Chad Chapnick

[redacted]

[redacted]

March 21st, 2018

Introduction

The objective of this assignment was to create a program which calculates the electric potential and creates a plot of the equipotential lines produced by an electromagnetic field. This was achieved by implementing an algorithm in Python which uses the finite-difference method to estimate the electric potential and solve Laplace's equation.

$$\nabla^2 \varphi = 0 \tag{1}$$

The finite-difference method begins by defining a rectangular mesh of uniformly spaced grid points. The nodes at each side of the grid's boundary are initialized to the appropriate voltage values. The remaining nodes are initialized to 0. The remaining values are calculated by averaging neighboring nodes until their values converge. Specifically, the value for each node is obtained from the average value of the four contiguous ones (Eq. 2). This process is repeated until the error of each node is below some predetermined threshold (Δ), where the error is calculated according to (Eq. 3).

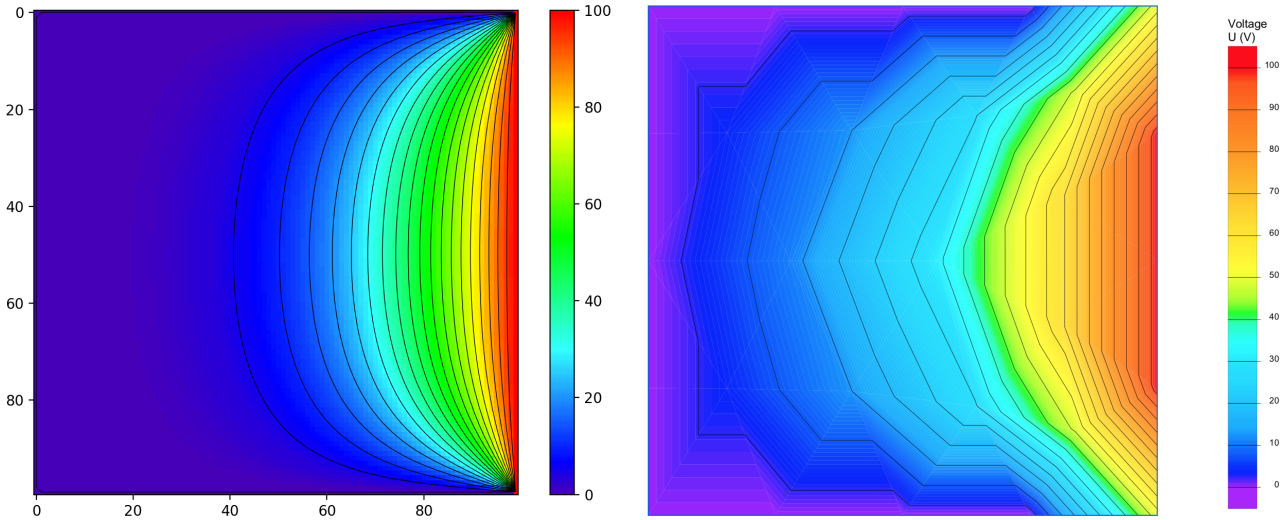
$$f(x,y) = \frac{f(x+h,y) + f(x-h,y) + f(x,y+h), f(x,y-h)}{4} \tag{2}$$

$$\text{error} = |\text{current} - \text{previous}| < \Delta \tag{3}$$

From (Eq. 2), it is clear that the finite difference method is the definition of a first order derivative between two points separated by a distance 'h'. After finding the electric potential at all grid points using the finite difference method, Quickfield simulation software was used for a qualitative validation of the results.

Problem 1

In problem 1, we consider four infinitely long conducting strips of equal width, situated such that the cross-section of the arrangement is square and held at potentials $\varphi_a = \varphi_b = \varphi_l = 0\ V$, $\varphi_r = 100\ V$, where $\varphi_a, \varphi_b, \varphi_l$, and φ_r are the potentials of the top, bottom, left, and right faces of the square, respectively. The results of the algorithmic approach and the Quickfield simulation are shown in table 1 and figure 1.



