Team name: AVS

Engineering Design Project: Circuit Simulator

Report

Abstract

AVS is a circuit simulator, written in C++, which uses Modified Nodal Analysis (MNA) to perform both DC Operating Point calculations and AC frequency sweeps on a circuit given by the user in the form of a SPICE Netlist.

Vladimir Marinov,
Alexandra Neagu and
Scott Vandenberghe

Word count: 10,007

Contents

Software requirements

This program is intended to be a general-purpose command-line Circuit Simulator capable of performing AC sweep, and quiescent operating point simulations on any circuit defined in the popular SPICE Netlist format [1]. The simulation output is stored in a file, which allows the user to easily plot, inspect, and process the results.

The simulator was created to be easy-to-use, efficient, reliable, and portable across all major operating systems (Windows, MacOS, Linux). It provides useful and readable error messages, to assist the user in fixing issues with input files and circuits.

Input

The input is taken from a file which is based on the industry-standard SPICE Netlist format. Each line of the input can be any one of three options:

1. A description of a component in the circuit.
2. A directive that defines simulation settings and parameters. (lines starting with .).
3. A comment, which is completely ignored by the program. (lines starting with *).

The syntax of a component line is:

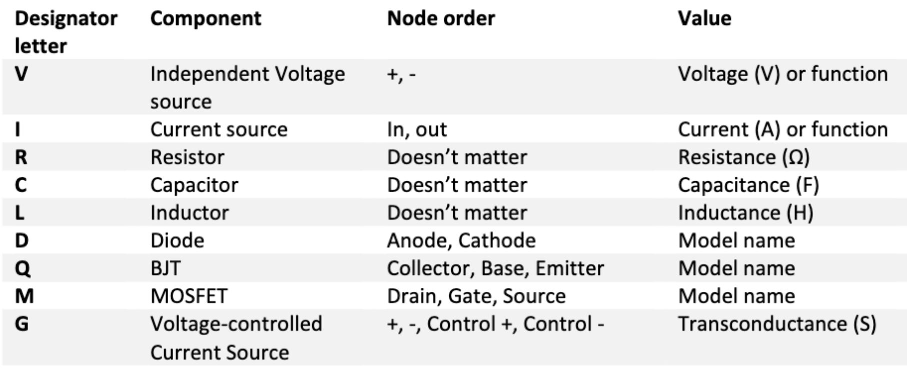
```
<designator> <node0> <node1> [<node2> [<node3>]] <value>
```

e.g., `R1 N001 0 10k` or `G1 N002 N003 N001 0 10u`

The first letter of the designator specifies the component type (as shown in Figure 1). The designator is also used as a unique identifier for every component in the circuit. Nodes 0 through 3 specify where in the circuit the component is connected (<node2> and <node3> are optional, they are used only for three-terminal or four-terminal devices). The order in which the nodes are expected is given in Figure 1. The last term of the component line is a value. Different components admit different types of values:

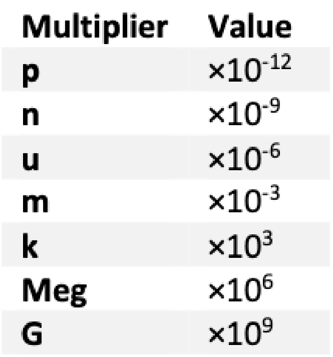
- Constant values: A real number, optionally followed by a multiplier (cf. Figure 2). This value type is expected for passive components and is also one of the two possible value types for sources.
- Model value: The name of a predefined component model, whose characteristics are hard coded into the simulator. The available model names are D, NPN, PNP, NMOS, and PMOS for diodes, BJTs, and MOSFETs, respectively.
- Function value: Describes an output signal of a source. The only option supported is the AC function with a syntax `AC <amplitude> <phase>`.

In our implementation of the specification, the output node (whose voltage is to be plotted) must be named "*out*".



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)

Figure 1: Component with specific designator[1]



Multiplier	Value
p	$\times 10^{-12}$
n	$\times 10^{-9}$
u	$\times 10^{-6}$
m	$\times 10^{-3}$
k	$\times 10^{3}$
Meg	$\times 10^{6}$
G	$\times 10^{9}$

Figure 2: Multiplier letter and its specific value[2]

There are three supported directives:

`.ac`	Tells the simulator to perform an AC sweep over a range of frequencies and calculate the output voltages at each point.
`.op`	Tells the simulator to perform a Quiescent Operating Point (DC) simulation.
`.end`	Marks the end of the Netlist, any lines after this directive will be ignored.

The only AC sweep type supported is decade with the following syntax:

```
    .ac dec <number of points per decade> <start frequency> <end frequency>
```

The frequency points in the sweep follow this equation:

- $f_n = 10^{\frac{n}{n_p}} f_s$, where f_n is the nth frequency point, n_p is the number of points per decade, and fs is the starting frequency.

NB: At least one simulation command (`.ac` or `.op`) is required. Having both simulation commands is supported but having multiple `.ac` commands in the same file is not.

Output

If an AC Sweep directive is present in the Netlist, the results of the simulations will be stored in a Comma Separated Value (CSV) file. The first line specifies the names of the columns in the file. Every other line in the file has the syntax: [Frequency], [Output Voltage at this frequency].

Such files can easily be plotted and processed with various other software packages such as MATLAB, Python, and Excel.

User Interface

The user interface is entirely command-line, which allows easy portability and easy use. To perform simulations the user must only run the program from his terminal and specify the input Netlist. The

[1] From specification: EE1 Project 2021–Circuit Simulator File Format [8]

[2] Idem.

command-line UI also allows the user to easily embed the simulator into other programs and scripts. For example, a user might easily write a Python script, that runs thousands of simulations and processes their results automatically. This can increase productivity significantly.

Design Process

During the initial research and design phase, we found that implementing a circuit simulator capable of Quiescent Operating Point and AC simulations can be broken down into solving the following four major subproblems:

1. Simulating DC and AC linear circuits.
2. Developing linear equivalent models (both small-signal and large-signal) for non-linear components.
3. Using the Newton-Raphson method to find the DC quiescent operating voltages in a large-signal equivalent circuit.
4. Using operating point voltages to construct a small-signal equivalent circuit and solving it at all frequencies in AC sweep, in order to find the desired output voltages.

After further research the following solutions were selected:

Solving purely linear AC and DC circuits is done via Modified Nodal Analysis (MNA). This method works by forming a combination of Kirchhoff current and voltage equations into a system of simultaneous circuit equations, that can be solved to find all nodal voltages. In our program, as is standard in computer simulation, the system of equations is represented and solved in matrix form. If a circuit is to be solved for DC, the matrix and equations are in the real domain. If a circuit must be solved for AC, the matrix and equations are in the phasor (complex) domain.

To solve non-linear circuits, we have used simple but standard linear equivalent models for non-linear components (e.g., the Ebers-Moll model for BJTs, and the Shichman-Hodges model for MOSFETs).

To find the quiescent operating point of a circuit, iterations of the Newton-Raphson method are run until all nodal voltages converge. Checking for convergence is done by comparing the current iteration's node voltages with those from the previous iteration. If the change from the previous to the current iteration is smaller than a specified absolute tolerance, then Newton-Raphson has converged on the correct operating point voltages for the given circuit.

After the circuit's quiescent voltages are found, all non-linear components can be substituted with their small-signal linear equivalent subcircuits. To find the output voltage, MNA is used to solve the small-signal circuit at all frequencies required by the frequency sweep. Those results are stored in an output file.

Additionally, to increase the readability of the output file, we decided we would create a Python script to plot the data on a Frequency-Magnitude plot. As a result, we would be able to check our results with the graphs from our reference software.

Theoretical Methods

Before delving into the concrete implementations of the solutions to the subproblems outlined in the previous section, it is beneficial to elaborate on their two key theoretical foundations: Modified Nodal Analysis and the Newton-Raphson method.

Modified Nodal Analysis [2]

A simple, easy, yet versatile way of solving linear circuits of any size is Modified Nodal Analysis (MNA). As the name suggests MNA builds upon the Nodal Analysis method.

The pure Nodal Analysis method relies on forming Kirchhoff current equations for each node in a circuit and solving the resultant system of equations to find the nodal voltages. However, only relying on equations for currents in the circuit, implies that circuits with voltage sources cannot be solved.

Thus, to be able to solve any linear circuit, even if voltage sources are present, some extension must be made to the simple Nodal Analysis method. Two common such extensions are used: The “Supernode” method and Modified Nodal Analysis.

The “Supernode” method is not very “algorithmic” in nature – it requires some insight into the circuit’s layout and properties that are intuitive to a person, but quite hard to describe in a computer program. Thus, this method is not ideal for a computer-based circuit simulator.

On the other hand, Modified Nodal Analysis, despite not being immediately intuitive for a person, is very “algorithmic” in nature, making it easy to implement in code, and thus, ideal for a circuit simulator.

MNA Methodology[3]

According to the method described on Cheever Erik’s webpage “An Algorithm for Modified Nodal Analysis” [2], MNA consists in generating and solving the following matrix equation:

$$A\,x = z$$

where the A matrix and z vector consist only of known quantities, and the x vector consists only of unknown quantities (nodal voltages and branch currents) that we are solving for.

[3] In literature, Modified Nodal Analysis refers only to the method of creating a solvable system of equations for any given circuit - how these equations are then stored, represented, manipulated, and solved is beyond the scope of MNA. However, in this project, those equations are generated and solved in matrix form, as this is most natural for a computer. Hence in this report “MNA” will usually refer to specifically the matrix implementation of the method, rather than a generalised one.

This equation is shorthand, as the A matrix and the x and z vectors themselves consist of matrices and vectors:

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix}, \ x = \begin{bmatrix} v \\ j \end{bmatrix}, \ z = \begin{bmatrix} i \\ e \end{bmatrix}$$

If we assume that we are analysing a circuit that consists of n nodes, and m independent voltage sources then:

***G* matrix** $(n \times n)$

$$G = \begin{bmatrix} G_{11} & -G_{12} & \cdots & -G_{1n} \\ -G_{21} & G_{22} & \cdots & -G_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -G_{n1} & -G_{n2} & \cdots & G_{nn} \end{bmatrix}$$

The G matrix is often referred to as the conductance matrix because its entries depend on the conductances between nodes in the circuit.

Its main-diagonal entries G_{ii} are equal to the sum of the conductances of all components connected to the i-th[4] node.

All other entries $-G_{ij}$ are equal to minus the sum of the conductances of all components connected between the i-th and j-th nodes.

In a DC Steady-state simulation, the only passive component that can contribute to the conductance matrix is a resistor, but in AC, inductors and capacitors also contribute a complex admittance.

If VCCSs are present, an additional step is required. The transconductance value is introduced into the matrix as follows (assuming introduction after negation of conductances in non-diagonal positions) [3]:

Row associated with:	Column associated with:	Sign of added transconductance:
Input	Positive control	+
Input	Negative control	-
Output	Positive control	-
Output	Negative control	+

[4] When talking about the i-th node in the circuit or k-th voltage source, there is no intrinsic correct ordering. As long as it is consistent across the execution of the whole algorithm, the nodes and voltage sources can be ordered arbitrarily. In our implementation of the algorithm the order of nodes and voltages source is as they were encountered during the parsing of the Netlist.

***B*, *C* matrices** $(n \times m)$ and $(m \times n)$ respectively

The entries of the B and C matrices are defined by the positions of voltage sources in the circuit.

$$B_{ik} = \begin{cases} 1, \text{if the } + \text{ terminal of the } i\text{-th voltage source is connected to the } k\text{-th node} \\ -1, \text{if the } - \text{ terminal of the } i\text{-th voltage source is connected to the } k\text{-th node} \\ 0, \text{otherwise} \end{cases}$$

If the only types of sources present in the circuit are voltage, current, and voltage-controlled current sources, then the C matrix can just be calculated as the transpose of B.

***D* matrix** $(m \times m)$

If the only types of sources present in the circuit are voltage, current, and voltage-controlled current sources, then all entries of D matrix equal zero.

***v* vector** $(n \times 1)$

The v vector is the vector of unknown nodal voltages – after solving the MNA equations the k-th entry of v will contain the voltage at node k.

***j* vector**[5] $(m \times 1)$

The j vector is the vector of unknown voltage source currents – after solving the MNA equations the i-th entry of v will contain the current through the i-th voltage source.

***i* vector** $(n \times 1)$

The value of the k-th entry of the i vector equals the sum of current from **only** constant current sources flowing **into** the k-th node of the circuit.

***e* vector** $(m \times 1)$

The value of the i-th entry of the e vector equals the voltage of the i-th voltage source.

With all those definitions in mind, the way to solve for a circuit's node voltages is to create the G, B, C, D matrices, and the i and e vectors from the known values of the components in the circuit.

Then solve for the unknown vectors v and j in the MNA matrix equation:

$$\begin{bmatrix} G & B \\ C & D \end{bmatrix} \begin{bmatrix} v \\ j \end{bmatrix} = \begin{bmatrix} i \\ e \end{bmatrix}$$

[5] The result of this vector is not of much significance, as the simulator would be trying to solve for node voltages (which are found in the v vector). The existence of the j vector is simply a by-product of how MNA deals with including voltage sources in the equations. Most of the time the contents of the j vector will simply be disregarded after a simulation.

After solving, the nodal voltages of the circuit can be found in the v vector.

Newton-Raphson Algorithm [1]

With the ability to simulate linear circuits in DC through the MNA algorithm explained in the previous section, we can now move on to simulating non-linear components (diodes, BJTs, MOSFETs…).

This is done with the infamous Newton-Raphson root-finding algorithm. Newton-Raphson is an iterative method – an initial guess is made for the solution of a desired equation, which is then refined using the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_{n)}}$$

Where each successive guess x_{n+1} is a function of the previous guess $x_{n.}$ When the difference between two successive guesses becomes sufficiently small, the algorithm has converged on the correct value for x, and no further iterations are required.

The above equations and method can easily be generalised both for systems of equations, and for multivariable functions, both of which are required for using the method for circuit simulators.

The simplest way to implement the Newton-Raphson method for solving non-linear circuits is to find an equivalent circuit for each type of non-linear component, such that solving this equivalent circuit with a voltage guess, will give you the next refined guess.

Finding such circuits by just looking at the classical Newton-Raphson equations is quite unintuitive, so we rather prefer to think of the geometric interpretation of the algorithm, which involves finding the tangent line of the equation at the current guess and using the intersection of the tangent and the x-axis as the refined next guess.

Thus, creating an equivalent subcircuit that implements Newton-Raphson for a non-linear component requires creating a subcircuit with the same IV characteristic as the tangent of the IV characteristic of the non-linear component at a given voltage guess. This also has the benefit that the equivalent circuit for the non-linear component is guaranteed to be linear (tangent must be linear), which means we can use the already existing DC simulation using MNA to implement the iterations of Newton-Raphson.

NB: The initial voltage guesses are predefined for each component (e.g., the initial guess for a diode voltage is 0.7V). Good initial guesses are important to ensure fast convergence of the algorithm. For each non-linear component we tested multiple initial voltage guesses and chose the ones that give the most stable and fast convergence.

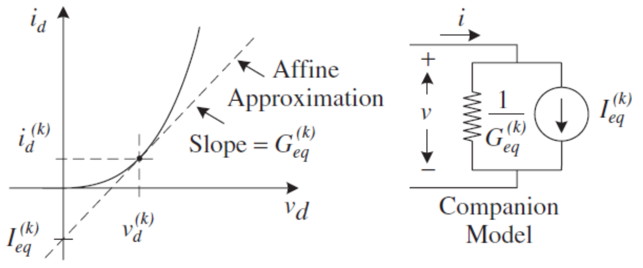
In the next section, we will go over the equivalent linear circuits that the Newton-Raphson algorithm uses.

Linearizing Non-linear Components for Newton-Raphson [4]

The linear equivalent models that are used in conjunction with Newton-Raphson to solve non-linear circuit were found in Farid N. Najm's Circuit Simulation [4].

Diode Model

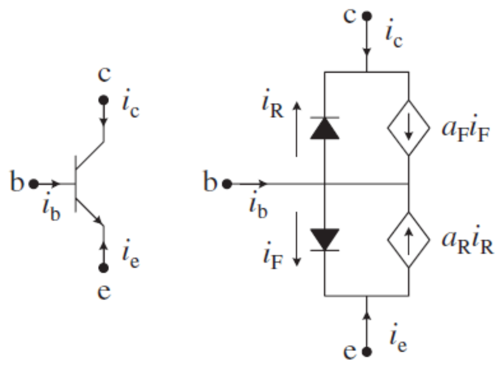
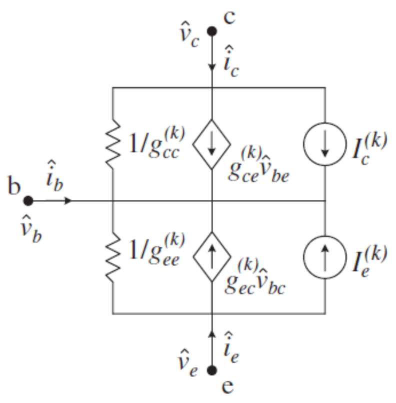
The diode is the simplest non-linear device considered. Its linearised model includes a resistor and a current source. This model and the equations for calculating the value of each linear part are shown in the following figures:

 i_d Affine Approximation $i_d^{(k)}$ Slope = $G_{eq}^{(k)}$ v_d $I_{eq}^{(k)}$ $v_d^{(k)}$ i + v − $\frac{1}{G_{eq}^{(k)}}$ $I_{eq}^{(k)}$ Companion Model	$i_d = I_S\left(e^{\frac{v_d}{V_T}} - 1\right)$ $G_{eq} = \frac{I_S}{V_T}\ e^{\frac{v_d}{V_T}}$ $I_{eq} = i_d - (v_d \times G_{eq})$
Figure 11: Companion Model for a diode[6]	*Formulas for creating equivalent resistor and current source*

Ebers-Moll BJT Model

After some research we decided on using the Ebers-Moll Model for BJTs, which allowed us to have only one equivalent circuit which is valid under all operating modes (cut-off, saturation, active). This is, in our opinion, far more desirable than working with different circuit models valid under different operating modes as that would have required the circuit to be simulated with each model, then excluding answers that violated the expected operating mode conditions.

NPN BJT Ebers-Moll Model

 c i_c b i_b i_e e c i_c i_R $a_F i_F$ b i_b i_F $a_R i_R$ e i_e	 c $\hat{v}_c$ $\hat{i}_c$ $1/g_{cc}^{(k)}$ $g_{ce}^{(k)}\hat{v}_{be}$ $I_c^{(k)}$ b $\hat{i}_b$ $\hat{v}_b$ $1/g_{ee}^{(k)}$ $g_{ec}^{(k)}\hat{v}_{bc}$ $I_e^{(k)}$ $\hat{i}_e$ $\hat{v}_e$ e
Figure 12: The Ebers-Moll model for an NPN BJT[7]	*Figure 13: A companion model for the NPN BJT*[8]

With characteristic values defined by the below equations.

[6] Figure 4.13 from Section 4.3 APPLICATION TO CIRCUIT SIMULATION of the book Circuit Simulation (p.156) [1]

[7] Figure 4.20 from Section 4.3 APPLICATION TO CIRCUIT SIMULATION of the book Circuit Simulation (p. 172) [4]

[8] Figure 4.21 from Section 4.3 APPLICATION TO CIRCUIT SIMULATION of the book Circuit Simulation (p. 174) [4]

$i_e = -I_{SE}\left(e^{\frac{v_{be}}{V_T}} - 1\right) + \alpha_R I_{SC}\left(e^{\frac{v_{bc}}{V_T}} - 1\right)$ $i_c = -I_{SC}\left(e^{\frac{v_{bc}}{V_T}} - 1\right) + \alpha_F I_{SE}\left(e^{\frac{v_{be}}{V_T}} - 1\right)$ Where $I_{SE} = \frac{I_S}{\alpha_R}, \quad I_{SC} = \frac{I_S}{\alpha_F}$ & $\alpha_F = \frac{\beta_F}{\beta_F+1}, \quad \alpha_R = \frac{\beta_R}{\beta_R+1}$	$g_{ee} = \frac{I_{SE}}{V_T}\, e^{\frac{v_{be}}{V_T}}$ $g_{cc} = \frac{I_{SC}}{V_T}\, e^{\frac{v_{bc}}{V_T}}$ $g_{ce} = \alpha_F g_{cc}$ $g_{ec} = \alpha_R g_{ee}$	$I_E = i_e + g_{ee}v_{be} - g_{ec}v_{bc}$ $I_C = i_c + g_{cc}v_{bc} - g_{ce}v_{be}$

Formulas for creating equivalent sub-circuit for the NPN companion model

PNP BJT Ebers-Moll Model

The textbook that we consulted did not explicitly provide the PNP model and equations, but we derived them ourselves, following the step used in the textbook to derive the NPN model.

