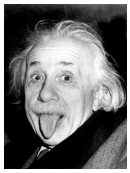
PH388 Project 1: Electrostatics

Hamilton, Robert
robert.hamilton.2017@uni.strath.ac.uk

Kennedy, Lewis
lewis.kennedy.2018@uni.strath.ac.uk

Holmes, Aaron
Aaron.holmes.2018@uni.strath.ac.uk

Hunter, Ross
Ross.Hunter.2018@uni.strath.ac.uk

Group: 

December 4, 2020

Contents

1 Kirchoff's Circuit Laws

1.1 Deriving Kirchhoff's Laws

Consider a volume V as shown in figure 1.

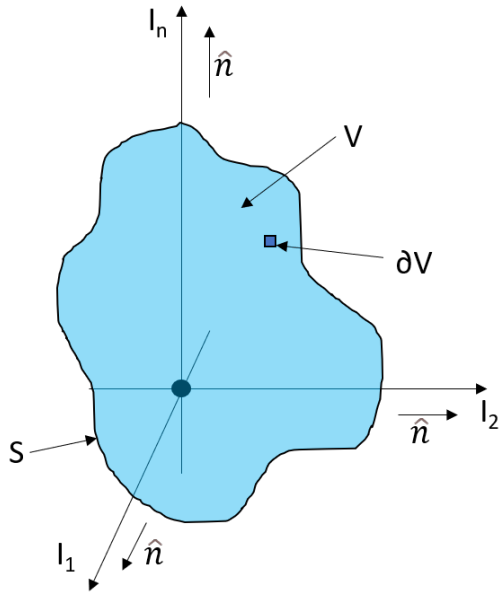


Figure 1: Surface around Volume

From Ampere's Law [1]:

$$\nabla \times \boldsymbol{B} = \mu_0 \boldsymbol{J} + \mu_0 \epsilon_0 \frac{\partial \boldsymbol{E}}{\partial t}$$

which, due to no time dependence in **E** in a DC circuit, reduces to

$$\nabla \times \boldsymbol{B} = \mu_0 \boldsymbol{J}$$

$$\text{since } \frac{\partial \boldsymbol{E}}{\partial t} = 0$$

Then, by taking the divergence of each side, we have

$$\boldsymbol{\nabla} \cdot (\nabla \times \boldsymbol{B}) = \mu_0 (\boldsymbol{\nabla} \cdot \boldsymbol{J})$$

Recognising that div(curl$\boldsymbol{B}$) = 0, as is true for the curl of any vector field, we are then left with the volume integral as follows

$$\mu_0 (\nabla \cdot \boldsymbol{J}) = 0$$
$$(\nabla \cdot \boldsymbol{J}) = 0$$
$$\int_V (\nabla \cdot \boldsymbol{J})\, dV = 0$$

By using the divergence theorem which states

$$\oiint_V \boldsymbol{J} \cdot \hat{n} ds = \int_V (\nabla \cdot \boldsymbol{J})\, dV$$

We can further reduce the problem using the description of the right hand side obtained in the previous steps.

$$\oiint_V \boldsymbol{J} \cdot \hat{n} ds = 0$$
$$\oiint_V J ds = 0$$

If the current is going out of the volume then J is positive (positive divergence) and if the current is going into the volume then the divergence is negative. Further reducing and taking out the sum of J, we have

$$\sum_i J_i \oiint_V ds = 0$$
$$\sum_i J_i S_i = 0,$$

where I_i = $J_i S_i$, leaving Kirchoff's 1st law [2] which states that the sum of all currents passing through a junction must be 0, in order to obey conservation of energy laws.

$$\sum_i I_i = 0$$

Kirchhoff's Voltage Law:

Now we consider a surface S as shown in figure 2.

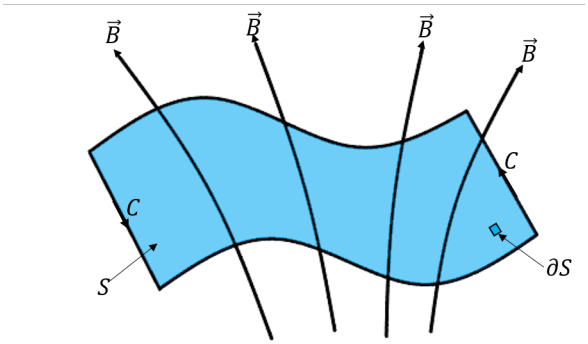


Figure 2: Closed loop around a surface

From Faraday's Law of Induction [1]:

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t}$$
$$\iint_S (\nabla \times \boldsymbol{E})\, dS = -\iint_S \frac{\partial \boldsymbol{B}}{\partial t} \cdot dS$$

Using Stokes Theorem the LHS can be written as:

$$\iint_S (\nabla \times \boldsymbol{E})\, dS = \oint_C \boldsymbol{E} \cdot dl$$
$$\oint_C \boldsymbol{E} \cdot dl = -\iint_S \frac{\partial \boldsymbol{B}}{\partial t} \cdot dS$$

The partial derivative can be taken outside of the integral to give:

$$-\iint_S \frac{\partial B}{\partial t} \cdot dS = -\frac{\partial}{\partial t} \iint_S B \cdot dS$$
$$\oint_C \boldsymbol{E} \cdot dl = -\frac{\partial \Phi_B}{\partial t}$$
$$\oint_C \boldsymbol{E} \cdot dl = 0 \text{ since } \frac{\partial \Phi_B}{\partial t} = 0$$
$$Ed = V = 0$$

In this case the distance is around the curve C so the integral can be written as the sum of the the voltages:

$$\sum_i V_i = 0$$

Which is Kirchhoff's Voltage Law [2] as required. This relationship states that the sum of all voltages around a closed loop circuit must be equal to 0 in accordance with conservation of energy laws. Meaning both of Kirchoff's laws are essentially conservation of energy applied to circuits.

1.2 Analytical Solution to a Circuit Problem

This section aims to provide an analytical solution to the circuit presented in appendix B. The solution is derived from 3 equations derived from Kirchhoff's laws with the values found in appendix A.

The equations derived from Kirchhoff's first and second laws applied to the circuit are as follows

$$\begin{aligned} I_1 - I_2 + I_3 &= 0 \\ -R_1 I_1 - R_2 I_2 &= U \\ -R_2 I_2 - R_3 I_3 &= U \end{aligned}$$

With the values for the resistances and voltage as given in appendix A. This system of equations can then be expressed as a 3×3 matrix in which the columns encode their corresponding currents and by transforming this matrix into a unit matrix, the values of the individual currents can be determined. This is achieved through a series of row swaps as shown.

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ -R_1 & -R_2 & 0 & U \\ 0 & -R_2 & -R_3 & U \end{array}\right]$$

$\underline{R_2} + R_1 \cdot \underline{R_1} \to \underline{R_2}$

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & -(R_1+R_2) & R_1 & U \\ 0 & -R_2 & -R_3 & U \end{array}\right]$$

$\underline{R_3} - \frac{R_2}{R_1+R_3} \cdot \underline{R_2} \to \underline{R_3}$

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & -(R_1+R_2) & R_1 & U \\ 0 & 0 & -R_3 - \frac{R_1 \cdot R_2}{R_1+R_2} & U - \frac{U \cdot R_2}{R_1+R_2} \end{array}\right]$$

$\underline{R_3} \cdot \frac{R_1+R_2}{R_1R_2+R_1R_3+R_2R_3} \to \underline{R_3}$

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & -(R_1+R_2) & R_1 & U \\ 0 & 0 & 1 & -(\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}) \end{array}\right]$$

$\underline{R_1} - \underline{R_3} \to \underline{R_1}$

$$\left[\begin{array}{ccc|c} 1 & -1 & 0 & (\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}) \\ 0 & -(R_1+R_2) & R_1 & U \\ 0 & 0 & 1 & -(\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}) \end{array}\right]$$

