1st Year Project Report

ELEC40006 - Electronics Design Project 1 2020-2021

Circuit Simulator

Dr Edward A Stott
Mrs Esther Perea

Amy Yau - CID 01868783
Kelvin Mang - CID 01852853
Charmaine Louie - CID 01711892

Word Count: 10998

Table of Contents

Introduction

As electronics evolve, so do the complexities of their circuits. With circuits containing a large number of components, inconsistencies appear when doing calculations by hand. Therefore, simulators have been a staple when building electronics. Not only do they perform the calculations in a time-efficient manner, but they are also more accurate than handwritten calculations. Furthermore, they reduce the waste of resources as the need for a prototype is dismissed. For this project, we have been asked to produce a circuit simulator. When a netlist is inputted, it will perform a small-signal analysis simulation and output a transfer function according to designated input and output nodes.

Project Specifications

The software should be able to perform a small-signal analysis on a netlist in reduced SPICE format. After parsing through the netlist from a file, it identifies the components and converts the nonlinear components into small-signal equivalents. The netlist will also contain a simulation command, which gives information about the frequency of the circuit. This frequency will be incorporated into the nodal analysis of the modified small-signal circuit. For the nodal analysis, we were introduced to a new method of calculating the voltages at different nodes of the circuit. In this method, a conductance matrix is used. Then we multiply the inverse of the conductance matrix to a vector containing the current in and out of each node, where each row represents a node. The result of this multiplication produces the voltage at each node.

After performing the nodal analysis, a transfer function is outputted according to the appropriate input and output nodes. Calculating the DC operating points requires additional calculations, involving the voltage at each node then using the Newton Raphson method to narrow down the values. The small-signal parameters of the nonlinear components were assumed from the LTSpice datasheet.

Functional & Non-Functional Requirements

Functional Requirements	Non-Functional Requirements
The program should be able to interpret the SPICE format correctly: - Understands the SPICE commands - Understands the order of the netlist - Ignores notes in the netlist (starting with *)	**Should be easy for the user to input their text files** - All .txt files should be easily linked to the program - Since the intended audience isn't necessarily an engineer, important for it to be user friendly
Nonlinear components should be converted to small signal equivalents - Able to remove the nonlinear component and replace it with the small-signal circuit	**Processing time should not require long periods of time** - Aim to be able to process complex circuits in less than 15 minutes
Calculates the voltage at each node - This means that the output should clearly represent each node	**The output should be easy to interpret by the user** - The output should be clear and concise and doesn't include irrelevant information
The output should be accurate - We can ensure this by simulating a netlist in LTSpice	**Should be usable on other devices** - The device has a piece of software able to run and compile the program
Answers are outputted row by row making it easy to convert to CSV	**Easy to add and delete components from the netlist**
Able to handle complex numbers - Since capacitors and inductors are included, therefore, we should be able to handle complex impedances	

Above are several main requirements to implement in the software. For the full SRS see Appendix 1.

Project Planning

Once we finished our group composition, we started discussing a group name and communication platforms for sharing files. After a fierce discussion, the name "3-to-HACK" came out. We all decided that Discord would be the main form of communication for our meetings as starting calls were much simpler and sharing our screen would work similarly to Teams. We also set up a Whatsapp group chat that would allow us to ask questions about the project and update team members when meetings were not scheduled. Initially, we wanted to use VSCode to share our code and work on the code collaboratively, however, we shortly found several difficulties with the collaboration installation. Therefore, we decided to write our functions and share the main.cpp on a Teams chat. We also decided that meetings would occur at least once a week to share our work and discuss any problems that arise. Extra meetings were scheduled as needed. During the meeting we put our notes in another Google Doc, this Doc contains what we discussed as well as questions that arose during discussions. We would bring these questions up when we had meetings with the TAs, this was especially helpful as we were able to get direct guidance for our program.

Team Roles

Firstly we had to establish team roles, which gave us a sense of place and direction. From a brief google search, we were introduced to the Belbin Team Roles, which helps us identify our strengths and weaknesses to work effectively in a group [1]. Through self-evaluation, we identified which of the 9 roles were our strengths and weaknesses.

Charmaine’s strengths are implementer and shaper, while her weaknesses are plant and completer finisher. Therefore, her role in the team was program designer, as she was able to turn her ideas into a program and had a clear idea of which functions were needed to be included in the program. Another role assigned to her was report coordinator since she is driven to complete the project. She was able to give the necessary motivation to complete the report by precisely listing the sections needed in the report to document our process.

Completer finisher and plant are Kelvin's strengths, while specialist and shaper are his weaknesses. His role in the team was as a researcher because he could develop different solutions or ideas depending on what was needed. Another role assigned to him was a program analyser. Besides debugging and integrating each section of the program into an overall picture, he made sure it met the task standards and checked for errors.

Amy’s strengths are monitor evaluator and teamworker, who can identify the work required and finish it on behalf of the team. While her weaknesses are plant and shaper, she lacks creativity and the courage to challenge herself and face obstacles. Therefore, her role in the team was programmer, as she could listen and follow her teammates’ ideas, analyse and judge all options accurately and convert them into structured program code and perform integration testing.

Diversity is evident in our group, but we remain balanced. Several of the weaker roles are the strongest for other team members, which makes the group complementary.

Team Management

According to Tuckman's Model, several stages occur during the beginning stages of a team. Tuckman calls these stages Forming, Storming, Norming and Performing [2]. During the forming stage, the team begins to get to know each other. We started thinking of team names and made a team group chat. This allowed us to have casual conversations and become acquainted with each other. The storming stage refers to the stage where conflict arises, whether about working styles or workload [3]. This stage occurred when conflict arose due to the large amount of work to complete but lack of direction. To solve this, we created a Gantt Chart that clearly showed each person’s workload and weekly meetings. Next is the norming stage, where the team becomes more amicable and comfortable with each other. Weekly meetings allowed us to communicate more effectively and ask for help when needed. Lastly, the performing stage occurs when the team is working at peak efficiency [2], meaning the team is working towards a clear attainable goal. This stage occurred near Phase 3, where we worked as a team to improve each other’s programs.

However, this model is not chronological, there were times where we would enter the storming stage again after working in the performing stage. For example, during the middle of our refinement, we were unable to solve the issue of converting the nonlinear components into small signal equivalents. This gave us a lack of motivation as we were unable to identify the problem and unsure of where to search for help. However, booking a session with TAs who helped answer some of our questions led us back to the performing stage.

Gantt Chart

We utilised a Gantt Chart to plan out the weeks of the project to use our time efficiently. We split up the 5 weeks into 5 phases, Research & Brainstorming, Prototyping & Design, Refining, Testing and Evaluation. A Gantt Chart can help us visualise our project timeline,

each task is represented at a bar on the chart and its length is the timeframe assigned. We also added the percentage of the task completed. This helped us to keep track of our work, for example, we may reach out to members with little progress to see whether they need additional help or run into errors that they were unable to solve. Tasks were also added at the end of our weekly meeting to confirm that we all have work to complete and understand the deadline. Fig. 1 is our final Gantt Chart, we colour coded them by names to easily identify our tasks. Tasks that were completed together would be labelled “All”. We added milestones at the end of the chart which allowed us to recognize the transition of one phase to another phase.

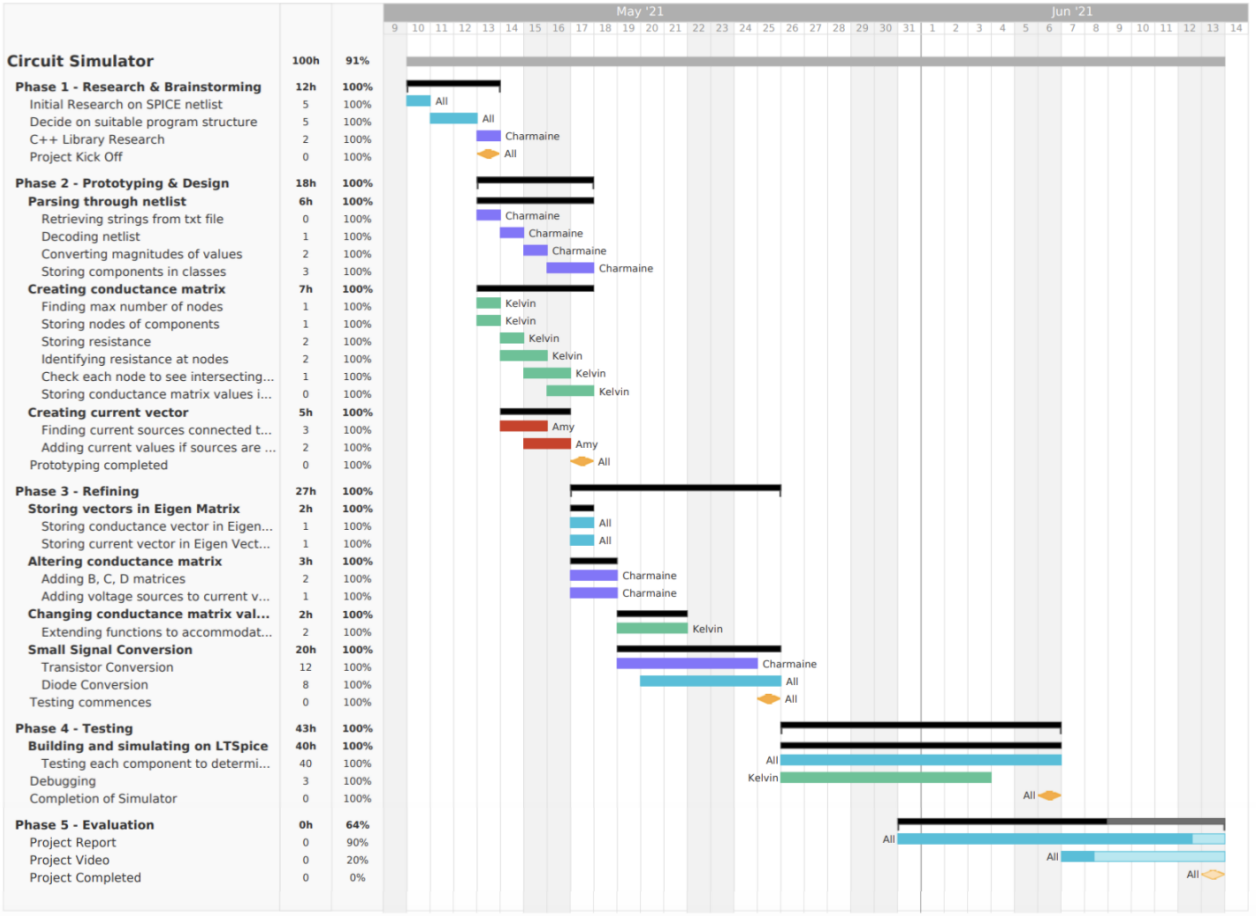


Figure 1. Final Gantt Chart

Since the tasks we decided on were quite broad and we could not foresee what other subtasks would be needed, we decided to add subtasks individually as we were working on the task. In between the meetings, we updated each other through the group chat and set up meetings as needed if there were urgent matters.

As seen in Figure 1, we put most emphasis on Phase 4. Testing was important for us as the logic in the code may seem right, however, there may be underlying issues or undefined

errors that we hadn't taken into account. Even though testing was done throughout Phase 2 & 3. Phase 4 allowed us to test the simulator as a whole instead of each function to confirm that all functions work together harmoniously. During the test, we came across a lot of SIGABRT errors that are related to memory allocation and fatal errors [3]. We were glad that we scheduled extra time for the testing to solve these issues.

During the testing process, we also started writing up the report on a new Google Doc. We put down bullet points regarding points that should be covered in each section. Then individually wrote different sections. Later, we had a meeting after completing the sections to ensure that each section flowed well when put together.

Tasks were distributed according to each person's strengths. For example, during the meetings when discussing parsing through the netlist, Charmaine gave ideas on the functions needed to complete this task and seemed to have a clear idea on how to accomplish this task. Therefore, this task was allocated to her. Kelvin was in charge of debugging the program as he was more meticulous and thought of all possible reasons which would cause the program to act differently. Amy was responsible for writing the current vector function, there was quite a lot of research that needed to be done as we were not familiar with the method of using a conductance matrix. During team meetings, she would quickly send links giving information about the topics in discussion. Therefore she was able to effectively complete the research and function.

Design Process

Initial Research

After evaluating the specification provided, we have initially identified several significant steps to complete this small-signal analysis program. Then, we started doing some research on how to implement it into our C++ program. Our research included:

- Specialised C++ libraries for matrix multiplication.
- How would the netlist details from the simulation file be extracted?
- Appropriate data structure to store components.
- Structure of the conductance matrix and how it works.
- Nonlinear components' small-signal equivalent circuits

We decided to spend a week doing independent research and discuss what we discovered in the next meeting. This allowed us to understand our project better and save time by only researching some topics. Another advantage would be we can learn from each other and provide different perspectives towards the same problem.

Data Structure

Firstly, this project has a large number of different working components therefore it was important to correctly store these variables to make them easily accessible in functions and the main. Initial data structures that came to mind were structs and classes.

Structs

Structs contain members of different data types, this was useful as we have many different variables which have different types (eg. string or int) [4]. After naming the struct, its member functions can also be easily retrieved and it was something that we were also familiar with. However, structs are not very suitable if we were to change the values of the member function throughout the program, therefore, ultimately it wasn't chosen.

Classes

Classes not only contain member variables, but they also contain data members [5]. This means that classes can have dependencies using the member variables declared in the class. Since we have quite a number of components, which share similar variables (eg. all components will have variables n0 and n1). We thought of creating a class that contained member variables shared across all the data members. Data members would be the different components allowed in the SPICE format. Each data member will also contain member variables that are specific to the component (eg. BJTs would have an extra node,

n2). We declared member functions used for small signal conversion, to make specific small-signal functions for the component. Therefore, this was the data structure that we chose as it was easy to use and fulfilled the needs of the project.

