EE2703: Applied Programming Lab
Assignment 5: Laplace Equation

Soham Roy
EE20B130

March 11, 2022

1 Introduction

The currents in a resistor will be evaluated. The currents depend on the shape of the resistor. The part of the resistor likely to get hottest is also visualised.

A wire is soldered to the middle of a copper plate and its voltage is held at 1 Volt. One side of the plate is grounded, while the remaining are floating. The plate is 1 cm by 1 cm in size.
By the conductivity:

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

As the Electric field is the gradient of the potential:

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

Continuity of charge yields:

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

Combining these equations:

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

Assuming that the resistor consists of a material of constant conductivity:

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

For DC currents the RHS is zero, thus:

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

2 Subquestions

2.1 Import the Libraries

The following libraries have been imported:

```
import sys
import argparse
import numpy as np
import matplotlib.pyplot as plt

from scipy.linalg import lstsq
```

sys and argparse are used to parse the command line arguments.
numpy and matplotlib are used for numerical computation and plotting.
scipy.linalg.lstsq is used to extract best fits for the linear system.

2.2 Define the Parameters

The following parameters have been defined:

```
Nx, Ny, radius, Niter = parse_cmdline_args()  # define the parameters
```

Here, parse_cmdline_args() is a function that parses the command line arguments and returns the parameters as either user inputs or their default values. It employs the following code:

```
try:  # try converting directly to values
    return [
        int(sys.argv[1]) if len(sys.argv) >= 2 else 25,
        int(sys.argv[2]) if len(sys.argv) >= 3 else 25,
        int(sys.argv[3]) if len(sys.argv) >= 4 else 8,
        int(sys.argv[4]) if len(sys.argv) >= 5 else 1500,
    ]
except ValueError:  # otherwise, parse keyword arguments
    parser = argparse.ArgumentParser(description="Resistor simulation")

    parser.add_argument(
        "-x", "--Nx", type=int, default=25, help="size along x axis"
    )
    parser.add_argument(
        "-y", "--Ny", type=int, default=25, help="size along y axis"
    )
    parser.add_argument(
        "-r", "--radius", type=int, default=8, help="radius of central lead"
    )
    parser.add_argument(
        "-n", "--Niter", type=int, default=1500, help="number of iterations"
    )

    args = parser.parse_args()

    return args.Nx, args.Ny, args.radius, args.Niter
```

2.3 Initialize the Potential Array

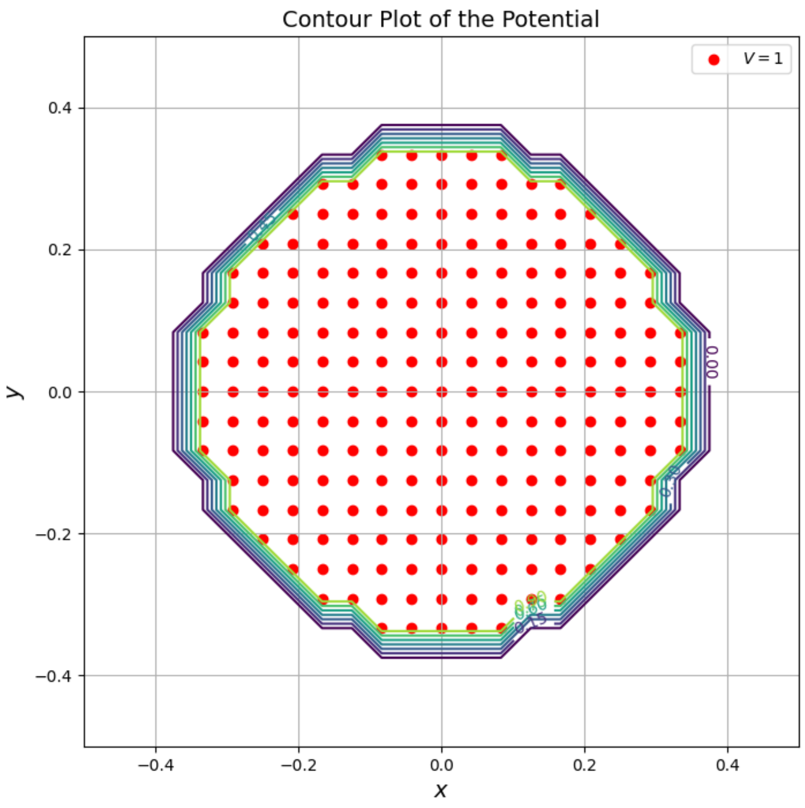
The potential array ϕ has been generated by:

```
phi = np.zeros((Ny, Nx))  # allocate the potential array
X, Y = np.meshgrid(np.linspace(-0.5, 0.5, Nx), np.linspace(-0.5, 0.5, Ny))

# scaled radius = 0.35 for default args, 5% margin for floating point errors
ii = np.where(X ** 2 + Y ** 2 <= (1.05 * radius / (min(Nx, Ny) - 1)) ** 2)
phi[ii] = 1.0  # initialize the potential array
```