$\underline{R_2} \cdot \underline{R_3} \to \underline{R_2}$

$$\left[\begin{array}{ccc|c} 1 & -1 & 0 & (\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}) \\ 0 & -(R_1+R_2) & 0 & \frac{U(R_3R_1^2+2R_1R_2R_3+2R_1^2R_2+R_2^2R_3+R_1R_2^2+R_1^3)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_2^2R_3+R_1R_2^2} \\ 0 & 0 & 1 & -(\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}) \end{array}\right]$$

$-\underline{R_2} \cdot \frac{1}{R_1+R_2} \rightarrow \underline{R_2}$

$$\left[\begin{array}{ccc|c} 1 & -1 & 0 & \left(\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}\right) \\ 0 & 1 & 0 & \frac{-U(R_3R_1^2+2R_1R_2R_3+2R_1^2R_2+R_2^2R_3+R_1R_2^2+R_1^3)}{(R_1+R_2)(R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_2^2R_3+R_1R_2^2)} \\ 0 & 0 & 1 & -\left(\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}\right) \end{array}\right]$$

$\underline{R_1} + \underline{R_2} \rightarrow \underline{R_2}$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & \frac{(-U)(R_1^2R_3+2R_1R_2R_3+R_2^2R_3)}{(R_1+R_2)(R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_2^2R_3+R_1R_2^2))} \\ 0 & 1 & 0 & \frac{-U(R_3R_1^2+2R_1R_2R_3+2R_1^2R_2+R_2^2R_3+R_1R_2^2+R_1^3)}{(R_1+R_2)(R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_2^2R_3+R_1R_2^2)} \\ 0 & 0 & 1 & -\left(\frac{(UR_1)(R_1+R_2)}{R_3R_1^2+2R_1R_2R_3+R_1^2R_2+R_3R_2^2+R_1R_2^2}\right) \end{array}\right]$$

Resulting in the solution:

$$I = [-1.09, -1.45, -0.36]A$$

As can be seen from the lengthy process above, deriving an analytical solution to even the smallest parallel circuit diagram can be time consuming and complex, with many opportunities for small miscalculations causing large errors. Furthermore the solutions largely only work for the given values, therefore any change to the circuit requires a brand new analytic solution. This shows that a matrix solver would be a far more ideal solution to any such problems, as they tend to be faster and more accurate than a first draft analytic solution.

1.3 Algorithms Developed

1.3.1 Gaussian Elimination Method

For the more general case involving a greater number of equations, a numerical solver based on the Gaussian elimination algorithm was created to solve the system of linear equations using computational methods rather than analytical methods, as shown in appendix D. Gaussian elimination uses the same techniques as outlined in task 2 to find the values of the solution vector **x** in a system of the form **Ax** = **b**, where **A** is the matrix containing the N×N system of linear equations and **b** is the RHS vector of length N.

The algorithm first reduces the input matrix **A** to upper triangular form as shown using a series of row operations carried out by the use of for loops

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1,n} \\ 0 & a_{22} & & a_{2,n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Once the desired form of the input matrix **A** had been achieved, the algorithm then solves the equation on the last row of the augmented system, $a_{nn}x_n = b_n$ for x_n, and then uses this solution along with back substitution to solve the equation for the remaining solution vector values.

For the simple 3x3 case, the series of calculations performed by the algorithm would solve the system using back substitution as follows

$$\begin{aligned} x_3 &= \frac{b_3}{a_{33}} \\ x_2 &= \frac{1}{a_{22}}\left(b_2 - a_{23}x_3\right) \\ x_1 &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2 + a_{13}x_3\right) \end{aligned}$$

This leads to the general solution for any vector element x_i which is described as follows

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{N} a_{ij}x_j\right), \quad i = \{N-1, ..., 1\}$$

Using this, we are able to solve for any solution vector **x** corresponding to a system of linear equations of the form **Ax** = **b**. A key consideration of this algorithm is the requirement of partial pivoting of the input matrix, as without such features there can be large errors induced due to rounding errors when subtracting elements with vastly different magnitudes. The use of a pivoting function can serve as a good work around to these errors and so a function to carry out this operation was created in PH388functions as shown in appendix E.

To confirm the accuracy of the solutions and to determine if the conservation laws held, the solutions were then substituted back into equations 1-6 to ensure each loop returned 0 as required by Kirchoff's voltage law. For all loops, it was confirmed the net voltage was within the machine precision threshold of 1×10^{-14} V and thus accurate solutions had been obtained.

The computation of the system of linear equations was then carried out using other methods such as Gauss-Seidel and Jacobi's methods to compare the differences between the accuracy and effectiveness of direct solvers vs iterative solvers.

1.3.2 Gauss-Seidel Method

The Gauss-Seidel method shown in appendix F, is an algorithm based on an iterative solver which essentially works by decomposing an input matrix **A** to separate the diagonal elements **D** and the elements below the diagonal, **L**, while disregarding the upper triangular elements and creating the augmented matrices as shown

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ a_{21} & 0 & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 0 \end{pmatrix} + \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} \\ 0 & 0 & a_{23} & a_{24} \\ 0 & 0 & 0 & a_{34} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Once the diagonal and lower matrices have been isolated and stored, the method then begins by approximating each element of the solution vector using the following iteration steps, with each iteration using the most recent solution in approximating the current iteration

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{D} + \boldsymbol{L})^{-1} \left\{ \boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)} \right\}$$

Initial guess parameters of 0 for x_2 and x_3 are required for this method and using these initial values, the first approximation of x_1 is made at the very first step of the process. Similar to how x_1 is calculated using 0 for x_2 and x_3, x_2 is approximated using the latest approximation of x_1 and 0 for x_3. This process is repeated for each element of the solution vector, before being repeated by the desired number of iterations until the system converges to the desired threshold.

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_2 - a_{23}x_3 - a_{24}x_4) \\ x_3 &= \frac{1}{a_{33}}(b_1 - a_{32}x_2 - a_{32}x_3 - a_{34}x_4) \end{aligned}$$

The convergence threshold is checked by comparing the latest iteration against the last iteration and is defined as shown

$$x_n - x_{n-1} <= 1 \times 10^{-z}$$

where z can be defined to be any number depending on the order of magnitude of the tolerance desired, which in our case was of the order $1{\times}10^{-13}$.

A key feature of this method is that the solution will only converge if the spectral radius of the iteration matrix is < 1. That is, if the absolute value of the maximum eigenvalue of the Jacobi iteration matrix **J** is < 1 as shown

$$|\rho(\boldsymbol{J})| < 1$$

where the spectral radius is given by

$$\rho\left(\boldsymbol{J}\right) = max(|\lambda_1|, ...|\lambda_N|)$$

and where the Jacobi iteration matrix is given by

$$\boldsymbol{J} = \boldsymbol{D}^{-1}\left(\boldsymbol{L} + \boldsymbol{U}\right)$$

If this condition is not met, the solutions will simply not converge and no accurate solution vector will be returned. The full source code can be found in appendix F.

1.3.3 Jacobi Method

The Gauss-Seidel algorithm was then modified slightly to create a Jacobi method solver as shown in appendix G. The 2 algorithms are very similar, with only very minor differences in how the computation of the solution vector is carried out. The main difference is that while in the Gauss-Seidel method we make our first approximation for x_1 in the very first step of calculation using initial guess parameters of $x_2 = x_3 = 0$ to approximate x_1, for the Jacobi method we must instead make a first initial guess for x_1 before it can then be used in approximating the next iterations. As before, a requirement for this method is diagonal dominance however without diagonal dominance, the method *may* still converge while incurring performance penalties in runtime.

