Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2023

Project Title:	**ADDIE: Analog Design & Debugging Integrated Environment**
Student:	**Archontis - Angelos Pantelopoulos**
CID:	**01760342**
Course:	**EIE4**
Project Supervisor:	**Dr Thomas J. W. Clarke**
Second Marker:	**Dr Christos Papavassiliou**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

I have used ChatGPT v3 as an aid in the preparation of my report. I have used it to improve the quality of my English throughout, however, all technical content and references come from my original text.

Acknowledgements

I would like to express my deepest gratitude to Dr. Thomas J. W. Clarke, my esteemed supervisor, for his invaluable guidance and exceptional mentorship throughout the duration of this dissertation. Dr. Clarke's expertise and insight have been instrumental in shaping the final deliverable and enhancing the quality of my work. His constructive feedback for the past 2 years has been truly inspiring, and I am profoundly grateful for his mentorship. I also want to express my sincere gratitude to my family for their everlasting support and love. Their continuous belief in my abilities has been the driving force behind my accomplishments.

Abstract

The aim of this dissertation is to develop an innovative Analog Circuit Simulator as a Desktop Application tailored specifically for first-year engineering university students. The existing tools in the field, such as LTSpice, have been found to be challenging and unnecessarily complex for novice users. To address this issue, the proposed simulator focuses on three key differentiating factors: (i) a wide range of visualizations, (ii) a simple and intuitive user interface (UI), and (iii) an exceptional graphical user interface (GUI) enabling rapid circuit creation. By incorporating these features, the ultimate goal is to provide an accessible and user-friendly platform that facilitates a smooth learning experience for students. This thesis presents the design, development, and evaluation of the simulator, demonstrating its superiority over existing tools in terms of usability and efficiency, with a particular emphasis on fulfilling the needs of first-year undergraduate students in the field of Analog electronics.

Contents

List of Figures

List of Algorithms

List of Tables

Table of Acronyms

Acronym	**Definition**
AC	Alternating Current
ADC	Analysis & Design of ciruits
ADDIE	Analog Design & Debugging Integrated Environment
API	Application Programming Interface
BEng	Bachelor of Engineering
CLR	Common Language Runtime
DC	Direct Current
DECA	Digital Electronics & Computer Architecture
DOM	Document Object Model
DU	Discriminated Union
EEE	Electrical & Electronic Engineering
EIE	Electronic & Information Engineering
GUI	Graphical User Interface
IDE	Integrated Development Environment
ISSIE	Interactive Schematic Simulator and Integrated Editor
KCL	Kirchhoff's Current Law
KVL	Kirchhoff's Voltage Law
MEng	Master of Engineering
MVU	Model - View - Update
MNA	Modified Nodal Analysis
OS	Operating System
SVG	Scalable Vector Graphics
UI	User Interface

Chapter 1

Introduction

1.1 Project Motivation

The department of Electrical & Electronic Engineering of Imperial College London faces one major issue over the past couple of years for which no solution has been found yet. Imperial's EEE department offers two undergraduate courses: BEng/Meng in Electrical & Electronic Engineering (EEE) and BEng/MEng in Electronic & Information Engineering (EIE) in which students take modules from both the Electrical Engineering and Computing departments. Both courses share a common 1^{st} year where students are taught modules that cover the whole 'Hardware-Software' space (see figure 1.1). Students can choose after their 1^{st} year which of the 2 courses they want. The problem is that each year, more and more students choose the EIE path, leaving less than 50% of students in the EEE path, which, after all, is the main degree of the department. According to Dr. Zahid Durrani, reader in Optical and Semiconductor devices and associate senior tutor, the main source of this problem is the high level of difficulty of the Analysis and Design of Circuits (ADC) 1^{st} year module, compared to the Digital Electronics and Computer Architecture module (DECA). It is believed that several students, despite being more interested in the Hardware-related modules, don't choose the EEE path as they find ADC much harder than DECA, which results in them selecting the EIE path because they performed better in DECA over their first year. This fact is also proven by the averages of these two modules each year where the average of DECA is significantly higher than the average of ADC.

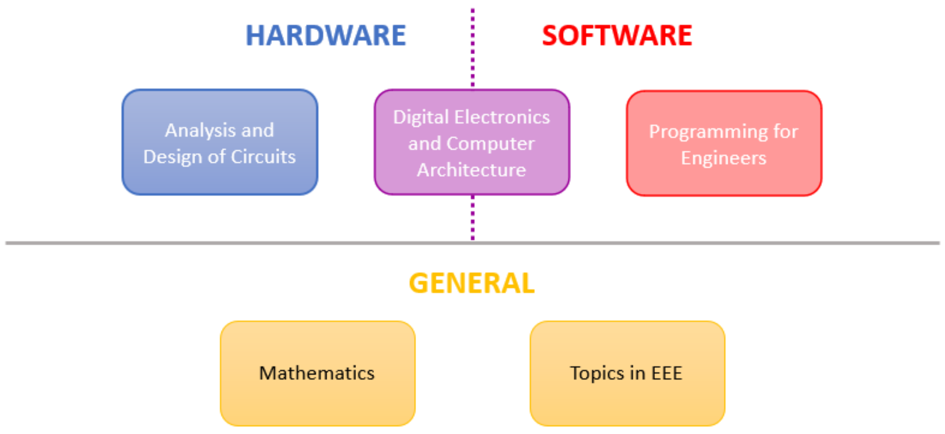

HARDWARE
SOFTWARE
Analysis and Design of Circuits
Digital Electronics and Computer Architecture
Programming for Engineers
GENERAL
Mathematics
Topics in EEE

Figure 1.1: Imperial EEE/EIE 1st-year modules

Another aspect that increases students' preference in DECA is the lab sessions that accompany the ADC and DECA modules. According to Dr. Zohaib Akhtar, Senior Teaching Fellow who demonstrates in both DECA and ADC labs, "it is not surprising students prefer DECA – you only have to go to the labs where in ADC labs they spend all their time diagnosing op-amps with disconnected supplies". It is a fact that ADC labs are harder and more complicated and this is because they are performed on actual equipment, meaning that students have to spend time on unexpected problems such as not working oscilloscopes, disconnected wires, etc.

Of course, lab work on actual circuits is a vital skill that all engineers must have, and thus, cannot be replaced by any other exercise/material. Therefore, the purpose of the final deliverable of this project, an Analog Circuit Editor and Simulator desktop application, is not to replace the work currently completed in the lab sessions, but an easy-to-use tool that focuses on the basics, to help students better understand and interpret the Analog Electronics concepts and the results they obtain in the lab sessions.

Nowadays, various Analog Circuit Simulators exist, however, as analyzed in section 2.2, these either consist of a complex Graphical User Interface (GUI) or they offer a wide range of features, not relevant for the purposes of this project's application, as they are mostly targeted to experienced Engineers, making the applications hard-to-use and unsuitable for novice students. This leads to the following project definition (section 1.2) and requirements (section 3).

1.2 Project Definition

The main aim of this project is to explore novel ways in which visualizations and simulations can help students understand basic analog concepts and combine those under a desktop application that will be capable of designing and simulating Analog circuit designs. These concepts include but are not limited to: large signal and small signal analysis, frequency response, bode plots, Nyquist diagrams, etc. The final delivered application should comprise of a very easy-to-use and uncomplicated UI which does not need a user guide ante-use, and which specifies the possible errors in the circuit to the users in a clear and straightforward manner.

All the work done on this project should align with the core principles of ISSIE [13], a digital circuit simulation tool used and maintained by Imperial's EEE department, both in terms of feature implementation and software quality. These are analyzed in subsection 1.2.1. ISSIE is an open-source application and - given the relevance between the two applications - its code may be used for the current project as well.

1.2.1 Core Principles of ISSIE

All features implemented for the current project must follow ISSIE's core principles and thus be:

1. **Robust:** Able to handle errors and behave correctly under erroneous circumstances. No matter how malformed the circuit is the application should not crash.

2. **Obvious:** The visualizations should make what is happening / what they mean obvious without the need to refer to a user manual. Specifically: (i) required features should be exposed as prominent buttons, (ii) preconditions to use a feature should be explained clearly if they are not satisfied when a button is pressed, and (iii) the use of advanced features should be guided via popups.

3. **Intuitive:** The UI should be designed in a way such that it is clear to all users how to use the application and its features.

Furthermore, the code delivered should follow ISSIE's coding guidelines as specified in Coding Guidelines for ISSIE. These focus on the readability, maintainability, and extensibility of the code, along with its proper documentation.

Chapter 2

Background

This section analyses the theoretical concepts which will help readers understand the use of the current project in the analog electronics teaching procedure, as well as provide context to the technological decisions taken during the implementation of the project. Firstly, the advantages of visualizations in learning will be discussed (subsection 2.1). This will set a framework to reflect on existing analog design simulators and consider their limitations (subsection 2.2). Then, the Technology Stack used will be analyzed and evaluated. Finally, an analysis of Analog Circuit Simulation types will be presented, along with their algorithms.

2.1 Educational Considerations

As already explained in the Introduction (section 1), the main purpose of this application is to serve as a helper tool for 1^{st} year students which will be used primarily to verify their paper analysis results in various exercises, as well as examine simulation results for various circuits as the components' values change to help them better understand the main Analog Circuit Analysis concepts. This extra source of visual information will have various advantages for students, which are analyzed below in section 2.1.1.

2.1.1 Importance of Visuals in Learning

Visuals, such as images and diagrams, play a crucial role in learning and understanding new information. They can help to make complex concepts more accessible, increase engagement and retention, and aid in the transfer of knowledge [10].

One of the main benefits of visuals is that they stick to long-term memory [10]. It has been proven that one of the easiest ways to ensure that learners store information in their long-term memory is to accompany them with meaningful visuals. This is because the brain processes visual information differently and is able to create a mental image of the information, making it easier to recall later. Specifically, it was found that after three days, a user retained only 10-20% of written or spoken information, but more than 60% of visual information [2].

Another important aspect of visuals in learning is they help in the transmission of knowledge. When students are able to see how a concept is applied in a real-world scenario, it helps to solidify their understanding and makes it easier for them to apply what they have learned in the future.

Overall, the goal of the application is to provide students with a source of visual information which is directly linked to their module's curriculum. This is expected to be very beneficial for students, helping them in the learning of Analog Electronics laws and concepts.

2.2 Existing Analog Circuit Simulation Tools

Analog Circuit Simulation applications exist for more than 30 years and are used for various purposes around the world. This section will reflect on the currently most-used tools, discussing their advantages and disadvantages.

There are a number of analog circuit design tools available, each with its own set of capabilities. Some well-known examples are listed in Table 7.4. Tools such as LTSpice [32], NGSpice [36], and Qucs [46] are designed for use in industrial settings, and are geared towards experienced engineers. They offer a wide range of features that may not be necessary or understandable for novice students. Having a large number of features can make it difficult for users to find what they need and can make the learning process more complicated. Given the very tight time schedule of students, it is important to make it as easy as possible for them to learn the concepts, rather than focussing on how to use complicated tools. On the contrary, CircuitLab [42] and iCircuit [26] have a simpler user interface and are primarily used as teaching tools, so they prioritize simplicity over the number of features offered. A short description of each tool is available in Table 2.2.

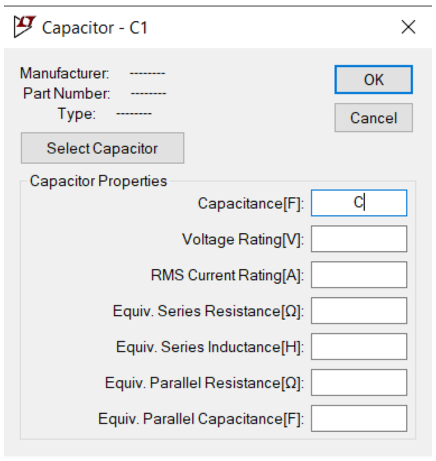
	LTspice[32]	NgSpice[36]	CircuitLab[42]	Qucs[46]	iCircuit[26]
Diagram Editor	✓	✗	✓	✓	✓
Netlist editor	✓	✓	✗	✗	✗
Simulation	✓	✓	✓	✓	✓
Teaching Oriented	✗	✗	✓	✗	✓
Open-source	✓	✓	✗	✓	✗
Actively maintained	✓	✓	✓	✗	✗
Supported platforms	Windows, MacOS	Windows	All (Web-app)	All	All+Mobile

Table 2.1: Comparison of existing Analog Circuit Design tools

Analog Simulator	**Description**
LTspice [32]	LTspice is a widely used and distributed SPICE-based analog simulator. It is one of the most powerful simulation tools nowadays, however, it has a strange and hard-to-follow user interface which makes the design of circuits quite slow. The simulation options it offers cover all possible needs, however, again, setting-up a simulation and configuring the components of a circuit requires multiple different steps which make it hard for a beginner to familiarise with. For example, figure 2.1 shows the menu for configuring a single capacitor.
NGSpice [36]	NgSpice is a similar simulation tool to LTspice, but it has the advantage of having strong community support because it was developed by merging three open-source packages: XSpice, Cider, and Spice3f5 [55]. However, a major disadvantage of Ngspice is that circuits can only be created through the command line, making it challenging to visualize the circuit.
CircuitLab [42]	CircuitLab is the application with the most user-friendly interface. All components are categorized in a LHS pane with their icon, so that they can be quickly identified and added to a circuit with Drag and Drop. Also, the components can be connected easily using the ports that exist on the components. It offers all simulation modes, however, all the nodes for which the users want to see results must be given a distinct name by the user. This is a very time-consuming process which complicates the configuration of a simulation. Finally, it is not an open-source simulator.

Analog Simulator	Description
Qucs [46]	Qucs is an open-source simulator that can be used for both Analog and Digital Circuits. Again, as for LTSpice, it does not have a very user-friendly interface, but, it includes a Component Library on the LHS of the screen which includes all the components as icons, making it much easier to add components to a schematic (compared to LTSpice). However, Qucs is not maintained, with its latest release being in 2017.
iCircuit [26]	iCircuit has one major advantage over all the other applications: it features a real-time, always-on analysis that is supported by a very powerful simulation engine. In other words, this allows users to view instantly the changes they make in the circuit. It also offers live visual feedback on the wires so users can see the direction and speed of electrons in a circuit. However, I find this feature confusing in the way it is implemented. Moreover, as CircuitLab, it is not open-source.

Table 2.2: Analysis of existing Analog Circuit Design tools


Capacitor - C1

Manufacturer: --------
Part Number: --------
Type: --------

Capacitor Properties

Capacitance[F]:	C
Voltage Rating[V]:	
RMS Current Rating[A]:	
Equiv. Series Resistance[Ω]:	
Equiv. Series Inductance[H]:	
Equiv. Parallel Resistance[Ω]:	
Equiv. Parallel Capacitance[F]:	

Figure 2.1: Configuration of a capacitor in LTSpice

Each simulation tool has its advantages compared to the other tools. However, there is one major feature that none of the simulators support: a detailed and thorough error analyzer that can explain all the errors in the circuit via a simple and informative error message, along with visual feedback to identify the location of the error. Furthermore, the (currently) open-source tools do not offer a friendly User Interface which makes them non-ideal for young Engineering students. Lastly, none of the tools analyzed gives on-circuit simulation results which would make the results easier to read and understand. In summary, it is evident that all existing simulators fail to meet the requirements for ease of use as discussed in section 1.2.

2.3 Technological Considerations

The creation of a sizable software application implies that many technological choices need to be made in advance. As already explained in section 1, the application must be cross-platform and easy to maintain. The EEE department of Imperial College London, uses and maintains an open-source application used for digital electronics simulations, which goes under the name **ISSIE**: **I**nteractive **S**chematic **S**imulator and **I**ntegrated **E**ditor [13]. Given its purpose, ISSIE, as an application, shares many characteristics with the current project. These include but are not limited to:

- Main canvas to create circuits
- Movable components using SVG
- Wiring functions: Auto-routing, Manual Routing
- Main UI functionalities: snapping of components, copy/paste, undo, etc.

Furthermore, ISSIE is written in F# , a language with major advantages given the requirements of the current project. Firstly, Imperial College EE students learn F# in the High Level Programming [16] module, allowing them to maintain the app in the future. In addition, F# can be combined with the Elmish [17] framework to create User Interfaces using Functional Reactive Programming techniques. Finally, strongly typed functional code (what F# provides) is typically simple to maintain and test because the type-checker provides significant assistance [53].