The contour_plot() function obtains a contour plot of the potential, with the nodes in the $V = 1$ region marked red, by the following:

```
plt.clabel(plt.contour(X, Y, phi))
plt.scatter(X[0, ii[1]], Y[ii[0], 0], color="r", label="$V=1$")
```



2.4 Perform the Iterations

The iterate() function realizes the following logic:

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

Through this loop:

```
for k in range(Niter):
    oldphi = phi.copy()

    # updating the potential
    phi[1:-1, 1:-1] = (  # update interior points to average of neighbors
        phi[1:-1, :-2] + phi[1:-1, 2:] + phi[:-2, 1:-1] + phi[2:, 1:-1]
    ) / 4

    # boundary conditions
    phi[1:-1, 0] = phi[1:-1, 1]  # left
    phi[1:-1, -1] = phi[1:-1, -2]  # right
    phi[-1, :] = phi[-2, :]  # top
    phi[ii] = 1.0  # central area corresponding to electrodes to 1

    errors[k] = np.max(np.abs(phi - oldphi))
```

Subsections 2.5 and 2.6 explain the updating of the potential and the boundary conditions respectively.

2.5 Updating the Potential

The potential is updated according to the following equation:

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

This is accomplished by the following code:

```
phi[1:-1, 1:-1] = (  # update interior points to average of neighbors
    phi[1:-1, :-2] + phi[1:-1, 2:] + phi[:-2, 1:-1] + phi[2:, 1:-1]
) / 4
```

2.6 Boundary Conditions

The boundary condition at the left boundary, for example, is:

$$\frac{\partial \phi}{\partial x} = 0$$

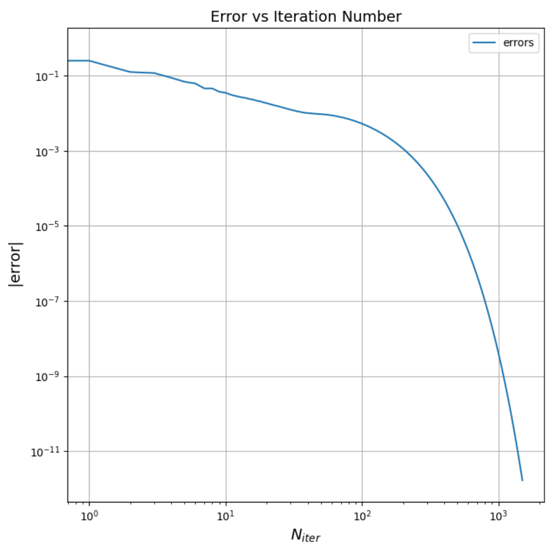
This is accomplished by the following code:

```
phi[1:-1, 0] = phi[1:-1, 1]  # left
phi[1:-1, -1] = phi[1:-1, -2]  # right
phi[-1, :] = phi[-2, :]  # top
phi[ii] = 1.0  # central area corresponding to electrodes to 1
```

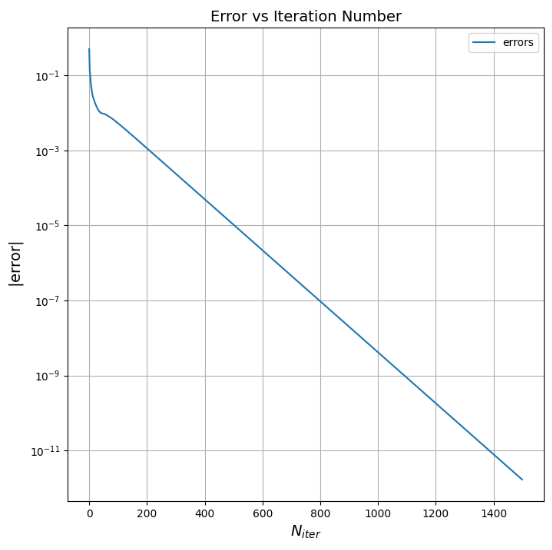
The potential of the nodes corresponding to the electrode region has been ensured to be 1.