The PNP's companion model has almost exactly the same form as the NPN model, with the exception that the direction of the VCCSs and current sources are flipped. The characteristic equations are similar to those of the NPN, but with subtle differences:

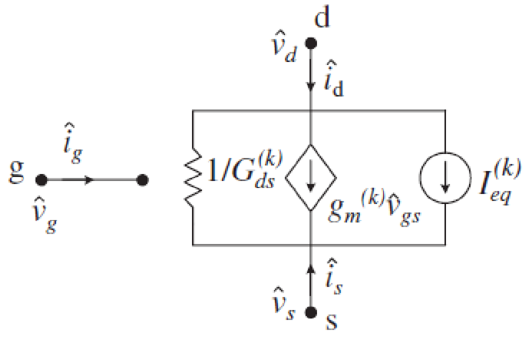
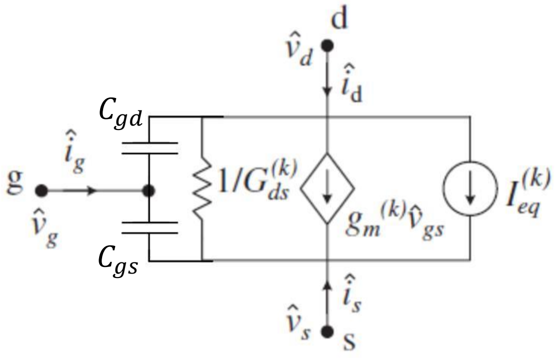
$i_e = I_{SE}\left(e^{\frac{v_{eb}}{V_T}} - 1\right) - \alpha_R I_{SC}\left(e^{\frac{v_{cb}}{V_T}} - 1\right)$ $i_c = I_{SC}\left(e^{\frac{v_{cb}}{V_T}} - 1\right) - \alpha_F I_{SE}\left(e^{\frac{v_{eb}}{V_T}} - 1\right)$ Where $I_{SE} = \frac{I_S}{\alpha_F}, \quad I_{SC} = \frac{I_S}{\alpha_R}$ & $\alpha_F = \frac{\beta_F}{\beta_F+1}, \quad \alpha_R = \frac{\beta_R}{\beta_R+1}$	$g_{ee} = \frac{I_{SE}}{V_T}\, e^{\frac{v_{eb}}{V_T}}$ $g_{cc} = \frac{I_{SC}}{V_T}\, e^{\frac{v_{cb}}{V_T}}$ $g_{ce} = \alpha_F g_{cc}$ $g_{ec} = \alpha_R g_{ee}$	$I_E = -i_e + g_{ee}v_{eb} - g_{ec}v_{cb}$ $I_C = -i_c + g_{cc}v_{cb} - g_{ce}v_{eb}$

Formulas for creating equivalent sub-circuit for the PNP companion model

MOSFET *Shichman-Hodges* Model

For modelling the MOSFET we chose the Shichman-Hodges' large signal model described in Farid N. Najm's book section 4.3 [4]. To improve on this model, we decided to also include gate capacitance (gate-drain and gate-source capacitances specifically).

The model diagrams and equations derived are found below:

 d $\hat{v}_d$ $\hat{i}_d$ g $\hat{i}_g$ $\hat{v}_g$ $1/G_{ds}^{(k)}$ $g_m{}^{(k)}\hat{v}_{gs}$ $I_{eq}^{(k)}$ $\hat{i}_s$ $\hat{v}_s$ s	 d $\hat{v}_d$ $\hat{i}_d$ C_{gd} g $\hat{i}_g$ $\hat{v}_g$ C_{gs} $1/G_{ds}^{(k)}$ $g_m{}^{(k)}\hat{v}_{gs}$ $I_{eq}^{(k)}$ $\hat{i}_s$ $\hat{v}_s$ s
Figure 14: A simple companion model for the n-channel MOSFET[9]	*Figure 15: Added Capacitors onto the n-channel MOSFET*[10]

n-Channel MOSFET Shichman-Hodges *Model*

The following table shows the operation condition and respective modes for an NMOS transistor:

For CUT_OFF REGION $V_{gs} < V_t$	$i_d = 0$ $G_{ds} = 0$ $g_m = 0$	Where $\beta = KP \times \frac{W}{L}$ $\lambda = \frac{1}{V_a}$
For LINEAR REGION $V_{gs} \geq V_t$ $V_{gs} - V_t > V_{ds}$	$i_d = \beta\left(\left(V_{gs} - V_t\right)V_{ds} - \frac{1}{2}V_{ds}^2\right)$ $G_{ds} = \beta\left(V_{gs} - V_t - V_{ds}\right)$ $g_m = \beta\, V_{ds}$	
For SATURATION REGION $V_{gs} \geq V_t$ $V_{gs} - V_t \leq V_{ds}$	$i_d = \frac{1}{2}\beta\left(V_{gs} - V_t\right)^2(1 + \lambda V_{ds})$ $G_{ds} = \frac{1}{2}\beta\,\lambda\left(V_{gs} - V_t\right)^2$ $g_m = \beta\left(V_{gs} - V_t\right)(1 + \lambda V_{ds})$	
$I_{EQ} = i_d - G_{ds}V_{ds} - g_m V_{ds}$		
Formulas for creating equivalent sub-circuit for the NMOS companion model		

p-Channel MOSFET Shichman-Hodges Model

Based on the NMOS model, the PMOS will have the same model and similar parameter equations, but different operation conditions. The below equations follow the format explained in the paper "Companion Models for Basic Non-Linear and Transient Devices" [5] .

[9] Figure 4.23 from Section 4.3 APPLICATION TO CIRCUIT SIMULATION of the book Circuit Simulation (p. 174) [4]

[10] Ibid. with our modifications (added capacitors)

For CUT_OFF REGION $V_{gs} > V_t$	$i_d = 0$ $G_{ds} = 0$ $g_m = 0$	Where $\beta = KP \times \frac{W}{L}$ $\lambda = \frac{1}{V_a}$
For LINEAR REGION $V_{gs} \leq V_t$ $V_{gs} - V_t < V_{ds}$	$i_d = -\beta\left((V_{gs} - V_t)V_{ds} - \frac{1}{2}V_{ds}^2\right)$ $G_{ds} = -\beta\,(V_{gs} - V_t - V_{ds})$ $g_m = -\beta\,V_{ds}$	
For SATURATION REGION $V_{gs} \leq V_t$ $V_{gs} - V_t \geq V_{ds}$	$i_d = -\frac{1}{2}\beta\,(V_{gs} - V_t)^2(1 + \lambda V_{ds})$ $G_{ds} = \frac{1}{2}\beta \times \lambda\,(V_{gs} - v_t)^2$ $g_m = -\beta\,(V_{gs} - V_t)\,(1 + \lambda V_{ds})$	
$I_{EQ} = i_d - G_{ds}V_{ds} - g_m V_{gs}$		
Formulas for creating equivalent sub-circuit for the NMOS companion model		

AC Small-signal models

The linear equivalent models for diodes, BJTs, and MOSFETs explained above are used in the Newton-Raphson method to find a circuit's quiescent operating point.

However, when performing an AC small-signal simulation for an AC sweep, the non-linear components must be linearised to their small-signal equivalents.

Those small-signal equivalents are intrinsically tied to the large-signal linearised equivalent used for the Newton-Raphson method – both methods use a linear approximation around an operating point.

Thus it is easy to convert from the large-signal models outlined in the previous section, to the AC small-signal models, simply removing all constant voltage and current sources (the standard method of transforming a linear large-signal circuit into its small-signal equivalent).

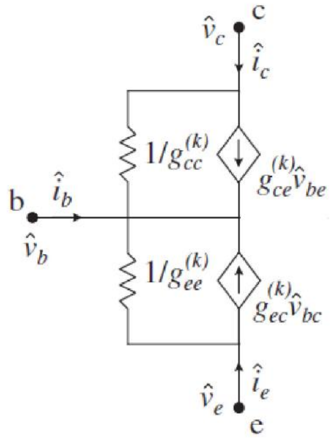
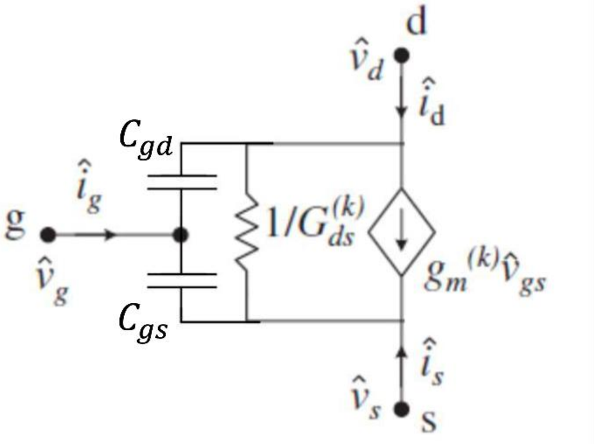
The small-signal parameters are calculated around the DC operating point found by the Newton-Raphson method.

Diode small-signal model

After the constant current source from the large-signal linear equivalent of the diode is removed, we are only left with the dynamic resistance of the diode. Thus, the small-signal model of the diode is just a resistor.

Transistor small-signal models

Once constant sources are removed from the large-signal models of the BJT and MOSFET, these are the equivalent small-signal models we are left with:

<table>
<tr>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td><em>Figure 16: Small signal model for the NPN BJT</em>[11]</td>
<td><em>Figure 17: n-channel MOSFET small signal model</em>[12]</td>
</tr>
</table>

The values of the linear components in the models are calculated with the same equations as those used for the large-signal models.

[11] Figure 4.21 from Section 4.3 APPLICATION TO CIRCUIT SIMULATION of the book Circuit Simulation [4]

[12] Ibid. with our modifications (added capacitors)

Implementation

Full Algorithm Summary

AVS-Simulator's algorithm consists of four main stages. The following flowchart presents the process:

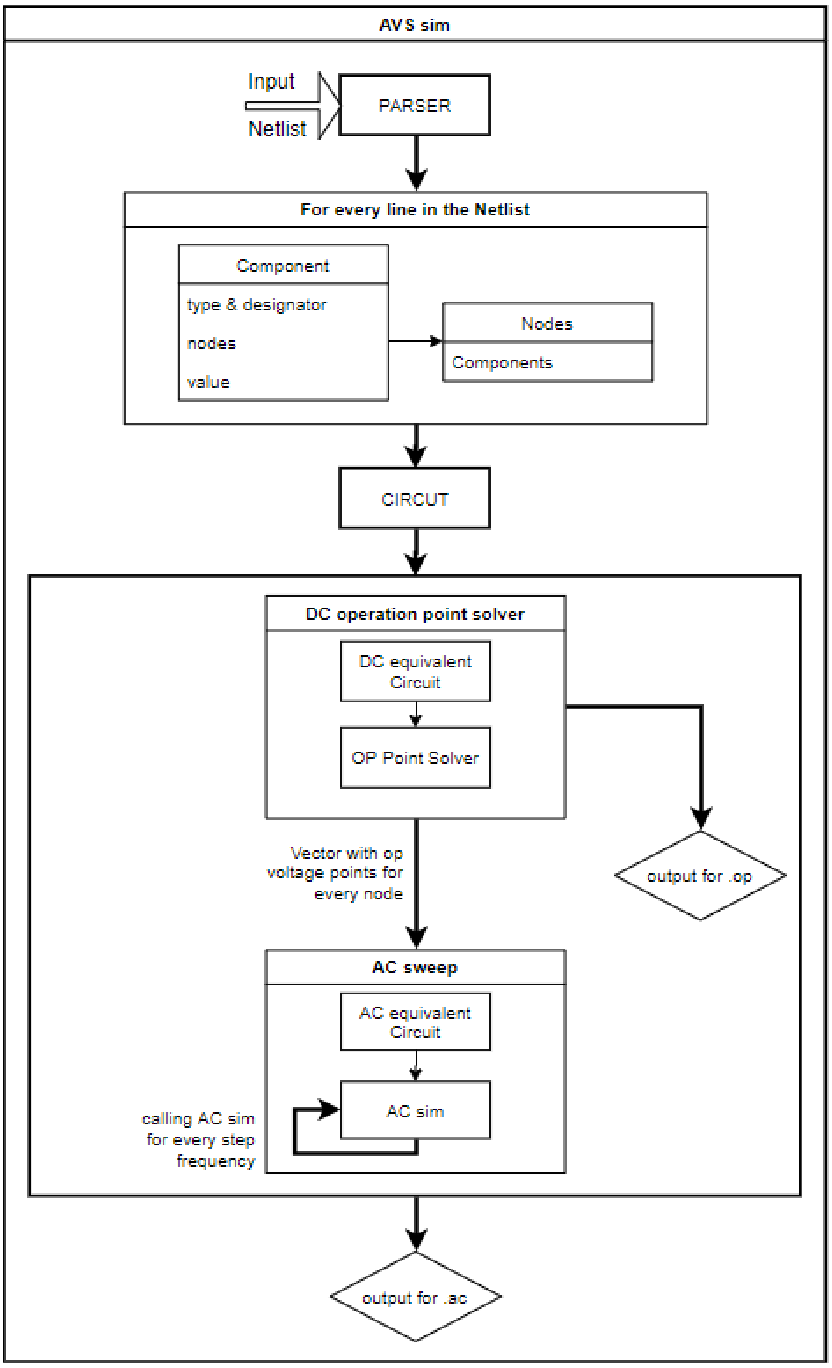


Figure 3: Flowchart explaining the algorithm behind AVS[13]

1. First, the circuit's Netlist is fed to a parser which identifies the components and directives for the simulation. The components are then combined into Node data structures, which are then formed into a Circuit.

2. The second step is for a DC Steady-state equivalent circuit to be obtained (by open-circuiting capacitors, setting AC sources to zero, and shorting inductors). This circuit is then solved by the `OP_Point_Solver` class using the Newton-Raphson method to find the quiescent operating voltages of the circuit. If only an `.op` simulation is required, the simulation ends here.

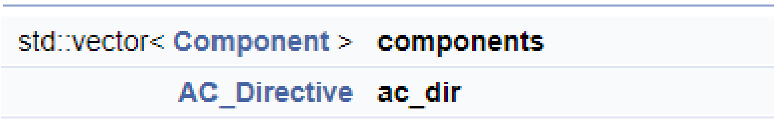
3. If an AC sweep is required, the simulation continues. The operating voltages are used to calculate the small signal parameters for all non-linear components and create the small-signal equivalent circuit.

4. Finally, the small-signal circuit is solved for each frequency point in the requested frequency sweep. The resulting output voltage is written to the output .csv file, finishing the simulation.

[13] Flowchart created in the Visual Paradigm Online website [7]

Parser

The parser is the entry-point of the entire simulator – it is tasked with reading the input Netlist and identifying each component and its parameters, and also any simulation directives.



std::vector< Component >	**components**
AC_Directive	**ac_dir**

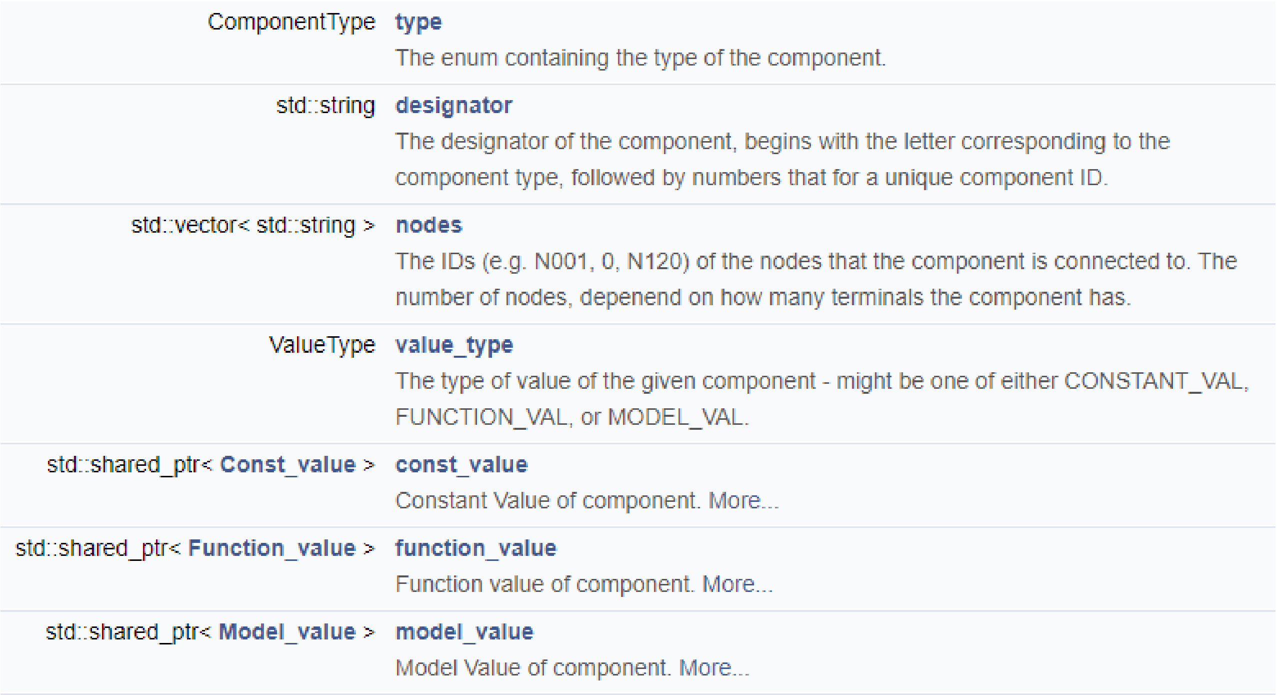
Figure 4: Attributes of `Parser`[14]

For the functions used in the `Parser`, check section 1 of Appendix.

While parsing component lines we distinguish between 2, 3 and 4 terminal devices in order to avoid empty parameters in their vector of nodes. This leads to the node vector having the right size for any specific component.

Component data-structure

The component data-structure is the core of representing data inside the simulator.



ComponentType	**type** The enum containing the type of the component.
std::string	**designator** The designator of the component, begins with the letter corresponding to the component type, followed by numbers that for a unique component ID.
std::vector< std::string >	**nodes** The IDs (e.g. N001, 0, N120) of the nodes that the component is connected to. The number of nodes, depenend on how many terminals the component has.
ValueType	**value_type** The type of value of the given component - might be one of either CONSTANT_VAL, FUNCTION_VAL, or MODEL_VAL.
std::shared_ptr< Const_value >	**const_value** Constant Value of component. More...
std::shared_ptr< Function_value >	**function_value** Function value of component. More...
std::shared_ptr< Model_value >	**model_value** Model Value of component. More...

Figure 5: Attributes of `Component`[15]

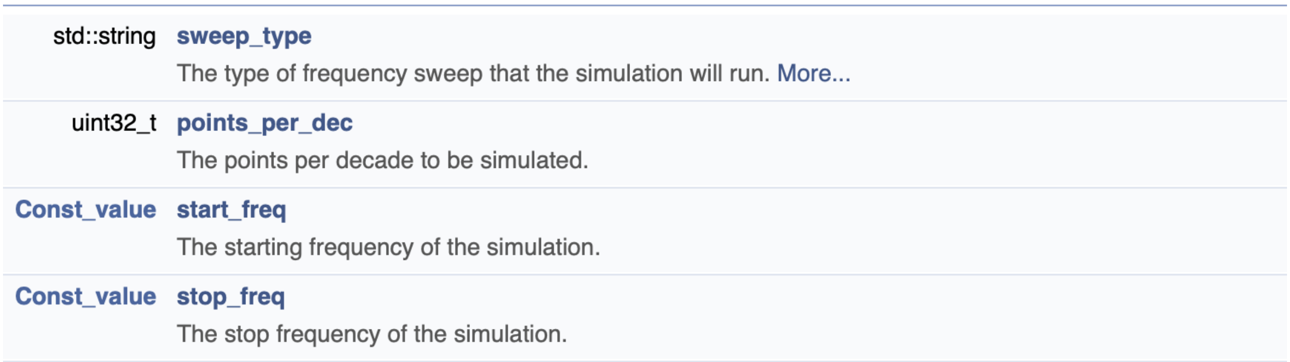
The design of this class was done following the "composition over inheritance" programming principle. Thus, instead of having many classes for different component types, which all inherit from one base Component class, we have one single component data structure that can represent any component type, with any possible value (constant, AC, or model). Composition was preferred in

[14] Screenshot of Doxygen implementation used for creating the documentation.
[15] Idem.

order to keep the number of classes and files in the project lower. We also felt that this increased readability, although this is a purely subjective opinion.

AC directive

An `AC_directive` contains the properties of an AC simulation including the sweep type, and the desired start and stop frequencies. This structure is used only to conveniently manage all AC simulation properties and thus only contains member data, without any member functions.



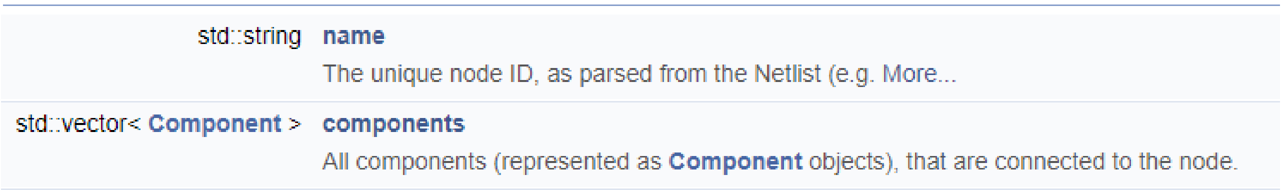
std::string	**sweep_type** The type of frequency sweep that the simulation will run. More...
uint32_t	**points_per_dec** The points per decade to be simulated.
Const_value	**start_freq** The starting frequency of the simulation.
Const_value	**stop_freq** The stop frequency of the simulation.

Figure 6: Attributes of `AC_directive`[16]

Node

This data-structure represents a physical node in the simulated circuits. It is defined by a node identifier (e.g., N001, out, N021, …) and a list of components which are connected to the node (each represented by an object of the component class).

