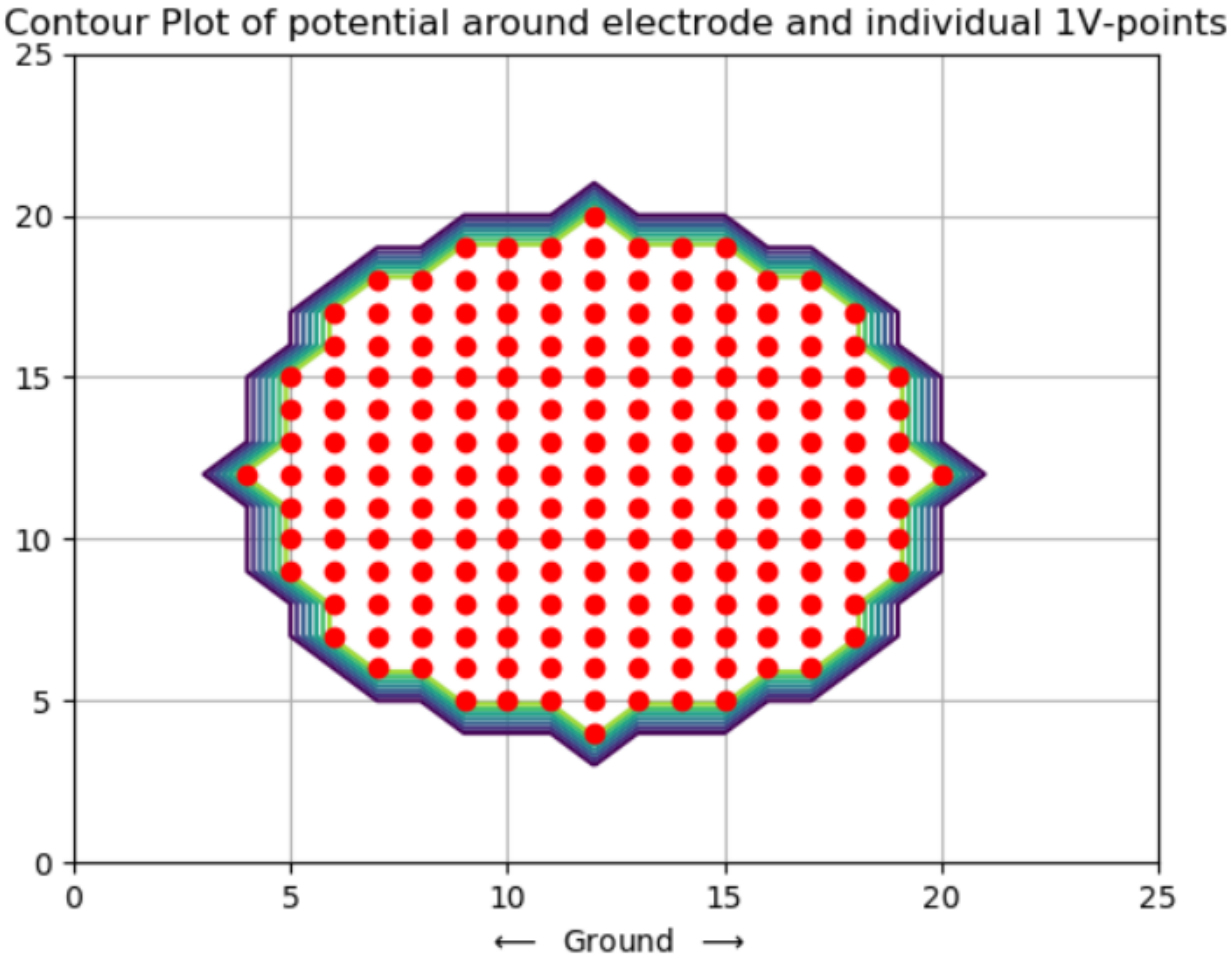

Figure 1: Contour plot of the Electric potential

The code is as follows:

```
#Plotting the contour plot of potential:
plt.contour(np.arange(0,Nx,1),np.arange(0,Ny,1),phi)
plt.title('Contour Plot of potential around electrode and individual 1V-points')
plt.plot(ii[1],ii[0],'o',color='r')
plt.xlim(0,Nx)
plt.ylim(0,Ny)
plt.grid()
plt.xlabel('$\longleftarrow$  Ground  $\longrightarrow$')
plt.show()
```

Performing an Iteration:

Each iteration has the following steps:

- **Saving a copy of phi(ϕ)**:
  To hold the values of the previous iteration, we make a copy of it and store it as follows:

  ```
  for i in range(Niter):
    #Saving copy of phi
    oldphi = phi.copy()
  ```

