1 Introduction

When electric potential is applied to conductors an electric field is created, which in turn creates arbitrary potentials between conductors. This is a study of the potential, electric field and charge involved in conductors. This also discusses capacitance of arbitrary conductors and how that is related to charge.

Specifically this will discuss a concentric square cylinder capacitor like that shown in Figure 1.

![Conductor plates for investigation](image)

2 Background

Laplace’s equation (Equation 1 also notated as Equation 2) describes many physical systems. In this particular case it describes the potential between given “plates” of potential. There is no completely general solution to Laplace’s
equation, only general solution techniques, which in this case of boundary conditions do not create a particularly solvable problem. A method for solving Laplace’s equation will be presented in Section 3.

\[
\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0 \quad (1)
\]

\[
\nabla^2 V(x, y, z) = 0 \quad (2)
\]

Given the calculated potential, a calculation of the electric field is a simple operation. Electric field is related to the potential as such:

\[
\vec{E} = -\nabla V \quad (3)
\]

This is given from the definition of voltage:

\[
V = \int \vec{E} \cdot ds \quad (4)
\]

The surface charge for a conductor is related to an electric field by:

\[
\sigma = \varepsilon_0 |\vec{E}| \quad (5)
\]

It is actually related to the value of the electric field normal to the conductor, but since the electric field parallel to a conductor has to be zero since the potential difference in a conductor is zero, the magnitude of the electric field is accurate. The only flaw in this representation is that the sign of the charge is lost, but for these calculations this is unimportant.

This relationship can be verified with dimensional analysis. Since it is known that \( \sigma \) is \( \frac{Q}{m^2} \) and \( \varepsilon_0 \) is \( \frac{F}{m} \) and \( E \) is \( \frac{V}{m} \) a simple multiplication reveals:

\[
\frac{F \ V}{m \ m} = \frac{FV}{m^2} \quad (6)
\]

\[
F = \frac{Q}{V} \quad (7)
\]

\[
\frac{Q \ V}{m \ m} = \frac{Q}{m^2} = \sigma \quad (8)
\]

Since this is an infinitely long body, exact capacitance cannot be calculated. The only value that can be calculated is capacitance per unit length. This value of capacitance can be found from the surface charge by integrating along the sides and dividing by the potential difference in the capacitor.

\[
\frac{Q}{L} = 4 \int_0^L \sigma ds \quad (9)
\]

\[
\frac{C}{L} = \frac{Q}{\Delta V} \quad (10)
\]
3 Simulation

3.1 Simulation Methods

Laplace’s problem can be solved in several different ways, but when dealing with steady state systems in two dimensions, one of the better methods is the relaxation method. The relaxation method is derived by taking the taylor expansion of Laplace’s Equation in each of the four directions from a given point.

\[
V_N = V_0 + a \partial_y V_{x_0} \ldots \tag{11}
\]
\[
V_S = V_0 - a \partial_y V_{x_0} \ldots \tag{12}
\]
\[
V_E = V_0 + a \partial_x V_{y_0} \ldots \tag{13}
\]
\[
V_W = V_0 - a \partial_x V_{y_0} \ldots \tag{14}
\]

These equations can be rewritten so that \(V_0\) is a function of the four border parts. The arrangement of this can be seen in Figure 2. When they are solved as a function of \(V_N, V_S, V_E, V_W\) result is as follows.

\[
V_0 = \frac{1}{4} (V_N + V_S + V_E + V_W) \tag{15}
\]

Figure 2: If the average of all values surrounding coordinate \(i,j\) is taken to be the value of \(i,j\) a smoothing occurs. This is the relaxation method.

The code for this is implemented in lines 133-4 of of the listing in Section 6. As can be seen in the code, the system can be represented as a lattice. This lattice structure can be solved by setting \(V_0\) to be equal to the average of its neighbors repeatedly until it settles (relaxes) into a final shape that will agree with the laplace equation. This provides a simple solution method to a complex equation.

The simulation continues to calculate the corresponding electric field from the potential by using Equation 3. This a simple derivative of the potential in both the \(x\) and the \(y\) directions to produce a vector describing the electric field. This is done using a differencing algorithm seen in lines 152-214 of the
listing in Section 6. The algorithm is designed to calculate the derivatives at the
points where the potentials are calculated as opposed to the simplest algorithm
which calculates the derivative in the spaces between the points. It does this as
follows.

\[
\frac{\partial V}{\partial x} = \begin{cases} 
\frac{V_{i+1} - V_i}{2 \text{spacing}} & \text{all inside points} \\
\frac{V_{i+1} - V_i}{\text{spacing}} & \text{when } i = 0 \\
\frac{V_i - V_{i-1}}{\text{spacing}} & \text{when } i = N - 1 
\end{cases}
\]  