Library Selection

Since this method of solving a circuit requires solving a conductance matrix, a function that achieves inverse matrix multiplication is essential. Initially, we had thought of writing a function that would output the inverse of a matrix and another matrix that would multiply 2 matrices. Through some brief research, we found out that the easiest method of finding the inverse matrix would be to find the adjoint of the matrix [6]. However, this required a large number of functions just to accomplish the inverse matrix, therefore, we revisited the project specifications and did some research upon external C++ libraries. Before research, we had little understanding of C++ libraries except for the standard one [7]. Firstly we searched up external libraries which were commonly used for matrix manipulation.

Eigen

Eigen is a versatile C++ library that is capable of matrix manipulation and also has geometric features [8]. It is also extremely easy to install since it doesn't have any other special dependencies. Another benefit would be the dynamic size of the matrix. For this purpose, dynamic sizes would be useful as the size of the matrix depends on the number of nodes on the input file. Since it is dependent on the standard library, this meant that it also used commands from the standard library which we were already familiar with.

Armadillo

Armadillo is another C++ library specialising in linear algebra and scientific computation [9]. It uses a similar syntax as MatLab, which we have previously used in Topics in EE. Similar to Eigen, it also allows dynamic sizing which is beneficial in this project. However, its installation seems more complex as compared to Eigen as pre-built packages are also needed to complete its installation. Although we have previously used MatLab briefly, we haven’t learnt the MatLab language, therefore, the MatLab syntax wasn’t beneficial to us.

IMSL

IMSL has been used by different programmers for 50 years, proving to be reliable. It also has an extensive list of functions, making it easier to find the correct commands needed. Not only does it support linear algebra, but it also supports statistics and data mining [10], showing that it is a well-developed library. However, it required a license to install, therefore, we did not select this library in the end.

By evaluating the available C++ libraries, we decided that Eigen would be the most suited for our purpose. This was because the program only required the inverse matrix and matrix multiplication function. Installation was also very simple, since this was our first time using C++ libraries we weren't familiar with methods of installation. With dependencies only on the standard library, using Eigen syntax was much easier.

Input Method

Input file

Since the project specification, specifies that a plain text file containing the netlist will be inputted. We have included a *cin* line to retrieve the name of the file as well as the input and output nodes for the transfer function. If the file cannot be found, an error message will appear.

Data structure

Two data structures, vectors and linked lists were considered at first.
In C++, vectors are dynamic arrays that are used to store data and provide a greater level of flexibility. Vectors can resize themselves automatically when an element is added or removed. [11]. A linked list is a linear data structure used to store data items. Unlike arrays, it does not store items in contiguous memory locations.[12] . Vectors are adequate for random reads, insertion and deletion in the back but are not as effective for insertion and deletion in the front. In contrast, linked lists are particularly good for adding and removing items in front or back (constant) [13]. Since we have to read each part of the netlist (shown in Fig 2.) and the format of each line is fixed, vectors are more appropriate to hold the information in the netlist.

```
* A test circuit to demonstrate SPICE syntax
V1 N003 0 AC(1 0)
R1 N001 N003 1k
C1 N001 0 1µ
I1 0 N004 0.1
D1 N004 N002 D
L1 N002 N001 1m
R2 N002 N001 1Meg
Q1 N003 N001 0 NPN
.ac dec 10 10 100k
.end
```

Figure 2. Example Netlist

Decoding the input

Since the input variables are separated by spaces, using the *infile.open()* function would not work as it doesn't take the whole line of the input [14]. Therefore, the function *getline* was used, it can get the whole line of a string which was useful in this case [15]. After using *getline*, the vector of strings is passed through a for loop. A function called *decodeNetlist* is needed to separate the input string according to the spaces in the string. Then the first character of the string is analysed to see which component it belonged to. As mentioned above, classes are used to store the component variables, therefore, each data member has an *add* function that initialises the correct variables according to the SPICE format. Since we have also created a vector for each type of component, depending on the first character, the *add* function is called and the new data member variable is added into the corresponding vector. To access the different components, we indexed the components vectors and then retrieved the member variables from each index. Not only was it important to store the components, but it was also important to store the variables from the command line of the netlist. Variables were declared in the main, then they were initialised according to the reduced SPICE format. The for loop stops when it reaches the last line containing ".end".

Matrix Formation

Conductance matrix

Next, we considered constructing and solving the conductance matrix. The conductance matrix (A) is formed by the combination of 4 smaller matrices (G, B, C, D) (Fig 3.). The conductance matrix A has the dimensions nxm, which "n" represents the number of nodes in the circuit, "m" is the number of independent voltage sources [16].

$$\mathbf{A} = \begin{bmatrix} \mathbf{G} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

Figure 3. New conductance matrix format

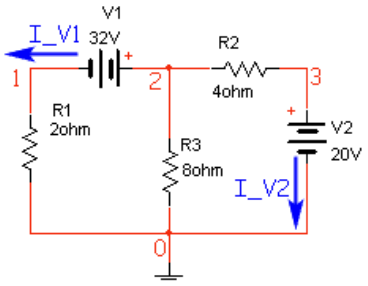


Figure 4. Sample Circuit [16]

Fig. 5 is the G matrix for Fig. 4, it has dimensions *nxn* and is determined by the connections between their passive circuit elements (resistors). A two-step process can be used to form it. The diagonal matrix is equal to the sum of the conductance (one over the resistance) of

each element connected to the corresponding node. Therefore, the first diagonal element is the sum of conductances connected to node 1, the second diagonal element is the sum of conductances connected to node 2, and so on. The non-diagonal part of the elements are the negative conductances of the element connected to the pair of corresponding nodes. For instance, a resistor between nodes 1 and 2 goes into the **G** matrix at location (1,2) and (2,1) [16]. As our netlist includes capacitors and inductors, they should also be included in the conductance matrix. Hence, the matrix would include admittance.

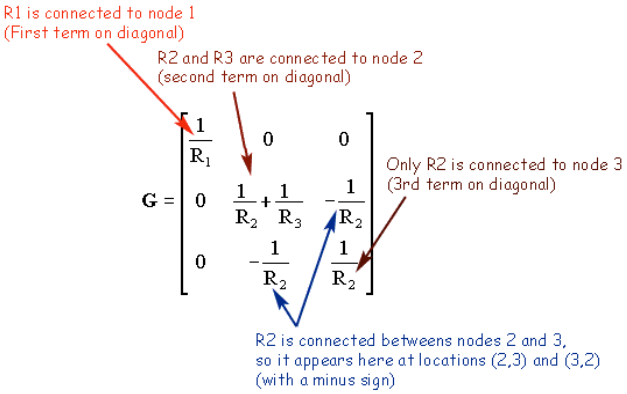


Figure 5. Sample matrix G [16]

The B matrix in Fig. 6 is size nxm and is determined by the connection between the voltage sources. It is an *nxm* matrix with only 0, 1 and -1 elements. In the matrix, each location corresponds to a voltage source (first dimension) or a node (second dimension). If the positive terminal of the *i*th voltage source is connected to node *k*, then the element (*i,k*) in the **B** matrix is a 1. If the negative terminal of the *i*th voltage source is connected to node *k*, then the element (*i,k*) in the **B** matrix is a -1. Otherwise, elements of the **B** matrix are zero. The C matrix is *mxn* and is determined by the connection of the voltage sources. Matrix B and C are closely related, in which C is the transpose of the B matrix. [16]

$$\mathbf{B} = \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 6. Sample matrix B [16]

The D matrix (Fig. 7) is *m*x*m* and is composed entirely of zero if only independent sources are considered [16]. However, it can be non-zero if the BJT small signal model was included in the netlist.

$$\mathbf{D} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Figure 7. Sample matrix D [16]

As the small-signal equivalent model of the BJT included a voltage-controlled current source, the submatrix G must be altered to account for the current from the VCCS now. Transconductance was added to the rows and columns, respectively, but the size of G remained the same since no voltage source was added [17]. To use our calculated conductance matrix in the Eigen library, we stored it as a vector. For example, index 0 stores the matrix location (1,1), index 1 stores (1,2), etc.

The idea we could implement was creating a vector for each node whose index was the impedance of passive components connected to each node. In the next step, these vectors were stored in a large vector named, std::vector<std::vector<std::complex<double>>> rnode. The matrix contained all the nodes as well as the values of each node's impedance. For example, if node 1 has resistors with 50 and 30 Ohms, 50 was stored in rnode[0][0] and 30 in rnode[0][1] and so on. Then we would compare each node and see if they had the same passive components and push that back into our conductance matrix vector.

Current vector

To find the unknown voltages at each node, we used nodal analysis by inspection to solve simultaneous equations from Kirchoff's current law (KCL), and applied them at different nodes in the circuit using matrices.
For example, when we considered the circuit shown in Fig. 8 for nodal analysis, we could do it by inspection and plugging in numbers into the matrices, but knowing the equations behind them are also very important.

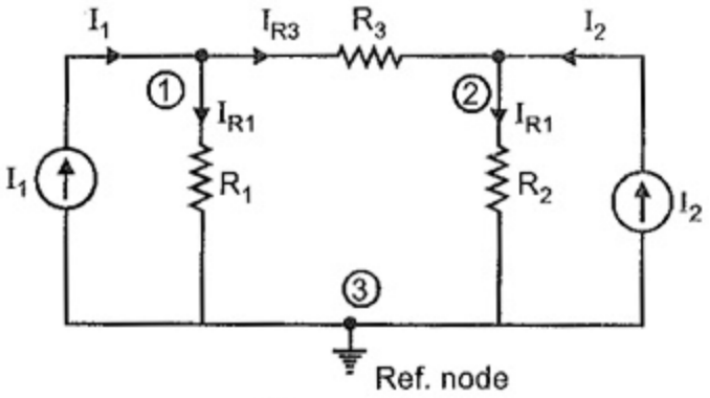


Figure 8. Sample circuit [18]

At Node 1, by applying KCL, the total current entering the circuit's junction is exactly equal to that of leaving the same junction: $\sum I_{in} = \sum I_{out}$, the idea is also known as the **Conservation of Charge**.

$I_1 = I_{R1} + I_{R3}$ $I_1 - I_{R1} - I_{R3} = 0$	where $I_{R1} = V_1 / R_1$ $I_{R3} = (V_1 - V_2)/ R_3$
$I_1 – (V_1 / R_1) – ((V_1 - V_2)/ R_3) = 0$ $I_1 – V_1G_1 – V_1G_3 + V_2G_3 = 0$ $V1(G1 + G_3) - V_2G_3 = I_1$	where $G_1 = 1/R_1$ $G_3 = 1/R_3$

After also considering Node 2, we would have got this matrix shown in Fig. 9:

$$\begin{bmatrix} G_1 + G_3 & -G_3 \\ -G_3 & G_2 + G_3 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \end{bmatrix}$$

Figure 9. Sample matrix [18]

The generalised node equations can be written as [G][V] = [I]

G is the node conductance matrix

V is the column matrix of the node voltages with respect to the reference node

I is the column matrix of input currents

Now knowing how we got the matrices from equations, we can do it by inspection. The previous part has already explained how we created the conductance matrix, so in this part, we would focus on the current matrix.

First, we convert all the voltage sources to equivalent current sources:

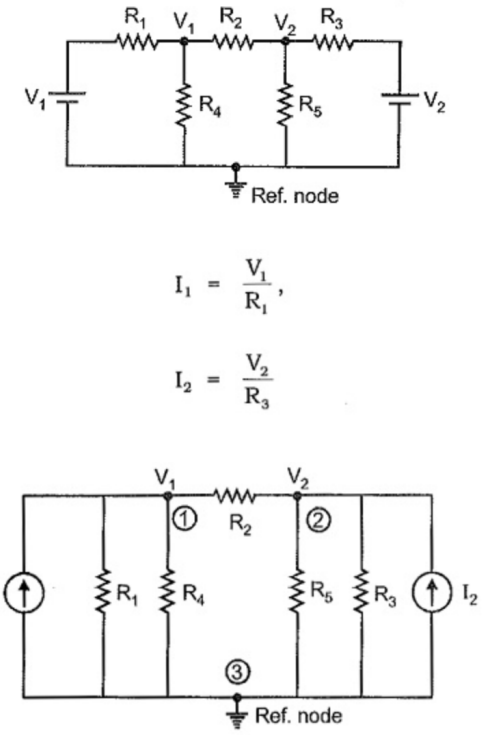


Figure 10. Sample circuits [18]

In Fig. 10, I_1 denotes the value of the current source to node 1 and is written on the right-hand side of the equation. The sign of I_1 is positive if it is flowing towards node 1, otherwise, it is negative. If no current source is connected to node 1, then $I_1 = 0$ [18].

So, in order to find the net current entering the node, we need to write a function that sums up all the current entering the node respectively and deducts all the current leaving the node respectively again.

Voltage and Current sources

In AC analysis, voltage and current sources were removed (shorted and opened respectively), except for input and dependent sources.

When we used the voltage source we converted it into its ideal equivalent resistance to find the overall resistance. The internal resistance is zero for an ideal voltage source, so it behaves as a short circuit. Similarly, we did the same thing to the current source, its ideal resistance was almost infinity, so we took it as an open circuit.

To be easily understood and calculated, we only needed a circuit with resistance.

Implementation

Input

Decode Netlist

decodeNet takes in a string and returns a vector of strings. The function aims to separate the input string into a vector of strings that are separated by a space in the input. As seen in Fig. 11, the whole string is known as “string” and “netList” is the vector that we will return at the end of the function.

