Development of Open Source and Terminal Based Analog Circuit Simulator

M.A. Salam¹, A. T. Kersü¹, Z. D. Salman¹, C. Rollas², and A. Gumus²

¹Fanshawe College
²Izmir Institute of Technology

April 25, 2023

Abstract

Circuit simulators are complex tools that depend highly on modeling of electrical circuit components. Therefore, employing an open-source circuit simulator within individuals’ learning processes of electronics can be beneficial, as in both a stronger grasp of circuit component as well as circuit theory, and deeper understanding of software development. This paper demonstrates the development of an analogous circuit simulator of passive electrical components, aiming to provide a useful and open-source tool that can be used as a study project or be expanded by being further developed for practice purposes. In order to validate the correctness of the outputs of the terminal-based simulator, the simulation results of an exemplary circuit of the terminal-based simulator were compared to the simulation results of LTSpice for the same circuit.

Development of Open Source and Terminal Based Analog Circuit Simulator

¹A. T. Kersü, ²Z. D. Salman, ³C. Rollas, ⁴M.A. Salam*, ⁵A. Gumus*

¹Faculty of Applied Science and Technology
Fanshawe College
London, Ontario, Canada
a_kersu@fanshawonline.ca

²Faculty of Engineering
Izmir Institute of Technology
London, Ontario, Canada
zdsalman97@gmail.com

³Faculty of Engineering
Izmir Institute of Technology
Gulbahce, Izmir, Turkey
canrollas@gmail.com

⁴Faculty of Applied Science and Technology
Fanshawe College
London, Ontario, Canada
Circuit simulators are complex tools that depend highly on modeling of electrical circuit components. Therefore, employing an open-source circuit simulator within individuals’ learning processes of electronics can be beneficial, as in both a stronger grasp of circuit component as well as circuit theory, and deeper understanding of software development. This paper demonstrates the development of an analogous circuit simulator of passive electrical components, aiming to provide a useful and open-source tool that can be used as a study project or be expanded by being further developed for practice purposes. In order to validate the correctness of the outputs of the terminal-based simulator, the simulation results of an exemplary circuit of the terminal-based simulator were compared to the simulation results of LTSpice for the same circuit.

1. Introduction

Circuit simulators are software tools that calculate the expected behaviors of real-life applications of circuits and aim to provide detailed information per component. They rely on detailed modeling of electrical circuit components, which are elements with two terminals that make up a circuit (Najm, F. N., 2010), and they perform numerical methods to solve input circuits. They are widely used in the fields of electrical and electronics engineering. These simulators enable designers to analyze and optimize complex circuits before they are physically built, thus, reducing the cost and time required for product development.

The C programming language is known to provide more control over memory and execute tasks faster compared to the other languages (Kwame, A. E., et al. 2017). Hence, what it lacks in ease of development and its non-object-oriented nature, it makes up with execution speed and lower memory requirements. These properties would play a significant role in a software tool such as a circuit simulator. Hence, circuit simulators take advantage of these properties by employing “.c” files within their project files that carry out the computations. Therefore, the C programming language is an important part of the skillset of electrical engineers, as well as individuals who aim to work in related fields.

There are only a few circuit simulators whose source code is available online, and even fewer that are written in the C programming language. Most of the available simulators use object-oriented languages such as Java, which does not benefit the potential learners of C programming language. Additionally, many of these simulators are quite complex, with extensive circuit analysis capabilities and support for multiple types of analysis. This can make it difficult for users to choose where to start when trying to understand the code. Moreover, while there are some programming challenges for learners available online, many of them do not go beyond a certain level of difficulty and can be repetitive even when presented in different contexts. Therefore, it can be difficult for beginner programmers to find a project that challenges them to code an advanced program while teaching them various aspects of the language at the same time. (The Need) An open-source circuit simulator developed in C programming language can serve as an educational tool for individuals who have basic programming skills and aim to enhance their abilities by learning new concepts and developing complex algorithms. This software can provide a break from repetitive coding exercises and allow students or learners to engage in a substantial project, thereby gaining experience with circuit analysis methods and the application of formulas and concepts in an integrated development environment (IDE). Hence, by employing this simulator, individuals can reinforce their programming skills while expanding their knowledge of circuit analysis, which can ultimately serve as a valuable asset for future endeavors in the field.
In this study, the proposed idea is to facilitate the learning progress of individuals through an educational, open-source, and terminal-based circuit simulator that can run non-transient DC analysis for passive circuit components. In other words, the project aims to equip individuals with a higher level of proficiency in C programming, a better grasp of circuit analysis methods, and a deeper understanding of implementing circuit theory into the software. The simulator is not only an advanced programming exercise but is also a base for anyone who would like to further develop it by adding custom libraries or modifying the source code to implement various other circuit analysis techniques. Throughout the paper, several topics are discussed, such as implementing circuit theory in an integrated development environment, the workflow of SPICE simulators, and the benefits of using a simulator for educational purposes.

