1ST YEAR PROJECT REPORT

Ziyad Ansari - CID: 01897568

Ismail Sajid - CID: 01924979

Oliver Thompson - CID: 01906133

ELEC40006 – Electronics Design Project 2020-21

CIRCUIT SIMULATOR

Word count: 8718

INTRODUCTION

BACKGROUND AND TASK

In the modern era of electronic circuit design, circuit simulators are becoming increasingly important for engineers. They are used for various purposes, with the most common example being to test the efficiency and outputs of newly constructed circuits and gauge whether they meet desired requirements. This is a key step in any analogue or digital circuit design process because it provides valuable insight and feedback into the possible flaws that a circuit may have as well as any potential improvements that could be added to achieve the desired output in a more efficient way. Crucial to this process is the fact that simulators eliminate the requirement for early investment in the expensive circuit components in case they may be in excess, and in general it saves resources in the process. In addition to this established use of circuit simulators, they are also extremely useful in academia in many engineering programs, as they provide useful visualizations of any learning material which enhances the teaching aspect of education greatly, as well as students' ability to retain the information.

Our task for this project selection was to essentially replicate the circuit simulation software we used for our various first year lab tasks and eventual self-understanding, LTSpice, by writing our own program to carry out the small-signal AC simulation of a circuit.

HOW IT WORKS

The main principle of our program was that we would be carrying out nodal analysis with the use of matrices. The user would need to input the parameters and components of a particular circuit into the program, and then the program would obtain results akin to those which LTSpice, or any other existing simulation software, would produce. The program will carry out the nodal analysis process using a conductance matrix, a nodal voltages vector and a currents vector. The latter two vectors are of the matrices of the order nx1, where n is the total number of nodes in the circuit. The elements of the

conductance (G) matrix are total conductance between any two nodes. Each element labelled by its position can also represent between which nodes it measures the conductance; for example, G_{ij} will represent the total conductance between nodes i and j and it will also be the element of the

matrix at a position row i and column j. Kirchhoff's current law is followed, and the inverse of the conductance matrix is then found by the program using the Gaussian elimination method, and multiplied by the current/voltage matrix to give the vector of unknown voltages/currents labelled v1, v2, etc. according to the node they represent, as shown below.

$$\begin{bmatrix} G_{11} & -G_{12} & \dots & -G_{1n} \\ -G_{21} & G_{22} & \cdots & -G_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -G_{n1} & -G_{n2} & \dots & G_{nn} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix}$$

The program will calculate the unknown values of the circuit using this method of Gaussian elimination.

A special case had to be considered for when a voltage source was present in the circuit; if one end of the source was connected to a reference node then the first node voltage can be considered as equal to V(source). Since a voltage source has an infinite conductance, it can be ignored, and the matrix is written like this:

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ -G_{21} & G_{22} & \cdots & -G_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -G_{n1} & -G_{n2} & \dots & G_{nn} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_{\text{src}} \\ i_2 \\ \vdots \\ i_n \end{bmatrix}$$

For the case where the voltage source is connected between two nodes which are non-reference nodes, the voltage vector of unknowns is appended with an additional current, I(Vsource) which is the unknown current in the voltage source. This allows for the writing of a KCL equation for each node, because the conductance is incalculable and so we directly add the current to the matrix.

THE DESIGN CRITERIA

Our program design had several criteria which needed to be met. The specifications for the components and information about the circuit were provided as a netlist in a text file in reduced SPICE format. The netlist would describe the circuit in individual strings; the first character in each line designates the component type (resistor, capacitor, etc.). The nodes to which this specific component is connected to are given next in the same line, specified in order. Lastly, the final element in each line represents the value of the component, and it can contain multipliers depending on the size of the value. The timestep and stop time are also denoted within each netlist file. A transient analysis begins, and at each timestep, calculations for voltages and currents take place and this process only terminates at the stop time.

The analysis produces an output, and this is written into a formatted comma-separated-values file which has each frequency step as a row. A MATLAB script is used to provide an accurate representation of the results and plot them on a graph.

We kept our desired maximum processing time at 1 second.

TEAM DISCUSSIONS

We held our meetings on Microsoft Teams, making an entirely new team just for the project. This helped particularly because we were able to share links and any other materials easily using the Teams channel and they were accessible without difficulty, as we had grown accustomed to it throughout the year. Meetings were held at the beginning to brainstorm our program design, as well as to browse the internet for any potentially useful links. We also held meetings in the final days leading up to the deadline to edit our report together and ensure this and the video were cohesive and informative. We took the help of Teaching Assistants in advice sessions for help with the design of our code. A WhatsApp group was also created where we updated each other on our progress and discussed our project plans in further detail.

IMPLEMENTATION AND DESIGN

Our first and foremost task was to download the relevant libraries in order to carry out the mathematical operations we needed for the circuit simulator. This mainly involved matrix operations, as well as basic arithmetic at ease and finding inverses of matrices. The C++ library did not contain any matrix operations and therefore we had to research online for one.

DESIGN PROCESS

INITIAL RESEARCH

Some of the initial aspects of the project we had to consider were:

- What were the main tasks our program was to perform, and how would it achieve this?
- What libraries were available for this task, and how would we choose between them?
- How would we carry out the task in the most efficient way?

Our program's main task was to calculate unknown nodal voltages and currents of a given circuit and it would use matrix multiplication and the process of Gaussian elimination to calculate the results. We would be achieving this by writing an algorithm that would take the conductance and known values of the circuit and perform the necessary elimination to find the inverse of the matrix and obtain the values of the unknowns.

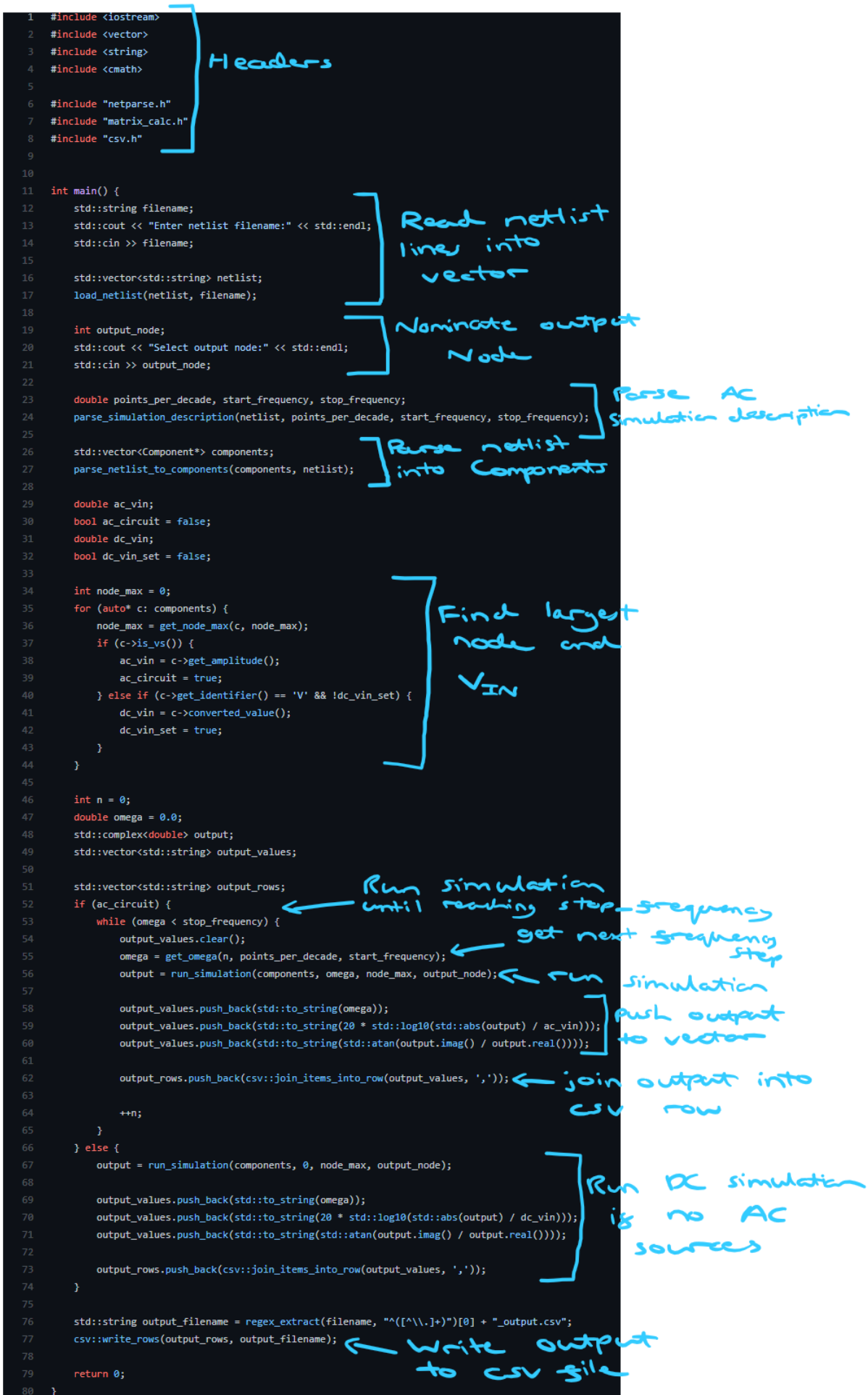
The library we chose was to use Eigen, because it seemed the most relevant to our project, and could perform any matrix-related operations without much hassle or difficulty. Eigen was also easy to download and use so we could begin using it in our IDE straight away. It is also a very cleanly presented library and was very naturally implemented into our code. Eigen's versatility and ability to handle matrices of all sizes and types was also crucial to our decision because we wanted our circuit simulator to be able to work with as many nodes as possible.