2.7 Graph the Results

The plot_errors () function plots the errors vs iteration number:

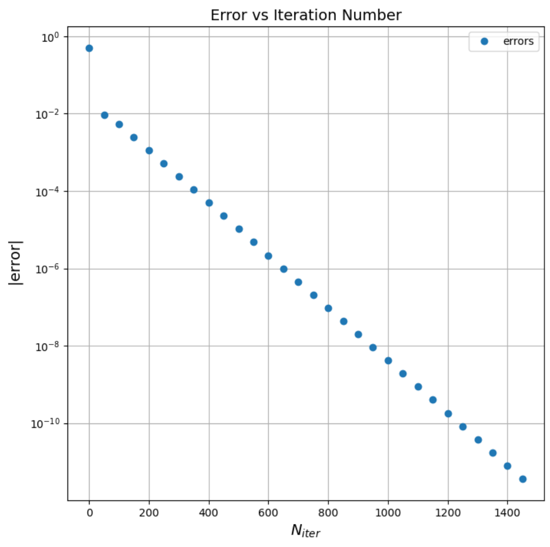


log-log scale



semilog scale

To see individual data points, every 50th point has been plotted:



The data may be fit to an exponential, especially for larger iteration numbers (i.e. beyond 500):

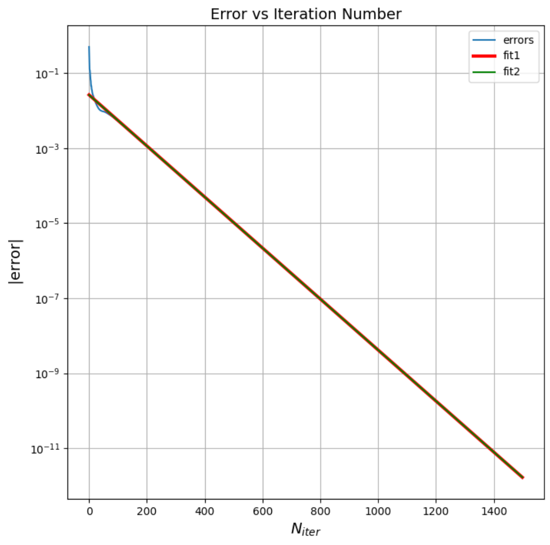
$$y = Ae^{Bx} \implies \log y = \log A + Bx$$

The code used for this is:

```
fit1 = lstsq(np.c_[np.ones(len(errors)), n], np.log(errors))[0]
fit2 = lstsq(np.c_[np.ones(len(errors)), n][500:], np.log(errors)[500:])[0]

plt.semilogy(np.exp(fit1[0] + fit1[1] * n), "r", lw=3, label="fit1")
plt.semilogy(np.exp(fit2[0] + fit2[1] * n), "g", label="fit2")
```

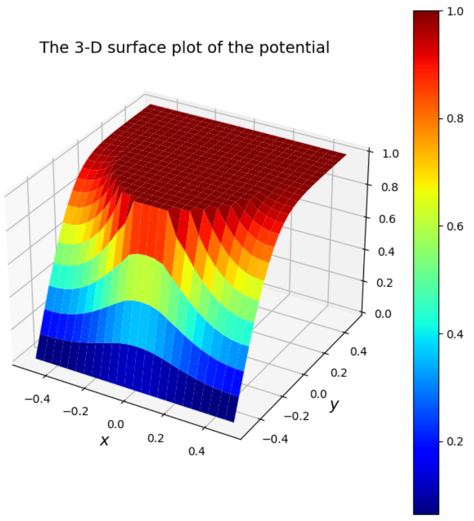
Where n is defined as np.arange(**len**(errors)).