The educational value of the terminal-based simulator lies in its ability to facilitate learning about circuit analysis and programming simultaneously. While there are open-source circuit simulators available, such as the Falstad (Falstad, P. 2019) simulator, they can be complex and overwhelming for beginners to study and code themselves. This makes the terminal-based simulator a valuable resource for individuals who want to learn at their own pace.

There are numerous studies that have shown the benefits of using circuit simulators as educational tools. One study found that circuit simulators greatly assisted undergraduate students studying power electronics (Hart, D. W, 1993) and related courses. Another study compared the learning outcomes of a controlled group, where some members studied real circuits with the assistance of a simulator and some without it (Jaakkola, T., et al. 2008). The group using the simulator during the coursework achieved better results, highlighting the simulator’s effectiveness in aiding individuals in studying circuits.

Furthermore, as many undergraduate students in electrical and computer engineering departments are concurrently taking courses in power electronics or circuit analysis as well as programming courses in C or C++ programming languages, the terminal-based simulator can be a valuable tool to help them learn and apply their knowledge across multiple disciplines. By taking up this educational project on their own, students can enhance their understanding of circuit analysis and programming and develop important skills and knowledge that can benefit them in their academic and professional careers.

The key contributions of this study can be summarized as follows:

* Employing advanced capabilities of C programming language to include a comprehensive set of C language constructs and making use of nearly all of the C language’s fundamental elements.
* Creating complex algorithms for making connections between circuit components, setting a ground (GND) node, node detection in circuits, and handling matrix operations of arbitrary sizes.
* Creating custom libraries for component models, matrix calculations, as well as user interaction with the terminal-based simulator.
* Implementing theoretical knowledge of circuit design into terminal-based simulation software.
* Facilitating the learning process for individuals who study electrical circuits and programming.

The terminal-based simulator follows a similar, but a rather less complex, approach to SPICE simulators, as in using Kirchhoff’s Current Law (Node Analysis) to analyze the input circuits. The terminal-based simulator can be thought of as a stripped-down version of other circuit simulators, such as SPICE simulators, due to its lesser scope of non-transient DC analysis. This would be to the benefit of individuals as it means neither the programming part nor the circuit theory behind the simulator is overly complex and would encourage beginners to work on it. The flowchart of the terminal-based simulator’s workflow can be seen in Figure 1.
2. Methods

2.1 Background on Computer Aided Circuit Analysis

A typical algorithm of a SPICE program, which was one of the first simulators (Pederson, D., 1984), would be following the steps of initialization, creating of linear component models, matrix formulation, calculation, and output, with some conditional steps of going back to step 2 in cases where output does not converge. To briefly explain these points, initialization refers to the arrangement of circuit topology and specifying some default values -unless user input on it is available- such as operating temperature. Creating linear component models is a step that is omitted in linear circuit analysis which is replacing non-linear device models with an equivalent model that is linear. Matrix formulation and calculation consists of using the Gauss Elimination Method to solve the matrix composed of, say, voltage values of circuit nodes, and lastly, the output is where the user is provided with the analysis results. As far as the terminal-based simulator in this paper is concerned, step 2 of creating linear component models for non-linear devices is skipped as it is out of scope, and the rest of the steps are demonstrated in the following subsections of methods section.