For the Jacobi method, the input matrix **A** is decomposed into (L)ower, (U)pper and (D)iagonal matrices as in the Gauss-Seidel method

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ a_{21} & 0 & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 0 \end{pmatrix} + \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} \\ 0 & 0 & a_{23} & a_{24} \\ 0 & 0 & 0 & a_{34} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

This method essentially works by approximating the solution vector using:

$$\boldsymbol{x}^{k+1} = \boldsymbol{D}^{-1}\left\{\boldsymbol{b} - (\boldsymbol{L} + \boldsymbol{U})\boldsymbol{x}^{k}\right\}$$

which works towards a more accurate solution with each iteration by first taking an initial guess for $\mathbf{x}^{(0)}$, using it to solve for the first approximation of $\boldsymbol{x}^{(1)}$ and then solving for $\boldsymbol{x}^{(2)}$ and so on as follows

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4) \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_2 - a_{23}x_3 - a_{24}x_4) \\ x_3 &= \frac{1}{a_{33}}(b_1 - a_{32}x_2 - a_{32}x_3 - a_{34}x_4) \end{aligned}$$

The convergence of the solution matrix can then be checked by comparing the most recent solution against the latest iteration and if the difference between the iterations falls within the threshold, convergence had been achieved. In our case, we set the convergence threshold to 1×10^{-13} to account for errors related to the limitations of machine precision.

1.4 Solving a Complex Circuit System

In tasks 4, 5 and 6 we are given an extended complex circuit as shown in appendix C with known resistance and voltages and are tasked with solving the system for the unknown currents. The problem was set up as a system of linear equations in the form **Ax** = **b** using equations derived through applying Kirchoff's circuit laws to the circuit, with the unknown current solution vectors computed using the Gaussian elimination, Gauss-Seidel and Jacobi algorithms as developed in tasks 3, 5 and 6 with their respective effectiveness' compared.

The system of equations resulting from Kirchhoff's first and second law are as follows:

$$I_2 - I_1 - I_3 = 0 \tag{1}$$
$$I_4 - I_3 - I_5 = 0 \tag{2}$$
$$I_6 - I_5 - I_7 = 0 \tag{3}$$
$$I_8 - I_7 - I_9 = 0 \tag{4}$$
$$I_{10} - I_9 - I_{11} = 0 \tag{5}$$
$$U_a + I_1 R_1 + I_2 R_2 = 0 \tag{6}$$
$$U_b - U_a - I_2 R_2 - I_4 R_4 = 0 \tag{7}$$
$$U_c - U_b + I_4 R_4 + I_6 R_6 = 0 \tag{8}$$
$$U_d - U_c - I_6 R_6 - I_8 R_8 = 0 \tag{9}$$
$$U_e - U_d + I_8 R_8 + I_{10} R_{10} = 0 \tag{10}$$
$$U_e + R_{10} I_{10} + R_{11} I_{11} = 0 \tag{11}$$

This 11x11 system of equations was then set up in matrix form, ensuring the required conditions of diagonal dominance in the LHS were met with the corresponding changes to the RHS as shown

$$\begin{pmatrix} 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -R_1 & -R_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & R_2 & 0 & R_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -R_4 & 0 & R_6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & R_6 & 0 & R_8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -R_8 & -R_{10} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -R_{10} & -R_{11} \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ I_6 \\ I_7 \\ I_8 \\ I_9 \\ I_{10} \\ I_{11} \end{pmatrix} = \begin{pmatrix} 0 \\ U_a \\ 0 \\ U_b - U_a \\ 0 \\ U_c - U_b \\ 0 \\ U_d - U_c \\ 0 \\ U_e - U_d \\ U_e \end{pmatrix}$$

This same system of linear equations was then kept constant throughout each task and solved using the various different methods outlined above, as the structural requirements of diagonal dominance was the same for all methods used.

1.5 Results

For each method used, we calculated the solution vector I, the net voltage V_T of the system which should be as close to 0 as possible and some performance metrics such as runtime and iteration counts for the iterative solvers.

Gaussian elimination
I = [-1.1198, 0.0599, 1.1797, 0.2201, -0.9596, -0.3134, 0.6463, 0.3600, -0.2862, -0.3880, -0.1018] A
V_T = [-1.4211e-14, 2.8422e-14, 0.0, 0.0, 0.0] V
Runtime = 0.0005s

Gauss-Seidel
I = [-1.1198, 0.0599, 1.1797, 0.2201, -0.9596, -0.3134, 0.6463, 0.3600, -0.2862, -0.3880, -0.1018] A
V_T = [1.4211e-14, -1.4211e-14, 2.8422e-14, 0.0, 0.0] V
Runtime = 0.0443s
Iterations = 71

Jacobi Method
I = [-1.1198, 0.0599, 1.1797, 0.2201, -0.9596, -0.3134, 0.6463, 0.3600, -0.2862, -0.3880, -0.1018] A
V_T = [0.0, 0.0, 2.8422e-14, 0.0, 0.0, 0.0] V
Runtime = 0.1967s
Iterations = 161

1.6 Discussion

While all of the methods used returned an almost identical solution vector, there were clear advantages in using the Gauss-Seidel method over the other iterative method since for the Jacobi method, non-zero initial guess parameters are a required input and if the wrong guess vector is passed in, there can be quite significant errors in the final solution vector output. This is a feature of the Jacobi method and is not a requirement of the Gauss-Seidel method, which takes initial guess parameters of $x_2 = x_3 = 0$, meaning there is less room for errors when using the Gauss-Seidel method. Moreover, the number of iterations required to converge to an accurate solution using the Jacobi method were much greater than the Gauss-Seidel method. In fact, the Gauss-Seidel method was around 2.5 times faster at an average 71 iterations vs the 161 of the Jacobi method which is also reflected in the runtime of each algorithm. The only real apparent benefit in using the Jacobi method is that it seems to have returned the V_T with values closest to 0 of the 3 methods but this is negligible as the threshold set is already relatively low.

One of the reasons for the slower performance of the Jacobi method in this case may be due in part to the lack of strict diagonal dominance due to the rows in which we have the equations obtained from Kirchoff's junction rule, as the true definition of diagonal dominance isn't met on these rows and as such the LHS matrix is only *somewhat* diagonally dominant. Moreover, since the Jacobi method requires an initial guess for x_1 before it can then begin approximating x_2, by also using approximations from previously stored iterations, it is fundamentally slower than the Gauss-Seidel algorithm which begins approximating at the very first step of calculation without the requirement of an initial guess for the first iteration.

For the circuit given however, the Gaussian elimination method outperforms both iterative methods in runtime by quite a significant amount as it is roughly 2 orders of magnitude faster than the Gauss-Seidel method and almost 400 times faster than the Jacobi method. This method does have it's disadvantages however, as it seems to mostly only perform well when used with smaller matrices. This is due to the nature of how the algorithm is performed, involving a relatively high number of processes which have a significant memory cost when solving systems of greater dimensions. More generally, when dealing with a large number of processes limited by machine precision, algorithms such as Gaussian elimination do not scale well when used in solving much larger systems of equations and so iterative solvers would be preferred in those cases.

2 Electrostatic Field in Two Dimensions

2.1 Finite-Difference Method

Method
Task 7 required us to set up the coefficient matrix **A** and vector **b** as shown in appendix I, for the 2D Poisson equation for electrostatic potential, derived from Gauss's law and Faraday's law:

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}$$

Because the magnetic fields do not vary with time, the electric field is curl free. Therefore the electric potential can be defined as a function:

$$\mathbf{E} = -\nabla\Phi$$

Combining the two previous functions:

$$\Delta\Phi + \frac{\rho}{\epsilon_0} = 0$$

Finally the electrostatic potential in 2D:

$$(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2})\Phi + \frac{\rho}{\epsilon_0} = 0$$

We were required to solve the electrostatic potential across a 2D grid using the finite difference method.