(16)

\[
\frac{\partial V}{\partial y} = \begin{cases} 
\frac{V_{j+1} - V_j}{2 \text{spacing}} & \text{all inside points} \\
\frac{V_{j+1} - V_j}{\text{spacing}} & \text{when } j = 0 \\
\frac{V_j - V_{j-1}}{\text{spacing}} & \text{when } j = N - 1 
\end{cases}
\]  

(17)

The calculation of surface charge is shown in the listing in Section 6 on lines
275-302. It is calculated by using the fact that for all cases where a “plate”
of potential exists, the derivative (and thus the electric field) is zero parallel to
the plate and thus the surface charge is just a function of the magnitude of the
electric field and the permittivity of free space.

\[
\sigma = \epsilon_0 \sqrt{E_x^2 + E_y^2}
\]  

(18)

Finally the capacitance can be found by integrating the surface charge across
the plates. Since this is already a sampled function, the integration is merely a
summation of the surface charge on a plate. The capacitance is finally calculated
by dividing by the voltage differential in the system.

### 3.2 Simulation Validity

The simulation can be verified by comparing to a known case. Since this shape
of capacitor provides no known case, the only comparison that can be made is
to a similar shape. This can be compared to a cylindrical capacitor to within
at least an order of magnitude.

Given that the outside square is 100 meters across and the inside square
is 20 meters across, the perimeter of the outside square is 400 meters and the
inside square is 80 meters. Calculating circles to compare to would be done as
follows.

\[
400m = 2\pi r_{\text{out}} \\
r_{\text{out}} = 63.662m \\
80m = 2\pi r_{\text{in}} \\
r_{\text{in}} = 12.732m
\]  

(19)  (20)

The capacitance of a cylindrical capacitor can be found to be as shown in
Equation 21. Using that equation and plugging in the appropriate values of \(r_{\text{in}}\)
and $r_{\text{out}}$ the capacitance can be calculated to be $C_{\text{cylinder}} = 3.4566 \times 10^{-11} \frac{F}{m}$.

This is quite close to the programatically calculated value for the squares, which is $C_{\text{square}} = 3.6727 \times 10^{-11} \frac{F}{m}$. Given the closeness of these values, the simulation is accurate within reasonable bounds.

$$\frac{C}{L} = \frac{2\pi \varepsilon_0}{\ln \left( \frac{r_{\text{out}}}{r_{\text{in}}} \right)} \quad (21)$$

### 4 Results

The calculated potential based on a 5V inner plate and a 10V outer plate can be seen in Figure 3. In this you can see the equipotential lines projected on the bottom showing how the potential creates a smooth surface connecting all portions. This is because of the way the algorithm averages each point with the points around it.

![Figure 3: Potential as calculated by the relaxation method](image)

The electric field should always be perpendicular to lines of equipotential. This can be seen in Figure 4. The arrows point in the direction of the electric field. Again the equipotential lines are drawn for comparison.

The surface charge, which is calculated from the electric field. It can be seen in Figure 5. Because electric field is stronger in convex corners and weaker in concave corners it goes to zero in the outside corner and goes to infinity in the inside corners. The outside surface charge can be seen accurately in Figure 6. This figure demonstrates the shape of the charge density and its linearity near the corners.

The inside surface charge presents a problem. Because the corners are singularities, the normal summation technique doesn’t work for calculating the
Figure 4: Electric field calculated from potential

Figure 5: Surface charge as calculated from the electric field
Figure 6: Surface charge on one side of the outside of the capacitor

integral of the density. For this reason, the shape of the inside is accurate, but
the summation isn’t since theoretically the corners would be infinite. The inside
surface charge can be seen in Figure 7.

Given these surface charges and the configuration of the middle square being
20% of the outside square, the capacitance per unit length is $3.6727 \times 10^{-11} \frac{F}{m}$. 
5 Conclusion

The results demonstrate a reasonable system for calculating capacitance of shapes other than those easily calculable using gaussian surfaces. This alternative solution could be extended into a three dimensional system usable for calculating capacitances of real capacitors easily and accurately. Once a good system of input is devised, the program could become a powerful tool for capacitor design. The one big problem is the singularities in the inner square. A new system for integrating should be devised to reduce this error. This error does not cause a problem for systems without singularities, and due to the redundancy in the calculation of charge (both plates must have the same magnitude of charge), one should not have a singularity and thus the capacitance should be calculable.