We scheduled several calls with Professor Ed Stott and other teaching assistants in order to ensure implementation maintained a high level of

efficiency, as well as to help with any issues we encountered in our design. We also collectively revised each function we defined as well as other elements of our code, so that there was minimal redundancy.

IMPLEMENTATION

OVERVEW OF MAIN


Headers
Read netlist lines into vector
Nominate output Node
Parse AC simulation description
Parse netlist into Components
Find largest node and VIN
Run simulation until reaching stop frequency
get next frequency step
run simulation
push output to vector
join output into csv row
Run DC simulation if no AC sources
Write output to csv file

PARSING THE NETLIST FILE

READING THE FILE

Reading the netlist file was relatively simple. To do this, a function was defined – load_netlist – which takes the netlist filename and an output vector and populates the vector with each line in the file, skipping any comments (lines beginning with an asterisk). The function stops reading lines after the terminator line is found (the line containing ".end").

```
// Reads the given netlist file and returns a reference to a vector of lines in the file
void load_netlist(std::vector<std::string>& out_vector, const std::string& filename) {
    std::ifstream infile(filename);

    std::string line;
    while (std::getline(infile, line)) {

        if (line.rfind('*', 0) == 0) {
            // Skip to the next line in the file if the current one is a comment
            continue;
        } else if (line.rfind(".end", 0) == 0) {
            // Stop looking for extra lines if .end has been found by breaking out of the loop
            break;
        }

        out_vector.push_back(line);
    }
}
```

PARSING NETLIST LINES

Before any analysis can be performed, the netlist must be parsed into a form that can be easily processed. To do this, a class – Component – was created to represent each component in the circuit.

```
class Component {
public:
    Component(ParseReturn* parsed_line);
    char get_identifier() const {return identifier;};
    int get_id() const {return id;};
    std::string get_value() {return value;};
    bool has_converted_value();
    double converted_value();
    std::vector<int> get_nodes() {return nodes;};
    bool is_vs();
    float get_amplitude();
    float get_phase();

private:
    char identifier;
    int id;
    std::string value;
    std::vector<int> nodes;
};
```

In order to parse a single netlist line into a Component object, the line is first run through a custom parser which takes the netlist format and splits it on whitespace into parts, except for the component value which must take into account the brackets for an AC voltage source. Below you can see the parsing code which takes in the netlist line and populates the output vector with the different parts.

```cpp
void split_netlist_line(std::vector<std::string>& segments, const std::string& line) {
    int stack = 0;
    std::string current_segment;

    for (char c: line) {
        if (c == ' ' && !stack) {
            if (!current_segment.empty()) {
                segments.push_back(current_segment);
                current_segment.clear();
            }
            continue;
        }

        if (c == '(') {
            ++stack;
        } else if (c == ')') {
            --stack;
        }

        current_segment.push_back(c);
    }
    if (!current_segment.empty()) {
        segments.push_back(current_segment);
    }
}
```

This code works by keeping track of a stack value which represents how many open/close brackets have been encountered. The parser will continue consuming characters of the string until whitespace is encountered or, alternatively, until a close bracket and then whitespace is encountered when the stack value is zero.

The Component class needs to consider that component values can be given in a string including non-numerical digits and must therefore extract and convert the value appropriately depending on the multiplier. To do this, a function was written to run a regex search across a string for a given pattern. This was used to extract the numerical part and multiplier part from the component value.

```
std::vector<std::string> regex_extract(const std::string& s, const std::string& re) {
    std::regex r(re);
    std::smatch m;
    std::regex_search(s, m, r);

    std::vector<std::string> matches;
    for (auto v: m) {
        matches.push_back(v);
    }
    return matches;
}
```

These parts could then be converted using a map from string to double to convert into the correct value.

```
std::unordered_map<std::string, double> value_map = {
        {"p", 1.0e-12},
        {"n", 1.0e-9},
        {"u", 1.0e-6},
        {"m", 1.0e-3},
        {"", 1},
        {"k", 1.0e3},
        {"Meg", 1.0e6},
        {"G", 1.0e9}
};
```

However, not all components have a value that can be converted in this way as for diodes, BJTs, MOSTFETS and AC voltage sources the value is not given in a numerical form and instead refers either to a model number or to a function. To account for this, the Component class includes a member function which returns a Boolean indicating whether the value can be converted.

```
bool Component::has_converted_value() {
    // Value for diodes, BJTs and MOSFETs cannot be converted as it is a model number.
    if (identifier == 'D' || identifier == 'Q' || identifier == 'M' || is_vs()) {
        return false;
    }
    return true;
}
```

The final item that must be parsed is the simulation description line. This is given in the format shown below.

```
.ac dec <points per decade> <start frequency> <stop frequency>
```

As this line always follows this format, it is relatively simple to parse it into its three components. This was done using a function to split a string on whitespace into a vector of substrings.

```
void split_string_by_space(std::vector<std::string>& out_vector, const std::string& string) {
    std::istringstream iss(string);
    for(std::string s; iss >> s;)
        out_vector.push_back(s);
}
```

For example, this would convert the line ".ac dec 10 10 100k" into a vector containing the following parts: ".ac", "dec", "10", "10", "100k". The 2nd, 3rd and 4th items can then be converted using the same conversion map used previously for component values and stored in three variables as doubles. The parsing process is seen below.

```
void parse_simulation_description(const std::vector<std::string>& netlist, double& points_per_decade, double& start_frequency, double& stop_frequency) {
    std::string simulation_description;
    for (const auto& line: netlist) {
        if (line.substr(0, 3) == ".ac") {
            simulation_description = line;
            break;
        }
    }
    std::vector<std::string> simulation_parts;
    split_string_by_space(simulation_parts, simulation_description);
    points_per_decade = convert_value(simulation_parts[2]);
    start_frequency = convert_value(simulation_parts[3]);
    stop_frequency = convert_value(simulation_parts[4]);
}
```

This concludes the implementation of the netlist parsing. Additional details can be found in the source code.