- **Updating the phi(ϕ) array**:
  We update each element of the array to the average of its neighbours (as derived above).

  ```
  #Updating phi array
  phi[1:-1,1:-1] = (phi[2:,1:-1]+phi[:-2,1:-1]+phi[1:-1,2:]+phi[1:-1,:-2])/4
  ```

- **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. Therfore, for the edges of the copper plate, potential shouldn't change normal to the boundary. We also make sure that the potential at all the points inside the electrode (central lead) is 1V.

  ```
  #Asserting boundaries
  phi[1:-1,0] = phi[1:-1,1] #left
  phi[1:-1,-1] = phi[1:-1,-2] #right
  phi[0,:] = phi[1,:] #top
  phi[ii]=1.0
  ```

- **Finding and storing the maximum error in each iteration**:
  We expect the error to reduce after each iteration. To observe this, we store the error obtained in each iteration and plot them accordingly. The code for this is as follows:

```
#Allocating an error vector:
errors = []
# For each iteration
# Appending the maximum error in this iteration
errors.append((abs(phi-oldphi)).max())
```

Plotting the Error Plots:

```
#Plotting errors
plt.plot(np.arange(0,Niter,1),errors[::1])
plt.title('Errors in each Iteration')
plt.xlabel('Iteration number $\longrightarrow$')
plt.ylabel('Error $\longrightarrow$')
plt.grid()
plt.show()

plt.semilogy(np.arange(0,Niter,1),errors[::1], label='All points')
plt.semilogy(np.arange(0,Niter,50),errors[::50],'o',label='50th points')
plt.title('Errors in each Iteration: Semilogy')
plt.xlabel('Iteration number $\longrightarrow$')
plt.ylabel('log(Error) $\longrightarrow$')
plt.legend()
plt.grid()
plt.show()

plt.loglog(np.arange(0,Niter,1),errors[::1])
plt.title('Errors in each Iteration: Loglog Plot')
plt.xlabel('log(Iteration number) $\longrightarrow$')
plt.ylabel('log(Error) $\longrightarrow$')
plt.grid()
plt.show()
```