6 Listing

```
1 #include <iostream>
2 #include <stdio>
3 #include <cmath>
4 #include <stdlib>
5 #include <fstream>
6 #include <vector>
7 #include <string>
8 #include <ctime>
9
10 // #define win32
11 6if define win32
13 void srand48(unsigned int i)
14 { srand(i);
16 }
18 double drand48()
19 { return rand() / 32768.0;
```
```cpp
using namespace std;

typedef struct a_vec {
  double x;
  double y;
} vec;

class lattice {
  public:
    // data members
    vector<vector<double>> data;
    vector<vector<bool>> mask;
    vector<vector<double>> edata;
    vector<vector<bool>> adjacentmask;
    double worst_error;
    double average_error;
    int N;
    double spacing;

    // member functions
    lattice(int n, double sp) {
      int i = 0;
      N = n;
      spacing = sp;
      worst_error = 1e100;
      average_error = 1e100;
      // initialize the data structure to all 0
      for (i = 0; i < n; i++)
        { vector<double> temp;
          for (int j = 0; j < n; j++)
            temp.push_back(0.0);
          data.push_back(temp);
        }
      // initialize the mask structure to all 0
      for (i = 0; i < n; i++)
        { vector<bool> temp;
          for (int j = 0; j < n; j++)
            temp.push_back(false);
          mask.push_back(temp);
        }
      // initialize edfield structure to all 0
      for (i = 0; i < n; i++)
        { vector<vec> temp;
          for (int j = 0; j < n; j++)
            { vec v;
              v.x = 0;
              v.y = 0;
              temp.push_back(v);
            }
          edata.push_back(temp);
        }
      // initialize surfacecharge structure to all 0
      for (i = 0; i < n; i++)
        { vector<double> temp;
          for (int j = 0; j < n; j++)
            temp.push_back(0.0);
          surfacecharge.push_back(temp);
        }
      // initialize adjacentmask to all false
      for (int x = 0; x < N; x++)
        { }

    }

    lattice() {
      int i = 0;
      N = 0;
      spacing = 0;
      worst_error = 1e100;
      average_error = 1e100;
      // initialize the data structure to all 0
      for (i = 0; i < n; i++)
        { vector<double> temp;
          for (int j = 0; j < n; j++)
            temp.push_back(0.0);
          data.push_back(temp);
        }
      // initialize the mask structure to all 0
      for (i = 0; i < n; i++)
        { vector<bool> temp;
          for (int j = 0; j < n; j++)
            temp.push_back(false);
          mask.push_back(temp);
        }
      // initialize edfield structure to all 0
      for (i = 0; i < n; i++)
        { vector<vec> temp;
          for (int j = 0; j < n; j++)
            { vec v;
              v.x = 0;
              v.y = 0;
              temp.push_back(v);
            }
          edata.push_back(temp);
        }
      // initialize surfacecharge structure to all 0
      for (i = 0; i < n; i++)
        { vector<double> temp;
          for (int j = 0; j < n; j++)
            temp.push_back(0.0);
          surfacecharge.push_back(temp);
        }
      // initialize adjacentmask to all false
      for (int x = 0; x < N; x++)
        { }
```
```cpp
vector<bool> temp;
for (int y = 0; y < N; y++)
{
    temp.push_back(false);
}
adjacemask.push_back(temp);
}

double lattice::converge(int inc)
{
    int totalsels = 0;
    double totalerrs = 0.0;
    double maxerr = 0.0;
    for (int i = 1; i < N - 1; i++)
    {
        for (int j = (i%2 + inc%2)%2 + 1; j < N - 1; j += 2)
        {
            if (!mask[i][j])
            {
                double lastval = data[i][j];
                data[i][j] = 0.25 * (data[i-1][j] + data[i][j+1] + data[i+1][j] + data[i][j-1]);
                totalerr += currerr;
                totalsels += 1;
                if (currerr > maxerr)
                    maxerr = currerr;
            }
        }
    }

    worst_error = maxerr;
    average_error = totalerr / (double)totalsels;
    return maxerr;
}

void lattice::efield(string outfile)
{
    ofstream fout;
    fout.open(outfile.c_str());
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            int i1, i2;
            int j1, j2;
            double ai = 2*spacing, aj = 2*spacing;
            if (i == 0)
            {
                i1 = 0;
                i2 = i + 1;
                ai = 1*spacing;
            }
            else if (i == N - 1)
            {
                i1 = i - 1;
                i2 = N - 1;
                ai = 1*spacing;
            }
            else
            {
                i1 = i - 1;
                i2 = i + 1;
            }
            double Ex = -(data[i2][j] - data[i1][j]) / ai;
            if (j == 0)
            {
                j1 = 0;
                j2 = j + 1;
                aj = 1*spacing;
            }
            else if (j == N - 1)
            {
                j1 = j - 1;
                j2 = N - 1;
                aj = 1*spacing;
            }
            else
            {
                j1 = j - 1;
                j2 = j + 1;
            }
            double Ey = -(data[i][j2] - data[i][j1]) / aj;
            double E = sqrt(Ey*Ey + Ex*Ex);
            double Exhat = Ex/E;
            double Eyhat = Ey/E;
            if (E == 0.0)
            {
                Exhat = 0.0;
                Eyhat = 0.0;
            }
        }
    }
}
```