CONSTRUCTING THE MATRIX

MATRIX INITIALIZATION

Building the matrix was no doubt the most computationally expensive part of the project, being able to build a circuit, and solve it for unknown variables in less than a second. The matrix class in Eigen is defined by 6 parameters, out of which 3 are optional[1], which we opted to leave out. The first parameter defined as "typename scalar" would initialize the type of each coefficient in the conductance matrix, and vector of knowns/unknowns. This allowed coefficients to be of any type defined, such as int, float, double and complex etc. Since conductance is the reciprocal of resistance, this meant we needed a type that allowed accuracy to decimal precision. Different data types such as floats and doubles allowed this but during testing, we opted to go for double as it allows for double the value precision when compared to floating point values, but also sacrifices speed. Later on, we settled on a complex datatype with both real and imaginary parts stored as doubles to allow for AC reactive components and this is written in depth in the "Advanced input management" section.

The next 2 parameters, known as RowsAtCompileTime and ColsAtCompileTime[2] would, as suggested, initialize the number of rows and columns of the conductance matrix. This allows you to initialize the matrix dimensions either at compile time – which makes for faster operations on it – or dynamically at runtime – which makes for slower calculations but is better in this application as we do not know what the dimensions will need to be when the program is compiled. The Eigen typedef[3] can be used as a shorthand for initializing the matrix size and type, where the Eigen class is defined as Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>, can instead be written as MatrixNt, where N is the number of rows or columns, and t is the type. Since the conductance matrix is built upon Kirchhoff's current law, which in turn is derived from Maxwell's equations[4], a characteristic of the matrix is that it must be invertible, which in turn means it must be a square

[1] https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html
[2] https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html
[3]
[4] https://www.cpp.edu/~pbsiegel/supnotes/nts2341.pd

matrix. Therefore the "N" in the Eigen typedef represents both the rows and columns because the matrix must be square and invertible. The Eigen typedef MatrixXd was the popular choice, since it allowed the coefficients to be of type double, whilst keeping the size of the matrix as dynamic, allowing it to be changed when appropriate. Using a fixed sized matrix would be more efficient than a dynamic one in terms of performance[5], however it was decided that we would just use a dynamically sized matrix, since most of our matrix/vector sizes can be defined by the maximum number of nodes and independent voltage sources, which would not be known at compile time. This meant that the 3 optional parameters of the Eigen class were not needed, and in the end made code more readable and easier to manipulate.

The most complicated part of this part of the program, was initializing the conductance matrix, because once this was done it was straightforward to define matrices of one dimension, i.e., vectors, and finding the inverse of the conductance matrix also turned out to be straightforward using specialized Eigen methods[6].

The first obstacle was correctly constructing the size of the conductance matrix so that it appropriately scales to a suitable size defined by the nth node and kth voltage source. To achieve this, the use of a function mat_Populate() was defined, in which it would populate the coefficients of the 3 matrices, but we will discuss the two vectors later and focus on the main conductance matrix for now.

Using the typedefs for the matrix, and similar typedefs[7] for the vectors, a total of 3 matrices were constructed. The first, Cond, was of type double and dynamic size using the typedef MatrixXd. The 2 vectors, also known as matrices with one column, used the typedef VectorXd which were also of type double and dynamic size. As per the project specification for the netlist template, the first string would be the designator of the component, and the next 2 strings represent the connected terminal nodes of the component. Knowing this, a function node_max() was defined that would return the largest node in our parsed std::vector<std::vector <std::string>> netList by looping

[5] https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html
[6] https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html
[7] https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html

through each line within netList and appropriately assigning the largest node back. This was important as the maximum node was required when defining several other matrices that would construct the conductance matrix and vectors.

```
// Finds the largest node in the vector of strings, so the matrix can be initialised with it's size
int get_node_max(Component* component, int node_max) {
    int max_elem = node_max;

    for (auto elem: component->get_nodes()) {
        if (elem > max_elem) {
            max_elem = elem;
        }
    }

    return max_elem;
}
```

The next objective is to identify the number of Independent Voltage Sources, which will be needed to scale the size of the matrix and vectors. For this, a simple iteration of the parsed netList is required to locate every instance of a voltage source, which can be done by checking if the type is equal to “V” and is then counted by another variable, total_Vsources.

```
int total_Vsources = 0; // Total number of Independant Voltage Sources
int Vsource_pos = 0;  // Keeps track of the kth voltage source

bool is_AC = false; // Whether the source is AC/DC

for (auto* component: components) {
    if (component->get_identifier() == 'V') { // If a voltage source is found...
        total_Vsources += 1; // Add to the total number of IVSs
        if (component->is_vs()) {
            is_AC = true;
        }
    }
}
```

The main conductance matrix, known as Cond (cond_Mat in run_simulation) in the function, can be resized to size (n x k), where n is the number of nodes and k is the number of voltage sources. This is another method that comes with the Eigen library, allowing the resizing of dynamic and fixed type matrices

using .resize()[8]. However, the coefficients within the matrix are still uninitialized, so we require the setZero()[9] method to set all coefficients to zero. Not doing this leads to some random value within the matrix due to C behavior (declared values are never automatically set to a default), which in our testing equated to e^+6 which affects results dramatically when solving the matrix for the vector of unknowns.

The vector of knowns, Curr, and the vector of unknowns, Volt, were both initialized and resized within the function mat_Populate, which was used to initialize the coefficients of matrix cond_Mat. We used the Eigen typedef VectorXd to define the 2 vectors of dynamic size and coefficients of type double. Since the overall result was to do matrix multiplication between the conductance matrix and the vector of knowns, this required that the vectors multiplying the conductance matrix were multipliable, meaning their rows had to match the number of columns in Cond. This was defined as the number of nodes + the number of independent voltage sources and using eigen method .resize() within mat_Populate(), we were able to initialize the two vectors correctly. Setting the coefficients of the two vectors to zero using setZero() was important, because it helped to remove random uninitialized terms, but also helped with future calculations. For example, if a node was not connected to any current source, the current into that node in the current vector will be zero. So this saves time as we don't need to implement further code to change some unknown value to zero in the vector.

```
Cond.resize(node_max + total_Vsources, node_max + total_Vsources); // Conductance matrix resized to the nth node plus the kth IVS
Volt.resize(node_max + total_Vsources); // Vector of unknowns
Curr.resize(node_max + total_Vsources); // Vector of knowns

Cond.setZero(), Volt.setZero(), Curr.setZero(); // Initialise coefficients to 0, otherwise they have a random value
```

MATRIX CALCULATIONS AND INITIALIZING COEFFICIENTS

Now that we had initialized the sizes of Matrix Cond, and Vectors Volt and Curr, we began to add support for different components that were possible to place in the netlist file.

[8] https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html
[9] https://eigen.tuxfamily.org/dox/group__TutorialAdvancedInitialization.html

To populate the matrix coefficients, the first parameter that would be required is the vector of vectors netList, as this contained all the information we needed, such as the type of component, their node connections and their values. Using the vector of vectors, we can define new variables ith, jth, val and type. Ith and jth are the corresponding [ith, jth] position in the conductance matrix, defined by the first and second node for each Component object in the circuit. If a component is connected between nodes 1 and 2, its position in cond_Mat would be [1, 2]. An important aspect of the Eigen library to note is that contrary to linear algebra in mathematics, the matrix starts from the [0, 0] position since indexes in programming are defined by their location in memory, which often starts from zero. In fact, the [1, 2] position will need to be adjusted to [0, 1], which is why ith and jth are defined as the node found in each vector minus 1. A suitable data type would be int since a node cannot be anything but an integer. Val is defined as the pre-processed value for each Component object (Component::converted_value()), since the value is prescribed after both the designator and the connected nodes during construction of netlist as well as on the project specification. The instinctive decision would be to assign val as type double, since the spectrum of values for reactive components such as capacitors and inductors can reach well below the micro (10^-6) scale, and double will allow for more precision than float, so we went with that. There is also the designator, gotten from Component::get_identifier(), and overridden to be “VS” if the component is an AC voltage source. We can leave this as a string type since its only use is to identify the type of component and cannot conveniently be converted into anything else that is useful.

```
for (auto* component: components) {
    ith = component->get_nodes()[0] - 1;
    jth = component->get_nodes()[1] - 1;
```