Having this data-structure instead of just working with a collection of the components in the input Netlist, is very useful when implementing the Modified Nodal Analysis algorithm to solve AC and DC circuits. As the “Nodal” part of the name of MNA suggests, nodes are the natural and convenient representation of data when creating and processing the matrices required for solving any circuit.



std::string	**name** The unique node ID, as parsed from the Netlist (e.g. More...
std::vector< **Component** >	**components** All components (represented as **Component** objects), that are connected to the node.

Figure 7: Attributes of `Node`[17]

[16] Screenshot of Doxygen implementation used for creating the documentation.
[17] Idem.

Circuit

The Circuit data-structure neatly combines all the needed nodes and component objects into one single entity. This is both intuitive to work with for the programmers, and also a great interface between all parts of the system – the parser creates a Circuit object from the Netlist which is then passed on to the different simulations that the user has requested.

Some other useful utility functionality and information is also implemented in the Circuit class:

1. Any circuit object can return a copy of itself, but without the ground node present in the list of nodes. This is useful in implementing the MNA algorithm, where the ground node is not expected to contribute anything to the matrix calculations and must thus be discarded.
2. Any circuit object can also return a Steady-state DC Equivalent, or AC Equivalent copy of itself. These equivalent circuits are a starting point in both OP Point and AC sweep simulations, respectively.
3. Each circuit object contains a separate count of both DC and AC voltage sources. These values are used in calculating the necessary sizes of many matrices and vectors in the MNA algorithm and come up frequently enough to warrant saving them as properties of a circuit object, rather than calculating them every time they are needed.

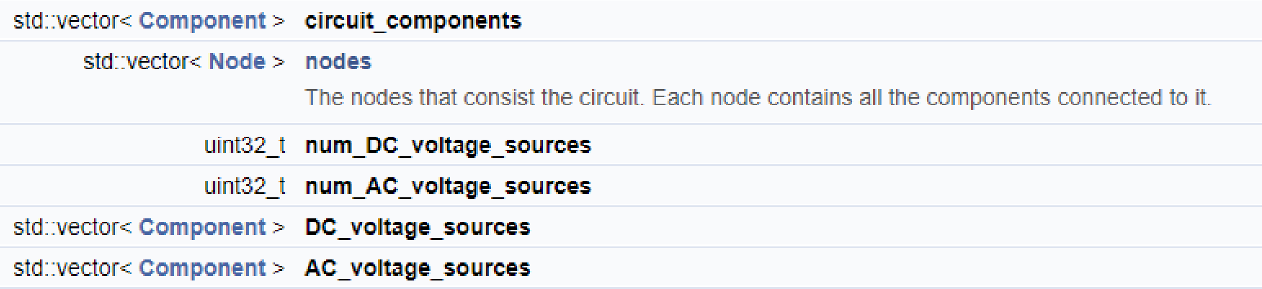
To better understand the process of creating DC Steady-state and AC equivalents of a given circuit, this table shows what happens to every type of component during the transformations:

	Resistors	*Capacitors*	*Inductors*	*AC Sources*	*DC Sources*	*Non-linear devices*
DC Equivalent	unchanged	open circuit	short circuit	removed	unchanged	unchanged
AC Equivalent	unchanged	unchanged	unchanged	unchanged	removed	unchanged

The simplest way to transform a component into an open circuit is to just remove it from consideration entirely. Thus, in our implementation such components are simply removed from all relevant data structures.

The method we adopted for short circuiting a component is to replace it with a 0V voltage source. This approach was chosen as we want to preserve the original nodes in the manipulated circuit (an alternate method would have involved merging nodes which would have required a substantial amount of work and would have rendered the equivalent circuit quite different to the original).

To remove a source, we simply set its value to zero – for a voltage source that translates into making it a short circuit, while for a current source it means transforming it into an open circuit. Sources are transformed into open and short circuits with the methods explained above, in the same manner as any other component would be.



std::vector< Component >	**circuit_components**
std::vector< Node >	**nodes** The nodes that consist the circuit. Each node contains all the components connected to it.
uint32_t	**num_DC_voltage_sources**
uint32_t	**num_AC_voltage_sources**
std::vector< Component >	**DC_voltage_sources**
std::vector< Component >	**AC_voltage_sources**

Figure 8: Attributes of `Circuit`[18]

MNA Implementations – DC_Simulator and AC_Simulator classes

In AVS-Sim the MNA algorithm is implemented in two classes – `AC_Simulator` and `DC_Simulator`. The two are almost identical, with two significant differences:

1. AC_Simulator works with the AC equivalent of a circuit, while DC_Simulator operates on the Steady-state DC equivalent circuit. Those equivalent circuits are generated by the Circuit class as outlined above.
2. DC simulations are in the real domain, while AC is done in the (complex) phasor domain.

To implement generating and solving the MNA equations in matrix for, our project uses the Eigen C++ matrix library [3].

Matrices and vectors consisting of real (`double`)entries are used for DC Simulations. For matrices and vectors in AC Simulations complex (`std::complex<double>`) entries are used.

Newton-Raphson Implementation – OP_Point_solver class

The Newton-Raphson method for solving the quiescent operating point of a circuit is implemented in the OP_Point_Solver class.

Here is a brief overview of the Newton-Raphson process as implemented in our simulator:

1. Linearise the original non-linear circuit based on a set of initial voltage guesses.
2. Solve the resulting linear circuit using the DC Simulator and use the resulting voltages as the new refined guess.
3. Compare the previous and current voltage guesses – if the difference between them is smaller than a predefined threshold, the method has converged.
4. Otherwise, use the new guess to update the linear equivalent circuit and repeat steps 2,3, and 4 until convergence is reached (or a predefined maximum number of iterations is passed, at which point return a non-convergence error to the user)

To linearise non-linear components in the circuit, OP_Point_Solver uses the static linearizing functions in the Linearizer utility class.

[18] Screenshot of Doxygen implementation used for creating the documentation.

This class also stores as properties some constants for the Newton-Raphson method, such as the minimum absolute voltage tolerance under which convergence is reached, and also the maximum number of iterations that the method will try before returning a non-convergence error.

AVS_Sim class

AVS_Sim is the main class of the program – it is in charge of running and coordinating all subsections of the program.

The first thing and AVS-Sim instance will do is run the parser on the supplied netlist. It then passes the parsed circuit into an OP_Point_Solver to find its operating point.

If the user requested an AC Sweep simulation, this operating point is then used to create and solve the small-signal equivalent circuit at every required point in the frequency sweep. AVS_Sim is also in charge of writing the output of the simulation into the output file.

Step-by-step description of the algorithm on an example

For a better understanding of how the algorithm processes the input and calculated the simulation output, this section presents a step-by-step working on a Common Emitter Amplifier example.

Firstly, the input Netlist file is received by the `Parser` and separated into specific components and directives.

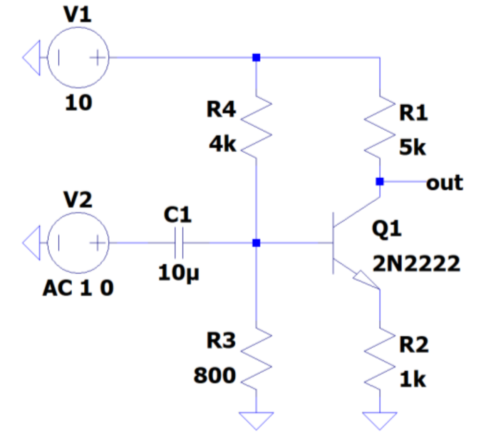


Figure 18: Common Emitter Amplifier example circuit[19]

```
*CE Amplifier example
Q1 out N003 N004 NPN
R1 N001 out 5k
R2 N004 0 1k
V1 N001 0 10
V2 N002 0 AC 1 0
R3 0 N003 800
R4 N003 N001 4k
C1 N003 N002 10u
.ac dec 100 10 10k
.end
```

Netlist of the example circuit

Parser

By parsing the Netlist, the program sorts the lines and creates the components by following the formats below:

```
BJT Component:
   type = BJT
   designator = Q1
   nodes = [out, N003, N004]
   value = MODEL_NAME(NPN)

RESISTOR Component:
   type = RESISTOR
   designator = R1
   nodes = [N001, out]
   value = CONSTANT_VAL(1k)

RESISTOR Component:
   type = RESISTOR
   designator = R2
   nodes = [N004, 0]
   value = CONSTANT_VAL(5k)

VOLTAGE_SOURCE Component:
   type = VOLTAGE_SOURCE
   designator = V1
   nodes = [N001, 0]
   value = CONSTANT_VAL(10)
```

```
VOLTAGE_SOURCE Component:
   type = VOLTAGE_SOURCE
   designator = V2
   nodes = [N002, 0]
   value = FUNCTION_VAL(1,0)

//where all parameters of FUNCTION_VAL are
themselves CONSTANT_VAL

RESISTOR Component:
   type = RESISTOR
   designator = R3
   nodes = [0, N003]
   value = CONSTANT_VAL(800)

RESISTOR Component:
   type = RESISTOR
   designator = R4
   nodes = [N003, N001]
   value = CONSTANT_VAL(4k)

RESISTOR Component:
   type = CAPACITOR
   designator = C1
   nodes = [N003, N002]
   value = CONSTANT_VAL(10u)
```

```
AC_directive:
   type = dec
   nb_points_per_decade = 100
   start_frequency = 10
   stop frequency = 10k
```

[19] Our illustration: made in LT Spice XVII

Circuit

With the components parsed, the program organises the nodes following their order of appearance in the Netlist. Thus, this new vector of nodes is used to build the `Circuit` structure. For this example, we have the following `Circuit` created:

```
        Circuit [out, N003, N004, N001, 0, N002]
```

This *Circuit* holds nodes that have formats as shown below:

```
        out  = Components [Q1, R1]
        N003 = Components [Q1, R3, R4, C1]
        N004 = Components [Q1, R2]
        N001 = Components [R1, V1]
        N002 = Components [V2, C1]
```

All circuits built also have their reference node (Ground) removed in order for them to comply with the requirements of Modified Nodal Analysis which is done for both the DC and AC simulations.

For the DC OP point calculations, the `Circuit` creates a DC equivalent of the original circuit (the one based on the Netlist). This step converts the AC sources to DC ones and sets their value to 0V and open circuits the capacitor (i.e., does not include them in in the component vector). The DC equivalent is shown in *Figure 19.*

The `Circuit` class also prepares an AC equivalent circuit of the original example, to be used for the AC simulation step. This is done by setting the DC voltage source to 0 and substituting the capacitor with its associated complex impedance. Shown in *Figure 20*.

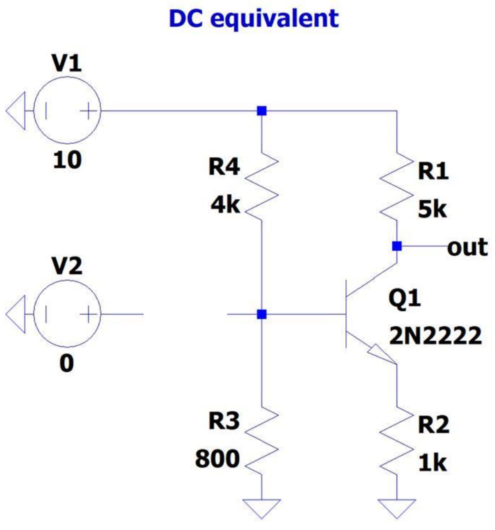


Figure 19: DC equivalent circuit of the CE Amplifier example[20]

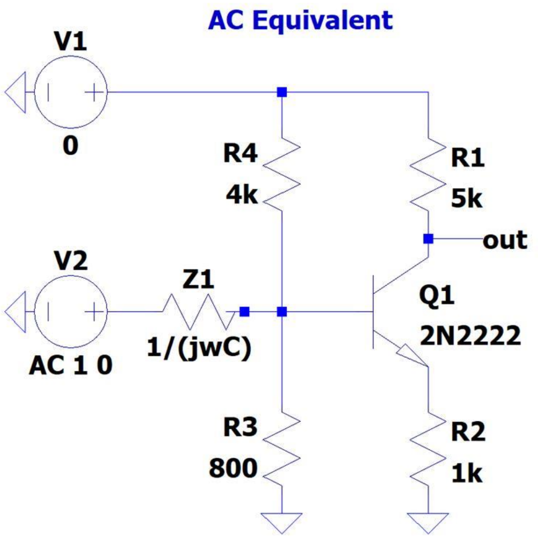


Figure 20: AC equivalent circuit of the CE Amplifier example[21]

[20] Our illustration: made in LT Spice XVII

[21] Idem.

DC_Simulator, Linearizer and OP_Point_Solver

With the DC equivalent circuit created, the program now finds all non-linear components and switches them with their large signal equivalent models. In this case, the NPN is substituted by the appropriate companion model.

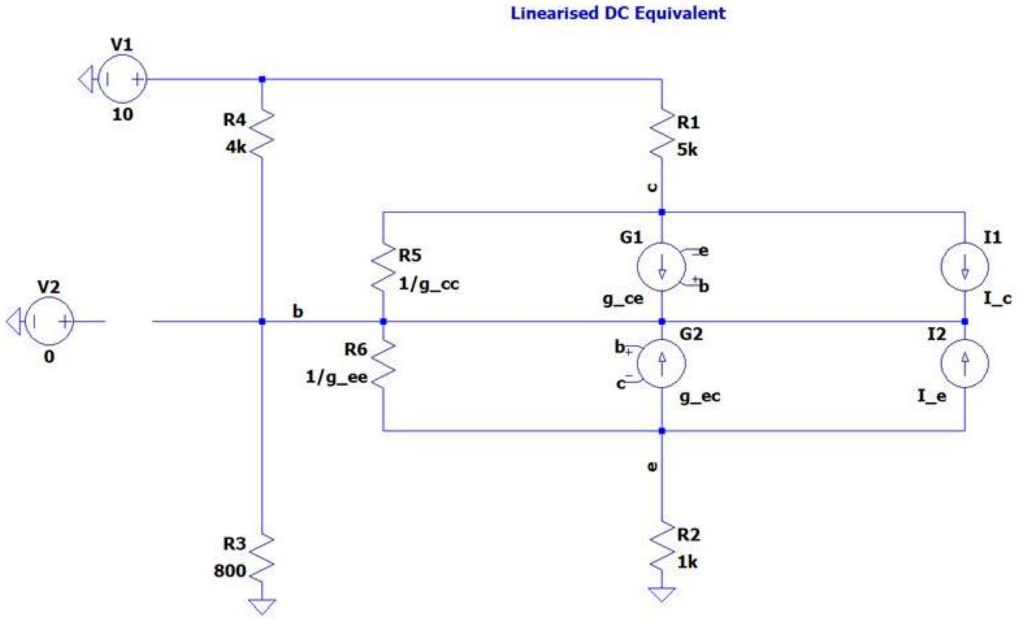


Figure 21: Linearised DC equivalent circuit of the CE Amplifier example[22]

Now, the `OP_Point_Solver` uses a method based on the Newton-Raphson method to find the convergence voltage point for every node in the circuit, thereby solving the circuit. This is done by reapplying MNA calculations as in the `DC_Simulator` onto the DC large signal equivalent circuit.

For this example, the result message of the `OP_Point_Solver` can be seen below:

```
OP POINT SOLVER CONVERGED IN 6 ITERATIONS WITH THE FOLLOWING VOLTAGES:
out:  5.0009958237582497
N003: 1.6633339972181072
N004: 1.0047998394211741
//N002: 0
V1:   10
V2:   0
```

[22] Our illustration: made in LT Spice XVII

AC_Simulator and AVS_sim

This example has an ac directive, meaning the user requested an AC simulation of the circuit. So, the program continues on from the output of the `OP_Point _Solver` and uses its output as the bias point voltages for the Small Signal Equivalent Model (SSEM) circuit.

`AC_Simulator` receives the OP voltage points and uses them as reference voltages for the calculations of the small signal equivalent circuit. This SSEM circuit is modified from the AC equivalent circuit given by the `Circuit` class. The `Linearizer` converts the NPN into its respective small signal model. The linear model has predefined parameters which are used to construct the equivalent components and their values.

The SSEM circuit is shown in *Figure 22*.

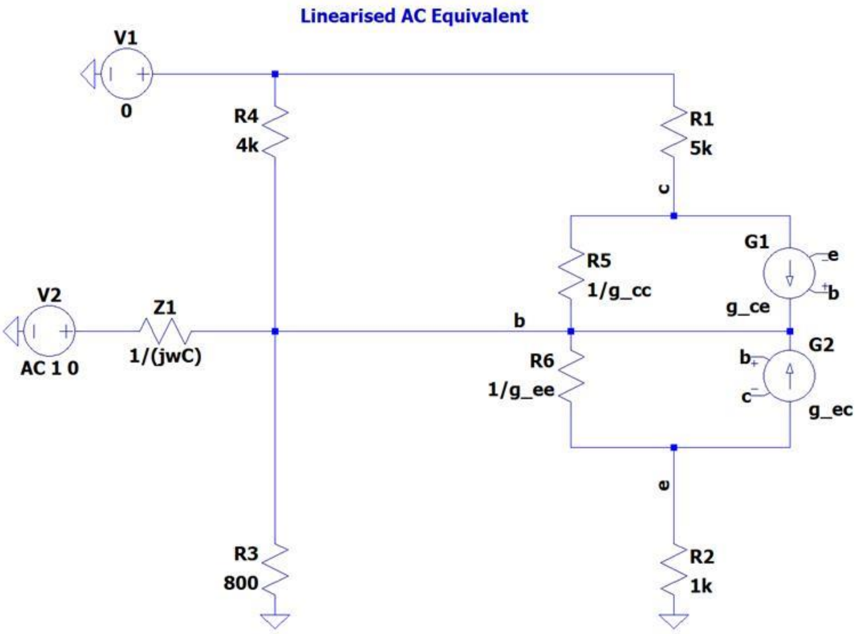


Figure 22: Linearised AC equivalent circuit of the CE Amplifier example[23]

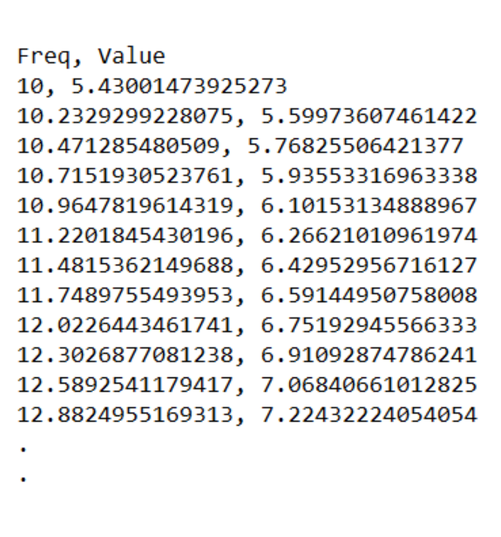
This SSEM circuit is then used to calculate the voltage vector through MNA for every step frequency in a given range described by the AC directive. The iteration through the frequency range is handled by the `AVS-sim` data structure.

`AVS-sim` is also in charge of constructing all component-classes of the program and calling their relevant member functions. It also prints each voltage of the output node its associated frequency step into an output file. The printing follows a CSV format with the first row specifying the column names. The first few lines of the output file for this Common Emitter Amplifier example are shown in *Figure 23.*

Additionally, the project includes a Python script which plots the data in the `output.txt` file in order for the user to easily interpret the results and make observations on the data's behaviour

[23] Our illustration: made in LT Spice XVII

(without having to open MATLAB or Excel). The script plots a lin-log graph of the magnitude of the gain (in dB) against frequency in the specified range. The graph can be seen below.



```
Freq, Value
10, 5.43001473925273
10.2329299228075, 5.59973607461422
10.471285480509, 5.76825506421377
10.7151930523761, 5.93553316963338
10.9647819614319, 6.10153134888967
11.2201845430196, 6.26621010961974
11.4815362149688, 6.42952956716127
11.7489755493953, 6.59144950758008
12.0226443461741, 6.75192945566333
12.3026877081238, 6.91092874786241
12.5892541179417, 7.06840661012825
12.8824955169313, 7.22432224054054
.
.
```

Figure 23: Beginning of the output file for an *.ac dec 100 10 10k directive*

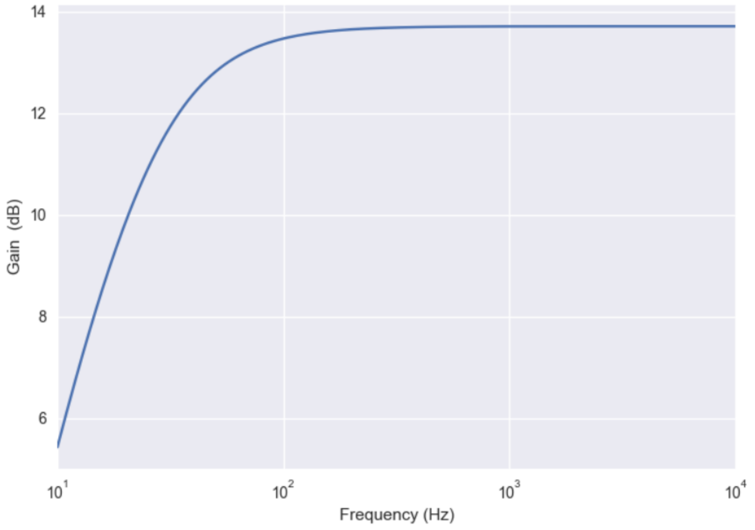


Figure 24: Plot of the output voltage for the Common Emitter Amplifier example[24]

Comparing the result with LTSpice

Now that we have the results of both our simulation program and LTSpice's simulation, we can observe the behaviour of both and compare them.

