

ELEC40006
Electronic Design Project
2021
Circuit Simulator

word count: 10036

Jacky Jiang, Haoran Wu, Tanglitong Zhang

Table of Contents

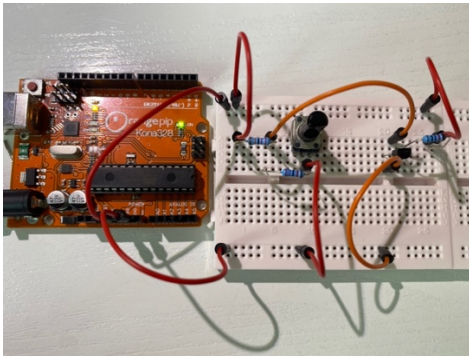
Introduction	3
Technical Problem	4
Software Requirement Specification	6
Purpose	6
Intended Audience.....	6
Product Feature	6
Functional Requirements	6
Nonfunctional Requirements	7
Design process.....	8
Overview	8
Input.....	8
Matrix.....	9
AC Analysis	12
DC Analysis	13
Output	13
Implementation.....	14
Overview	14
Input.....	15
1) Classification.....	15
2) Reading	20
3) Extraction.....	20
4) Set Up	22
Matrix Algorithm	24
1) Overview.....	24
2) General Conductance Matrix	24
3) Col_b.....	27
DC analysis.....	28
1) Preparation	28
2) Guess.....	30
3) Newton Raphson Linear Matrix	31
4) Results.....	36
AC analysis.....	37
Output	39
1) Input stage.....	39
2) DC operating point.....	40
3) Small-signal analysis.....	40
4) Output stage	40
Project Planning and Management	41
Overview	41
Initialisation.....	42

Planning & Management	42
Programming Process	43
Report & Video	44
Testing	45
Input.....	45
DC Analysis	48
AC Analysis	51
Total Testing	54
Evaluation	57
Conclusion	63
Appendix 1	64
Appendix 2	69
Appendix 3	72
References.....	75

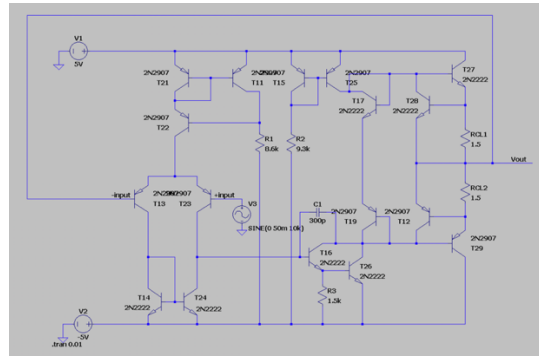
Introduction

“What started as a student project results in an industry standard for IC design [1, p. 36]” is the title for the book *“Shaping the History of SPICE,”* written by Andrei Vladimirescu, who is a professor at the University of California, Berkeley, associated with the research and development of SPICE 2G6 in 1981[1, p. 39]. When looking back to the history of circuit simulator programs, the Simulation Program with Integrated Circuit Emphasis (SPICE) was written first by students, tested by students became a design revolution as the standard for the circuit design industry. Inspired and impressed by the birth of SPICE, in 2021, we, as a team of three students at Imperial College London, see this end-of-year project as an opportunity ambitiously to not only replicate this victory of students but also to integrate our programming and design knowledge like SPICE integrates circuits as the students did in the 1970s.

Technical Problem



(Figure 1: breadboard design)



(Figure 2: LT SPICE design)

In our ADC LAB, a breadboard is used for board-level design with discrete components, and simulation software (LT SPICE) is used for transistor-level design composed of different circuit models. This is similar in industrial level design as it is not practical to breadboard the actual integrated circuit before simulation due to high costs of manufacturing prerequisites such as photomasks. Furtherly, testing performed on breadboarded circuits may have varying results due to parasitic component variations and component manufacturing tolerances. Finally, the level of complexity for analysis increases with that of the circuit and therefore would become impractical by manual calculations. Therefore, a simulation program is given birth to tackle the above-mentioned problems.

The purpose of the project is not as complex as to design a robust software like LT SPICE, but there are several technical problems the team needs to address:

- 🧩 How does the program read input from a pre-stored netlist file?
- 🧩 In what way does the program select a single source as the input source for a circuit composed of multiple input sources?
- 🧩 How does the program identify the number of nodes that a component is connected to and its voltage?

- LT SPICE allows the user to first select component icons, construct the circuit on a two-dimensional platform, and then run the analysis.



Does the program need to construct the circuit first and then run the analysis like LT SPICE, or is there another way of representing the circuit?

- How does the program conduct AC small-signal analysis with linear components such as resistors?
- How does the program conduct AC small-signal analysis with non-linear components such as BJT and MOSFET?
- How does the program separate linear component analysis and non-linear component analysis?
- How does the program visualize its output so the user can understand it?

Software Requirement Specification

Purpose

- ✚ The purpose of this software package is to build a simulation platform for circuits using C++.

Intended Audience

- ✚ Students, teaching fellows, and Professors at Imperial College London Electrical Engineering Department.

Product Feature

- ✚ DC analysis: calculates the quiescent operating point.
- ✚ AC small-signal analysis calculates the voltage at each node and gives the transfer function of the target circuit.

Functional Requirements

- ✚ Downloaded and installed Eigen Library.
- ✚ C++ compiler.
- ✚ The input file must be compatible with SPICE and ends with the line: “.end [2, p. 1]”

Designator letter	Component	Node order	Value
V	Independent Voltage source	+, -	Voltage (V) or function
I	Current source	In, out	Current (A) or function
R	Resistor	Doesn't matter	Resistance (Ω)
C	Capacitor	Doesn't matter	Capacitance (F)
L	Inductor	Doesn't matter	Inductance (H)
D	Diode	Anode, Cathode	Model name
Q	BJT	Collector, Base, Emitter	Model name
M	MOSFET	Drain, Gate, Source	Model name
G	Voltage-controlled Current Source	+, -, Control +, Control -	Transconductance (S)

(Table 1: netlist format specification 1, [2, p. 1])

With the format of each line being:

```
<designator> <node0> <node1> [<node 2> [<node 3>]] <value>
```

(Figure 3: netlist format specification 2, [2, p. 1])

Correct Simulation Settings


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

(Figure 4: command line specification, [2, p. 2])

Nonfunctional Requirements

 BJT: NPN: 2N2222 PNP: 2N2907

 User selection of input source and output node.

 User input of input file name and output file name.

Design process

Overview

After days of thorough discussions, we have categorized the above-mentioned technical problems into five parts: Input, Output, Matrix, DC analysis, and AC analysis.

Input

What the program needs to do in this input stage is to be able to read the whole netlist file, extract 'information,' and store them in appropriate locations that are easy to be accessed. About reading, the first thing is to push back the entire file line by line into a vector of type string because every single line has meaningful 'information' except for the first line beginning with '*', which is a comment line. Then, we need to figure out an appropriate and convenient way to store the values and assign them to the components in the extraction stage. After many attempts and discussions, we decided to construct the classes of various components first. Then we made them the derived classes of a general class, which means addresses of all different components can be stored within a vector of a single type (pointers of type general).

Matrix

The first thing we have decided is to represent the circuit using matrices. Even though it is suggested by the guide to use matrix, the reasons we hold for using matrices is because of the following: First, unlike manual calculation that requires circuit diagram to conduct nodal analysis, the program does not need to look at a diagram to start analyzing. Instead, all that the program needs is “information” of the circuit, meaning resistances, currents, voltages, etc. Second, the method we plan to store this “information” is matrix algebra $A\underline{x} = \underline{b}$ [3, p. 6] which produces the voltages at all nodes in one step.

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

(Figure 5: matrix representation 1, [3, p. 6])

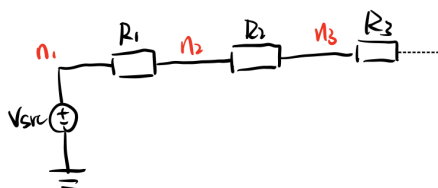
Therefore, finding and multiplying the inverse of the conductance matrix gives:

$$A^{-1}A\underline{x} = A^{-1}\underline{b}$$

$$\underline{x} = A^{-1}\underline{b}$$

However, writing a program to find the inverse of the conductance matrix may not be as easy as it seems. Thus, we will introduce the use of the Eigen library later.

The above algorithm is a generalized situation. In a real circuit, voltage sources need to be considered specially because of their position in the circuit. Take a simple RC circuit for example,



(Figure : RC circuit 1, v_n represents the voltage at node n)

Nodal analysis using KCL gives:

$$\begin{aligned}
 n_1: v_1 &= v_{src} \\
 n_2: \frac{v_2 - v_1}{R_1} + \frac{v_2 - v_3}{R_2} &= i_2 \\
 n_3: \frac{v_3 - v_2}{R_2} + \frac{v_3}{R_3} &= i_3 \\
 &\vdots
 \end{aligned}$$

Rearranging (G_{mn} is the conductance directly connectin node m and n):

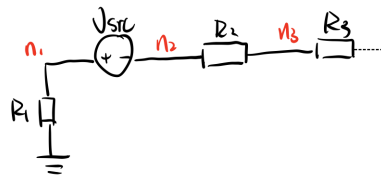
$$\begin{aligned}
 n_2: G_{21}(v_2 - v_1) + G_{23}(v_2 - v_3) &= i_2 \\
 -G_{21}v_1 + (G_{21} + G_{23})v_2 - G_{23}v_3 &= i_2 \\
 -G_{21}v_1 + G_{22}v_2 - G_{23}v_3 &= i_2 \\
 \\
 n_3: G_{32}(V_3 - V_2) + G_3V_3 &= i_3 \\
 -G_{32}V_2 + G_{33}V_3 &= i_3 \\
 &\vdots
 \end{aligned}$$

Now, this “information” represents the circuit in diagram 3. In this case, one terminal of the voltage source v_{src} is connected to the reference node (ground), and the other terminal is connected to n_1 , so that the conductance connected to n_1 can be ignored and we can express this as $v_1 = v_{src}$. Then, by observing the pattern of the rest of the equations, we concluded that the current at each node is a sum of the product of the conductance and the unknown voltage. Therefore, we can input these figures into matrices and form a beautiful algorithm $A\underline{x} = \underline{b}$.

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

(Figure 7: matrix representation 2, [3, p. 7])

If v_{src} is connected between two non-reference nodes, for example, between n_1 and n_1 as follows:



(Figure 8: RC circuit 2)

Then:

$$v_{src} = v_1 - v_2$$

$$n_1: \frac{v_1}{R_1} + i_{vsrc} = i_1$$

⋮

Because v_{src} is connected between n_1 and n_2 , we have to consider the current i_{vsrc} flowing through v_{src} and the conductance between n_1 and n_2 is invalid.

Rearranging:

$$n_1: G_{11}v_1 + i_{vsrc} = i_1$$

⋮

This extra i_{vsrc} can be added as an extra column to the matrices:

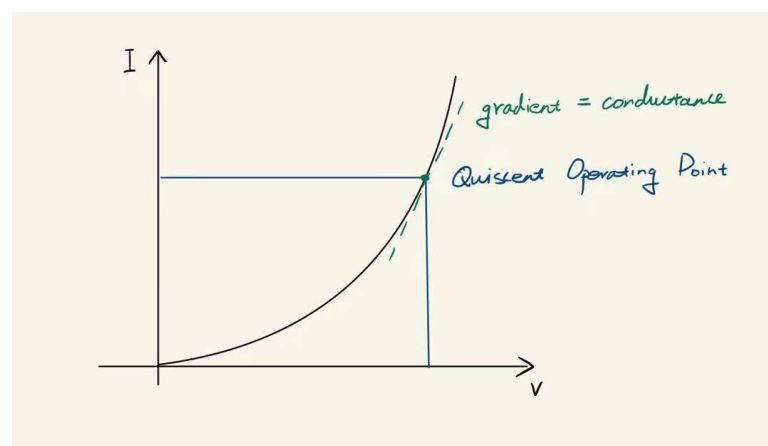
$$\begin{bmatrix} 1 & -1 & \cdots & 0 & 0 \\ G_{11} & 0 & \cdots & -G_{1n} & -1 \\ 0 & G_{22} & \cdots & -G_{2n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -G_{n1} & -G_{n2} & \cdots & G_{nn} & 0 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \\ i_{vsrc} \end{bmatrix} = \begin{bmatrix} v_{src} \\ i_1 \\ i_2 \\ \cdots \\ i_n \end{bmatrix}$$

(Figure 9: matrix representation 3, [3, p. 7])

AC Analysis

AC Analysis is based on the result from DC analysis. It constructs a linear circuit since all the non-linear components like BJTs, MOSFETs and Diodes are transformed into small-signal components. Therefore, we would be able to find the transfer function between the input source and the node voltage.

The software aims to perform small-signal AC analysis simulations of circuits. In this analysis, we plan to delete the DC sources in the circuit and find the AC voltage on each node. However, since some components are not linear, such as BJTs, MOSFETs, and Diodes, we have to linearize them into small signal components. This is because linearization could vastly improve the efficiency of the calculation. We could directly use the matrix method provided in the material to solve the result. One valid and efficient way is to use the DC analysis to find the quiescent operating point and to use the partial derivative to find the equivalent resistance or transconductance to replace the non-linear devices. According to the graph below, after finding the operating point, the gradient could be the equivalent conductance through the derivative at that point.



(Figure 10: operating point)

DC Analysis

DC Analysis is mainly used to find the quiescent operating point of each node, and it provides the information for the circuit to be transformed into SSEM. To find this, we used the Newton Raphson Method.

Output

Our initial thought was to apply functions one by one in the main, but then we realized that it was messy and decided to integrate all stages into one single void function: `find_final_sol`

Implementation

Overview

Our implementation inherits the logic and has five central algorithms and several sub algorithms as discussed in the design process.