\begin{verbatim}
203     }
204     double iEx = i + Exhat;
205     double jEy = j + Eyhat;
206     edata[i][j].x = Ex;
207     edata[i][j].y = Ey;
208     }
209     fout << "set arrow from " << i << ", " << j
210     << ", 2.5" << iEx << ", " << jEy
211     << ", 2.5" << endl;
212     //printf("set arrow from [%d,%d] to [%f,%f] in ", i, j, iEx,jEy);
213     fout.close();
214 }
215 }
216 }
217 }
218 void lattice::print(string outfile)
219 {
220     ofstream fout;
221     fout.open(outfile.c_str());
222     for (int i = 0; i < N; i++)
223     {
224         for (int j = 0; j < N; j++)
225             fout << i << ", " << j
226             << ", " << data[i][j] << endl;
227     }
228     fout.close();
229 }
230 }
231 }
232 void lattice::fill()
233 {
234     srand48((unsigned)time(NULL));
235     int numels = 0;
236     double total = 0.0;
237     int i;
238     for (i = 0; i < N; i++)
239     {
240         for (int j = 0; j < N; j++)
241             if (mask[i][j])
242                 numels++;
243                 total += data[i][j];
244     }
245     double average = total / (double) numels;
246     for (i = 0; i < N; i++)
247     {
248         for (int j = 0; j < N; j++)
249             if (!mask[i][j])
250                 data[i][j] = drand48() * average + average / 2;
251     }
252 }
253 }
254 void lattice::solve(bool useAVG, double maxerr)
255 {
256     int increment = 0;
257     while (!useAVG && (worst_error > maxerr))
258         converge(increment);
259     printf("%d completed in %d iterations\n", increment);
260 }
261 }
262 void lattice::findsurfacecharge(string outfile)
263 {
264     double e0 = 8.85418782e-12;
265     ofstream fout;
266     fout.open(outfile.c_str());
267     double sigma;
268     for (int i = 0; i < N; i++)
269     {
270         for (int j = 0; j < N; j++)
271             if (mask[i][j] == true)
272                 //sigma = sqrt(pow(data[i][j].x, 2) + pow(data[i][j].y, 2));
273                 + pow(data[i][j].x, 2)) * e0
274                 surfacecharge[i][j] = sigma * spacing;
275     }
276     fout << i << ", " << j << ", " << iEx << ", " << jEy << endl;
277 }
\end{verbatim}
C
MAX
determine
line
filename
Capacity:
Length
surface
charge.
maximum
lattice
looking
for

{328
324
327
326
333
330
f or

1
{
if (x + i < 0) |
x + i >= N)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y++)
for
y = -1; y <= 1; y+)
385  printf("The third line is the maximum error amount\n");
386  printf("The next lines are the lattice initial conditions\n");
387  printf("Each element is separated by whitespace\n");
388  printf("A character in an element indicates that it is calculated\n");
389  exit(0);
390 }
391 if (argc > 2)
392  potentialout = argv[2];
393 if (argc > 3)
394  fieldout = argv[3];
395 if (argc > 4)
396  surfacechargeout = argv[4];
397 ifstream fin;
398 if (!fin.open(argv[1]))
399  {
400    printf("Could not find file %s\n", argv[1]);
401    exit(0);
402  }
403  int N;
404  double spacing;
405  fin >> N;
406  fin >> spacing;
407  lattice l(N, spacing);
408  string type;
409  fin >> type;
410  bool useAVG = false;
411  if (type == "AVG")
412    useAVG = true;
413  double maxerror;
414  fin >> maxerror;
415  for (int i = 0; i < N; i++)
416  {
417    for (int j = 0; j < N; j++)
418    {
419      string input;
420      if (fin.eof())
421        {
422          printf("Invalid file format\n");
423          exit(0);
424        }
425        fin >> input;
426        if (input == "C")
427        {
428          l.mask[i][j] = true;
429          l.data[i][j] = atof(input.c_str());
430        }
431      }
432    }
433  }
434  l.fill();
435  l.solve(useAVG, maxerror);
436  l.print(potentialout);
437  l.print(fieldout);
438  l.field(fieldout);
439  l.findsurfacecharge(surfacechargeout);
440  l.findcharges();
441 )