(a) Algorithm (b) Quickfield

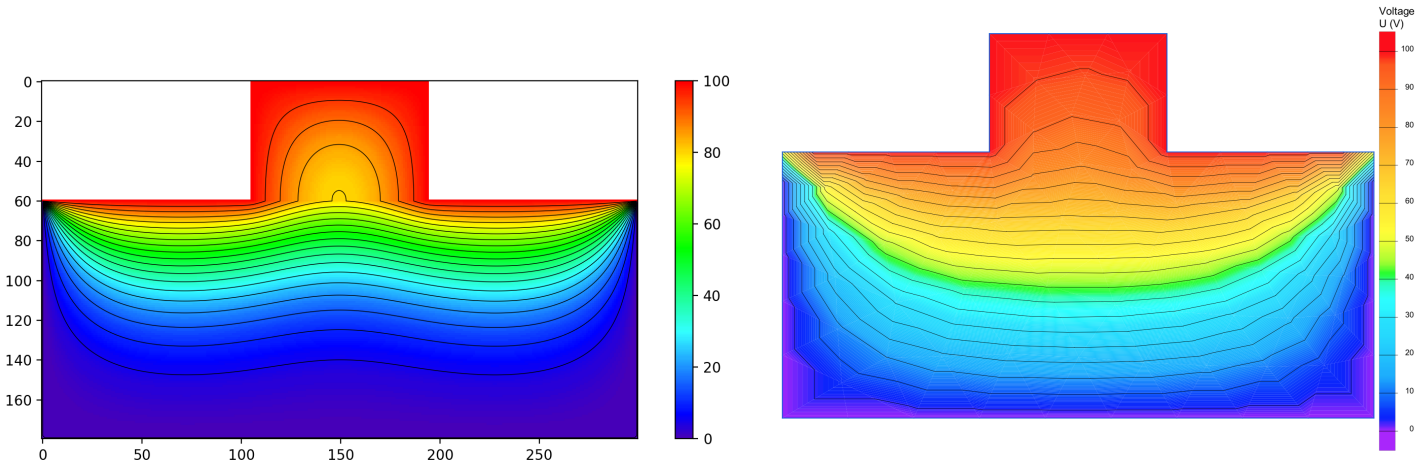
Figure 1: Potential Distribution in Geometry 1

Table 1: Problem 1 Results

N	Δ	Number of Iterations
100	0.02	965

Problem 2

In problem 2, we consider a potential distribution bounded by the geometry shown in figure 2. The dimensions of the rectangle on the top are 2cm x 3cm, and the dimensions of the rectangle on the bottom are 4cm x 10cm. There is a gap between the three faces which make up the bottom of section the geometry and the five faces which make up the top section. The conducting strips of the bottom section are held at a potential of $\Phi = 0$, while the conducting strips of top section are held at $\Phi = 100$. The results of the algorithmic approach and the Quickfield simulation are shown in table 2 and figure 2.



(a) Algorithm

(b) Quickfield

Figure 2: Potential Distribution in Geometry 2

Table 2: Problem 2 Results

N	Δ	Number of Iterations
300	0.02	2149

Discussion

The finite difference implementation appears to reasonably approximate the electric potential distribution due to the conducting strips. In problem 1, the computation converged after 965 iterations with N = 100 and $\Delta = 0.02$. In problem 2, the computation converged after 2149 iterations with N = 300 and $\Delta = 0.02$ It is worth noting that these approximations could be improved by increasing the number of voltage samples, N. This would decrease the distance between samples and thereby improve the accuracy of the first order derivative used in the esimation of electric potential.

A qualitative analysis of the results from the Quickfield simulation and the results from the algorithmic approach suggest that the electromagnetic fields are approximately the same in problems 1 and 2. Thus, we can conclude that the finite-difference method is a reasonable computational technique used to model the behavior of simple electromagnetic systems. Overall, we were able to demonstrate the ability to solve Laplace‘s equation numerically and the ability to visualize the electric potential distribution within simple geometric boundaries.