Input

1) Classification

```
#include<cmath>
#include<string>
#include<iostream>
#include<vector>
#include<complex>
#include<fstream>
#include<cstdlib>
#include<sstream>
#include<math.h>
#define PI 3.14159265
const double VT = 0.02585;
//bjt
// pick 2N2907 PNP VAF = 120 BF = 250 Is = 1E-14 BR = 3;
// pick 2N2222 NPN VAF = 100 BF = 200 Is = 1E-14 BR = 3;
double VAnpn = 120;
double VApnp = 100;
double BFnpn = 200;
double BFpnp = 250;
double BR = 3;
double Is = pow(10, -14);
double Isc = ((1+BR) / BR) * pow(10, -14);
//mosfet
// for NMOS vt = 2.9 kp = 0.9m l = 10u w = 100u lambda = 0.01 VA = 100;
// for PMOS vt = -2.5 kp = 0.4m l = 10u w = 100u lambda = 0.03 VA = 300;
double Kn = 0.009;
double Kp = 0.004;
double vtn = 2.9;
double vtp = -2.5;
double VAn = 100;
double VAp = 300;
```

(Figure 11: classification)


The first step is to define BJT and MOSFET parameters with specific model obtained from LT SPICE.


BJT: NPN-2N2222, PNP-2N2907, Diode: 1N914

We chose to define these classes and use virtual functions because general contains virtual functions that are used to obtain specific data. Components do not share the same parameters and behaviour, so we cannot simply apply one function to get data for all of them. Inherited from the general, the component classes below have their unique parameters as private members so that they can overwrite the virtual functions using their own parameters.

```
31 > class general{ ...
73 > class Resistor : public general{ ...
88 > class Capacitor : public general{ ...
103 > class Inductor : public general{ ...
118 > class Diode : public general { //model 1N914 ...
144 > class voltsrc : public general{ ...
191 > class currsrc : public general{ ...
238 > class bjt: public general{ ...
549 > class mosfet : public general{ ...
717 > class v_currsrc : public general{ ...
```

(Figure 12: class)

 **class general:** **conductance** is used to calculate conductance at a given angular frequency. **nodes** indicates the number of nodes connecting to the component, which in most cases is two. Thus, **get_node** and **get_polarity** only consider the situation where the number of nodes is two. The former is to used arrange the two nodes in ascending order; the latter is designed to indicate the positive and negative polarities of the two nodes since polarity plays an essential role for specific components. **get_type** is used to return the name of the component: R for resistor, etc. The last function is a destructor, The nodes are defined as protected so that they can be accessed and used in the derived classes below.

 **class Resistor:** This is a derived class of the class general and inherits from the public, so we input the nodes, types and also resistance a unique member of Resistor in the

class constructor list. The `conductance` is rewritten to return $1/R$ to calculate the conductance of a resistor (constant over all frequencies).

✚ `class Capacitor`: Unlike resistors with real conductance, capacitors are reactive components that react to change and thus have purely imaginary conductance.

✚ `class Inductor`: Similar to capacitors, inductors are also reactive components and have imaginary and negative conductance.

✚ `class Diode`: LT SPICE model: 1N914. `id` calculates the diode current using formula:

$$i = I_s(e^{v/V_T} - 1) \text{ [4, p. 184]}$$

`g` calculates the small-signal conductance using formula:

$$r_d = \frac{V_T}{I_D} \text{ [4, p. 197]}$$

✚ `class voltsrc`: Two constructor models depend on the number and data type of the input. The first one is for DC voltage, and the second one is for AC voltage with a phase. `get_volt` gives the value of voltage (DC) or the amplitude (AC). `dc_ac` determines the type of the voltage source. `search_name` gets the name of the source as given in the input file.

✚ `class currsrc`: Similar to `class voltsrc`, but voltage is replaced by current instead.

✚ `class bjt`: Allocates the nodes to their corresponding parts of the BJT. `initialize_bjt` contains calculation of the collector, base, and emitter currents for both NPN and PNP

BJTs under the four different operating modes (active, saturation, reverse active, and cut-off).

For active mode:

$$i_C = I_S e^{\frac{V_{BE}}{V_T}} \cdot \left(1 + \frac{V_{CE}}{V_A}\right)$$

For the rest three modes, we use the Ebers–Moll equations to calculate the three currents out from the BJT.

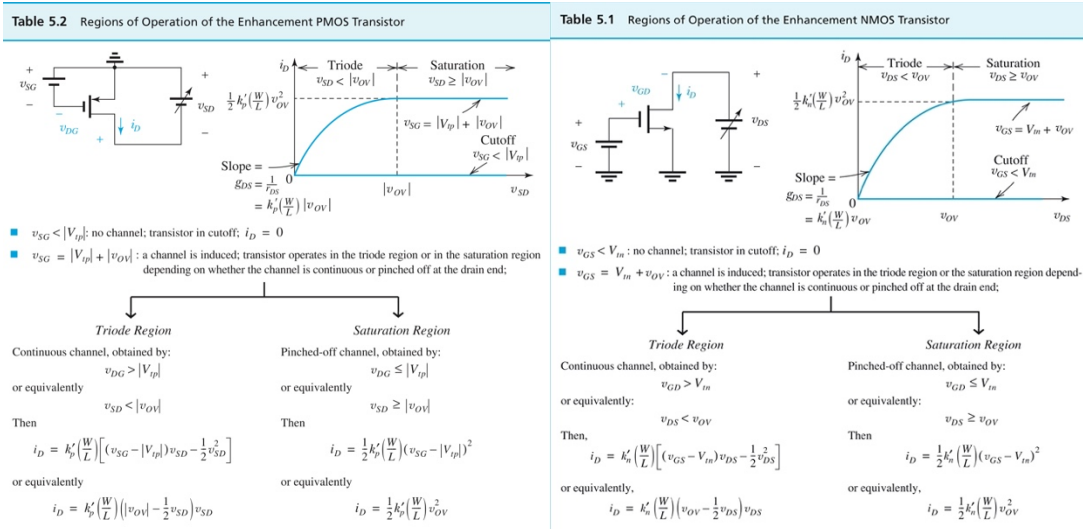
$$i_C = I_S \left[\left(e^{\frac{V_{BE}}{V_T}} - e^{\frac{V_{BC}}{V_T}} \right) - \frac{1}{\beta_R} (e^{\frac{V_{BC}}{V_T}} - 1) \right] [5]$$

$$i_B = I_S \left[\frac{1}{\beta_F} (e^{\frac{V_{BE}}{V_T}} - 1) + \frac{1}{\beta_R} (e^{\frac{V_{BC}}{V_T}} - 1) \right] [5]$$

$$i_E = I_S \left[\left(e^{\frac{V_{BE}}{V_T}} - e^{\frac{V_{BC}}{V_T}} \right) + \frac{1}{\beta_F} (e^{\frac{V_{BE}}{V_T}} - 1) \right] [5]$$

A private member state is used to distinguish between the four modes. `get_polarity` stores the nodes in order of C, B, E for later uses. `rbe` gives the small-signal base-emitter resistance in the simplified hybrid- π model. `r0` gives the small-signal output resistance. `gm` gives the small-signal transconductance. Functions below are used for iteration matrix and perform partial derivatives to each of the three nodes with respect to voltages at three nodes. The first letter indicates which current we are differentiating; the second one shows which node voltage we are differentiating with respect to. For instance, `b_b` means getting a partial derivative of base current with respect to base voltage. Also, here we use I_{SC} to represent $\frac{I_S}{\beta_F}$ in the Ebers-Moll equations above.

class mosfet: MOSFETs are similar to BJTs in the way that they are both 3-terminal transistors. However, the only difference is that MOSFETs have no rbe. There are three states for MOSFETs, and the same process is applied, i.e., partial derivative to drain current.



(Figure 13: PMOS, [4, p. 275])

(Figure 14: NMOS, [4, p. 275])

class v_currsrc: A voltage-controlled current source has four nodes, and a unique member is the transconductance. `get_i` gives the current source.

multiplier: The input figure can be followed by a multiplier which means the exact value has to be obtained by scaling the number through multiplication according to the multiplier. `multiplier` takes a double and a character as input, with the double using passing by reference. This is because passing by reference stores the input data address, so any changes made to that double below will change the original value in that address directly. The function identifies the character following the figure and applies the corresponding multiplication process to the double, producing an exact numerical value.

2) Reading

- ✚ **ReadInput**: The first step of doing the analysis is reading the text in the input file and storing values that we need for future analysis. **ReadInput** takes a file name as input and reads the file, and stores it into a vector line by line.

```
804 > vector<string> ReadInput (string filename){ //read file, store in vector of type string-
824
825 > void multiplier(double &value,char m){-
848 > string getword(string tmp, int wd){ //the leftmost one of the line is word 1, not word 0-
856 > string lastword(string tmp){-
864 > double ConvertFromString(string str) { //string to double-
871 > void multi(double &value, string s){-
931 > vector<double> ac (vector<string> input){-
948
949 > void setting (vector<string> input, vector<general*> &component, int &max_node){ //split lines into parts, store in vector of type comp
```

(Figure 15: input)


3) Extraction

- ✚ **Getword[6]**: This is used to get a specific word (a sequence of string between spaces. For example, the second word of string "I am a girl" is "am"), with the leftmost text being word 1. Header `<sstream>` is included so that `istringstream` can be used, separating the input string into a shorter string with respect to space. So the function takes a long string and an integer as the input and produces the word at the position of the value of integer.
- ✚ **lastword**: This is used to obtain the last text of the long input string that needs to be used later with the **multiplier**.
- ✚ **ConvertFromString[7]**: `istringstream` is also used in this function to convert the number in string format to double format.

✚ **multi**: This function takes a string which will be a specific word of a line of the input file, and a value (in most cases 0 since it will be reset after operations in this function) as input parameters passed by reference. Since this function is expected to give numerical values after multiplications according to multipliers, several cases need to be considered. Our first attempt was to write a few functions to implement the features of **multi**, which was theoretically correct but messy and unsmart. So we improved it by integrating all of them into one single function. There are two parts: one considers the case where the input word contains an AC phase, the other considers the input word being either an AC amplitude or a DC value. In each part, the function performs the corresponding operations depending on the existence of a multiplier.

✚ **ac**: This aims to store useful values for the AC analysis: points per decade, starting and stopping frequencies for the sweep. It reads from the second last line of the input file and makes use of **multi**, and **getword**.

4) Set Up

 **setting:** This is the most crucial part after taking in the input. In this function, components are initialized and given a dynamically allocated memory area to store their characteristics. It takes three things as input, a vector that stores the input text, a vector that stores all components, and an integer that indicates the maximum number of nodes. The outer **for** loop makes sure the program only focuses on up to the second last line (not including it). Lines beginning with ‘*’ are comment lines that are not useful for analysis may be written into the file as well. Therefore we created a condition, so the program only starts computing when the current line is not a comment. We started by storing the two nodes. Since we would like to convert nodes from string to integer by using the **stoi**, an **if** condition is added to avoid the situation where the input of ‘stoi’ function is 0. At first, we did not take this into consideration and got an error as a result. Another function, ‘substr’ is used to get a sequence of string starting from a specific index to the end. So far, we only dealt with two nodes, so a simple comparison is made to give the larger value to the integer `max_node`. Now the program can initialize the components respectively by using **if** condition for each type. Components are distinguished by the designator field. For voltage and current sources, since their values can be either a function (AC with amplitude and phase) or a numerical value, the program separates these two cases again using **if** condition as the constructor lists are different for them. We can quickly get the amplitude and phase with the help of function **mul**, then assign data to the right place in the constructor list, and finally, use a pointer of type `general` to locate the dynamically allocated memory area which represents the component. Then the program repeats the same process for resistors, capacitors, inductors, and diodes. We introduced a third node when it comes to transistors, converting it to an integer and making the same comparison as above to update the

max_node. The last one to consider is the voltage-controlled current source, which has four nodes. So two more nodes need to be introduced. At the end of `if` condition for each component, the program stores that pointer into the vector, a referenced input. Eventually, after calling this function, there will be a vector storing all addresses of the given components and a number indicating the number of nodes in the circuit. Notice that we gave the components dynamically allocated memory area, but the pointers to find them are of type `general`. We kept wondering how to call the unique functions that are in the public section of a certain type of component. We cannot simply call it because it is not legal. We can only use those when written as `virtual`. Fortunately, we came to a solution called `dynamic_cast`, which would be applied and explained later.

Matrix Algorithm

1) Overview

Almost all the matrices we built in different functions to serve multiple purposes are based on the general case conductance matrix and the `col_b` column matrix. Here we would introduce how we built these two matrices step by step. The only assumption here is that all the components are linear. These two matrices could be directly used to calculate the voltage vector in AC analysis as all the non-linear components are linearized already.

2) General Conductance Matrix

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

(Figure 16: matrix representation 1, [3, p. 6])

The conductance matrix is the matrix on the leftmost and contains the conductance from one node to the other. Firstly, without considering the voltage sources, each row represents a KCL equation for a node. For example, in terms of node 2, the second row of the conductance matrix times the voltage vector matrix gives the current in node 2,

$$-G_{21} * V_1 + G_{22} * V_2 - G_{23} * V_3 - \dots - G_{2n} * V_n = i_2$$

In terms of all the conductance in this equation, it stands for the sum of all the conductance from this point to others (including the ground node). After replacing this with other conductance, we have:

$$G_{20} * V_2 + G_{21} * (V_2 - V_1) + G_{23} * (V_2 - V_3) + \dots + G_{2n} * (V_2 - V_n) - i_2 = 0$$

i_2 is the current flowing into that node, and the currents in all nodes together form the `col_b` matrix, which we will discuss later.

The voltage source also needs to be taken into consideration. As illustrated in the design process, there are two ways that a voltage source can be connected, and the connection affects the structure of the matrices. One is to connect one node of the source to the ground:

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

(Figure 17: matrix representation 2, [3, p. 7])

And the other is that both ends of the voltage source are connected to nodes:

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

(Figure 18: matrix representation 2, [3, p. 7])

Different from the first conductance matrix, a new row on the top of the conductance matrix represents the voltage source connecting to two nodes. Besides, a new column on the right side represents the current flowing through the source when multiplied by the voltage vector.

