AEP4380 HW6
Boundary Value Problems and Relaxation

Dan Girshovich

Mar 28, 2013

1 Overview

This document details the results of the included C++ program which uses successive over-relaxation to solve a boundary value problem for a three-dimensional electrostatic potential. After finding the potential in the region of interest, the electric field and capacitance are also found. The setup is axially symmetric, so the problem is reduced to solving the cylindrical finite difference Laplace equations in two dimensions.

2 The Electrostatic System

2.1 Boundary Conditions

The objective is to find the potential in a region of space near three electrodes. Each one is shaped like a disk with a centered circular hole and their axes are all aligned. The center electrode is held at 3000 V and the outer two at 0 V. Given the size of each disk and the distance between them, an appropriate bounding grounded cylindrical shell can be chosen to completely determine the system in a finite region. If the shell has a length and radius that are large compared to the region occupied by electrodes, the solution for the potential within the shell is approximately the same as is would be without the shell.

2.2 Discretization

To solve the system numerically, the first step is to define a finite grid of equally spaced points that lie in a plane containing the axis of the cylinder. This can be defined as follows:

$$z = z_{min} + ih; \;\; i = 0, 1, 2...i_{max}$$
$$r = r_{min} + jh; \;\; j = 0, 1, 2...j_{max}$$

Here, h is the grid spacing constant and it is small relative to the ranges of z and r.

The Laplace equation for cylindrical coordinates in a axially symmetric system is:

$$\nabla^2\Phi(z,r) = \left(\frac{\partial^2}{\partial z^2} + \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r}\right)\Phi(z,r)$$

where $\Phi(z,r)$ is the electric potential. On the finite grid this becomes:

$$\Phi_{i,j} = \frac{1}{4}\left(\Phi_{i,j+1} + \Phi_{i,j-1} + \Phi_{i+1,j} + \Phi_{i-1,j}\right) + \frac{1}{8j}\left(\Phi_{i,j+1} - \Phi_{i,j-1}\right),$$

except on the axis ($j = 0$):

$$\Phi_{i,j} = \frac{1}{6}\left(4\Phi_{i,j=1} + \Phi_{i+1,j=0} + \Phi_{i-1,j=0}\right).$$

The second derivative terms are transformed into averages and the first derivative term into a weighted central difference. Note, the first derivative term is discounted on the axis. This discrete version of the Laplace equation is implemented in `solution::laplace`.

3 Relaxation

3.1 Overview

A relaxation method allows the potential at each grid point to converge to the solution potential. The idea is to "relax" each local potential value by iteratively applying the discrete Laplace equation. After many iterations, the information contained in the fixed potential at the boundaries and electrodes will be spread over the region and the effect of the relaxation decreases.

3.2 Successive Overrelaxation

The relaxation method can be generalized beyond simply applying the Laplace equation such that it also considers the value exactly at each point (and not just the values of neighboring points). Let $\Phi_{i,j}^{Laplace}$ be the value produced by the finite Laplace equation given $\Phi_{i,j}^{n}$. Then:

$$\Phi_{i,j}^{n+1} = \omega * \Phi_{i,j}^{Laplace} + (1-\omega) * \Phi_{i,j}^{n}$$

For $1 < \omega < 2$, this technique is called "Successive Overrelaxation (SOR)" since it overshoots the value given by directly applying the Laplace equation. A form of this equation and a deeper discussion of relaxation methods can be found in section 20.5 of Numerical Recipes [1]. The iteration terminates once the largest change in potential after a pass over the grid is less than a specified tolerance called $|\Delta V|_{MAX}$. See the implementation of this method in `solution::next_v` and `solution::solve`.

3.3 Optimal ω

The optimal value of ω, ω_{opt}, is the one that minimizes the number of iterations used in the SOR algorithm. In general, the value for ω_{opt} is dependent on h. However, a decent approximation can be found by trying various values of ω for a fixed h. The results of applying this method to the electrostatic system with $h = 0.1$mm and $|\Delta V|_{MAX} = 1$V are plotted here:

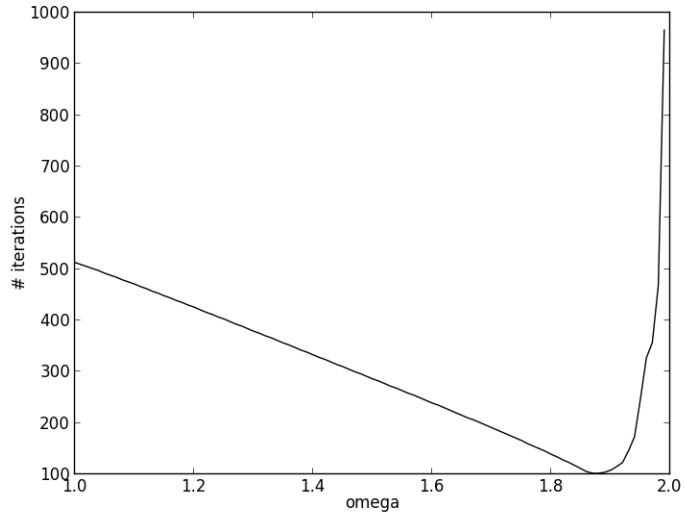


Figure 1: # iterations vs. ω. Data generated in `part1`.

The minimum occurs at $\omega_{opt} \approx 1.85$ and this value for ω is used for the remainder of the results. This value was rounded down from the true minimum to be safe from the sharp spike on the right side of the plot.

3.4 Tuning for Accuracy

The following table was created to find values for h and $|\Delta V|_{MAX}$ that produce sufficiently consistent results for $\Phi(0, 0)$. The table entries have units of V:

		$\|\Delta V\|_{MAX}$ (V)			
		1	0.1	0.01	0.001
h (mm)	0.1	849.397	851.306	852.015	
	0.05	789.687	843.054	846.268	
	0.025	616.706	817.933	842.419	
	0.0125	33.3294	748.795	834.825	842.163

Starting from the top left and moving to the bottom right, the diagonal elements of the table converge towards 842V. Note, only the diagonal element of the last column was computed to save time. The results to follow are found using $h = 0.025$mm and $|\Delta V|_{MAX} = 0.01$V to roughly achieve 3 significant figures of accuracy in voltage (at least near the origin). The

smallest values of these in the table could not be used because the computation time becomes too large. The table data is generated in `part2`.

4 Results

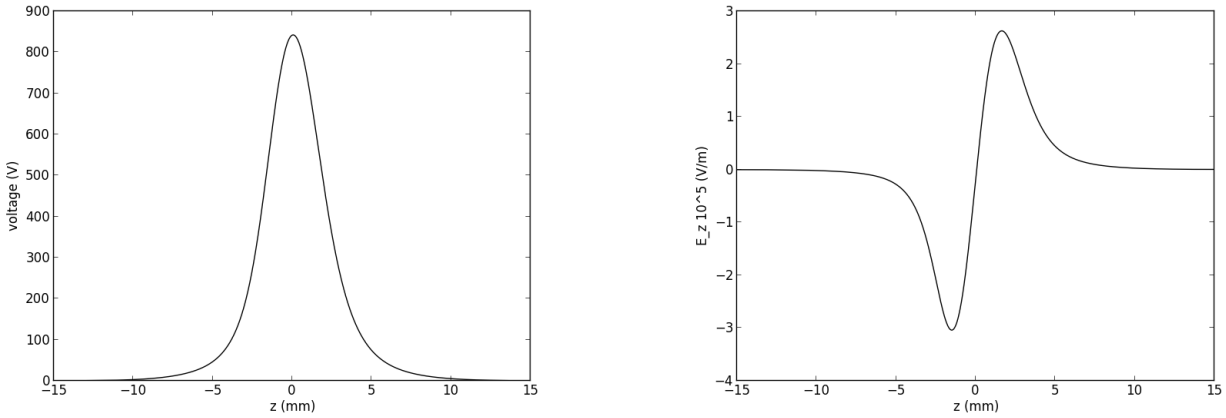
The following results were calculated using SOR with the ω_{opt}, h, and $|\Delta V|_{MAX}$ already found. Since the electrode configuration can be used to focus a beam of electrons, the results will be discussed in the context of this application. Also, note that the SOR method only produces the values for the potential, so the values for E_z and E_r are derived quantities. These were found using central difference differentiation, implemented in `solution::get_e_z` and `solution::get_e_r`. The relevant equations are:

$$E_z = -\frac{\partial \Phi(r,z)}{\partial z} \approx -\frac{\Phi(r,z+h) - \Phi(r,z-h)}{2h}$$
$$E_r = -\frac{\partial \Phi(r,z)}{\partial r} \approx -\frac{\Phi(r+h,z) - \Phi(r-h,z)}{2h}.$$

Care was taken to return in units of V/m after taking the finite difference and to special case the boundary derivatives.