The finite difference method entails setting up a linear system of equations in matrix form. The system should be in the form:

$$\mathbf{Au} = \mathbf{b}$$

Where:
u is the $(M \cdot N)$ dimensional solution vector with components $u_{i,j}$
b is the $(M \cdot N)$ dimensional right hand side vector with components $b_{i,j}$
A is the $(M \cdot N) \times (M \cdot N)$ dimensional coefficient matrix with elements $A_{k,l,i,j}$

The structure of matrix A is:

$$\mathbf{A} = \begin{bmatrix} \mathbf{C} & \mathbf{I} & 0 & 0 & 0 & \cdots & 0 \\ \mathbf{I} & \mathbf{C} & \mathbf{I} & 0 & 0 & \cdots & 0 \\ 0 & \mathbf{I} & \mathbf{C} & \mathbf{I} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \mathbf{I} & \mathbf{C} & \mathbf{I} & 0 \\ 0 & \cdots & 0 & 0 & \mathbf{I} & \mathbf{C} & \mathbf{I} \\ 0 & \cdots & 0 & 0 & 0 & \mathbf{I} & \mathbf{C} \end{bmatrix}$$

Where **C** and **I** are:

$$\mathbf{C} = \begin{bmatrix} -4 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 1 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

After using the finite difference method to create our coefficient matrices, we ended with a linear system of equations that look like the matrix below (for the case M=N=3 as we were required to test this specific case in the instructions).

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{21} \\ u_{31} \\ u_{12} \\ u_{22} \\ u_{32} \\ u_{13} \\ u_{23} \\ u_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Our code scales with M=N=n, so all that is required is to change the value for n and the code will handle the rest. Also it is important to note that although the charge is positive, it must be negative in the b column vector as the charge is moved over to the RHS.

2.2 Solving the 2D Poisson equation with Successive Over-Relaxation and Gauss-Seidel Methods

2.2.1 Successive Over-Relaxation

Task 8 required us to build a Successive Over-Relaxation (SOR) algorithm as shown in appendix J. This algorithm was used to solve the system of equations as created in task 7. The premise of the SOR algorithm is to receive the already created **A** and **b** matrices, decompose the **A** matrix into; **D**, **L** and **U** matrices (diagonal, lower and upper, respectively) as already explained in section 1.3.2 Gauss-Seidel Method. Then solve for each value $u_{i,j}$ of the solution vector **u**, by using the equation:

$$\mathbf{x}^{k+1} = (\mathbf{D} + w_b\mathbf{L})^{-1}(w_b\mathbf{b} - [w_b\mathbf{U} + (w_b - 1)\mathbf{D}]\mathbf{x}^k)$$

Where w_b is the Optimal Relaxation Parameter which is the value for which the spectral radius is minimized and as such gives the fastest rate of convergence, and is described as:

$$w_b = \frac{2}{1 + \sqrt{1 - \rho(\mathbf{J})^2}}$$

Where $\rho(\mathbf{J})$ is the Spectral Radius of the Jacobi Iteration Matrix $\mathbf{J} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$. The Spectral Radius is simply the largest absolute eigenvalue of the Jacobi Iteration Matrix. Successive Over-Relaxation will converge for $1 < w < 2$.

Also where x^{k+1} is the current iteration of the solution and x^k is the previously calculated iteration of the solution. The initial value of x can be anything, we began at 0.

The SOR algorithm will run for a specified amount of iterations, or until convergence is achieved. We can determine if convergence has occurred when $|x^{k+1} - x^k| \leq 10^{-13}$

2.2.2 Gauss-Seidel Solver

The Gauss-Seidel algorithm was implemented the same as in section 1.3.2 and as shown in appendix F.

Results for M=N=7
As can be seen in figure 3, the potential ranges from $0.0165 < \Phi < 0.0551$ at the boundary as in the task 8 description. The potential field for M=7 was also visualised as shown in figure 4.

Figure 3: Solutions for N=M=7

```
 [0.01654412 0.03308824 0.04779412 0.05514706 0.04779412 0.03308824
 0.01654412 0.03308824 0.06801471 0.10294118 0.125      0.10294118
 0.06801471 0.03308824 0.04779412 0.10294118 0.17095588 0.23897059
 0.17095588 0.10294118 0.04779412 0.05514706 0.125      0.23897059
 0.48897059 0.23897059 0.125      0.05514706 0.04779412 0.10294118
 0.17095588 0.23897059 0.17095588 0.10294118 0.04779412 0.03308824
 0.06801471 0.10294118 0.125      0.10294118 0.06801471 0.03308824
 0.01654412 0.03308824 0.04779412 0.05514706 0.04779412 0.03308824
 0.01654412]
Optimal Relaxation Parameter:  1.4464626921717016
Iterations required for convergence:  40
```

2.2.3 Comparison of Successive Over-Relaxation and Gauss-Seidel Methods

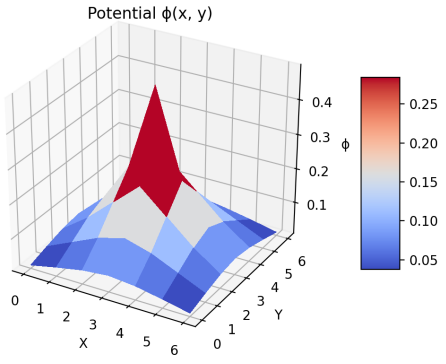
Finally task 8 asked us to compare the number of iterations required for convergence, between the SOR and Gauss-Seidel algorithms.

M=N	Iterations: Successive Over-Relaxation	Iterations: Gauss-Seidel	w_b
11	60	372	1.58879
21	107	1196	1.75083
31	154	-	1.82137
41	200	-	1.86093

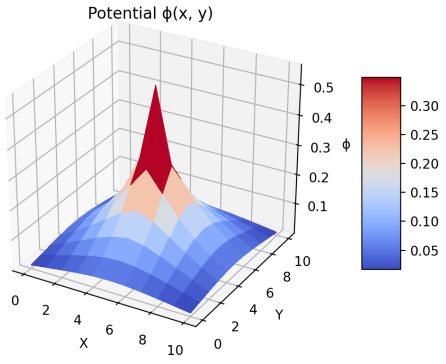
Here it is easy to see that the SOR algorithm becomes increasingly more effective than the Gauss-Seidel algorithm as the number of solutions M=N increases. In fact when running our Gauss-Seidel at dimensions of M=31 and M=41; we incurred high runtime and large memory costs without the use of Numba.

From figure 4 it can be seen that with increased M, the resolution increases dramatically. M=41 is plotted on the next page with M=51.

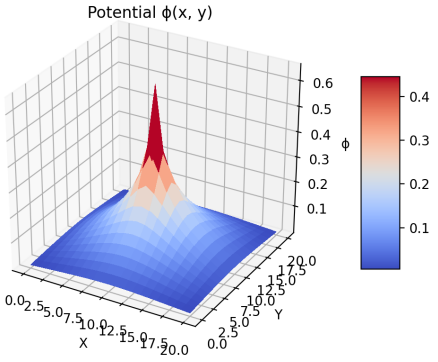
Figure 4: Electrostatic potential with increasing resolution



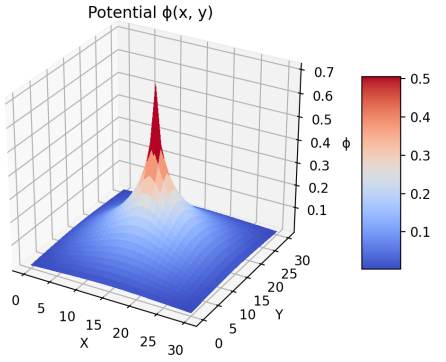
M=7



M=11



M=21



M=31

2.3 Increased Resolution N=M=51

Task 9 involved simply running the same code using SOR but for the case M=N=51.