In order to build the matrix, our method is to build it row by row. We first considered the situation where the voltage source is connected to two nodes. We defined the number of this kind of voltage source connection as `num_de`. The size of this matrix is, therefore, the sum of the `num_de` and the number of nodes. Then, we built the first row that represents the voltage source. If the index of the current row is less than the `num_de`, then it means we still need to build the voltage source row. We put 1 and -1 to the corresponding position according to the node of the voltage source. If the row index is equal to the sum of the maximum number of nodes and `num_de` and there is a voltage source connecting to the ground and this node, the program will continue to build a row that contains only number 1 in that row. If the program is building the voltage source row, there is a need to use the loop to construct each column of this row. That is why we put constructing voltage source in the initial part of the function. For the

rest of the components, we created a chain of loops to build each cell of the matrix. The inner loop is designed to search for components connected to two nodes in the input vector. If they are detected, the conductance between the two nodes they are connected to will be stored in a new defined variable `cond`. After this process, the program returns to the outer loop column and starts doing the same research and storing process with the next column. Finally, the program returns to the outmost row loop and adds the `cond` from all columns to `tot_cond` and input this value into the matrix at position G_{mn} . However, this value does not fully cover the total conductance. We still need to consider the situation when the component has one node connecting to the ground. Therefore, in the last inner loop, we stored the conductance into a new variable called `ground_cond`. For special components VCCS (voltage controlled current sources), we could not add their current value directly to the column matrix on the right even though they are current sources. This is because the current value depends on the voltage difference between two nodes. Therefore, we decided to put them into the conductance matrix by adding their transconductance to the following four positions. Since each row inside the conductance matrix represents the KCL equation for a node, the node that connects to the negative side of this source has the equation:

$$G_{trans} * v_{control+} - G_{trans} * v_{control-} + else = i$$

For the node that connects to the positive side of this source:

$$G_{trans} * v_{control-} - G_{trans} * v_{control+} + else = i$$

Therefore, we chose to add or subtract its transconductance to the cells where the row represents the two nodes, and the columns represent the two control nodes according to the two equations.

3) Col_b

The column matrix on the right only contains the value of the current source or voltage source.

For the rows where the conductance matrix represents the voltage source, the `col_b` should be the value of this voltage source. In terms of other rows, it only contains the value of the current flowing into that node.

DC analysis

```
1  > #include<iostream>
2  > #include<complex>
3  > #include<vector>
4  > #include<cmath>
5  > #include<string>
6  > #include"part1.cpp"
7  > #include<eigen3/Eigen/Dense>
8  > struct shortcircuit{ ...
12 > std::vector<general*> short_circuit(std::vector<general*> in, std::vector<shortcircuit> &str
106 > std::vector<int> short_str(std::vector<shortcircuit> sc, int maxnodein){ ...
121 > int find_index(std::vector<int> input, int val){ ...
130 > void dc_volt (std::vector<voltsrc*> tmp, int &num, std::vector<int> &n, std::vector<int> &g,
145 > void classify_comp(std::vector<general*> input, std::vector<Resistor*> &r, std::vector<bjt*>
174 > std::vector<general*> reorganizedc(std::vector<general*> input, int& extra_node){ ...
211 > Eigen::MatrixXd recover_circuit(Eigen::MatrixXd stand_volt, std::vector<shortcircuit> sc){ ...
233 > Eigen::MatrixXd build_iterate_matrix (std::vector<general*> in, int maxnode, Eigen::MatrixXd
510 > Eigen::MatrixXd build_fvm_matrix(std::vector<general*> in, int maxnode, Eigen::MatrixXd vin)
688 > Eigen::MatrixXd build_guess_volt (std::vector<general*> in, int maxnode){ ...
859 > bool compare_volt(Eigen::MatrixXd result, Eigen::MatrixXd test){ ...
868 > Eigen::MatrixXd get_standart_volt(std::vector<general*> input, int maxnode){ ...
```

(Figure 19: DC analysis)


1) Preparation


✚ **struct shortcircuit**: It is used to construct a type to contain the reference node and the short node. These two nodes are short-circuited and have the same voltage. Here we chose to keep the reference node and delete the short node in the following functions.


✚ **short_circuit**: This function is used to delete the short nodes and change all the components connecting to these nodes. First, the function searches for the components with zero resistance and stores those nodes in the form of the type we defined before. Then, to change the node, the function also searches for all the input components, deletes the corresponding pointer, and adds a new pointer with the nodes changed to the reference nodes. Lastly, **short_circuit** helps connect the short node and reference node in the recover state for the following part.


✚ **short_str**: This function is used to store the reference nodes inside a vector. It would be useful when filling in the matrix. This is due to the fact that each row inside the matrix

(Except for the condition where there is a voltage source connecting to the two nodes), should correspond to a node voltage. Since some of the nodes are deleted, the row of the matrix is no longer directly related to the voltage from v_1 to v_n . Within this vector, the row of the matrix could be translated as the index of the vector and consequently be related to the nodes again.

 **find_index**: This function is like the reversed process of the **short_str** function. This function aims to search for the index of the vector. It is used to store values in the matrices.


 **dc_volt**: This is built to distinguish DC voltage sources further. We wanted to put the DC voltage source into two vectors. One of them is used to store DC voltage with one node connecting to the ground, and the other is to store the DC voltage source connecting to two different nodes. This classification helps us decide whether we should add more rows and arrange the position of the voltage value inside the matrices.

 **classify_comp**: This is created to classify the components (Resistor, BJT, MOSFET, voltage source, and current source), simplifying the calculation process in the following matrices.

 **reorganizedc**: A reorganize function that changes the original input pointer vector containing non-linear components to voltage sources. The input parameter **extra_node** indicates the number of the additional voltage sources connecting to two nodes. This is because this type of node would increase the matrices' size and needs to be deleted in the end.

2) Guess

This part is mainly used to provide the initial condition for the Newton Raphson Method. The core strategy is to treat the non-linear components as a set of voltage sources to assume the node voltage briefly.

 **build_guess_volt**: For the input of the guess part, we need first to use the short circuit function to change the vector; then, we apply the reorganize function to reach our requirement. Then, as usual, we built the conductance matrix and the column matrix on the right and solved to get the guess voltage. The process is more complex with the introduction of the short circuit method. This is because it could no longer form a row inside the matrix for the short node, and we have to skip it. **short_str** and **find_index** are used to decide the precise row. After deleting the extra voltage source value in the last rows, we get the guess voltage Matrix.

3) Newton Raphson Linear Matrix

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_n^- \mathbf{f}(\mathbf{x}_n)$$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}_{n+1} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}_n - \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \mathbf{J}^-(\mathbf{x}_n) \begin{bmatrix} f_1(\mathbf{x}_n) \\ \vdots \\ \vdots \\ \vdots \\ f_M(\mathbf{x}_n) \end{bmatrix}$$

(Figure 20: Newton Raphson Linear Matrix, [8])


This method lightens a way to find the accurate solution of the matrix $f(x_n)$. In our program we made the $f(x_n)$ be fvm , which contains all the current flowing into or out of the node. The matrix $J^-(x_n)$ is the matrix that contains all the partial derivatives of the $f(x_n)$ with respect to all the x_n . We named it the iterate matrix here, and each row of this matrix represent the equation with respect to all the node voltage.

$$\begin{bmatrix} \frac{\partial f v_1}{\partial v_1} & \dots & \frac{\partial f v_1}{\partial v_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f v_n}{\partial v_1} & \dots & \frac{\partial f v_n}{\partial v_n} \\ v_1 & & v_n \end{bmatrix}$$

In terms of those non-linear components, in order to take the current out from them or the partial derivatives, the voltage on those nodes needs to be calculated first. Therefore, to construct the iterate matrix or the fvm matrix, we made the voltage column vector as an input for the two functions to build the matrix.

Apart from that, to start the iteration, voltage vector v_0 needs to be an initial input. In the following function explanation part, we used the v_n to represent the voltage on the right and v_{n+1} to represent the result of this matrix equation. During the iteration part, we put the v_{left} on the left to the right v_{left+1} , obtaining the next v_{left+2} for every

cycle, meaning $v_n = v_{n-1}$ for the previous cycle, $v_{n+1} = v_n$ for the current cycle, and $v_{n+2} = v_{n+1}$ for the next cycle.

 **build_iterate_matrix:** This function is the core algorithm of the Newton-Raphson Method. It contains the partial derivative of fvm with respect to different voltages. Each row of this matrix contains the partial derivatives for the same row of the fvm matrix with respect to all the voltage nodes. The arrangement conforms to the index sequence of the nodes. We constructed a matrix that only contains resistors, voltage-controlled current sources, and two ways of voltage source connections for the first part. The way we constructed this matrix is slightly different from the general one as we used the short circuit method that changed the size of the matrix and input data into the matrix.

```
int num = num_de + maxnode - shortc.size();
Eigen::MatrixXd mat(num, num);
for(int standposi = 0; standposi < num; standposi++){
    int row = 0;
    if (standposi < num_de){
        row = standposi;
    }
    else{
        row = stand_posi_in_matrix[standposi - num_de] + num_de - 1;
    }
}
```

(Figure 21)

In terms of the size of the matrix, since we deleted some of the nodes, we reduced the size of the matrix by the number of these nodes, which changed the index of rows in the matrix. Therefore, we defined a new variable called **standposi** to represent the index of the row matrix after eliminating the nodes in the short circuit and then regrouped the index in ascending order. Through selections, the row should be equal to **standposi** within the number of the extra **num_de** (the number of the voltage source connecting to the two nodes) in order to give the empty row for calculating the voltage.

If `standposi` is larger than `num_de`, it represents the index of the node. In terms of column matrices, we applied the same method to regroup the index under short circuit conditions.

Unlike the rows, we did not need to pay attention to the `num_de` as the voltage source component does not contribute to the start of the columns for each row.

Then, for the following part, we added the partial derivative for the current of the non-linear components. For Diodes, they can be connected in the same way as the voltage sources.

```
for (int i = 0; i < d.size(); i++){
    int anode = find_index(stand_posi_in_matrix , d[i]->get_polarity()[0]);
    int cathode = find_index(stand_posi_in_matrix , d[i]->get_polarity()[1]);
    if(d[i]->get_node()[0] != 0){
        //anode row
        mat(anode + num_de, cathode) += - d[i]->g(volt_vector[anode], volt_vector[cathode]);
        mat(anode + num_de, anode) += d[i]->g(volt_vector[anode], volt_vector[cathode]);
        //cathode row
        mat(cathode + num_de, cathode) += d[i]->g(volt_vector[anode], volt_vector[cathode]);
        mat(cathode + num_de, anode) += -d[i]->g(volt_vector[anode], volt_vector[cathode]);
    }
    else{
        if(d[i]->get_polarity()[0] == 0){
            mat(cathode + num_de, cathode) += d[i]->g(0, volt_vector[cathode]);
        }
        if(d[i]->get_polarity()[1] == 0){
            mat(anode + num_de, anode) += d[i]->g(volt_vector[anode], 0);
        }
    }
}
```

(Figure 22)

We again used `find_index` to find the position of the voltage value stored in `volt_vector`. This is because this vector is already short-circuited, which means the short node is deleted. We also discussed the condition when the diode is connecting to the ground.

For the BJTs, it is much more complex. Firstly, we found the voltage on the three terminals of BJTs. (Collector, Emitter, Base). Also, except for using the `find_index` function to obtain the corresponding node voltage, we also discussed the condition when some terminals of the BJT are connecting to the ground.

```

for(int m = 0; m < b.size(); m++){
    double VB, VC, VE;
    int b_node, c_node, e_node;
    c_node = find_index(stand_posi_in_matrix, b[m]->get_polarity()[0]);
    b_node = find_index(stand_posi_in_matrix, b[m]->get_polarity()[1]);
    e_node = find_index(stand_posi_in_matrix, b[m]->get_polarity()[2]);

    if(b[m]->get_polarity()[2] != 0){
        VE = volt_vector[e_node];
    }
    else{
        VE = 0;
    }
    if(b[m]->get_polarity()[1] != 0){
        VB = volt_vector[b_node];
    }
    else{
        VB = 0;
    }
    if(b[m]->get_polarity()[0] != 0){
        VC = volt_vector[c_node];
    }
    else{
        VC = 0;
    }
}

```

(Figure 23)

Since we used `find_index` to locate the index of the nodes, we could access the cell by simply eg. `b_node + num_de, c_node`. After obtaining VB VE VC, we initialized the BJT by putting these into `class bjt`.

```

}
b[m]->initialize_bjt(VB, VE, VC);
// for bjt with b on that row.
if(b[m]->b() != 0){
    mat(b_node + num_de, b_node) += b[m]->b_b();
    if(b[m]->e() != 0){
        mat(b_node + num_de, e_node) += b[m]->b_e();
    }
    if(b[m]->c() != 0){
        mat(b_node + num_de, c_node -1) += b[m]->b_c();
    }
}
}
// for bjt with e on that row.
if(b[m]->e() != 0){
    mat(e_node + num_de, e_node) -= b[m]->e_e();
    if(b[m]->b() != 0){
        mat(e_node + num_de, b_node) -= b[m]->e_b();
    }
    if(b[m]->c() != 0){
        mat(e_node, c_node) -= b[m]->e_c();
    }
}
// for bjt with c on that row
if(b[m]->c() != 0){

```

(Figure 24)

The iteration process for MOSFETs is similar to that of BJTs.

+ **build_fvm_matrix**: We made each row of this matrix the KCL equation with the right side of it equal to zero theoretically, considering all the current flowing in or out of this node, to each node except for the row representing the voltage sources. If the voltage vector provided does not make the equation to be zero, then this value would affect the generation of the next voltage vector. For the two ways of voltage source connections, we used $v_{node1} - v_{node2} - v_{src} = 0$ to form the fvm matrix voltage row. In terms of the DC current source, we changed the general **col_b** matrix on the right to the left by simply multiply -1 for each row. Apart from that, in terms of the non-linear components, since their current is contained in their own class, we only needed to add that current to the row where their nodes are represented. Lastly, for the Resistor, we considered two ways of connections. For resistors connecting to the two nodes, we added

$$(v_{node1} - v_{node2}) * Cond + else = 0$$

```

// consider resistor
for(int y = 0; y < r.size(); y++){
  int posi, nega;
  posi = find_index(stand_posi_in_matrix, r[y]->get_polarity()[0]);
  nega = find_index(stand_posi_in_matrix, r[y]->get_polarity()[1]);
  if(r[y]->get_node()[0] != 0){
    col_b(posi+ num_de , 0) += (guess_volt[posi] - guess_volt[nega]) * r[y]->conductance(0).real();
    col_b(nega+ num_de , 0) += (guess_volt[nega] - guess_volt[posi]) * r[y]->conductance(0).real();
  }
  else{
    col_b(find_index(stand_posi_in_matrix, r[y]->get_node()[1]) + num_de, 0) += guess_volt[find_index(stand_posi_in_matrix, r[y]->get
  }
}
  
```

(Figure 25)

For resistors connecting to the ground, we only considered the node that is not connecting to the ground and added the product of the voltage of that node times the conductance. That is all for fvm matrix.