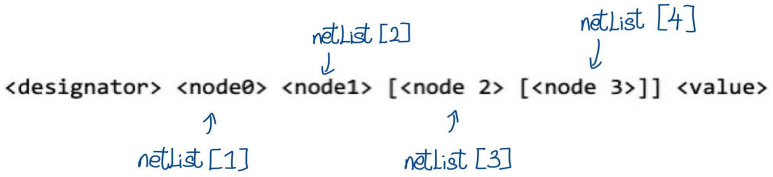


Figure 11. Decoded netlist

Within the function in Fig. 12, another string variable, “s” is declared. A for loop is also used to traverse across the string, checking each character of the string. ‘s’ is cleared at the start of the for loop. As we traverse the loop, the characters of the input string are added into “s” with the “+=” operator. The loop checks for a space, when detected it will promptly stop the addition of new characters.

```
std::vector<std::string> decodeNet (std::string string){
    std::vector <std::string> netList;
    std::string s;

    for (int j = 0; j < string.length(); j++){
        s.clear();
        while (string[j] != ' ' && j < string.length() && !check_A(string[j])){
            s += string[j];
            j++;
        }
        if (check_A(string[j])){
            while (j < string.length()){
                s += string[j];
                j++;
            }
        }
        netList.push_back(s);
    }

    return netList;
}
```

Figure 12. decodeNet code

“s” now contains one section of the input string (eg. netList[0]), this new string is pushed back into “netList” which is the returned vector. However, there is a special case, since an AC source is inputted as AC(1 0), this algorithm wouldn’t necessarily work as there is a space between the phase and amplitude. *check_A* is used to check whether there is an A in the next character of the string, if so, it will add the rest of the characters of the input string into “s”. Then “s” is pushed back normally into the “netList” vector. After separating the input string, these values need to be added to the correct component vector. Therefore the *add* function is needed.

The *add* function varies for each component, it returns a variable of that component type and is passed through by a vector of strings. It is declared in the data member inherited by the class NetList. Below is an example of the voltage source *add* function.

```
voltageSource add (std::vector<std::string> netList){
    voltageSource volt;
    if (netList[3][0] == 'A'){
        volt.ind = 1;
        std::vector<double> vec = convertAmpPhase(netList[3]);
        volt.complexVal = std::polar (vec[0], (vec[1] * pi / 180));

    }
    else{
        volt.ind = 0;
        volt.dcVal = std::stoi(netList[3]);
    }
    volt.n0 = node(netList[1]);
    volt.n1 = node(netList[2]);
    return volt;
}
```

Figure 13. voltageSource add function

Fig. 13 returns a variable of the component type. In this case, it will return a “voltageSource”. Since there are no extra nodes for a voltage source, its nodes are stored in variables ”n0” and ”n1”. *node* is a function that will convert the string into an int value (eg. N001 becomes 1). As the sources have a special case, the AC values are split into amplitude and phase. The value of the voltage source is in index 3 of the input vector. If it is a special case, its amplitude and phase will need to be extracted using the *convertAmpPhase* function, then converted to its polar form. This simplifies the process of inserting the value of the voltage source into the conductance matrix. The *convertAmpPhase* function is similar to the *decodeNet* function, it separates the amplitude and phase from an input string and returns the 2 values in a form of a vector (eg. AC(1 0) gets separated into vec[0] = 1 and vec[1] = 0).

Another version of the *add* function (Fig. 14) for a resistor includes a function *splitVal*. This function takes in a string and returns a type double. *splitVal* will scan the last character of the input to analyse its magnitude and multiply the numerical value accordingly. Except for a special case where magnitude is Meg, the value is usually obtained by deleting the last character of the string, in this case, the last 3 characters are deleted.

```
resistor add (std::vector<std::string> netList){
    resistor r;
    r.n0 = node(netList[1]);
    r.n1 = node(netList[2]);
    r.res = splitVal(netList[3]);
    return r;
}
```

Figure 14. resistor add function

```
for (int i = 0; i < string.size(); i++){
    if (string[i][0] != '.' && string[i][0] != '*' ){
        netList = decodeNet(string[i]);
        store_node(netList, nodes);

        if (netList[0][0] == 'V'){
            voltageSource volt;
            v.push_back(volt.add(netList));
        }
        else if (netList[0][0] == 'I'){
            currentSource curr;
            I.push_back(curr.add(netList));
        }
```

Figure 15. decodeNet and add functions used in main

These *add* and *decodeNet* functions are used in a large for loop in the main as seen in Fig. 15. The loop traverses through the strings retrieved from the input file. The strings are then decoded and the first character of the strings are checked to see which component is in the netlist. Then the component is

converted to its corresponding type and pushed back into its corresponding vector.

Small Signal Equivalent

The small-signal equivalent function is also declared in the data members. This is because not all components are nonlinear and each nonlinear component has a different small-signal equivalent.

Diode

A diode is replaced by a resistor for small signal analysis. In this function, assumptions are made about its saturation current, ideality factor and thermal voltage, these assumptions were found from the LTSpice data sheet which is found within the LTSpice software [19]. By using these assumptions, the current across the diode is calculated, then it is used to find the value of the SSEM resistor. The function *ssem* in Fig. 16 takes in 2 vectors, one contains the information about the diode component and the other is the resistor vector, which is passed by reference. The first vector “netList” provides information about the nodes connected to the diode. The resistor vector has to be passed in as a reference as we need to push back a new resistor to represent the diode.

```
void ssem (std::vector<std::string> netList, std::vector<resistor>& res){
    std::vector<std::string> netListr;
    resistor r;
    double id;
    double vd = 0.7;
    double is = 2.52 * pow(10,-9);
    double vt = 0.025;
    double n = 1.752;

    id = is * (exp(vd / (n * vt)) - 1);

    double rd = vt / id;

    std::string node_an = netList[1];
    std::string node_ca = netList[2];

    // for diode resistor
    addNetListResistor(netListr, node_an, node_ca, rd);
    res.push_back(r.add(netListr));

    netListr.clear();

}
```

Figure 16. ssem function for diode

The equation to find the current across the diode is known as the Shockley Equation [20]:

$$i_d = i_s \times (e^{\frac{v_d}{n \times v_t}} - 1)$$

The resistance is calculated using the thermal voltage and current:

$$r_d = \frac{v_t}{i_d}$$

These equations are used to find the resistance of the resistor, then the nodes of the diode are extracted by indexing the ”netList” vector. *addNetListResistor* is a function that returns a vector following the SPICE format, as that means the *add* function (explained earlier) is still applicable in this case. Then the new resistor is pushed back into the referenced resistor component vector.

BJT

The small-signal equivalent of a BJT is represented by Fig. 17.

Figure 17. BJT Small Signal Circuit

The SSEM contains 2 resistors and 1 voltage-controlled current source. Similar to the diode, the collector current, thermal voltage, early voltage and beta value are all taken from the LTSpice datasheet in the software [19]. By using the assumed values, r_o and r_{be} can be calculated with the below equations.

$$g_m = \frac{i_c}{v_t}$$

$$r_{be} = \frac{\beta}{g_m}$$

$$r_o = \frac{v_T}{i_c}$$

```
void ssem (bjt b, std::vector<controlSource>& g, std::vector<resistor>& r, std::vector<std::string> netList){
    double ic = 0.8;
    double vt = 0.025;
    double va = 100;
    double beta = 200;
    std::vector<std::string> netListr;
    std::vector<std::string> netListg;
    resistor res;
    controlSource cs;

    double rbe;
    double ro;
    double gm = ic / vt;

    rbe = beta / gm;
    ro = va / ic;

    std::string node_c = netList[1];
    std::string node_b = netList[2];
    std::string node_e = netList[3];

    //for rbe
    addNetListResistor(netListr, node_b, node_e, rbe);
    r.push_back(res.add(netListr));

    netListr.clear();

    //for ro
    addNetListResistor(netListr, node_c, node_e, ro);
    r.push_back(res.add(netListr));

    netListr.clear();

    //for current source
    addNetListCS(netListg, node_c, node_e, node_b, node_e, gm);
    g.push_back(cs.add(netListg));

    netListg.clear();
}
```

Figure 18. ssem function for BJT

In Fig. 18, the *ssem* function for the BJT, a vector for ”netList” is passed in and the resistor vector is also passed in by reference. Since this small signal equivalent has an extra dependent source, the voltage-controlled current source is also passed in by reference. The resistances and transconductance are calculated using the equations defined above. The new resistors and dependent source are added similar to the diode, where *addNetListResistor* is used but *addNetListCS* is also used. *addNetListCS* acts the same way as *addNetListResistor* but it follows the dependent source SPICE format.

Construct and Solve the Conductance matrix

Conductance matrix

store_nodercl (Fig. 20) takes in three vectors of type resistor, capacitor and inductor and returns a vector of integers. By using this function, you can store the nodes associated with every passive component. We stored the information of the resistor, capacitor and inductor into three vectors r, c and l with type “resistor”, “capacitor”, and “inductor”, respectively. Hence, in the main (Fig. 19), we put in “r”, “c”, and “l” into the *store_nodercl* function and store it into a vector of type int “node_list”.

```
std::vector<int> node_list = store_nodercl(r, c, l);
```

Figure 19. Reference in the main

A vector of type int "nodesize" was declared within Fig. 20, and three for loops are used to push back the node values into it. The operation of the three for loops were similar, traversing the vectors of r, c or l and pushing back the elements stored in them by calling member variables ".n0", ".n1", where their nodes are stored, respectively. Hence, the vectors contained the nodes associated with resistors first, then capacitors and inductors.

```
//Store each nodes into an int vector
std::vector<int> store_nodercl (std::vector<resistor> node1 , std::vector<capacitor> node2 , std::vector<inductor> node3 ){
  std::vector<int> nodesize ;
  for (int i = 0 ; i < node1.size() ; i++){
    nodesize.push_back(node1[i].n0);
    nodesize.push_back(node1[i].n1);
  }
  for (int i = 0 ; i <node2.size() ; i++){
    nodesize.push_back(node2[i].n0) ;
    nodesize.push_back(node2[i].n1) ;
  }
  for (int i = 0 ; i < node3.size() ; i++){
    nodesize.push_back(node3[i].n0) ;
    nodesize.push_back(node3[i].n1) ;
  }
  return nodesize ;
}
```

Figure 20. Function *store_nodercl*

Take the below netlist Fig. 21 as an example, we could see that the sample netlist involved R1, R2 and C1.

Hence the node_list first push_back the nodes of resistors, then that of capacitors. So the node_list = [1, 2, 3, 0, 2, 0].

```
V1 N001 0 AC(1 0)
R1 N001 N002 1k
C1 N002 0 1u
R2 N003 0 1Meg
I1 N003 0 0.1
.ac dec 10 10 100k
.end
```

Figure 21. Sample Netlist

store_node (Fig. 23) takes in a vector of strings and vector of integers and is passed by reference. The function can store all the node information of all the components as a vector of type int “nodesize”. In the main function, we initialised the vector of type int “nodes” and used a for loop to traverse the netList from the top to the bottom. Then, the function

store_node was called, and the “netList” and “nodes” were placed inside. During this operation, “nodes” will contain all the nodes involved in each component.

```
std::vector <int> nodes;
```

```
for (int i = 0; i < string.size(); i++){
    if (string[i][0] != '.' && string[i][0] != '*' ){
        netList = decodeNet(string[i]);
        store_node(netList, nodes);
```

Figure 22. Reference *store_node* in the main

According to Fig. 11, we can see that different indexes of netList represent different meanings. netList[1] would be node0 , netList[2] would be node1 etc.

Within Fig. 23, we utilised the if statement to help store the nodes in “nodesize”. If the designator is not “Q” or “G” or “M” as they have more than 2 nodes associated with them. Then we would push_back netList[1] and netList[2] into “nodesize”. If the designator is “Q” or “M”, 3 nodes in “netList” would be pushed back into the vector so we need netList[3] as well. If it is none of the above, the remaining one would be “G”, a voltage-controlled current source, which involves four nodes.

```
void store_node (const std::vector<std::string> netList, std::vector<int>& nodesize){
    if (netList[0][0] != 'Q' || netList[0][0] != 'G' || netList[0][0] != 'M'){
        nodesize.push_back(node(netList[1]));
        nodesize.push_back(node(netList[2]));
    }
    else if (netList[0][0] == 'Q' || netList[0][0] == 'M'){
        nodesize.push_back(node(netList[1]));
        nodesize.push_back(node(netList[2]));
        nodesize.push_back(node(netList[3]));
    }
    else{
        nodesize.push_back(node(netList[1]));
        nodesize.push_back(node(netList[2]));
        nodesize.push_back(node(netList[3]));
        nodesize.push_back(node(netList[4]));
    }
}
```

Figure 23. Function *store_node*

The *max_node_size* (Fig. 24) function returns the largest integer from a vector of integers. Our program used it to find the highest number of nodes from the vector “nodes” or “node_list”. The function scans through the vector and stores the highest number of integers into the variable “max”. It was used by other functions to build the conductance matrix. The highest number in “nodes” and “node_list” would also be 3 in our sample netlist.

```cpp
// Find the highest number of the node
int max_node_size (std::vector<int> nodesize){
    int max = nodesize[0];
    for (int i = 0; i < nodesize.size(); i++){
        if (nodesize[i] > max){
            max = nodesize[i];
        }
    }
    return max ;
}
```

Figure 24. Function *max_node_size*

overall_imp (Fig. 26) takes in all the r,c,l vectors and the number of frequencies, and returns a vector of complex numbers representing the admittance of the passive components in the netlist. For the main function (Fig. 25), we would put in r, c, l, and frequency. “freq” is the testing frequency since it will vary during our simulation.

```cpp
std::vector<std::complex<double>> test = overall_imp(r, c, l, freq) ;
```