2.2 Modeling Components

The modeling of circuit components requires several parameters of different data types. Therefore, a composite data type, otherwise referred to with the “struct” keyword in the C programming language, is used for modeling. Elements of the struct are variables that store component and analysis-related information such as component type, component value, or the current flowing through the component. Moreover, component models have two elements of the same type called nodeInfo, which are models created for circuit nodes. The node models have three properties that are: a node ID, the node’s ground status, and node voltage.

Circuit elements are stored in the form of a singly linked list, which is the simple, unidirectional type of linked list. This facilitates the development process as it allows for the components to be located easily, as well as making it possible to perform an operation on every circuit component simply by providing the address of the head node and running the operation in a loop, performing it on each component one by one.

Hence, one of the elements of the component model is a pointer to the next circuit element in the list, that
is to form the linked list. The component model created for the terminal-based simulator can be seen in Figure 2.

![Component model and its input/output node model.](image)

Figure 2: Component model and its input/output node model.

### 2.3 Building Custom Libraries

Designing a software tool that simulates circuits involves performing a multitude of operations, ranging from implementing circuit analysis methods like nodal analysis into software, to doing numerous matrix operations. To accomplish these operations, custom header files were developed that store functions that perform tasks of similar contexts, namely `components.h`, `topology.h`, `calculator.h`, and `visual.h`. The component library, for instance, contains dedicated functions for adding components, identifying circuit nodes, and assigning nodes as ground (GND) nodes. The topology library, on the other hand, contains functions that determine the connections of parallel or series. The calculator library comprises mathematical functions for analysis, while the visual library contains the functions for displaying the main menu, sub-menus and switching between the sub-menus and the main menu.

### 2.4 Modeling the Circuit

#### 2.4.1 Nodes and Terminals

The simulator employs two distinct node definitions, namely trivial nodes and unique nodes. Trivial nodes represent node IDs that are assigned to newly added components by default, and do not correspond to the actual nodes of the circuit. For instance, if a user adds two resistors, the input and output terminals of the resistors will be assigned the node IDs (1,2) and (3,4), respectively. In a physical sense, considering the case where these resistors are connected in series, the trivial node IDs of 2 and 3 would refer to the same node. However, for programming purposes, the terminals of circuit elements must receive IDs that differ in ID, even though they may represent the same node in reality. Hence, these IDs are referred to as trivial. Unique nodes, on the other hand, represent the actual understanding of a node in a circuit, and are assigned to nodes only after users establish the connections between components. In other words, a unique node ID overwrites multiple trivial node IDs to comply with the definition of a circuit node. Consequently, in the aforementioned example, once the user connects the two resistors, the simulator overwrites the IDs of 2 and 3 with a new and unique node ID of 5. Figure 3 explains the trivial and unique node definitions of the simulator where trivial nodes are labeled as t4, t5, and t7, and the unique node is labeled as u16.
2.4.2 Adding Components to Circuit

Building a circuit by adding components is achieved by a simple function that fills in the information for a component model. Some of this information is generated by the program, such as the components trivial node IDs, component ID, ground status of its terminals which is $FALSE$ by default, and the other are user inputs, such as component type and component value. Users can add components by typing in component values such as $1000 \text{ pF}$ or $25 \text{ mH}$. Afterward, the program will parse the input string to obtain the information as in the component value being $1000$, order magnitude being $\text{p}$ which refers to $\text{pico}$ that is $10^{-12}$, and lastly, the component type being $F$ which stands for $\text{Farad}$ which would indicate that the input component is a capacitor. Furthermore, for safety purposes, component models do not store the component type in variables of type char, and instead use a more explicit way of storing a descriptive word about the component type. For example, if a user adds a component by typing in $30 \text{ mV}$, the program multiplies $30$ by $10^{-3}$ due to the order of magnitude being $\text{m}$ as in $\text{milli}$, and assigns the value to the variable $\text{componentValue}$, and then copies a string of $\text{voltage}$ into the char array type of variable $\text{componentType}$.