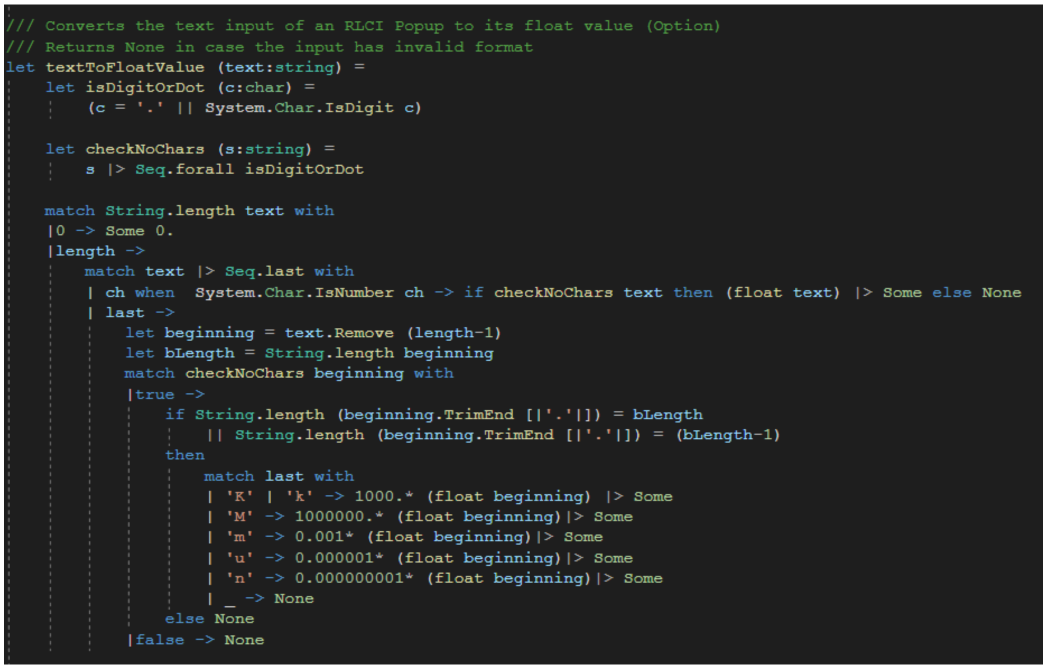
These considerations motivate the use of ISSIE as the basis of my application. In the following sections, I will provide an overview of ISSIE's technology stack, evaluate it, and confirm its suitability for this project.

2.3.1 Programming Language - F#

ISSIE is mostly written in F# : an open-source, cross-platform, interoperable programming language for writing succinct, robust, and performant code [9]. F# is a hybrid language: it is principally a functional programming language, including many features found in other functional languages, but at the same time allows users to use mutable data and classes (Object-Oriented-Programming features). Functional programming languages follow a declarative programming philosophy and use immutable data. In this approach, data values can't be changed after they are first assigned, and functions use this data as input and produce - new - output data. F# functions are deterministic, in the sense that the output of the function is exclusively dependent on the input, and that there is no internal state affecting the function's behavior. This deterministic nature of functions makes them simple to comprehend as operations on data don't have any external effects and are simple to follow. This, not just simplifies the debugging process, but also results in fewer bugs in the code. Furthermore, F# has a very powerful type inference algorithm that allows it to be statically typed, but at the same time, without requiring the specification of the type of an object. This, in addition to the stylistic advantages (clean code), it also allows for the early detection of errors at compile time. Types can also be inferred by IDEs such as Visual Studio, making it straightforward for developers to identify the errors and track the correctness of their code. [29]

F# also offers some F# -specific types which allow users to minimize the appearance of exceptions and NULLs at runtime, which are considered to be the most common errors in coding, after syntax errors. However, runtime errors cannot be caught by any IDE or compiler. F# tries to reduce such errors by introducing: (i) `Option`, (ii) `Result`, and (iii) `Discriminated Union` (DU). [43]

The `Option` type can take either the value `Some <type T>` or the value `None`, allowing a safe way of representing 'nothing' that is clear before compile-time. `Result` follows a similar logic, taking either the value `Ok <type A>`, or the value `Error <type B>`, giving to the developer a safe way of representing success - failure, without runtime exceptions. The example in figure 2.2, which shows the `textToFloatValue` function used in the current project, demonstrates the advantages of `Option`. Instead of directly using the `string -> float` parsing function which would lead to exceptions at runtime with inputs such as: ”100.0.0”, ”100j”, ”123.k”, this function returns an `Option<float>` and then in order to use this result we have to extract the value from the `Option`. This, as already explained, prevents runtime errors.



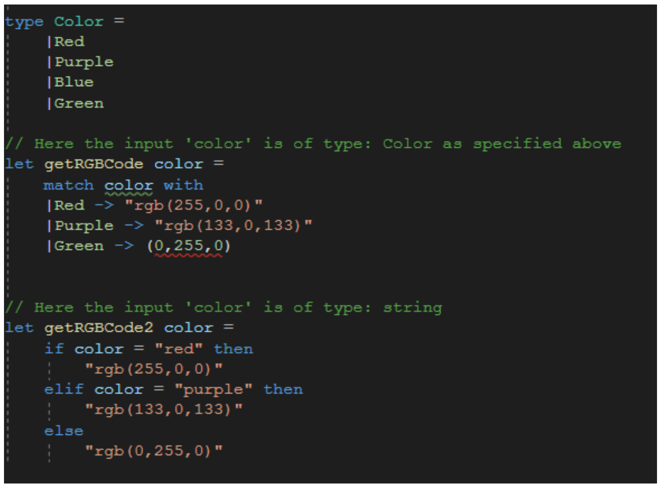
```
/// Converts the text input of an RLCI Popup to its float value (Option)
/// Returns None in case the input has invalid format
let textToFloatValue (text:string) =
    let isDigitOrDot (c:char) =
        (c = '.' || System.Char.IsDigit c)

    let checkNoChars (s:string) =
        s |> Seq.forall isDigitOrDot

    match String.length text with
    |0 -> Some 0.
    |length ->
        match text |> Seq.last with
        | ch when  System.Char.IsNumber ch -> if checkNoChars text then (float text) |> Some else None
        | last ->
            let beginning = text.Remove (length-1)
            let bLength = String.length beginning
            match checkNoChars beginning with
            |true ->
                if String.length (beginning.TrimEnd [|'.'|]) = bLength
                    || String.length (beginning.TrimEnd [|'.'|]) = (bLength-1)
                then
                    match last with
                    | 'K' | 'k' -> 1000.* (float beginning) |> Some
                    | 'M' -> 1000000.* (float beginning)|> Some
                    | 'm' -> 0.001* (float beginning)|> Some
                    | 'u' -> 0.000001* (float beginning)|> Some
                    | 'n' -> 0.000000001* (float beginning)|> Some
                    | _ -> None
                else None
            |false -> None
```

Figure 2.2: Use of `Option` in the code

`Discriminated Union` (DU) allows the creation of custom types, which are useful when a variable can only take specific values under some context. The use of DUs in the code allows the IDE and the compiler to detect unmatched cases and block type errors which would lead to exception errors at runtime. The example in figure 2.3 demonstrates the merits of DUs. A DU named `Color` has been created with 4 possible values: Red, Purple, Blue, Green. The goal is to create a function which translates the color to its RGB decimal code.



```
type Color =
    |Red
    |Purple
    |Blue
    |Green

// Here the input 'color' is of type: Color as specified above
let getRGBCode color =
    match color with
    |Red -> "rgb(255,0,0)"
    |Purple -> "rgb(133,0,133)"
    |Green -> (0,255,0)


// Here the input 'color' is of type: string
let getRGBCode2 color =
    if color = "red" then
        "rgb(255,0,0)"
    elif color = "purple" then
        "rgb(133,0,133)"
    else
        "rgb(0,255,0)"
```

Figure 2.3: Use of `DU` in the code

The first function, `getRGBCode`, expects one input: color of type Color. Then, using the match statement we are able to split the input into 3 branches (3 different colors) and return different results. The IDE gives 1 warning and 1 error in this case. The green line (warning), on hover, informs the programmer that not all cases of the DU have been covered by the match statement, stating: "Incomplete pattern matches on this expression. For example, the value 'Blue' may indicate a case not covered by the pattern(s)". The red line, on hover, informs the user that this function must return a string, as the previous two branches returned a string. The error specifically mentions: "All branches of a pattern must return the same type which is string. This branch returns `int * int * int`".

The second function, `getRGBCode2`, expects the same input color, but this time it is a string. Therefore, the function can be called by '`getRGBCode2 "red"`' for example. First of all, we notice that this function gives no warning about the uncovered cases as it doesn't know what values to expect (it just expects a string). This means that apart from the value "blue" which we have not taken into account, the function will produce wrong results even when used with wrong capitalization, e.g. "Red" instead of "red".

While F# is a functional-first programming language, it also allows programmers to use other programming styles. For example, this includes the use of mutable data, classes, operator overloading, etc. which are all Object-Oriented-Programming techniques.

To conclude, the features discussed in this section are primarily found only in F# or are implemented in a better way in F# compared to other popular programming languages for desktop application development like JavaScript and Python. Therefore, given also the extremely low bug rate of F# , it can be concluded that the use of F# is appropriate and optimal for the current project.

2.3.2 Software Ecosystem

F# is a programming language created to run on the .NET framework and the Microsoft Common Language Runtime (CLR) [31]. However, it can also be used in the JavaScript ecosystem using third-party transpilation tools [8]. This means that F# can be used to create desktop applications using .NET, as well as web apps using JavaScript. ISSIE, was built using the second method, where F# code is converted to JavaScript and then compiled as a desktop application using `Electron`. This section analyses the tools that were selected for this process and the reasons why they were chosen.

Fable

As stated above, the code gets compiled into a desktop application using Electron. The problem is that Electron only works with JavaScript code, which, however, has various drawbacks compared to F# . Fable, a compiler that lets you use F# to build applications that run in the JavaScript ecosystem [19], acts as the bridge between F# and Electron. Specifically, Fable converts the F# code into clean JavaScript code, which then is used as an input to Electron. Fable also includes binding packages [18] for various (the most famous) JavaScript packages such as React [20]. In other words, programmers can use JavaScript-only packages, by using the binding packages which have incorporated the necessary functions into F# functions. Also, in the case a binding package isn't provided by Fable, users can write their own bindings for the JS package they want to use by specifying the number and type of the parameters expected by each function, along with the type it returns.

External Libraries

As explained above, users can use third-party JavaScript packages either by using the pre-existing Fable binding packages or by writing their own JavaScript - F# binding packages. This yields a question on whether JavaScript or .NET libraries should be used in the code. Both environments provide plenty of libraries that cover various possible needs. For the current project, possible needs include (for the time being) a library to perform matrix construction/inversion for DC Operating Point Analysis as explained in section 2.4.1 and a library to plot graphs for AC and Transient Analysis.

Given that the final application runs on the JS ecosystem via Electron, .NET libraries not supported by Fable should be avoided. This is because such libraries can only be used if they run outside the main application, as a separate executable. In such cases, data can be transferred using an intermediate file for example. However, Fable has a complete guide on how to Call JS from Fable. For example, to use the `highlight` function of the PrismJS package (example taken from ISSIE), which performs real-time code highlighting, the necessary binding code is shown in listing 2.1.

```
type PrismCore =
    abstract highlight : string * obj -> string

[<ImportAll("../VerilogComponent/prism.js")>]
let Prism: PrismCore = jsNative
```

Listing 2.1: Calling JavaScript packages from Fable

Electron

Electron is an open-source framework that allows developers to build cross-platform applications using HTML, CSS, and JavaScript [6]. Electron bundles NodeJS [37] and the Chromium Web Browser [12] into a single package, thus allowing programmers to run JavaScript outside of a browser (NodeJS framework), and then display content designed with HTML and CSS (Chromium Rendering Engine). This means that Electron applications can also access OS-level APIs such that the file system ones.

However, electron also comes with various drawbacks. First of all, as Electron bundles NodeJS and Chromium, a single application will often be larger than 80 MBs in size, despite not having a large source code file. Moreover, Chromium is a very RAM-intensive browser which means that on not very powerful machines (< 4 GB RAM) the app will be less responsive compared to native desktop applications. Lastly, Electron-based applications are essentially web applications, so they contain common web vulnerabilities such as cross-site scripting (XSS), SQL injection, and authentication and authorization flaws [54]. Nevertheless, as is the case with ISSIE, ADDIE will only be run locally, and will not have access to the internet, therefore, such vulnerabilities can be considered as not-applicable.

Despite its drawbacks, Electron has worked really well with ISSIE over the past 3 years, without performance issues. Therefore, it can be concluded that its major advantage: the creation of cross-platform applications using a single source code, is more significant than the limitations described above, and is, therefore, a good fit for ISSIE, and consequently, ADDIE.

2.3.3 User Interface

An effective and organized GUI is crucial for any software, but especially for applications like ISSIE and ADDIE whose focus is on user interactivity and ease of use. In such applications, user interaction occurs mostly through mouse actions (clicking and dragging). Thus, the source code must be able to efficiently handle a continuous flow of mouse events and respond immediately. This section analyses ISSIE's user interface framework and the reasons behind this selection.

Elmish: Model - View - Update (MVU)

The Elm Architecture, or MVU Architecture, is a design pattern used to build interactive programs. It originates from the Elm programming language [49]. Elm is a functional language primarily used for creating web applications and websites [28]. Elm is similar to F# in that it also compiles to JavaScript, and can be used for web applications. ISSIE uses the Elmish library [17], which allows users to use the Model-View-Update architecture for F# apps. Elmish also works well under the Fable compiler, which results in the current smooth and stable UI of ISSIE.

The Model-View-Update (MVU) architecture breaks the code into three parts:

1. **Model:** A data structure that holds the current state of the application. In an application like ISSIE and ADDIE the model is a very big record as it contains: (i) all the components along with their values and the connections between them, (ii) selected/copied/deleted/nearby components/wires, (iii) simulation results and much more

2. **View:** A function that transforms the current state of the application (stored in the Model) into HTML and CSS which are then displayed to the user.

3. **Update:** A method for changing the application's state as a result of events like user input. These events are conveyed within the program using ***messages***, which are triggered by user interactions with the GUI. The update function is a function that takes in two inputs: the message and the current model; It interprets the message and returns the updated model.

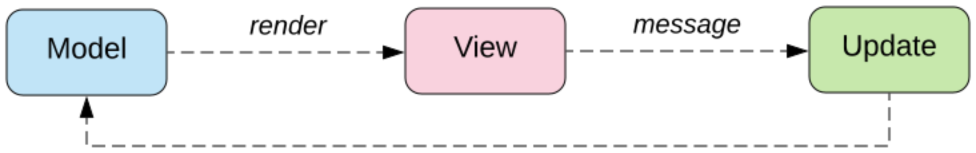


Figure 2.4: Model-View-Update Diagram [4]

Apart from the above 3 parts, the user must also define an ***init*** function which returns the initial Model of the application: the one rendered (View) at start-up. As the user interacts with the application, each interaction generates a message which is sent to the update function. The update function processes the message and returns an updated version of the Model if necessary, reflecting the outcome of the interaction. The View function will then re-render the updated state of the application, displaying the result of the user's action to them.