The two plots have the same overall shape with equal start and end points. Our simulation reaches values of around 13.714dB for the High Frequency Asymptote. As for LTSpice, it also reaches around 13.699dB. The corner frequency for our program is around 23.4Hz. LTSpice gives one around 23.5Hz which is almost identical to our result.

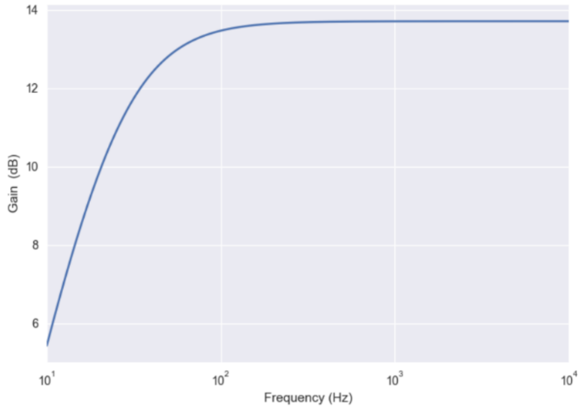


Figure 25: Plot of the output voltage for the Common Emitter Amplifier example[25]

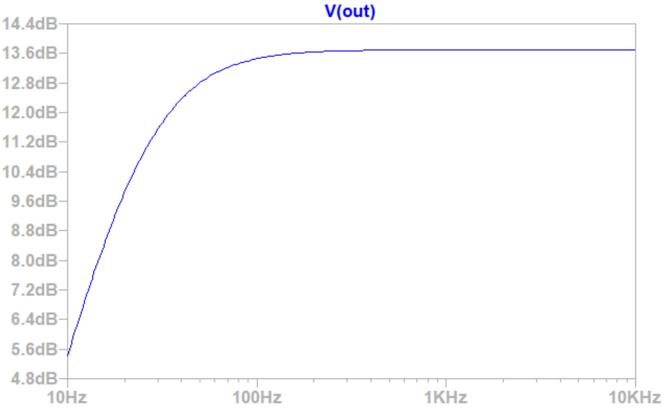


Figure 26: LTSpice graph of the same circuit[26]

[24] Output of our program: using MatPlot

[25] Idem.

[26] Our illustration: made in LT Spice XVII

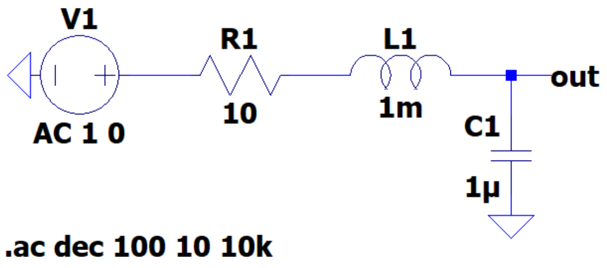
Another comparison that we have made was of the run time of both our `AVS-sim` and LT Spice for the given example. This turned out to show a big discrepancy between the two programs as the duration of simulation in our circuit simulator was 0.734ms for OP point solving alone and 14.6ms for both `Op_Point_solver` and `AC_simulation`. In comparison, our testing indicates that LTSpice seems to take 65ms for `.op` simulation alone (over 88 times longer than our `Op_Point_solver`). For more details regarding the timing on other operating systems tested, check section 2 of the appendix.

Observations and Discussion on testing

This section presents multiple example circuits which are simulated by AVS and the output results of the same circuits in LTSpice. There is a short comparison between every result on the two platforms. The comparisons of the outputs present the accuracy of our program, and the time duration differences shows the efficiency. For more details regarding the timing on all other operating systems tested, check section 2 of the appendix.

RLC Filter

The circuit and its respective netlist for a RLC Filter are found below:

 V1 AC 1 0 R1 10 L1 1m out C1 1µ .ac dec 100 10 10k	* RLC example V1 N001 0 AC 1 0 L1 N002 out 1m C1 0 out 1u R1 N002 N001 10 .ac dec 100 10 10k .end
Figure 27: RLC filter[27]	Netlist of the circuit

By simulating the circuit on both platforms with a directive of `.ac dec 100 10 10k`, we can notice that the plots have a similar behaviour overall, with the same start (at 0dB) and end (-9.5dB) of the graph including its resonant frequency around 4.9kHz. The two graphs can be found below.

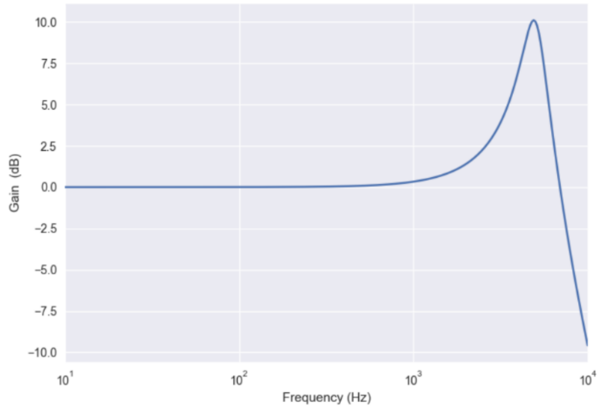
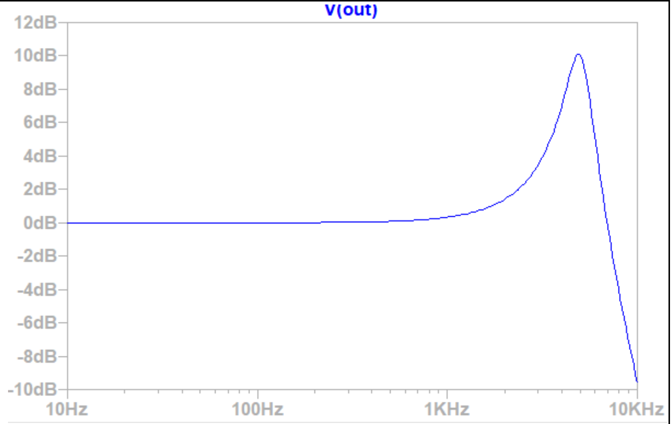
	

Figure 28: RLC filter – AVS-sim plot[28]	*Figure 29: LTSpice plot of the same circuit*[29]

[27] Our illustration: made in LT Spice XVII

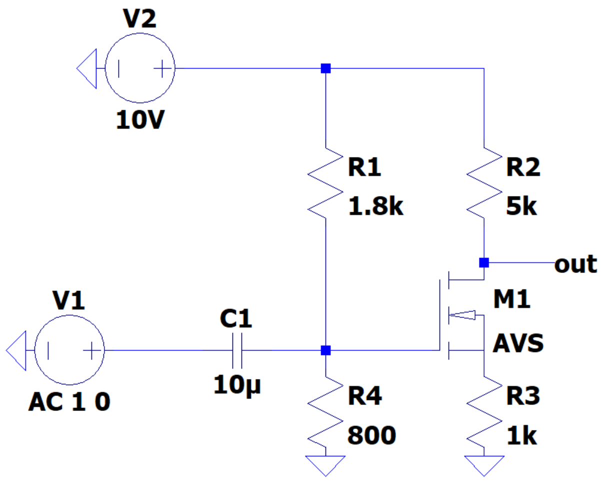
[28] Output of our program: using MatPlot

[29] Our illustration: made in LT Spice XVII

Additionally, by comparing the running time of both simulations, we notice that LTSpice completes `.op` in 64ms, whereas our AVS simulator does it in 0.263ms. Both DC and AC simulations (the later requiring the former) are run in 8.1ms with AVS.

NMOS Common Source Amplifier

The circuit and its associated netlist for a NMOS amplifier are found below:



.model AVS VDMOS(Kp=0.2 lambda=0.01 Vto=2 W=20 L=8)

Figure 30: NMOS Common Collector amplifier[30]

```
* NMOS Amp
M1 out N003 P001 NMOS
C1 N003 N002 10u
V1 N002 0 AC 1 0
V2 N001 0 10
R1 N001 N003 1.8k
R2 N001 out 5k
R3 P001 0 1k
R4 N003 0 800
.ac dec 100 10 10k
.end
```

Netlist of the circuit

By simulating the circuit on both platforms with a directive of `.ac dec 100 10 10k`, we can notice that the plots have a similar overall shape, with the same start (at 4dB) and end (13.7dB) of the graph including its corner frequency around 28Hz. The two graphs can be found below.

Additionally, by comparing the running time of both simulations, we notice that LTSpice completes the `.op` in 78ms, whereas our AVS simulator does `.op` in 20.893ms and both DC and AC simulation in 15.1ms.

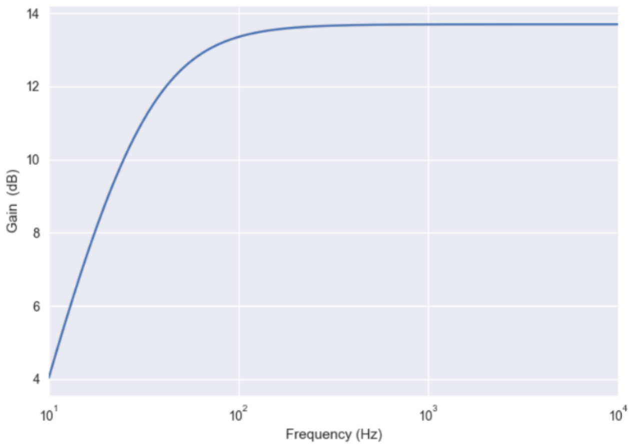


Figure 31: NMOS Amplifier – AVS-sim plot[31]

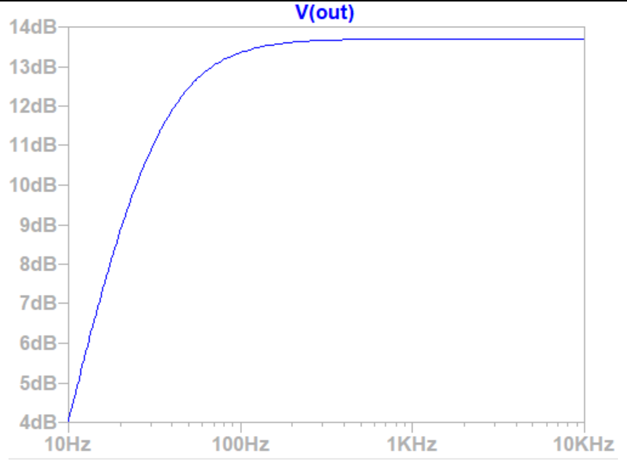


Figure 32: LTSpice plot of the same circuit[32]

[30] Our illustration: made using LT Spice XVII

[31] Output of our program: using MatPlot

[32] Our illustration: made in LT Spice XVII

NPN Differential Amplifier

The circuit and its associated netlist for a NPN Differential amplifier are found below:

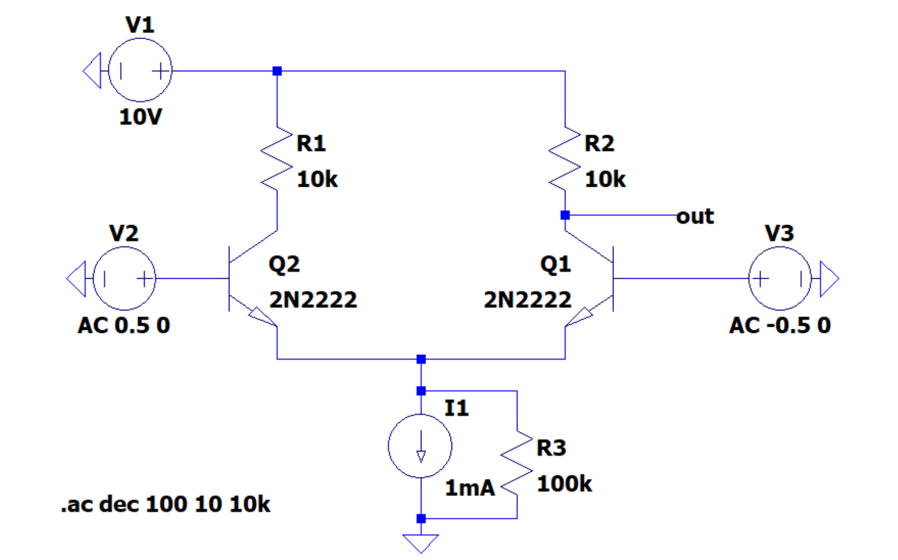


Figure 33: NPN Differential amplifier[33]

```
* NPN Differential Amp
Q1 out N004 N005 NPN
Q2 N002 N003 N005 NPN
R1 N001 N002 10k
R2 N001 out 10k
V1 N001 0 10
V2 N003 0 AC 0.5 0
V3 N004 0 AC -0.5 0
I1 N005 0 1m
R3 N005 0 100k
.ac dec 100 10 10k
.end
```

Netlist of the circuit

By simulating the circuit on both platforms with a directive of `.ac dec 100 10 10k`, we can notice that the plots have a similar overall shape (AVS shows 39.5dB gain, whereas LTSpice gives 39.1dB, a difference which signifies a 4% error). Another difference between the two graphs is that LTSpice presents a drop in the gain at high frequencies. This is most likely the cause of the capacitances considered by LTSpice for physical components. The two graphs can be found below.

Additionally, by comparing the running time of both simulations, we notice that LTSpice completes the `.op` in 81ms, whereas our AVS simulator does `.op` in 1.414ms,but AVS completes both DC and AC simulation in 24.7ms.

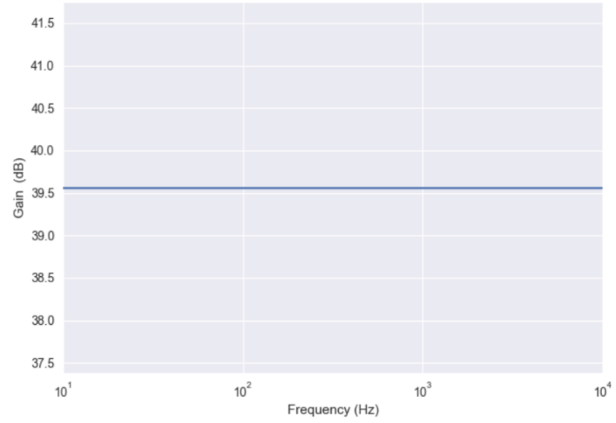


Figure 34: NPN Differential amplifier – AVS-sim plot[34]

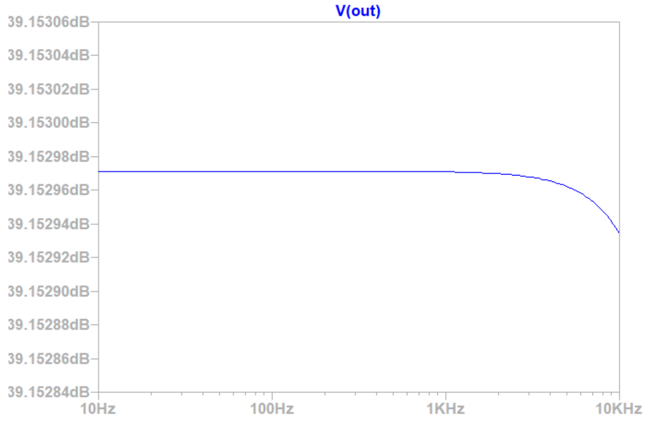


Figure 35: LTSpice plot of the same circuit[35]

[33] Our illustration: made in LT Spice XVII

[34] Output of our program: using MatPlot

[35] Our illustration: made in LT Spice XVII

Darlington Pair

The circuit and its associated netlist for a Darlington Pair are found below:

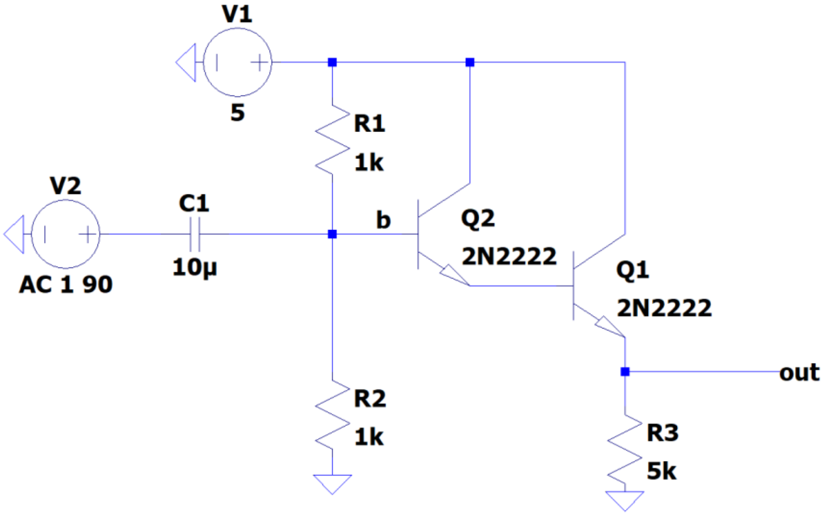


Figure 36: Darlington Pair example circuit [36]

```
* Darlington Pair
Q1 N001 N003 out NPN
Q2 N001 N004 N003 NPN
R1 N001 N004 1k
R3 out 0 5k
R2 N004 0 1k
V1 N001 0 5
C1 N004 N002 10u
V2 N002 0 AC 1 90
.ac dec 100 10 10k
.end
```

Netlist of the circuit

By simulating the circuit on both platforms with a directive of `.ac dec 100 10 10k`, we can notice that the plots have a similar overall shape, with the same start (at -10.7dB) and end (-0.3dB) of the graph including its corner frequency around 36Hz. The two graphs can be found below.

Additionally, by comparing the running time of both simulations, we notice that LTSpice completes `.op` in 85ms, whereas our AVS simulator does it in 1.628ms. AVS completes both DC and AC simulation in 27.4ms.

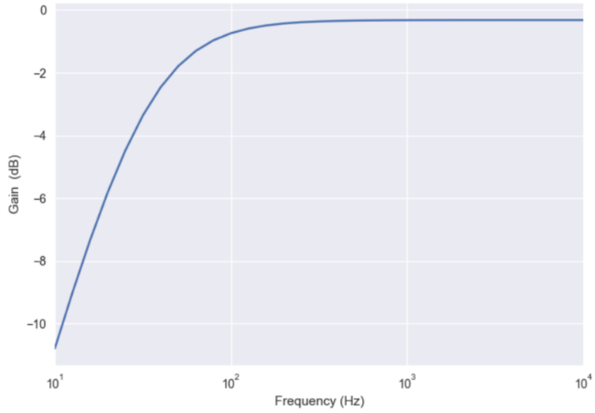


Figure 37: PNP Differential amplifier – AVS-sim plot[37]

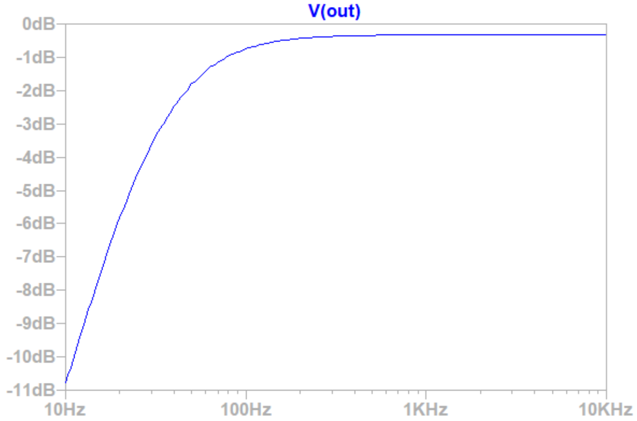


Figure 38: LTSpice plot of the same circuit[38]

[36] Our illustration: made in LT Spice XVII

[37] Output of our program: using MatPlot

[38] Our illustration: made in LT Spice XVII

Project management

Team background

The programming skill levels of our group members varied quite widely when starting the project. This fed into our decision to work together on the code (pair programming) to ensure no one fell behind or was left with an oversized share of the workload.

Meeting management and work distribution

At the beginning of the project, we decided that we would be writing most of AVS's code together. This would be a three-person equivalent of pair-programming: an agile programming practice that has proved itself efficient in reducing bugs and boosting productivity.

We organised team calls on every working day during which each team member took turns to share their screens and code, discussing problems as they arose and stopping to make design decisions, often jotting down ideas using notes applications.

Almost all programming work was done collaboratively apart from problems that could not be solved quickly by the group. In such cases each group member would perform research, debug, test, and try to implement a solution. When a viable solution was found, it would be reviewed by the whole team, and then integrated into the codebase (merged into the master branch).

Advantages of pair-programming

This approach to coding was beneficial for three main reasons:

Firstly, because every group member was involved in writing each part of the program, a holistic understanding of the code was insured. Each group member had at least basic knowledge of the entire code structure and functionality. This was invaluable for debugging given the inevitable interdependence between different parts of the program and the bugs that occurred due to a subtle detail in the operation of functionality somewhere else in the codebase.

Secondly, simple logical and syntactical errors that a single programmer could miss are easily and immediately spotted by the team members watching. This is one of the biggest advantages of pair-programming and saved us significant time in debugging.