4) Results

The final process of obtaining the DC operating point is illustrated below:

✚ **compare_volt**: It is used to compare x_n and x_{n+1} . If the value differences of rows between them are less than 0.001, the **compare_volt** function would return true and terminate the loop.

✚ **recover_circuit**: Finally, since the column voltage matrix is in the form of short circuits, this function is built to recover the shorted node. Basically, we put the voltage for the reference nodes to the row position of the short nodes.

```
209 Eigen::MatrixXd recover_circuit(Eigen::MatrixXd stand_volt, std::vector<shortcircuit> sc){
210     Eigen::MatrixXd tmp (stand_volt.rows() + sc.size(), 1);
211     for(int t = 0; t < tmp.rows(); t++){
212         tmp(t, 0) = 0;
213     }
214     std::vector<int> node_in_short = short_str(sc, tmp.rows());
215
216     // fill in gap that do not need alter position
217     for(int g = 0; g < node_in_short.size(); g++){
218         tmp(node_in_short[g] - 1, 0) = stand_volt(g,0);
219     }
220     // fill the shorted one
221     for(int m = 0; m < sc.size(); m++){
222         if(sc[m].ref_node == 0){
223             tmp(sc[m].short_node - 1, 0) = 0;
224         }
225         else{
226             tmp(sc[m].short_node - 1, 0) = tmp(sc[m].ref_node-1);
227         }
228     }
229     return tmp;
230 }
```

(Figure 26)

We first built a new matrix with the row number equal to the sum of the short circuit voltage matrix and the number of short nodes. Then we filled in the new matrix with the short circuit voltage to the position determined by **short_str** that returns the vector of the reference node position. Finally, we filled in the short nodes by putting the value of the reference nodes to the short nodes.

✚ **get_standard_volt**: This function applies the Newton Raphson method, which uses the **compare_volt** and **recover_circuit** and produces the DC operating point for each node.


AC analysis


All characteristics of the non-linear component are based on the results of the above DC analysis

```
ssem.cpp > ...
1 #include<iostream>
2 #include<complex>
3 #include<vector>
4 #include<cmath>
5 #include<string>
6 #include<eigen3/Eigen/Dense>
7 #include"_DCanalysis.cpp"
8 > std::vector<double> frequency (std::vector<double> ac){...
22 > void num_of_acvolt ( std::vector<voltsrc*> input, int &num_de, std::vector<int> &g , std::ve
34 > std::vector<general*> shortdcsource(std::vector<general*> input, std::vector<shortcircuit>& :
81 > void classify_comp_more(std::vector<general*> input, std::vector<Resistor*> &r, std::vector<
114 > Eigen::MatrixXcd SSEM (std::vector<general*> input , int maxnode, double frequency, Eigen::M
250 > std::vector<general*> reorganize(std::vector<general*> input , int maxnode, std::vector<doub
293 > Eigen::MatrixXcd build_acb (std::vector<general*> input, int maxnode, std::vector<shortcircu
347 > Eigen::MatrixXcd recover_complex_circuit(Eigen::MatrixXcd stand_volt, std::vector<shortcircu
369 > void find_final_sol (){...
```

(Figure 27)

- 🔧 **frequency**: We transferred the AC range read from the input netlist into the **frequency** function and put different frequencies into a vector to start AC analysis. The input parameter of this function is in the form of a vector that contains the number of nodes per decade and the start frequency and the end frequency. After the calculation, we store the frequency into another vector defined in the function.
- 🔧 **num_of_acvolt**: Similar to DC analysis, this function is built to classify AC voltage sources.
- 🔧 **shortdcsource**: This function is similar to the **short_circuit** in DC analysis, but the difference is that in AC analysis, we can only consider to short circuit the DC source, and in the meantime, as the circuit at this stage is already in its small-signal equivalent model, there would be no need to consider the BJTs and the MOSFETs especially because they are resistors and voltage-controlled current sources now.

 **reorganize**: This function is built to linearize the non-linear components (BJTs, MOSFETs, and Diode). The first thing the function does is to read from the netlist file and locates these components. Then, it reads again from the `stand_volt` vector, which is the result from DC analysis. It is worth mentioning that we don't need to use `find_index` to find the voltage of the corresponding nodes like we did when building the small-signal model in DC analysis. This is due to the fact that the `stand_volt` vector contains voltages for all nodes, including the ones being recovered. Therefore, by finding the node that the BJT is connected to, we can find its voltage. After initialization, the linearized non-linear components can be added to the original input vector. The construction of the conductance matrix and `col_b` belongs to the general case.

 **recover_complex_circuit**: Because at the end of DC analysis, the return is a matrix in `Xd <double>` format, and this is not suitable in AC analysis, we changed the format to `complex<double>`.

Output

✚ `find_final_sol`: This makes sure that the user only needs to input the name of this function and instructions in the main, and the rest of the instructions would appear in the terminal. The order of the implementation of the function is as follows:

1) Input stage

The program asks the user to give the name of an input file and then uses `ReadInput` to take in the content in the given file. With the given file, setting generates components accordingly, and `ac` and `frequency` obtain the list of frequencies required for the analysis. Before moving on to the next stage, we built upon what is required by the specification by allowing the user to choose the desired input source when there is more than one source. To make a choice clear, we created two vectors storing voltage and current sources, respectively, with a serial number and its designator. A third vector is created to store the designators of both sources, which is used as an indicator of the total number of input sources. When more than one source is detected, the program enters the `if` condition and prints the list of sources available for the user to choose. The user needs to enter the serial number corresponding to the desired input source, and then the value (voltage or current) will be found and assigned to a variable representing the input of type `complex double` (used later in the function). Another choice that needs to be made by the user is to nominate an output node. The program provides number ranging from 1 to the maximum number of nodes in the circuit for the user to choose from.

2) DC operating point

`get_standart_volt` obtains useful node voltages in preparation for small-signal parameters of certain components.

3) Small-signal analysis

This makes use of functions `reorganizedc`, `classify_comp`, `shortdcsource` to prepare the data needed. Then the program does the analysis with each required frequency. The output is a matrix of type complex double, and the program locates the voltage corresponding to the user's choice of the output node. At last, transform this into magnitude and phase, and store in the vectors, respectively.

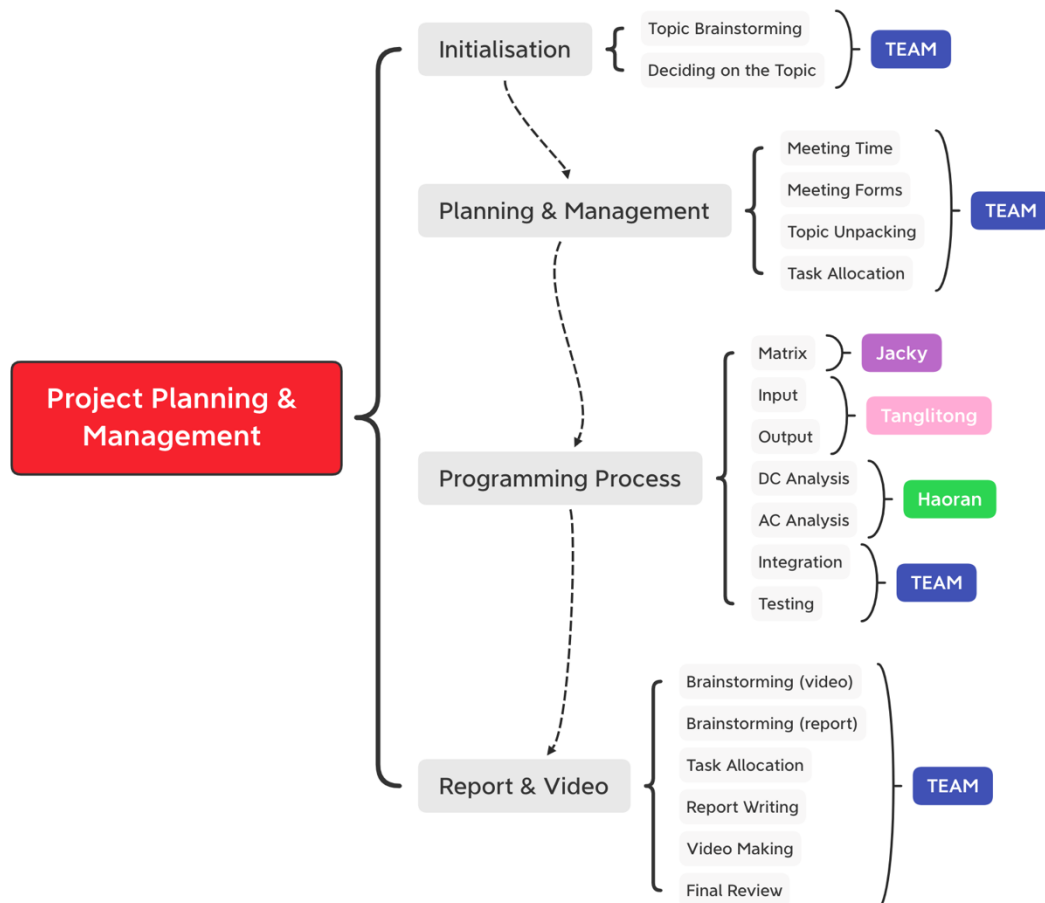
4) Output stage

The program asks the user to enter the name of the pre-created file that would store the output. The output has three values: frequency, magnitude, and phase (in degrees) of the transfer function calculated at that particular frequency, spaced using tab and written line by line according to the content of the three vectors.

Project Planning and Management

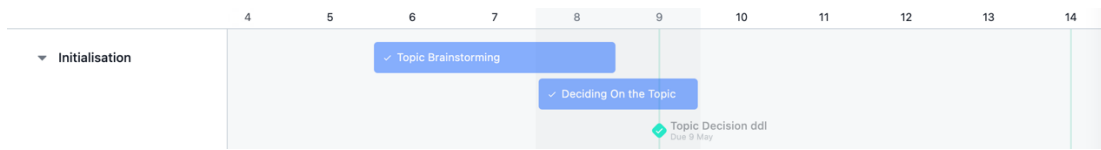
Overview

There are four stages in the overall planning of this end-of-year project: initialisation, planning & management, programming process and report & video.



(Figure 28: planning mindmap)

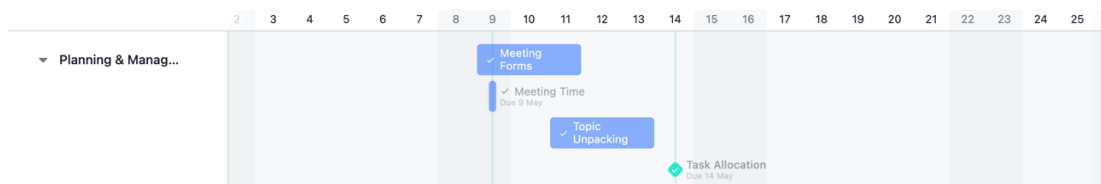
Initialisation



(Figure 29: initialization)

The whole process started on May 6th, a day after the project guideline has been released, and we came together as a group of three. After a brief first meeting, we all agreed that we should first look at all three topics and started generating ideas. Then the first milestone came on May 9th when we need to submit our chosen topic: circuit simulator.

Planning & Management

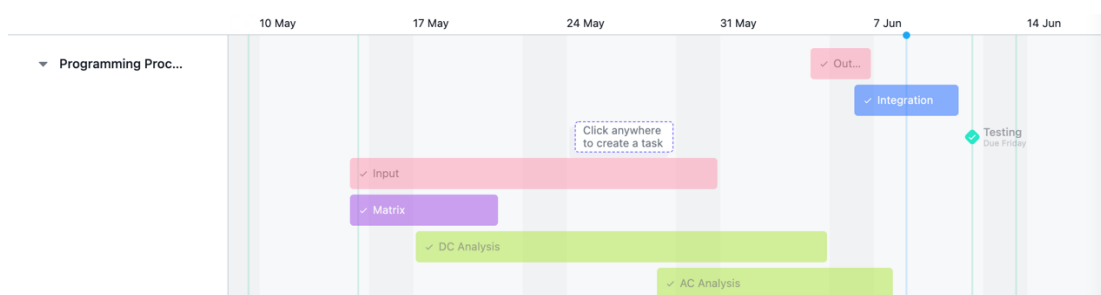


(Figure 30: planning & management)

We had two consecutive meetings on teams afterward to discuss our meeting structure and frequency. After a thorough discussion, we found that it was not easy to set an exact meeting time for future discussion as we could not predict the process. An idea came up that online meetings are not efficient enough, and since there are no traveling restrictions and we are all in China, we should come together in one place, increasing our efficiency and productivity. Therefore, we spent the next two days discussing plans for our destination, housing, etc. Finally, we have decided to go to Haoran's city, Chengdu, booked flight tickets, and rented an apartment so all three of us could work collaboratively and start discussing anytime. It may be argued that we wasted two days discussing, and the opportunity cost is that we could start unpacking the topic earlier.

Nevertheless, we as a team believe that beforehand planning before starting the project is better than planning in the middle of the project process. In the meantime, we started unpacking the topic from the given materials, separating the whole software package into five main stages, and assigning them to different team members (see design process for more details).

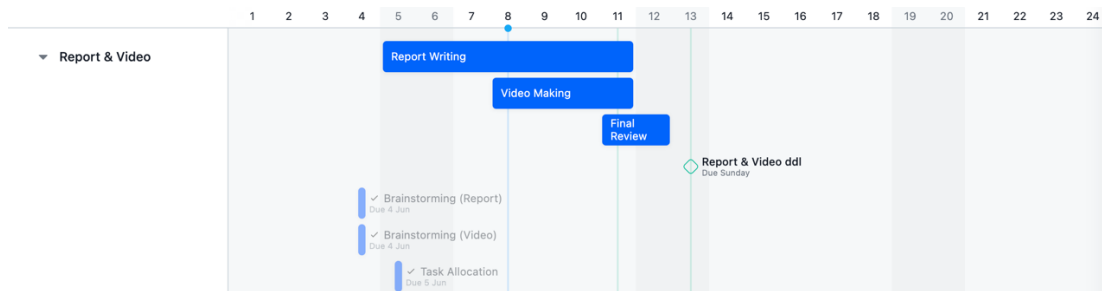
Programming Process



(Figure 31: programming process)

Matrix for Jacky (marked in purple), input and output for Tanglitong (marked in pink) and DC & AC analysis for Haoran (marked in green). Near the end of finishing our individual design, we spent three days integrating our parts and started testing for accuracy and efficiency, which is marked as another milestone of the project.