The plots are obtained are shown in Figures 2, 3 and 4

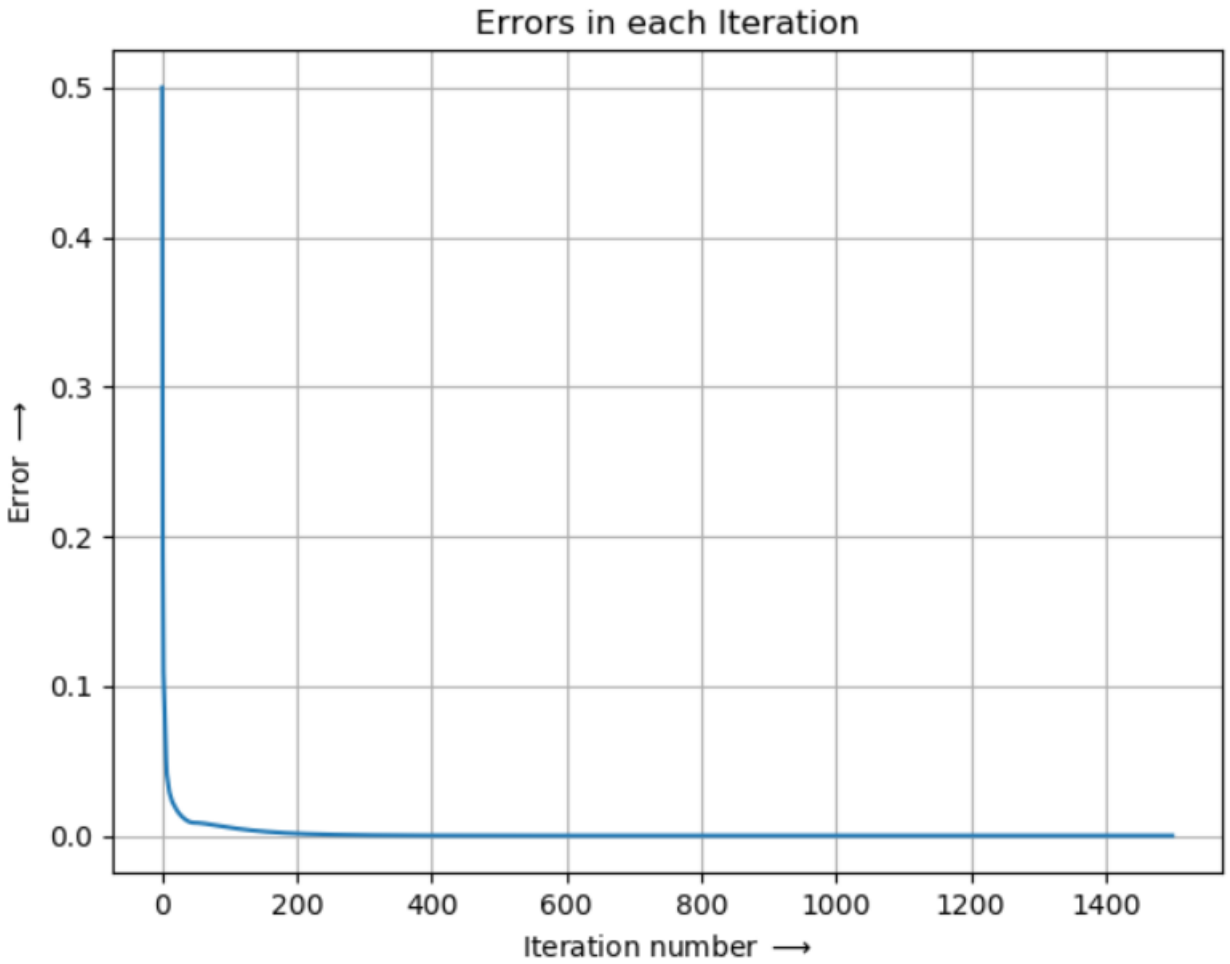

Figure 2: Error obtained in each iteration vs Number of Iterations

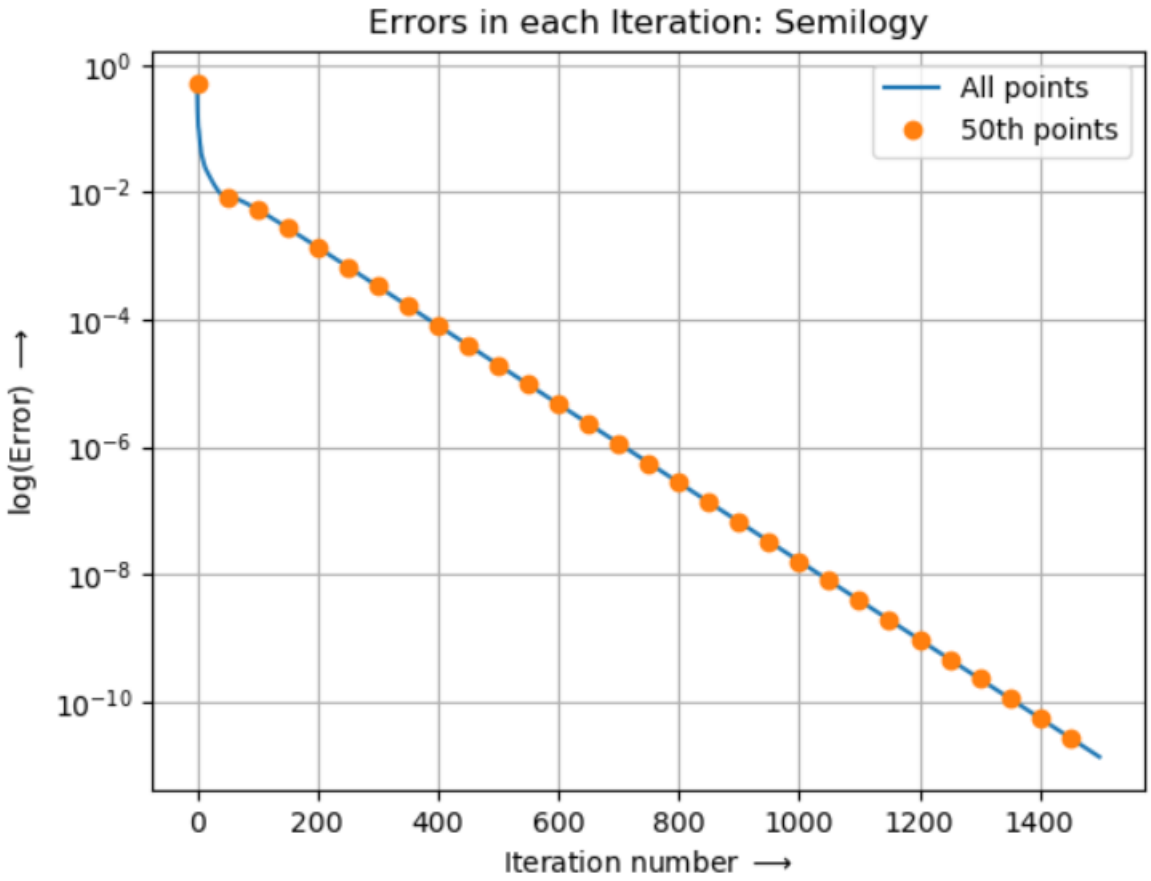

Figure 3: Semilogy: log(Error) from each iteration vs Number of Iterations

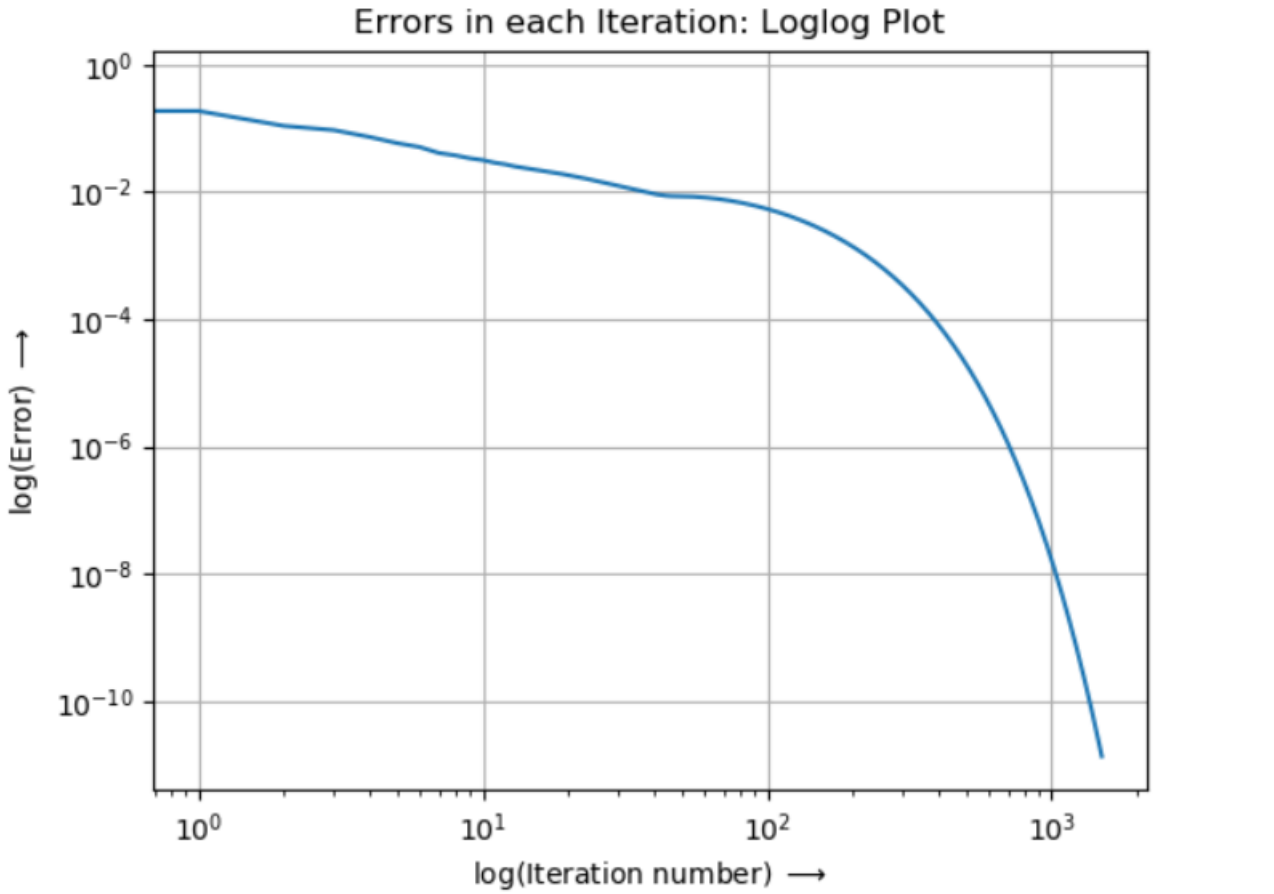

Figure 4: Error obtained in each iteration vs Number of Iterations: Loglog Plot

From the plots, we can observe that the error reduces, but very slowly. The loglog plot of Error vs Number of iterations is fairly linear upto about 200 iterations (for 25x25 grid), but beyond that falls rapidly (Figure 4).

We also plot the error obtained after every 50 iterations with semilog-Y axis. This is plotted along with the error vs Number of Iterations (semilog-Y), with every 50th error data point plotted using circles.

The semilogy plot is a straight line - which means that the error has exponential variation. Therefore, we can express the error as an exponential function:

$$y = A.e^{Bx}$$

Taking log on both sides, we get:

$$logy = logA + Bx$$

Fitting the Error:

We make use of the *polyfit()* funtion from the *numpy* library to fit the Error (log) to a straight line. We get the $logA$ and B values from this, and we perform this twice:

- **Fit 1:** For error obtained after each iteration i.e., for the enitre vector of errors.
- **Fit 2:** For error entries after the 500th iteration.