Finally, although it might appear that this approach would be inefficient as we did not work on different tasks simultaneously, this was not an issue. On the contrary, we managed to maintain a steady pace throughout the project only getting significantly slowed down by major issues that would have required group discussion regardless.

In retrospect, it is our sentiment that this approach made our progress faster. Quick identification of little errors, the lack of need for code review sessions (which were practically baked into the programming sessions), and the guaranteed inter-compatibility between all sections of the code translated into smooth sailing.

Disadvantages of pair-programming

While this approach was on the whole beneficial, it did have a few minor downsides:

Firstly, pair-programming is meant to be done in person, while for the majority of our project we were forced to work remotely due to being in different countries. This did slow us down, but only marginally.

Secondly, requiring that all team members be present in each coding session, made us dependent on each other’s schedules. However, we never found scheduling to be a big issue, and managed to easily work around each other’s timetables.

Overall, these slight disadvantages did not prove to hinder our progress.

Version control, Build system, Documentation and Coding Standard

Despite being purely issues of logistics, a good choice of a Version Control and Build Automation systems ensured smooth collaboration between all team members.

Version control system

Our choice of version control was git, in combination with GitHub. Git and GitHub are both industry standard, with plenty of support. GitHub was chosen over other similar services (like GitLab and Bitbucket) because all team members had previous experience with it.

Automated Build System

Since all three of our team members use a different operating system (Windows, MacOS and Linux), we decided to use `Make` as our automated-build system. `Make` is cross-platform, and not connected to any specific IDE or editor, thus allowing each team member to use their preferred editor.
This method allowed us to create specific run commands for different constructions of the program.

Below there are presented the major commands created and used:

`make`	Recompiles modified files and rebuilds the program
`make clean`	Deletes all temporary files (*.o) and compiles the program
`make run netlist=’…’`	Compiles and runs the program with the specific input netlist file mentioned in between commas
`make plot data=’…’`	Calls the Python script to plot the data saved in the output file with the name specified in between the commas
`make test`	Calls the Python script for automatically testing multiple circuits and checking their results with the predefined expected results

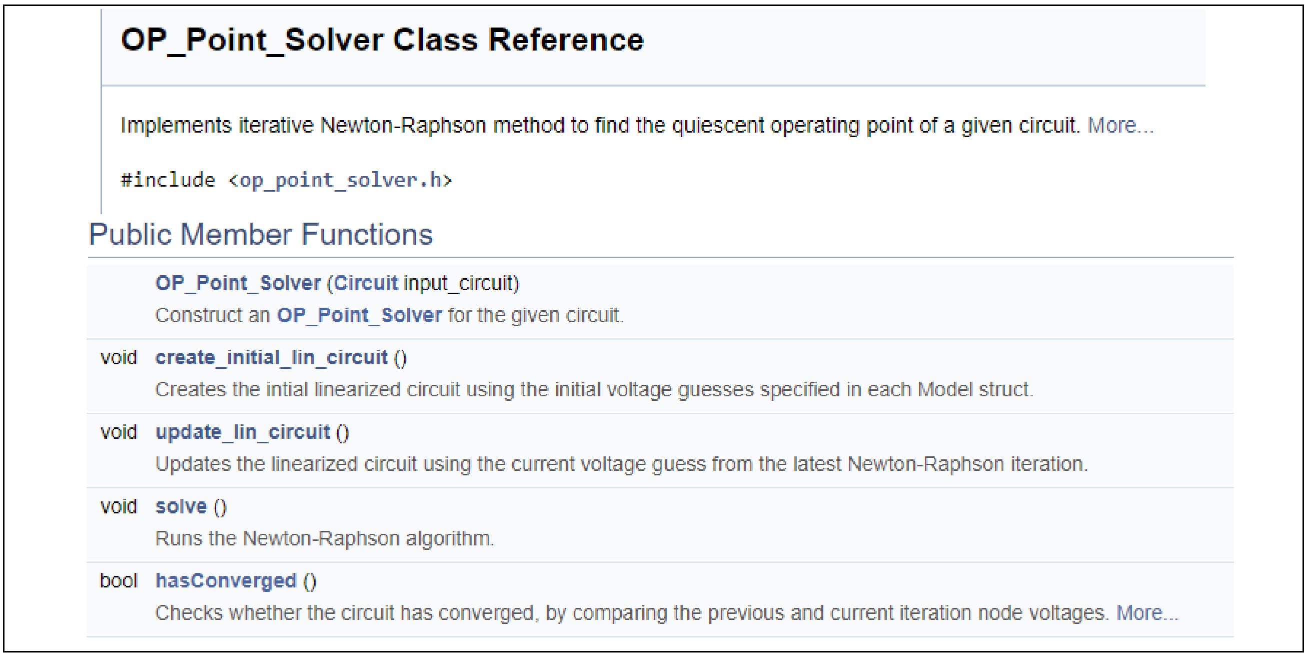
`make mem_check`	Calls the Valgrind tool on a specific named example to check for memory leaks in the program
`make compiler_info`	Returns the environment name, the compiler name and Python version on the PC
`make docs`	Calls Doxygen tool to create or update the html documentation in the ./avs-sim/doc folder
`make cloc`	Calls the cloc terminal command to count the number of lines of code in the project, excluding the Eigen library and the documentation files

The g++ compiler is used for building under Windows and Linux, while clang++ was used for MacOS (clang++ was chosen because of better native integration with MacOS).

Documentation

Doxygen, an industry standard C++ documentation tool, was chosen for auto-generating documentation from commented source code. The documentation proved itself invaluable for debugging purposes and for the writing of this report. The documentation contains descriptions, not only of the classes themselves, but also of the parameters and functions found within them. Each page of the documentation is connected to that of appropriate parent and child classes with hyperlinks. This leads to the user easily being able to search for specific details regarding a class by passing through the whole inheritance tree of the program.

An example of the documentation page for the `OP_Point_Solver` can be found below. It includes the main description of the class, its member functions, attributes, and the functions' detailed explanations.



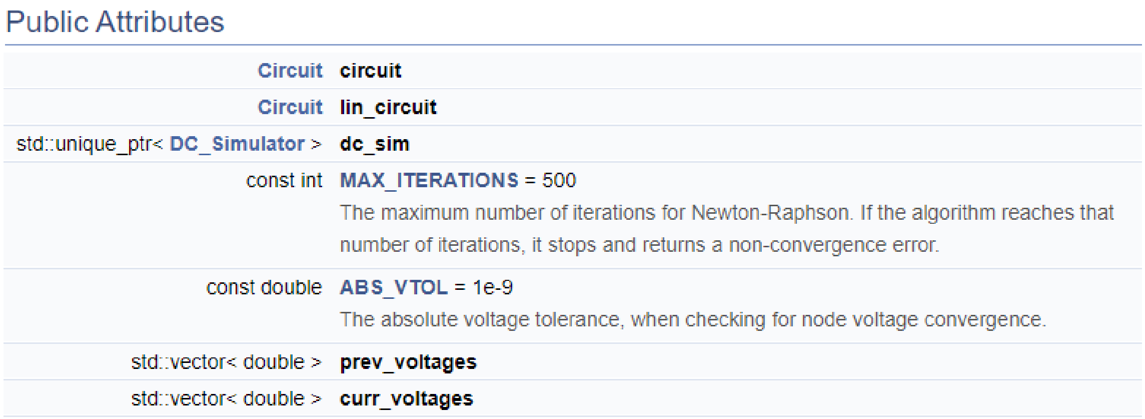
OP_Point_Solver Class Reference

Implements iterative Newton-Raphson method to find the quiescent operating point of a given circuit. More...

```
#include <op_point_solver.h>
```

Public Member Functions

	OP_Point_Solver (**Circuit** input_circuit) Construct an **OP_Point_Solver** for the given circuit.
void	**create_initial_lin_circuit** () Creates the intial linearized circuit using the initial voltage guesses specified in each Model struct.
void	**update_lin_circuit** () Updates the linearized circuit using the current voltage guess from the latest Newton-Raphson iteration.
void	**solve** () Runs the Newton-Raphson algorithm.
bool	**hasConverged** () Checks whether the circuit has converged, by comparing the previous and current iteration node voltages. More...



Public Attributes

Circuit	**circuit**
Circuit	**lin_circuit**
std::unique_ptr< DC_Simulator >	**dc_sim**
const int	**MAX_ITERATIONS** = 500 The maximum number of iterations for Newton-Raphson. If the algorithm reaches that number of iterations, it stops and returns a non-convergence error.
const double	**ABS_VTOL** = 1e-9 The absolute voltage tolerance, when checking for node voltage convergence.
std::vector< double >	**prev_voltages**
std::vector< double >	**curr_voltages**

Figure 39: Documentation for the `OP_Point_Solver`[39]

Although the documentation is auto generated by Doxygen, we had to manually write all function descriptions and characteristics of the attributes in the code.

Screenshots of the documentation for all the classes of the program can be found in the Appendix section.

Coding Standard

During the early days of the project, we agreed on a small list of rules, naming conventions, and good programming practices, that all code would follow. This ensured a safe, uniform, and readable codebase.

For the entire coding standard check Appendix section 1.

[39] Screenshot of Doxygen implementation used for creating the documentation.

Testing and Evaluation

Simulation Timing and Performance Improvements[40]

In order to measure the runtime of our program, we wrote a function which counts the time from when the program starts until the output file is completed. The output of this function is printed onto the terminal along any other messages from the program. With the time measured, we were able to compare the efficiency of our code with that of LTSpice when used on the same circuits and under the same conditions. In *Figure 40* we can see the very drastic difference in **.**op simulation runtime between LTSpice and our simulator.

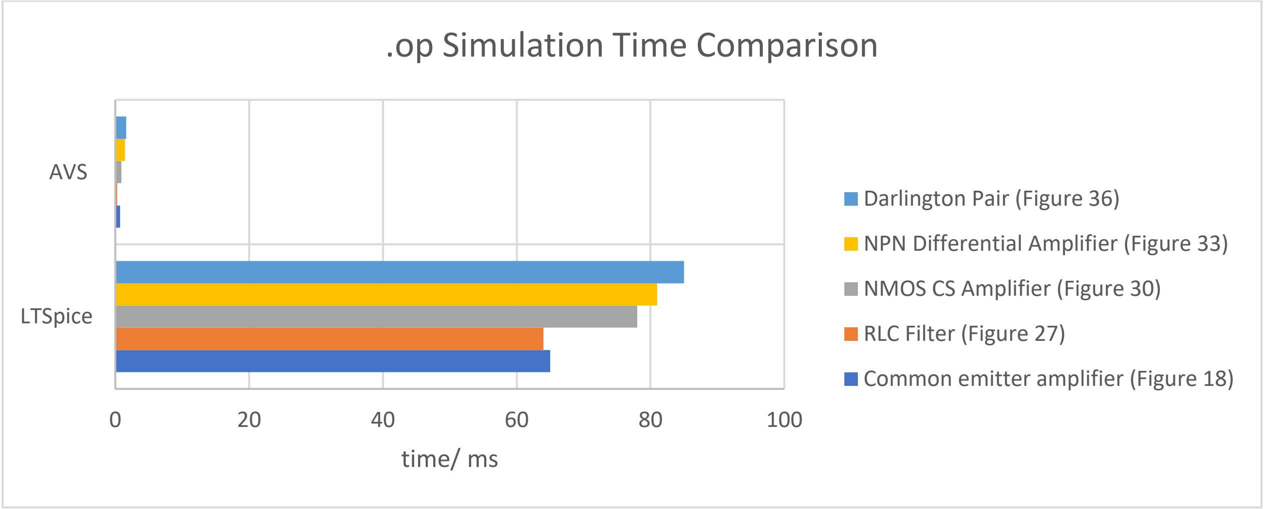


Figure 40: Runtime for .op simulation on both AVS and LTSpice

At a later stage in the development cycle, we ran clang-tidy (a debugging and optimisation tool) which recommended a number of changes (most notably the replacement of passes by copy with passes by reference). After making these changes we used the timing function and observed a very large performance improvement as seen in the figures below. All examples were run under an .ac 100 10 10k simulation.

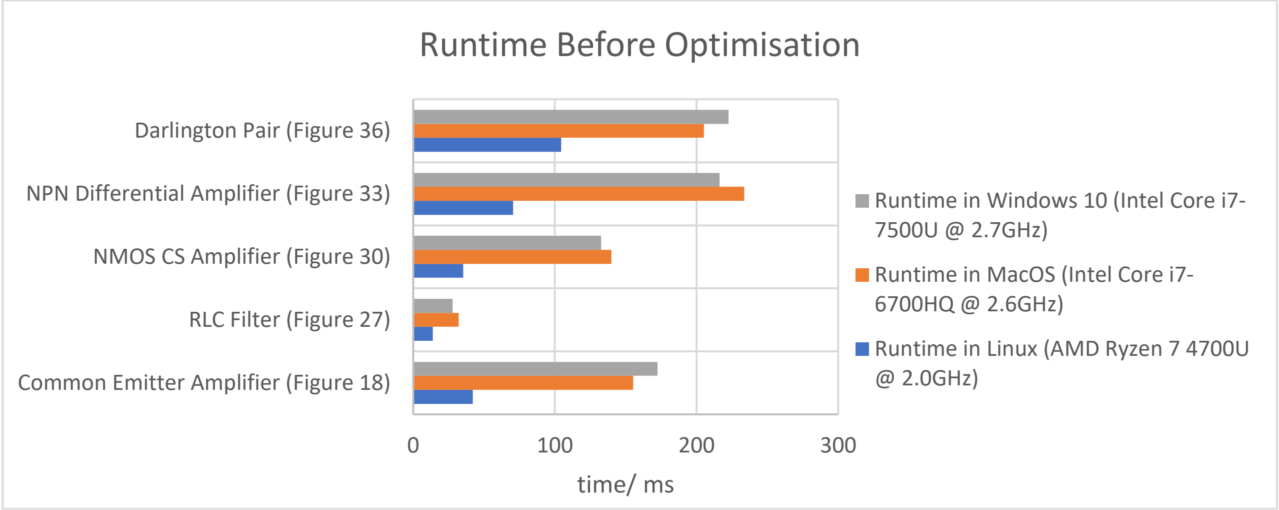


Figure 41: Runtime before optimisation for .op & .ac simulation on all tested OSs

[40] cf. appendix 2 and 3 for the experimental data for this section in tabular form.

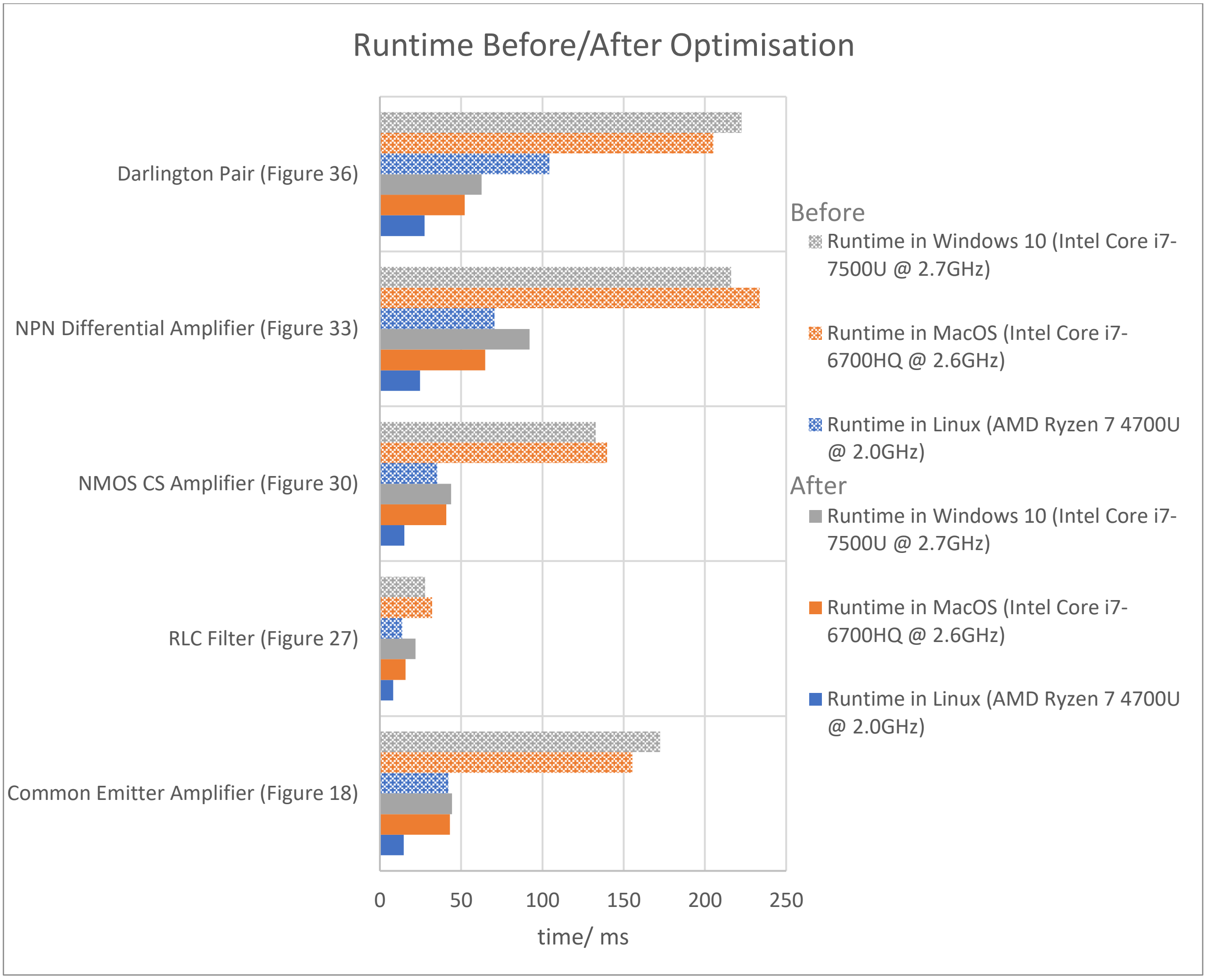


Figure 42: Runtime optimisation for .op & .ac simulation on all tested OSs

From the graph in *Figure 42*, we can observe the large decrease in runtime for the examples with lots of iterations during the Newton-Raphson process. Those examples include the Darlington pair (12 iterations), NMOS Common Source Amplifier (10 iterations), NPN Differential Amplifier (6 iterations), and the CE Amplifier (6 iterations). The other example, the RLC filter, does not require any Newton-Raphson calculation cycles as it does not contain a non-linear component. Hence its runtime decrease depends on optimisations of other code areas.

Testing Script

We wrote a Python script which automatically compares the results of multiple circuit examples with their expected outputs. This script is called in the terminal by using the ”`make test`” command. The command runs every circuit Netlist given in the ./avs-sim/test/examples folder and check their output files with the predefined expected results from ./avs-sim/test/answers. The checks can be done on both .op and .ac simulations as the check is done by comparing every line in the two files. After the check is done, on the terminal, there will be a confirmation written of whether the run of the circuit has returned the expected output or not.

An example of those messages can be found in the *Figure 40*.

```
-------------------------------------------------------
Running example 1  -  test/examples/LPF.avs ...
INCORRECT TEST CASE! ENDING TEST RUN....
EXAMPLE IS:  test/examples/LPF.avs
-------------------------------------------------------
Running example 2  -  test/examples/RLC.avs ...
CORRECT TEST CASE
-------------------------------------------------------
```

Figure 40: example messages from the testing script

Error Messages

We wrote useful and readable error messages for the to aid the user in fixing issues with input files and circuits. In the table below, there are examples of some of the error messages for the `Parser`:

PARSER ERROR: In netlist ./examples/errors/component_redifinition.avs on line 4: ` 4	R1 N003 N001 100` `	^` A component with the same designator already exists... Component designators must be unique!	$ ./avs-sim.exe ./examples/errors/no_dir.avs PARSER ERROR: Found no simulation directives... Nothing to simulate...
PARSER ERROR: In netlist ./examples/errors/expected_const_valu.avs on line 4: `  4| L1 N002 out AC 1 0` `	^` Components of this type can only have a constant value.	PARSER ERROR: In netlist ./examples/errors/unexpected_neg_val.avs on line 3: `  3| R1 N002 N001 -10m` `	^` Components of this type cannot have a negative value
PARSER ERROR: In netlist ./examples/errors/expected_model_val.avs on line 5: `  5| M1 N001 N002 0 10m` `	^` The value of a component of this type must be a model. Please specify the model name of this component.	PARSER ERROR: In netlist ./examples/errors/unknown_component_des.avs on line 4: `  4| F1 N002 out 1m` `	^` Unrecognized component designator letter.
PARSER ERROR: In netlist ./examples/errors/multiple_ac_dir.avs on line 7: ` 7	.ac dec 100 100 100k` `	^` Multiple .ac directives found - a netlist can contain a maximum of one.	

Future debugging

At the moment, there are two limitations that we are aware of.

The first involves the p-type MOSFET which we modelled. Our understanding of the Shichman-Hodges model tells us that, both n and p type MOSFETs share very similar characteristic equations. With the only difference being the operating mode conditions and current direction definitions. This observation led us to assume that the changes between our NMOS and PMOS large signal models should revolve around altering the orientation of both dependent and independent current sources (VCCSs and DC current sources). And also change the operating mode conditions under which each characteristic equation is to be used. Yet, despite having an NMOS model that gives outputs identical to those given by LTSpice, our PMOS model has shown to encounter some limitations in efficiency and accuracy. Multiple attempts at finding changes in equations or a different model from theory where made. After very extensive, but unsuccessful debugging, both theoretical and practical (in the code), the initial model applied has not been changed.