Figure 25. Reference o*verall_imp* in the main

A complex number vector "tmp" was created within the function and used to store all admittance of “r”, “c”, and “l”. The member function ".imp" returns the impedance of that component as a complex number depending on its type. As before, the admittance of resistors was stored first, followed by capacitors and inductors. Because resistor impedance is not affected by frequency, it remains constant. With capacitors and inductors, their impedance formula involves angular frequency ω, and their impedance will change for different frequencies.

```cpp
//a vector store all the impedance of the R , C and I
std::vector<std::complex<double> > overall_imp (std::vector<resistor> R , std::vector<capacitor> C , std::vector<inductor> I , double freq){
  std::vector<std::complex<double> > tmp ;
  for (int i = 0 ; i<R.size() ; i++){
    tmp.push_back(1.0/R[i].imp()) ;
  }
  for (int i = 0 ; i < C.size() ; i++){
    tmp.push_back(1.0/C[i].imp(freq));
  }
  for (int i = 0 ; i < I.size() ; i++){
    tmp.push_back(1.0/I[i].imp(freq)) ;
  }
  return tmp ;
}
```

Figure 26. Function *overall_imp*

In *rcl_in_vec* (Fig. 27), we returned a vector containing the admittance values of the components connected to a node. The function takes in an integer “n”, which would be the node number (N001 = 1, N002 = 2 etc.), the vector “node_list”, the output of function

overall_imp and the vector “nodes”. The function first uses a for loop to scan all the components inside “node_list”. In the loop, if the content inside “node_list” is equal to “n”, we will push that relative admittance back to the vector we created at the beginning of this function. As resistors, capacitors, and inductors have only two relative nodes, overall_imp[i/2] was the relative admittance. The way “node_list” and “overall_imp” were constructed was consistent, and both stored the components in order. A zero would be inserted automatically when the number of nodes in the netlist is larger or equal to “n”, or if the size of the “Node” is less than *max_node_size* of all components. As all locations had already been assigned, segmentation faults were prevented.

```
// a function that store the resistor values in a vector when the resistor connected to the node n
std::vector<std::complex<double> > rcl_in_vec (int n , std::vector<int> node_list ,std::vector<std::complex<double> > overall_imp , std::vector<int> node){
  std::vector<std::complex<double> > Node ;
  for (int i = 0 ; i < node_list.size() ; i++){
  if (node_list[i] == n){
      Node.push_back(overall_imp[i/2]) ;
    }
  if ( n >= max_node_size(node) && (max_node_size(node_list)<max_node_size(node))){
   Node.push_back(0) ;
  }
  }
  int x = (int) Node.size() ;
  if (Node.size() < max_node_size(node)){
    for (int i = 0 ; i < (max_node_size(node)- x) ; i++){
     Node.push_back(0) ;
    }
  }
return Node ;
}
```

Figure 27. Function *rcl_in_vec*

Node_vec was placed in another function because we did not use this in the main function. All nodes would be stored in std::vector<std::vector<std::complex<double> > > rnode. The first index in the vector would represent a vector of node1, the second index represent that of node2, etc., and the sub index of the vector would be the admittance value associated with that node. In our sample netlist (Fig. 21) , rnode[0][0] = 1/1000 , rnode[1][1] = admittance of C1 , rnode[0][2] = 0 etc.

```
// a function that create a vector <vector <double>> and store the Node in first index and corresponding resistor value in second index
void Node_vec (const std::vector<int> &node_list ,std::vector<std::complex<double> > overall_imp , std::vector<std::vector<std::complex<double> > > &rnode , std::vector<int> &node) {
    for (int i = 1 ; i <= max_node_size(node) ; i++){
    rnode.push_back(rcl_in_vec(i , node_list , overall_imp , node)) ;
  }
}
```

Figure 28. Function *Node_vec*

The final step was to combine everything into our conductance matrix. Using the function *conduc_mat*, we will create a vector that contains the information of the conductance matrix. It was built through two for loops, with “i” and “j” representing the two nodes of the matrix. For example , G_{11} (i = 1 , j = 1) is the total conductance connected to node 1 and G_{23} (i = 2 , j

= 3) is the conductance directly connected to node 2 and node 3 . For searching the values in the vector, two functions were used. *search_vector* (Fig. 29) would return the index of the vector if the content inside is equal to “n”. *intersection_vector* (Fig. 29) would return a vector containing the values of the two input vectors that overlap.

```
int search_vector(std::complex<double> n, const std::vector<std::complex<double> > vin){
    for(int i=0; i<vin.size() ; i++){
        if(vin[i]==n){
            return i;
        }
  }
  return -1;
}

void intersection_vector(const std::vector<std::complex<double> > &vin1, const std::vector<std::complex<double> >& vin2,std::vector<std::complex<double> >& vout){
    for(int i = 0 ; i < vin1.size() ; i++){
        if(search_vector(vin1[i], vin2)!=-1){
            vout.push_back(vin2[search_vector(vin1[i],vin2)]);
        }
    }
}
```

Figure 29. Functions of *search_vector* and *intersection_vector*

In the for loops, a new complex variable cond = 0 and a vector cond_mat were created. We then used the function *intersection_vector* to find the intersect components in two vectors, which would be rnode[i-1] and rnode[j-1], and returned into cond_mat. Hence, if i = 1 and j = 2, we would be finding the intersection components between node 1 and node 2. Afterwards, the variable cond would then be the sum of the admittance inside the mat cond_mat and was pushed back into our vector “final”. If “i” is equal to “j”, the final will be the value of “cond”. Otherwise, it would be the negative value of “cond”.

```
// function to store the conductance matrix in a vector
void conduc_mat (const std::vector<int> &nodes , const std::vector<std::vector<std::complex<double>>> &rnode , std::vector<std::complex<double> > &final ){
  for (int i = 1; i <= max_node_size(nodes);i++){
    for (int j = 1; j <= max_node_size(nodes); j++){

    std::complex<double>  cond = 0 ;

    std::vector<std::complex<double> >  cond_mat ;

    intersection_vector(rnode[i-1] , rnode[j-1] , cond_mat);

    for (int v = 0 ; v < cond_mat.size() ; v++){
      cond = cond + cond_mat[v] ;
    }
    if (i == j){
      final.push_back(cond) ;
    }
    else{
      final.push_back(-cond) ;
    }
    cond_mat.clear() ;
    }
  }
}
```

Figure 30. Function *conduc_mat*

After producing the conductance matrix, we noticed that if the netlist contained a dependent source, we would have to consider the transconductance in the conductance matrix. The equation for transconductance is:

$$g_m = \frac{i}{v}$$

This means that it can be added to the conductance matrix without finding its reciprocal. To alter the conductance matrix for dependent sources. We add the transconductance to the correct index of the conductance matrix [17]. The SPICE format of a voltage-controlled current source and the diagram of the current source is as shown in Fig. 31.

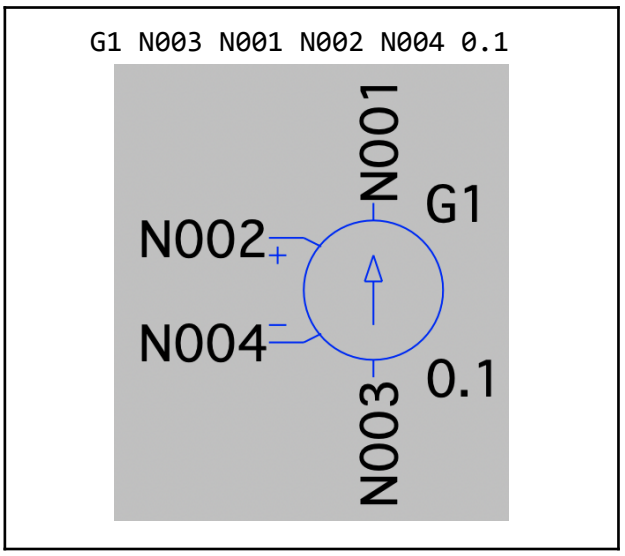


Figure 31. Voltage-controlled current source in LTSpice

Where the 1st and 2nd node represents the positive and negative side of the dependent source respectively. The 3rd and 4th node represents the positive and negative side of the reference source. To add the transconductance, we need to look at the nodes and add the transconductance to corresponding rows and columns.

In this case, the first 2 nodes correspond to the row and the last 2 nodes correspond to the column of the conductance matrix. The value at the row corresponding to the first node and the column corresponding to the 3rd and 4th nodes will add g_m and $-g_m$ respectively. Similarly, the value at the row corresponding to the second node and the column corresponding to the 3rd and 4th nodes will add $-g_m$ and g_m respectively [17]. The table below illustrates the algorithm of the conductance matrix for a voltage-controlled current source.

Row/Col	3rd Node	4th Node
1st Node	$+ g_m$	$- g_m$
2nd Node	$- g_m$	$+ g_m$

Since the conductance matrix is already built using commands from the Eigen library and for loops, the easiest way to implement this algorithm would be to write a function that can check the nodes then add g_m accordingly.

The function *check_depV* in Fig. 32 takes in a type "controlSource" that checks the nodes, and it also takes in the row and column from the conductance matrix. It returns -1, 0 or 1 depending on the row, column and nodes. The returned value is then multiplied with g_m and added to the value at the corresponding row and column of the conductance matrix.

```
int check_depV (controlSource g, int col, int row){
    if (g.n0 == row){
        if (g.n2 == col){
            return 1;
        }
        else if (g.n3 == col){
            return (-1);
        }
        else{
            return 0;
        }
    }
    else if (g.n1 == row){
        if (g.n2 == col){
            return -1;
        }
        else if (g.n3 == col){
            return 1;
        }
        else{
            return 0;
        }
    }
    return 0;
}
```

Figure 32. Function to check for dependent sources

As seen in Fig. 33, in the main we used 2 for loops to traverse and initialise the conductance matrix

```
if (g.size() != 0){
    for (int row = 0; row < N; row++){
        currVec(row) = currReal[row+1];
        for (int col = 0; col < N; col++){
            condMatrix(row, col) = (conmatreal[k]) + (g[0].transc * check_depV(g[0], col, row));
            k++;
        }
    }
}
```

Figure 33. Initialisation of conductance matrix

Current Vector

Nodal analysis by inspection can be performed by computing matrices for each node that satisfies Kirchoff's current law. To compute the current matrix, we needed a function to calculate the net current entering each node.

Firstly, from Fig. 34, we have declared:

- *curr_in* as the current entering a node
- *curr_out* as the current leaving a node
- *curr_total* as the net current entering a node

```
std::vector<double> curr_in;
std::vector<double> curr_out;
std::vector<std::complex<double>> curr_total;
```

Figure 34. Declaring the vectors

First for loop

Fig. 35 shows that we have pushed back 0 for all 3 variables until the loop has reached the "m". *"m"* represents the maximum number of nodes stored in the *node_list,* so that we could ensure every single node in the circuit has a cell.

```
for (int i = 0; i <= m; i++){
    curr_total.push_back(0);
    curr_in.push_back(0);
    curr_out.push_back(0);
```

Figure 35. Pushing back 0 in the current vector

Second for loop

> The format of each line is:
>
> `<designator> <node0> <node1> [<node 2> [<node 3>]] <value>`

In the sample netlist, a current source is computed as:

```
I1 N004 0 0.1
```

This line is telling us that there's 0.1A current entering Node 4 and 0.1A leaving Node 0 (ground node). Therefore, even if the netlist has changed, each line is still going to tell us that `<value>` current is entering `<node0>` and leaving `<node1>`.

From the previous loop, we have pushed back "m" cells for *curr_in* and *curr_out*, the first cell is going to be the current entering `N001` for *curr_in* and leaving `N001` for *curr_out*.

In the loop in Fig. 36,

```
curr_in [curr[1].n0] = curr_in [curr[1].n0] + curr[1].dcVal
curr_out [curr[1].n1] = curr_out [curr[1].n1] - curr[1].dcVal
```

`curr[1].n0` would give us the node that receives current from I1, while `curr[1].n1` would give us the node that gives current to I1.

From the sample netlist, `curr[1].n0` is going to give us 4 because there is 0.1A entering Node 4 for `I1`, and `curr_in[4]` is going to access the current in the 4th element in the vector, which is the cell for `N004`. Since we have to increment the current entering a single node, we sum up the value entering that specific node with the command `curr[j].dcVal`.

curr.size() represents the number of cells in the component `I`. In the sample netlist, there is only 1 current source which is `I1`, so curr.size() is 1. *If (j < curr.size())* is going to make sure we have gone through every single line with the designator `I`, which we could also interpret as all current sources are included in the calculation.

```
for (int j = 0; j <= m; j++){
    if (j < curr.size()){
        curr_in[curr[j].n0] += curr[j].dcVal;
        curr_out[curr[j].n1] += -curr[j].dcVal;
    }
```

Figure 36. Inputting current values into curr_in and curr_out

Last for loop

The for loop shown in Fig. 37 adds up the current entering the n^{th} node and leaving the n^{th} node and saves the values into *curr_total* for the n^{th} node. It then loops through every node and exits when we have pushed back the net current entering every node. At last, return *curr_total* to use it in the conductance matrix.

```
for (int n = 0; n <= m; n++){
    curr_total[n] = curr_in[n] + curr_out[n];
}

return curr_total;
```

Figure 37. Inputting current values into curr_total

Finding DC Operating Points

We realised that the small-signal equivalent's values did not depend on the circuit itself. To solve this we used the Newton Raphson Method to calculate the DC operating points. The Newton Raphson Method is used to approximate the root of a function [21]. An initial guess is made, then the root gets more and more refined.