This is done as follows:

```
#Removing the points where the error goes to zero (log(0) is not finite)
errors_new=[]
Niter_new=0
for error in errors:
  if error != 0:
    errors_new.append(error)
    Niter_new+=1
#Finding best fit for entire vector of errors:
try:
  B1,logA1 = np.polyfit(np.arange(0,Niter_new,1), np.log(errors_new), 1)
except Exception:
  print("ERROR! Couldn't solve the equation for Fit1: Fit for the entire vector
    of errors")
#Finding best fit for error entries after 500th iteration:
try:
  B2,logA2 = np.polyfit(np.arange(500,Niter_new,1), np.log(errors_new[500:]), 1)
except Exception:
  print("ERROR! Couldn't solve the equation for Fit2: Fit for error entries after
     500th iteration")
```

The plot is obtained as follows: **Figure 5**

```
#Plotting fit1, fit2 and errors
plt.semilogy(np.arange(0,Niter,1),np.abs(errors),'k--',label='Errors')

plt.plot(np.arange(0,Niter,1), np.exp(B1*np.arange(0,Niter,1)+logA1), 'y',label='
    Fit1')

plt.plot(np.arange(500,Niter,1), np.exp(B2*np.arange(500,Niter,1)+logA2),'r',
    label='Fit2')

plt.title('Best fits (Fit1, Fit2) and actual Errors in each iteration: Semilogy')
plt.xlabel('Iteration number $\longrightarrow$')
plt.ylabel('log(Error) $\longrightarrow$')
plt.grid()
plt.legend()
plt.show()
```