The second bug that we plan on removing is the one that occurs when two transistor terminals are shorted (e.g., the Base and Collector of a BJT as would be done when building a current mirror). This situation is setting the two terminals to the same node name in the netlist, as seen in the example in *Figure 41*. The scenario creates large signal linear components with unexpected values. For example, during the Newton-Raphson calculations for solving the op points, we encounter NaN or inf component values that render the entire output useless. Hence, in this scenario, one erroneous value can spread to the whole circuit when iterating through the Newton-Raphson method. This is caused by the fact that if one or more components have two of their terminals connected to the same node in the large signal equivalent circuit, some of the linearised components will be shorted. Thus, creating an infinite current loop around a current source and a VCCS. The solution we considered (cf. *Figure 41*) is to simply connect between two terminals a 0V source, thereby ensuring both nodes have the same voltage while maintaining their unique and distinct identifier. This process can however not be automated as there is no way of knowing, from the input Netlist alone, where to introduce this voltage source relative to other components connected with the two terminals.

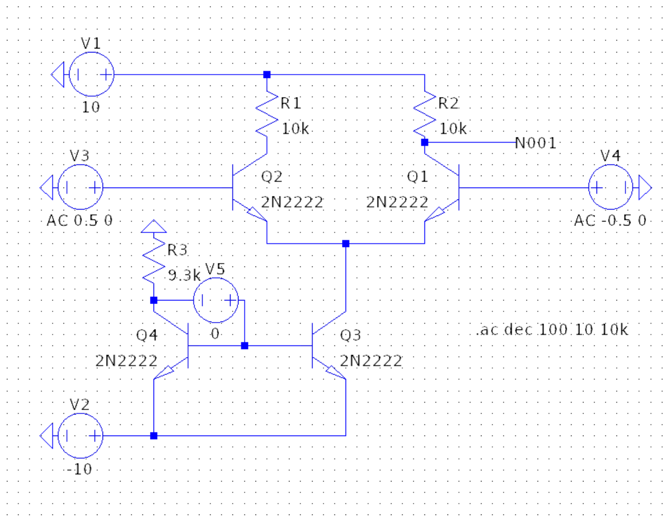


Figure 41: Differential pair with a current mirror with Base-collector shorted for Q4[41]

[41] Our illustration: made in LT Spice XVII

Below we are showed the old Netlist with the peculiar situation and its equivalent working Netlist. The components changed are written in bold.

V1 N003 0 10 R1 N003 out 10k R2 N003 N002 10k Q1 N002 N005 N006 NPN Q2 out N004 N006 NPN Q3 N006 N007 N008 NPN **Q4 N007 N007 N008 NPN** R3 0 N007 9.3k V2 N008 0 -10 V3 N005 0 AC 0.5 0 V4 N004 0 AC -0.5 0 .ac dec 100 10 10k .end	V1 N003 0 10 R1 N003 out 10k R2 N003 N002 10k Q1 N002 N005 N006 NPN Q2 out N004 N006 NPN Q3 N006 N007 N008 NPN Q4 **N009** N007 N008 NPN **V5 N009 N007 0** R3 0 **N009** 9.3k V2 N008 0 -10 V3 N005 0 AC 0.5 0 V4 N004 0 AC -0.5 0 .ac dec 100 10 10k .end
Differential pair with a current mirror with Base-collector shorted for Q4	*Error solved by making the short between the base and collector as a DC voltage source of 0V*

Extension and future plans

There are a number of features that we would like to implement in the future. We built AVS with the idea in mind that we want to have the code in such a way that additional features could be quite easily implemented. This adaptability is the result of a high-level of functional and structural decomposition. This is particularly evident when observing the parser. It identifies and stores each token in a different data structure, meaning that new component parameters or directives can easily be added.

Parsing non-ASCII characters

A useful feature that was implemented but not submitted in the final version was the ability to identify the Greek letter μ as the micro multiplier (in addition to the u as required in the specification). Seeing as LTSpice's "View Spice Netlist [from a GUI schematic]" feature uses μ instead of u for the multiplier, testing would have been made a bit quicker and removed the need to adapt the Netlist for use with AVS-sim. This involved handling wide characters as the normal character variable type in C++ only supports ASCII characters. Our implementation would read two successive characters in a standard string, cast them to integers and check whether they match up with the two values used to represent the μ character (-62 followed by –75). If they did, the micro multiplier was used, otherwise, the unknown character was ignored.
This method proved effective when testing with the Clang++ compiler on both MacOS and Linux. The reason for which we did not include this feature in our final submission is that we were not confident in the fact it would work on all systems we set out to support. We were particularly concerned about compatibility with Windows which handles wide characters in a different fashion.

Temperature changes

One of the simplest features that could be implemented would be to allow the user to set the environment temperature for the simulation. This implies setting the value of thermal voltage used in non-linear small signal parameters to something other than the default value (26 mV), which for now is predefined. This could quite easily be done by identifying the `.TEMP` directive when parsing the Netlist and setting a temperature value in a separate data structure. Thereby obtaining a value for thermal voltage using the following formula:

$$V_t = \frac{k \times T}{q}$$

where k is the Boltzmann constant, T is the temperature in Kelvin (assuming user input is in °C, this will be calculated by the program as being equal to user input + 273.15°) and q is the charge of an electron (approx. $1.6 \times 10^{-19}\, C$).

AC sweep simulations for other types

Another potential improvement would be to allow the user to perform AC sweep simulations other than the decade type. By taking the types present in LTSpice as examples, the potential AC simulations we could add include octave, linear and list. Both octave and linear types could easily be calculated by adding appropriate formulae for the frequency step. The List simulation could be performed by forming a vector of frequencies from user input which would be used in AC simulation. This would be similar to what our current frequency step loop is doing, but it would only call for the output voltage at the frequencies included in the input vector of the list directive.

Convergence for currents

To improve the accuracy of our simulator, a second criteria could be added when checking for convergence of non-linear circuits with the Newton-Raphson method. One could conceive that a circuit might converge in voltage, while still being relatively far from the actual solution when considering currents.

In a similar way to the voltages, the variation of the currents in the entire circuit would be required to remain under a certain threshold. This feature would be particularly relevant given the fact that both BJTs and Diodes obey an exponential relationship between voltage and current. Hence, a small variation in voltage can lead to a large variation in current.

It is worth noting that, while we were concerned this would be an issue, in all the circuits we tested, a voltage convergence condition alone yielded exceptionally accurate results. Hence adding this check would only further increase the accuracy.

Transient simulation

Finally, transient simulation could be added. While this would require writing a substantially different solver (from our brief reading, transient simulation hinges on a completely different algorithm), it should be possible. Our node, `component` and `circuit` objects could be used as well as our parser. In fact, the way in which AC sources are handled means that function types other than the generic AC can be handled. Our code is adaptable to AC simulation with sources outputting a variety of waveforms.

Appendix:

1. Coding Standard

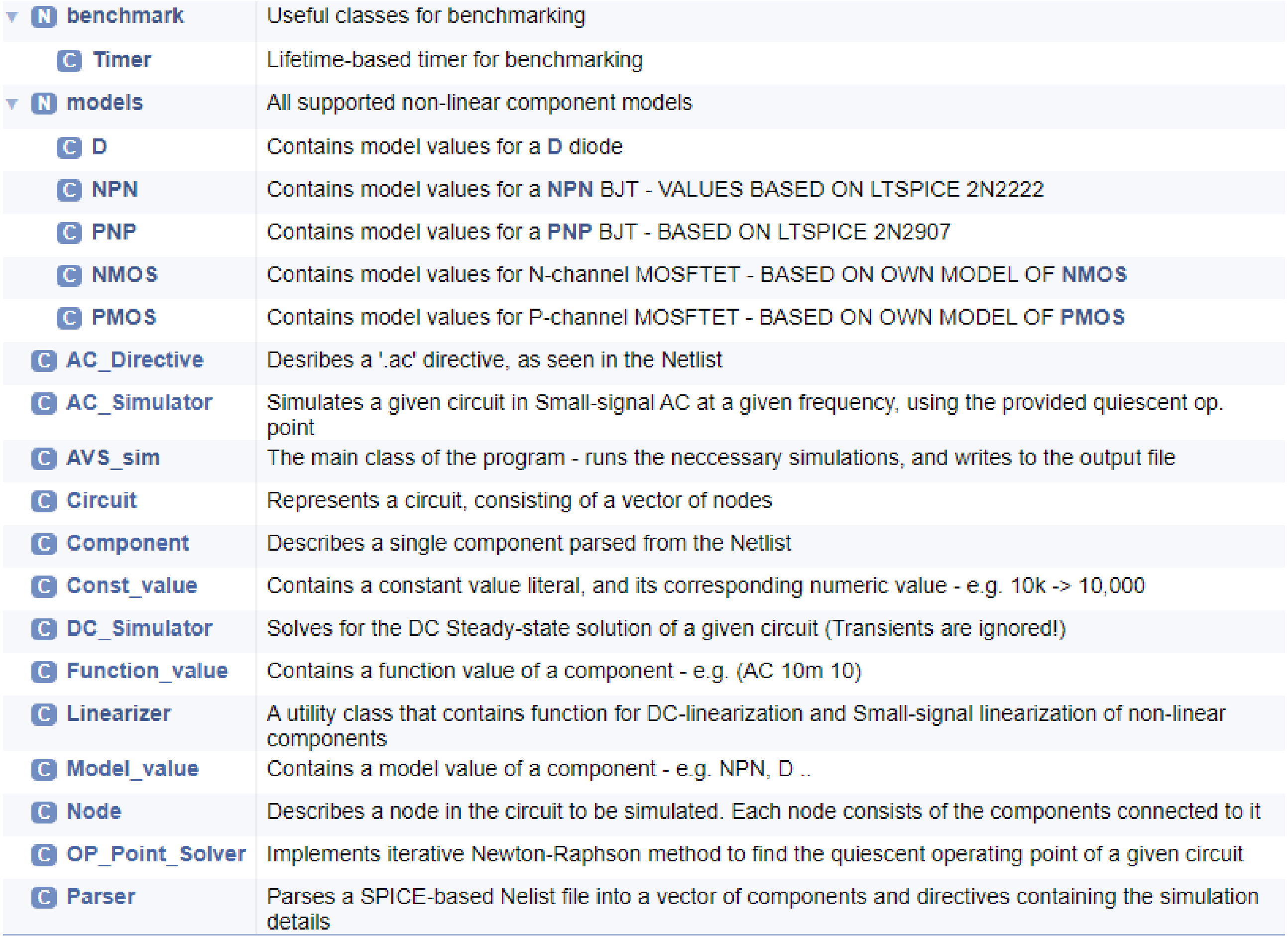
Here are the set of agreed rules that the code of this project must adhere to:

1. Variable and function naming must follow the `snake_case` convention. Class names must begin with a capital letter, object instances of those classes must begin with a lowercase letter. All variables must be named descriptively, except for common conventions such as for iterators.
2. The following whitespace conventions are used – tabs must be 2 spaces long. The opening curly bracket after a function definition, if statement, while statement, and for statement must be on a new line.
3. Smart pointers must be used instead of raw pointers. When possible, prefer a unique pointer to a shared pointer.
4. When possible, pass function parameters by const reference instead of by value or use other appropriate move semantics to ensure no unnecessary copying. Adhering to this rule contributes to a great performance optimization over the duration of a big simulation.
5. Getter and setters are to be avoided if no invariants must be ensured. If a member variable has no invariants, it can be made public.

2. Essential Classes

The program includes its own documentation saved locally in the main project folder. This can be found on …/avs-sim/doc/html/index.html

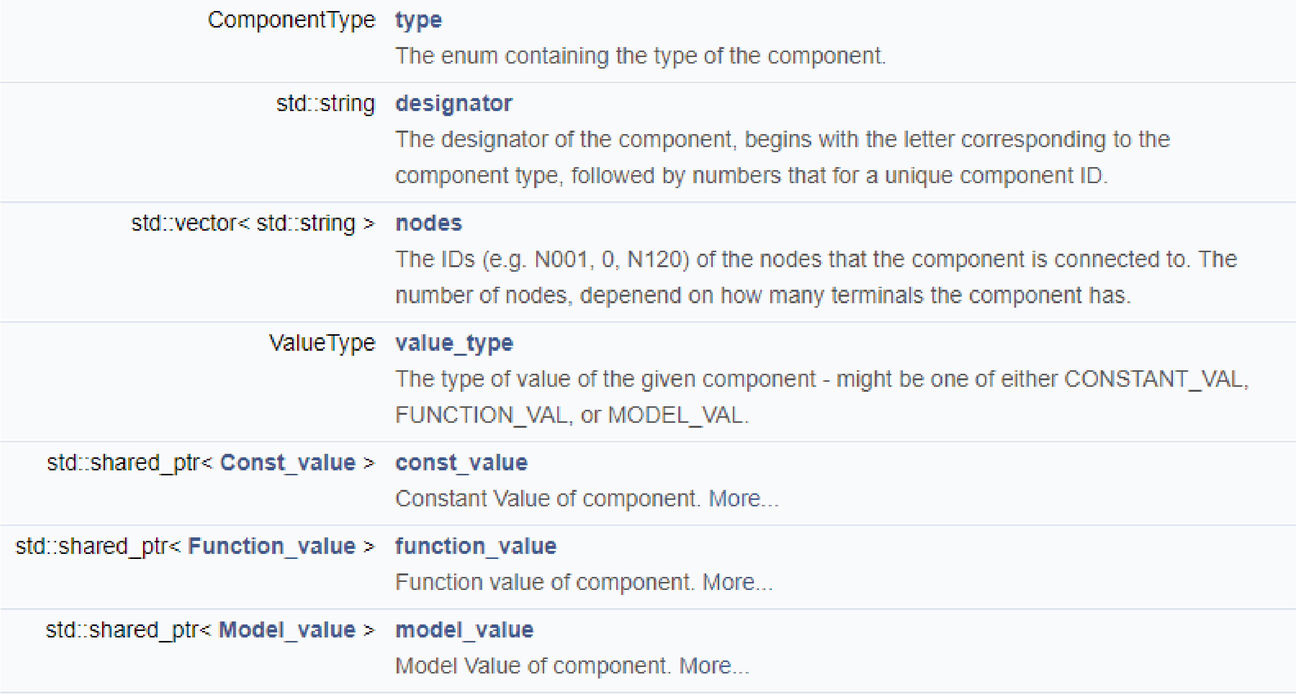
The following list presents all elements of the project.



▼ N **benchmark**	Useful classes for benchmarking
C **Timer**	Lifetime-based timer for benchmarking
▼ N **models**	All supported non-linear component models
C **D**	Contains model values for a **D** diode
C **NPN**	Contains model values for a **NPN** BJT - VALUES BASED ON LTSPICE 2N2222
C **PNP**	Contains model values for a **PNP** BJT - BASED ON LTSPICE 2N2907
C **NMOS**	Contains model values for N-channel MOSFTET - BASED ON OWN MODEL OF **NMOS**
C **PMOS**	Contains model values for P-channel MOSFTET - BASED ON OWN MODEL OF **PMOS**
C **AC_Directive**	Desribes a '.ac' directive, as seen in the Netlist
C **AC_Simulator**	Simulates a given circuit in Small-signal AC at a given frequency, using the provided quiescent op. point
C **AVS_sim**	The main class of the program - runs the neccessary simulations, and writes to the output file
C **Circuit**	Represents a circuit, consisting of a vector of nodes
C **Component**	Describes a single component parsed from the Netlist
C **Const_value**	Contains a constant value literal, and its corresponding numeric value - e.g. 10k -> 10,000
C **DC_Simulator**	Solves for the DC Steady-state solution of a given circuit (Transients are ignored!)
C **Function_value**	Contains a function value of a component - e.g. (AC 10m 10)
C **Linearizer**	A utility class that contains function for DC-linearization and Small-signal linearization of non-linear components
C **Model_value**	Contains a model value of a component - e.g. NPN, D ..
C **Node**	Describes a node in the circuit to be simulated. Each node consists of the components connected to it
C **OP_Point_Solver**	Implements iterative Newton-Raphson method to find the quiescent operating point of a given circuit
C **Parser**	Parses a SPICE-based Nelist file into a vector of components and directives containing the simulation details

Component

All components we simulate are represented by this class and have the following attributes:



ComponentType	**type** The enum containing the type of the component.
std::string	**designator** The designator of the component, begins with the letter corresponding to the component type, followed by numbers that for a unique component ID.
std::vector< std::string >	**nodes** The IDs (e.g. N001, 0, N120) of the nodes that the component is connected to. The number of nodes, depenend on how many terminals the component has.
ValueType	**value_type** The type of value of the given component - might be one of either CONSTANT_VAL, FUNCTION_VAL, or MODEL_VAL.
std::shared_ptr< **Const_value** >	**const_value** Constant Value of component. More...
std::shared_ptr< **Function_value** >	**function_value** Function value of component. More...
std::shared_ptr< **Model_value** >	**model_value** Model Value of component. More...

- Where the *Nodes* element is a vector of strings representing all the nodes connected to a component. The number of elements in this vector can range from two (e.g., for a resistor) to four (for a voltage controlled current source). It is worth noting that the order of elements in this vector is hugely important for non-linear components and sources. By convention the order of terminals for those components is:
    - For diodes: Anode followed by cathode.
    - For BJTs: collector, base then emitter.
    - For MOSFET: drain, gate then source.
    - For independent sources, passive sign convention is followed (+, -), as well as for the VCCS, including an extra two elements for positive, respective, negative control terminals.

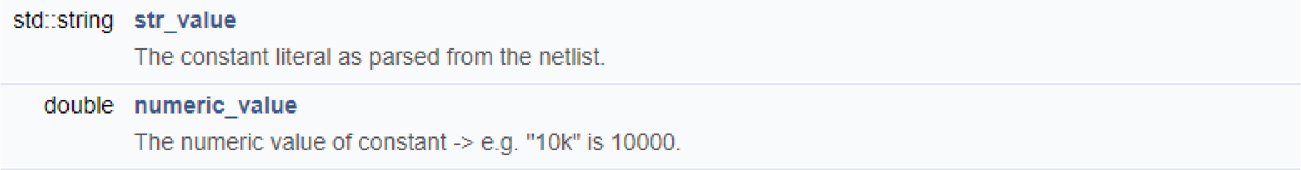
- Value_type: given that this class can be used to represent a variety of different components, it is useful to define what the nature of its characteristic value is. Therefore, an enum called value_type is defined which can have a value of CONSTANT_VAL (used for linear components and DC sources), FUNCTION_VAL (used for AC sources that are characterized by an amplitude and a phase)[42] and MODEL_VAL (for nonlinear components – e.g., BJTs where a distinction must be made between NPN and PNP devices).

Pointers to CONSTANT_VAL, FUNCTION_VAL and MODEL_NAME are unique pointers to data structures. These value classes each have different data members[43] :

[42] NB: Frequency is a defined by the simulation command. Not the component value…

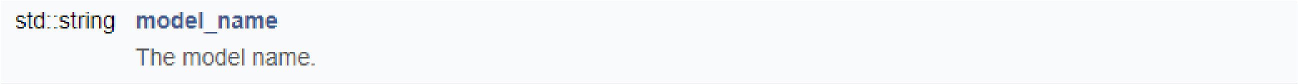
[43] Member functions not listed below.

- Constant_value:



std::string	**str_value** The constant literal as parsed from the netlist.
double	**numeric_value** The numeric value of constant -> e.g. "10k" is 10000.

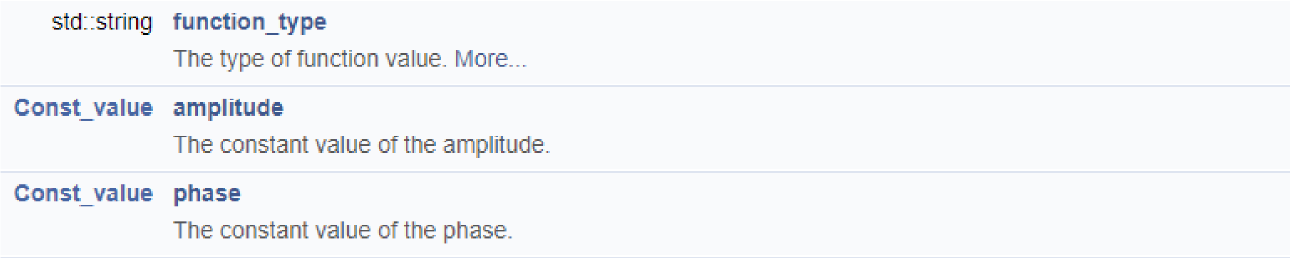
- Model_value:



std::string	**model_name** The model name.

The model_name can have any shape which signifies a predefined custom transistor (e.g., NPN, PNP, NMOS or PMOS).