Now we started to initialize the contents of matrix Cond. This involved a form of Modified Nodal Analysis[10]. The first step was to break down the conductance matrix into 4 smaller matrices G, B, C, D. Matrix G would contain the passive and reactive components of our circuit and would be of size (n x n) where n was the number of nodes. A for loop was used to pass each vector in the vector of vectors into the new function, G_matrix() which constructs the G

[10] https://www.swarthmore.edu/NatSci/echeeve1/Ref/mna/MNA3.html

matrix. To place these components into the matrix, we need the ith, jth nodes, the value of that component, and its type which are defined by their respective positions of each vector in netList.

```
Eigen::MatrixXcd G(node_max, node_max); // Used in G_matrix()
Eigen::MatrixXcd B(node_max, total_Vsources); // Used in B_matrix()
Eigen::MatrixXcd C(total_Vsources, node_max); // Transpose of B
Eigen::MatrixXcd D(total_Vsources, total_Vsources); // Zero matrix

G.setZero(), B.setZero(), C.setZero(), D.setZero();

for (auto* component: components) {
    ith = component->get_nodes()[0] - 1; // ith position ( "-1" is there because Eigen::Matrix matrices start from [0,0]
    jth = component->get_nodes()[1] - 1; // jth position --

    if (component->get_identifier() == 'V') { // If an IVS is found...
        if (component->is_vs()) {
            B_matrix(ith, jth, Vsource_pos, B); // --
            Curr_matrix(node_max, ith, jth, component->get_amplitude(), "VS", Vsource_pos, Curr); // --
            Vsource_pos += 1; // --
        } else {
            B_matrix(ith, jth, Vsource_pos, B); // Constructs B matrix, which is used to make the C matrix
            Curr_matrix(node_max, ith, jth, component->converted_value(), "V", Vsource_pos, Curr); // IVS added to vector of
            Vsource_pos += 1; // Only used for more than one source
        }
    } else if (component->get_identifier() == 'I') { // If a Current Source is found...
        Curr_matrix(node_max, ith, jth, component->converted_value(), "I", Vsource_pos, Curr); // Adds the corresponding sou
    } else {
        G_matrix(ith, jth, component->converted_value(), std::string(1, component->get_identifier()), is_AC, omega, G); // C
    }
}
```

The processing of G_matrix() was straightforward, and we began with adding support for resistors since they were the easiest passive component to implement. A simple nested conditional was used to check if the type of component was “R”, verifying that it is a resistor. This conditional was defined by the Component’s identifier, gotten through the member function Component::get_identifier(). If the component is indeed a resistor, another conditional within would check to see if the resistor is connected to the reference (ground) node or not, since this connection would affect the position of the resistor’s conductance in the matrix. If a component was connected to ground, then its conductance would be placed along the diagonal of the matrix using the node that it is connected to. So, a resistor between ground and node 2, would be stored in the [2, 2] position ([1, 1] in Eigen) with its conductance equal to 1/val. This is because the reference node is not included in the conductance matrix, so it is not included in the sub matrix G. If connected between 2 non-reference nodes, then the negative conductance of the resistor would be taken and stored in matrix position corresponding to the ith and jth

node. The reason why we take the negative conductance involves simple nodal analysis equations. If a resistor is connected between two nodes, 1 and 2. The voltage drop across the resistor would be (V1-V2)/R and rewriting -V2/R as (-1/R)*V2 shows that we do indeed have a negative conductance here. Since G_matrix() was called in a for loop within mat_Populate(), it would be updated with every passive component in the vector of vectors, so the G matrix would have been initialized and its coefficients populated by the end of the for loop.

```
void G_matrix(int ith, int jth, double val, const std::string& type, bool is_AC, double omega, Eigen::MatrixXcd &G) {
    std::complex<double> complex_val; // Used for capacitors/inductors in an AC circuit

    if (type == "R") { // If the type is a Resistor...
        if (ith == -1) { // If connected to ground...
            G(jth, jth) += (1 / val); // Store the reciprocal (Conductance) on the main diagonal
        }
        else if (jth == -1) { // --
            G(ith, ith) += (1 / val); // --
        }
        else { // If not connected to ground...
            G(ith, jth) += (-1 / val); // The corresponding negative conductance of the value is stored in the [ith, jth] position
            G(jth, ith) += (-1 / val); // Because of Kirchoff's Current Law, the matrix is symmetric
        }
    }
```

We then started on capacitors and inductors. In DC, capacitors charge up to the voltage they are connected to, and as a result hinder the flow of current. Therefore, the conductance of a capacitor tends to infinity like a voltage source, and so it can be treated like one and ultimately ignored from the G matrix, since there are no voltage sources in G. With inductors, since there is no voltage drop across them, they act as a short circuit and so we used the normal impedance of an inductor(similar to a resistor) This does lead to some inaccuracy problems when integrating AC circuits, as the larger the inductance, a greater margin of error was visible in our results which we resolved later on.

```
else if (type == "L") { // If the type is Inductor...
    if (is_AC) { // --
        complex_val.imag(1/(val*omega)); // Reactive component is
        if (ith == -1) {
            G(jth, jth) += -complex_val; // Admittance of an induct
        }
        else if (jth == -1) {
            G(ith, ith) += -complex_val; // Minus is used because
        }
        else {
            G(ith, jth) += complex_val;
            G(jth, ith) += complex_val;
        }
    }
```

The next sub matrix of Cond is called B, and its size is defined as (n x k), where n is the number of nodes and k is the number of independent voltage sources.

```
B(node_max, total_Vsources), // Used in B_matrix()
```

The B_matrix() function constructs this matrix, only populates it with 1, 0 or -1, and is found within the same for loop that G_matrix() is in, with the only exception that it falls under a conditional to check if the type == “V”. The reasoning behind this was because matrix B only shows the position of independent voltage source(s) in the circuit and the sources’ values would be stored in the vector of knowns. An independent voltage source is allocated a position based on whether it is grounded or not, and allocated a value based on the direction of positive and negative terminals. If a source was grounded then it will only appear once, since the reference node is not defined in the matrix, and it will be positioned on the ith/jth row, depending on which node is the reference node i.e. = -1(we defined earlier why minus 1 is used when referring to matrix positioning). If the ith node is ground, this means that the jth node receives a negative voltage as defined by the netlist specification where the first node mentioned is positive and the second node is negative, and so the position [jth, Vsource_pos] is equated to -1. Here, Vsource_pos is set as the kth voltage source. If one voltage source is passed in, its position is 0. But if a second and/or third source is in the circuit, Vsource_pos keeps track of the kth source, and the kth column of matrix B is updated. If an independent

voltage source is connected between 2 non-reference nodes, this equates to the KCL equation Vi - Vj, so the ith node is allocated a 1 and the jth node a -1, and the appropriate column is chosen based on how many voltage sources there are. If this was the second voltage source out of 2, then these values would be stored in the second column etc.

```
void B_matrix(int ith, int jth, int Vsource_pos, Eigen::MatrixXcd &B) {

    if (ith == -1) { // If the ith node is 0 (meaning positive terminal
        B(jth, Vsource_pos) = -1; //
    }
    else if (jth == -1) {
        B(ith, Vsource_pos) = 1;
    }
    else if (ith >= 0 && jth >= 0) {
        B(ith, Vsource_pos) = 1;
        B(jth, Vsource_pos) = -1;
    }
}
```

The C matrix is ultimately the transpose of the B matrix, since the connection between nodes i and j are equal to the connection between nodes j and i. The transpose is found using the Eigen method .transpose(), flipping rows and columns and storing into C.

```
C = B.transpose(); // As mentioned
```

The D matrix is of size (k x k), where k is the number of independent voltage sources, and it is initialized using the setZero() method. The D matrix serves no purpose other than to complete the full Cond matrix by making it square and invertible.

```
Eigen::MatrixXcd D(total_Vsources, total_Vsources); // Zero matrix
```