Report & Video



(Figure 32: report and video)

Due to time pressure, we did not wait to start the report and the video after we finished testing. Also, writing the report while testing helps to consolidate our memories and spotting errors. The tasks that are still in progress are in a brighter view; those finished are in a darker view.

Testing

In order to examine the overall functionality and accuracy of our software package, it is reasonable to conduct exhaustive testing started for each section separately and then combined.

Input

Multiplier testing:

```
int main(){
//testing for multiplier
vector<char> m;
m.push_back('p');
m.push_back('n');
m.push_back('u');
m.push_back('m');
m.push_back('k');
m.push_back('g');
m.push_back('G');
cout << "testing for multiplier" <<endl;
for(int i=0; i<m.size(); i++){
double v = 2;
multiplier(v, m[i]);
cout << v <<endl;
}
}
```

```
testing for multiplier
2e-12
2e-09
2e-06
0.002
2000
2e+06
2e+09
candydy@MacBook-Pro circuit_simulator6 %
```

(Figure 33)

(Figure 34)

Listing all the possibilities for the multiplier helps to examine whether the **multiplier** function produces the correct exact numerical value for the input figure from the netlist and the results matched the values we obtained from a calculator.

ReadInput testing:

```
int main(){
//testing for ReadInput
vector<string> in = ReadInput("example.txt");
cout << "testing for ReadInput" << endl;
for(int i=0; i<in.size(); i++){
cout << in[i] << endl;
}
}
```

```
testing for ReadInput
* A test circuit to demonstrate SPICE syntax
V1 N003 0 AC(1 0)
R1 N001 N003 1k
C1 N001 0 1u
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 35)

(Figure 36)

This test is designed to examine the accuracy of the reading process: does the printing of the vector string match the original netlist file? The printing matched the input netlist file.

getword testing:

```
int main(){
//testing for getword
string tmp = "R1 N001 N003 1k";
cout << "testing for getword" << endl;
for(int i=1; i<=4; i++){
    cout << getword(tmp, i) << endl;
}
}
```

```
testing for getword
R1
N001
N003
1k
```

(Figure 37)

(Figure 38)

This test is designed to examine the functionality of the `getword` function to see if it can print the correct text with the corresponding word position as the input. The sequence of printing matched the content in the line.

lastword testing:

```
int main(){
//testing for lastword
string tmp = "I am a girl!";
cout << "testing for lastword" << endl;
cout << lastword(tmp) << endl;
}
```

```
testing for lastword
girl
candydy@MacBook-Pro circuit simulator6 %
```

(Figure 39)

(Figure 40)

This test is used to check whether the `lastword` function can correctly identify and print the last word of a line. The result 'girl' matched with the last word of the given line.

ConvertFromString testing:

```
int main(){
//testing for ConvertFromString
string tmp = "2.345";
double d = ConvertFromString(tmp);
cout << "testing for ConvertFromString" << endl;
cout << d << endl;
}
```

```
testing for ConvertFromString
2.345
candydy@MacBook-Pro circuit simulator6 %
```

(Figure 41)

(Figure 42)

By defining the variable `d` as double limits the correct output to be a numerical value instead of a string value. 2.345 was clearly a number because a string would give errors.

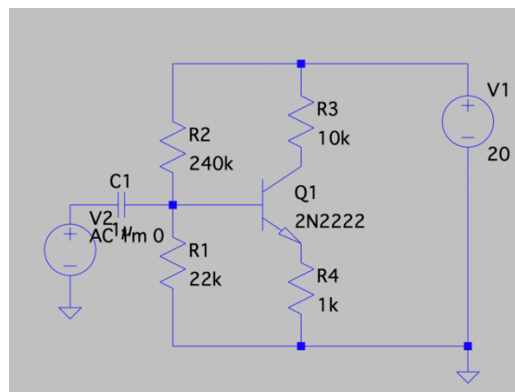
DC Analysis

First, we created a text file representing the netlist of a circuit.

```
input.txt
1 R1 0 N001 22k
2 R2 N001 N002 240k
3 R4 0 N003 1k
4 R3 N002 N004 10k
5 Q1 N004 N001 N003 NPN
6 V1 N002 0 20
7 C1 N005 N001 1u
8 V2 N005 0 AC(1m 0)
9 .ac dec 10 10 100
10 .end
```

(Figure 47)

The circuit for this test is shown in Figure 48



(Figure 48)

We wrote a test function to examine whether every matrix is built successfully and the DC analysis result.

```
int main(){
    vector<string> s;
    vector<general*> g;
    s = ReadInput("input.txt");
    int maxn = 0;
    setting(s,g,maxn);
    Eigen::MatrixXd test;
    test = build_guess_volt(g, maxn);
    std::cout << "result for build_guess_volt: " << std::endl;
    std::cout << test << std::endl;
    std::cout << std::endl;
    Eigen::MatrixXd tmp;
    tmp = build_iterate_matrix(g, maxn, test);
    std::cout << "result for build_iterate_matrix: " << std::endl;
    std::cout << tmp << std::endl;
    std::cout << std::endl;
    Eigen::MatrixXd final;
    final = get_standart_volt(g, maxn);
    std::cout << "result for get_standart_volt: " << std::endl;
    std::cout << final << std::endl;
}
```

(Figure 49)

As seen in the guess process result, we first considered the node voltage for the BJT. The BJT in this circuit has its collector connected to Node 4, its base connected to Node 1, and its emitter connected to Node 3. Since we replaced the BJT with two voltage sources, 0.7v and 1v, V_{be} is equal to $V_1 - V_3 = 0.7$, and V_{ce} is equal to $V_4 - V_3 = 1V$. Apart from that, although we have five nodes in total, the size of the guess column matrix is four as there is an AC source connecting to the ground and Node 5, which leads the short circuit in analysis. It shows that our short circuit method works. Therefore, it reaches our requirement for the guess function.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
result for build_guess_volt:
0.484484
    20
-0.215516
0.784484

```

(Figure 50)

The iterate matrix result is challenging to check in each cell. Here we check it by using the guess voltage vector. By writing out the anticipated value in each cell through the KCL equation mentioned before, we check that the result is also acceptable.

```

result for build_iterate_matrix:
0.00296624 -4.16667e-06 -0.00291662      0
0          1          0          0
-0.58624  0          0.58724 -0.000121124
0.583324  -0.0001    -0.583444  0.000220521

```

(Figure 51)

Now we started to check the Fvm column matrix. Using a similar process as we checked the iterate matrix, the result of fvm is shown in Figure 52. We firstly checked Node 2 as there is a constant DC voltage source connected to this node and the ground. Therefore, the voltage on that should be constant, and the value in Fvm on that row should be zero as well. Apart from that, we also do calculations for each row. Finally, the result also confirms our assumption.

```

Fvm
1.36228e-05
-3.55271e-15
-0.0148715
0.0126615

```

(Figure 52)

After checking all the functions inside the `get_standard_volt` function, we can finally start our Newton Raphson iteration test. The result is shown in Figure 53.

```

result for get_standart_volt:
1.58376
  20
0.953774
10.5097
  0
candydy@MacBook-Pro circuit simulator team2 %

```

(Figure 53)

```

--- Operating Point ---
V(n001):(n001) 1.59257 voltage
V(n001):(n002) 20 voltage
V(n002):(n004) 10.6323 voltage
V(n005):(n003) 0.941078 voltage
V(n003):(n005) 0 voltage
Ic(Q1): 0.00093677 device_current
Ib(Q1): 4.30785e-006 device_current
Ie(Q1): -0.000941078 device_current
I(C1): 1.59257e-018 device_current
I(R4): 0.000941078 device_current
I(R3): 0.00093677 device_current
I(R2): 7.66976e-005 device_current
I(R1): 7.23898e-005 device_current
I(V2): 1.59257e-018 device_current
I(V1): -0.00101347 device_current

```

(Figure 54)

As seen from the result we obtained, comparing with the expected result we obtained, it is noticeable that the result is almost close to the standard one but not that accurate. We noticed that this is probably caused by choice of the thermal voltage value. After changing it to 0.02585V, which is the standard thermal voltage in 300K environment [9], we get the final result in our program, which is shown in Figure 55. Comparing with the LTspice result after changing the temperature to 300K. It is much closer.

```

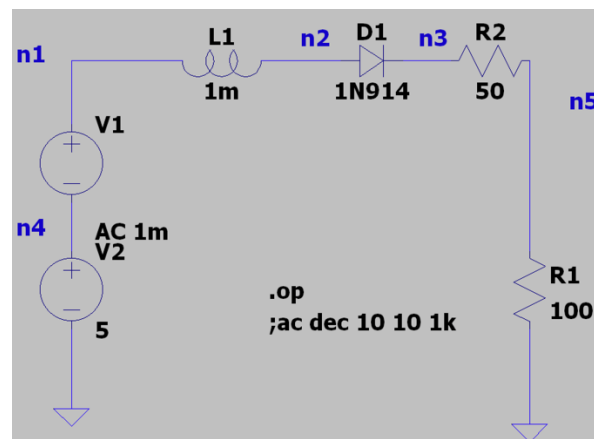
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
0.484484
  20
-0.215516
0.784484
0.00117295 -4.16667e-06 -0.00112333 0
  0 1 0 0
-0.225788 0 0.226788 -4.82366e-05
0.224665 -0.0001 -0.224713 0.000147997
result for get_standart_volt:
1.58566
  20
0.934816
10.6983
  0
candydy@MacBook-Pro circuit simulator team4 %

```

(Figure 55)

See Appendix 1 for the AC analysis test results of this circuit.

AC Analysis



(Figure 56: sample circuit)

The above shows a circuit in LTspice set up for testing, which is built by two voltage sources (one DC and one AC), two resistors, an inductor, and a diode of model 1N914 (same as the one used in our program). The corresponding netlist following the restrictions of our program is shown below, written in advance in a text file named “diode.txt.”

```
diode.txt
1  L1 N002 N001 1m
2  R1 N005 0 100
3  D1 N002 N003
4  V1 N001 N004 AC(1m 0)
5  V2 N004 0 5
6  R2 N005 N003 50
7  .ac dec 10 10 1k
8  .end
```

(Figure 57)

We always check the critical outcomes before using `find_final_sol` to write output into a file, so this is what we write in the ‘main.’ We first set the input source to be 10Hz and found the voltage node on each point.

```

int main(){
    std::vector<std::string> g;
    std::vector<general> b;
    g = ReadInput("diode.txt");
    int maxn = 0;
    setting(g,g,maxn);
    std::vector<general> testn;
    Eigen::MatrixXcd dcvolt;
    dcvolt = get_standar_volt(g, maxn);
    std::vector<double> tmp;
    for( int i = 0; i < dcvolt.rows(); i++){
        tmp.push_back(dcvolt(i,0));
    }
    setting(g, testn, maxn);
    std::vector<general> test = reorganize(testn, maxn, tmp);
    int num_de = 0;
    std::vector<Resistor> r;
    std::vector<Q> b;
    std::vector<mosfet> m;
    std::vector<diode> d;
    std::vector<volsrc> v;
    std::vector<cursrc> a;
    classify.comp(test, r, b, m, d, v, a);
    std::vector<shortcircuit> test3;
    std::vector<general> test2 = shortcsource(test,test3 ,v);
    Eigen::MatrixXcd testcol = SSEM(test2, maxn, 100, dcvolt, test3);
    Eigen::MatrixXcd test5 = build_acb(test2, maxn, test3);
    std::cout << "result for build_acb: " << std::endl;
    std::cout << test5 << std::endl;
    std::cout << std::endl;
    std::cout << "result for SSEM: " << std::endl;
    std::cout << testcol << std::endl;
    std::cout << std::endl;
    Eigen::MatrixXcd result = testcol.colPivHouseholderQR().solve(test5);
    Eigen::MatrixXcd final = recover_complex_circuit(result, test3);
    std::cout << "result for recover_complex_circuit: " << std::endl;
    std::cout << final << std::endl;
}

```

(Figure 58)

Following the same procedure, as we checked for DC analysis, we also checked all every function before going to the final testing. The checking process is more straightforward as there is no iteration involved. We started by checking the `col_b` function built by the `build_acb` function.

```

result for build_acb:
(0.001,0)
(0,0)
(0,0)
(0,0)
(0,0)

```

(Figure 59)

Since there is only one AC voltage source with an amplitude of 1m connecting to the row, for row one, it should be the amplitude of this source, and it is correct. Now we come to the SSEM, the result is shown in Figure 60.

```

result for SSEM:
(1,0) (0,0) (0,0) (0,0)
(0,1.59155) (0.852764,-1.59155) (-0.852764,0) (0,0)
(0,0) (-0.852764,0) (0.872764,0) (-0.02,0)
(0,0) (0,0) (-0.02,0) (0.03,0)

```

(Figure 60)

Here we checked row by row according to the way we build it. The result matches our anticipation. Having checked all the two main functions required, we start to check the final one. It is shown in Figure 61.

```

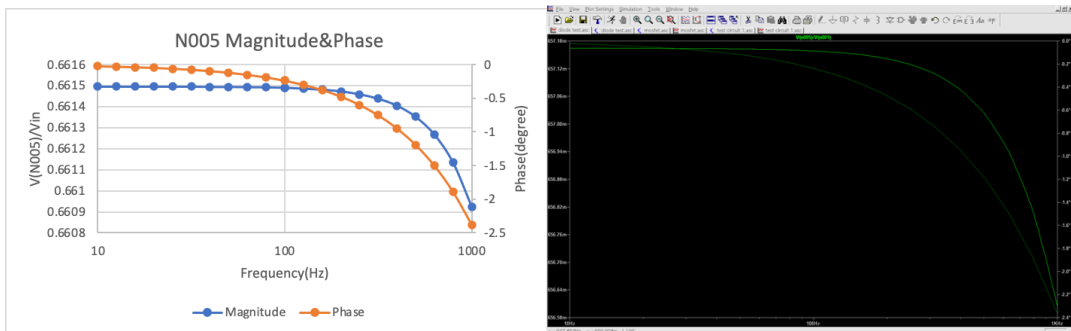
result for recover_complex_circuit:
(0.001,4.69362e-20)
(0.000999983,-4.15623e-06)
(0.000992226,-4.12399e-06)
(0,0)
(0.000661484,-2.74932e-06)
candydy@MacBook-Pro circuit simulator team4 %

```

(Figure 61)

Clearly, the results are free of errors after comparing with the LT SPICE result, so we can do the final testing for all nodes. The nominated output node is N001. Node 1 connects the positive side of the input source, so the transfer function is of magnitude 1 and phase near 0.