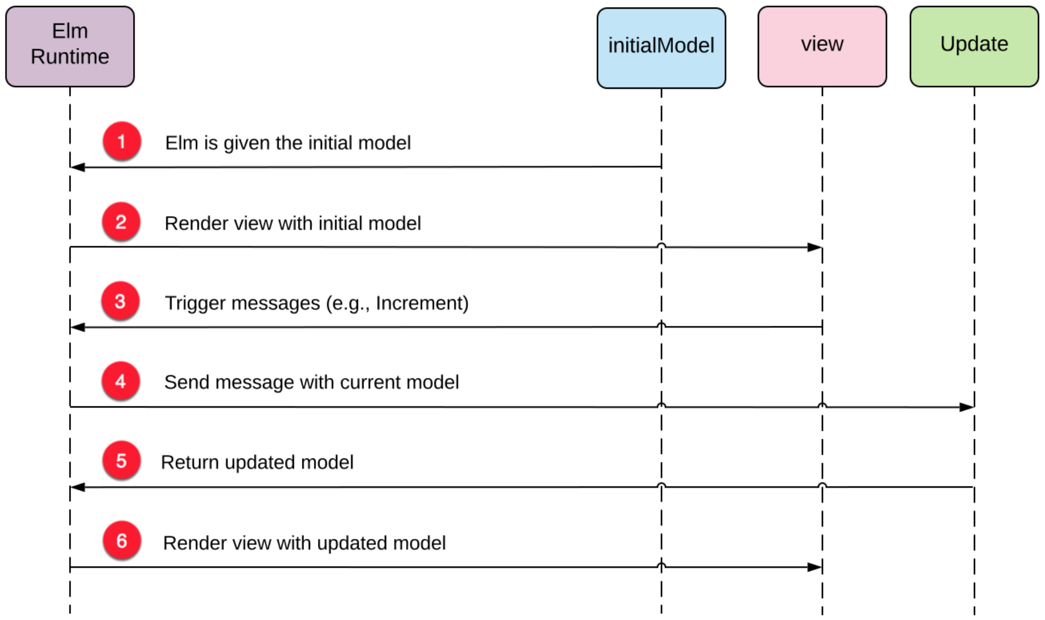

Elm Runtime
initialModel
view
Update
1
Elm is given the initial model
2
Render view with initial model
3
Trigger messages (e.g., Increment)
4
Send message with current model
5
Return updated model
6
Render view with updated model

Figure 2.5: Elm runtime interaction [4]

The MVU Architecture is well-suited for use in such applications due to its structure, which is based on an immutable model and deterministic view and update functions. This structure allows for data to flow in a single direction and for events that trigger changes in the application's state to be clearly identified. This makes the rendering process easier for programmers to understand, allowing them to spend less time debugging. The MVU architecture is especially well-suited for functional languages such as F# , which has features that integrate well with the MVU architecture, further strengthening it. Overall, the MVU architecture is an ideal fit for ISSIE, providing developers with an improved experience and a performant user interface.

At first glance, it may seem that the MVU's approach of using a view function that continuously re-renders the Model whenever there is an update would be inefficient. This would be true if traditional methods for rendering HTML were used. The traditional process of rendering HTML involves parsing the HTML and constructing a DOM (Document Object Model), which is a tree representation of the HTML. When the state of the web application changes, the generated HTML changes as well, which means that the DOM tree needs to be updated accordingly. However, this can be a slow process for large web apps with a tall DOM tree, causing the UI to appear slow-moving. To address this issue, Elm uses an intermediate Virtual DOM, which is a lightweight and optimized version of the main DOM tree on which operations are computationally inexpensive. As shown in Figure 2.6, Elm exposes the virtual DOM to the programmer, who determines in the View function how the state should be rendered.

Each time the state changes, a new virtual DOM is constructed and is compared against the previous virtual DOM, which enables Elm to identify the exact changes to the UI. Both operations are fast due to the lightweight nature of the virtual DOM. This process is repeated for multiple changes in the UI, creating a list of updates that are performed at once on the Real DOM tree. This approach decreases the cost of updating the DOM, resulting in an efficient re-rendering of the application's state (Model), which also improves the overall performance of the application.

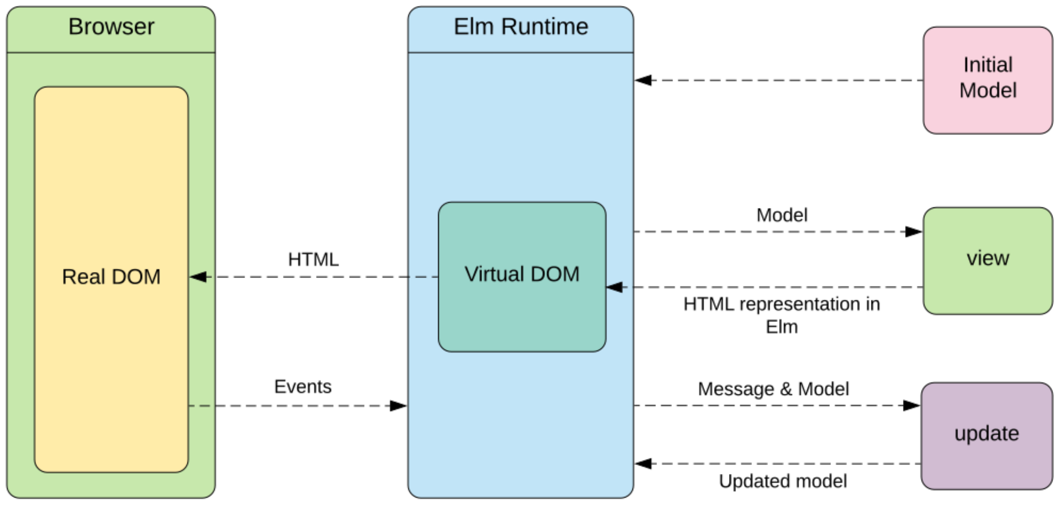


Figure 2.6: Elm runtime virtual DOM [4]

React

React is a JavaScript library used for creating user interfaces [1]. It follows a component-based architecture, where a UI is divided into small, independent, and reusable pieces called components. These components can be easily composed to create a more complex UI. React also uses a virtual DOM which optimizes the performance of updates by only rendering the parts of the UI that have changed. Therefore, the virtual DOM mentioned above actually refers to the React Virtual DOM, which is made up of React Elements - simple objects that are inexpensive to create [48]. The View function returns a React Element that displays the current state of the UI. React improves performance by only updating elements when necessary; a process known as Reconciliation [47].

Fulma

Fulma [23] is a set of CSS classes for the Bulma framework, which is a lightweight, responsive CSS framework based on Flexbox which provides ready-to-use front-end components for building web interfaces [7]. Fulma allows users to use these components in F# with fable-react [20]. Therefore, React components can now be created using the F# functions provided by Fulma. An example is displayed in listing 2.2.

```
Navbar.Item.div []
  [ Navbar.Item.div []
        [ Button.button
            [ Button.OnClick(fun _ -> PopupView.viewInfoPopup dispatch)
              Button.Color IsInfo
            ]
            [ str "Info" ]
        ]
  ]
```

Listing 2.2: Creation of a button in a navbar in F# using Fulma

2.3.4 Summary

Figure 2.7 provides an overview of the Technology Stack which will be used for this project.

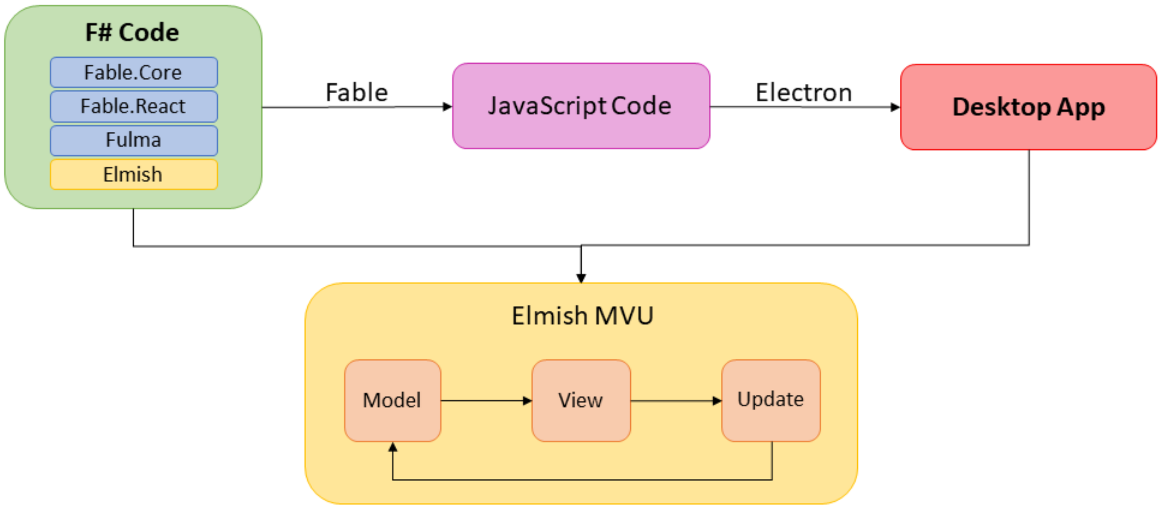


Figure 2.7: Technology Stack Summary

2.4 Types of Analog Circuit Simulation

Simulation of analog circuits involves using algorithms to predict and verify the performance of circuit models that can include both linear and nonlinear components (resistors, diodes, transistors, etc.). There are various types of simulations, including DC Operating Point analysis, AC analysis, and Transient Analysis, each providing different information about the circuit [9].

2.4.1 DC Operating Point Analysis

DC Operating Point Analysis calculates the behavior of a circuit when a DC voltage or current is applied to it; known as bias point. In many cases, DC analysis is used as an intermediate step for further analysis, such as AC Analysis, where the DC operating point is necessary to obtain linear, small-signal models for all the non-linear components of the circuit [40]. When running DC Operating Point Analysis all AC sources and inductors are treated as 0-resistance wires, and capacitors as open (∞ resistance).

Nodal Analysis

The most commonly used method for DC Operating Point Analysis is Nodal Analysis, which, utilizing Kirchhoff's Current Law (see Definition 1: KCL), determines the voltage at each node in an electrical circuit. KCL results in equation 2.1 in matrix form where the conductance matrix is multiplied with the voltage vector to give the current vector.

Definition 1. *Kirchhoff's Current Law (KCL): The algebraic sum of all currents entering and exiting a node must equal zero [30].*

$$\begin{bmatrix} G_{11} & -G_{12} & \cdots & -G_{1n} \\ -G_{21} & G_{22} & \cdots & -G_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -G_{n1} & -G_{n2} & \cdots & G_{nn} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix} = \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix} \tag{2.1}$$

Conductance Matrix

The conductance matrix, depicted in equation 2.1, is obtained as follows: First, we find the size of the conductance matrix by counting the number of nodes present in the circuit. The matrix will always be square and its size is $N \times N$ where N is the number of nodes. Each element of the matrix determines the conductance between the i-th and the j-th node. For example, the value at position G_{13} refers to the conductance between nodes 1 and 3. Clearly, $G_{ij} = G_{ji}$. The diagonal elements G_{ii} are calculated by taking the sum of conductances of passive components at node i. The pairs of nodes that are not directly connected via a single component get the value 0 [3].

The following flowchart analyses the Nodal Analysis steps.

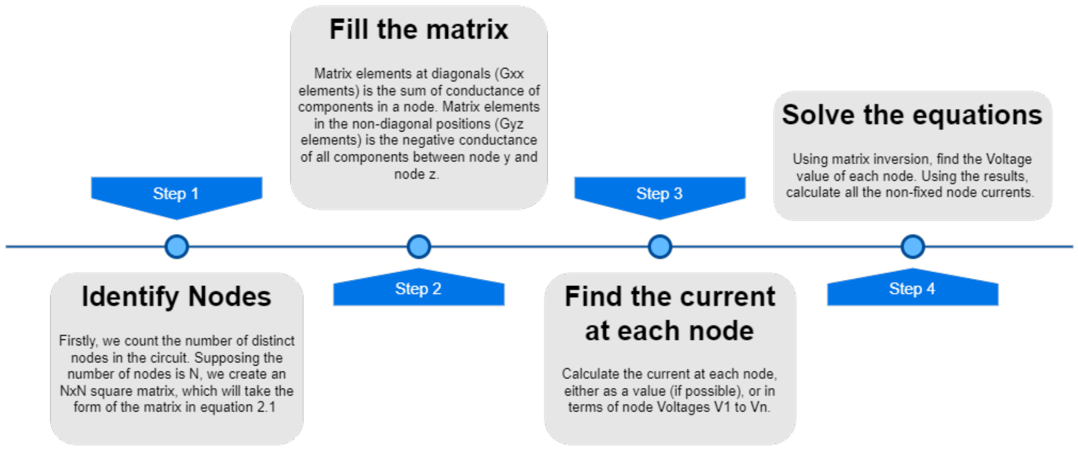

Fill the matrix
Matrix elements at diagonals (Gxx elements) is the sum of conductance of components in a node. Matrix elements in the non-diagonal positions (Gyz elements) is the negative conductance of all components between node y and node z.
Solve the equations
Using matrix inversion, find the Voltage value of each node. Using the results, calculate all the non-fixed node currents.
Step 1
Step 3
Step 2
Step 4
Identify Nodes
Firstly, we count the number of distinct nodes in the circuit. Supposing the number of nodes is N, we create an NxN square matrix, which will take the form of the matrix in equation 2.1
Find the current at each node
Calculate the current at each node, either as a value (if possible), or in terms of node Voltages V1 to Vn.

Figure 2.8: Nodal Analysis Algorithm

Matrix Libraries

Given that Fable does not support matrix operations, it is necessary to find a JavaScript Martix Library which will then be interfaced with F# . Three JavaScript libraries that fit the purposes of the current project are listed below:

1. ml-matrix Library [34]
2. math.js Library [33]
3. numbers.js Library [41]

All three fit the needs of this project, however, MathJS [33] offers a wider range of features, and also has more detailed documentation and is more actively maintained compared to the other two.

Nodal Analysis example

A simple circuit consisting of a single current source (I1) and two resistors in series (R1,R2) is depicted in figure 2.9. The source supplies 1A current, and the two resistors have values 5Ω and 10Ω respectively.

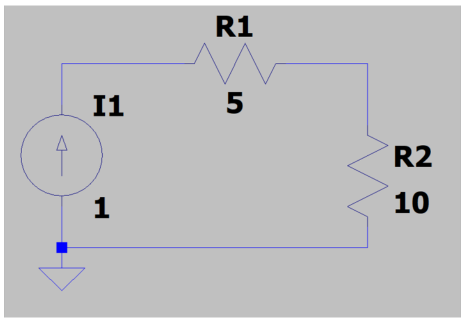


Figure 2.9: Linear Circuit Example

Firstly, we find the conductance matrix following the steps defined above. The circuit has only 2 nodes, therefore, the conductance matrix will be a 2×2 matrix of the following form:

$$G = \begin{bmatrix} G_{11} & -G_{12} \\ -G_{21} & G_{22} \end{bmatrix} \tag{2.2}$$

We obtain the diagonal indices of the matrix (G_{11} and G_{22}) by looking at nodes 1 and 2 individually. At each node, we take the sum of the conductance of the components connected at the node. For node 1, R1 is the only component connected to this node, resulting in $G_{11} = \frac{1}{R1}$. For node 2, both R1 and R2 are connected to it, giving $G_{22} = \frac{1}{R1} + \frac{1}{R2}$.

$$G = \begin{bmatrix} \frac{1}{R1} & -G_{12} \\ -G_{21} & \frac{1}{R1} + \frac{1}{R2} \end{bmatrix} \tag{2.3}$$

For the non-diagonal elements G_{12} and G_{21}, we need to sum the negative conductance of each component between nodes 1 and 2. In the current example, only R1 is between nodes 1 and 2, whose negative conductance is $-\frac{1}{R1}$, giving:

$$G = \begin{bmatrix} \frac{1}{R1} & -\frac{1}{R1} \\ -\frac{1}{R1} & \frac{1}{R1} + \frac{1}{R2} \end{bmatrix} = \begin{bmatrix} \frac{1}{5} & -\frac{1}{5} \\ -\frac{1}{5} & \frac{3}{10} \end{bmatrix} \tag{2.4}$$

To solve equation 2.1, we also need to find the current at nodes 1 and 2. In this case, it is trivial to obtain the values, as clearly, $I_1 = 1A$ and $I_2 = 0$, resulting in the final form of equation 2.1:

$$\begin{bmatrix} \frac{1}{5} & -\frac{1}{5} \\ -\frac{1}{5} & \frac{3}{10} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{2.5}$$

To calculate V_1 and V_2 we apply matrix inversion to find:

$$\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} 15 & 10 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 15 \\ 10 \end{bmatrix} \tag{2.6}$$

2.4.2 AC Analysis

In AC Circuit Analysis, we examine how a circuit behaves with respect to the frequency of its input signal. First, we perform DC Operating Point analysis to obtain the small-signal model of the circuit. Then, we analyse the circuit through a range of frequencies (typically specified by the user) to see the resulting voltages per node, per frequency. In other words, all DC sources are set to 0, and all AC sources are represented by the selected frequency at each step [15]. The results are displayed in bode plots showing both Magnitude and Phase against frequency (usually the graph is about the ratio Vout/Vin where Vout is our selected output node, and Vin the node in the positive pole of the AC source).

The procedure to perform AC Analysis does not differ much compared to the nodal Analysis algorithm analyzed in section 2.4.1. However, this time, the conductance matrix is calculated using the complex impedance of the components, as derived in table 2.3.