Using the Eigen header library, one can initialize a matrix using “<<”. This can be used to set a matrix with constant integers, float types or double types, or it can be used to initialize a matrix using other matrices. As mentioned before, Cond is made up of 4 smaller matrices G, B, C & D, so writing “Cond << G, B, C, D;” yields the same result, and we end up with a square, invertible matrix of

size (n + k) x (n + k). This completes the construction of the conductance matrix, and later on we will improve it to work with AC signals but for now, it works well with DC passive components. Matrix Cond was passed into the function mat_Populate by reference, so there is no need to return our matrix type to the run_simulation function, since it will update the conductance matrix globally, and hence further calculations on Cond take place in the run_simulation().

```
Cond << G, B, C, D; //
```

We have defined how the conductances are appropriately placed within a matrix based on their node connections, and that if they have one connected to ground, we just add them to the main diagonal. However, one important factor to note is that the main diagonal doesn't just contain conductances that are connected to ground, but all the conductances that are connected to that node. If 3 resistors were connected between nodes 1 to 2, 1 to 3 and 1 to 4, we have a total of 3 resistors connected to node 1 but none to ground. This means we will have negative conductance at positions [1, 2], [1, 3] and [1, 4] but also the total conductance at position [1, 1] will be equal to the sum of these conductances. To do this we defined a new function cond_Diagonal, that would simply look at the conductances in one row of the matrix, sum them up and place the total into the main diagonal. We take the absolute value of these conductances since they are negative outside of the main diagonal.

```
// Sums all the conductances connected to a node, and places it along the diagonal
void cond_Diagonal(Eigen::MatrixXcd& Cond, int k, int node_max) {
    double total_real = 0; // Total of the passive components
    double total_imag = 0; // Total of the reactive components
    for (int m = 0; m < node_max; m++) { // Iterate through all nodes
        if (Cond(k, m).real() != 0 || Cond(k, m).imag() != 0) { // If the [k, m] real or imaginary pos
            total_real += abs(Cond(k, m).real()); // The conductance at A[k, m] is added to the total
            total_imag += Cond(k, m).imag(); // The admittance --
        }
    }
    Cond(k, k).real(total_real); // Total real values are placed along the diagonal, e.g. all conducta
    Cond(k, k).imag(total_imag); // Total imaginary values --
}
```

Now that the conductance matrix has been initialized and its coefficients given a value, we can move onto the two vectors. The first vector, Volt, does not need

to be filled with values yet, since it behaves as our vector of unknowns with unknown voltages and currents. The second vector, Curr, is our vector of unknowns and is straightforward to update. We already constructed the size of Curr as ((n+k) x 1), the nth node and kth voltage source, meaning the number of values in this vector will equal to the sum of nodes and sources.

Kirchhoff's Current Law states that the sum of currents entering/leaving a node is 0, so if a node is not connected to any current sources, we can assume that the current is 0 at that node. Therefore, a KCL equation at node 1 would mean the first entry in the current vector is 0. However, when a node is connected to a current source, we must look at the way it is defined in the netList. Following the project guidelines, the first node mentioned in the "in" terminal and the second node is the "out" terminal. This means when looking at the currents going out of a node, as we have learnt to do so this year, if the "in" terminal is facing the current node, this means the current is flowing out of the node and fulfills the nodal equation = 0. However, this does not mean a zero is stored in the current vector, but instead the negative current will be stored in the vector at the nth position, since this is the same as rearranging a KCL equation for unknowns only. A positive current on the left of an equation, means a negative current on the right, so we place the opposite value of the current into the vector of knowns. Likewise, if a current source is floating, then the corresponding nodes connected to each terminal will be added to nth row of the vector, but with opposite signs since one will be entering the node, and one will be leaving as defined by their nodal equations. This processing was defined within a new function curr_Matrix().

MATRIX DECOMPOSITION

Now that both the conductance matrix coefficients and the current vector coefficients have been initialized, it was time to solve for the voltage vector. Our first thoughts were to use the Eigen method .inverse()[11] because we needed the inverse of the matrix Cond to be multiplied with the current vector on the right. However, as well as this, the eigen header offers a more versatile approach with methods that can be used to directly solve for some unknown

[11] https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html#title3

vector, and they are each rated by their accuracy, performance, and limitations[12].

The first method we considered using was ColPIvHouseholderQR(), since it boasted relatively fast decomposition speeds whilst also maintaining high accuracy. The tradeoff was that for larger matrices (circuits with more than 16 nodes), the speed of decomposition dramatically falls and so presents a challenge to efficiency for those larger circuits. Though we did not expect to evaluate circuits with so many nodes, it was clear that the algorithm would not be best for such scenarios.

Another method we considered was LLT(), as it was presented to have one of the fastest decomposition times within the entire list of methods. The limitation to this method was that it required values within the matrix to be positive definite, and we have shown that our matrix contains both positive and negative conductance values, so the LLT() method will not work. This also eliminated the use of LDLT(), which is just as fast as LLT() decomposition but also boasts a similar drawback whereby the coefficients of the matrix must be positive or negative semidefinite, meaning they can only either be positive or negative, whilst our matrix is both. It was worthwhile discussing the use of these methods, because if we had components that were only connected to ground and not between nodes, the matrix would be positive only, but this can only be used for very basic small circuits, and the extra logic that would be required to perform this decomposition specifically to small circuits was not very appealing.

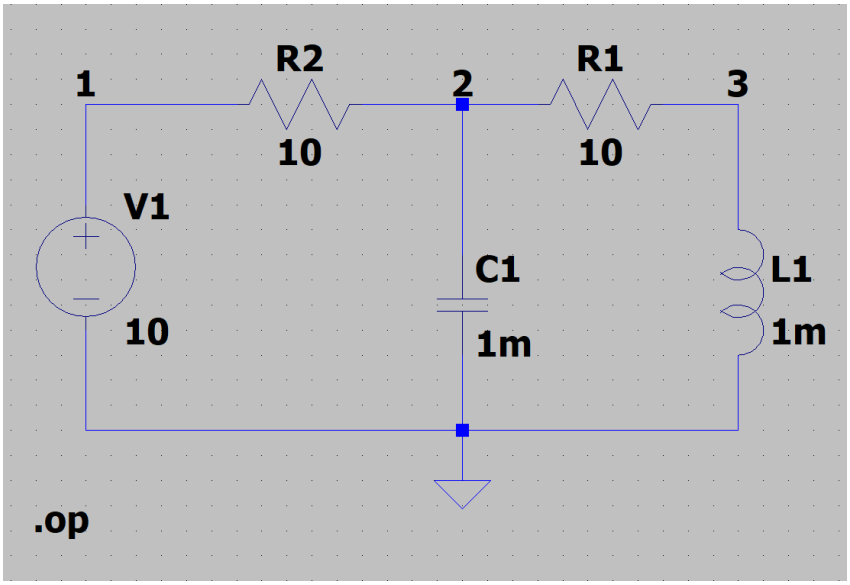
This left the final method we analyzed partialPivLU(), which offered very fast decomposition times of both small and large matrices, as well as a fairly accurate decomposition but not as good as other methods. However, since we were limited to using type double, the difference of accuracy was hardly noticeable and barely affected our results. partialPivLU() does have a drawback in that its matrices must be invertible. We have already discussed why our matrix must be invertible because of its relationship with Kirchhoff's Current Law derived from Maxwell's equations[13], so our matrices will always fulfill the condition of invertibility. When using decomposition methods, they

[12] https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html#title3
[13] https://www.cpp.edu/~pbsiegel/supnotes/nts2341.pdf

must be defined in the format, MatrixNt.partialPivLU.solve(VectorN). To move from A*x=b to x=A^-1*b, the first matrix defined is the matrix that will be made into its inverse, and the second matrix that will be multiplied with the inverse matrix is placed within the parameters of the solve() member function. This would give us all the node voltages within the voltage vector, and hence the solution has been found and can be used for AC analysis.