- Function_value:



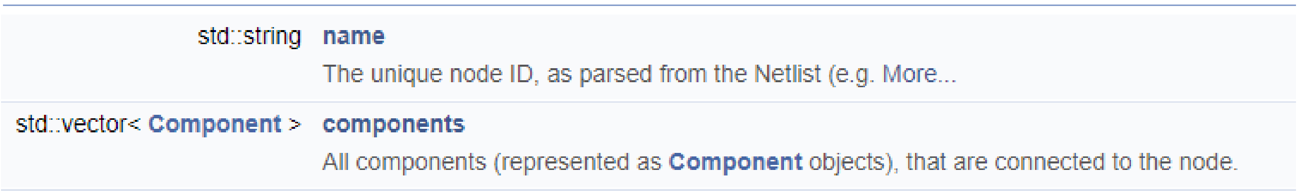
std::string	**function_type** The type of function value. More...
Const_value	**amplitude** The constant value of the amplitude.
Const_value	**phase** The constant value of the phase.

  - Function_type: a string specifying the type of simulation (always AC in the current implementation).
  - Amplitude: a CONSTANT_VAL containing the amplitude of the source.
  - Phase: a CONSTANT_VAL containing the phase of the source).

Node

Nodes are also represented with their own class which has the following members.

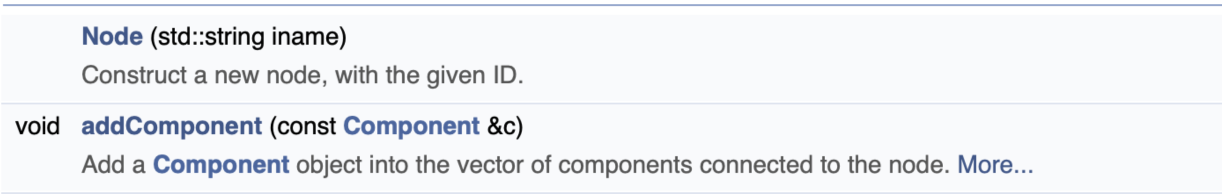
- Data:



std::string	**name** The unique node ID, as parsed from the Netlist (e.g. More...
std::vector< **Component** >	**components** All components (represented as **Component** objects), that are connected to the node.

  - Name: a string containing the name of a node (typically N followed by a unique number or out for the selected output node for simulation).
  - components: a vector containing all the components connected to the node.

- Functions:



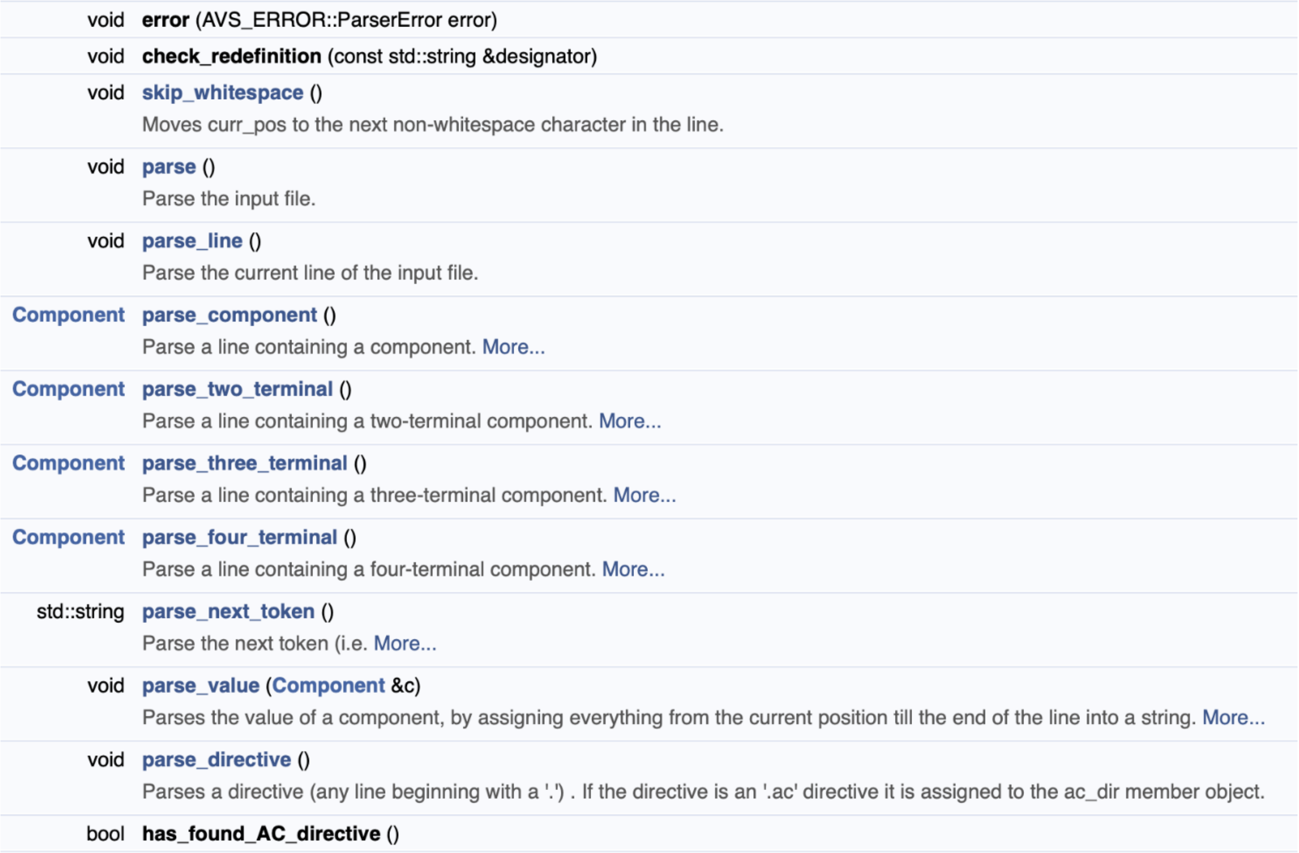
	Node (std::string iname) Construct a new node, with the given ID.
void	**addComponent** (const **Component** &c) Add a **Component** object into the vector of components connected to the node. More...

  - addComponent: function which adds a `component` object into the vector of components connected to the node.

Parser

The parser is in charge of taking in the Netlist and assigning the data into their respective data structures for all components and directives.

- Functions:

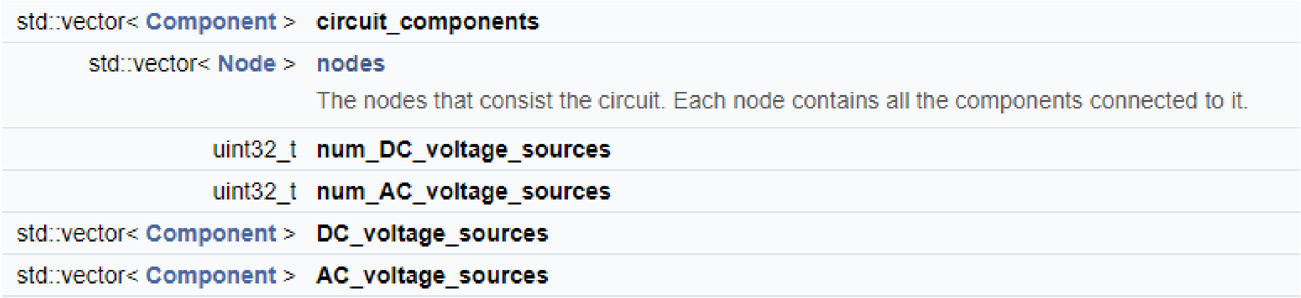


void	**error** (AVS_ERROR::ParserError error)
void	**check_redefinition** (const std::string &designator)
void	**skip_whitespace** () Moves curr_pos to the next non-whitespace character in the line.
void	**parse** () Parse the input file.
void	**parse_line** () Parse the current line of the input file.
Component	**parse_component** () Parse a line containing a component. More...
Component	**parse_two_terminal** () Parse a line containing a two-terminal component. More...
Component	**parse_three_terminal** () Parse a line containing a three-terminal component. More...
Component	**parse_four_terminal** () Parse a line containing a four-terminal component. More...
std::string	**parse_next_token** () Parse the next token (i.e. More...
void	**parse_value** (Component &c) Parses the value of a component, by assigning everything from the current position till the end of the line into a string. More...
void	**parse_directive** () Parses a directive (any line beginning with a '.') . If the directive is an '.ac' directive it is assigned to the ac_dir member object.
bool	**has_found_AC_directive** ()

Circuit

This class uses the node and component classes to represent a full circuit which can be manipulated by the other parts of the system that will aim to solve it. This class has the following noteworthy members:

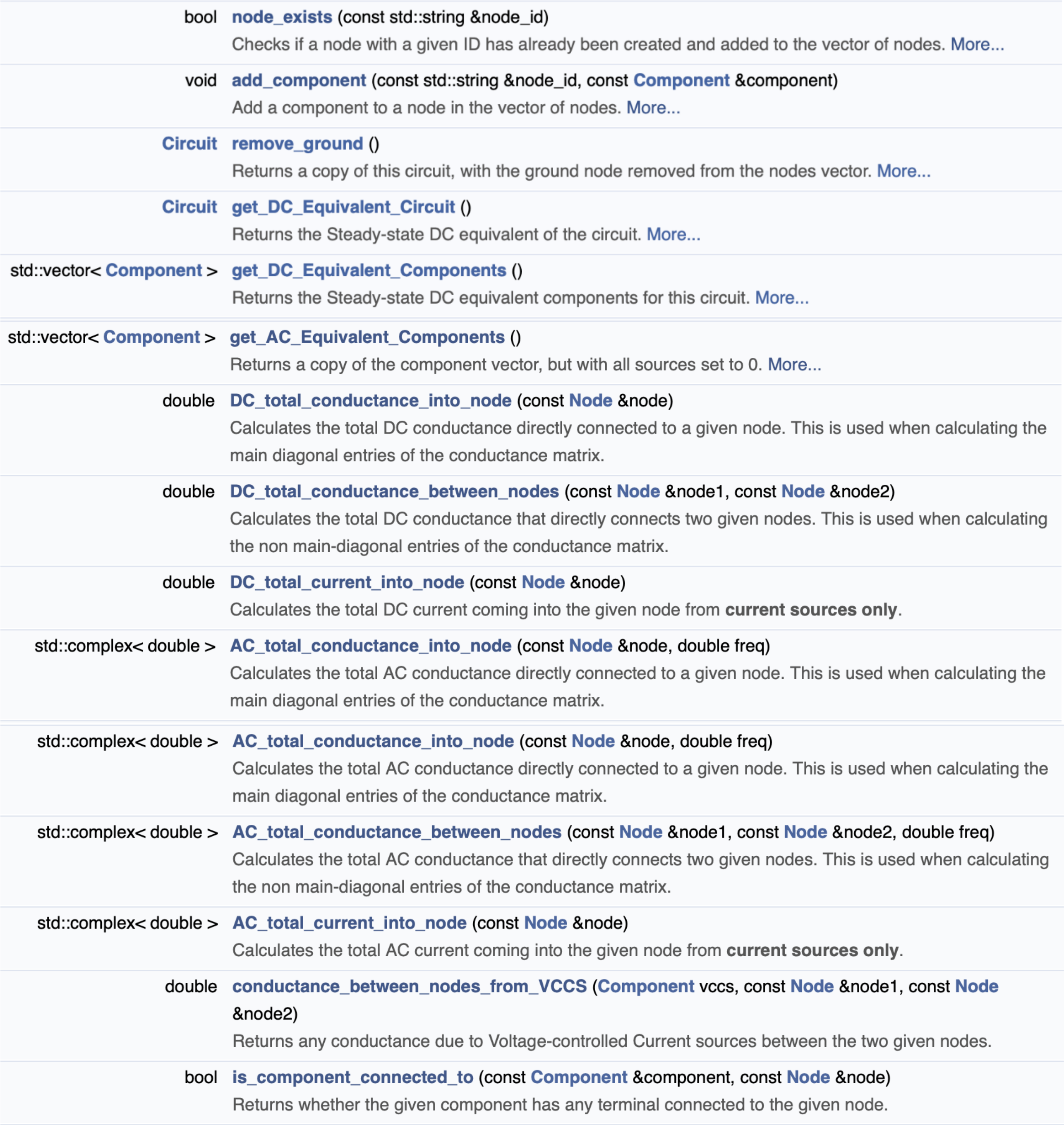
- Data:



std::vector< Component >	**circuit_components**
std::vector< Node >	**nodes** The nodes that consist the circuit. Each node contains all the components connected to it.
uint32_t	**num_DC_voltage_sources**
uint32_t	**num_AC_voltage_sources**
std::vector< Component >	**DC_voltage_sources**
std::vector< Component >	**AC_voltage_sources**

  - circuit_components: a vector of all components in the circuit.
  - nodes: a vector of all the nodes in the circuit
  - DC_voltage_sources: a vector containing all the DC voltage sources as components.
  - AC_voltage_sources: a vector containing all AC voltage sources as components.

- Functions:



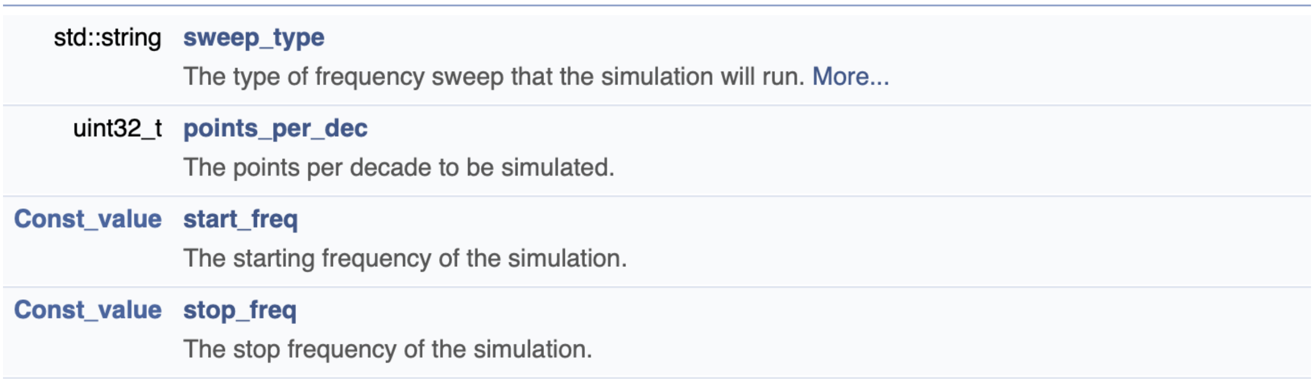
bool	**node_exists** (const std::string &node_id) Checks if a node with a given ID has already been created and added to the vector of nodes. More...
void	**add_component** (const std::string &node_id, const **Component** &component) Add a component to a node in the vector of nodes. More...
Circuit	**remove_ground** () Returns a copy of this circuit, with the ground node removed from the nodes vector. More...
Circuit	**get_DC_Equivalent_Circuit** () Returns the Steady-state DC equivalent of the circuit. More...
std::vector< **Component** >	**get_DC_Equivalent_Components** () Returns the Steady-state DC equivalent components for this circuit. More...
std::vector< **Component** >	**get_AC_Equivalent_Components** () Returns a copy of the component vector, but with all sources set to 0. More...
double	**DC_total_conductance_into_node** (const **Node** &node) Calculates the total DC conductance directly connected to a given node. This is used when calculating the main diagonal entries of the conductance matrix.
double	**DC_total_conductance_between_nodes** (const **Node** &node1, const **Node** &node2) Calculates the total DC conductance that directly connects two given nodes. This is used when calculating the non main-diagonal entries of the conductance matrix.
double	**DC_total_current_into_node** (const **Node** &node) Calculates the total DC current coming into the given node from **current sources only**.
std::complex< double >	**AC_total_conductance_into_node** (const **Node** &node, double freq) Calculates the total AC conductance directly connected to a given node. This is used when calculating the main diagonal entries of the conductance matrix.
std::complex< double >	**AC_total_conductance_into_node** (const **Node** &node, double freq) Calculates the total AC conductance directly connected to a given node. This is used when calculating the main diagonal entries of the conductance matrix.
std::complex< double >	**AC_total_conductance_between_nodes** (const **Node** &node1, const **Node** &node2, double freq) Calculates the total AC conductance that directly connects two given nodes. This is used when calculating the non main-diagonal entries of the conductance matrix.
std::complex< double >	**AC_total_current_into_node** (const **Node** &node) Calculates the total AC current coming into the given node from **current sources only**.
double	**conductance_between_nodes_from_VCCS** (**Component** vccs, const **Node** &node1, const **Node** &node2) Returns any conductance due to Voltage-controlled Current sources between the two given nodes.
bool	**is_component_connected_to** (const **Component** &component, const **Node** &node) Returns whether the given component has any terminal connected to the given node.

- total_conductance_into_node: This function calculates the sum of the conduces of the components connected to a node. This will be needed to construct matrices for MNA. There are both DC (returning a double) and AC (returning a complex double) versions of this function.

AC_directive

The AC_directive stores the parameters of the AC simulation requested by the input line starting with `.ac`.

- Data:

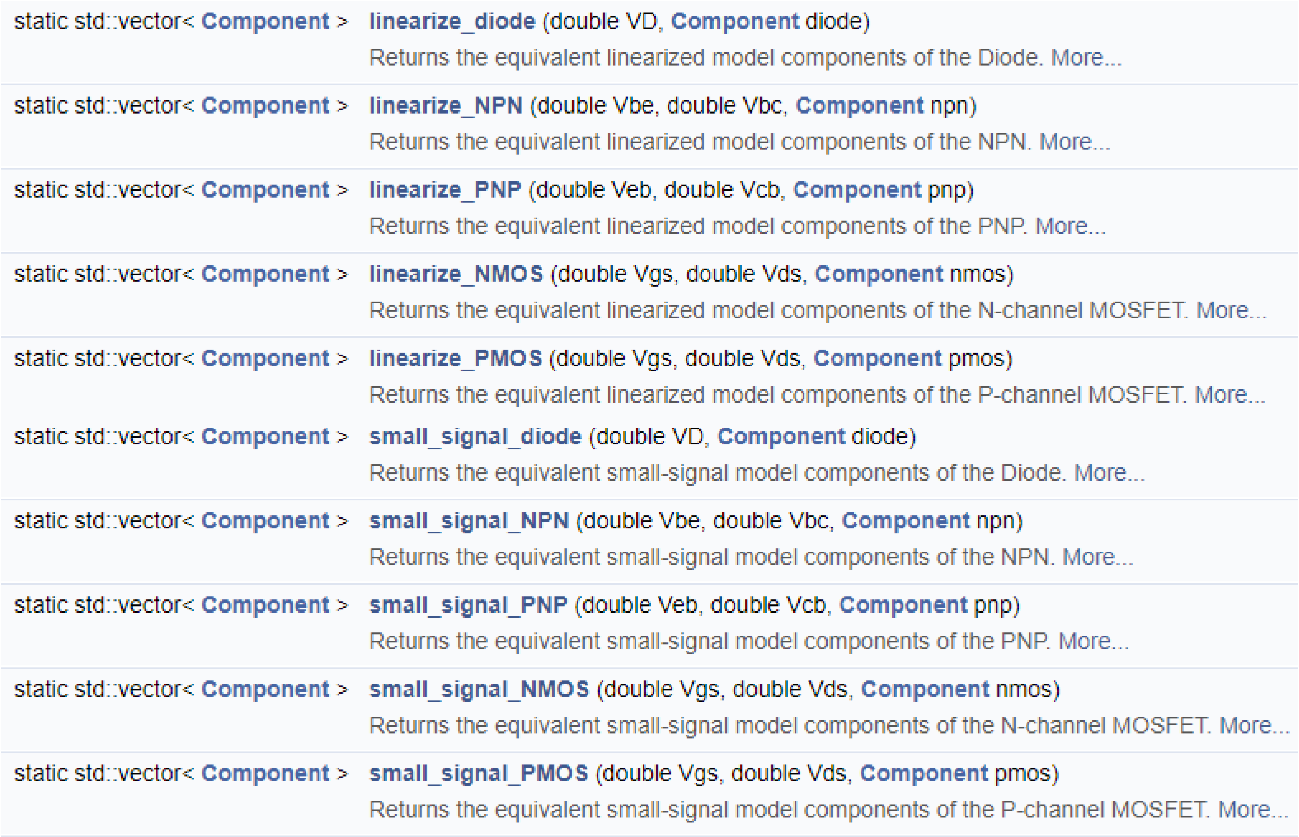


std::string	**sweep_type** The type of frequency sweep that the simulation will run. More...
uint32_t	**points_per_dec** The points per decade to be simulated.
Const_value	**start_freq** The starting frequency of the simulation.
Const_value	**stop_freq** The stop frequency of the simulation.

Linearizer

The `Linearizer` class converts the non-linear components into their linear large signal or small signal equivalent for either DC_Simulator or AC_Simulator respectively. All functions are strongly connected to the models.h file which defines the predefined parameters of the custom transistors.