2.4.3 Node Combining & Grounding

The process of connecting two components in either series or parallel in the terminal-based simulator is similar to that of using a circuit simulator with a user interface (UI). In order to connect two components in a simulator with a UI, the user selects the $\text{draw net}$ tool, clicks either terminal of the first component, and then clicks a terminal of the second component to make the connection. Similarly, in the case of the terminal-based simulator, users can choose the $\text{draw net}$ option from the main menu, provide an array of component IDs, as well as an array of terminal IDs that specify whether the $\text{INPUT}$ or $\text{OUTPUT}$ terminal of the component would be connected. The simulator then proceeds to make the connections. During this process, the program first pulls all trivial and unique node IDs, generates a new and unused node ID that will be a unique node ID by definition, and overwrites the old node IDs. Neither trivial nor unique node IDs can be assigned the ID of zero, as the node ID of zero is exclusively reserved for the ground node. The process of assigning a node as the ground node is relatively simple. Users can select the $\text{set GND}$ option from the main menu and provide a component ID and specify which terminal of the component to label as ground. The program then pulls all component information and overwrites node IDs that match the user input as zero, effectively designating the node as the ground node. The reason the program pulls all component information while setting a ground node is due to the nodes not being connected in reality, but being individual nodes of the same unique node ID. It is recommended to make all connections in the circuit before setting a ground node. If a user sets a trivial node as the ground node before making connections, the ground node of ID zero may be overwritten by a unique node assignment resulting from connecting the components, leading to unexpected behavior in the circuit simulator. Figure 4 shows an example circuit that is used for test purposes during development, where the trivial nodes are denoted by a lowercase $t$ followed by their trivial node IDs, and the unique nodes are denoted by a lowercase $u$ followed by unique node IDs generated by the program after connections are made between the components.
2.5 Locating Unique Nodes and Initializing Node Voltages

The program stores two types of node data, namely, trivial and unique. During the analysis, only unique nodes are used as they are the actual representations of real circuit nodes. Therefore, unique nodes are needed to be located in the circuit topology. First, a temporary search variable of type *compInfo is created, which is a pointer to the component model type. Then, a function loops through the entire linked-list of components through this search variable whilst temporarily storing the node IDs related to each component in a pointer with adequate memory allocated beforehand. The first component’s input and output node IDs are assumed to be unique by default, since there are no other node IDs during that time of execution. The program then checks the node IDs of the next components, building up the unique node ID list while comparing them to the current list to ensure they are unique. If any node ID matches an ID that is on the list already, it is discarded; otherwise, it is added to the list as a unique node. Once the function finishes execution, it returns a struct data type called uniqueNodeData, which has two elements: a pointer to the list of unique node IDs and an integer that stores the total number of unique nodes.

To ensure proper circuit functionality, the presence of a source is necessary. This means that the node voltages of nodes that have voltage sources connected to them are known pre-analysis, including the ground node, which is zero Volts (0 V) by default. The program initializes these known node voltages by iterating through the linked-list and checking every component to determine if it is a voltage source. If any are found, the voltage value of the voltage source is written to its positive terminal’s node voltage. Additionally, since the simulator replaces multiple trivial nodes with the same number of identical unique nodes, initializing the voltage for other components having the same unique node ID at either of their terminals is crucial. A dedicated function checks all components and assigns the appropriate voltage value to those with a matching node ID to the one subject to the operation of voltage initialization, ensuring that the simulator does not overlook any voltage values in the circuit. Similarly, the function that initializes ground nodes operates in a similar manner. Users provide a component ID and which terminal of the component to label as ground, and the program sets the voltage of all components connected to that node to 0 V, as well as assigning the node ID to zero, which is reserved for the ground node. It is recommended to initialize the ground once after adding all components, and making all connections, so as not to have the reserved node ID for GND nodes of zero overwritten by unique node assignments.