DC CIRCUIT EXAMPLE

To test that our conductance matrix logic worked well with DC circuits so far, we built a simple circuit on LTSpice using components that we had integrated; resistors, capacitors, inductors and voltage sources.



$$I\_V_1 + \frac{V_1 - V_2}{R_2} = 0$$

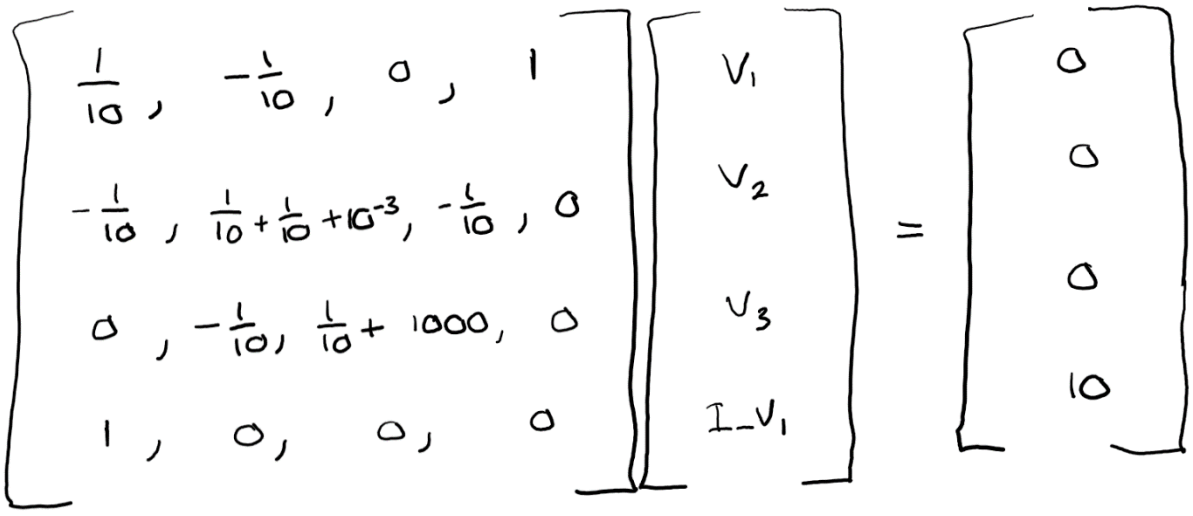
$$\frac{V_2 - V_1}{R_2} + V_2 \cdot C + \frac{V_2 - V_3}{R_1} = 0$$

$$\frac{V_3 - V_2}{R_1} + \frac{V_3}{L} = 0$$

Writing out the nodal analysis equations for this circuit, we can easily see how to decompose it to into matrix form. The voltage between node 1 and node 2 is (V1-V2)/R2 which means that R2 is common to both voltages. We can rearrange the equation to give V1(1/R2) + V2(-1/R2). This gives us 2 conductances, the positive one is connected to node 1 since we are looking at currents going out of the node, and the negative to node 2. We also have V1 = Vsource so we place a 1 in what we called the B matrix at the node 1 position. The C matrix is the transpose of B and can be seen below the conductances. And then the D matrix is the final zero matrix of size (1x1), since there's only one voltage source.

$$\begin{bmatrix} \frac{1}{R_2} & -\frac{1}{R_2} & 0 & 1 \\ -\frac{1}{R_2} & \frac{1}{R_2} + \frac{1}{R_1} + C & -\frac{1}{R_1} & 0 \\ 0 & -\frac{1}{R_1} & \frac{1}{R_1} + \frac{1}{L} & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I\_V_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ V_1 \end{bmatrix}$$

So we can see the full matrix is of size (n+k)x(n+k) or (4x4) as we defined before, we need the total number of nodes and voltage sources to build the matrix.



Inputting the values of each component into the matrix.

```
Conductance Matrix:
   (0.1,0)   (-0.1,0)      (0,0)      (1,0)
  (-0.1,0)    (0.2,0)   (-0.1,0)      (0,0)
    (0,0)   (-0.1,0) (1000.1,0)      (0,0)
    (1,0)     (0,0)      (0,0)      (0,0)
```

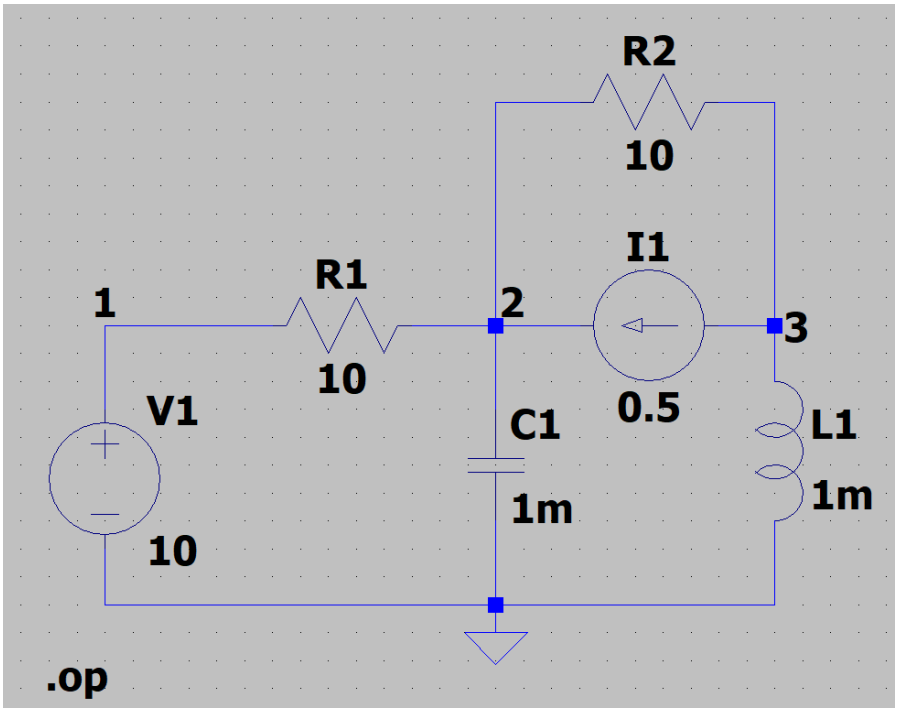
We can see here that the output of the conduction matrix from our code is the exact same as the one we managed to calculate by hand.

```
        --- Operating Point ---

V(2):           5.00025         voltage
V(3):           0.000499975     voltage
V(1):           10              voltage
I(C1):          5.00025e-015    device_current
I(L1):          0.499975        device_current
I(R2):          -0.499975       device_current
I(R1):          -0.499975       device_current
I(V1):          -0.499975       device_current
```

```
Voltage Solutions:
          (10,0)
     (5.00025,0)
(0.000499975,0)
   (-0.499975,0)
```

Here we have the outputted DC operating point of both LTSpice's circuit, and our own. We can see that the node voltages at all nodes are equal in both, so it is safe to conclude that our circuit simulator successfully calculates node voltages at DC!



We also tried circuits with a current source, to see whether the corresponding current vector would be updated with its value. In this case, we have a current source connected between 2 non-reference nodes, so the current is going away from node 3, and entering node 2. This would mean that when writing the nodal equations at this point, we would have a -0.5 at node 2, and a 0.5 at node 3. However, we only want the known values to be on the left, which is our conductance matrix, so we subtract the current and place it into its appropriate node position.

```
        --- Operating Point ---

V(1):           10              voltage
V(2):           7.50012         voltage
V(3):           0.000249987     voltage
I(C1):          7.50013e-015    device_current
I(L1):          0.249987        device_current
I(I1):          0.5             device_current
I(R2):          -0.749987       device_current
I(R1):          -0.249987       device_current
I(V1):          -0.249987       device_current
```

```
Current Vector:
   (0,0)
 (0.5,0)
(-0.5,0)
  (10,0)
Voltage Solutions:
         (10,0)
    (7.50012,0)
(0.000249988,0)
  (-0.249988,0)
```

Here, we can see the current vector successfully updated with the correct current values at each node, and if this were a current source connected to ground, we would only see the current at one node instead. We also see that the node voltages are the same with minute rounding errors due to our use of double type variables. Overall we can say our circuit works good for non-linear DC components.