- Total setup time : 9.7752e-06 s
- Spectral Radius: 0.9981755542233314
- Optimal Relaxation Parameter: 1.8861189705961687
- Iterations required for convergence: 246
- Time for program: 947.42s

Figure 5: Electrostatic Potential: M=41

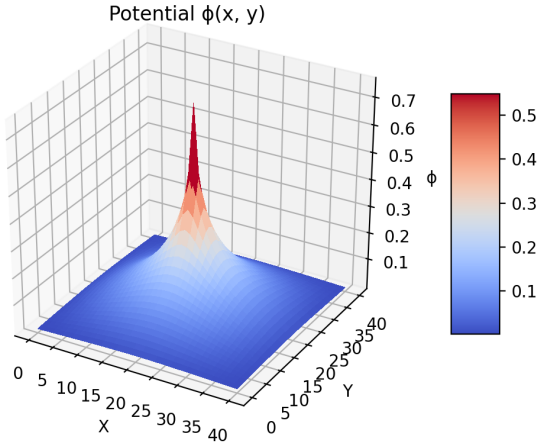
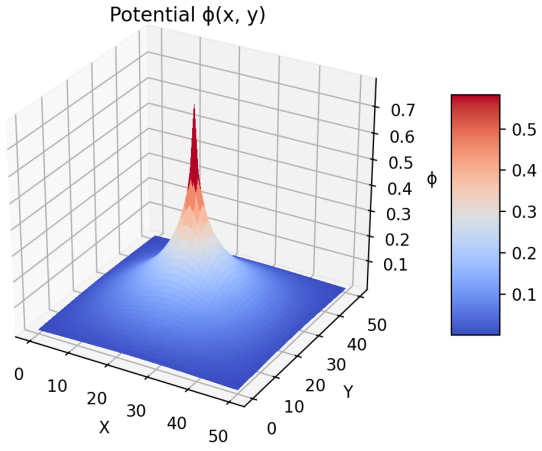


Figure 6: Electrostatic Potential: M=51



2.4 Electrostatic Potential With Dirichlet Boundary Conditions

In task 10 we implemented Dirichlet boundary conditions into the finite difference function as shown in appendix K. The difference between Dirichlet and von Neumann boundaries is that Dirichlet boundaries specify the value of the function on the surface, while von Neumann boundaries specify the derivative of the function on the surface.

The boundaries at the top and right hand sides, excluding (x_0, y_0) and (x_{M+1}, y_{N+1}):

$$\Phi|_{x=x_{M+1}, y=[y_0, \cdots, y_N]} = \Phi|_{x=[x_1, \cdots, x_{M+1}], y=y_0} = -0.3$$

The boundaries at the bottom and left hand sides, excluding (x_0, y_0) and (x_{M+1}, y_{N+1}):

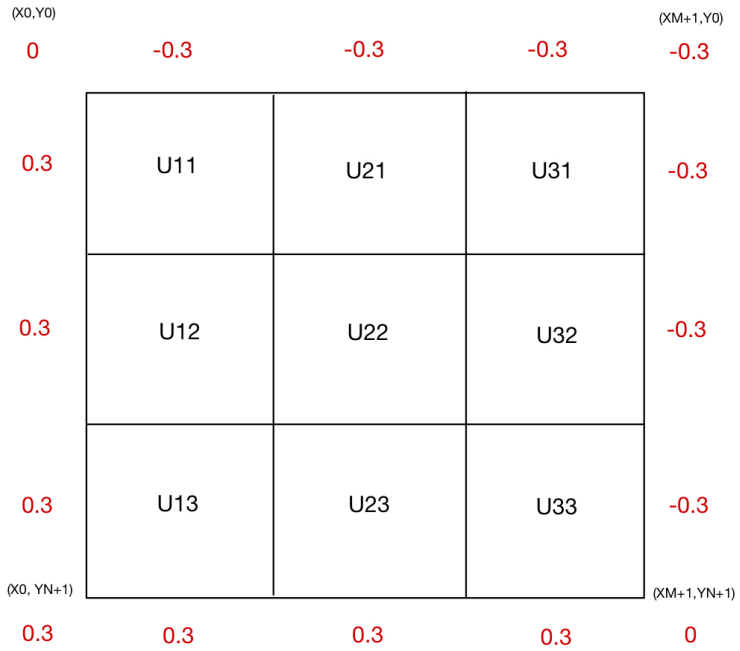
$$\Phi|_{x=x_0, y=[y_1, \cdots, y_{N+1}]} = \Phi|_{x=[x_1, \cdots, x_M], y=y_{N+1}} = +0.3$$

The corners at (x_0, y_0) and (x_{M+1}, y_{N+1}):

$$\Phi|_{x=x_0, y=y_0} = \Phi|_{x=x_{M+1}, y=y_{N+1}} = 0$$

This was then visualised with the use of a grid like shown in figure 7 below for the case N=M=3:

Figure 7: Visualisation of the grid for M=3 with Dirichlet boundaries imposed.



To apply these boundaries to the system of equations, the matrix **b** can be modified:

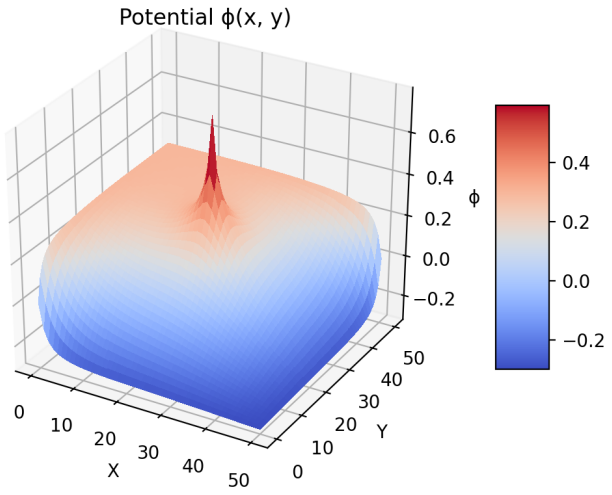
$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{21} \\ u_{31} \\ u_{12} \\ u_{22} \\ u_{32} \\ u_{13} \\ u_{23} \\ u_{33} \end{bmatrix} = \begin{bmatrix} \Delta^2\rho_{11} - u_{10} - u_{01} \\ \Delta^2\rho_{21} - u_{20} \\ \Delta^2\rho_{31} - u_{41} - u_{30} \\ \Delta^2\rho_{12} - u_{02} \\ \Delta^2\rho_{22} \\ \Delta^2\rho_{32} - u_{42} \\ \Delta^2\rho_{13} - u_{14} - u_{03} \\ \Delta^2\rho_{23} - u_{24} \\ \Delta^2\rho_{33} - u_{43} - u_{34} \end{bmatrix}$$

Since ρ is a point charge, we can describe the values of the terms including ρ using the Dirac delta function, which tells us that the ρ terms will be 0 everywhere other than at the centre of the charge. Using figure 7, we are able to visualise the elements of interest for our system of equations for the case M=3.

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{21} \\ u_{31} \\ u_{12} \\ u_{22} \\ u_{32} \\ u_{13} \\ u_{23} \\ u_{33} \end{bmatrix} = \begin{bmatrix} 0 - (-0.3) - 0.3 \\ 0 - (-0.3) \\ 0 - (-0.3) - (-0.3) \\ 0 - 0.3 \\ -1 \\ 0 - (-0.3) \\ 0 - 0.3 - 0.3 \\ 0 - 0.3 \\ 0 - (-0.3) - 0.3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.3 \\ 0.6 \\ -0.3 \\ -1 \\ 0.3 \\ -0.6 \\ -0.3 \\ 0 \end{bmatrix}$$

These boundaries were once again implemented using nested for loops and conditionals to determine the correct indices to apply each boundary condition. The example outlined was for the case M=3, however in figure 8, we visualised the potential field as a result of these boundary conditions for the case M=51, to generate a much higher resolution image and more accurate description of the potential field of a point charge under these specific boundary conditions.