2.6 Extracting Node Data

The circuit analysis process starts with some preparation steps, one of which is the extraction of node data. Extraction of node data may refer to several operations in the program, depending on where it is employed. The types of data extraction can be generating a list per node, of every other node of unknown voltage, of the components connected to it. Other types can be a list of component IDs that are connected to the input parameter node ID, as well as lists of unique node IDs and lists of nodes of unknown voltages in the
entire circuit. These data play a crucial role in the analysis part of the program, and lay a base for the simulator to be able to support possible future additions of performing analysis on more complex circuits. As an example of node data extraction, the following description explains the working behavior of how to list every other node of components that join on a common node. The program, first, loops through the available components and returns a list of component IDs that have either of their terminals connected to a given node ID. This operation lists an array of components that are directly connected to a particular node. Afterward, the program sorts the ID list in ascending order by using the bubble sort algorithm, which is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. The sorting step is not necessary, but it helps with development and bug-tracking, more specifically in the analysis part. Next, the program goes through the sorted list of components and finds the second node IDs of the components that are connected to the node of interest. This operation creates a list of adjacent nodes for a given node, which is then used to build a conductance matrix for further analysis.

2.7 Building Conductance Matrix and Solving the Circuit

The program uses matrix operations to do the analysis on input circuits, similar to SPICE simulators (Gajab, K. D., & Chitturi, A. K. 2013). Therefore, a conductance matrix has to be formed to be able to perform the analysis. A conductance matrix calls for the information obtained from node equations, where the number of nodes can be an arbitrary number of \( m \). Hence, the conductance matrix must be of arbitrary size. The arbitrary matrix size of \( m \) in this case, is essentially the number of nodes for which the node voltage is not known. Thus, the number of nodes with unknown voltage values that is determined by extracting the circuit’s node data sets the matrix size for this operation. The conductance matrix, the current matrix and the matrix of unknown voltages can be seen in equation (1).

\[
\begin{bmatrix}
G_{x1} \text{ amp} & G_{x2} \text{ amp} & \ldots & G_{xn} \text{ amp} \\
G_{y1} \text{ amp} & G_{y2} \text{ amp} & \ldots & G_{yn} \text{ amp} \\
\vdots & \vdots & \ddots & \vdots \\
G_{m1} \text{ amp} & G_{m2} \text{ amp} & \ldots & G_{mn} \text{ amp}
\end{bmatrix} \times
\begin{bmatrix}
V_x \\
V_y \\
\vdots \\
V_m
\end{bmatrix} =
\begin{bmatrix}
I_x \\
I_y \\
\vdots \\
I_m
\end{bmatrix}
\]  

(1)

The program takes a different approach to build the conductance matrix than it is done by hand. Instead of writing node equations and putting the coefficients of unknown node voltages in a matrix, the program pulls node data for the current node, which includes IDs of all components connected to that specific node and node IDs of every other node. Next, it loops through the list of adjacent node IDs whilst checking if they are of a known voltage value, or unknown, which have the value \( NaN \), a special value in the default library \(<\text{math.h}>\) of GNU that stands for \textit{Not a Number} . If the voltage value is known, the program divides that value by the component’s resistance that is in between the two nodes, and writes the value into the current matrix. If not, it writes the conductance value of the component into the conductance matrix. For example, in the case of the unique node of ID 16 of the test circuit, the program calculates the conductance value for each component whilst looping through the adjacent node list, all the while adding up the conductance values, such as \( \frac{1}{4} \) + \( \frac{1}{4} \) + \( \frac{1}{6} \) to get 0.916667 which is the \( G_{11} \) element of the G matrix. This way,
the program builds rows to be fed into the conductance matrix, as well as elements to feed into the current matrix. Later, to build the matrices, a function loops through all nodes, whilst storing a list of conductance rows in a pointer that has been allocated memory for, enough to store in conductance values, as well as the current values to be assigned to the current matrix. These pointers and current values are then fed into their respective matrices. Hence, the matrices are built up step by step, in a different fashion than how it is done by hand.

During the analysis process, through the preparation made beforehand, a few functions that do matrix operations are enough to calculate the analysis results for the input circuit. First, forward and backward Gauss Elimination is performed on the conductance matrix. Then, through basic matrix algebra, the conductance matrix is multiplied from the left side by its inverse matrix, and so is the zero matrix on the opposite side of the equation. This yields the unknown voltage matrix being equal to the inverse of conductance matrix, as seen in equations (2) and (3).

\[
[G]^{-1} \times [G] \times [V] = [G]^{-1} \times [I](2)
\]

\[
[V] = [G]^{-1} \times [I](3)
\]

The reason why the program does both Gauss Elimination and inverse matrix operations, when either could have been enough to calculate the results, is for the goal of laying the basis for future upgrades to the simulator. Users who want to customize the terminal-based simulator can choose to go with either Gauss Elimination, or inverse matrix and matrix multiplication. Though this does not have a big impact on matrices of sizes up to 3 by 3, which is the default maximum size, there may occur significant differences in the development process as well as the execution time, depending on the matrix size of the operation.