ADVANCED INPUT MANAGEMENT

Now that our circuit has been defined for DC circuits, it was time to implement AC logic. The first thing to look at when it comes to AC circuits is the frequency, omega. This means that we must bring complex numbers into the matrix now, since the admittance of capacitors and inductors in AC are jωC and 1/jωL. Eigen allows the use of complex type matrices[14] and can be defined using the Eigen typedef Eigen::MatrixXcd. Here, we are defining a matric of dynamic size,

[14] https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html

and its coefficients are of type complex double, which is what “c” and “d” in the typedef represent. Because our matrix will now involve complex numbers, this means all 3 matrices Cond, Volt, and Curr had to be initialized as complex double types. As we have discussed how Cond is made up of 4 smaller matrices G, B, C & D, these matrices also needed their typedef initializations to be updated. Although it is only sub matrix G that would hold the imaginary values of reactive components, whereas the other 3 matrices making up Cond do not, it is vital that all matrices that make up Cond are of the same type otherwise there will be initialization errors when constructing Cond.

When constructing matrices of type complex double, they now can store two values, one in the real component and one in the imaginary component. First, we need to distinguish to the matrix when to update its imaginary part and when to update its real part. To do this, we used a Boolean variable called is_AC. The function of this variable to always assume that a circuit is in DC until a component shows otherwise. In the for loop defined for counting the number of independent voltage sources into total_Vsources, by parsing the netlist and counting every mention of a line that starts with “V”, we can add the Boolean variable here and add a conditional to check whether a source is DC or AC. The distinction between an AC and DC source in SPICE, is that DC simply has one real value, whereas an AC source has value equal to its magnitude and phase. This is further emphasized with the value written as AC(mag, arg). So we can do a simple check to see if the value of a source starts with “A”, it is an AC source, and if it does not start with AC but a number, it is DC. When an AC source is found, bool is_AC is set to true because we have identified that the circuit is indeed AC.

This can then be passed on into the construction of the sub matrix G, along with the frequency step as these two variables are required when working with AC circuits. The conditional was simple, if the variable is_AC was true, and we the current component is of reactive type, i.e. capacitor or inductor, then we require the use of omega and the imaginary value in the [ith, jth] position. For capacitors, the value had to be changed from being capacitance alone, to being 1/jωC. This is the impedance of a capacitor, but the conductance matrix is filled with conductance, so we need the reciprocal of its value, jωC, its admittance. We do not need to write jωC directly into the [ith, jth] position of the matrix, but only need to take ωC and store that into the imaginary

component at the correct position. For example, a capacitor connected between nodes 1 and 2(0 and 1 in eigen) and when the circuit is AC, will be entered into the matrix using the code, G(0, 1).imag() = ωC. So when accessing this position again, we can read it as 0 + jωC, since we only changed the value in the imaginary and left the real part as 0.

The same logic can be applied for inductors, as they also need to have their values changed when looking at AC circuits. The impedance of an inductor is jωL, so the admittance (reciprocal) is equal to 1/jωL. This presents a slight issue with how complex variables work in C++, because we want to store 1/ωL into the imaginary component, if we just stored it normally, it would be equal to j/ωL which is not equal to 1/jωL. However, using some complex arithmetic knowledge from our mathematics course this year, we know that the reciprocal of an imaginary number, 1/j, is equal to itself multiplied by -1, -j. And we can just store (-1)/ ωL into the imaginary part.

The value of an AC voltage source is its amplitude, so therefore the only difference between a voltage source in DC and AC is the way the value is read from the netlist. However, the value that is stored into the current vector, is still done in the same way so it is not difficult to implement that with normal voltage sources in DC.

There were also some slight adaptations required to the cond_Diagonal function, namely because we have both real and imaginary parts, so the main diagonal must contain the sum of all real/imag conductances connected to a specific node. Using the .real() and .imag() methods made this relatively easy as it allowed us to sum up each part separately.

FREQUENCY INPUT

Using the equation given in the specification, we used the information in the last line of the netlist, which contained the points per decade, start frequency and stop frequency.

$$: f_n = 10^{n/n_p} f_s,$$

```
double get_omega(int n, double points_per_decade, double start_frequency) {
    return std::pow(10, (n/points_per_decade)) * start_frequency;
}
```

From this, we could iterate the value of omega by incrementing the current step n, and therefore retrieve a node voltage at our nominated node. If the frequency ranged from 10 to 100k, this would mean there are a total of 4 decades, and 10 points per decade means that we have 40 values of omega to pass in, excluding the start frequency. This also helped us to determine whether we had the right number of output values in our CSV file, as simple counting enabled us to check.

PROGRAM OUTPUT

The brief required the output to follow the CSV format. Program output follows the column template: “frequency step, gain, phase”. To do this, a CSV header file was created which contains useful functions for writing different formats to the output. This code was engineered for reusability and ease of use. You can see the function declarations in the header file “csv.h” below.

```
#ifndef ICL_CIRCUIT_SIMULATOR_CSV_H
#define ICL_CIRCUIT_SIMULATOR_CSV_H

#include <vector>
#include <string>

namespace csv {
    std::string join_items_into_row(const std::vector<std::string>& items, char delimiter);
    void write_row(const std::vector<std::string>& items, const std::string& filename);
    void write_rows(const std::vector<std::string>& rows, const std::string& filename);
}


#endif //ICL_CIRCUIT_SIMULATOR_CSV_H
```

To aid with scope and readability, all functions deal with csv operations and output file interactions are grouped under the “csv” namespace.

The function join_items_into_row takes a vector of strings (items) and concatenates them into a single string which will become a row in the output

csv, as well as a delimiter (usually a comma) which will act as the item separator once written to the file. The code can be seen below.

```
std::string csv::join_items_into_row(const std::vector<std::string>& items, char delimiter) {
    // Join a vector of items with the given delimiter.
    // The row will have a newline appended to it and no trailing delimiter.
    std::string row;

    for (int i=0; i < items.size(); i++) {
        row += items[i];
        if (i != (items.size() - 1)) {
            row.push_back(delimiter);
        }
    }
    row.push_back('\n');

    return row;
}
```

The function write_row takes a vector of strings (items) and internally concatenates them with commas and appends a newline. The row is then written to the output file with the given filename. The code can be seen below.

```
void csv::write_row(const std::vector<std::string>& items, const std::string& filename) {
    // Append a single row to the file with given name.
    // Items will be joined with commas and a newline appended before the row is written.
    std::string row = csv::join_items_into_row(items, ',');

    std::ofstream outfile;
    outfile.open(filename, std::ios_base::app);
    outfile << row;
    outfile.close();
}
```

The function write_rows takes a vector of fully formed CSV rows and a filename. The rows must be terminated with a newline character. It then loops over the given rows and appends them to the output file with the given filename in order. The code can be seen below.

```
void csv::write_rows(const std::vector<std::string>& rows, const std::string& filename) {
    // Append multiple rows to the file with given name.
    // Each row in the rows vector should end with a linebreak.
    std::ofstream outfile;
    outfile.open(filename, std::ios_base::app);

    for (const std::string& row: rows) {
        outfile << row;
    }
    outfile.close();
}
```

PROBLEMS ENCOUNTERED AND HOW WE RESOLVED THEM

EIGEN LIBRARY HEADER

One problem we faced early on, was determining how to install and use a new header, especially one of Eigen's magnitude. Naturally, we turned to the internet to see how to install the header, and the first issue we ran across was installing the wrong version. Though Eigen v3.4 was the latest that came out this year, it was also riddled with bugs in an attempt to expand and become efficient and it was not very kind to us at the start. We then went to install Eigen v3.3.9 which had released in early 2020 and had the added benefit of being much more stable overall, and provided us with more algorithms and methods than we could need. It was a challenge using the Windows PowerShell to install the library specifically for Windows x64 systems but in the end, we managed to do so, whilst learning about the PowerShell which none of use knew how to use before. It was a good learning experience.

READING AND PARSING

Initially, the plan was to create separate classes for each type of component, however due to the limitations due to C++ being statically typed this meant that there was no plausible way to use a single data structure to store every type of object available. Furthermore, the inability to override class attributes in subclasses meant that the subclasses would not be instantiated correctly. Due to these fairly significant setbacks, the parsing code would need to be rewritten two times before it reached its final (current) state.