In this case, the diode operating point can be found by using:

$$i_d = i_s \times (e^{\frac{v_d}{n \times v_t}} - 1)$$

To find v_d with the Newton Raphson Method, i_d is differentiated with respect to v_d . The equation then becomes:

$$i_d' = i_s \times \left(\left(\frac{1}{n \times v_t} \times e^{\frac{v_d}{n \times v_t}}\right) - 1\right)$$

This process is looped within the program until v_d are very similar (2 decimal places).

```
void newton (double& vd, std::vector<resistor>& res){
    resistor r;
    std::vector<std::string> netListr;
    double is = 2.52 * pow(10,-9);
    double vt = 0.025;
    double n = 1.752;
    double id;
    double idd;
    double rd;

    id = is * (exp(vd / (n * vt)) - 1);
    idd = is * ((1/(n*vt)) * (exp(vd / (n * vt))) - 1);
    vd = vd - (id/idd);

    id = is * (exp(vd / (n * vt)) - 1);
    rd = vt / id;

    addNetListResistor(netListr, ("N00" + std::to_string(n0)), ("N00" + std::to_string(n1)), rd);
    res.push_back(r.add(netListr));

    netListr.clear();

}
```

Figure 38. Newton Raphson function for diode

The *newton* function in Fig. 38 finds the approximation of v_d using the above equations. v_d is used to find i_d which is also used to find r_d. r_d then contributes the conductance matrix in the main. With the new addition of the r_d, the voltages at each node will be altered. By taking the voltage at the anode and the cathode of the diode, v_d will be adjusted and it is passed back into the *newton* function. This process repeats until the new v_d is very similar to its previous iteration. This value of v_d is the DC operating point of the diode.

Finding the DC operating point of the BJT is also similar, however, there are 2 variables v_b and v_c.

```
void newtonBJT(double& vbe, double& vce, std::vector<resistor>& r, std::vector<controlSource>& g){
    double is = pow(10,-14);
    double vt = 0.025;
    double va = 100;
    double ic = pow(1,-3);
    double ib;
    double beta = 200;
    std::vector<std::string> netListr;
    std::vector<std::string> netListg;
    resistor res;
    controlSource cs;

    ic = is * exp(vbe/vt) * (1 + vce/va);
    ib = ic / beta;

    double rbe;
    double gm = ic / vbe;
    double ro;

    rbe = beta / gm;
    ro = va / ic;

    std::string node_c = "N00" + std::to_string(n0);
    std::string node_b = "N00" + std::to_string(n1);
    std::string node_e = "N00" + std::to_string(n2);

    //for rbe
    addNetListResistor(netListr, node_b, node_e, rbe);
    r.push_back(res.add(netListr));

    netListr.clear();

    //for ro
    addNetListResistor(netListr, node_c, node_e, ro);
    r.push_back(res.add(netListr));

    netListr.clear();

    //for current source
    addNetListCS(netListg, node_c, node_e, node_b, node_e, gm);
    g.push_back(cs.add(netListg));

    netListg.clear();

}
```

Figure 39. Newton Raphson Function for BJT without derivatives

This function in Fig. 39 is similar to the diode *newton* function, where it is called similarly in the main and it also adds the newly calculated components into the conductance matrix. Due to time limitations, it was difficult for us to research newton raphson for 2 variables, therefore, instead of finding the derivative of the function, the new v_b and v_c values are calculated from the conductance matrix. With new values of v_b and v_c in each increment, it will change r_o, r_{be} and g_m which changes the conductance matrix.

Technical problems

In this project, we had to combine our knowledge from two modules - programming and ADC and integrate them into our circuit simulator, which was a challenging start for all of us since research was required for both. There were several problems that we encountered along the way.

Parse the netlist file

Choose a suitable data structure

Before coming up with code that performs a small-signal AC analysis simulation of a circuit. Firstly, we had to find a way to parse the SPICE netlist that is inputted and its simulation command, then store each component (which are split into *designator* denoting the component, name of a *node,* parameter *values* etc.) into a data structure allowing us to extract the data to use in the code later.

At first, we implemented a new type using structured data types and stored each element (as a vector) into *struct*.

Below is the first two *struct*s we drafted:

- Structure for each component
    - comp (string)
    - n0 (int)
    - n1 (int)
    - n2 (int)
    - n3 (int)
    - value (double)
    - model (string)
    - amplitude
    - phase (NULL if not used)
- Structure for impedance
    - complex
    - real

However, after analysing the technical differences between structure and class, we had concluded unanimously to change struct to class.

We had made a comparison table below [22]:

Structure	Class
Used to build a simple program	Used to build a complex program
Default public	Default private
Uses for grouping data	Provides flexibility to combine data and functions
Pass-by-reference (reference type) Object is created on heap memory	Pass-by-copy (value type) Object is created on stack memory
Can't inherit other classes	Support inheritance

Not only did we research the comparison, but we also discussed this problem with the professor and TAs. They suggested that using Class is better because a Class can be derived from more than one class, which means it can inherit data and functions from multiple base classes [22]. In most cases, the objects we use are long-living and would require a long memory footprint, which means we need them throughout the whole program. Therefore, with classes, we had a lot more flexibility in handling objects.

Read the file and split the vector of strings

The next technical problem we encountered during the extraction of data from the netlist was that we couldn't store the input variables in a structured way. Since there were a different number of elements for each component and the number of spaces in each line would mess up the algorithm, when we stored the file into a vector using purely *input.open()*. Charmaine proposed to use the *getline()* function to read lines from the netlist and store them in a correct vector space according to their designators. The *getline()* function extracts characters from the input file and appends it to the string object until the delimiting character is encountered [23].

```
std::ifstream file ("input.txt");
std::string str;

while (std::getline(file, str))
{
   string.push_back(str);
}
```

Figure 40. Splitting input strings

After extracting each line, we needed to decode the netlist, which is based upon the first letter to determine the format of the line.

```
The format of each line is:
<designator> <node0> <node1> [<node 2> [<node 3>]] <value>
```

Figure 41. Netlist Format

However, there is an exception to the circuit description:
Independent voltage and current sources can have a function. The only function required is an AC source written in the form:

AC <amplitude(Volts or Amps**)> <phase(**Degrees**)>**

In the sample netlist, it is computed as:

```
V1 N003 0 AC(1 0)
```

When we decoded the netlist, we separated the string from the vector according to the spaces in the string. Here came the problem: there was extra space in the AC source. How could we combine the whole AC source as one element and store it into an appropriate vector? Before using *getline(),* this part was very tricky. However, after figuring out the most suitable way to store the input variables, we could use an if else statement to specify this special situation. We had introduced a new data type called std::polar into the program. std::polar(r, theta) returns a complex number with magnitude r and phase angle theta. Without a doubt, the behaviour is undefined if r is negative, or if theta is infinite. As a result, it returns a complex number determined by r and theta [24].

Taking information from the simulation command

The simulation commands are slightly different to the circuit description lines. They all start with a dot. Since the specification requires us to write a software package that performs a small-signal AC analysis simulation, we have only focused on reading the .ac command from the netlist.

```
The AC analysis simulation is described with a line of the form:
.ac dec <points per decade> <start frequency> <stop frequency>
```

Figure 42. AC analysis simulation command

Using nodal analysis to find the voltage at each node

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

Figure 43. Equation for finding a vector of unknown node voltages from a conductance matrix

Conductance matrix formation

The first issue we found was the function finding the maximum number of nodes. When Kelvin constructed the conduction matrix, he only took into account the nodes that were connected to resistors and not all the nodes that were connected to different components. Due to this, the conductance matrix turned out to be smaller than expected, leading to many problems. We developed a function that automatically stores all nodes into a vector so we could easily input the highest number of nodes into other functions later on.

Wrong version:

```
//Store each nodes into an int vector
std::vector<int> store_noder (std::vector<resistor> node){
    std::vector<int> nodesize ;
    for (int i = 0; i < node.size(); i++){
        nodesize.push_back(node[i].n0) ;
        nodesize.push_back(node[i].n1) ;
    }

    return nodesize ;
}
```

Figure 44. Wrong function *store_noder*

Amended version:

```
void store_node (const std::vector<std::string> netList, std::vector<int>& nodesize){
    if (netList[0][0] != 'Q' || netList[0][0] != 'G' || netList[0][0] != 'M'){
        nodesize.push_back(node(netList[1]));
        nodesize.push_back(node(netList[2]));
    }
    else if (netList[0][0] == 'Q' || netList[0][0] == 'G'){
        nodesize.push_back(node(netList[1]));
        nodesize.push_back(node(netList[2]));
        nodesize.push_back(node(netList[3]));
    }
    else{
        nodesize.push_back(node(netList[1]));
        nodesize.push_back(node(netList[2]));
        nodesize.push_back(node(netList[3]));
        nodesize.push_back(node(netList[4]));
    }
}
```

Figure 45. Amended *store_node*

Then the "conductance matrix" term confused us. The conductance of an electrical element refers to *G=1/R*; where *R* equals resistance and *G* equals conduction [25]. Hence, we misunderstood the conductance matrix at first and only considered the resistors in the netlist. After consulting the TAs, we discovered that the impedance of capacitors and inductors should also be considered. Therefore, the type std::complex<double> was used instead of <double> to achieve functionalities that involve impedance.

While debugging the program, another issue arose. The "conduct_mat" function worked correctly in repl.it. However, there was a segmentation fault in other compilers such as VSCode and Xcode. The problem appeared to be that some index inside the vector did not have any values assigned. We believed that repl.it helped us autofill in the value of 0 for the unassigned index in the vector.

```
// function to store the conductance matrix in a vector
void conduc_mat (const std::vector<int> &nodes , const std::vector<std::vector<std::complex<double>>> &rnode , std::vector<std::complex<double> > &final ){
  for (int i = 1; i <= max_node_size(nodes);i++){
    for (int j = 1; j <= max_node_size(nodes); j++){

    std::complex<double>  cond = 0 ;

    std::vector<std::complex<double> >  cond_mat ;

    intersection_vector(rnode[i-1] , rnode[j-1] , cond_mat);

    for (int v = 0 ; v < cond_mat.size() ; v++){
      cond = cond + cond_mat[v] ;
    }
    if (i == j){
      final.push_back(cond) ;
    }
    else{
      final.push_back(-cond) ;
    }
    cond_mat.clear() ;
    }
  }
}
```

Figure 46. Function *conduc_mat*

Kelvin resolved this by pushing zeros into the unassigned indexes of the vector to prevent segmentation faults.

```
// a function that store the resistor values in a vector when the resistor connected to the node n
std::vector<std::complex<double> > rcl_in_vec (int n , std::vector<int> node_list ,std::vector<std::complex<double> > overall_imp , std::vector<int> node){
  std::vector<std::complex<double> > Node ;
  for (int i = 0 ; i < node_list.size() ; i++){
  if (node_list[i] == n){
      Node.push_back(overall_imp[i/2]) ;
    }
  if ( n >= max_node_size(node) && (max_node_size(node_list)<max_node_size(node))){
   Node.push_back(0) ;
  }
  }
  int x = (int) Node.size() ;
  if (Node.size() < max_node_size(node)){
    for (int i = 0 ; i < (max_node_size(node)- x) ; i++){
     Node.push_back(0) ;
    }
  }
return Node ;
}
```

Figure 47. Function *rcl_in_vec*

Creating current vector

Nodal analysis can be performed by writing an equation for each node that satisfies Kirchoff's current law and pushing them all back into a conductance matrix to solve the simultaneous equations. While Kelvin was working on the formation of the matrix, Amy was responsible to find out the current going in and out of the node depending on the order of the nodes. While figuring out the algorithm might be quick, we had spent one of our meetings debugging the combined code because we had each used different sets of classes and different declared variables. During our debug session, we found something interesting yet slightly discouraging. The total current going in and out of Node 0 has never come out as the right value. We suspected that it was related to the presentation of Node 0 in the netlist as it is presented as `0` instead of `Node 0.` However, we came to a consensus to stop figuring out the solution because the total current through Node 0 won't be included in the matrix and node 0 is the ground node, therefore it won't matter.

The other technical problem was the total current never increments when there was more than one current source supplying current to a node or leaving the node. At last, to make it incrementing, Charmaine changed the = sign into +=, and the whole function worked perfectly fine afterwards.

```
for (int j = 0; j <= m; j++){
    if (j < curr.size()){
        curr_in[curr[j].n0] += curr[j].dcVal;
        curr_out[curr[j].n1] += -curr[j].dcVal;
    }
```

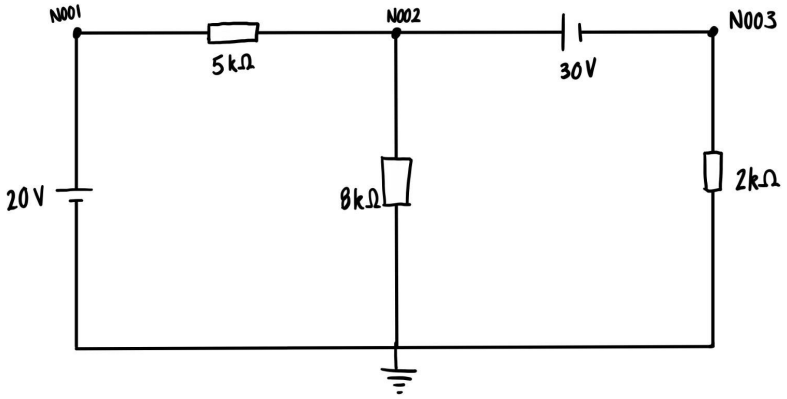
Figure 48. Inputting current values into curr_in and curr_out

Testing

Accuracy

Testing a basic circuit

To test whether the nodal analysis was working correctly with the conductance matrix method. We tried a very simple circuit that could be solved by hand as well to compare answers. Fig. 49 shows the calculations done by hand.



$$V @ N001 = 20\,V$$

$$V @ N002 = \frac{V_2 - 20}{5} + \frac{V_2}{8} + \frac{V_2 - 30}{2} = 0$$

$$\frac{8V_2 - 160 + 5V_2 + 20V_2 - 600}{40} = 0$$

$$33V_2 = 760$$

$$V_2 = 23.03\,V$$

$$V @ N003 = V_2 - V_3 = 30$$

$$V_3 = -6.97\,V$$

Figure 49. Calculations for the sample circuit

This is the circuit that we drew to test the conductance matrix. We then converted it into SPICE format (Fig. 50) according to the nodes in the drawing.

```
string.push_back("V1 N002 N003 30");
string.push_back("V2 N001 0 20");
string.push_back("R1 N001 N002 5k");
string.push_back("R2 N003 0 2k");
string.push_back("R3 N002 0 8k");
string.push_back(".ac dec 10 10 100k");
string.push_back(".end");
```

Figure 50. Netlist for sample circuit

Since there are no capacitors or inductors in this test circuit, the command line will not affect the transfer function. The transfer function would be a constant line.