4.1 Potential and Field on Axis of Symmetry

The values for the electric potential and field on the z axis are plotted here:



(a) Potential

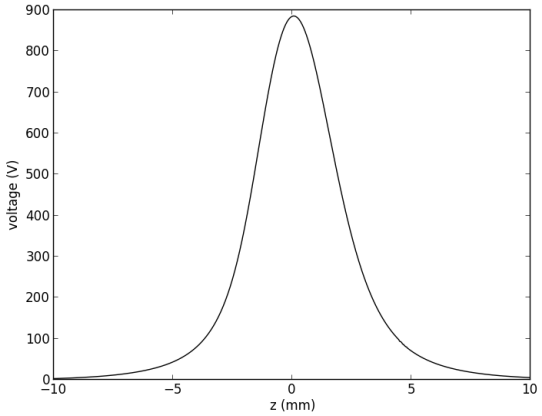
(b) Z Component of Electric Field

Figure 2: Solution values on the z axis. Data generated in `part3`.

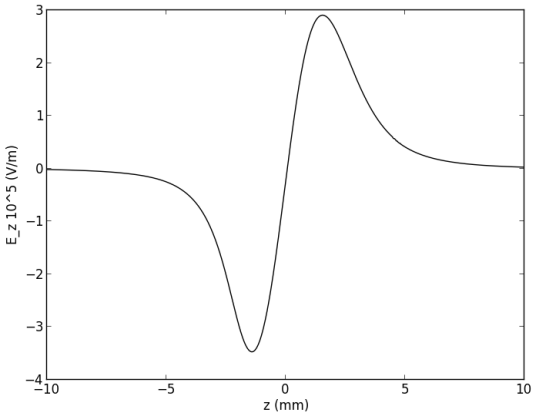
An electron traveling on the z axis through this potential in the increasing z direction would be accelerated until reaching $z = 0$, then it would be decelerated. The slight asymmetry in the electrodes has the effect of skewing the potential towards positive values of z.

4.2 Potential and Field off the Axis

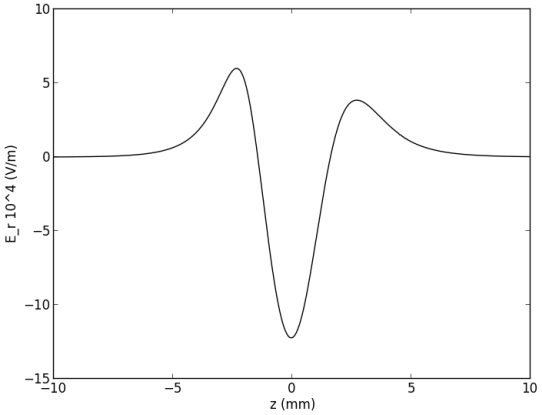
The values for the electric potential and field at $r = 0.75$ mm are plotted here:



(a) Potential



(b) Z Component of Electric Field



(c) R Component of Electric Field

Figure 3: Solution values at $r = 0.75$ mm. Data generated in `part4`.

An electron traveling slightly off of the z axis would feel a similar force in the z direction as in the previous case. However, the magnitude is slightly increased in this case since the electron is closer to the positive electrode. Also, there is now a force in the r direction that has the net effect of pushing the electron closer to the z axis. The plots alone do not directly imply this effect, since both the amount of time the electron spends in each region and the field in that region affect its trajectory. Looking at the force in the z direction, it seems like the $z < 0$ region is where electron would spend a large fraction of its time. In this case, the inward force there would be the dominant contribution and the it would deflect inward.

If instead of one electron, an electron beam was traveling along the z axis in the increasing z direction, it would be focused to a point. The electrons in the center of the beam would not feel a force in the r direction by the symmetry of the electrodes. However, electrons off-center

would feel a net inward force. This force would deflect them inwards so they converge at a point on the positive side of the z axis.

4.3 Full Potential Near Electrodes

The potential in the volume near the electrodes and the axis can be described by a two-dimensional contour plot, since it is axially symmetric. Such a plot is shown here:

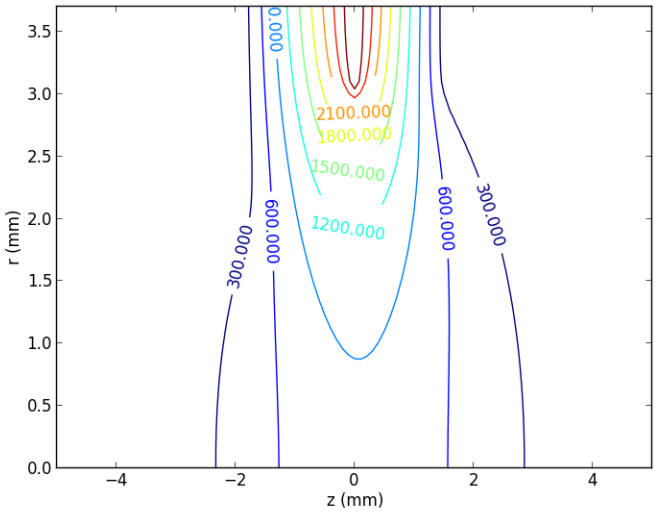


Figure 4: Contours of the potential (in V) near the electrodes and the axis.

The innermost contour is the boundary of the 3000V electrode. Also, the force on the electron at any point is parallel to the gradient of the potential at that point. So, the fringing contours on the left and right (like the 300V line) show the source of the focusing force.

4.4 Capacitance

The capacitance between the central and outer electrodes can be found by using:

$$W = \frac{1}{2}CV_C^2 = \frac{1}{2}\epsilon_0 \int |\vec{E}|^2 \mathrm{d}\tau$$

Where $V_C = V_{center} - V_{outer} = 3000$ V. The integral is performed in `solution::part6` using the derived values of E_z and E_r (third component is 0 by symmetry) and a simple Riemann sum. Specifically:

$$C = \frac{2\pi\epsilon_0}{V_C^2} \iint \left(E_z(r,z)^2 + E_r(r,z)^2\right) r \mathrm{d}r\mathrm{d}z \approx \frac{2\pi\epsilon_0 h^2}{V_C^2} \sum_{gridpoints} \left(E_z(r,z)^2 + E_r(r,z)^2\right) r$$

Using $h = 0.000025$m again, this gives $C \approx 3.42$ pF.

5 Implementation

5.1 Overview

The relaxation method is contained in the `solution` class. Upon calling the constructor of this class with h, *omega*, and $|\Delta V|_{MAX}$ as arguments, it automatically solves for $\Phi(r, z)$ at the grid points. Then, the results are accessed through `solution::get_v`.

The `solution` class uses the given array class along with the iterator abstraction `solution::for_each`. This allows other class methods to easily map functions over all grid points of both the potential matrix and the boundary point matrix. More importantly, it allows outside functions to compute things like capacitance without worrying about the implementation details of the SOR solution. The code uses this iterator in conjunction with the lambda syntax: [environment](arguments){function}.