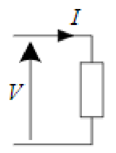
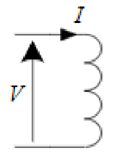
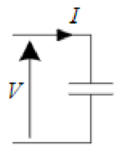
Component	Complex Impedance	Circuit
Resistor	$v(t) = Ri(t) \implies V = IR$ $\implies \frac{\mathbf{V}}{\mathbf{I}} = \mathbf{R}$	
Inductor	$v(t) = L\frac{di}{dt} \implies V = j\omega LI$ $\implies \frac{\mathbf{V}}{\mathbf{I}} = \mathbf{j\omega L}$	
Capacitor	$i(t) = C\frac{dv}{dt} \implies I = j\omega CV$ $\implies \frac{\mathbf{V}}{\mathbf{I}} = \frac{\mathbf{1}}{\mathbf{j\omega C}}$	

Table 2.3: Complex Impedance of Components

Graph Plotting Libraries

Two JavaScript graph plotting libraries that fit the needs of the current application are:

1. Plotly nodeJS Library [45]

2. Chart.js Library [11]

Plotly is the most commonly used graphing library nowadays, offering an extensive set of features. Also, PlotlyJS is already supported by Fable, making it the optimal decision.

2.4.3 Transient Analysis

Transient Analysis examines a circuit's response with respect to time, calculating the voltage and current per node at each segment of time. The user specifies the nodes and/or components for which he wants to see how the voltage/current changes, along with a period of time, and the resulting graph will display the change of voltage/current values over that period of time.

Transient Analysis can be separated into three different types of simulations. The first is circuits without reactive components, which can be simulated using the DC Operating Point Analysis algorithm for multiple time samples. The second is first-order circuits which can be solved using the algorithm used in the ADC module, where the transient $Ae^{\frac{-t}{\tau}}$ is used as the solution of the ODE. The third is circuits with no restrictions in terms of components used. Such circuits are solved using numerical integration methods, such as the trapezoidal rule or Gear's method, to solve the system of differential equations comprising a circuit [27]. Further analysis of the algorithms used in the current application can be found in section 5.8 of the Implementation chapter.

2.5 Simulation Algorithms

2.5.1 Modified Nodal Analysis (MNA)

Modified Nodal Analysis (MNA) is a widely used technique in the field of circuit analysis which provides a systematic and efficient approach for solving electrical circuits by formulating the circuit equations using nodal analysis and incorporating additional modifications to handle the presence of voltage sources and other components. The vector of unknowns becomes the concatenation of the set of node voltages and the set of independent voltage sources' currents.

The main steps of the algorithm for a circuit with n nodes and m voltage sources are [35]:

1. Circuit Formulation: Identify the nodes of the circuit, select a reference node (ground), and name the remaining (n-1) nodes. Also, label the independent current sources.

2. Nodal Analysis: Apply the nodal analysis algorithm to derive the circuit equations in terms of nodal voltages.

3. Modification for Independent Voltage Sources: By treating each independent voltage source in the circuit as a current source connected to a supernode, an additional equation is created for each voltage source

4. Augmented Matrix Representation: Combine all the equations to form a matrix equation of the form $\mathbf{Ax} = \mathbf{b}$.

5. Obtain the Solution: Solve the equations using matrix inversion. Use the results for the calculation of branch currents and/or other information of interest.

Algorithmic MNA

This section describes a stable algorithm that can be used in any circuit with passive elements and independent current and voltage sources. As seen above, this results in a matrix equation of the form $\mathbf{Ax} = \mathbf{b}$ where for a circuit with n nodes and m voltage sources:

- The $\mathbf{A}$ matrix is an $(n + m) \times (n + m)$ matrix of known quantities, where the $n \times n$ part of the matrix is the normal Conductance Matrix of the nodal analysis algorithm and the remaining consists of 0,-1,1 only.

- The **x** vector is an $(n+m) \times 1$ vector that holds the unknown quantities (node voltages and the currents through the independent voltage sources).
- The **b** vector is an $(n+m) \times 1$ vector that holds the known quantities: [top n elements] currents of nodes (from independent current sources), [bottom m elements] voltages of independent voltage sources.

The **A** matrix is separated into 4 smaller matrices, **G** $(n \times n)$, **B** $(n \times m)$, **C** $(m \times n)$, and **D** $(m \times m)$, as follows:

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix} \tag{2.7}$$

1. The **G** matrix is the conductance matrix and is formed in two steps (as already explained in section 2.4.1):
   (a) Each element in the diagonal of the matrix (G_{ii}) is equal to the sum of the conductance ($\frac{1}{R}$) of each element connected to the i^{th} node
   (b) The non-diagonal elements are equal to the negative conductance of the component connected between the i^{th} and j^{th} node.
2. The B matrix consists only of 0, 1, and -1. Each matrix element corresponds to a particular voltage source (i^{th} row) and a node (j^{th} column). If the positive terminal of the i^{th} voltage source is connected to node j, then the element B_{ij} in the B matrix is a 1. If the negative terminal of the i^{th} voltage source is connected to node j, then the element B_{ij} in the B matrix is a -1. All other elements of the B matrix are 0.
3. The **C** matrix is the transpose of the **B** matrix when no dependant sources or opamps are present ($\mathbf{C} = \mathbf{B}^T$).
4. The **D** matrix is always an $m \times m$ **0** matrix when no dependent sources are present in the circuit.

MNA Example

The following example shows how MNA is able to provide a solution for any type of circuit, regardless of the number of Voltage/Current sources and Resistors.

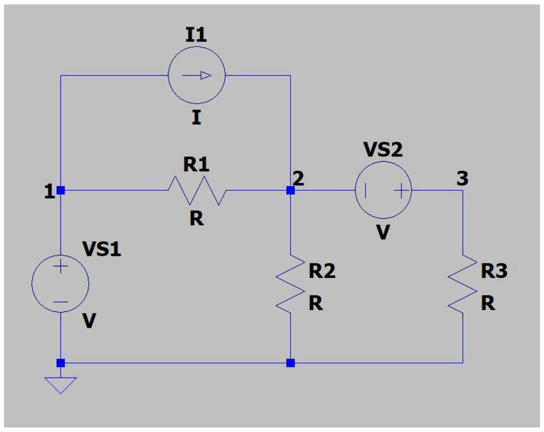


Figure 2.10: Linear Circuit Example with Voltage Sources

We get the following equations:

$$\begin{aligned}
I_{VS1} + I_1 + \frac{V_1 - V_2}{R_1} &= 0 && \text{node 1} \\
-I_1 + \frac{V_2 - V_1}{R_1} + \frac{V_2}{R_2} - I_{VS2} &= 0 && \text{node 2} \\
I_{VS2} + \frac{V_3}{R_3} &= 0 && \text{node 2} \\
V_1 &= VS1 && \text{Voltage Source 1} \\
V_3 - V_2 &= VS2 && \text{Voltage Source 2}
\end{aligned}$$

which using the MNA algorithm can be written in matrix form as:

$$\begin{bmatrix} \frac{1}{R_1} & -\frac{1}{R_1} & 0 & 1 & 0 \\ -\frac{1}{R_1} & \frac{1}{R_1} + \frac{1}{R_2} & 0 & 0 & -1 \\ 0 & 0 & \frac{1}{R_3} & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I_{VS1} \\ I_{VS2} \end{bmatrix} = \begin{bmatrix} -I_1 \\ I_1 \\ 0 \\ VS1 \\ VS2 \end{bmatrix}$$

The result is then obtained by inverting the matrix and multiplying it with the vector **b**.

Extension for Ideal Opamps

The MNA algorithm can be extended to work for Ideal Operational Amplifiers as follows:

- The rules for the **G** and **D** matrices are the same
- For the **B** matrix, the opamp is considered to be a Voltage Source with the positive end connected to the node of the output of the opamp and the negative end connected to the ground. Then the matrix is completed using the rules stated above for matrix **B**.
- For the **C** matrix, the opamp is considered to be a Voltage Source with the positive end connected to the node of the non-inverting input of the opamp and the negative end connected to the node of the inverting input of the opamp. Again, the matrix is completed using the rules stated above for matrix **C**.
- Lastly, in the **b** vector, the element that represents the opamp (which the algorithm treats as a voltage source) has the value 0.

Example with Opamp

We consider the circuit depicted in figure 2.11. Following the algorithm described above, we obtain:

$$\begin{bmatrix} \frac{1}{R_1} & -\frac{1}{R_1} & 0 & 0 & 1 & 0 \\ -\frac{1}{R_1} & \frac{1}{R_1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{R_2} + \frac{1}{R_3} & -\frac{1}{R_2} & 0 & 0 \\ 0 & 0 & -\frac{1}{R_2} & \frac{1}{R_2} & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ I_{VS1} \\ I_{Op} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ VS1 \\ 0 \end{bmatrix}$$

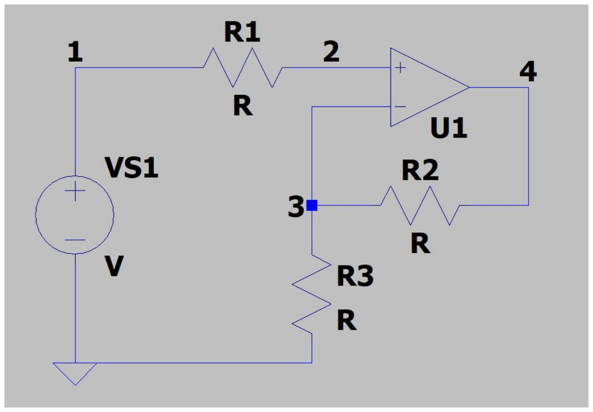


Figure 2.11: Linear Circuit Example with Operational Amplifier

2.6 Non-linear Components

Nodal analysis works perfectly for linear components, however, the circuit may include non-linear components as well. The equations in this case are called nonlinear algebraic systems and a common method used to solve them is the Newton-Raphson algorithm.

2.6.1 Newton-Raphson Algorithm

The Newton-Raphson algorithm solves a non-linear system of equations by taking an initial random solution and keep iterating to make the result more accurate [39]. At each iteration, the equation $f(x) = 0$ is solved, assuming that $f'(x)$ can be computed. Each iteration is obtained by calculating:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.8}$$

Of course, in this case x is not scalar, but a vector, resulting to 2.9

$$\boldsymbol{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \boldsymbol{f}(\boldsymbol{X}) = \begin{bmatrix} f_1(x_1, x_2, \cdots, x_n) \\ f_2(x_1, x_2, \cdots, x_n) \\ \vdots \\ f_n(x_1, x_2, \cdots, x_n) \end{bmatrix} \tag{2.9}$$

However, now, $f(x_n)/f'(x_n)$ is a matrix division, which is equivalent to the inverse, so we multiply $\boldsymbol{f}(\boldsymbol{X}^k)$ with the inverse of $f'(x_n)$. To obtain the derivative of f, we must derive the Jacobian matrix, represented by $\boldsymbol{J}$, shown below:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{2.10}$$

Then, by finding the inverse of the Jacobian matrix, we calculate:

$$\boldsymbol{X}^{k+1} = \boldsymbol{X}^k - \boldsymbol{J}^{-1}\left(\boldsymbol{X}^k\right)\boldsymbol{f}\left(\boldsymbol{X}^k\right) \tag{2.11}$$

The inverse Jacobian is not easy to calculate, so by re-arranging equation 2.10, we obtain:

$$\boldsymbol{J}\left(\boldsymbol{X}^k\right)\delta\boldsymbol{X}^k = -\boldsymbol{f}\left(\boldsymbol{X}^k\right) \tag{2.12}$$

where $\delta\boldsymbol{X}^k = \boldsymbol{X}^{k+1} - \boldsymbol{X}^k$.

Equation 2.12 will be used at each iteration to calculate the updated solution. The number of iterations is determined by the user, although, usually, a convergence criterion is used such that if $|v_n - v_{n-1}| < \varepsilon$ we assume that the algorithm has converged and we can stop the calculations. ε is again chosen by the user, based on the precision they require.

2.6.2 Diode

Diodes are one of the most common non-linear components. Its Current vs. Voltage relationship is described by equation 2.13 and is depicted in figure 2.12. I_s is the saturation current and V_t is the thermal voltage with typical values 10^{-15}A and 0.025875V respectively.

$$I_d = I_s(e^{\frac{V_d}{V_t}} - 1) \tag{2.13}$$

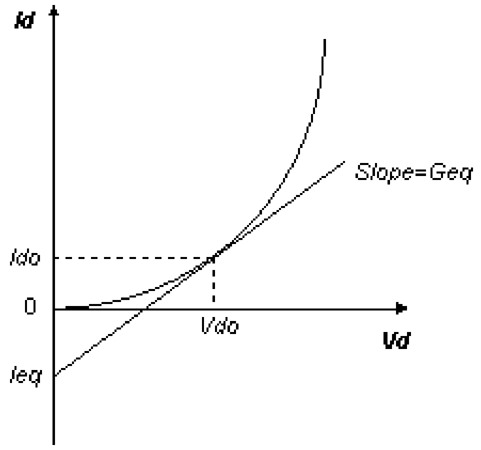


Figure 2.12: Diode characteristic curve and tangent line [38]

Around the operating voltage V_{do} we can approximate the behavior by a line tangent to the ideal curve [38]. This allows us to represent the diode by using a parallel combination of a Conductance G_{eq} and a current source I_{eq}. The conductance G_{eq} is the slope of the tangent at the operating point V_{do} as shown in figure 2.12, and thus:

$$G_{eq} = \frac{dI_d}{dV_d} = \frac{I_s}{V_t}e^{\frac{V_{do}}{V_t}} \tag{2.14}$$

$$I_{eq} = I_{do} - G_{eq}V_{do} \tag{2.15}$$

resulting in the following linear approximation circuit of a diode:

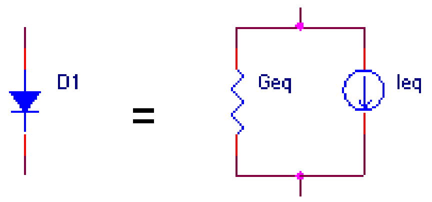


Figure 2.13: Diode Approximation Circuit [38]

Diode - Constant Voltage Drop Equivalent

Given the characteristic curve of a diode, a simplified diode version is often used for the paper analysis of circuits. This simplified equivalent is also used in the ADC module, when students are introduced to non-linear components and diodes. Essentially, the Current/Voltage depicted in 2.13 is approximated as two straight lines as shown in Figure 2.14 which intersect at 0.7 Volts. This means that its operation matches the ideal diode operation, but with a more accurate representation of the voltage drop of a real diode. The characteristics of this simplified diode are summarized in Table 2.4.

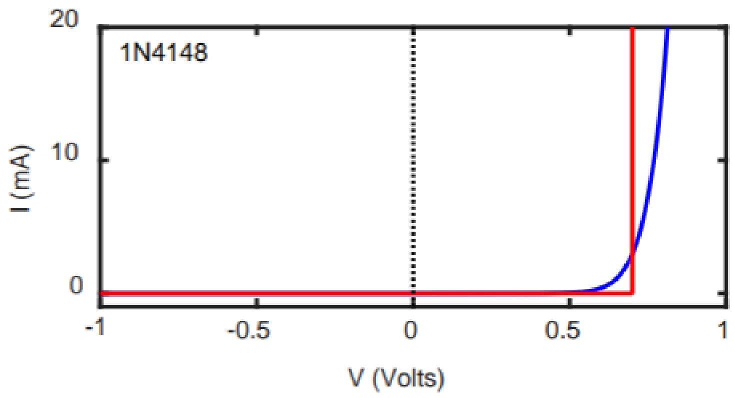


Figure 2.14: Simplified Diode Equivalent (from ADC lecture slides)

Region	Condition	Equation
Conducting Mode (“on”)	$I > 0$	$V = 0.7$
Non-conducting Mode (“off”)	$V < 0.7$	$I = 0$

Table 2.4: Linearized Real Diode Modes

2.7 Pathological Circuits

The identification of pathological circuits is a matter that requires special attention in the current project. The existing analog circuit simulators typically don’t handle pathological or erroneous circuits in a specific way. They simulate the circuit as designed, and if the circuit contains errors or is unstable, the simulation will reflect that either by stopping the simulation and returning undefined values (`undef`) and/or by returning an error message. It is well known that such circuits exist and various approaches exist to overcome them such as the modified nodal analysis (MNA) or using a framework based on a definition of port equivalence for admittance matrices [24].