2.8 Surface Plot of Potential

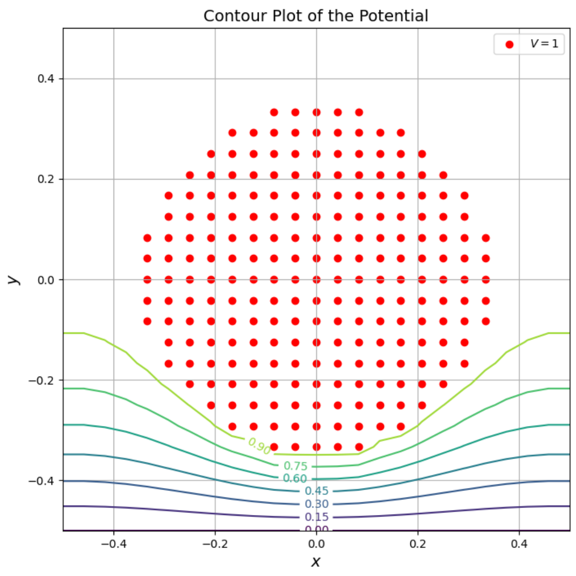
The surface_plot () generates the surface plot by the following code:

```
ax = plt.figure(figsize=(8, 8)).add_subplot(projection="3d")  # new way
plt.colorbar(ax.plot_surface(X, Y, phi, rstride=1, cstride=1, cmap=cm.jet))
```



2.9 Contour Plot of Potential

The same contour_plot() function from Subsection 2.3 has been used:



2.10 Vector Plot of Currents

The currents are given by the following equations:

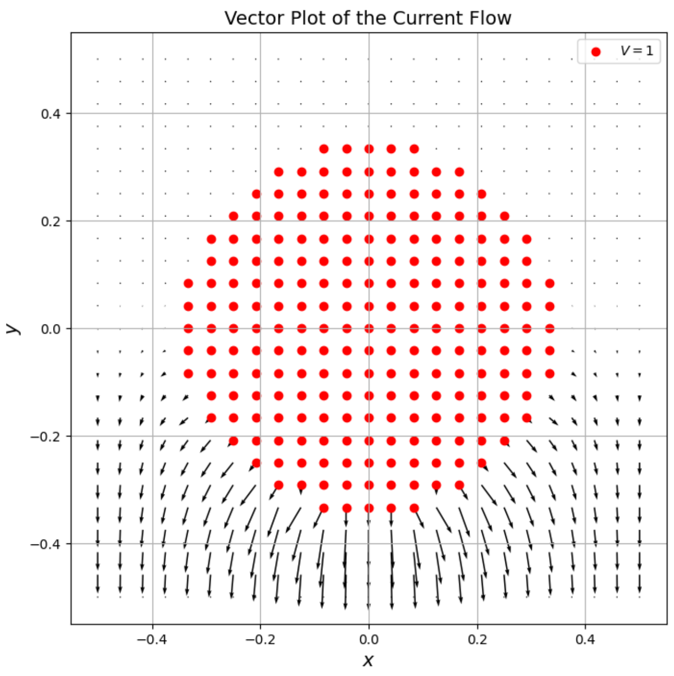
$$J_{x,ij} = \frac{1}{2}(\phi_{i,j-1} - \phi_{i,j+1})$$
$$J_{y,ij} = \frac{1}{2}(\phi_{i-1,j} - \phi_{i+1,j})$$

The plot_currents() function implements these equations as:

```
Jx, Jy = np.zeros_like(phi), np.zeros_like(phi)
Jx[:, 1:-1] = (phi[:, :-2] - phi[:, 2:]) / 2
Jy[1:-1, :] = (phi[:-2, :] - phi[2:, :]) / 2
```

These matrices are subsequently plotted as:

```
plt.quiver(X, Y, Jx, Jy, scale=4)
plt.scatter(X[0, ii[1]], Y[ii[0], 0], color="r", label="$V=1$")
```



3 Conclusion

The Laplace Equation has been solved using discrete differentiation iteratively. This method of solving the Laplace Equation is generally not recommended. This is because of the very slow coefficient with which the error reduces.