5.2 Source Code

```
/*
AEP 4380 HW #6
Dan Girshovich
3/28/13
Finds the potential and field inside a cylindrically symmetric electrode setup.
Compile with: g++ -std=c++0x -O3 -o gen_data hw6.cpp
Tested on Mac OSX 10.8.2 with a Intel Core 2 Duo
*/

#include <cstdio>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "limits.h"
#include <cmath>
#include <functional>
// #define ARRAYT_BOUNDS_CHECK
#include "array.hpp"

bool feq(float, float);
float part1(float, float);
void part2(float);
void part3(float, float, float);
void part4(float, float, float);
void part5(float, float, float);
float part6(float, float, float);

// units are mm
const float SPACE_TOL = 0.0001; // tol for counting grid pts as boundary pts
const float R_MIN = 0, R_MAX = 15, Z_MAX = 15, Z_MIN = -15;
const float Z1 = -2, Z_MID = 0, Z2 = 1.5, Z3 = 3, R1 = 2, R2 = 3, R3 = 10;
// units are V
const float V1 = 0, V2 = 3000, V3 = V1;
```

```
// floating point equality
bool feq(float a, float b) {
    return fabs(a - b) < SPACE_TOL;
};

class solution {
public:
    const int I_MAX, J_MAX;
    int num_its;
    solution(float _h, float omega, float dv_max);
    void for_each(std::function<void(int, int)> f);
    float i_to_z(int i);
    float j_to_r(int j);
    int z_to_i(float z);
    int r_to_j(float r);
    float get_v(float z, float r);
    float get_e_r(float z, float r);
    float get_e_z(float z, float r);
private:
    const float h; // mm
    array<bool> bound_flags; // true if point on boundary or electrode
    array<float> v; // values of the potential on the grid
    bool on_electrode_2(int i, int j);
    bool on_boundary(int i, int j);
    float laplace(int i, int j);
    float next_v(int i, int j, float omega);
    void solve(float omega, float dv_max);
};

// assumes valid spacing
solution::solution(float _h, float omega, float dv_max) :
    I_MAX((Z_MAX - Z_MIN) / _h), J_MAX((R_MAX - R_MIN) / _h),
    num_its(0), h(_h),
    bound_flags(I_MAX + 1, J_MAX + 1), v(I_MAX + 1, J_MAX + 1) {
    for_each([ = ](int i, int j) {
        bound_flags(i, j) = on_boundary(i, j);
        v(i, j) = on_electrode_2(i, j) ? V2 : V1;
    });
    solve(omega, dv_max);
}

// iterator for the arrays
void solution::for_each(std::function<void(int, int)> f) {
    for (int j = 0; j <= J_MAX; j++)
        for (int i = 0; i <= I_MAX; i++)
            f(i, j);
}

// --- Helpers ----
float solution::i_to_z(int i) {
    return Z_MIN + i * h;
}

float solution::j_to_r(int j) {
```

```
    return R_MIN + j * h;
}

int solution::z_to_i(float z) {
    return (z - Z_MIN) / h;
}

int solution::r_to_j(float r) {
    return (r - R_MIN) / h;
}

// --- Accessors ---
float solution::get_v(float z, float r) {
    return v(z_to_i(z), r_to_j(r));
}

// use numerical derivitive to find E_r(z, r) (V/m)
float solution::get_e_r(float z, float r) {
    int i = z_to_i(z), j = r_to_j(r);
    float e_r;
    if (j == 0) { // forward difference
        e_r = (v(i, j + 1) - v(i, j)) / h;
    } else if (j == J_MAX) { // backward difference
        e_r = (v(i, j) - v(i, j - 1)) / h;
    } else { // central difference
        e_r = (v(i, j + 1) - v(i, j - 1)) / (2 * h);
    }
    return -1000 * e_r; // V/m
}

// use numerical derivitive to find E_z(z, r) (V/m)
float solution::get_e_z(float z, float r) {
    int i = z_to_i(z), j = r_to_j(r);
    float e_z;
    if (i == 0) { // forward difference
        e_z = (v(i + 1, j) - v(i, j)) / h;
    } else if (i == I_MAX) { // backward difference
        e_z = (v(i, j) - v(i - 1, j)) / h;
    } else { // central difference
        e_z = (v(i + 1, j) - v(i - 1, j)) / (2 * h);
    }
    return -1000 * e_z; // V/m
}

// --- boundary helpers ---
bool solution::on_electrode_2(int i, int j) {
    float z = i_to_z(i), r = j_to_r(j);
    return feq(z, Z_MID) && r > R2 && r < R3;
}

bool solution::on_boundary(int i, int j) {
    float z = i_to_z(i), r = j_to_r(j);
    bool on_edge = feq(z, Z_MIN) || feq(z, Z_MAX) || feq(r, R_MAX);
    bool on_electrode_1 = feq(z, Z1) && r > R1 && r < R3;
```

```
    bool on_electrode_3 = z > Z2 && z < Z3 && r > R2 && r < R3;
    return on_edge || on_electrode_1 || on_electrode_2(i, j) || on_electrode_3;
}

// finite Laplace equation in cylindrical coordinates
float solution::laplace(int i, int j) {
    if (j == 0) { // special case on axis of symmetry
        return (1.0 / 6) * (4 * v(i, 1) + v(i + 1, 0) + v(i - 1, 0));
    } else {
        float a = v(i - 1, j), b = v(i, j + 1), c = v(i + 1, j), d = v(i, j - 1);
        return (a + b + c + d) / 4 + (b - d) / (8 * j);
    }
}

// performs an SOR step at i, j with omega