Examples of such circuits include cases where the simulation results in infinite or no solutions. A circuit that produces no solutions in DC operating point analysis is a circuit with conflicting constraints that cannot be simultaneously satisfied. That is, for example, a circuit where both ends of a voltage source are connected to ground (figure 2.15). For the multiple solutions case, this would be a circuit with two voltage sources connected in parallel (figure 2.16), creating a node with an infinite number of possible voltages, resulting in an infinite number of solutions. A key challenge of this project will be to provide understandable user feedback when dealing with a pathological circuit.

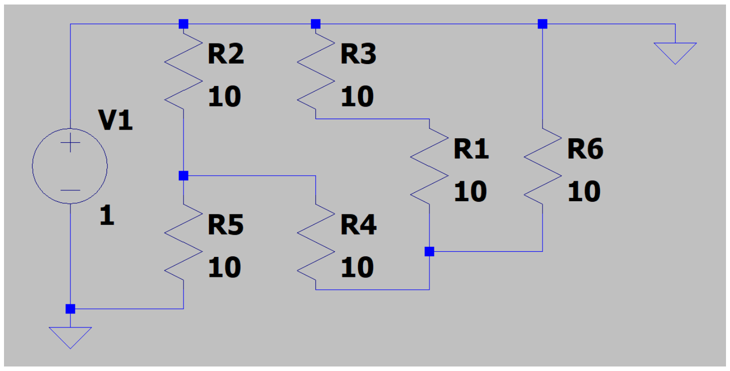


Figure 2.15: Example circuit with no solutions

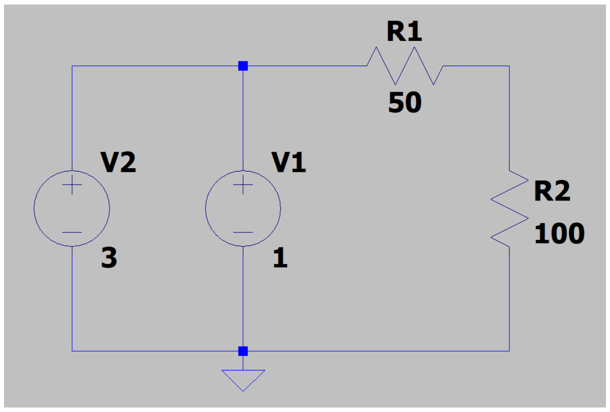


Figure 2.16: Example circuit with infinite solutions

Chapter 3

Project Requirements

In this section, a comprehensive set of requirements for this project will be specified. These requirements, create a more formal specification of the project aims and are the foundation for all the design choices that will be analyzed in the coming sections. Upon completion of the project, the final version of the produced application will be evaluated against the requirements laid out in this section, to evaluate the result.

3.1 Requirements for Analog Circuit Editor & Simulator

As explained in the introduction, the goal of this project is the creation of an Analog Circuit Design platform to improve the experience of first-year university students and make the learning of the basic Circuit Analysis principles easy and straightforward. The application is principally targeted at Imperial College Electrical & Electronic Engineering students, but given its generalized format, it can be adopted by other universities worldwide as well.

According to the Imperial College EE Analysis & Design of Circuits (ADC) module syllabus (module code: ELEC40002), students are expected to learn the following concepts: Voltage and Current in a circuit, KCL, KVL, Nodal Analysis, Phasors, Frequency Response, Amplifiers, Diodes, Transients. The application should provide visualization methods that help students comprehend the aforementioned concepts in the circuits they design. The requirements will be split into essential **(Ex)** and desirable **(Dx)**. For the project to be considered successful, all the essential requirements must be met.

Essential Requirements

E1.1 Include a graphical interface that allows users to easily design complex analog circuits containing any number of components

E1.2 Include a set of built-in components (resistor, capacitor, inductor, diode, operational amplifier, Voltage and Current sources) which the users can use to create their designs

E1.3 Allow users to easily connect components using wire auto-routing

E1.4 Provide thorough feedback for errors that carefully explain the issues that are present in a design.

- **E1.4.1** Errors should be visualized directly on the circuit diagram, to maximize the amount of guidance given to the users.

E1.5 Clearly indicate the locations of the nodes of the circuit on the canvas.

E1.6 Allow users to perform DC Operating Point Analysis by performing Modified Nodal Analysis. The algorithm should work for all available components. Summarize the node voltages and source currents as a table on the right-hand side of the screen.

E1.6.1 Calculate all the branch currents and include them in the DC Table.

E1.6.2 Allow users to visualize node voltages and branch currents on the canvas, to allow faster reading of DC results

E1.7 Allow the use of parameters for the values of components (DC voltage of sources, Resistance of resistor, etc.) which can be easily changed (slide-bar) with real-time effect in simulation results.

E1. Allow users to perform Frequency Response Analysis of circuits by specifying an input source and an output node.

E1.8.1 Present the AC Analysis Results as two distinct plots (Magnitude and Phase) in the same graph, with two distinct y-axes.

Desirable Requirements

D1.1 Using the same area as in E1.7 allow users to run time domain simulation of their circuit

D1.1.1 Allow users to run time domain simulations containing only non-reactive components (sources, resistors, diodes, amplifiers)

D1.1.2 Allow users to run time domain simulations containing up to one reactive component, specifying both the steady-state and transient solutions, and giving a clear indication of the transient parameters (time constant τ, transient amplitude, HF & DC Gains)

D1.1.3 Allow users to run time domain simulations of any circuit.

D1.2 Provide more flexibility with all the graph simulations.

D1.2.1 Allow users to change the units of the graph axes.

D1.2.2 Allow users to detach the graphs into separate windows.

D1.2.3 Provide both actual and straight-line approximations of frequency response in AC Analysis

D1.2.4 Provide high precision plots in AC Analysis

D1.3 Support non-linear components (starting from diodes) in the application.

D1.4 Allow users to obtain the Thevenin/Norton equivalent circuit seen by a two terminal circuit, by presenting (in a table) the Thevenin/Norton parameters.

D1.5 Present a set of straightforward equations that can be obtained from the circuit to help students understand the relations that formulate the solutions.

D1.6 Allow embedded exercises with the circuits users build (e.g. draw a straight-line approximation of magnitude and phase).

D1.7 Provide students with symbolic transfer functions of the relations they are simulating in AC Analysis.

3.2 Software/Documentation Quality Requirements

Given the nature of this project, the application will consist of more than 10.000 lines of F# code. The basis of the application will come from ISSIE (Interactive Schematic Simulator and Integrated Editor)[13], a tool currently used by Imperial College EE for digital design simulations. This section highlights the requirements for the quality of the code developed, along with its documentation.

Essential & Desirable Requirements

E2.1 Deliver bug-free and performant code that adheres to ISSIE's coding guidelines [14].

E2.2 Deliver easy-to-maintain code such that it can be improved in the future by Imperial students. This is very important considering the remarkable and impressive advancements of ISSIE over the years.

E2.3 Provide comments in the delivered code which explain the purposes of each function.

E2.4 Analyse any non-trivial algorithms and/or data structures in the GitHub Wiki of the repository

E2.5 Provide all the necessary information for users and developers in a readme markdown file, in the project's repository.

D2.1 Create a static website using Jekyll and GitHub pages to demonstrate the features of the application and advertise it.

Chapter 4

Analysis and Design

Right from the start of this project, it was clear that it was an open-ended project, with a large list of features that can potentially be implemented. Therefore, given the time constraints of a Final Year Project, the final deliverable of this project must be seen as the starting point for a multi-year development process, following an improvement pattern similar to ISSIE. The features I chose to implement represent a core subset with enough functionality to be user tested in the ADC module next year, while also allowing clean subsequent development.

4.1 Top-Level Planning

Before diving into the specifics, it was crucial to establish an overarching plan. This plan comprised of 5 distinct stages, each executed sequentially. The initial and pivotal step involved conducting comprehensive research on analog circuit simulation algorithms and the technology stack. Once the decision was made to leverage the main UI of ISSIE, the second step entailed refining the ISSIE code to include only the essential functionalities. This refined state served as the foundation for the remainder of the project.

Subsequently, the focus shifted towards building on top of the foundation, to complete the main UI of ADDIE (new components, new top-level UI, etc.). With the completion of the primary UI development for ADDIE, the subsequent step involved implementing simulation algorithms and visually representing the results. This is the largest step, which required a prioritization of the sub-tasks to be completed. This prioritization list is analyzed below in section 4.2.

Finally, the last step encompassed code cleanup and report writing to ensure a polished final product.

4.2 Prioritization of Simulation Tasks

As previously discussed, considering the abundance of potential features for a tool like ADDIE, a thoughtful priority list was meticulously crafted. This list will guide the implementation process, ensuring that the limited time available for this project is used as effectively as possible to deliver the most impactful and valuable outcome. The tasks that are crossed out indicate tasks that have been partially completed or have not been implemented yet. Details about each task's implementation can be found in chapter 5.

1. **Node Identification:** Parsing the circuit to identify the nodes and their location on the canvas. This is at the top of the list as any sort of simulation requires the identification and labeling of nodes.

2. **DC Analysis (Stage 1):** DC Operating Point Analysis for circuits containing only: (i) resistors and (ii) independent Voltage/Current sources. This is the bare minimum a simulation application must offer in order to be considered a simulator. It consists of an initial simplified implementation of Modified Nodal Analysis (MNA) on which I can build on to incorporate other components as well.

3. **DC Analysis (Stage 2):** Improve DC Operating Point Analysis: Expand the DC Analysis completed above to include reactive components (Capacitor/Inductor)

4. **DC Result Representation:** Summarize the results obtained by MNA (node voltages and sources currents) in a table under the DC Simulation sub-tub. The table should have two columns, one for the description (i.e. V(Node 1) or I(VS1)) and one for the value.

5. **DC Analysis (Stage 3):** Complete the MNA algorithm (as outlined in section 2.5.1) to support Operational Amplifiers. This concludes the DC Operating Point Analysis for linear components.

6. **AC Analysis (Stage 1):** Implement AC Analysis for simple RC and RL circuits. Extract the Magnitude and Phase for a wide range of frequencies.

7. **Graph for AC Simulation:** Use the results obtained above to create two plots on the same graph, one for Magnitude and one for Phase. Set up two different y-axes, and a logarithmic x-axis.

8. **AC Analysis (Stage 2):** Complete AC Simulation by supporting any number of components, including Operational Amplifiers.

9. **DC Current Calculation:** Use the DC Operating Point Analysis Results to obtain all the branch currents.

10. **DC Simulation Visualisations:** Display the DC Operating Point Analysis results and branch currents on the canvas. This visualization should be configurable by buttons to allow users to control their visibility.

11. **Time Simulation (Stage 1):** Implement time simulation for circuits without reactive components. Essentially this is an extension of DC Analysis for multiple time points for both DC and Sinusoid input sources. As for AC Analysis, plot both the input and output signals against time in the graph area.

12. **Time Simulation (Stage 2):** Implement time simulation for circuits consisting of at most one reactive component and one voltage source, and any number of other components. This implementation will be based on the algorithm used for the paper analysis of circuits, as taught in the ADC module. Use the graph area to plot: (i) steady-state, (ii) transient, (iii) full output signal, and (iv) input signal against time.

13. ~~**Algebra Implementation for Transfer Functions:**~~ Incorporate a library that supports symbolic mathematical operations, to demonstrate the transfer functions of the simulated AC Analysis.

14. ~~**Real Diode Implementation:**~~ Derive and implement a robust algorithm to support specific non-linear components in simulations (beginning from real diodes).

15. **Linearized Diodes (Stage 1):** Support for all 3 simulation types circuits that include one diode component. The diode supported will be a linearized version of the real diode, as discussed in section 5.6.3.

16. **Linearized Diodes (Stage 2):** Utilise the algorithm of stage 1 to create a top-level algorithm

that is able to support any number of diodes. This must be done unavoidably using exhaustive search as the diode operation modes might be related to each other.

17. **Linearized Diodes (Stage 3):** Optimise the algorithm of stage 2 by caching the diode modes that satisfy the necessary conditions.

18. **Support both dB/non-DB and f/ω axis:** Allow users to choose the units of the axis and change from dB to non-DB and f to ω and vice-versa via the use of buttons in the AC Simulation sub-tub.

19. ~~**Straight-line approximation in AC graph:**~~ Include an extra pair of plots in the graph that show the straight-line approximation of the frequency response. To ensure visual clarity within the graph area, these approximations will be initially hidden but will be indicated in the legend. This empowers users to choose whether they wish to view the straight-line approximations or not, granting them control over their visualization preferences.

20. **More points around regions of interest:** Have more data points around regions of interest (e.g. corner frequency), to better capture the response of the circuit. This can be implemented either by using an adaptive step size (using mutable variables) or by calculating the response twice: initially to find the regions of interest and then with more data points around those regions.

21. ~~**Multiple nodes in AC Simulation:**~~ Allow AC Simulation to visualize more than one node in the graph.

22. ~~**Graph Divider:**~~ Improve the UI of the graph by adding a draggable `dividerbar` that can resize the graph area, and consequently the graph.

23. **Thévenin Equivalent Circuit in DC:** The algorithm for finding the Thévenin equivalent seen by the two terminals of a component already exists as it is used for time simulation. Allow users to find the Thévenin equivalent circuits in DC Analysis by specifying the component from a drop-down.

24. **DC Equations:** Derive straight-forward equations formed by a circuit (e.g. Voltage of nodes connected to voltage source terminals or potential dividers) and display them under the DC Simulation sub-tub.

25. ~~**Embedded Exercises:**~~ Use the graph area to allow users to draw straight-line approximations before simulating their circuits. Then they can compare their result with the real straight-line approximation.

4.3 Decisions taken during the analysis stages

In this section, I will provide a concise summary of the key decisions I made during the project, which ultimately led to the creation of the final deliverable. These decisions were primarily driven by addressing challenges encountered during the implementation stages or managing the time available as effectively as possible.

Math.NET Symbolics Library

One of the project's desirable features was the inclusion of a symbolic algebra library, enabling users to examine transfer functions and other relationships within their circuits. The issue encountered was that there were no JavaScript libraries supporting symbolic operations. Instead, a .NET library was found, `Math.NET Symbolics`, which however wasn't supported by Fable. After some research, it was discovered that it is possible to run any .NET dll under `webassembly` which interfaces with `Electron` [44]. Nonetheless, the configuration process for this approach would be time-consuming, given the absence of pre-existing setups online. Also, the execution of .NET under `webassembly` is a very new and immature concept, and thus, its incorporation is considered high-risk at the moment, given that it could be prone to erroneous behavior that can affect the stability of the application. Therefore, considering the limited utility of the symbolic algebra library within the context of the current project (obtaining symbolic transfer functions), the decision was made to relegate this feature to a lower priority on the task list, which would be undertaken only if there is sufficient time available towards the end of the project.

Real Diodes and Non-Linear Components

The simulation of non-linear components like diodes and transistors requires an implementation of the iterative "Newton-Raphson" method. A stable algorithm to support non-linear components was derived and is analyzed below, in section 4.4. However, further research is required in order to properly assess the algorithm, its capabilities and drawbacks, and the potential issues that come with it. A brief overview of those is given in section 4.4.3. Given the restricted time of this project, and considering the fact that a robust implementation of non-linear simulations is a separate large project that can be completed as an extension of the current, it was decided to limit the supported features to the material of the first part of the ADC module and focus more on appropriate visualizations around that content. After all, the aim of this project is to explore novel ways in which visualizations and simulations can help first-year students understand basic analog electronics concepts. Creating from scratch a simulator that supports all the basic features of LTSpice, along with the extra visualizations, definitely exceeds the scope of a final-year project and cannot be accomplished within the given time constraints.

Time Simulation

As for the simulation of non-linear components, a universal time domain simulation requires the implementation of numerical integration methods, such as the trapezoidal rule or Gear's method, to solve the system of differential equations comprising a circuit [27]. This task requires a heavier math library with a larger set of features. However, universal time simulation is not a concept taught to first-year students. Instead, students focus on solving first-order circuits, consisting of a single reactive component, either a capacitor or an inductor. This allows for a much simpler simulation algorithm, which is what the final deliverable supports. From the beginning of this project, when defining the set of essential and desirable requirements, time simulation was included in the set of desirable features, because of the high complexity of a universal time simulation algorithm. The simplified time simulation algorithm used is thoroughly analyzed in section 5.8 where the implementation process for time simulation is discussed.