We used Excel to plot the magnitude and phase on one diagram as shown below, making a straightforward comparison with the result on LTspice. Note that the x-axis is logarithmic, but the y-axis is linear.



(Figure 62: test result 1)

(Figure 63: test result 2)

As shown by the result, the two simulation methods show similar trends. We also carefully checked the value for each node, showing comparable outcomes to the value we obtained from LT SPICE. The error is approximately within 0.01, indicating that the result is acceptable.

See Appendix 3 for the DC analysis test results of this circuit.

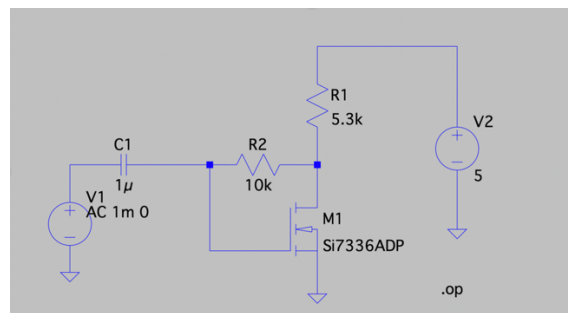
Total Testing

In order to test all the components, we built another circuit to test the MOSFET performance.

The input text is shown in Figure 64, and the circuit is shown in Figure 65.

```
1 R1 N001 N004 5.3k
2 R2 N004 N003 10k
3 C1 N003 N002 1u
4 V1 N002 0 AC(1m 0)
5 V2 N001 0 5
6 M1 N004 N003 0 NMOS
7 .ac dec 10 10 100
8 .end
```

(Figure 64)



(Figure 65)

We first did DC analysis:

```
Operating Bias Point Solution:
V(n001)          5  voltage
V(n004)         2.49989  voltage
V(n003)         2.49989  voltage
V(n002)          0  voltage
```

(Figure 66)

```
--- Operating Point ---
V(n001) :          5          voltage
V(n004) :        3.17256      voltage
V(n003) :        3.17256      voltage
V(n002) :          0          voltage
```

(Figure 67)

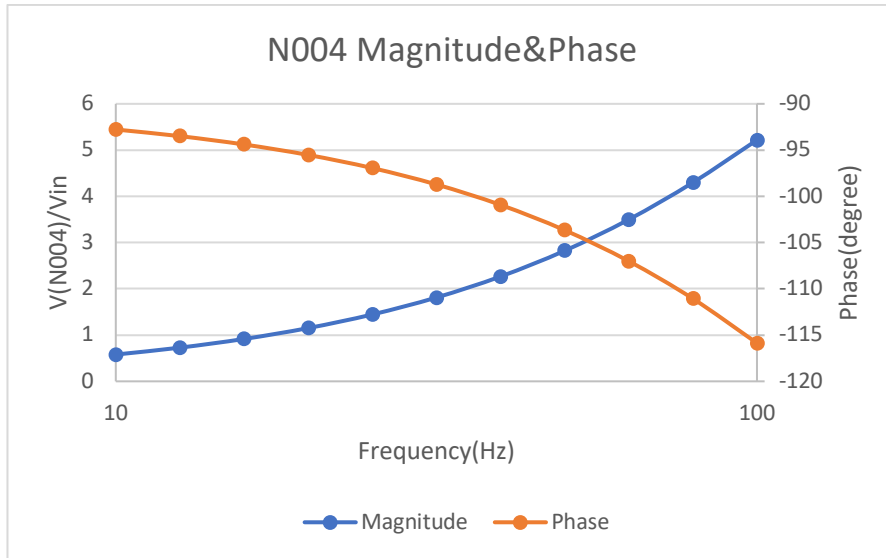
There is a significant error in the quiescent voltage for these nodes. After checking the entire program carefully, we discovered that this error is caused by the selection of conductivity parameter. This parameter depends on the width and length of the mosfet. After defining the values of them, we match this with the parameter in LTspice. The result is much better. However, there is still a small error exists. The further analysis is in the evaluation.

```
result for get_standart_volt:
5
0
3.09673
3.09673
```

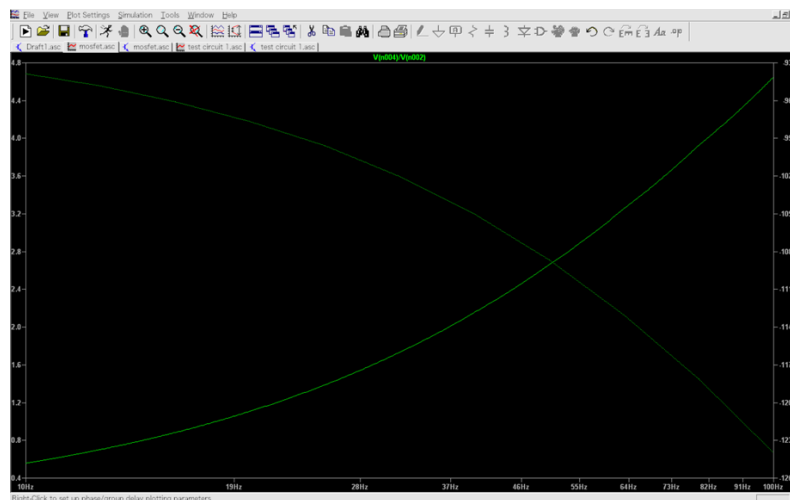
(Figure 68)

After solving this problem. We did AC analysis as well and chose Node 4 to find the output.

The rest of the nodes are listed in Appendix 2.



(Figure 69)



(Figure 70)

By comparing the trend, we can conclude that our simulation is very accurate. But in lower frequency, our result is slightly larger than the standard voltage. Due to time limitation, we temporarily ignored this and decided to investigate this in the future.

Evaluation

Before we started coding, we spent hours and hours finding an appropriate external algebra library that is best for our package, and we finally chose to use the Eigen Library. The reasons are justified as follows:

- ✚ It enables us to construct two types of matrix, `Eigen::MatrixXd` for double type data matrix and `Eigen::MatrixXcd` for `complex<double>` type data.
- ✚ We can access or change the content of the matrix easily.
- ✚ It has the essential matrix calculation function. For example, we can find the inverse of the matrix directly by using `A.inverse()`, and we can solve the equation $Ax=b$ through `A.colPivHouseholderQr.solve(b)`

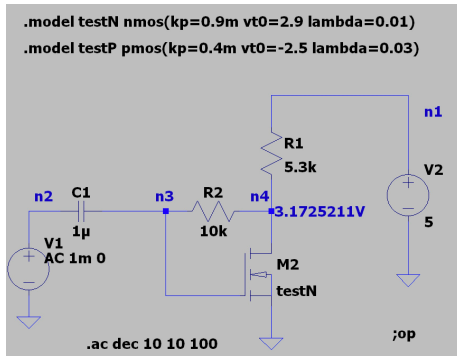
Even though the simulation results of our software packages matched the LT SPICE results, it still possesses some critical weaknesses that need to be addressed:

- ✚ A few methods we utilized to construct the matrix may be redundant and the program could be optimized further. In this program, since every matrix has some unique characteristics, we build a lot of matrices to meet the requirements of each function. However, some parts of it are redundant. For the construction of the matrix, it seems that the way we built it is not entirely efficient. For most of the cases, we built it row by row. For each row, we have a column loop to fill the data into each cell. Inside this loop, we even designed a third loop to go through all the components in the input vector and find out the components that have the specific two nodes which is suitable to put into a particular position. This whole process can be quite redundant. Apart from that, since we need to find the ground node conductance that should be added into the

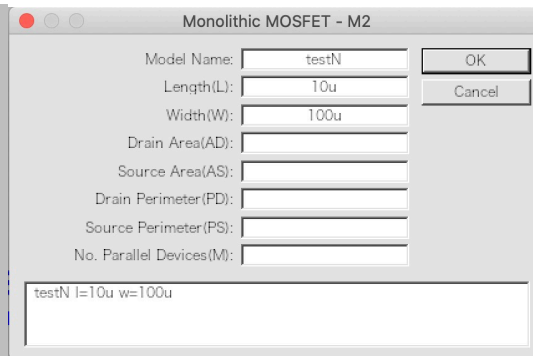
`tot_cond` as mentioned before, theoretically, we only need one loop for each row to search for the component that contains the ground node. However, due to this three-loop configuration, we have to get the `ground_cond` many times and only take the value of the final one. This could be quite wasteful.

- ✚ It is suspected that the state of BJTs and MOSFETs cannot be changed during the process of Newton Raphson method and it is easy to diverge during state changes.
- ✚ The program has limited features since it only performs dc and ac small-signal analysis, wherein LT SPICE, simulation commands such as transient, noise, and dc sweep can be conducted as well.
- ✚ Even though the software package successfully serves the purpose of circuit simulation. The functional requirements of the input netlist file could be laborious and complicated for circuits with great complexity because it requires the user to look at the circuit diagram and identify components and the nodes it is connected to. Also, there is no visual representation of the circuit in any form. If more time is given, we can come up with a human-friendlier user interface that allows the user to visualize the circuit and the simulation process
- ✚ Because there is a variety of models for the non-linear components (BJTs, MOSFETs, and Diodes) and the fact that their parameters have to be pre-defined first, we have chosen a specific model for each device (e.g., 2N222 for NPN BJT). Therefore, it limits the range of our simulation program to specific models, reducing the program's effectiveness. Also, we failed to find the value for channel width and channel length for

MOSFETs when calculating the conductivity parameter after spending an enormous amount of time digging into the internet. We first chose to approximated values, but the results diverged heavily from the LT SPICE simulation.



(Figure 71)



(Figure 72)

Therefore, we created a new model in LT SPICE for MOSFET with parameters we found on the internet and used that in our program. The results matched the LT SPICE simulation.

✚ Lastly, there are still some bugs in the program that we have not solved yet. It happens when the non-linear components are in a state different from what we assume it to be. This makes our Newton Raphson method diverge sometimes. Our assumption for this problem is that Newton Raphson might work only when those components are in the same state. In the LT SPICE simulation, the op analysis result is also confusing. It does not meet Ebers–Moll equations we found for BJTs in saturation state. Because of the time limit, we have to leave this for further investigation after the submission of our work.

Despite the weaknesses, our design does have outstanding strengths:

- ✚ We carefully considered all the components provided and designed parameter calculations.
- ✚ We have taken all the different connections for each component into account. For example, whether a voltage source has one or neither node connecting to the ground had been thoroughly considered because ways of connections could have an impact on the size and the structure of the matrix.
- ✚ We also came up with an intelligent algorithm to handle the short circuit condition in which some of the components might be short-circuited under DC and AC analysis. Although it could be challenging to deal with the short circuit situation, one valid and straightforward shortcut is to use the infinite value (probably the maximum of double type data) for the conductance between the two nodes. However, higher precision is the ultimate goal we wanted to achieve, and therefore we designed a more complex but more precise way to do it correctly. We wrote several functions to change the node. For example, we created a function to delete the component connected to the two nodes which are short-circuited. Besides, this function could also automatically keep one node and makes all the components connected to the other node connect to it instead.
- ✚ We have done a few measures to improve the accuracy of our program. Higher accuracy comes from better equations. For MOSFETs, we obtained different equations involving considerations of early voltage for nMOS and pMOS in 3 operating modes from the lecture notes. For BJTs, we also considered two models: NPN and PNP. We could simply use a single general equation given in the lecture for all four operating modes

for ease of calculation. However, the equation does not take early voltage into consideration that would affect the current calculations in a significant voltage situation. As a result, we chose to apply the equation that includes early voltage as a variable for the active mode and conducted research and discovered the unapproximated Ebers-Moll equations for the rest three operating modes. Apart from that, in terms of the iteration algorithm, we designed in a way such that the iteration will keep running until the difference between the two voltage vectors is less than 0.0001, up to four decimal places.

✚ The use of `build_guess_voltage` improves the program's efficiency by empowering it with the ability to reduce the number of loops to get the converged result in Newton Raphson. We replaced BJTs with two voltages, one is 0.7 volt (connecting to base and emitter), and the other is 1 volt (connecting to collector and emitter). Similarly, we put a 0.7v voltage source to replace the diode. For MOSFETs, we simply replaced them with two voltage sources as well. But the value of the voltage source depends on the V_T of the MOSFET. Since the active state and saturation state is the most general case for BJTs and MOSFETs respectively, we started the assumption by guessing they are in these states.

✚ In terms of the reading of the input, we apply some tricks for simplification. First, we defined a general public class that contains the nodes and a few general properties (I_b , I_e , I_c). Then we defined a class for each component that contains some of its unique properties. Because of the law of inheritance, the class has the properties of the general class. After reading from the input file, we used a pointer to store all the components into the general pointer type vector. If we need to use some of the unique properties of

some specific components later, we could use dynamic cast to change the general type class to its own class, and therefore we could assess all the functions inside the class. Apart from that, this method gives the program the ability to easily change the component inside that vector. For example, if we want to replace BJTs with two voltage sources, we only need to add the pointer of the new voltage sources to the vector, and no further change needs to be made when inputting the vector into the matrix. It is exceptionally useful in short circuit functions. We could use the change node function to change the nodes of the components and add or delete components by changing this vector.

Power consumption and Energy needed

✚ The simulation time mainly depends on the number of loops for the Newton Raphson method to find the quiescent operating point in DC analysis. If the initial solution or voltage vector we guessed is closer to the actual value, the calculating time would be shorter. Since we do not know how to decide the exact state of the non-linear components, we assumed them in a particular state. For example, we assume BJTs are in the active state and MOSFETs in the saturation state, and Diodes are forward conducting. Therefore, it would probably take a longer processing time and even some mistakes if they are in different states. Apart from that, since we used many loops to build different matrices, they would make the program more complex and consume more energy to run. Lastly, to be precise, we made the program calculate the result as accurately as possible. For instance, we used the double type data and kept many constants to around 4 to 5 decimals, which could significantly increase the computing consumption. According to our measurements, it took around 7 to 8 seconds for the program to run with circuits composed of less than 4 to 5 components,

Conclusion

Time flies in this incredible one-month journey. Through this design process, we have adapted the ability to search beyond textbooks and lectures. The circuit simulator project we have given birth to is not only a milestone for our first-year academic studies at Imperial College London but also a landmark for our collaboratively working, independent learning, and critical thinking. Even though our final design can never be as successful as the students had accomplished in the 1970s, it is a massive victory for us at this stage of studying. We are more than thankful for this opportunity that has opened the door of Engineering for us. With this tremendous experience as a firm basis, our future studies and designs can only be better.