Program Listing

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors


def compute_averages(M):
    nrows, ncols = M.shape
    for i in range(1, nrows - 1):
        for j in range(1, ncols - 1):
            new_value =  1/4 * (  M[i, j-1]
                                + M[i, j+1]
                                + M[i-1, j]
                                + M[i+1, j])
            M[i, j] = new_value
    return M



# Set up the colormap...
def truncate_colormap(cmap, minval=0.0, maxval=1.0, n=100):
    new_cmap = colors.LinearSegmentedColormap.from_list(
        'trunc({n},{a:.2f},{b:.2f})'.format(n=cmap.name, a=minval, b=maxval),
        cmap(np.linspace(minval, maxval, n)))
    return new_cmap


cdict = {'red': ((0.0, 0.0, 0.0),
                 (0.1, 0.5, 0.5),
                 (0.2, 0.0, 0.0),
                 (0.4, 0.2, 0.2),
                 (0.6, 0.0, 0.0),
                 (0.8, 1.0, 1.0),
                 (1.0, 1.0, 1.0)),
        'green':((0.0, 0.0, 0.0),
                 (0.1, 0.0, 0.0),
                 (0.2, 0.0, 0.0),
                 (0.4, 1.0, 1.0),
                 (0.6, 1.0, 1.0),
                 (0.8, 1.0, 1.0),
                 (1.0, 0.0, 0.0)),
        'blue': ((0.0, 0.0, 0.0),
                 (0.1, 0.5, 0.5),
                 (0.2, 1.0, 1.0),
                 (0.4, 1.0, 1.0),
                 (0.6, 0.0, 0.0),
                 (0.8, 0.0, 0.0),
                 (1.0, 0.0, 0.0))}

new_cmap = matplotlib.colors.LinearSegmentedColormap('my_colormap',cdict,256)
new_cmap = truncate_colormap(new_cmap, 0.145, 1, 256)
```

```
# Problem 1
phi_l = phi_a = phi_b = 0
phi_r = 100

N = int(100)
A = np.zeros((N, N))
A[:, -1] = phi_r

delta = 0.02

done = False
count=0
while not done:
    previous = np.copy(A[1:-1, 1:-1])
    A = compute_averages(A)
    current = A[1:-1, 1:-1]
    if np.all(np.abs(current - previous) < delta):
        done = True
    count += 1
    avg_diff = np.nanmax(np.abs(current - previous))
    print(f'Iteration {count}')
    print(f'Convergence {avg_diff}')



plt.imshow(A, cmap=new_cmap)
plt.colorbar()
plt.contour(A,
        cmap=plt.cm.get_cmap('binary', 1),
        levels=range(int(A.min()), int(A.max()), 5),
        linewidths=0.5
    )
```

```
# Problem 2 (small square 2cm x 3cm, large square 4cm x 10cm)
N = 300
phi = 100

# set up the matricies
A = np.zeros((round(0.6 * N), round(N)))

offset_lr = round(0.7/2 * N) + 1
offset_vert = round(0.2* N)

A[:offset_vert, :offset_lr] = np.nan
A[:offset_vert, -offset_lr:] = np.nan

# +1 for the overlap row that pulls from A2
A1 = np.zeros((round(0.2*N) + 1, round(0.3*N)))
A1[:, 0] = phi
A1[0, :] = phi
A1[:, -1] = phi

A2 = np.zeros((round(0.4*N), N))
A2[0, :offset_lr] = phi
A2[0, -offset_lr:] = phi

# reconstruct the whole thing
A[:offset_vert,offset_lr-1:-(offset_lr-1)] = A1[:-1, :]
A[offset_vert:,:] = A2

delta = 0.02
done = False
count=0
while not done:
    previous = np.copy(A)

    # take the top row of A2 and make it the bottom plate of A1
    A1[-1, 1:-1] = A2[0, offset_lr:-offset_lr]
    A1 = compute_averages(A1)

    A2[0, offset_lr:-offset_lr] = A1[-2, 1:-1]
    A2 = compute_averages(A2)

    # reconstruct the whole thing
    A[:offset_vert,offset_lr-1:-(offset_lr-1)] = A1[:-1, :]
    A[offset_vert:,:] = A2

    if np.all(np.abs(np.nan_to_num(A) - np.nan_to_num(previous)) < delta):
        done = True
    count += 1

    avg_diff = np.nanmean(np.abs(A - previous))
    print(f'Iteration {count}', f'Avg diff {avg_diff}')

plt.imshow(A, cmap=new_cmap)
plt.colorbar()
plt.contour(A,
        cmap=plt.cm.get_cmap('binary', 1),
        levels=range(int(np.nanmin(A)), int(np.nanmax(A)), 5),
        linewidths=0.5
    )
```