- Functions:

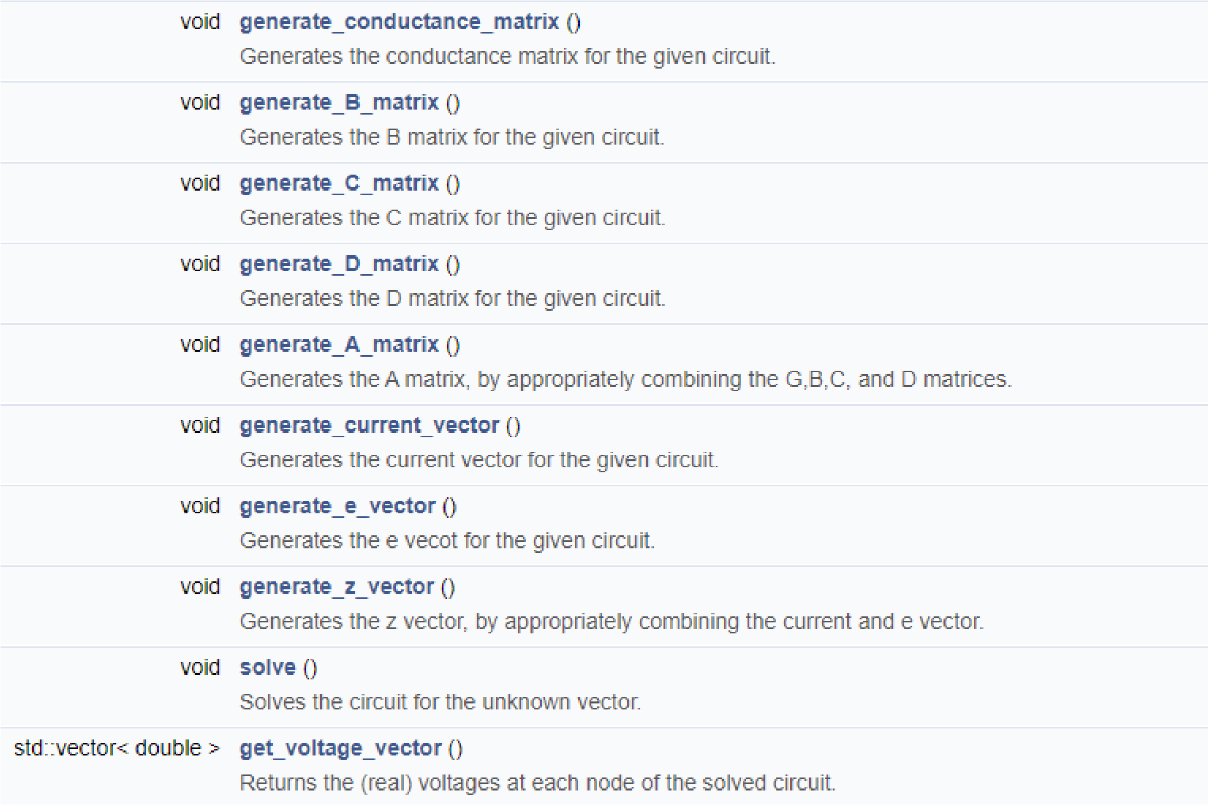


static std::vector< **Component** >	**linearize_diode** (double VD, **Component** diode) Returns the equivalent linearized model components of the Diode. More...
static std::vector< **Component** >	**linearize_NPN** (double Vbe, double Vbc, **Component** npn) Returns the equivalent linearized model components of the NPN. More...
static std::vector< **Component** >	**linearize_PNP** (double Veb, double Vcb, **Component** pnp) Returns the equivalent linearized model components of the PNP. More...
static std::vector< **Component** >	**linearize_NMOS** (double Vgs, double Vds, **Component** nmos) Returns the equivalent linearized model components of the N-channel MOSFET. More...
static std::vector< **Component** >	**linearize_PMOS** (double Vgs, double Vds, **Component** pmos) Returns the equivalent linearized model components of the P-channel MOSFET. More...
static std::vector< **Component** >	**small_signal_diode** (double VD, **Component** diode) Returns the equivalent small-signal model components of the Diode. More...
static std::vector< **Component** >	**small_signal_NPN** (double Vbe, double Vbc, **Component** npn) Returns the equivalent small-signal model components of the NPN. More...
static std::vector< **Component** >	**small_signal_PNP** (double Veb, double Vcb, **Component** pnp) Returns the equivalent small-signal model components of the PNP. More...
static std::vector< **Component** >	**small_signal_NMOS** (double Vgs, double Vds, **Component** nmos) Returns the equivalent small-signal model components of the N-channel MOSFET. More...
static std::vector< **Component** >	**small_signal_PMOS** (double Vgs, double Vds, **Component** pmos) Returns the equivalent small-signal model components of the P-channel MOSFET. More...

DC_Simulator

`DC_Simulator` finds the unknown voltage vector by calculating the LU decomposition on the A_matrix formed by the Modified Nodal Analysis.

- Functions:

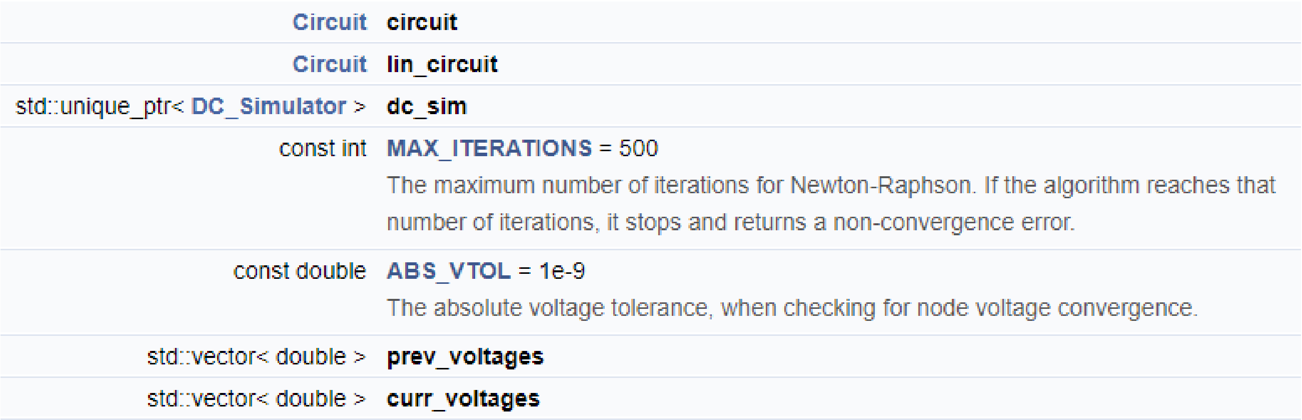


void	**generate_conductance_matrix** () Generates the conductance matrix for the given circuit.
void	**generate_B_matrix** () Generates the B matrix for the given circuit.
void	**generate_C_matrix** () Generates the C matrix for the given circuit.
void	**generate_D_matrix** () Generates the D matrix for the given circuit.
void	**generate_A_matrix** () Generates the A matrix, by appropriately combining the G,B,C, and D matrices.
void	**generate_current_vector** () Generates the current vector for the given circuit.
void	**generate_e_vector** () Generates the e vecot for the given circuit.
void	**generate_z_vector** () Generates the z vector, by appropriately combining the current and e vector.
void	**solve** () Solves the circuit for the unknown vector.
std::vector< double >	**get_voltage_vector** () Returns the (real) voltages at each node of the solved circuit.

OP_Point_Solver

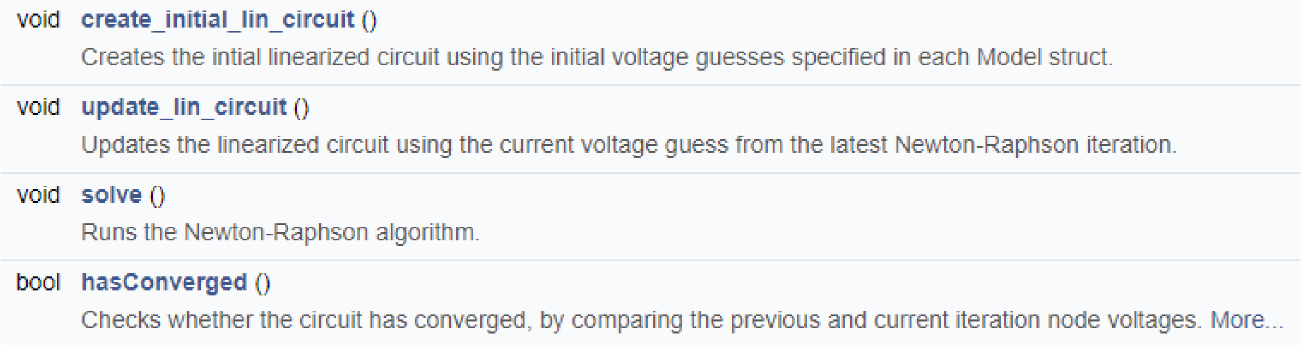
This class uses the Newton-Raphson methodology to find the convergence voltage points of the voltage vector calculated in the DC_Simulator.

- Data:



Circuit	**circuit**
Circuit	**lin_circuit**
std::unique_ptr< DC_Simulator >	**dc_sim**
const int	**MAX_ITERATIONS** = 500 The maximum number of iterations for Newton-Raphson. If the algorithm reaches that number of iterations, it stops and returns a non-convergence error.
const double	**ABS_VTOL** = 1e-9 The absolute voltage tolerance, when checking for node voltage convergence.
std::vector< double >	**prev_voltages**
std::vector< double >	**curr_voltages**

- Functions:

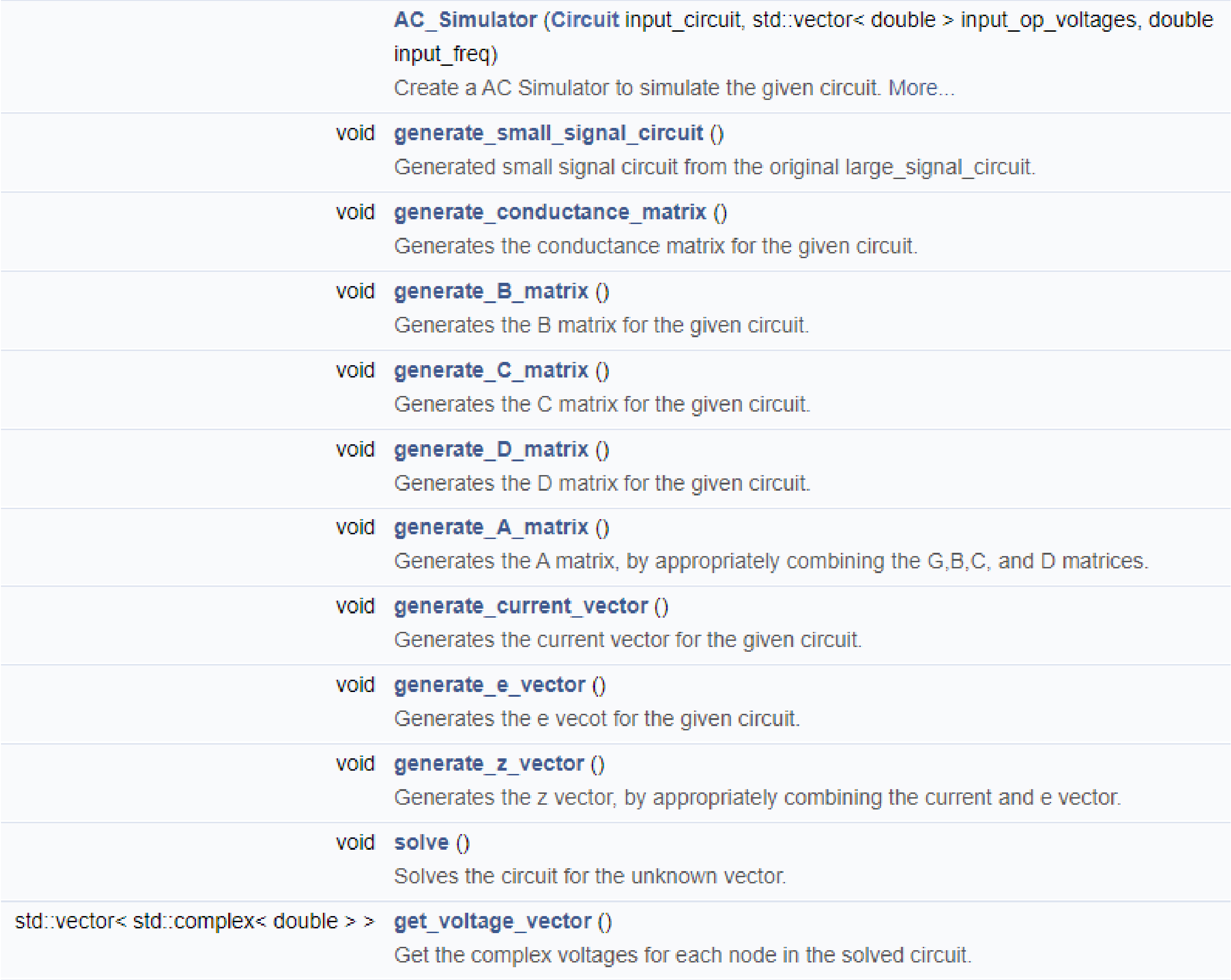


void	**create_initial_lin_circuit** () Creates the intial linearized circuit using the initial voltage guesses specified in each Model struct.
void	**update_lin_circuit** () Updates the linearized circuit using the current voltage guess from the latest Newton-Raphson iteration.
void	**solve** () Runs the Newton-Raphson algorithm.
bool	**hasConverged** () Checks whether the circuit has converged, by comparing the previous and current iteration node voltages. More...

AC_Simulator

This class calculates the final voltage vector through Modified Nodal Analysis for every frequency step requested by the range in the `AC_directive`.

- Functions:

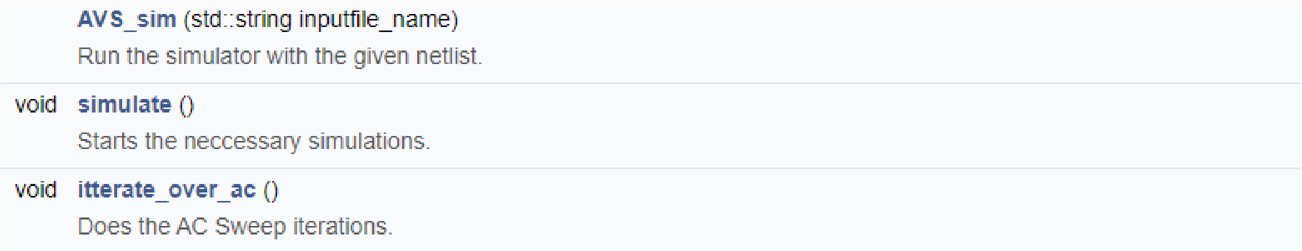


	AC_Simulator (**Circuit** input_circuit, std::vector< double > input_op_voltages, double input_freq) Create a AC Simulator to simulate the given circuit. More...
void	**generate_small_signal_circuit** () Generated small signal circuit from the original large_signal_circuit.
void	**generate_conductance_matrix** () Generates the conductance matrix for the given circuit.
void	**generate_B_matrix** () Generates the B matrix for the given circuit.
void	**generate_C_matrix** () Generates the C matrix for the given circuit.
void	**generate_D_matrix** () Generates the D matrix for the given circuit.
void	**generate_A_matrix** () Generates the A matrix, by appropriately combining the G,B,C, and D matrices.
void	**generate_current_vector** () Generates the current vector for the given circuit.
void	**generate_e_vector** () Generates the e vecot for the given circuit.
void	**generate_z_vector** () Generates the z vector, by appropriately combining the current and e vector.
void	**solve** () Solves the circuit for the unknown vector.
std::vector< std::complex< double > >	**get_voltage_vector** () Get the complex voltages for each node in the solved circuit.

AVS-sim

This file brings together the simulations and completes the final simulation step by writing the results onto an output.txt file.

- Functions:



AVS_sim (std::string inputfile_name)
Run the simulator with the given netlist.

void **simulate** ()
Starts the neccessary simulations.

void **itterate_over_ac** ()
Does the AC Sweep iterations.

3. Run time of the program before and after optimisation on all OSs tested.

	Runtime in Linux (AMD Ryzen 7 4700U @ 2.0GHz)		**Runtime in MacOS (Intel Core i7-6700HQ @ 2.6GHz)**		**Runtime in Windows 10 (Intel Core i7-7500U @ 2.7GHz)**	
Example Name	**Before (ms)**	**After(ms)**	**Before (ms)**	**After(ms)**	**Before (ms)**	**After(ms)**
Common emitter amplifier (Figure 18)	42.1	14.6	155.37	43.13	172.5	44.3
RLC Filter (Figure 27)	13.8	8.1	32.21	15.71	27.9	21.9
NMOS CS Amplifier (Figure 30)	35.2	15.1	139.88	40.78	132.7	43.8
NPN Differential Amplifier (Figure 33)	70.6	24.7	234.2	64.77	216.2	92.1
Darlington Pair (Figure 36)	104.4	27.4	205.2	52.2	222.6	62.5

4. .op simulation time comparisons

(AMD Ryzen 7 4700U @ 2.0GHz)	**LTSpice**	**AVS**
Example Name	**(ms)**	**(ms)**
Common emitter amplifier (Figure 18)	65	0.734
RLC Filter (Figure 27)	64	0.263
NMOS CS Amplifier (Figure 30)	78	0.893
NPN Differential Amplifier (Figure 33)	81	1.414
Darlington Pair (Figure 36)	85	1.628

5. Modified Nodal Analysis Examples

1) Potential divider:

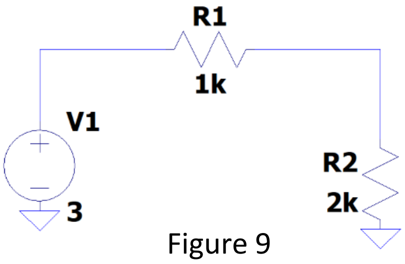


Figure 9 [44]

Following the rules for creating the conductance matrix we would get the following result:

$$G = \begin{pmatrix} 10^{-3} + 5 \times 10^{-4} & -10^{-3} \\ -10^{-3} & 10^{-3} \end{pmatrix}$$

Then, considering the voltage source (and ignoring the connection to ground).

$$A = \begin{pmatrix} 10^{-3} + 5 \times 10^{-4} & -10^{-3} & 0 \\ -10^{-3} & 10^{-3} & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{With} \quad x = \begin{pmatrix} V_{nout} \\ V_{n2} \\ I_{v1} \end{pmatrix} \text{ and } \quad z = \begin{pmatrix} I_{nout} \\ I_{n2} \\ V_1 \end{pmatrix}$$

Once these matrices are produced: they are ready to be solved. Any matrix calculator can be used to verify that solving the matrix equation above will give the correct unknown nodal voltages.

2) Dependent current source:

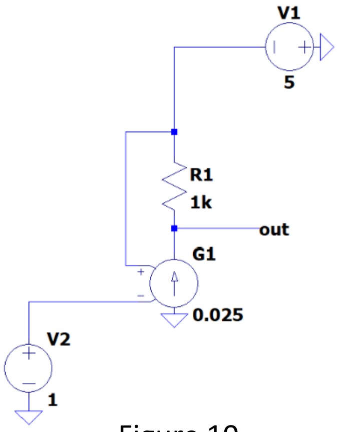


Figure 10 [45]

First the conductance matrix is constructed:

$$G = \begin{pmatrix} 10^{-3} & -10^{-3} & 0 \\ -10^{-3} & 10^{-3} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

[44] Our illustration: made in LT Spice XVII

[45] Our illustration: made in LT Spice XVII

Then, the B, C and D matrices are added to the A matrix (considering independent sources):

$$A = \begin{pmatrix} 10^{-3} & -10^{-3} & 0 & 0 & 0 \\ -10^{-3} & 10^{-3} & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{pmatrix} \quad \text{with} \quad x = \begin{pmatrix} V_{nout} \\ V_{n2} \\ V_{n3} \\ I_{v1} \\ I_{v2} \end{pmatrix} \text{and} \quad z = \begin{pmatrix} I_{nout} \\ I_{n2} \\ I_{n3} \\ V_1 \\ V_2 \end{pmatrix}$$

Finally, conductances associated with the dependent current source are inserted (ignoring those that, due to the connection to ground, are not associated with any column/row index).

$$A = \begin{pmatrix} 10^{-3} & -10^{-3} - 0.025 & 0.025 & 0 & 0 \\ -10^{-3} & 10^{-3} & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{pmatrix}$$

Now A and z can be used to solve the circuit.

References

[1] R. M. Kielkowski, Inside SPICE, Second Edition ed., McGraw-Hill, Inc., 1998, pp. 1-2, 49-58.

[2] E. Cheever, “An Algorithm for Modified Nodal Analysis,” Swarthmore College, 2011. [Online]. Available: https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA3.html. [Accessed May 2021].

[3] E. Cheever, “Advanced SCAM with dependent sources,” Swarthmore College, 2011. [Online]. Available: https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNADep.html. [Accessed May 2021].

[4] F. N. Najm, “Linearizarion of Multiterminal Elements,” in *Circuit Simulation*, Johm Wiley & Sons, Inc., 2010, pp. 171-176.

[5] A. L. Steven Herbst, “Companion Models for Basic Non-Linear and Transient Devices,” 30 December 2008. [Online]. Available: http://dev.hypertriton.com/edacious/trunk/doc/lec.pdf. [Accessed June 2021].

[6] Tuxfamily, “Eigen,” 19 April 2021. [Online]. Available: https://eigen.tuxfamily.org/index.php?title=Main_Page.

[7] Visual-Paradigm, “VP Online - Diagrams,” Visual-Paradigm, 2021. [Online]. Available: https://online.visual-paradigm.com/app/diagrams/. [Accessed June 2021].

[8] S. Edward, “EE1 Project 2021–Circuit Simulator File Format,” 2021. [Online]. [Accessed May 2021].