The voltage matrix that is solved for, as mentioned previously, can be considered as the analysis results only partially. For the next step, the program applies Ohm’s law of \( V / R = I \) to find the current values. As all node voltages are known at this point, and so are the component values, the program subtracts the input and output node values of components from each other, and divides by the component value, only to finish off by assigning the calculated current value to the dedicated variable for that in the component model, for every component. This part concludes the non-transient DC analysis of passive circuit elements. Figure 5 shows the analysis results for the aforementioned example circuit.

![Figure 5: Terminal-based simulator outputs for the example circuit.](image)

### 3. Results and Discussions

#### 3.1 Exemplar Circuit

The example circuit mentioned in the paper is a resistive circuit with two fixed DC voltage sources. The circuit has two nodes whose node voltages are unknown, which are the unique nodes with IDs \( u16 \) and \( u17 \). Figure 6 shows the results of calculations regarding these nodes, and their voltage values as 14.22 V and...
18.22 V respectively which are marked inside a red rectangle on the bottom right of the screenshot. For comparison, on the left side of the figure, can be seen LTSpice outputs of the same circuit.

Figure 6: Comparison of the previously unknown node voltages between LTSpice and terminal-based simulator.

3.2 Simulation Results

In any circuit simulator, such as LTSpice, current values in analysis results sometimes tend to be negative numbers, which is nothing but an indicator of the direction in which the current flows. This tends to happen due to the nature of modeling of a component, which does not meet exactly the real-life versions of them (C. C. McAndrew, 2003). In reality, there are circuit components that do not have polarity, such as resistors. However, in programming, for the purposes of modeling a component, the two terminals must be assigned different IDs that result in a component such as a resistor appearing as if it is a polarized circuit element. Therefore, the negative values of currents only represent the flow direction of the current and may differ from simulator to simulator, as well as from circuit to circuit on the same simulator. Figure 7 shows the currents flowing through all of the five resistors in the example circuit and was obtained by running a simulation on LTSpice. Comparing the results on this figure to the simulator outputs provided under section 2.7 validates the outputs of the terminal-based simulator.

Figure 7: LTSpice results for comparison with terminal-based simulator outputs in section 2.7.
3.3 Possible Improvements

In the implementation of the terminal-based simulator discussed in this paper, Nodal Analysis is utilized for each node of unknown voltage. Additionally, matrix operations, including forward and backward Gauss Elimination, the inverse of a matrix, and matrix multiplication are employed. The program’s ability to handle matrices of varying sizes determines the complexity of circuits it can solve. For instance, adding a function to perform matrix operations for an \( n \times n \) matrix, where \( n \) is a positive integer, would enable the simulator to solve circuits with up to \( n \) nodes with unknown voltage values. The default limit in the simulator is set to 3 by 3 for simplicity, which allows for the solution of simple circuits that have up to three nodes whose voltages are not known pre-analysis. Understanding the limitations and capabilities of the simulator is essential for users when selecting and designing circuits to be simulated, as well as choosing how to make their own additions to the simulator. In terms of flexibility, the terminal-based simulator is designed to support the addition of upgrades to expand its capabilities beyond the default limit of 3 nodes of unknown voltages. Specifically, the program was developed in a way that it would support the pre-calculation part of the analysis, and the limit only applies to the dedicated matrix operation functions found in <calculator.h>. This means that users who study this educational tool can add a few functions that would handle larger matrices, such as four by four or larger, allowing the terminal-based simulator to solve more complex circuits. In fact, the ability to add upgrades to the simulator is one of its greatest strengths. The program was developed with a modular architecture, allowing users to easily modify or extend its functionality by adding new features, such as additional analysis methods or support for new types of circuit elements. This flexibility makes the simulator a valuable educational tool for students or individuals interested in the subject, as it allows them to explore and experiment with different circuit designs and analysis techniques. In addition to its existing functionality, the terminal-based simulator also has functions that can detect components connected in series or parallel. This feature provides a strong foundation for future expansion, as users could potentially incorporate additional circuit analysis techniques that involve identifying components in series or parallel. Furthermore, the simulator has the ability to include capacitors and inductors in circuits, making it possible to analyze circuits with AC sources and perform AC analysis. This expansion of capabilities further enhances the educational value of the simulator, as individuals studying it can apply the concepts of AC circuits and analysis to their work. The flexibility of the simulator means that it can be used in a variety of settings, from academic classrooms to personal projects, and can be adapted to meet the needs of different users.