Figure 8: Electrostatic Potential with Dirichlet boundaries for N=M=51



Appendices

A Table of Values

Quantity	Name	SI Unit
$\mathbf{E}$	electric field	$V \cdot m^{-1}$
ρ	charge density	$A \cdot s$
ϵ_0	vacuum permittivity	$8.85 \times 10^{-12} A^2 \cdot s^4 \cdot kg^{-1} \cdot m^{-3}$
$\mathbf{B}$	magnetic flux density	$kg \cdot A^{-1} \cdot s^{-2}$
t	time	s
μ_0	vacuum permeability	$1.25 \times 10^{-6} kg \cdot m \cdot s^{-2} \cdot A^{-2}$
J	Joule	$C \cdot \boldsymbol{V}$
$\mathbf{H}$	Magnetic field intensity	$\boldsymbol{B}\mu^{-1} - M$
$\mathbf{M}$	Magnetisation	$A \cdot m^{-1}$
$\mathbf{D}$	Electic flux density	$C \cdot m^{-2}$
V	Volt	Kg $\cdot m^2 \cdot s^{-3} \cdot A^{-1}$
Q	charge	C
I	Electric Current	A
ℓ	length	m
ϕ	Magnetic flux	Wb
I_1	Current 1	$-\frac{12}{11}$A
I_2	Current 2	$-\frac{16}{11}$A
I_3	Current 3	$-\frac{4}{11}$A
R_1	Resistor 1	100Ω
R_2	Resistor 2	200Ω
R_3	Resistor 3	300Ω
R_4	Resistor 4	400Ω
R_6	Resistor 6	600Ω
R_8	Resistor 8	800Ω
R_{10}	Resistor 10	1000Ω
R_{11}	Resistor 11	1100Ω
U	Voltage source	400 V
U_a	Voltage source	100 V
U_b	Voltage source	200 V
U_c	Voltage source	300 V
U_d	Voltage source	400 V
U_e	Voltage source	500 V

Table 1: Here you will find all of the SI units presented in the order they appear within the document.
The Key to the notation : **Bold** quantities represent vectors, whereas regular quantities are scalars. The unit of each quantity and the universal constants is given in the *Système International d'Unité* (SI units)

B Circuit

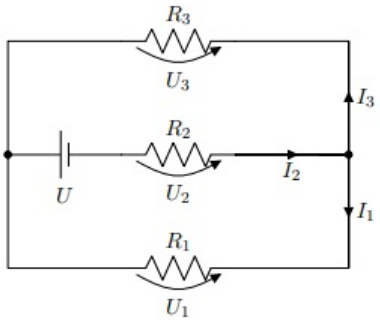


Figure 9: circuit system

C Extended Circuit

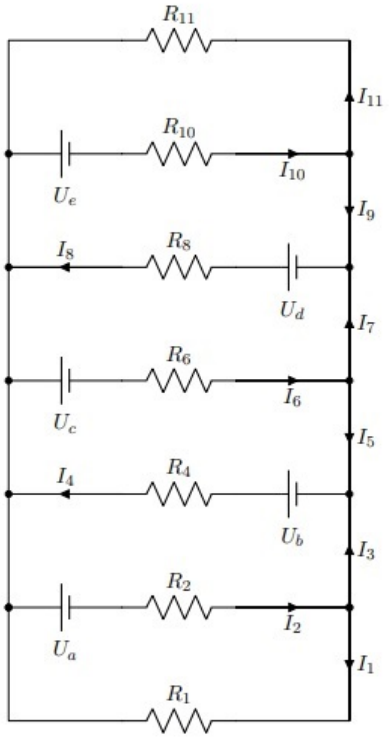


Figure 10: extended circuit system

D Gaussian Elimination code

```
def gaussian_elimination(m, b):
    """
    Solves a system of linear equations by gaussian elimination for input matrices
    A and b, returning solution vector x.
    """

    n = len(b)
    x = np.zeros(n)
    A, w = pivot(m.copy(), b.copy())  # pivot

    for i in range(n):  # sets coeff = a_ji/a_ii, subtracts this val from jth row for each k element, reducing matrix
        for j in range(i+1, n):
            if A[j][i] == 0:  # if element is already 0, continue
                continue

            coeff = A[j][i]/A[i][i]  # sets the coefficient term to be subtracted in producing augmented A
            w[j] = (w[j] - coeff * w[i])  # calculates corresponding rhs vals

            for k in range(i, n):
                A[j][k] = A[j][k] - coeff * A[i][k]  # multiplies each row by coeff and subtracts it from each
                    element

    x[n-1] = w[n-1] / A[n-1][n-1]  # solves the last row of the upper triangle matrix for X

    for i in range(n-1, -1, -1):  # moves backwards to -1 in steps of -1
        sum = 0  # starts counter for the sum
        for j in range(i+1, n):
            sum += A[i][j] * x[j]  # computes sum

        x[i] = (w[i] - sum)/A[i][i]  # solves for given x by subtracting unwanted coefficient

    return x
```

E Pivot

```
def pivot(m, b):
    """
    Swaps rows to ensure diagonal elements contain the largest values. This returns a diagonally dominant matrix for
        matrices which can be made diagonally dominant and if not, returns the original matrix with warning.
    """

    n = len(b)
    A = m.copy()

    for row_swaps in range(n):  # ensures the function runs enough times to carry out all potential row swaps
        for i in range(n):
            for j in range(n):
                if abs(A[i][i]) < abs(A[i][j]): # condition that the diagonal is largest, if it's not, perform row
                    switches
                    A[[i, i+1]] = A[[i+1, i]] # row switches
                    b[[i, i+1]] = b[[i+1, i]]

    for rows in range(n):
        if abs(A[i][i]) < abs(A[i][j]):  # if the diagonal is still lower than off diagonals, then the matrix isn't
            diagonally dominant
            print('Warning: Matrix not fully diagonally dominant')

    return A, b
```

F Gauss-Seidel code

```
def gauss_seidel(m, b, iterations=100, x=None, convergence=1e-13):
    """
    Gauss-Seidel iteration method for input matrices A and b. Returns solution vector x
    & iterations required for convergence, respectively.

    The method sets up the D, L & U matrices first. Then calculates the spectral radius, sr. By finding the largest
    absolute eigenvalue of the Jacobi Iteration Matrix.

    With sr found, this function iterates over each element of the solution vector (if sr < 1) using
    the equation given in the lecture slides. And it does this for a specified amount of iterations,
    defined in the function call.
    """

    # Sets up initial 0 matrices for D, L & U. Also sets up n.
    n = len(b)
    D = np.zeros((n, n))
    L, U = D.copy(), D.copy()
    A, b = pivot(m.copy(), b.copy()) # not strictly diagonally dominant but performs a partial pivot

    if x is None:
        x = np.zeros(n)

    # Find diagonal, lower and upper matrices D, L, U respectively
    for i in range(n):
        for j in range(n):
            if j == i:
                D[i, j] = A[i, j]
            elif j-i > 0:
                L[i, j] = A[i, j]
            else:
                U[i, j] = A[i, j]

    # Finds spectral radius, using eigenvalues of Jacobi iteration matrix
    j = np.dot(np.linalg.inv(D), (L + U))
    eigvals, eigvec = np.linalg.eig(j)
    sr = np.absolute(eigvals).max()

    # If spectral radius < 1, iterates through input number of iterations and for
    # each solution, checks to see if the solution has converged. If the convergence
    # threshhold of 1e-13 has been met, the loop ends.
    z = [x]
    convergence_counter = 0
    updated_counter = i
    if sr < 1:
        for j in range(n):
            for i in range(1, iterations):
                x = np.dot(np.linalg.inv(D + L), (b - np.dot(U, x)))
                z.append(np.array(x))
                if np.absolute(z[i][j] - z[i-1][j]) < convergence:
                    print('Solution vector component {} converges at N={}'.format(j+1, i))
                    updated_counter = i
                    break
                else:
                    continue

            # Updates the counter that tracks largest amount of iterations, when the function ends it gives
            # iterations required for convergence
            if updated_counter > convergence_counter:
                convergence_counter = updated_counter

            if np.absolute(z[i][j] - z[i-1][j]) > convergence: # checks the very last iteration for convergence to
                 check for non-convergence due to low iteration count
                print('Low iteration count')
            else:
                continue
    else:
        print('No solutions')

    return x, convergence_counterframe
```