Fig. 51 shows the voltage at node 1 until node 3, then the bottom 2 values are the current through the 2 voltage sources, however, they are not needed in this case.

```
          (20,0)
     (23.0303,0)
      (-6.9697,0)
(-0.00348485,0)
(0.000606061,0)
```

Figure 51. Output matrix

These values are the same as the ones calculated in Fig. 49. We can also easily confirm that nodes 2 & 3 are correct, by finding its difference to confirm that it is equal to the 5V voltage source situated between the 2 nodes. There is no imaginary part that is correct as there is no complex impedance without the capacitors or inductors.

Testing a linear circuit

All of the circuit parameters in a linear circuit are constant. In other words, a circuit whose parameters are not changed with respect to current and voltage is called **Linear Circuit** [26].

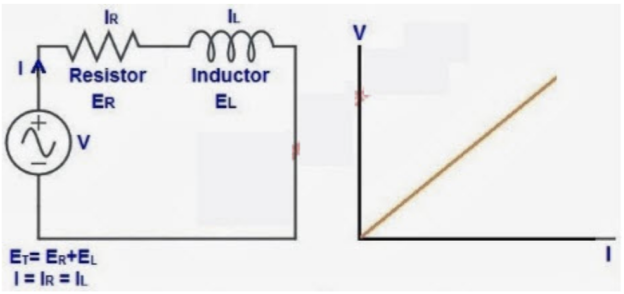
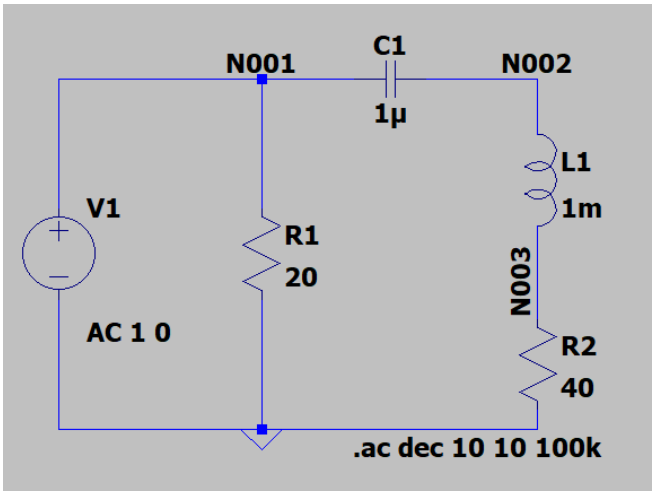


Figure 52. Linear Circuit

straight line I-V graph:
the current flow is directly proportional to the voltage.
parameters: resistance, inductance, capacitance, waveform, frequency etc.
(Fig. 52 is from [26])

To test a linear circuit, we drew a circuit in the LTSpice containing resistors, a capacitor and an inductor.




```
V1 N001 0 AC(1 0)
R1 N001 0 20
C1 N002 N001 1u
L1 N002 N003 1m
R2 N003 0 40
.ac dec 10 10 100k
.end
```

Figure 53. Sample linear circuit

We used cursors to find out the transfer function of V(N002) and V(N003), then compared our output in our circuit simulator:

LtSpice	Circuit Simulator
Cursor 1 V(n002) Freq: 10Hz Mag: 2.513342mV Phase: 89.945994° Group Delay: 15.001893µs Cursor 2 V(n003) Freq: 10Hz Mag: 2.5132761mV Phase: 89.855996° Group Delay: 40.00121µs	10 (1,0) (2.36872e-06,0.00251328) (6.31656e-06,0.00251327) (-0.0500002,-6.28317e-05) To make it easier to compare, we have converted radians to degrees using a calculator. V(N001): (1, 0) V(N002): (2.3687 x 10^{-6}, 89.856000) V(N003): (6.3166 x 10^{-6}, 89.856000)

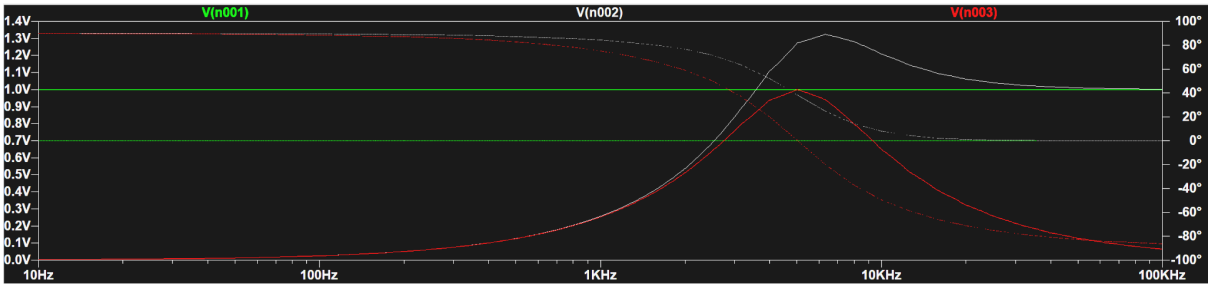


Figure 54. Bode plot of a linear circuit from LTSpice

Since the actual magnitude is almost zero at the beginning, the difference between LTSpice and our circuit simulator doesn't matter. However, we would say the phases are very similar. The variation might come from the different assumption values and equations we used.

The last row in the output is the current of the voltage source, but we have ignored it throughout the testing since our focus is on the transfer function of the nodes.

To make sure everything else is on the right track, we decided to take another value at the frequency of around **6300Hz**.

LtSpice	Circuit Simulator
Cursor 1 V(n002) Freq: 6.3120395KHz Mag: 1.3244236V Phase: 24.904065° Group Delay: 24.105162µs Cursor 2 V(n003) Freq: 6.3120395KHz Mag: 940.50706mV Phase: -19.849153° Group Delay: 36.819299µs	6309.57 (1,0) (1.20119,0.558082) (0.884989,-0.319035) (-0.0721247,0.00797588) To make it easier to compare, we have converted radians to degrees using a calculator. V(N001): (1, 0) V(N002): (1.20119, 31.97574322) V(N003): (0.884989, -18.27935902)

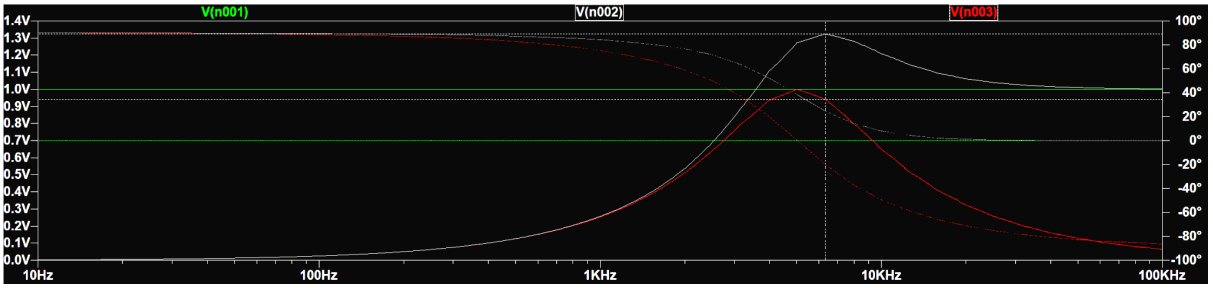
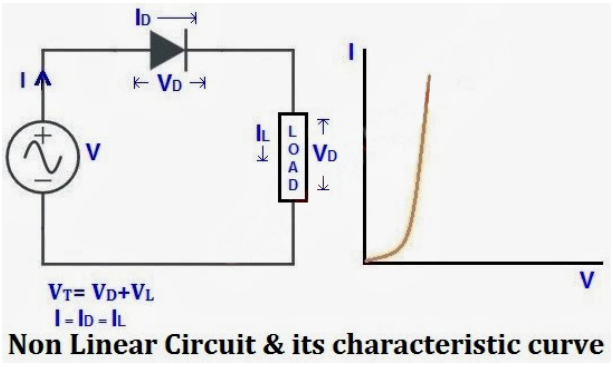


Figure 55. Bode plot with cursors from LTSpice

For the magnitude, we had a +/- 0.1V error, and for the phases, we had got a much larger error, possibly because we were testing the peak area.

Testing a nonlinear circuit



A **non-linear** circuit is the exact opposite of a linear circuit.

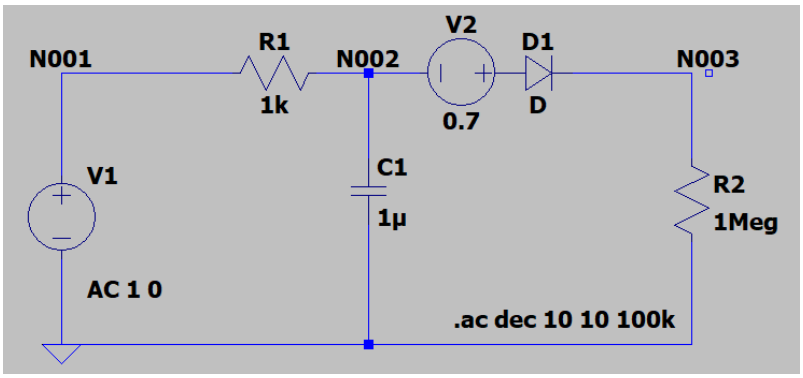
curve line I-V graph:

the current flow is not directly proportional to the voltage.

parameters: Diode etc., in which the current is an exponential function of the voltage. Another example is a transistor.

Figure 56. Nonlinear Circuit

To test a non-linear circuit, we drew a circuit in the LTSPICE containing a diode which is defined as a non-linear component.




```
V1 N001 0 AC(1 0)
R1 N002 N001 1k
C1 N002 0 1u
R2 N003 0 1Meg
D1 N002 N003 D
.ac dec 10 10 100k
.end
```

Figure 57. Sample nonlinear circuit

I used cursors to find out the transfer function of V(N002) and V(N003), then compared our output in our circuit simulator:

LTSpice	Circuit Simulator
Cursor 1 V(n002) Freq: 10Hz Mag: 997.12909mV Phase: -3.5920173° Group Delay: 995.36211µs Cursor 2 V(n003) Freq: 10Hz Mag: 906.37218mV Phase: -3.5920173° Group Delay: 995.36211µs	10 (1,0) (0.99508,-0.0624603) (0.99508,-0.0624603) (-4.91958e-06,-6.24603e-05) To make it easier to compare, we have converted radians to degrees: V(N001): (1, 0) V(N002): (0.99508, -3.578711577) V(N003): (0.99508, -3.578711577)

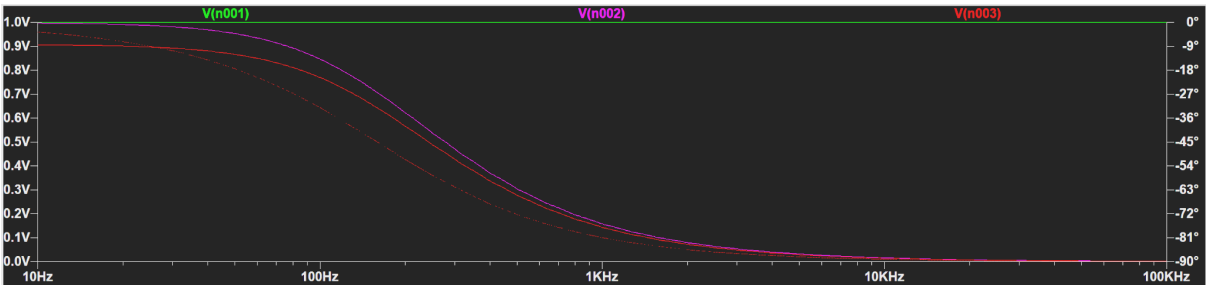


Figure 58. Bode plot of a non-linear circuit from LTSpice

V(N003) has a slightly smaller magnitude in LTSpice than in the circuit simulator, possibly because we have assumed the diode is fully biased, so V(N002) and V(N003) would be the same in our circuit simulator, but slightly different in the LTSpice since, in reality, a diode is not fully biased at the voltage of 0.7V.

To make sure everything else is on the right track, we decided to take another value at the frequency of around **400Hz**.