4.4 Incorporation of Modified Nodal Analysis with Newton-Raphson

A major challenge of this project was deriving a stable algorithm that would allow the incorporation of the Modified Nodal Analysis algorithm, with the iterative Newton-Raphson method used to approximate non-linear circuits. Newton-Raphson is already analyzed in section 2.6.1, and it works well with classic nodal analysis, with many examples available online on how to incorporate the two. The challenge was to extend the algorithm to work with Modified Nodal Analysis (MNA). In this section, two examples will be presented, analyzing the maths behind the implementation.

4.4.1 Newton-Raphson Example 1

Starting from MNA, we formulate the system of equations following the algorithm defined in section 2.5.1. This system does not account for the diode, which is assumed to be a 0.7 voltage source following the logic described in sections 2.6.2 and 5.6.3.

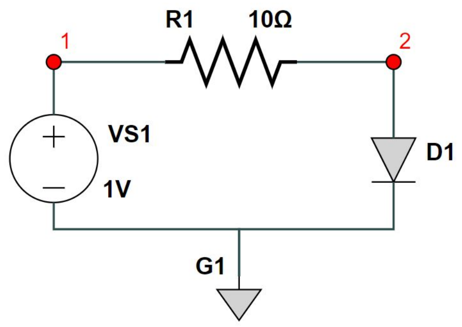


Figure 4.1: Diode Circuit Example 1

$$\begin{bmatrix} \frac{1}{R} & -\frac{1}{R} & 1 \\ -\frac{1}{R} & \frac{1}{R} & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ I_{VS} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ V(VS) \end{bmatrix} \tag{4.1}$$

Therefore, we have our set of unknowns $\mathbf{X} = [V_1, V_2, I_{VS}]$ and the following set of three equations from KCL and KVL that formulate the matrix above.

$$f(\mathbf{X}) = \begin{cases} \frac{V1}{R} - \frac{V2}{R} + I_{VS} \\ \frac{V2}{R} - \frac{V1}{R} + I_D & = \mathbf{0} \\ V_1 - 1 \end{cases} \tag{4.2}$$

where I_D is the diode current given by equation 2.13, $I_D = I_s(e^{\frac{V_d}{v_t}} - 1)$, I_s and V_t being the saturation current and thermal voltage respectively. As the initial solution, we will use the vector $\mathbf{X^0} = [1, 0.7, -0.03]$ which is the result following from the MNA implementation as analyzed in section 5.6.3 which uses the linearized equivalent diode model.

Then, following the procedure of Newton-Raphson, we obtain the Jacobian matrix $\mathbf{J}$, differentiating the three equations in 4.2.

$$\boldsymbol{J}(\mathbf{X^0}) = \begin{bmatrix} \frac{\partial f_1}{\partial V_1} & \frac{\partial f_1}{\partial V_2} & \frac{\partial f_1}{\partial I_{VS}} \\ \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial I_{VS}} \\ \frac{\partial f_3}{\partial V_1} & \frac{\partial f_3}{\partial V_2} & \frac{\partial f_3}{\partial I_{VS}} \end{bmatrix} = \begin{bmatrix} \frac{1}{R} & -\frac{1}{R} & 1 \\ -\frac{1}{R} & \frac{1}{R} + \frac{I_s}{V_t} e^{\frac{V_2}{v_t}} & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.1 & -0.1 & 1 \\ -0.1 & 0.12169 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{4.3}$$

as $(I_s(e^{\frac{V_2}{v_t}} - 1))' = \frac{I_s}{V_t} e^{\frac{V_2}{v_t}} = 0.0216845$. Then we calculate

$$f(\mathbf{X^0}) = \begin{bmatrix} 0 \\ -0.03 + 0.000561 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.029439 \\ 0 \end{bmatrix} \tag{4.4}$$

and then using equation 2.12, we find $\delta\mathbf{X^0}$ by calculating

$$\delta\mathbf{X^0} = \mathbf{J}^{-1}(\mathbf{X^0})(-\mathbf{f}(\mathbf{X^0})) = \begin{bmatrix} 0.1 & -0.1 & 1 \\ -0.1 & 0.12169 & 0 \\ 1 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0.029439 \\ 0 \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 8.2236 & 0.82236 \\ 1 & 0.82236 & -0.017 \end{bmatrix} \begin{bmatrix} 0 \\ 0.029439 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.2420 \\ 0.02420 \end{bmatrix} \tag{4.5}$$

Then, we derive the next solution, $\mathbf{X^1}$, by calculating:

$$\mathbf{X^1} = \mathbf{X^0} + \delta\mathbf{X^0} = \begin{bmatrix} 1 \\ 0.7 \\ -0.03 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.2420 \\ 0.02420 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.9420 \\ -0.0058 \end{bmatrix} \tag{4.6}$$

.

The procedure is then continued by calculating $\mathbf{J}(\mathbf{X^1})$, $f(\mathbf{X^1})$ and $\delta\mathbf{X^1}$ to obtain $\mathbf{X^2}$, and is repeated, until the solution converges ($\mathbf{X^{k+1}} - \mathbf{X^k} < \epsilon$). For the current example, using $\epsilon = 10^{-7}$, the algorithm converges after 11 iterations.

Iteration i	$\mathbf{X^i}$
0	$[1,\ 0.7,\ -0.03]$
1	$[1,\ 0.94192825,\ -0.00580718]$
2	$[1,\ 0.9160869,\ -0.00839131]$
3	$[1,\ 0.89033132,\ -0.01096687]$
4	$[1,\ 0.86485449,\ -0.01351455]$
5	$[1,\ 0.84023943,\ -0.01597606]$
6	$[1,\ 0.818079,\ -0.0181921]$
7	$[1,\ 0.80173672,\ -0.01982633]$
8	$[1,\ 0.79444791,\ -0.02055521]$
9	$[1,\ 0.79333937,\ -0.02066606]$
10	$[1,\ 0.79331765,\ -0.02066823]$
11	$[1,\ 0.79331764,\ -0.02066824]$

Table 4.1: Diode Circuit Example 1: Solutions after each iteration

4.4.2 Newton-Raphson Example 2

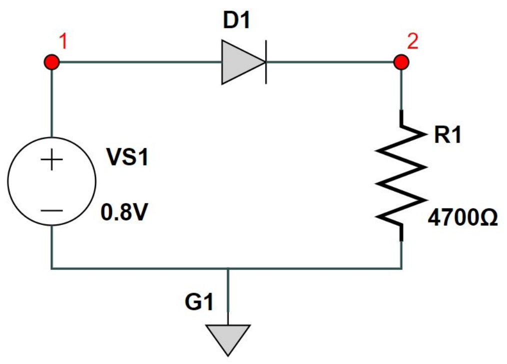


Figure 4.2: Diode Circuit Example 2

This circuit is slightly more complex given that the voltage across the diode V_d is the difference of V_1 and V_2 and cannot be simplified as in the previous example (where $V_d = V_2$). This second example serves as a compelling proof of concept, illustrating the versatility and effectiveness of the algorithm across different types of circuits. Again, the set of unknowns is $\mathbf{X} = [V_1, V_2, I_{VS}]$ and the three formulated equations are:

$$f(\mathbf{X}) = \begin{cases} I_s(e^{\frac{V_1-V_2}{v_t}} - 1) + I_{VS} \\ -I_s(e^{\frac{V_1-V_2}{v_t}} - 1) + \frac{V2}{R} \\ V_1 - 0.8 \end{cases} = \mathbf{0} \tag{4.7}$$

As the initial solution, we will use the vector $\mathbf{X^0} = [0.8, 0.1, -0.0000213]$ which is the result following from the MNA implementation as analyzed in section 5.6.3 which uses the linearized equivalent diode model. Then, following the procedure of Newton-Raphson, we obtain the Jacobian matrix $\mathbf{J}$, differentiating the three equations in 4.7. In this example, we will use $I_s = 3.3nA$ and $V_t = 0.04621V$.

$$\boldsymbol{J}(\mathbf{X^0}) = \begin{bmatrix} \frac{\partial f_1}{\partial V_1} & \frac{\partial f_1}{\partial V_2} & \frac{\partial f_1}{\partial I_{VS}} \\ \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial V_1} & \frac{\partial f_2}{\partial I_{VS}} \\ \frac{\partial f_3}{\partial V_1} & \frac{\partial f_3}{\partial V_2} & \frac{\partial f_3}{\partial I_{VS}} \end{bmatrix} = \begin{bmatrix} \frac{I_s}{V_t}e^{\frac{V_1-V_2}{v_t}} & -\frac{I_s}{V_t}e^{\frac{V_1-V_2}{v_t}} & 1 \\ -\frac{I_s}{V_t}e^{\frac{V_1-V_2}{v_t}} & \frac{I_s}{V_t}e^{\frac{V_1-V_2}{v_t}} + \frac{1}{R} & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.27075 & -0.27075 & 1 \\ -0.27075 & 0.27077 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{4.8}$$

as $(I_s(e^{\frac{V_1-V_2}{v_t}} - 1))' = \frac{I_s}{V_t}e^{\frac{V_1-V_2}{v_t}} = 0.27075$ for $I_s = 3.3nA$ and $V_t = 0.04621V$. Then we calculate

$$f(\mathbf{X^0}) = \begin{bmatrix} 0.0124902 \\ -0.01249022 \\ 0 \end{bmatrix} \tag{4.9}$$

and then using equation 2.12, we find $\delta\mathbf{X^0}$ by calculating

$$\begin{aligned} \delta\mathbf{X^0} = \mathbf{J}^{-1}(\mathbf{X^0})(-\mathbf{f}(\mathbf{X^0})) &= \begin{bmatrix} 0.27075 & -0.27075 & 1 \\ -0.27075 & 0.27077 & 0 \\ 1 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 0.0124902 \\ -0.01249022 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 3.6905 & 0.9992 \\ 1 & 0.9992 & -0.000213 \end{bmatrix} \begin{bmatrix} 0.0124902 \\ -0.01249022 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.0460951 \\ -0.00000978 \end{bmatrix} \end{aligned} \tag{4.10}$$

Then, we derive the next solution, $\mathbf{X^1}$, by calculating:

$$\mathbf{X^1} = \mathbf{X^0} + \delta\mathbf{X^0} = \begin{bmatrix} 0.8 \\ 0.1 \\ -0.0000213 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.0460951 \\ -0.00000978 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.1461 \\ -0.00003108 \end{bmatrix} \tag{4.11}$$

.

The procedure is then continued by calculating $\mathbf{J}(\mathbf{X^1})$, $f(\mathbf{X^1})$ and $\delta\mathbf{X^1}$ to obtain $\mathbf{X^2}$, and is repeated, until the solution converges ($\mathbf{X^{k+1}} - \mathbf{X^k} < \epsilon$). For the current example, using $\epsilon = 10^{-7}$, the algorithm converges after 10 iterations.

Iteration i	$\mathbf{X^i}$
0	$[0.8,\ 0.1,\ -0.0000213]$
1	$[0.8,\ 0.146095182,\ -0.000031084]$
2	$[0.8,\ 1.91896255,\ -0.000040828]$
3	$[0.8,\ 2.367469772,\ -0.000050371]$
4	$[0.8,\ 2.78732942,\ -0.000059304]$
5	$[0.8,\ 3.13169282,\ -0.000066631]$
6	$[0.8,\ 3.33002552,\ -0.000070851]$
7	$[0.8,\ 3.38079534,\ -0.000071931]$
8	$[0.8,\ 3.38343837,\ -0.000071988]$
9	$[0.8,\ 3.38344505,\ -0.000071988]$
10	$[0.8,\ 3.38344505,\ -0.00001988]$

Table 4.2: Diode Circuit Example 2: Solutions after each iteration

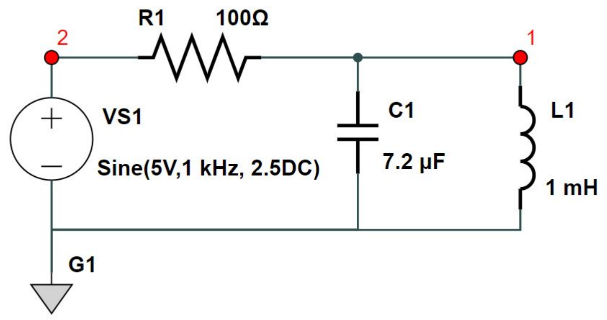
4.4.3 Evaluation of the Algorithm

The algorithm designed above works well for simple circuits containing only diodes. However, if it were to be extended for BJTs and other non-linear components, the cases where the algorithm does not operate as expected must be examined. A simple problematic scenario that can occur is when the initial solution is too far from the true solution and the algorithm fails to converge. This can be resolved by trying another initial solution, closer to the expected solution of the circuit. A more advanced problem, that will be encountered when dealing with more complex circuits, is finding all the operating points (DC Solutions) of a non-linear circuit. This is a fundamental problem in the computer-aided design of analog circuits, and the following paper by Prof. Stanisław Hałgas [25], presents a SPICE-Oriented Method for Finding Multiple DC Solutions in Nonlinear Circuits using a deflation-based technique [5].

4.5 High Precision AC Analysis

The simple algorithm used for AC Analysis, as analyzed in section 5.7, consists of investigating the Magnitude and Phase of the ratio (Output Node/Input Source) for frequencies starting at 1 rad/s up to 10000000 rads/s, with 20 equally spaced frequency points in the logarithmic axis. The produced plots are relatively accurate and smooth, however, they tend to be erroneous around the corner frequencies when a specific frequency point (20 per decade) does not coincide with a corner frequency. An example of such scenario is depicted in figure 4.3 where the peak at the corner frequency is not visible, as the corner frequency is not in the list of predefined frequency points. The optimal solution to this problem requires the incorporation of a symbolic math library, which would allow the explicit calculation of the corner frequencies. Following the decision not to incorporate such a library for the reasons discussed in section 4.3, an alternative high-precision AC Analysis algorithm is presented in this section.

The essence of the algorithm revolves around the concept of the frequency response of the phase for any network. Specifically, phase remains constant in the regions far from the corner frequencies, and changes abruptly near them. This is a characteristic that can be exploited for the purposes of AC Analysis, in order to apply an adaptive step size, instead of a constant one (20 points per decade). Specifically, the algorithm will start again from 1 rad/s with the existing step size (20 points per decade), and then when a change of more than 1° is detected between the two previous frequency points, the step size will become 2-10 times smaller (depending on the precision needs). Of course, in order to be able to detect the end of the region of interest, the specified difference between two points that the algorithm looks for will have to be reduced an equal amount of times (2-10). Then, when the difference between the last two points is larger than the updated difference, the step size will be set back to 20 points per decade.



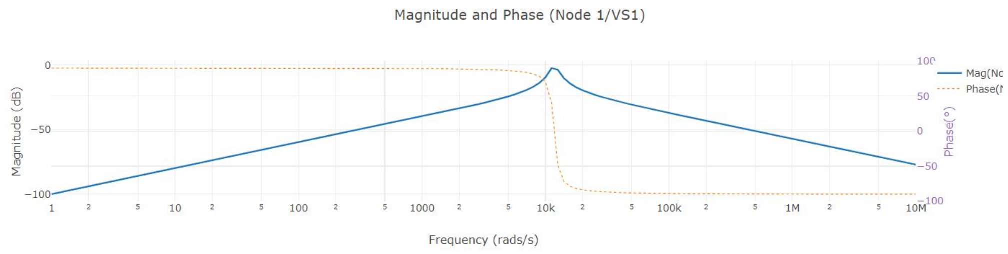


Figure 4.3: Non-optimized Frequency Response Plot

This algorithm, of course, can be varied accordingly, by introducing various different thresholds (differences between the last two points) for further improving the accuracy as the frequency points approach the corner frequency. However, even with this approach, it is not guaranteed that the plot will reflect 100% accurately the corner frequency peaks; but, it is a significant improvement compared to the initial non-adaptive-step-size algorithm.

Chapter 5

Implementation

This chapter analyses the implementation of all the features enumerated in Chapter 3, focusing on the specific algorithms and data structures used. The code changes and additions are implemented in F# , and the complete codebase can be found in the Project's GitHub Repository. The specific repository serves also as a logbook of the work done for this project, as the commit history of the repository specifies analytically the code alterations made, along with their timestamp.