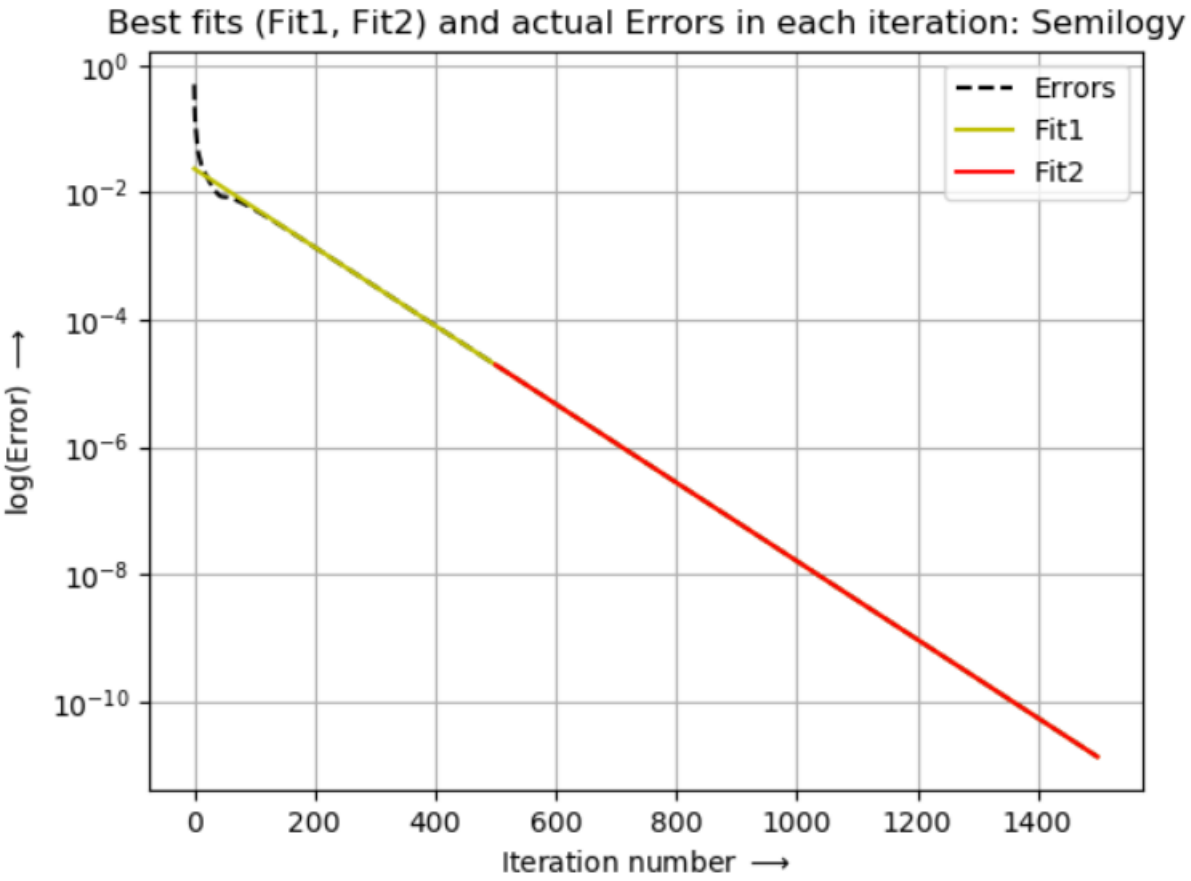

Figure 5: Errors, Fit 1 and Fit 2: Semilogy

Surface Plot and Contour Plot of Electric Potential

We plot the Surface plot of Potential at each point in a 3-D plot, to visualize it. Contour Plot for the Potential is also plotted. To obtain a 3-D Surface Plot, we will need to import the following:

```
import mpl_toolkits.mplot3d.axes3d as p3
```

```
#Plotting 3-D plot of the potential:
figure = plt.figure(4)
ax = p3.Axes3D(figure)
plt.title('3-D Plot of the Potential')
surf = ax.plot_surface(Y, X, phi.T,rstride=1, cstride=1, cmap='jet',linewidth=0,
    antialiased=False)
plt.xlabel('$\longleftarrow$ Ground $\longrightarrow$')
plt.ylabel('y $\longrightarrow$')
plt.show()
```

```
#Contour plot of potential:
c = plt.contour(np.arange(0,Nx,1),np.arange(0,Ny,1),phi[::-1,:],levels=np.arange
    (0,1,0.125))
plt.clabel(c,np.arange(0,1,0.25),inline=True)
plt.plot(ii[0],ii[1],'o',color='r',label='Central lead: 1V region')
plt.title('Contour plot of potential')
plt.xlim(0,Nx)
plt.ylim(0,Ny)
plt.legend()
plt.show()
#Plotting 1V electrode points
plt.plot(ii[0],ii[1],'o',color='r',label='Central lead: 1V Region')
```

The plots obtained are as follows:

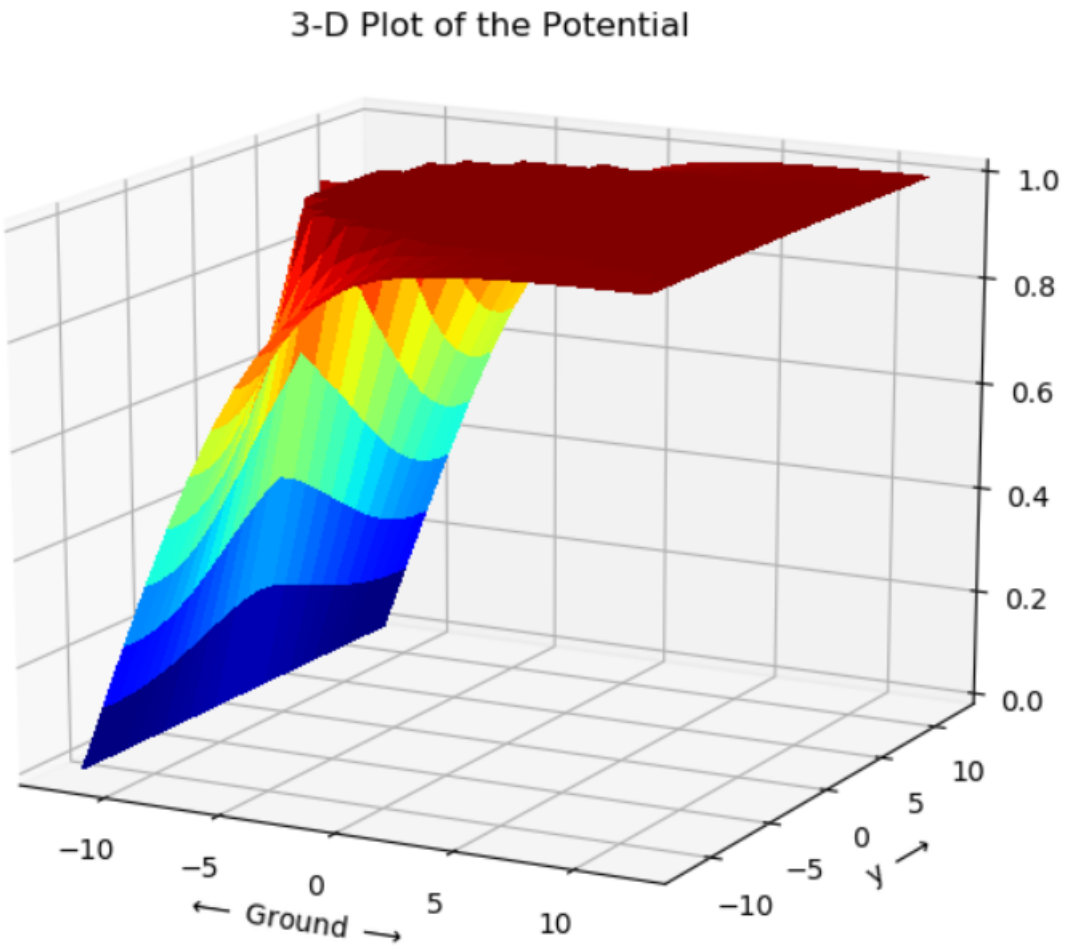

Figure 6: 3D Surface Plot of Electric Potential

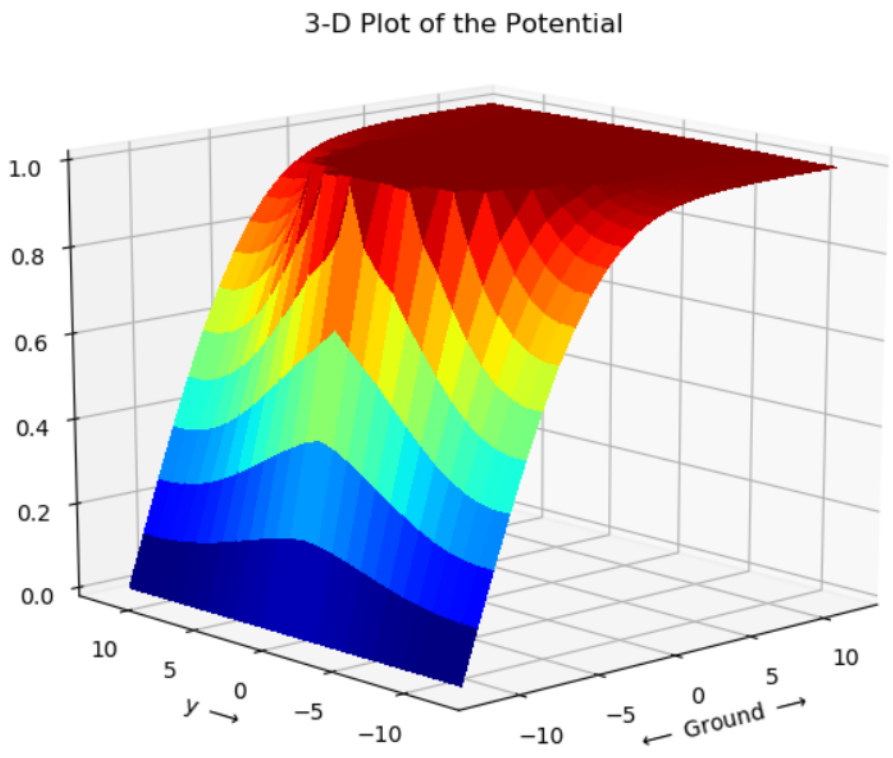

Figure 7: 3D Surface Plot of Electric Potential

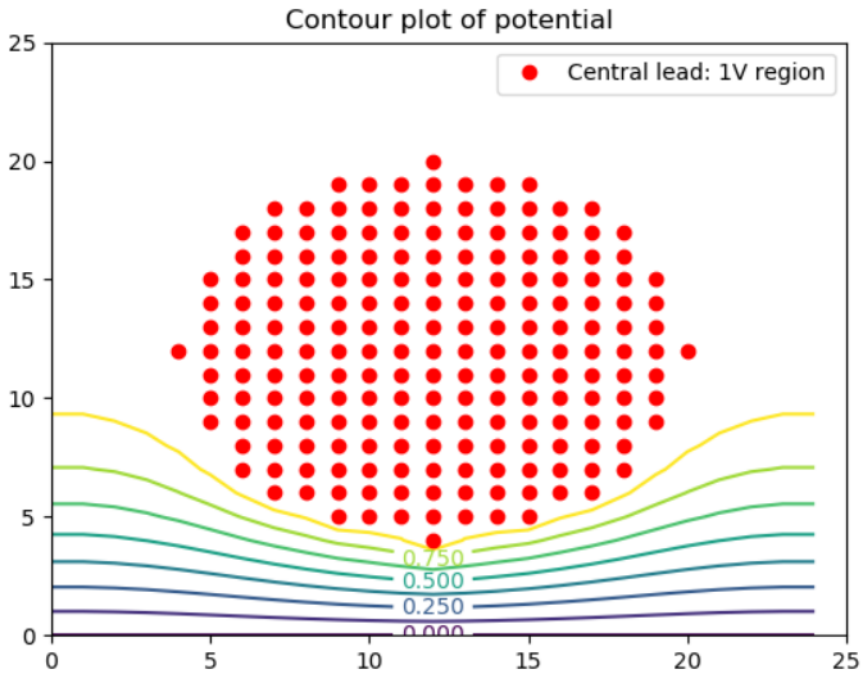

Figure 8: Contour Plot of Electric Potential

Vector Plot of Currents

We now obtain the currents from the following equations:

$$j_x = -\frac{\partial \phi}{\partial x}$$
$$j_y = -\frac{\partial \phi}{\partial y}$$

(σ is set to unity, as its value does not matter to the shape of the current profile)

This numerically translates to:

$$j_{x,i,j} = \frac{1}{2}(\phi_{i,j-1} - \phi_{i,j+1})$$
$$j_{y,i,j} = \frac{1}{2}(\phi_{i-1,j} - \phi_{i+1,j})$$

We plot the vector plot of currents using the *quiver()* function from the *matplotlib* library.