LTSpice	Circuit Simulator
Cursor 1 V(n002) Freq: 398.41646Hz Mag: 370.96175mV Phase: -68.20408° Group Delay: 142.81053µs Cursor 2 V(n003) Freq: 398.41646Hz Mag: 337.19747mV Phase: -68.20408° Group Delay: 142.81053µs	398.107 (1,0) (0.1379,-0.344595) (0.1379,-0.344595) (-0.0008621,-0.000344595) To make it easier to compare, we have converted radians to degrees: V(N001): (1, 0) V(N002): (0.1379, -70.25616) V(N003): (0.1379, -70.25616)

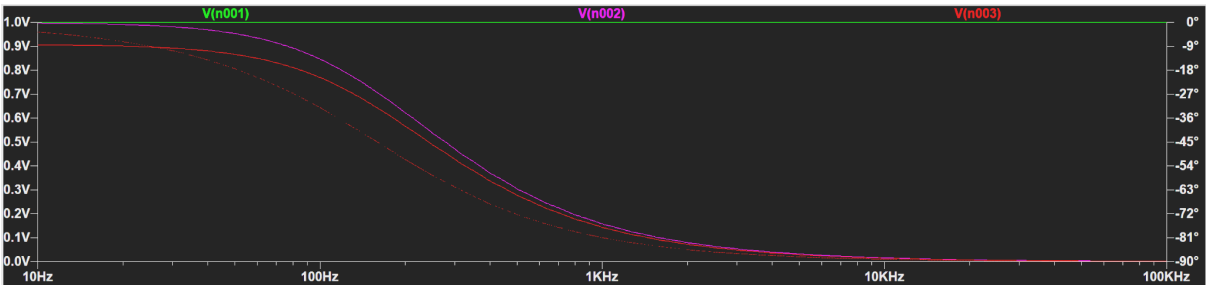


Figure 59. Bode plot from LTSpice

The circuit simulator has followed the general trend of the bode plot. There was variation between the results, we assumed it was due to the slight difference in the frequency values we took, and the equations we used.

Testing the Newton Raphson

To test the Newton Raphson function we drew a circuit in LTSpice containing a diode and looked at the transfer function plotted out. When we assumed the DC operating point of the diode, we assumed that it was 0.7V. Therefore, we tested it with a 0.7V voltage source before the diode. If N003 and N002 are equal, this means that the diode is correctly biased (as there is a voltage source before it) and the value of the DC operating point from the Newton Raphson function is correct.

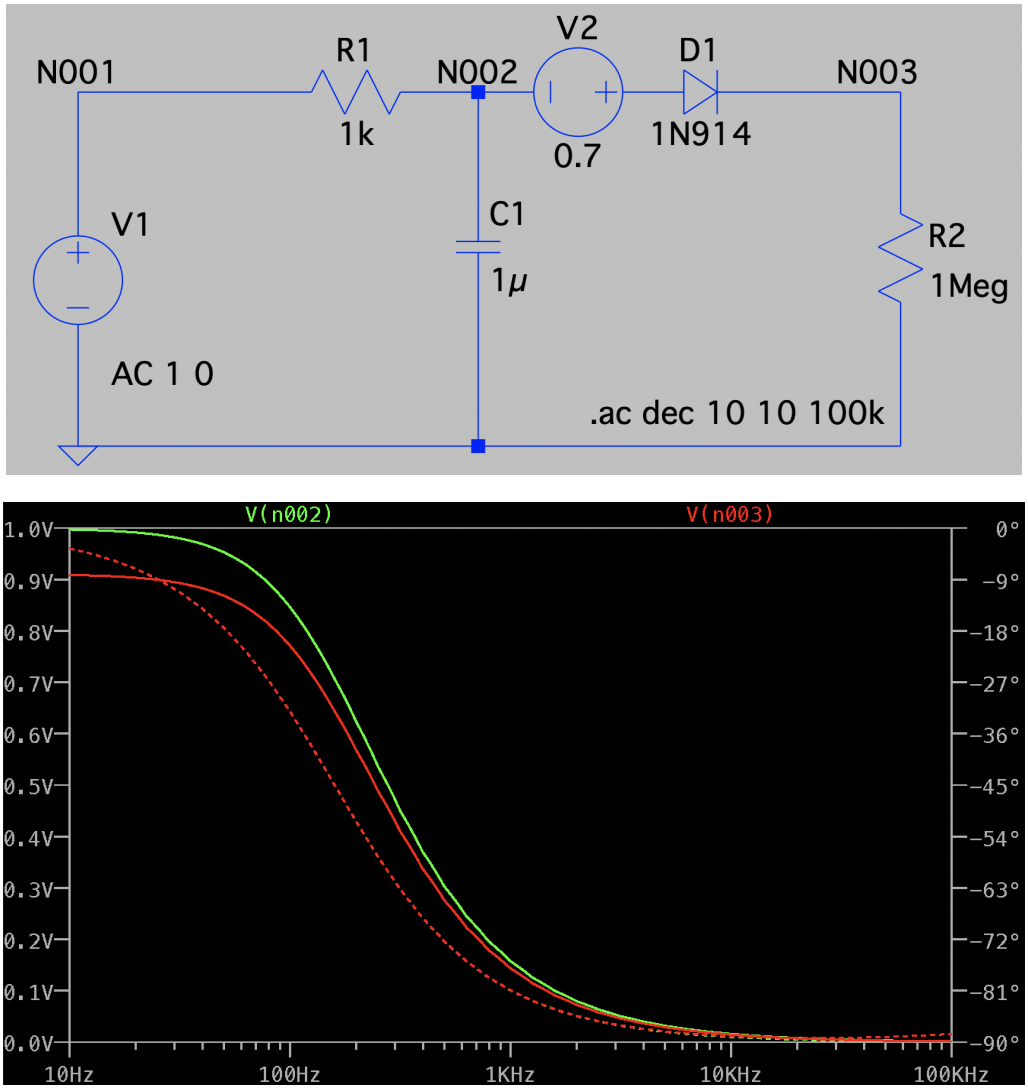


Figure 60. LTSpice circuit diagram and frequency response

As we can see from Fig. 60 at 0.7V, the diode is not entirely forward biased, meaning that voltage at N002 is not equal to N003. Meaning there is a 0.1V difference between the 2

curves. From our Newton Raphson function, we found that the DC operating point of the diode to be around 0.999V, this value was found using the do-while loop in the main and the *newton* member function in class “diode”. Since the real DC operating point is greater than 0.7, therefore, we changed the value of the voltage source and Fig. 61 was the result.

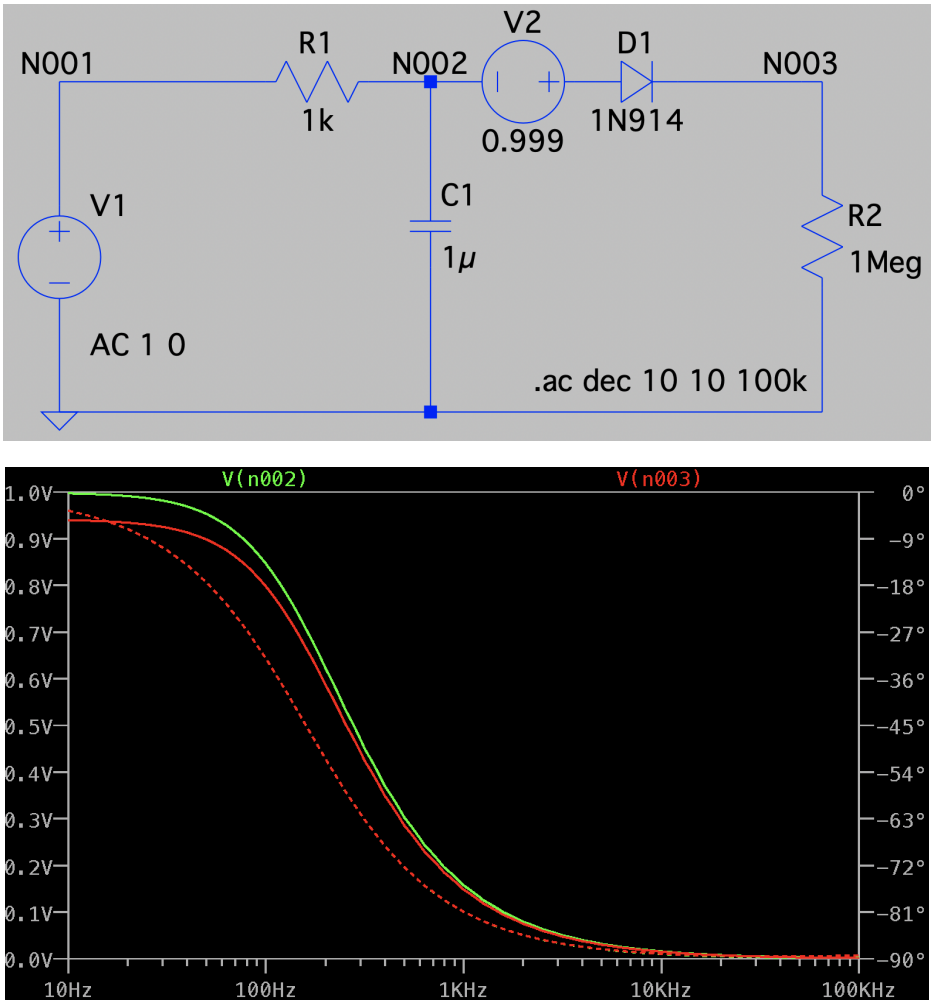


Figure 61. LTSpice circuit diagram and frequency response with 0.999V source

As we can see from the result, changing the voltage source to 0.999V has improved the curve in terms of moving the transfer function of N003 towards N002. However, it is still not entirely equal, this means that there is a tolerance in our Newton Rapheson function. This error could arise from the rounding of v_d in the do-while loop, since it was rounded to the 2nd decimal place, the accuracy of v_d may be compromised, however, there would be a tradeoff between processing time and accuracy. During testing, it required much longer to arrive at a suitable number when rounding with more decimal places, this was why we decided rounding to 2 decimal places would be sufficient. Another reason could be due to the diode model in LTSpice not being ideal, therefore, the voltage at each node may differ. However,

the Shockley Equation models an idealised diode [20], therefore, varying values may also arise with our use of equations.

Fig. 62 is our output for the first frequency increment since we used the equation of an ideal diode, therefore, the voltages at N002 and N003 are the same.

If we calculated the amplitude and the phase, we would get 0.997V for the amplitude and -3.592º for the phase. Compared to LTSpice, there is little variation in the amplitude but a small difference in phase. This may be due to the equations that we have used in our program or issues with round errors.

```
                    (1,0)
          (0.99508,-0.0624603)
          (0.99508,-0.0624603)
(-4.91958e-06,-6.24603e-05)
```

Figure 62. Output from simulator

Efficiency

We decided that several most important points would be the processing time and its memory used to test the efficiency. Through some research, we found ways to measure the processing time within the code. We found out that by using the std::chrono library, to time the start and end of the program. There were several clocks available in the library such as *system_clock, steady_clock* and *high_resolution_clock*. We decided to use *high_resolution_clock,* because our program doesn't require long periods to compile and run, using *high_resolution_clock* would be the most accurate measurement [27]. Fig. 54 was added to the start and end of the code:

```
auto start = std::chrono::high_resolution_clock::now();

auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double> elapsed = end - start;
```

Figure 63. Initialising the clock

"start" records the time where the program starts and "end" records the time where the program ends. "elapsed" shows the time difference between the start and end of the program.

Several factors were considered when measuring the time taken, not only did we have to look at the number of nodes, but we also had to take into account the number of components and the different type of components (eg. resistors should take less time than inductors or capacitors as we need to calculate the impedance of the capacitors and inductors using frequency as well). Therefore, to reduce the number of changing factors, we decided to only time the program when using resistors of 1kΩ in series with a voltage source. This ensured that variables that could affect the measurements remained constant. We also believe that we could extrapolate our results for other components and these measurements will be able to give the general trend for our program.

Every time we increased the number of nodes, we also increased the number of resistors by the same amount. Fig. 64 shows our results.

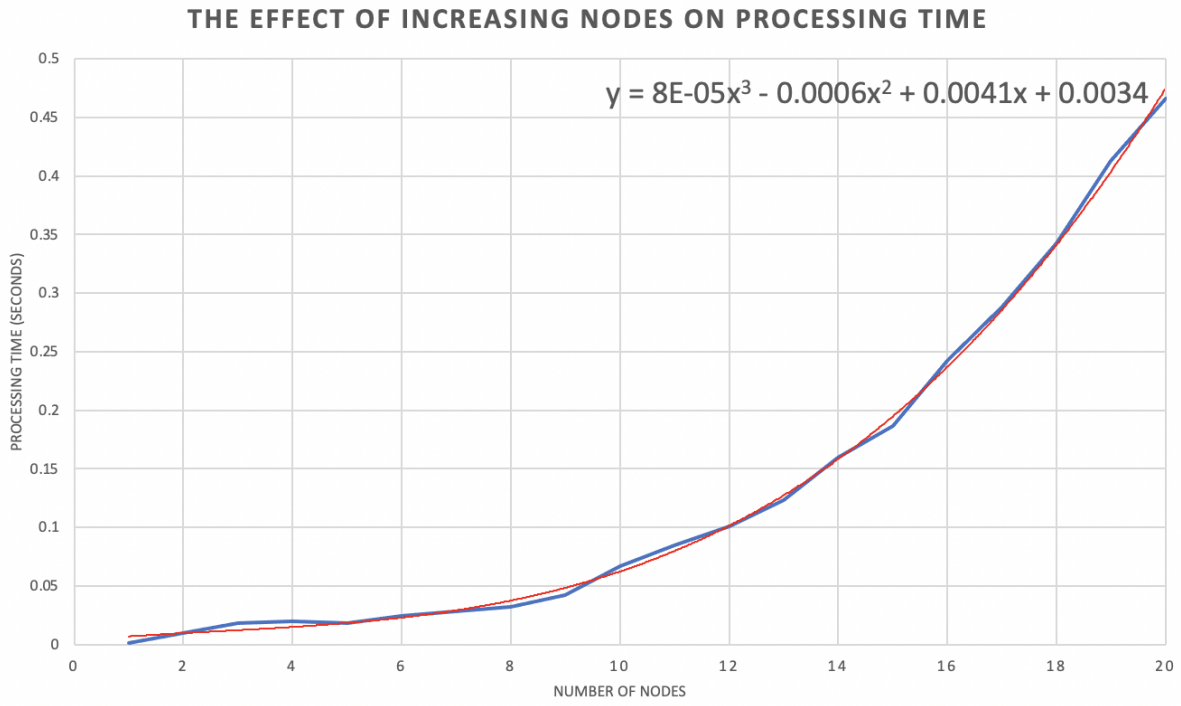


Figure 64. Graph showing how the processing time varies with changing the number of nodes

The blue line represents the data that we measured and the red line is the trendline of the curve. As we can see, as the number of nodes increases, so does the processing time. This is due to the larger conductance matrix due to the increase in nodes as well as the longer processing time when calculating the conductances in the conductance matrix. From the trendline we were also able to extrapolate and approximate the time needed to process a netlist with 100 nodes, we found that it was about 74 seconds. If time permits, we would like

to investigate how this changes with different components, whether they do follow this general trend and how the processing time varies with different components.