float solution::next_v(int i, int j, float omega) {
    float v_old = v(i, j);
    if (bound_flags(i, j)) { // electrodes & edges are at fixed V
        return v_old;
    } else {
        return omega * laplace(i, j) + (1.0 - omega) * v_old;
    }
}

// performs SOR with omega until the maximum change is < dv_max
void solution::solve(float omega, float dv_max) {
    float largest_dv = 0;
    do {
        num_its++;
        largest_dv = 0;
        for_each([ = , &largest_dv](int i, int j) {
            float v_new = next_v(i, j, omega);
            float dv = fabs(v(i, j) - v_new);
            if (dv > largest_dv) largest_dv = dv;
            v(i, j) = v_new;
        });
    } while (largest_dv > dv_max);
}

int main() {
    float best_omega = part1(0.1, 1);
    part2(best_omega);
    float h = 0.025, dv_max = 0.01; // from part2 data
    part3(h, best_omega, dv_max);
    part4(h, best_omega, dv_max);
    part5(h, best_omega, dv_max);
    printf("%f\n", part6(h, best_omega, dv_max));
}

// num_its vs. omega
float part1(float h, float dv_max) {
    return 1.85; // decided on this value
    float best_omega = 0.0;
    int smallest_num_its = INT_MAX;
```

```
    std::ofstream f;
    f.open("omega.dat");
    for (float omega = 1; omega < 1.99; omega += 0.05) {
        solution s(h, omega, dv_max);
        if (s.num_its < smallest_num_its) {
            smallest_num_its = s.num_its;
            best_omega = omega;
        }
        std::cout << omega << " " << s.num_its << std::endl;
    }
    f.close();
    return best_omega;
}

// finds v(r = 0, z = 0) for various h and dv_max
void part2(float omega) {
    std::ofstream f;
    f.open("accuracy_table.dat");
    for (float dv_max = 1; dv_max > 0.0001; dv_max /= 10) {
        for (float h = 0.1; h > 0.001; h /= 2) {
            solution s(h, omega, dv_max);
            f << h << " " << dv_max << " " << s.get_v(0, 0) << std::endl;
        }
    }
    f.close();
}

// Plots V and Ez as a function of z along r = 0
void part3(float h, float omega, float dv_max) {
    std::ofstream f_v, f_e_z;
    f_v.open("v_on_axis.dat");
    f_e_z.open("e_z_on_axis.dat");
    solution s(h, omega, dv_max);
    for (int i = 0; i <= s.I_MAX; i++) {
        float z = s.i_to_z(i);
        f_v << z << " " << s.get_v(z, 0) << std::endl;
        f_e_z << z << " " << s.get_e_z(z, 0) << std::endl;
    }
    f_v.close();
    f_e_z.close();
}

// Plots V, Er, and Ez with r = 0.75mm for -10mm < z < 10mm
void part4(float h, float omega, float dv_max) {
    std::ofstream f_v, f_e_z, f_e_r;
    f_v.open("v_at_r75.dat");
    f_e_z.open("e_z_at_r75.dat");
    f_e_r.open("e_r_at_r75.dat");
    solution s(h, omega, dv_max);
    float r = 0.75;
    for (float z = -10; z < 10; z += h) {
        f_v << z << " " << s.get_v(z, r) << std::endl;
        f_e_z << z << " " << s.get_e_z(z, r) << std::endl;
        f_e_r << z << " " << s.get_e_r(z, r) << std::endl;
```

```
    }
    f_v.close();
    f_e_z.close();
    f_e_r.close();
}

// produces data for a contour plot of V near the electrodes and axis
void part5(float h, float omega, float dv_max) {
    std::ofstream f_v, f_z, f_r;
    f_v.open("v_contour.dat");
    f_z.open("z_contour.dat");
    f_r.open("r_contour.dat");
    solution s(h, omega, dv_max);
    // select range near electrodes and axis
    int lower_i = (s.I_MAX / 3), upper_i = 2 * (s.I_MAX / 3);
    int lower_j = 0, upper_j = s.J_MAX / 4;
    s.for_each([&](int i, int j) {
        if (i >= lower_i && i <= upper_i && j >= lower_j && j <= upper_j) {
            float z = s.i_to_z(i), r = s.j_to_r(j);
            f_v << s.get_v(z, r) << " ";
            if (i == upper_i) {
                f_r << r << std::endl;
                f_v << std::endl;
            }
            if (j == lower_j) f_z << z << std::endl;
        }
    });
    f_v.close();
    f_z.close();
    f_r.close();
}

// calculates the capacitance of the electrodes (pF)
float part6(float h, float omega, float dv_max) {
    float eps_0 = 8.8542, pi = 4.0 * atan(1.0), field_energy = 0.0;
    solution s(h, omega, dv_max);
    s.for_each([&](int i, int j) {
        float z = s.i_to_z(i), r = s.j_to_r(j);
        float e_z = s.get_e_z(z, r), e_r = s.get_e_r(z, r);
        field_energy += (e_z * e_z + e_r * e_r) * j;
    });
    field_energy *= pi * eps_0 * h * h * h * 1e-9; // mm -> m
    float v_cap = V2 - V1;
    return 2 * field_energy / (v_cap * v_cap);
}
```

References

[1] W.H. Press, S.A Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipies, The Art of Scientific Computing*. Camb. Univ. Press, 3rd Edition, 2007.