Electric field, and hence the currents are flowing from the electrode into the ground. There is almost no current close to the top edge, because this edge is located between the resistor and air, and charge cannot flow through air. Current always flows through the path of least resistance.

The code is as follows:

```
#Defining the directions of Current density:
Jy=np.zeros([Ny,Nx])
Jx=np.zeros([Ny,Nx])
Jy[1:-1,1:-1]= (phi[0:-2,1:-1]-phi[2:,1:-1])/2
Jx[1:-1,1:-1]= (phi[1:-1,0:-2]-phi[1:-1,2:])/2
```

```
#Vector plot of Current densities:
plt.quiver(np.arange(0,Nx,1),np.arange(0,Ny,1),Jx[::-1,:],-Jy[::-1,:],scale=5,
    label='Current Density')
plt.title('Vector Plot of Current flow')
plt.legend(loc='upper right')
plt.show()
```

The plot obtained is shown in **Figure 9**

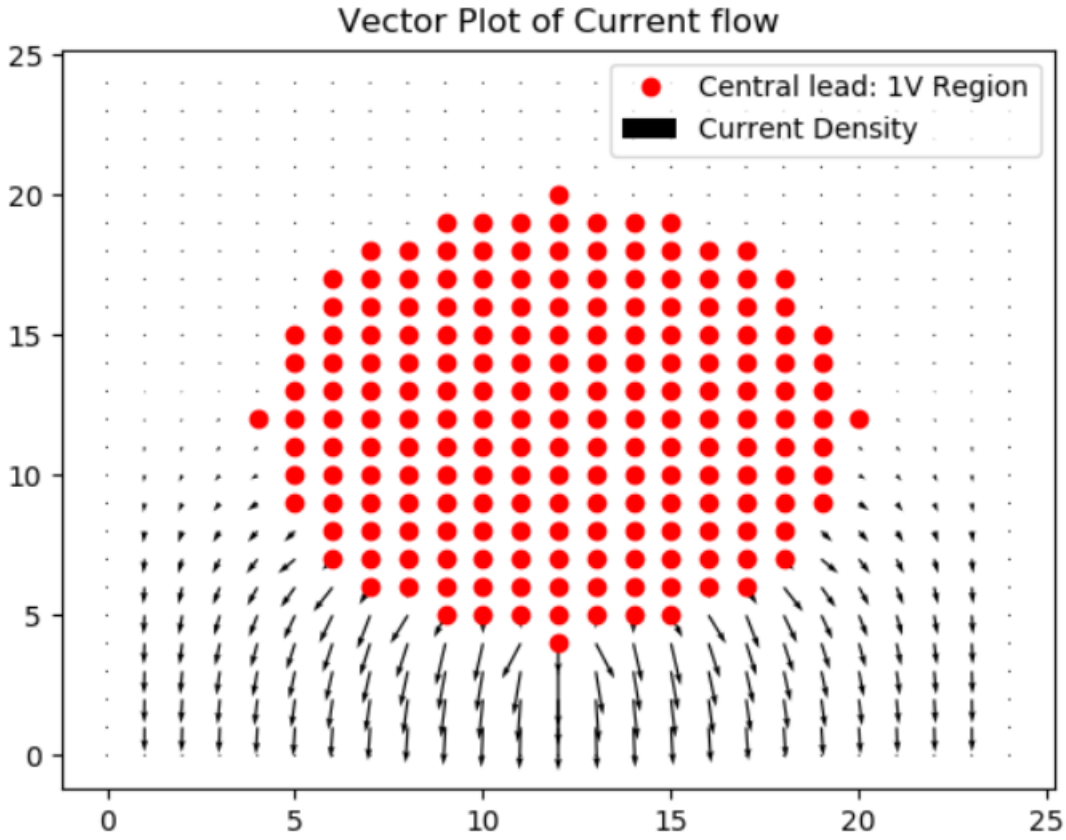

Figure 9: Vector Plot of Currents

Conclusion

We solved the Resistor Problem - Solving for currents and Potentials in a Resistor. We obtained the relevant plots required to visualize these parameters using *matplotlib* and *numpy*. We also have learnt how to plot 3-D Surface Plots. Run-time is reduced and efficiency is increased by the use of *vectors*.
From the plots, we observed that the error after each iteration is decreasing exponentially. Using this observation, we fit the error to an exponential function Ae^{Bx}. We used the function *polyfit* from the *numpy* library to obtain the best fit.