3.4 Educational Approach

Throughout the paper, an open-source and terminal-based circuit simulator has been presented that can run non-transient DC analysis for passive circuit components, developed using the C programming language. The terminal-based simulator’s intended purpose is to facilitate the learning process of individuals studying it in many subjects such as circuit analysis and circuit theory, C programming language, implementing theoretical knowledge in a software platform, and complement any other related work individuals may have.

The terminal-based simulator is a beneficial educational tool due to its open-source nature. The source code can be downloaded and modified to create user-custom versions, providing individuals with a deeper understanding of how theoretical knowledge can be implemented into software programs. The project files contain detailed and explanatory comments that provide guidance on each function’s purpose and how they are tied together to meet the program’s intended use. By reviewing the building blocks of the program, individuals can gain a better understanding of how software engineering concepts and practices are applied from real-world scenarios.

Furthermore, instructors of programming courses can also adopt the terminal-based simulator as an educational tool. The source code can be broken down into parts to be completed weekly, and students can be assigned these parts as weekly lab work, to have programmed a functioning circuit simulator by the end of the course or term. This would enable students to apply the theoretical knowledge they have learned in the classroom to practical scenarios, leading to a better understanding of the subject matter.
Overall, it would be highly beneficial for individuals studying circuit theory, power electronics, or related subjects to follow along with the provided steps needed to program a circuit simulator of their own, either at their own pace or through assignments given by an instructor. By doing so, they can gain a deeper understanding of the subject matter, improve their programming skills, and be better equipped to apply theoretical knowledge to real-world scenarios.

Conclusions

The terminal-based simulator is a useful educational tool for individuals interested in learning about circuit analysis and programming. It provides users with the ability to input circuit schematics and analyze them using Nodal Analysis, as well as perform matrix operations using Gauss Elimination and basic matrix algebra such as the inverse of a matrix or matrix multiplication. The simulator, by default, is designed to handle circuits with up to three nodes that are of unknown voltage, but has the foundations to be expanded to handle more complex circuits by adding additional functions. A simple DC circuit with several resistance and voltage sources is considered and analyzed using the proposed method. The node voltages and the associated currents are calculated. The same circuit is designed in the LTSpice and the results of both simulators were compared, therefore, validating the terminal-based simulator’s operation. The terminal-based simulator has significant educational value as it allows individuals to learn about circuit analysis and programming simultaneously. By using the simulator, users can experiment with different circuit configurations and analyze their behavior in a safe and controlled environment. Furthermore, the program’s modular design and use of custom libraries provide users with insight into software engineering concepts and practices. In the future, the simulator can be expanded as it includes additional features such as the ability to detect components connected in series or parallel, which would allow for different types of circuit analysis methods. Additionally, the program has component models for capacitors and inductors, allowing for the future addition of AC sources and AC analysis. With these additions, the simulator can be used to analyze a wider range of circuits and provide a more comprehensive learning experience for users. Overall, the terminal-based simulator is a valuable educational tool for individuals interested in learning about circuit analysis and programming. Its modular design and use of custom libraries make it easy for users to understand and modify the program. With the potential for future expansions, the simulator can continue to provide users with a comprehensive learning experience in the fields of both circuit analysis and programming.

Acknowledgement

We would like to extend our thanks to the Izmir Institute of Technology for allowing us in their laboratories and providing us with the computers to help the development process of this project.

References