G Jacobian Code

```
def jacobi(m, b, iterations=100, guess=None, convergence=1e-13):
    """
    Systems of linear equations solver using jacobian iteration method for input matrices A and b, initial guess
        parameter for x and a desired convergence threshold returning solution vector x.
    """

    n = len(b)
    D = np.zeros((n, n))  # empty 0 array to be used in creating diagonal matrix
    A, b = pivot(m.copy(), b.copy())  # not strictly diagonally dominant but performs a partial pivot

    if guess is None:
        x = np.zeros(n)  # if no guess vector entered, creates 1
    else:
        x = guess

    for i in range(n):
        for j in range(n):
            if i == j:
                D[i][j] = A[i][j] # conserves the diagonal elements of A, leaving off diags as 0
            else:
                continue

    LU = A - D  # removes the diagonal elements leaving a lower upper matrix
    j = np.dot(np.linalg.inv(D), LU)  # computes the iteration matrix j
    eigvals, eigvec = np.linalg.eig(j)  # computes eigenvalues of the iteration matrix j
    sr = np.absolute(eigvals).max()  # calculates spectral radius, must be < 1 for convergence

    # If spectral radius < 1, iterates through input number of iterations and for each solution, checks to see if the
        solution has converged.
    z = [x]
    if sr < 1:  # if SR meets convergence requirements, continues to compute solution vector
        for j in range(n):
            for i in range(1, iterations):
                x = np.dot(np.linalg.inv(D), (b - np.dot(LU, x)))  # calculates solution vector element x
                z.append(np.array(x))
                if np.absolute(z[i][j] - z[i-1][j]) < convergence: # convergence checker with threshhold 1e-13
                    print('Solution vector component {} converges at N={}'.format(j+1, i))
                    break
                else:
                    continue

            if np.absolute(z[i][j] - z[i-1][j]) > convergence: # checks the last iteration for convergence, if not
                converged yet, low count
                print('Low iteration count')
            else:
                continue
    else:
        print('No solutions')  # no solutions if SR > 1

    return x
```

H Section 1 Tasks Code

```
import numpy as np
import PH388functions as PH388

# Task 4, 5 and 6 - Gaussian elimination, Gauss-Seidel and Jacobi's methods extended to solve the given circuit

R1, R2 = 100., 200.  # known resistance values for the circuit
R4, R6 = 400., 600.
R8, R10, R11 = 800., 1000., 1100.
Ua, Ub = 100., 200. # known voltage values for the circuit
Uc, Ud, Ue = 300., 400., 500.

R = np.array([

[-R1, -R2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, R2, 0, R4, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, -R4, 0, -R6, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, R6, 0, R8, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, -R8, 0, -R10, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, -R10, -R11],
[1, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, -1, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, -1, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 1, -1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 1]
])

U = np.array([Ua, Ub-Ua, Uc-Ub, Ud-Uc, Ue-Ud, Ue, 0, 0, 0, 0, 0])
I_ge = PH388.gaussian_elimination(R, U)
I_gs = PH388.gauss_seidel(R, U)[0]
I_jc = PH388.jacobi(R, U, 1000, guess=np.ones((len(U),)))

# Checking zero-ness of each loop, all are zero or extremely close to zero
# Gaussian Elimination
a = -R1*I_ge[0] - R2*I_ge[1] - Ua
b = R2*I_ge[1] + R4*I_ge[3] - Ub + Ua
c = -R4*I_ge[3] - R6*I_ge[5] - Uc + Ub
d = R6*I_ge[5] + R8*I_ge[7] - Ud + Uc
e = -R8*I_ge[7] - R10*I_ge[9] - Ue + Ud
f = -R10*I_ge[9] -R11*I_ge[10] - Ue

print('gaussian_elimination solution vector I:\n', I_ge)
print('Zero-ness of each loop:\n', a, b, c, d, e)

print(I_gs)
# Gauss-Seidel
a = -R1*I_gs[0] - R2*I_gs[1] - Ua
b = R2*I_gs[1] + R4*I_gs[3] - Ub + Ua
c = -R4*I_gs[3] - R6*I_gs[5] - Uc + Ub
d = R6*I_gs[5] + R8*I_gs[7] - Ud + Uc
e = -R8*I_gs[7] - R10*I_gs[9] - Ue + Ud
f = -R10*I_gs[9] -R11*I_gs[10] - Ue

print('Gauss-Seidel solution vector I:\n', I_gs)
print('Zero-ness of each loop:\n', a, b, c, d, e)

# Jacobi method
a = -R1*I_jc[0] - R2*I_jc[1] - Ua
b = R2*I_jc[1] + R4*I_jc[3] - Ub + Ua
c = -R4*I_jc[3] - R6*I_jc[5] - Uc + Ub
d = R6*I_jc[5] + R8*I_jc[7] - Ud + Uc
e = -R8*I_jc[7] - R10*I_jc[9] - Ue + Ud
f = -R10*I_jc[9] -R11*I_jc[10] - Ue

print('Jacobi solution vector I:\n', I_jc)
print('Zero-ness of each loop:\n', a, b, c, d, e)
```

I Tasks 7-9: Finite Difference Method

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import PH388functions as PH388


def finiteDifference(M, rho, epsilon, delta=1,):
    """
    This function sets up the matrices A and b. The overall method for matrix A is done by
    using two nested for loops, the inner loop runs through each column and the outer loop
    runs through each row. These for loops assign the correct number to each specific index
    of the matrix A.

    The procedure for b is much more simple, the charge everywhere is 0 except the middle index
    so the function simply finds the middle index and inserts -1 there.
    (The charge is +1 but the equation moves the charge to the RHS so it is -1 in the code.)
    """

    # Setting up initial matrix and length
    m = np.zeros((M, M))
    n = len(m)
    A = np.zeros((n*n, n*n))

    # Nested for loops assigning correct values to each index
    for i in range(len(A)):
        for j in range(len(A)):
            if i == j:
                A[i][j] = -4
            elif i == j + 1:
                A[i][j] = 1
            elif i == j + n:
                A[i][j] = 1
            elif i == j - 1:
                A[i][j] = 1
            elif i == j - n:
                A[i][j] = 1
            else:
                continue

    for i in range(1, len(A)):
        for j in range(1, len(A)):
            if i % n == 0 and j == i - 1:
                A[i][j] = 0
            elif j % n == 0 and i == j - 1:
                A[i][j] = 0

    # creating grid, this is a bit obsolete here since np.arange does
    # the same job for delta = 1 but including for completeness atm

    xgrid = np.arange(0, (len(m)), delta)
    ygrid = xgrid.copy()

    # Setting up b column vector by finding middle index
    b = np.zeros((n*n, 1))
    midb = int((n*n)/2)
    b[midb] = -rho/epsilon

    return A, b, xgrid, ygrid

# Initial variables
rho = epsilon = delta = 1
M = 51 # desired dimensions

# Function calls
A, b, x, y = finiteDifference(M, rho, epsilon, delta)  # takes M = N value as argument
U, w, i = PH388.SOR(A, b.flatten())
#U2, i2 = PH388.gauss_seidel(A, b.flatten(), 1000)

# Print values
print('Solution Vector: ', U)
print('Optimal Relaxation Parameter: ', w)
print('Iterations for convergence :', i)

# Plots
fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.meshgrid(x, y)
U = U.reshape((M, M))
```

```
surf = ax.plot_surface(X, Y, U, rcount=100, ccount=100, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.55, aspect=5)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('  ')
ax.set_title('Potential   (x, y)')
plt.show()
```

J Tasks 8-10: Successive Over-Relaxation Code

```
## Successive Over Relaxation method

def SOR(A,b, iterations=10000, x=None, convergence=1e-13):
    """
    Successive Over Relaxation iteration method for input matrices A and b. Returns solution vector x,
    optimal relaxation parameter & iterations required for convergence, respectively.