5.1 Initial GUI Modifications - From ISSIE to ADDIE

As already explained in section 2.3, the current application shares many characteristics with ISSIE, an open-source Digital Design Editor and simulator used and maintained by the EEE department of Imperial College London. Furthermore, the source code of ISSIE is written and structured with readability, re-usability, and efficiency in mind - implying that it is a very easy-to-understand and re-use code. Therefore, I decided to follow ISSIE's technology stack and use its source code as a basis for my application.

Following this decision, the initial implementation stages consisted of GUI alterations and source code deletions to transform the application into a state which would act as a basis for the remainder of the project. Analytically, this required the deletion of all the non-needed features, such as simulation, truth table, and verilog source code, and the modification of the GUI into the format of ADDIE (new components, main layout changes, etc.).

5.1.1 Current structure of ISSIE

The directory tree of the **Renderer** directory which is essentially the source of the application is as follows:

```
Renderer
├─ Common
├─ Drawblock
├─ Interface
├─ Simulator
├─ UI
│   ├─ WaveSim
│   └─ TruthTable
├─ VerilogComp
└─ Renderer.fs
```

1. `Common`: Contains all types used throughout ISSIE (Component, Connection, etc.) as well as various helper files (DrawHelpers.fs, TimeHelpers.fs, etc.) used in various parts of the codebase, and the Width Inference code.
2. `Drawblock`: Contains all the code related to the Canvas (drawing/-moving/connecting/copying/deleting components). Code is separated into Sheet, BusWire, and Symbol.
3. `Interface`: Electron and JS interface files; all file read/write code.
4. `Simulator`: ISSIE's step simulator logic
5. `UI`: Contains all the non-canvas elements of the application, along with code to handle users' interaction. The 2 sub-folders contain the logic behind the waveform simulator and the truth table.
6. `VerilogComp`: Contains code used for the creation of ISSIE's Verilog Component.

5.1.2 Changes per Directory

The table below summarizes the changes made to the ISSIE code base per directory.

Directory	**Alterations**
Common	Width Inference code was deleted. Appropriate changes were made in the CommonTypes.fs file to match the new purpose of the application (New components, no input/output ports, no custom components). All the helper files were kept as they will be useful in the current application as well.
DrawBlock	Lots of code related to non-used canvas operations was deleted (mostly around custom components). Symbol drawing functions were altered to match the new components (resistor, capacitor, etc. drawn in SVG). Wiring functions needed some changes to adapt to the new system of connections (no input/output).
Interface	No changes.
Simulator	All source code was deleted.
UI	TruthTable and WaveSim code was deleted. Various parts of the main UI were deleted as well (Popups, Catalogue tab, Simulation tab, Memory, Custom-component-related code), and others were changed to match the new UI (components on the top navbar, etc.)
VerilogComp	All source code was deleted.

Table 5.1: Analysis of changes to the ISSIE code base

5.2 Overview of new Data Types

After choosing to employ the ISSIE code as the foundation for ADDIE, the exclusive adjustments required for the Data Types used in the project were specifically related to simulation results. Within this section, a concise depiction of each novel type will be presented, accompanied by its application within the code.

Data Type	Description

`SimData`	Record containing the necessary information to perform AC and Time Analysis: {
	• `ACSource: string option //input VoltageSource id`
	• `ACOutput: string option //output node`
	• `ACMagInDB: bool`
	• `ACFreqInHz: bool`
	• `TimeInput: string option`
	• `TimeOutput: string option`
	}

`ComplexC` - `ComplexP`	Fable does not support the .NET Complex Library, and therefore, all the logic had to be coded from scratch, even for basic operations such as addition and multiplication. The two types `ComplexC` and `ComplexP` where C: Cartesian and P: Polar are both used throughout the code with the proper helper functions to convert from one to the other. `ComplexP = { Mag: float; Phase: float}` and `ComplexC = { Re: float; Im: float}`. Close attention was needed when calling the MathJS functions with complex numbers as input. More on this follows in subsection 5.4.1.
`DCSimulationResults`	Record containing the results of DC Operating Point Analysis: { • `MNA: float array //result produced by MNA` • `ComponentCurrents: Map<ComponentId, float>` • `NodeList: (Component*int option) list list` • `Equations: string list` } In `NodeList`, the top list represents the nodes (index 0 $\rightarrow$ Ground, index 1 $\rightarrow$ node 1, etc.) and the internal list per node contains the components connected to that node, along with the Port Number of the component connected to the node.
`ACSimulationResults`	Currently, `ACSimulationResults` is simply a `ComplexP list` containing the Magnitude and Phase per frequency. The frequency points are set to 20 steps per decade from 10^0 to 10^7.

`TimeSimulationResults`	Record containing the results of time simulation: {
	• `TimeSteps: float list`
	• `Transient: float list`
	• `SteadyState: float list`
	• `Tau: float`
	• `Alpha: float`
	• `HFGain: float`
	• `DCGain: float`
	}

Table 5.2: New Data Types Summary

Appropriate changes were made to the model and the list of MVU Messages (Msg), to store and update the information related to the types mentioned in Table 5.2. Apart from the above, the following fields were added to the model, to control the state of the application:

Field and Type	Description
`PrevCanvasStateSizes: int * int`	A tuple containing the length of the Component list and the Connection list which form the `CanvasState`. This information is used by the main `runSimulation()` function to detect whether there has been a deletion/addition on the circuit (either component or wire).

Field and Type	Description
`PreviousDiodeModes: bool list`	The bool list serves as a repository for the diode modes observed in the preceding simulation that fulfill the given conditions. To decode the list: true represents the conducting mode, while false represents the non-conducting mode. Its purpose is to optimize performance: when the `runSimulation()` function is invoked once more, the algorithm initially attempts the modes from the prior simulation. This results in a substantial performance boost, particularly for time simulations demanding execution across numerous discrete time points.
`NodeLocations: XYPos list`	This list contains the on-canvas positions of the nodes. Considering the node numbering convention (starting from 0 for the ground), employing a Map is unnecessary since the index directly corresponds to the node number. More information on the algorithm used to find the positions can be found in section 5.5
`ComponentCurrents: Map<ComponentId,float>`	Maps components to the current flowing through them. The sign of the current (positive/negative) depends on the orientation of the component on the canvas, assuming that the current arrow points from left to right or from top to bottom. This enables the straightforward and robust drawing of current arrows on the main canvas.
`NodeVoltages: float list`	A list containing the Voltage at each distinct node, using the same assumption (node number = index) as in `NodeLocations`.
`UpdateSim: bool`	Boolean which acts as the primary determinant for whether the simulation requires updating. It is set to true upon detecting any changes in the circuit and promptly reset to false once the simulation results have been updated
`CanRunSimulation: bool`	Boolean which becomes true when the circuit is devoid of any known errors prior to simulation
`SimulationRunning: bool`	True when simulation is running. Used as a control for all the visualizations.

Table 5.3: New Fields in the Model

5.3 Top-level UI Changes

This section describes a list of Top-level UI changes conducted to adapt to the needs of the new application. All changes are depicted in Figure A.1 in Appendix A.

5.3.1 File Menu Changes

Considering the limited number of components supported by ADDIE (a total of 8), it was determined that incorporating them as buttons in the File Menu area would enhance the convenience of adding them to the canvas. This approach involved utilizing conventional React buttons, each embedded with SVG to replicate the shape of corresponding components. The SVG drawing process involved employing the same functions and data utilized to draw each component in the canvas, as done in `SymbolView.fs`

5.3.2 Graph Area

The final deliverable encompasses both AC and Time Domain Analyses, both of which necessitate a graphical representation of the results. To accommodate this requirement, an area was added at the bottom side of the application, initially hidden from view and with a height equal to 40% of the screen, which serves as the designated space for displaying the graph visualizations.

5.3.3 Sliders

In compliance with requirement **E1.6**, a new field was introduced in the Properties tab by leveraging the pre-built Fulma Slider extension [23]. This extension proved instrumental as it supplied all the required parameters and enabled subscription to the `OnChange` event, streamlining the process of updating the `CanvasState` and initiating the re-execution of the simulation

As already stated, all three changes (File Menu, Graph area, Sliders) are shown in Figure A.2 in Appendix A which demonstrates the updated UI.

5.4 Incorporation of JS Libraries

Right from the project's inception, one of the most significant challenges encountered was identifying and integrating suitable libraries capable of executing operations that were not supported by Fable. Specifically, the project required libraries for complex number and matrix operations as well as graph plotting. It became apparent early on that the chosen math library would need to be manually incorporated, following the procedure outlined in section 2.3.2 for external JS libraries. On the other hand, integrating the graph plotting library turned out to be relatively simpler as it was discovered that one of the suggested libraries (PlotlyJS) had already been supported by Fable, thanks to the provided bindings by the Fable community [22].

5.4.1 MathJS

After considering all the possible math libraries analyzed in section 2.4.1, MathJS [33] was selected as it offers a wider range of features compared to the other two libraries, and also has more detailed documentation and is better maintained. Two methods were considered for the incorporation of the library; the `ts2fable` [50] transpiler and the manual creation of the necessary bindings. The `ts2fable` [50] transpiler is a tool developed by the Fable community which parses TypeScript declaration files and automatically generates the required F# interface. The result produced by the tool had several errors, which could not be fully resolved by manual inspection. For this reason, it was decided to identify which features were necessary and write the bindings manually. The list of selected features resulted in the following MathJS type-safe F# implementation:

```
type MathsJS =
    abstract complex : float * float -> obj
    abstract re: obj -> float
    abstract im: obj -> float
    abstract reshape : ResizeArray<obj> * ResizeArray<int> -> obj
    abstract inv : obj -> obj
    abstract multiply : obj*obj -> ResizeArray<obj>
    abstract divide : obj*obj -> obj
    abstract det : obj -> float
```

Listing 5.1: MathJS function bindings

The first three functions are used to compensate for the fact that Fable does not support complex numbers. The first function, `complex`, converts the two float parameters, which represent the real and the imaginary part, to a complex number in the JavaScript ecosystem. This is the reason behind the `obj` type the function returns, it is a JavaScript object which cannot be manipulated further. The other two functions, `re` and `im` are used to extract the real and imaginary parts of a complex JavaScript `obj`. Therefore, to convert from the created F# `ComplexC` record type (`ComplexC = { Re: float; Im:  float}`) to the JavaScript complex `obj` and back, two helper functions were written:

1. `toComplexJS : ComplexC -> obj`

2. `toComplexF : obj -> ComplexC`

Another problem encountered was the necessity to use 2D arrays (Matrices) which Fable does not support. However, despite not supporting 2D Arrays, it is allowed to have an Array of Arrays, which is how matrices were represented in F# . To convert to mathJS, the following algorithm was used:

$$\texttt{(ComplexC array array)} \xrightarrow{\text{flatten}} \texttt{(ComplexC array)} \xrightarrow[\text{map}]{\text{toComplexJS}} \texttt{(obj array)} \xrightarrow{\text{flattenedToMatrix}} \texttt{obj}$$

where:

- `flatten : (ComplexC array array) -> (ComplexC array)`: Flattens the matrix by collecting the rows
- `flattenedToMatrix : (obj array) -> (obj)`: Given that the matrix is guaranteed to be square by construction, the function first checks that the length of the array is a perfect square. Then, the square root of the array's length is found (let it be `sqrt`), and the `reshape` MathJS function is called with inputs the `flattenedMatrix` and the Array `[|sqrt,sqrt|]` which specifies the sizes of the reshaped matrix.

All of the above functions are wrapped inside one general function which solves the MNA equations, `safeSolveMatrixVecComplex`, which has two inputs: (i) the flattened matrix and (ii) the vector **b**, and performs the following operations:

1. Convert to a matrix, using `flattenedToMatrix`
2. Find its determinant, using the MathJS `det` function
3. If the determinant is 0, return `None`, else invert the matrix and continue
4. If the dimension of the matrix does not match with the length of the vector b, return `None`, else perform the multiplication of the inverted matrix with vector b and return the solutions (as an `Option`).

Algorithm 1 `safeSolveMatrixVecComplex`

```
matrix ← flattenedToMatrix flattened
det ← Maths.det(matrix)
if det = 0.0 then
    Result ← None
else
    dim ← flattened >> Array.length >> float >> sqrt >> int
    invM ← Maths.inv(matrix)
    if dim = Array.length vec then
        Result ← Maths.multiply (invM, vecB) >> toComplexF
    else
        Result ← None
    end if
end if
```

5.4.2 Plotly

Plotly stands out as an exceptionally versatile graphing library, boasting an extensive array of features that surpass the project's specific requirements. Nonetheless, its capabilities extend more than enough to fulfill the necessary tasks of this project. These tasks encompass scatter plots, the ability to incorporate multiple plots within a single graph, support for multiple axes, and even logarithmic axes. It's worth noting that Plotly encompasses a plethora of features, which results in its substantial size when included in the project. However, the plotly.js team offers a range of reduced functionality distributions that can be aliased to significantly reduce the bundle size [45].

5.5 Parsing the Circuit

Prior to initiating the simulation algorithms, it is crucial to parse the circuit to acquire a representation that serves as the foundation for the simulation. Additionally, this step involves inspecting the circuit for potential errors that could lead to unforeseen behavior during the simulation. In this case, the chosen representation is a list of nodes and the components connected to each node. This decision stems from the fact that the primary simulation algorithm, Modified Nodal Analysis, relies on identifying nodes and calculating the conductance sum of components connected between or to nodes. This section delves into the pre-processing procedures, the node identification algorithm, and the error-checking algorithm, elucidating their significance in the simulation process.

5.5.1 Necessary Modifications

An essential pre-processing step involves consolidating all ground components into a single main ground. This ensures that the node identification algorithm functions reliably, regardless of the number of ground symbols present in the circuit. This straightforward task involves two steps: (i) selecting a random ground component in the circuit (will serve as the main ground), and (ii) traversing through the list of connections and updating any connection that starts or ends at a ground symbol to instead start or end at the chosen main ground.

5.5.2 Identify Nodes

The identification of nodes stands as the pivotal pre-simulation step, as all subsequent simulation algorithms heavily rely on it. This process employs a modified Breadth-First Search (BFS) algorithm, commencing from the main ground of the circuit, which is guaranteed to be one, after the modification discussed in 5.5.1. Due to the unique structure of an analog circuit, certain adaptations are necessary for BFS. Notably, all components' ends must be interconnected, eliminating any isolated endpoints, and the components connected to the first port do not comprise the entire node. For instance, in the circuit provided, only R1 is directly connected to the ground (starting point), while both R2 and R3 (linked to the bottom port of R1) and R4 (linked to the bottom port of R3) form part of the 0^{th} node.

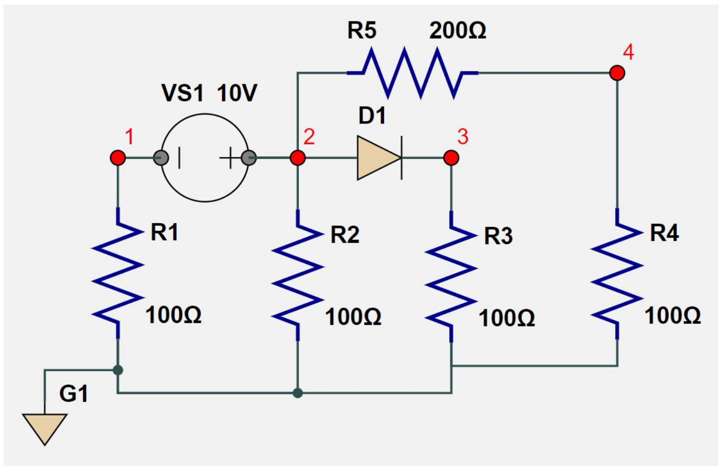


Figure 5.1: Nodes on Canvas

The search process is implemented using two mutually recursive functions, as analyzed in Algorithms 2 and 3, initially called by: `searchCurrentCanvasState [] [ground.IOPorts[0]] [[]]`

Algorithm 2 `searchCurrentNode`

```
    Inputs (visited: Port list), (toVisitLocal: Port list), (localList: (Component*int option) list)
    Output (LocalList: (Component*int option) list)
  if toVisitLocal = [ ] then
      return (localList,visited)                                                    ▷ Exit
  else
      currentPort ← toVisitLocal[0]
      connectedToCurrentPort ← $fun()$         ▷ ports connected to the current port (not visited)
      visited' ← visited @ [toVisitLocal[0]]
      toVisit' ← toVisitLocal[1..-1] @ connectedToCurrentPort
      localList' ← localList @ connectedToCurrentPort                      ▷ the components
       return (searchCurrentNode visited' toVisit' localList')                  ▷ Recursion
  end if
```

Algorithm 3 `searchCanvasState`

```
    Inputs (visited: Port list), (toVisit: Port list), (nodeList: (Component*int option) list list)
    Output (nodeList: (Component*int option) list list)
  if toVisit = [ ] then
      return nodeList                                                               ▷ Exit
  else
      toVisitFirstComp ← find comp.Id = toVisit[0].HostId
      newNodeList, visited' ← searchCurrentNode visited [toVisit[0]] []
      diff ← visited' - visited                                            ▷ Set Difference
      oppositePorts ← findOtherEndPort diff
      toVisit' ← toVisit[1..-1] @ oppositePorts
      nodeList' ← nodeList @ newNodeList @ toVisitFirstComp
       return (searchCanvasState visited' toVisit' nodeList')                   ▷ Recursion
  end if
```

On-Canvas Locations of Nodes

Considering that simulation results are obtained based on the nodes derived from the function mentioned earlier, it becomes highly important to provide users with a clear visualization of the nodes' locations on the circuit. This visual representation enables users to easily identify the precise location of each node. For instance, when a result like V(Node 4) = 2.5V is displayed on the 'Simulations' tab, it becomes much simpler to comprehend when the location of Node 4 is clearly visible on the canvas, as in Figure 5.1, compared to a purely textual representation, where a node's location is described by the list of components connected to it. Visualizing the node locations is especially valuable when configuring AC and Time simulations, as users need to accurately select the desired node for running the simulation.