Next, we tried to test how much memory this program required. The software that we used for this project allowed us to find out its memory consumption [28].

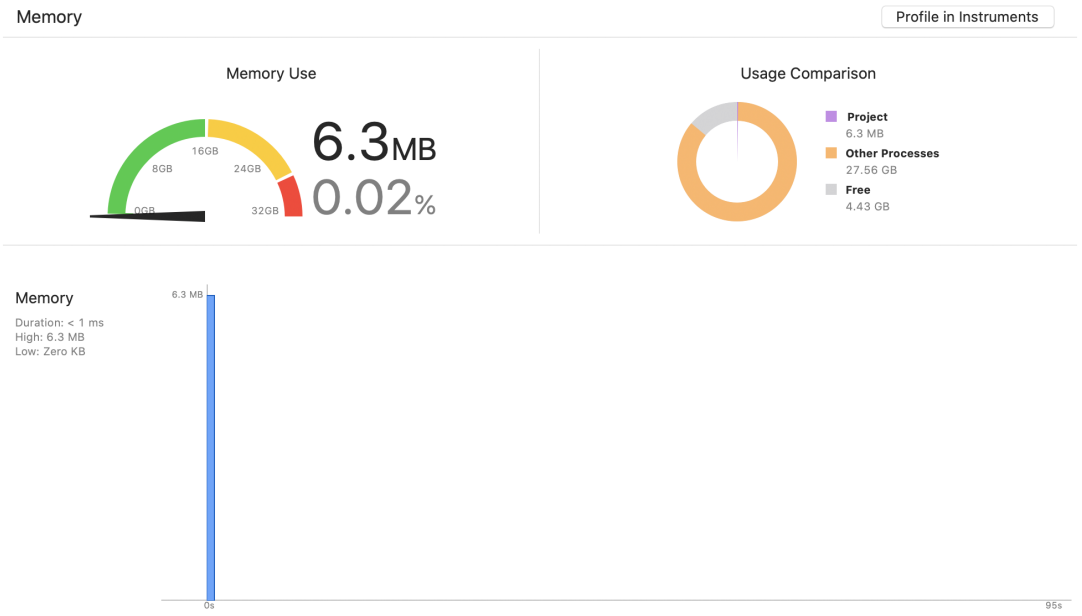


Figure 65. Memory Consumption Analysis

Fig. 65 shows that the program does not take a significant amount of memory, this may be due to the short processing time, which means that its memory consumption is also reduced. From research, we found out that the software has an indication system, when the arrow is in the yellow region it may trigger a warning, if it is in the red region, there is a risk of termination [28]. We believe that it required less memory as we tried to minimise the number of functions needed to accomplish our goal.

Conclusion & Evaluation

What we did

We have successfully designed a circuit simulator that reads an input SPICE netlist file defining the circuit, performs a small-signal analysis simulation and creates a CSV file that outputs the transfer function.

Our simulator can process a circuit with the following components:

- Independent voltage source (V)
- Current source (I)
- Resistor (R)
- Capacitor (C)
- Inductor (L)
- Diode (D)
- BJT (Q)

What went well

- We have researched the perfect formulae for computing a conductance matrix which worked amazingly within our simulator
- Although we haven't found a platform to collaborate our code, with sufficient communication, we managed to share our program file and used each other's function during implementation
- We were able to use Newton Raphson to find the small-signal equivalent of a diode

What didn't go well

- We didn't figure out the way to plot our results using MATLAB
- (Advanced) Due to the time limit, we didn't compute Newton Raphson to find the DC Operating Point of the BJT
- During the middle of our refinement, we were unable to solve the issue of converting the non-linear components into small signal equivalents

How we solved our problems

- Our team worked well together by arranging regular meetings and bouncing ideas around. When we couldn't make up our mind, we immediately proposed our problems and everyone would look into them simultaneously
- Since each consultation session was quite short, we treasured every session and prepared our questions in advance, so that our project could run smoothly

Accomplishments that we are proud of

- We have managed to build the conductance matrix without a clear step-by-step interpretation in the specification
- We have also managed to code a function that finds the DC Operating Point of a diode using Newton Raphson

What will we do next?

- Due to time limitations, we were unable to fully complete the MOSFET SSEM function, however, they would follow the same logic as the BJT and diode but include a different set of equations
- We would like to extend this program to other types of AC analysis (eg. octave or linear)
- Instead of looping the BJT equations, we would like to try to perform Newton Raphson on its equations

Bibliography

[1] "The Nine Belbin Team Roles," *Belbin*. [Online]. Available: https://www.belbin.com/about/belbin-team-roles. [Accessed: 05-Jun-2021].

[2] the Mind Tools Content Team By the Mind Tools Content Team, "Forming, Storming, Norming, and Performing: Tuckman's Model for Nurturing a Team to High Performance," From MindTools.com. [Online]. Available: https://www.mindtools.com/pages/article/newLDR_86.htm. [Accessed: 04-Jun-2021].

[3] user1444426, ecatmur, Andrew, and Rafael Baptista, "What causes a SIGABRT fault?," *Stack Overflow*, 24-Jun-2013. [Online]. Available: https://stackoverflow.com/questions/11161126/what-causes-a-sigabrt-fault. [Accessed: 04-Jun-2021].

[4] "C++ Structs - javatpoint," *www.javatpoint.com*. [Online]. Available: https://www.javatpoint.com/cpp-structs. [Accessed: 06-Jun-2021].

[5] GeeksforGeeks, "C++ Classes and Objects," *GeeksforGeeks*, 17-May-2021. [Online]. Available: https://www.geeksforgeeks.org/c-classes-and-objects/#:~:text=Class%3A%20A%20class%20in%20C%2B%2B,a%20blueprint%20for%20an%20object. [Accessed: 06-Jun-2021].

[6] P. Richard, *C++ Program to Find Inverse of a Graph Matrix*, 26-Apr-2019. [Online]. Available: https://www.tutorialspoint.com/cplusplus-program-to-find-inverse-of-a-graph-matrix#:~:text=Begin%20function%20INV()%20to,%2F%20DET(matrix)%20End. [Accessed: 05-Jun-2021].

[7] "C++ Standard Library headers," *cppreference.com*, 02-Apr-2021. [Online]. Available: https://en.cppreference.com/w/cpp/header. [Accessed: 05-Jun-2021].

[8] B. Jacob, "Main Page," *Eigen*, 19-Apr-2021. [Online]. Available: https://eigen.tuxfamily.org/index.php?title=Main_Page#Overview. [Accessed: 05-Jun-2021].

[9] “C++ library for linear algebra & scientific computing,” *Armadillo*. [Online]. Available: http://arma.sourceforge.net/. [Accessed: 05-Jun-2021].

[10] “IMSL Numerical Libraries,” *IMSL Numerical Libraries | Battle-Tested Functions and Algorithms*. [Online]. Available: https://www.imsl.com/. [Accessed: 05-Jun-2021].

[11] “What are Vectors in C++ ? All You Need to Know,” *Edureka*, 18-Jul-2020. [Online]. Available: https://www.edureka.co/blog/vectors-in-cpp/. [Accessed: 06-Jun-2021].

[12] “Linked List Data Structure In C++ With Illustration,” *Software Testing Help*, 30-May-2021. [Online]. Available: https://www.softwaretestinghelp.com/linked-list/. [Accessed: 06-Jun-2021].

[13] JoniJoni , “Linked List vs Vector,” *Stack Overflow*, 01-May-1962. [Online]. Available: https://stackoverflow.com/questions/19039972/linked-list-vs-vector. [Accessed: 06-Jun-2021].

[14] “Input/output with files,” *cplusplus.com*. [Online]. Available: https://www.cplusplus.com/doc/tutorial/files/. [Accessed: 06-Jun-2021].

[15] “std::getline (string),” *cplusplus.com*. [Online]. Available: https://www.cplusplus.com/reference/string/string/getline/. [Accessed: 06-Jun-2021].

[16] E. Cheever, *An Algorithm for Modified Nodal Analysis*. [Online]. Available: https://www.swarthmore.edu/NatSci/echeeve1/Ref/mna/MNA3.html. [Accessed: 06-Jun-2021].

[17] S. C. Erik Cheever, “Advanced SCAM with dependent sources,” *Linear Physical Systems - Erik Cheever*. [Online]. Available: https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNADep.html. [Accessed: 06-Jun-2021].

[18] “Nodal Analysis: Node Equations by Inspection Method”. *Eeeonline.* 03-Mar-2020. [Online]. Available: https://www.eeeonline.org/nodal-analysis/. [Accessed: 08-Jun-2021]

[19] "LTspice," *LTspice Simulator | Analog Devices*. [Online]. Available: https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html. [Accessed: 07-Jun-2021].

[20] Libretexts, "3: Ideal Diode Equation," *Engineering LibreTexts*, 21-Oct-2020. [Online]. Available: https://eng.libretexts.org/Bookshelves/Materials_Science/Supplemental_Modules_(Materials_Science)/Solar_Basics/D._P-N_Junction_Diodes/3%3A_Ideal_Diode_Equation. [Accessed: 09-Jun-2021].

[21] "Newton Raphson Method," *Brilliant Math & Science Wiki*. [Online]. Available: https://brilliant.org/wiki/newton-raphson-method/#:~:text=The%20Newton%2DRaphson%20method%20(also,straight%20line%20tangent%20to%20it. [Accessed: 08-Jun-2021].

[22] Parikshit Ankit Mohari, "Why should we use Class instead of structure?". *Quora*. [Online] Available: https:// www.quora.com/Why-should-we-use-Class-instead-of-structure [Accessed: 06-Jun-2021].

[23] "getline (string) in C++". *GeeksforGeeks*. 05-Mar-2021. [Online]. Available: https://www.geeksforgeeks.org/getline-string-c/. [Accessed: 06-Jun-2021].

[24] "std::polar(std::complex)". *cppreference.com.* [Online]. Available: https://en.cppreference.com/w/cpp/numeric/complex/polar. [Accessed: 07-Jun-2021]

[25] M. Williams, "What is Conductance?," *Universe Today*, 04-Nov-2016. [Online]. Available: https://www.universetoday.com/82339/conductance/. [Accessed: 07-Jun-2021].

[26] "What are linear and non-linear circuits and It's Difference*". ElProCus.* 25-Jan-2017. [Online]. Available: https://www.elprocus.com/linear-and-non-linear-circuit-with-differences/. [Accessed: 09-Jun-2021]

[27] "Date and time utilities," *cppreference.com*. [Online]. Available:

https://en.cppreference.com/w/cpp/chrono. [Accessed: 10-Jun-2021].

[28] “Gathering Information About Memory Use,” *Apple Developer Documentation*. [Online]. Available: https://developer.apple.com/documentation/xcode/gathering-information-about-memory-use. [Accessed: 10-Jun-2021].

[29] “SPICE netlist - Multisim Live,” *NI Multisim Live*. [Online]. Available: https://www.multisim.com/help/simulation/spice-netlist/#:~:text=A%20SPICE%20netlist%20is%20a,simulation%20errors%20and%20convergence%20issues. [Accessed: 11-Jun-2021].

[30] M. Rouse, “TXT File Format,” *TechTarget*, Jul-2010. [Online]. Available: https://whatis.techtarget.com/fileformat/TXT-ASCII-text-formatted-data#:~:text=TXT%20is%20a%20file%20extension,encoded%20into%20computer%2Dreadable%20formats. [Accessed: 11-Jun-2021].

Appendix 1

SRS Document

Introduction:

This document will outline the purpose, audience, scope and use of our circuit simulator. It will also summarise its assumptions, dependencies and explain some technical language used in the specification.

Purpose:

- Analyse circuits and run a small signal stimulation
- Save time and cost of building a prototype circuit

Intended Audience:

- People who are familiar with the SPICE format and have large circuits to stimulate

Intended Use:

- Used to run a small signal simulation and calculate the frequency response transfer function of the circuit, given designated input and output nodes
- Can be used to replace the need of building prototype circuits

Scope:

- A piece of software that can perform a small signal stimulation on an inputted netlist which is in a .txt file and the netlist will be in SPICE format
- The software should be able to handle both linear and non-linear components, limited to resistors, capacitors, inductors, voltage & current sources, diodes, BJTs, MOSFETs and voltage-controlled current source

Definitions and Acronyms:

- SPICE format refers to the text interpretation of a circuit [29], used by many other circuit simulators such as LTSpice
- A .txt file is a plain text file that can be interpreted by a computer [30]
- BJT is the acronym for bipolar junction transistor
- MOSFET is the acronym for metal oxide semiconductor field-effect transistor

Overall Description:

- The software will be built from scratch and contain functions to interpret the netlist and finally output the transfer function of the frequency response

User Needs:

- The user should be able to create a .txt file containing the circuit netlist in SPICE format
- The user should also be able to input the input and output nodes for the program to output the transfer function

Assumptions and Dependencies:

Assumptions	**Dependencies**
The user has continued access to a computer and the software to run and compile the program	Dependent on the standard C++ library
The user should provide the program with a txt file	Dependent on a library that can perform linear algebra operations (Eigen Library)
User should provide the input and output nodes for their stimulation	Dependent on a piece of software that can run and compile a C++ program
The scope of the project should remain the same throughout the project	