Appendix 1

This section is the AC analysis test for the BJT circuit (Figure 48). The test procedure is the same as the AC analysis for the circuit that has a diode and an inductor (Figure 56).

Frequency: 10Hz

```
result for build_acb:
(0,0)
(0,0)
(0,0)
(0.001,0)

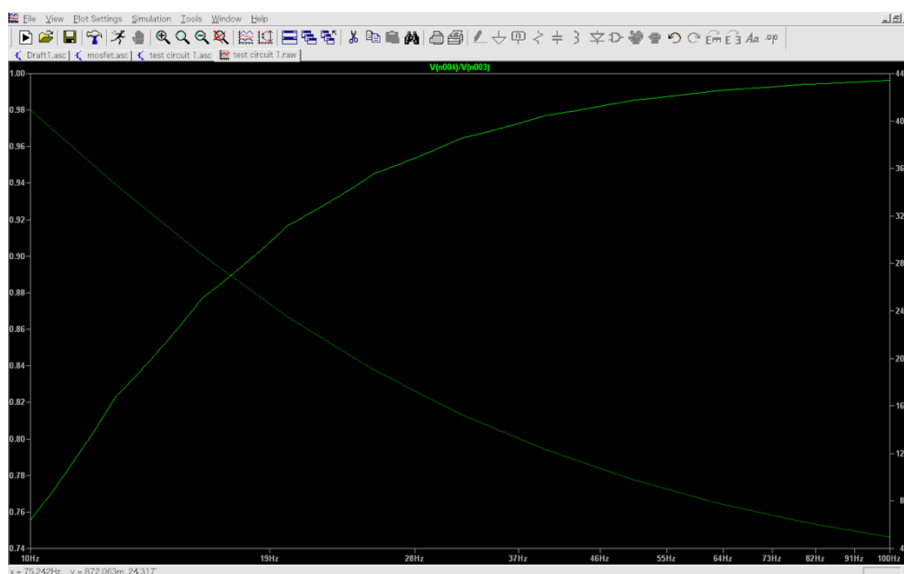
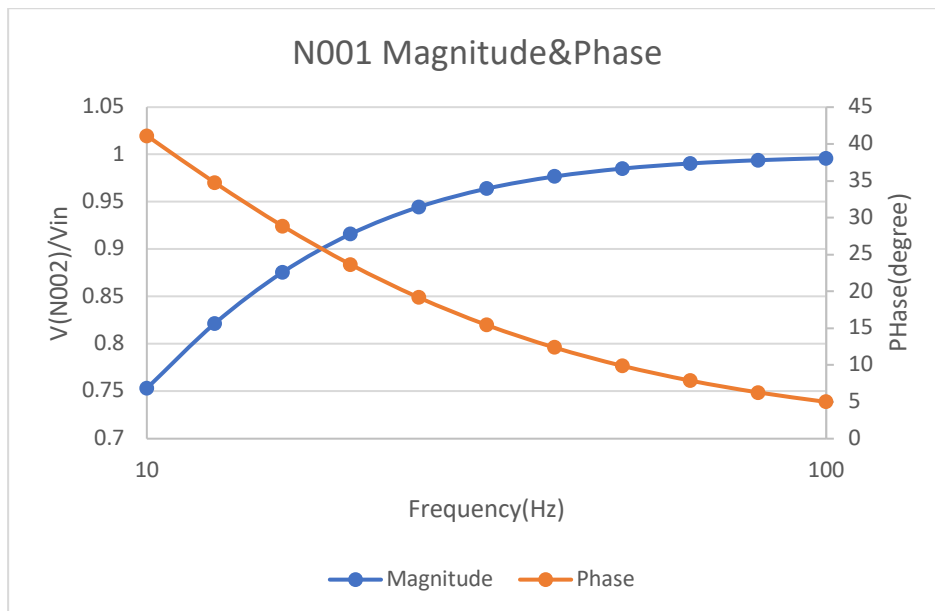
result for SSEM:
(0.000239427,6.28319e-05)      (-0.000189806,0)      (0,0)      (0,-6.28319e-05)
(-0.038151,0)      (0.0391589,0)      (-7.90858e-06,0)      (0,0)
(0.0379612,0)      (-0.0379691,0)      (0.000107909,0)      (0,0)
(0,0)      (0,0)      (0,0)      (1,0)

result for recover_complex_circuit:
(0.000567271,0.000495454)
(0,0)
(0.000551562,0.000481734)
(-0.0054858,-0.0047913)
(0.001,9.1111e-24)
candydy@MacBook-Pro circuit simulator team2 %
```

AC analysis result:

Nominated output node: N001

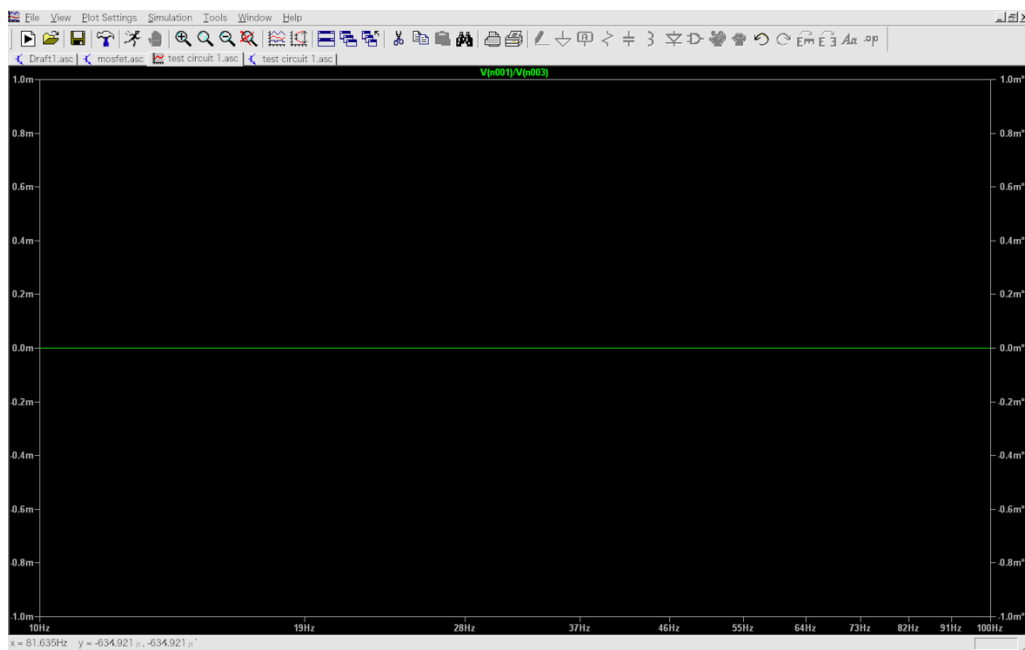
```
outnode1.txt
1 frequency magnitude phase
2 10 0.753174 41.134
3 12.5893 0.821631 34.7516
4 15.8489 0.875817 28.8582
5 19.9526 0.916078 23.6408
6 25.1189 0.944532 19.1729
7 31.6228 0.963911 15.4398
8 39.8107 0.97677 12.374
9 50.1187 0.985153 9.88544
10 63.0957 0.990555 7.88106
11 79.4328 0.994009 6.27472
12 100 0.996208 4.99154
13
```



Nominated output node: N002

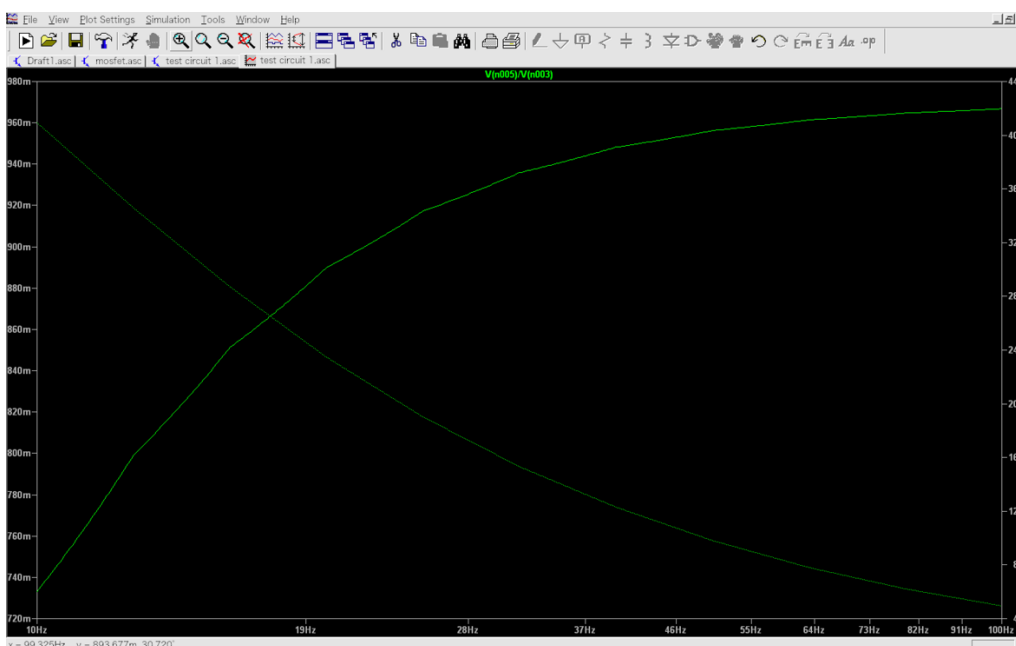
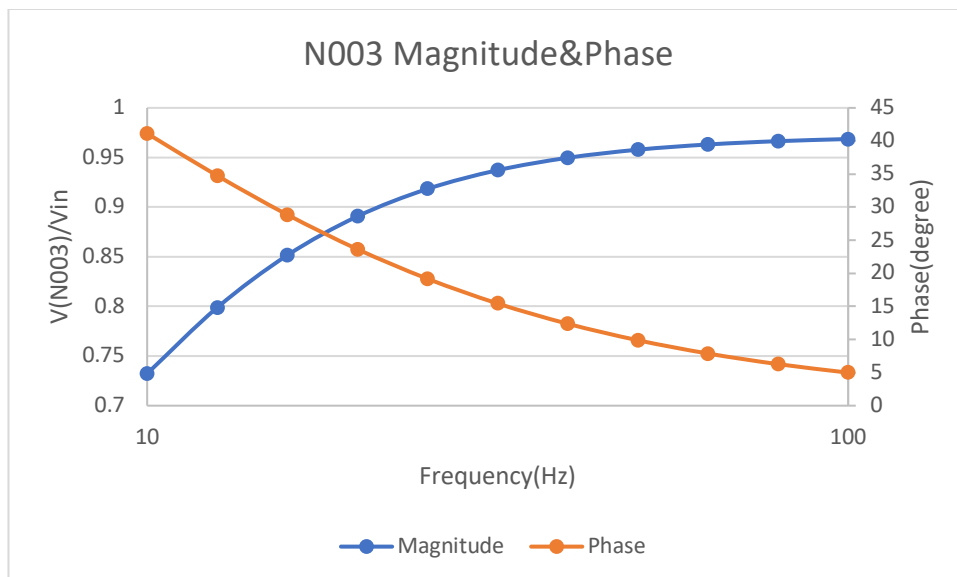
```
outnode2.txt
```

1	frequency	magnitude	phase
2	10	0	0
3	12.5893	0	0
4	15.8489	0	0
5	19.9526	0	0
6	25.1189	0	0
7	31.6228	0	0
8	39.8107	0	0
9	50.1187	0	0
10	63.0957	0	0
11	79.4328	0	0
12	100	0	0
13			



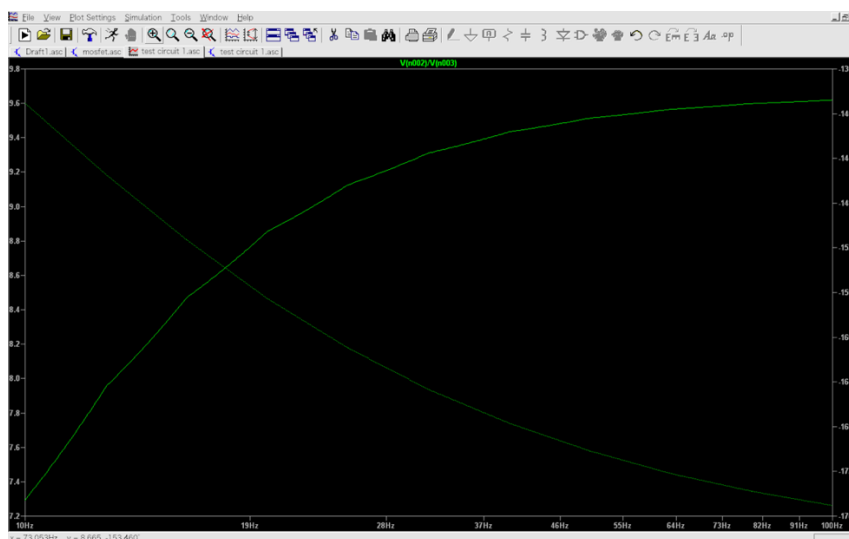
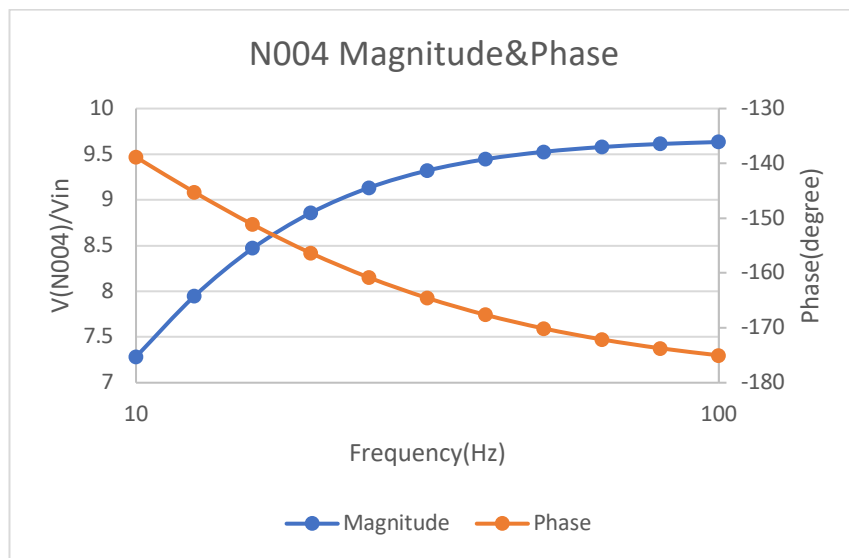
Nominated output node: N003

```
outnode3.txt
1  frequency  magnitude  phase
2  10  0.732317  41.134
3  12.5893  0.798878  34.7516
4  15.8489  0.851564  28.8582
5  19.9526  0.89071  23.6408
6  25.1189  0.918376  19.1729
7  31.6228  0.937218  15.4398
8  39.8107  0.949721  12.374
9  50.1187  0.957872  9.88544
10 63.0957  0.963124  7.88106
11 79.4328  0.966483  6.27472
12 100 0.968621  4.99154
13
```



Nominated output node: N004

```
outnode4.txt
1 frequency magnitude phase
2 10 7.28358 -138.866
3 12.5893 7.9456 -145.248
4 15.8489 8.4696 -151.142
5 19.9526 8.85895 -156.359
6 25.1189 9.13411 -160.827
7 31.6228 9.32152 -164.56
8 39.8107 9.44587 -167.626
9 50.1187 9.52694 -170.115
10 63.0957 9.57918 -172.119
11 79.4328 9.61259 -173.725
12 100 9.63384 -175.008
13
```



It is reasonable to conclude that our simulation results are very close to the LT SPICE standard results.