    The method sets up the D, L & U matrices first. Then calculates the optimal relaxation parameter
    by finding the spectral radius of the Jacobi iteration matrix. i.e. the largest absolute eigenvalue
    of the Jacobi iteration matrix. Then uses the equation given in the lecture slides to calculate
    the optimal relaxation parameter, w.

    With w found this function iterates over each x value if 1 < w < 2, there should be n of them. And it does this
    for a specified amount of iterations, defined in the function call.
    """

    # Sets up initial 0 matrices for D, L & U. Also sets up n.
    n = len(b)
    if x is None:
        x = np.zeros(len(b))

    D = np.zeros((n, n))
    L = D.copy()
    U = D.copy()

    # Find diagonal and lower and upper matrices D, L, U respectively
    for i in range(n):
        for j in range(n):
            if j == i:
                D[i, j] = A[i, j]
            elif j-i > 0:
                U[i, j] = A[i, j]
            else:
                L[i, j] = A[i, j]

    # Relaxation parameter W calculation

    J = np.dot(np.linalg.inv(D), (L + U))
    eigvals, v = np.linalg.eig(J)
    eigvals_abs = np.absolute(eigvals)
    sr = np.max(eigvals_abs)
    print('Spectral Radius: ', sr)
    w = 2 / (1 + (1-sr**2)**0.5)

    # Calculates terms 1, 2 and 3 as required for the final equation for x(k+1)
    # because the equation is long
    t1 = np.linalg.inv(D + (w * L))
    t2 = (w * b)
    t3 = (w * U + (w - 1) * D)

   # If optimal relaxation parameter: 1 < w < 2; iterates through input number of iterations and for
    # each solution, checks to see if the solution has converged. If the convergence
    # threshhold of 1e-13 has been met, the loop ends.
    if 1 < w < 2:
        z = [x]
        convergence_counter = 0
        for j in range(n):
            for i in range(iterations):
                x = np.dot(t1, (t2 - np.dot(t3, x)))
                z.append(np.array(x))
                if np.absolute(z[i][j] - z[i-1][j]) < convergence:
                    updated_counter = i
                    break
                else:
                    continue
```

```
            # Updates the counter that tracks largest amount of iterations, when the function ends it gives
            # iterations required for convergence
            if updated_counter > convergence_counter:
                convergence_counter = updated_counter

            # checks the very last iteration for convergence to check for non-convergence due to low iteration count
            if np.absolute(z[i][j] - z[i-1][j]) > convergence:
                print('Low iteration count')

            else:
                continue

    else:
        print('No Solutions.')

    return x, w, convergence_counter
```

K Task 10: Finite Difference Method With Dirichlet Boundaries Code

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import PH388functions as PH388

def finiteDifferenceDirichlet(m, rho, epsilon, delta, topBound, rightBound, leftBound, bottomBound):
    """
    This function sets up the matrices A and b. The overall method for matrix A is done by
    using two nested for loops, the inner loop runs through each column and the outer loop
    runs through each row. These for loops assign the correct number to each specific index
    of the matrix A.

    The procedure for b is to assign boundary conditions to each node. b is created as a
    matrix first so its more easy to visualise, then using nested for loops, like in matrix A,
    it loops through each column of each row and assigns each node its respective boundary value.
    Then b is flattened into an array.
    The boundary conditions can be changed in the function call for other problems to do
    with Dirichlet boundarys.
    """

    # Set up initial matrix A and value n.
    n = len(m)
    A = np.zeros((n*n, n*n))

    # Nested for loop assigning correct values to each index
    for i in range(len(A)):
        for j in range(len(A)):
            if i == j:
                A[i][j] = -4
            elif i == j + 1:
                A[i][j] = 1
            elif i == j + n:
                A[i][j] = 1
            elif i == j - 1:
                A[i][j] = 1
            elif i == j - n:
                A[i][j] = 1
            else:
                continue

    for i in range(1, len(A)):
        for j in range(1, len(A)):
            if i % n == 0 and j == i - 1:
                A[i][j] = 0
            elif j % n == 0 and i == j - 1:
                A[i][j] = 0

    # Creating grid, this is a bit obsolete here since np.arange does the
    # same job for delta = 1 but including for completeness atm
    xgrid = np.zeros((len(m)))
    delta = 1
    for i in range(1, len(xgrid)):
        xgrid[i] = xgrid[i-1] + delta

    ygrid = xgrid.copy()

    # Vector b is simply the flat version of boundary matrix
    boundaryMatrix = np.zeros((n, n), float)

    # Assign each index of boundary matrix with the correct boundary
    # conditions, loop i through each row, j through each column
    for i in range(n):
        for j in range(n):

            # TOP LEFT
            if i == 0 and j == 0:
                boundaryMatrix[i][j] = 0

            # ONE BOUNDARY TOP ROW
            if i == 0 and j != 0 and j != n-1:
                boundaryMatrix[i][j] = - topBound

            # TWO BOUNDARYS TOP RIGHT
            if i == 0 and i % n == 0 and j == n-1:
                boundaryMatrix[i][j] = - topBound - rightBound

            # ONE BOUNDARY LEFT COLUMN
            if j == 0 and i != 0 and i != n-1:
```

```
                boundaryMatrix[i][j] = - leftBound

            # ONE BOUNDARY BOTTOM ROW
            if i == n-1 and j != 0 and j != n-1:
                boundaryMatrix[i][j] = - bottomBound

            # ONE BOUNDARY RIGHT COLUMN
            if i != 0 and i != n-1 and j == n-1:
                boundaryMatrix[i][j] = - rightBound

            # TWO BOUNDARYS BOTTOM LEFT
            if i == n-1 and j == 0:
                boundaryMatrix[i][j] = - bottomBound - leftBound

            # BOTTOM RIGHT
            if i == n-1 and j == n-1:
                boundaryMatrix[i][j] = 0

            # CENTER
            if i == int(n/2) and j == int(n/2):
                boundaryMatrix[i][j] = -rho/epsilon

    # Flatten boundary matrix as b should be an array
    b = boundaryMatrix.flatten()

    return A, b, xgrid, ygrid

# Setup the initial variables
M = N = 11
rho = epsilon = delta = 1
topBound = rightBound = -0.3
bottomBound = leftBound = 0.3
m = np.zeros((M, N))

# Call functions
A, b, x, y = finiteDifferenceDirichlet(m, rho, epsilon, delta, topBound, rightBound, leftBound, bottomBound)
U, w, i = PH388.SOR(A, b.flatten())

midU = int(M**2/2)

# Print values
print('Solution vector: ', U)
print('Optimal Relaxation Parameter: ', w)
print('Iterations Required (SOR): ', i)
print('Peak: ', U[midU])

# Plots
fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.meshgrid(x, y)
U = U.reshape((M, M))
surf = ax.plot_surface(X, Y, U, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.55, aspect=5)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('   ')
ax.set_title('Potential   (x, y)')

plt.show()
```

References

[1] Maxwell, J.C, 1865, 'A Dynamical Theory of the Electromagnetic Field', Philosophical Transactions of the Royal Society of London 155, 459–512.

[2] Kirchhoff, G, 1845, "Ueber den Durchgang eines elektrischen Stromes durch eine Ebene, insbesonere durch eine kreisförmige," [Mitglied des physikalischen Seminars zu Konigsberg] Annalen der Physik und Chemie, Vol. 64, No. 4, pp. 487 - 514