A further problem encountered was a problem with the parser itself. Due to the format of some lines for specific components in the netlist, it meant that the parsing method at the time (splitting the line on all whitespace) could not possibly work for our purpose. For example, when attempting to parse the line for an AC voltage source ("V1 N001 N004 AC(1 0)"), it would result in parts as follows: "V1", "N001", "N004", "AC(1", "0)". The expected result from parsing is as follows: "V1", "N001", "N004", "AC(1 0)". This meant that the value for any AC source in the circuit was not being extracted correctly. To solve this,

the new stack-based parser was written which can be seen above in the parsing implementation section.

MATRIX INITIALIZATIONS

One problem faced when initializing the coefficients of a complex matrix, was being able to access both the real and imaginary parts of a matrix's position and changing them to our desired value. Initially we went with directly accessing the position with code "G(1, 1).imag(val)". However, this would not update the value in the imaginary part of our matrix even though it was valid logic. To work around this, we defined a new variable of type complex<double> and called it complex_val. To prevent uninitialized values within the complex variable, we set it to (0, 0) and if we wanted to access the imaginary part, we just write complex_val.imag() = val*ω. It turned out setting the imaginary part to a value rather than entering its parameters with a value did update the matrix correctly and so the problem was solved.

LTSPICE SIMULATIONS

When testing the frequency response of Low Pass and High Pass filters, it was odd to see how the AC analysis in LTSpice would sometimes go above the 0dB magnitude. This was odd because the output voltage could not have been higher than the input voltage in these type of filters, they usually level out at 0dB. We could only assume there might have been a slight issue with the spice simulation, because our code worked as it should and levelled off towards 0dB.

PROJECT PLANNING AND MANAGEMENT

Our overall objective was to do ample research at the beginning and then begin with the actual technical parts of the project having been confirmed by that point. We did have a slight distraction in between – a programming assignment – which would require us to divert our attention from the project for the final ten days of May. Therefore, we held a meeting on the 15th where we collectively made decisions such as the programming language we would be using, which library we would use for the matrix implementation, and which parts of the project each of us would be doing.

After these crucial project decisions were made, we did further research into how to best write our program, by watching videos on YouTube as well as consulting articles posted online by other engineering and computing students to gain some insight. We regularly updated each other on our progress through a WhatsApp group chat and held calls with teaching assistants to discuss our implementation and see how we could improve our design further.

After submitting our programming assignments on May 28th, we resumed the work in our project full throttle and spent two weeks on our code and writing the non-technical aspects of the report and creating the video template. By June 10th, we had most of our code working barring smaller parts which needed finalizing.

TEAM ROLES

After holding a Teams meeting to discuss our respective roles in the project, we felt that our team dynamic would work best with Ziyad acting as the project management head and keeping tabs on issues such as time management or any extraneous details needed in the final simulator based on the requirements. He maintained a sense of urgency to ensure the group did not lack in the communication aspect and produced results at the right pace. Oliver's vast experience with C++ and coding in general meant he was an invaluable asset to the group as he provided the knowledge and expertise to carry out tasks such as parsing the netlist input file and brainstorming and then implementing the algorithm. We consulted him anywhere we got stuck, and he would immediately provide a solution as well as possible feedback and

explanations. Ismail was also indispensable to the group as he worked tirelessly on the implementation of the matrix using the Eigen library, one which he had never used before, and he continuously provided updates on his progress which made it extremely easy to gauge our progress in the project.

We feel our team dynamic worked well because we chose team roles which complemented our individual skills, and which also overlapped with each other's skills in case we needed help with our respective tasks.

RECOMMENDATIONS FOR FUTURE WORK

For someone on this project or a similar one in the future, and designing a circuit simulator, one thing to keep in mind is the importance of code efficiency. The processing time, albeit not something which poses as much of an issue to the project designer, would be a determining factor for future users of the simulator as even small differences would signify a less efficient code which probably runs unnecessary loops which prolong the metric. A possible improvement on this circuit simulator would be a corresponding user interface which would allow for the addition of a visual component to the simulator. This would greatly increase the ease of understanding concepts if the simulator was used by engineering students in their learning.

Another suggestion would be to use a different language, for example Python, which is easier to read and understand for a user. If they encounter a problem, they will be able to see where it lies in the code easily. Everyone has different preferences for the programming languages they like to use, so it would be interesting to see implementations in different ways.

Future work could be done in adding the possible compatibility with diodes and transistors to this circuit simulation program. We prioritized the more commonly used and essential circuit components and thus did not implement functionality for these two extraneous components. Further work could be done in attempting to make all-inclusive test cases for this simulator, so that all the functionality with any combination of circuit components could be tested. We did not do this as it would require a large amount of manual work to carry out this process.

CONCLUSION AND REFLECTIONS

We have designed a fully functioning circuit simulator which uses matrix multiplication and Gaussian elimination to calculate the unknown values in a circuit. Having spent the last month first brainstorming and then implementing our design using C++, we feel we have learnt several new things; both new tricks in programming as well as new ways to think about problems and how to approach them.

We did not manage to get all components working, such as non-linear components (Diodes and Transistors) and dependant voltage sources. Perhaps it was because of time constraints but also the urgency to build a circuit simulator to a working and efficient standard, rather than going for quantity over quality. We also wanted to use what knowledge we had about frequency responses of Low/High Pass Filters, and see if we could mimic a system to perform in the same way we had learnt they do.

Working on a group project of this magnitude, albeit online, was always going to be a challenging and unfamiliar scenario to be in. However, we have made that work very well and, in the process, we have also developed good relations with each other, developing a great team ethic. Our team dynamics improved by the day and at this point we can say that we definitely work well as a team.

APPENDIX

SOFTWARE DESIGN SPECIFICATION

1.1 Introduction

This document describes the implementation and methodology of the design of a circuit simulator program, which produces results identical to existing simulators such as SPICE.

1.2 Intended Audience

The audience this is intended for is either aspiring engineers who are currently studying their undergraduate or master's degrees, or even people working in the engineering field.

1.3 Intended Use

The intended use for this circuit simulator is to perform real-time calculations of the voltages and currents within a circuit.

1.4 Project Scope

We anticipate that this circuit simulator is efficient enough to eventually be utilized in a work environment and by professional engineers in their testing of new products.

2.1 Overall Description

Our software is similar to many existing circuit simulation software - albeit without visual circuit aesthetics - and it provides efficient, straightforward solutions to any problems it is tasked with solving.

2.2 User Needs

We hope that engineering students as well as people working within the field of engineering can make use of this simulator and that it increases their testing efficiency.

2.3 User Assumptions

We assume that the user will know, or be able to calculate, the values of the conductance of each component in the circuit. We also assume that the currents or voltages of the circuit are known so that they can be arithmetically used with the conductance matrix to calculate the unknown currents or voltages.

2.4 Operating Environment

Operating System: Windows

Programming Language: C++

External C++ Library used: Eigen v3.3.9

3.1 Functional Requirements

- The program is expected to be able to read and process a netlist from a text file, as described above in the Design Criteria section.
- The user of the program should have some experience working with circuit simulators as well as knowledge about circuits.
- The output must be written to a comma-separated-values file with rows representing steps in time and columns representing all parts of the circuit including components and nodes.
- A MATLAB script will be used to plot the results onto a graph.

4.1 Non-functional requirements

- The processing and running time of the program should be quick, and it should not suffer any unexpected delays due to user input.
- Any inaccuracies in calculations should be within 10% of the actual value.
- The program should be as reliable as any existing circuit simulators, such as SPICE.
- The simulation software should be able to handle very large (>50) numbers of nodes.

BIBILIOGRAPHY:

Eigen software used: https://eigen.tuxfamily.org/index.php?title=Main_Page

Electronic circuit simulation - Wikipedia

Circuit Simulator: How It Helps You Understand Any Circuit | Electronics (alliedcomponents.com)

Circuit simulation (onmyphd.com)