Appendix 2

This section is complementary to the MOSEFT circuit (Figure 67) in the total testing.

```
mosfet.txt
1 R1 N001 N004 5.3k
2 R2 N004 N003 10k
3 C1 N003 N002 1u
4 V1 N002 0 AC(1m 0)
5 V2 N001 0 5
6 M1 N004 N003 0 NMOS
7 .ac dec 10 10 100
8 .end
```

We check the AC analysis part by test the circuit when the frequency is 100Hz

```
result for build_acb:
(0.001,0)
(0,0)
(0,0)
```

```
result for SSEM:
(1,0) (0,0) (0,0)
(0,-0.000628319) (0.0001,0.000628319) (-0.0001,0)
(0,0) (0.00349554,0) (0.00029227,0)
```

```
result for recover_complex_circuit:
(0,0)
(0.001,0)
(0.000190314,0.000392549)
(-0.00227614,-0.00469486)
candydy@MacBook-Pro circuit simulator team4 %
```

AC analysis Results:

Since some of the node has constant magnitude, we do not plot the graph for them. But we still confirm the magnitude and phase are correct in comparison with LTspice.

Nominated output: N001

```
mosfetn1.txt
1 frequency magnitude phase
2 10 0 0
3 12.5893 0 0
4 15.8489 0 0
5 19.9526 0 0
6 25.1189 0 0
7 31.6228 0 0
8 39.8107 0 0
9 50.1187 0 0
10 63.0957 0 0
11 79.4328 0 0
12 100 0 0
13
```

Nominated output: N002

```
mosfetn2.txt
```

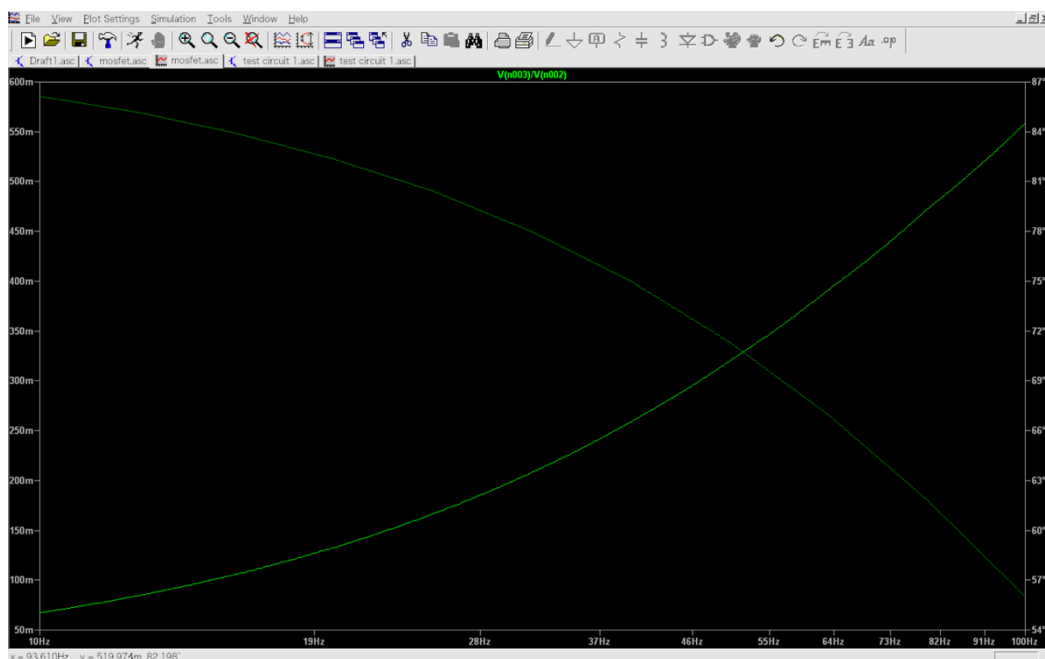
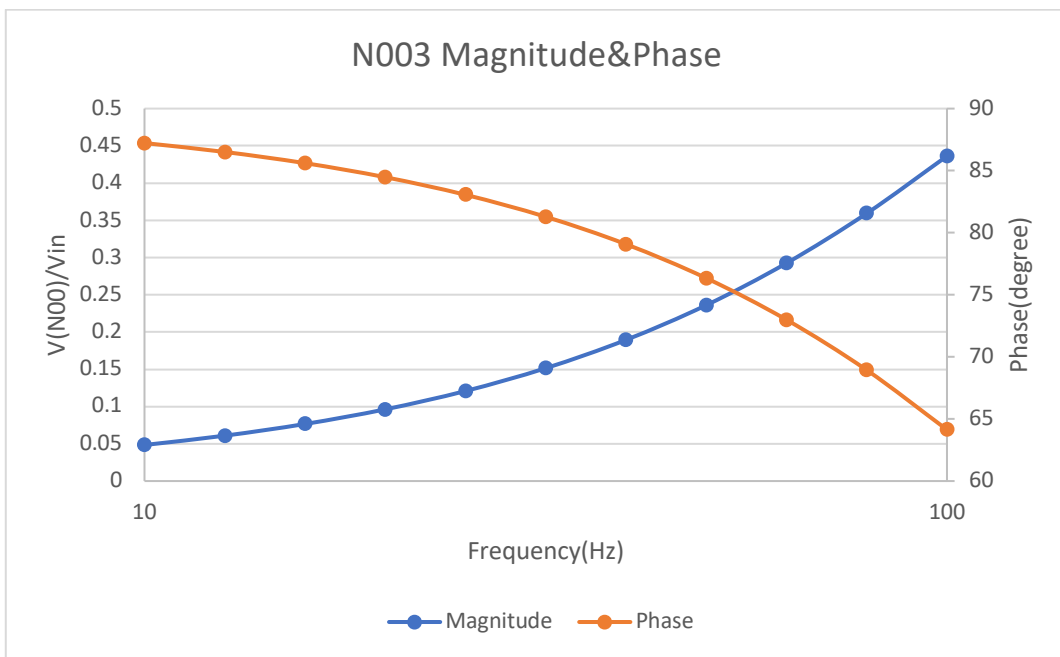
1	frequency	magnitude	phase
2	10 1	2.8927e-24	
3	12.5893 1	2.8927e-24	
4	15.8489 1	-3.47124e-23	
5	19.9526 1	2.31416e-23	
6	25.1189 1	6.94247e-23	
7	31.6228 1	-4.62832e-23	
8	39.8107 1	9.25663e-23	
9	50.1187 1	-5.55398e-22	
10	63.0957 1	-3.70265e-22	
11	79.4328 1	7.40531e-22	
12	100 1	0	
13			

Nominated output: N003

```

mosfetn3.txt
1  frequency  magnitude  phase
2  10  0.0484247  87.2244
3  12.5893  0.0609213  86.5073
4  15.8489  0.0766123  85.6061
5  19.9526  0.0962841  84.4748
6  25.1189  0.120887  83.0567
7  31.6228  0.151542  81.2837
8  39.8107  0.189511  79.0757
9  50.1187  0.236113  76.3427
10 63.0957  0.292518  72.9912
11 79.4328  0.359375  68.9382
12 100 0.43625  64.1352
13

```



Appendix 3

This section is complementary to the circuit composed of a diode and an inductor (Figure 54)
DC analysis result:

```
5
5
4.39746
5
2.93164
```

* C:\users\crossover\Desktop\My Mac Desktop\LTSpice\tutorial\diode

--- Operating Point ---

```
V(n002) :      4.99997      voltage
V(n001) :      5          voltage
V(n005) :      2.83199      voltage
V(n003) :      4.24799      voltage
V(n004) :      5          voltage
I(D1) :      0.0283199      device_current
I(L1) :      -0.0283199      device_current
I(R2) :      -0.0283199      device_current
I(R1) :      0.0283199      device_current
I(V2) :      -0.0283199      device_current
I(V1) :      -0.0283199      device_current
```

AC analysis result:

Nominated output node: N001

```
Frequency magnitude phase
1 10 1 -2.02422e-15
3 12.5893 1 -7.0296e-15
4 15.8489 1 -6.44472e-15
5 19.9526 1 -6.80597e-15
6 25.1189 1 -1.63226e-14
7 31.6228 1 -5.20197e-15
8 39.8107 1 -2.23682e-15
9 50.1187 1 4.04149e-15
10 63.0957 1 -6.73135e-15
11 79.4328 1 8.79737e-15
12 100 1 2.68924e-15
13 100 1 2.68924e-15
14 125.893 1 1.17916e-14
15 158.489 1 -3.77024e-15
16 199.526 1 -2.56901e-15
17 251.189 1 1.11203e-14
18 316.228 1 -6.87014e-15
19 398.107 1 6.04666e-15
20 501.187 1 -2.63587e-15
21 630.957 1 5.72699e-16
22 794.328 1 1.00935e-14
23 1000 1 2.24223e-15
24
```

Nominated output node: N002

```

P: dmod2.in
1 frequency magnitude phase
2 10 1 -0.8238138
3 12.5893 1 -0.8299798
4 15.8489 1 -0.8377424
5 19.9526 1 -0.8475148
6 25.1189 0.999999 -0.8588176
7 31.6228 0.999999 -0.8753859
8 39.8107 0.999999 -0.8948845
9 50.1187 0.999998 -0.119352
10 63.0957 0.999997 -0.150255
11 79.4328 0.999995 -0.189159
12 100 0.999991 -0.238137
13 125.893 0.999986 -0.299796
14 158.489 0.999978 -0.377418
15 199.526 0.999966 -0.475137
16 251.189 0.999946 -0.598155
17 316.228 0.999914 -0.753816
18 398.107 0.999863 -0.947959
19 501.187 0.999783 -1.19335
20 630.957 0.999656 -1.50221
21 794.328 0.999455 -1.89091
22 1000 0.999137 -2.38001
23
24

```

Nominated output node: N003

```

P: dmod3.in
1 frequency magnitude phase
2 10 0.992243 -0.8238138
3 12.5893 0.992243 -0.8299798
4 15.8489 0.992243 -0.8377424
5 19.9526 0.992243 -0.8475148
6 25.1189 0.992242 -0.8588176
7 31.6228 0.992242 -0.8753859
8 39.8107 0.992242 -0.8948845
9 50.1187 0.992241 -0.119352
10 63.0957 0.99224 -0.150255
11 79.4328 0.992238 -0.189159
12 100 0.992234 -0.238137
13 125.893 0.992234 -0.299796
14 158.489 0.992229 -0.377418
15 199.526 0.992221 -0.475137
16 251.189 0.992189 -0.598155
17 316.228 0.992157 -0.753816
18 398.107 0.992107 -0.947959
19 501.187 0.992028 -1.19335
20 630.957 0.991902 -1.50221
21 794.328 0.991783 -1.89091
22 1000 0.991387 -2.38001
23
24

```

Nominated output node: N004

```

P: dmod4.in
1 frequency magnitude phase
2 10 0 0
3 12.5893 0 0
4 15.8489 0 0
5 19.9526 0 0
6 25.1189 0 0
7 31.6228 0 0
8 39.8107 0 0
9 50.1187 0 0
10 63.0957 0 0
11 79.4328 0 0
12 100 0 0
13 125.893 0 0
14 158.489 0 0
15 199.526 0 0
16 251.189 0 0
17 316.228 0 0
18 398.107 0 0
19 501.187 0 0
20 630.957 0 0
21 794.328 0 0
22 1000 0 0
23
24

```

This connects the DC source, which is short circuited in small signal AC analysis.

References

- [1] V. Andrei. (2011). "Shaping the History of SPICE," in *IEEE solid state circuits magazine*, IEEE, 2011, Vol.3(2), pp. 36-39.
- [2] *EE1 Project 2021-Simulation File Specification*, May 2021.
- [3] Ed Stott, Esther Perea. (May 2021), *EEE1 Project 2021*.
- [4] Adel S. Sedra, Kenneth C. Smith, *Microelectronic Circuits*, 7th ed. Oxon.
- [5] WIKIPEDIA (2021, Jun. 1). *Bipolar junction transistor* [online]. Available: https://en.wikipedia.org/wiki/Bipolar_junction_transistor#Large-signal_models
- [6] CSDN (2015, Aug. 26). *The uses of istringstream* [online]. Available: https://blog.csdn.net/u010025211/article/details/48007847?utm_medium=distribute.pc_relevant.none-task-blog-baidujs&utm_term=0&spm=1001.2101.3001.4242
- [7] CSDN (2014, Jun. 7). *Conversion between string and double in C++* [online]. Available: https://blog.csdn.net/weixin_30478923/article/details/99396624?ops_request_misc=&request_id=&biz_id=102&utm_term=c++string
- [8] Fourier.eng.hmc.edu. (2015, Feb. 12). *Newton-Raphson method* [online]. Available: <http://fourier.eng.hmc.edu/e176/lectures/NM/node21.html>
- [9] WIKIPEDIA (2021, Jun. 8). *Boltzmann constant* [online]. Available: https://en.wikipedia.org/wiki/Boltzmann_constant