Before inspecting the algorithm employed to find the nodes' locations on the canvas, it is crucial to analyze the `Connection` type used to represent wires. As seen in Listing 5.2, each wire is represented by: (i) a unique **Id**, (ii) a **Source** Port, (iii) a **Target** Port, and (iv) a list of **Vertices**. Vertices represent the wire segments and start by default with 7 vertices where: (i) the first and last segments produced are small 'sticks' connected to the ports, to avoid weird wiring around them, and (ii) the remaining are used for the actual wire. This means that for specific wires (e.g. straight wire), there are duplicates in the Vertices list. The procedure to find the locations is analyzed in Algorithm 4.

```
type Connection = {
        Id : string
        Source : Port
        Target : Port
        Vertices : (float * float * bool) list
    }

type Port = {
        Id : string
        PortNumber : int option
        HostId : string
    }
```

Listing 5.2: Connection and Port Types

Algorithm 4 `findNodeLocation`

```
    Input (connsOnNode:Connection list)
    Output (pos: XYPos list)
  switch List.length connsOnNode do
      case 0
          failwithf "Impossible"
      case 1
          pos ← connsOnNode[0].Vertices >> removeDuplicates >> removeSticks >> findMid
      case _
          posList ← vertices >> removeDuplicates >> removeSticks >> countBy id
          if posList[0].count != 1 then
              pos ← posList[0].pos
          else
              pos ← connsOnNode[0].Vertices >> removeDuplicates >> removeSticks >> findMid
          end if
```

5.5.3 Error Checking

During this step, the CanvasState and NodeList are carefully examined to detect common error scenarios that could break the simulation algorithms. The output of this process is utilized to generate informative error messages for users, pinpointing the relevant components and/or connections associated with the identified issues. By proactively identifying and highlighting these errors, users are equipped with valuable information to address and rectify any errors that would obstruct the smooth execution of the simulation. Analytically, the function checks for:

1. Absence of ground
2. Ports and/or components which are not connected to the main circuit
3. No in-parallel Voltage Sources
4. No in-series Current Sources
5. No Short Circuits
6. No Double connections (two wires overlapping)
7. Time domain simulation errors (only prior to time simulation):
    (a) More than one or zero Voltage Sources
    (b) More than one Capacitor or Inductor

The time-domain-specific errors are related to the limitations of the algorithm for time-domain analysis. More on this follows in section 5.8

5.6 DC Operating Point Analysis

DC Operating Point Analysis stands as the most commonly employed simulation technique due to its fundamental role as a prerequisite for other simulations. The algorithm chosen to perform DC analysis is Modified Nodal Analysis (MNA), which serves as an extension of the classical nodal analysis. The utilization of MNA offers numerous advantages, which are thoroughly outlined in section 2.5.1. Furthermore, the methodology of MNA is also described in detail in section 2.5.1. This section primarily concentrates on the implementation of MNA for attaining the DC solution of the circuit, describing in detail the procedures to handle reactive components (Capacitors and Inductors), Operational Amplifiers, and linearized diodes.

5.6.1 Handling Capacitors and Inductors

As per the principles of Analogue Electronics, it is well-established that in DC analysis, capacitors are regarded as open circuits, while inductors act as short circuits. When extending the Modified Nodal Analysis (MNA) algorithm to accommodate reactive components, we initially consider the complex impedance of capacitors and inductors, denoted as $Z_C = \frac{1}{j\omega C}$ and $Z_L = j\omega L$ respectively. This implies that within the MNA's G matrix, the terms corresponding to capacitors become $j\omega C$, while those for inductors are $\frac{1}{j\omega L}$. Consequently, for DC analysis (where $\omega = 0$), these values simplify to 0 and ∞ respectively.

Regarding capacitors, it becomes apparent that they do not significantly influence the MNA algorithm due to their conductance being 0. Consequently, dealing with capacitors proves to be straightforward as they are treated as having a value of 0 within the calculations for the G matrix.

On the other hand, however, inductors require special attention, as their conductance in DC ($\omega = 0$) is ∞. An ∞ term in the matrix will inevitably break the simulation and not produce any results. To overcome this issue, prior to simulation each inductor component is locally transformed to a short circuit, by altering its type from `(Inductor _)` to `(VoltageSource (DC 0.0))`. By implementing this transformation on the CanvasState, the MNA algorithm can operate seamlessly without requiring any further modifications. Moreover, this technique enables the calculation of the current flowing through the inductor, which will be part of the MNA solution.

5.6.2 Modified Nodal Analysis Implementation

The implementation of the Modified Nodal Analysis algorithm involves four key steps, outlined below. This procedure closely follows the process described in section 2.5.1 of the Background chapter. It is important to note that the terminology employed in this section, including matrix names, assumes familiarity with the Modified Nodal Analysis terminology detailed in section 2.5.1.

1. Transformation of CanvasState:
   (a) One common ground (as per 5.5.1)
   (b) Capacitors/Inductors (as per 5.6.1)
   (c) Diodes (see below, 5.6.3)
2. Matrix **A** formulation
3. Vector **b** formulation
4. Solution Extraction

Matrix A Formulation

Once the transformed CanvasState has been obtained, the Modified Nodal Analysis (MNA) algorithm begins the process of formulating the system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ represents the vector of unknowns. In accordance with the established procedure, the **A** matrix is divided into four sub-matrices:

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

The formulation of the matrix is then completed by the main function `calcMatrixElement` (Algorithm 5) which uses 3 helper functions, one for each matrix (G, B, and C), analyzed in Algorithms 6, 7, 8. Initially, a square $S \times S$ zero matrix is created, with S equals the number of nodes plus the number of voltage sources and opamps. Then, each matrix element is updated using the `calcMatrixElement` function that derives the correct value.

Vector b Formulation

Vector **b** is the vector of known quantities of the circuit and is formed by the set of independent Voltage and Current Sources. The calculation of its values for a circuit with n nodes and m voltage sources is as follows: (i) the first n elements are the sum of current flowing into each node, and (ii) the remaining m elements are simply the DC value of each voltage source. Analytically:

$$\mathbf{b}^T = [I(\text{Node } 1),\ \ldots,\ I(\text{Node } n),\ V(VS_1),\ \ldots,\ V(VS_m)]$$

The code follows an identical procedure as for the calculation of matrix **G** (Algorithms 5 and 6), summing only current or voltage values.

Solution Extraction

Once matrix **A** and vector **b** have been obtained, the next step is to invoke the `safeSolveMatrixVecComplex` function, which will solve the linear system and provide the solution. This solving function is described in detail in section 5.4.1 and can be found in algorithm 1.

Algorithm 5 `calcMatrixElement`

```
  Inputs row, col, nodeList, omega
  Output (value: ComplexC)
if row < nodesNo && col < nodesNo then                                          ▷ Matrix G
    if row == col then
        value ← ({Re=0;Im=0}, nodeList[row])|| > fold (matrixGCompValue comp omega)
    else
        value ← ({Re=0;Im=0}, (compsBetween row col))|| > fold (matrixGValue comp omega)
    end if
else
    if row ≥ nodesNo && col ≥ nodesNo then
        value ← {Re=0;Im=0}                                             ▷ matrix D is always 0
    else
        if row < nodesNo then                                                   ▷ Matrix B
            value ← ({Re=0;Im=0}, nodeList[row])|| > fold (matrixBValue comp col)
        else                                                                    ▷ Matrix C
            value ← ({Re=0;Im=0}, nodeList[col])|| > fold (matrixCValue comp row)
        end if
    end if
end if
```

Algorithm 6 `findMatrixGCompValue`

```
  Inputs (comp,portNo), omega
  Output (value: ComplexC)
switch comp.Type do
    case Resistor (r,_)
        value ← $\{\mathrm{Re} = (1/r); \mathrm{Im} = 0\}$
    case Capacitor (c,_)
        value ← $\{\mathrm{Re} = 0; \mathrm{Im} = c * omega\}$
    case Inductor (l,_)
        value ← $\{\mathrm{Re} = 0; \mathrm{Im} = (-1/(l * omega))\}$
    case _
        value ← $\{\mathrm{Re} = 0; \mathrm{Im} = 0\}$
```

Algorithm 7 `findMatrixBValue`

```
    Inputs (comp,portNo), (vsAndOpamps: Component list), col, nodesNo
    Output (value: ComplexC)
switch comp.Type do
    case VoltageSource _ || Opamp
        if comp.Id = vsAndOpamps[col-nodesNo].Id then          ▷ match column with vector B row
            switch (comp.Type,portNo) do
                case (VoltageSource _, Some 1)
                    value ← {Re = −1; Im = 0}                      ▷ Negative terminal of VS
                case (VoltageSource _, Some 0)
                    value ← {Re = 1; Im = 0}                       ▷ Positive terminal of VS
                case (Opamp, Some 2)
                    value ← {Re = 1; Im = 0}                       ▷ Output port of Opamp
                case (Opamp, _)
                    value ← {Re = 0; Im = 0}
        else
            value ← {Re = 0; Im = 0}
        end if
    case _
        value ← {Re = 0; Im = 0}
```

Algorithm 8 `findMatrixCValue`

```
    Inputs (comp,portNo), (vsAndOpamps: Component list), row, nodesNo
    Output (value: ComplexC)
switch comp.Type do
    case VoltageSource _ || Opamp
        if comp.Id = vsAndOpamps[row-nodesNo].Id then          ▷ match row with vector B row
            switch (comp.Type,portNo) do
                case (VoltageSource _, Some 1)
                    value ← {Re = −1; Im = 0}                      ▷ Negative terminal of VS
                case (VoltageSource _, Some 0)
                    value ← {Re = 1; Im = 0}                       ▷ Positive terminal of VS
                case (Opamp, Some 0)
                    value ← {Re = 1; Im = 0}                       ▷ Positive Input of Opamp
                case (Opamp, Some 1)
                    value ← {Re = −1; Im = 0}                      ▷ Negative input of Opamp
                case (Opamp, _)
                    value ← {Re = 0; Im = 0}
        else
            value ← {Re = 0; Im = 0}
        end if
    case _
        value ← {Re = 0; Im = 0}
```

Region	Condition	Equation
Conducting Mode ("on")	$I > 0$	$V = 0.7$
Non-conducting Mode ("off")	$V < 0.7$	$I = 0$

Table 5.4: Linearized Real Diode Modes

5.6.3 Handling Linearized Diodes

As already known from Analogue Electronics principles, diodes have two operating modes: conducting and non-conducting mode. Thinking of ideal diodes for simplicity, when the voltage across the diode is negative, no current can flow, and the ideal diode behaves like an open circuit (non-conducting/reverse biased/off mode). On the other hand, if the voltage is not negative, it conducts current, behaving like a short circuit with a 0V voltage drop (conducting/forward biased/on mode). The diode supported by this simulator is neither an ideal diode, nor a real diode (which has the characteristics defined in section 2.6.2), but a linearized equivalent of a real diode with the characteristics defined in Table 5.4. The choice to support the specific linearized diode model is based on the fact that this is the diode model used in the ADC module of Imperial, along with the significant extra time that would be required to include non-linear components in the simulation algorithms as discussed in section 4.3.

The algorithm employed to identify the mode of each diode is identical to the on-paper algorithm, and consists of:

1. Make a guess about the operating mode of the diode
2. Using that mode, solve the circuit
3. If the condition is met, the guess was correct and the correct solution has been obtained. Otherwise, the correct mode is guaranteed to be the not-selected mode.

For a circuit with a single diode, for example, the algorithm will initially transform locally the diode to a `VoltageSource (DC 0.7)`. Then, MNA will run (no diode present, run as before) and the condition will be checked ($I > 0$). If the condition is satisfied, the `transformDiodes` function returns the transformed circuit (with the `VoltageSource (DC 0.7)`). However, if the condition is not met, the diode is transformed into a `CurrentSource (0.0)`. This means that the `ModifiedNodalAnalysis` and `transformDiodes` functions are mutually recursive, as for the latter to produce a correctly transformed CanvasState, the former must be used to check the conditions.

When dealing with circuits containing multiple diodes, simply checking each diode individually is insufficient to guarantee an accurate solution. This is because the mode of one diode may depend on the state of another diode. To address this challenge, diode circuits are solved using an exhaustive search approach. In other words, all possible combinations of diode modes are checked, resulting in a total of 2^n combinations for a circuit with n diodes.

To enhance performance, the algorithm stores the diode modes that meet the necessary conditions along with the MNA results. In subsequent runs of the algorithm, the stored modes are checked first, leading to a significant overall performance improvement. This optimization is particularly beneficial in time domain analysis, where the MNA algorithm needs to be executed for ≈ 200 time steps in a single simulation.

The diode modes are represented as a list of booleans (one per diode), where true stands for conducting mode and false for non-conducting mode. The initial guess is that all diodes are in conducting mode, which is equivalent to an all-true list. Then the procedure used to find the next modes is the following:

$$\text{modes (bool list)} \xrightarrow[\text{fold}]{\text{toString}} \text{“11111”} \xrightarrow{int} 31 \xrightarrow{(-1)} 30 \xrightarrow[\text{base2}]{\text{toString}} \text{“11110”} \xrightarrow{\text{toBoolList}} \textbf{new} \text{ modes (bool list)}$$

The top-level function (`transformDiodes`) is analysed in algorithm 9. It is worth noting that the (`transformDiodes`) function is the only new[1] function that uses mutable data. Because of the nature of this function, the use of mutable data makes the code clearer and faster. Given the local use of the mutable data (only for the while loop), it is guaranteed that no issues will arise from their use.

Algorithm 9 `transformDiodes`

```
    Inputs (comps,conns), cachedDiodeModes
    Outputs (comps',conns'), newDiodeModes
if cachedDiodeModes = [ ] then
    diodeModes ← Array.create diodesNo true
else
    diodeModes ← cacheDiodeModes
end if
allConditions ← false
iter ← $2^{\text{diodesNo}}$
while not allConditions do
    comps' ← (runTransformation diodeModes comps)
    res ← (modifiedNodalAnalysis comps' conns)
    allConditions ← checkConditions diodeModes res
    if iter = 0 then                                        ▷ Force Stop
        allConditions ← true
    end if
    if allConditions then                                   ▷ Exit
        return (comps',conns,diodeModes)
    end if
    diodeModes ← nextModes
end while
```

5.6.4 Calculating Currents through Components

The currents flowing through Voltage Sources and Opamps are already calculated by Modified Nodal Analysis, therefore no extra calculation is needed. For resistors, the current is calculated by finding the voltage difference and dividing it by the value of the resistance. Given that currents will eventually be visualized on canvas, it is important to assure that the current has the right sign and direction. The `calculateComponentCurrents` accepts an extra parameter, which is the `Rotation` of each component. In the end, the returned `Map<ComponentId,float>` that maps components to currents, assumes that current flows from left to right, or from top to bottom. This means that if the value in the Map is negative, the direction is either right to left or bottom to top.

5.6.5 Visualisation Of Results

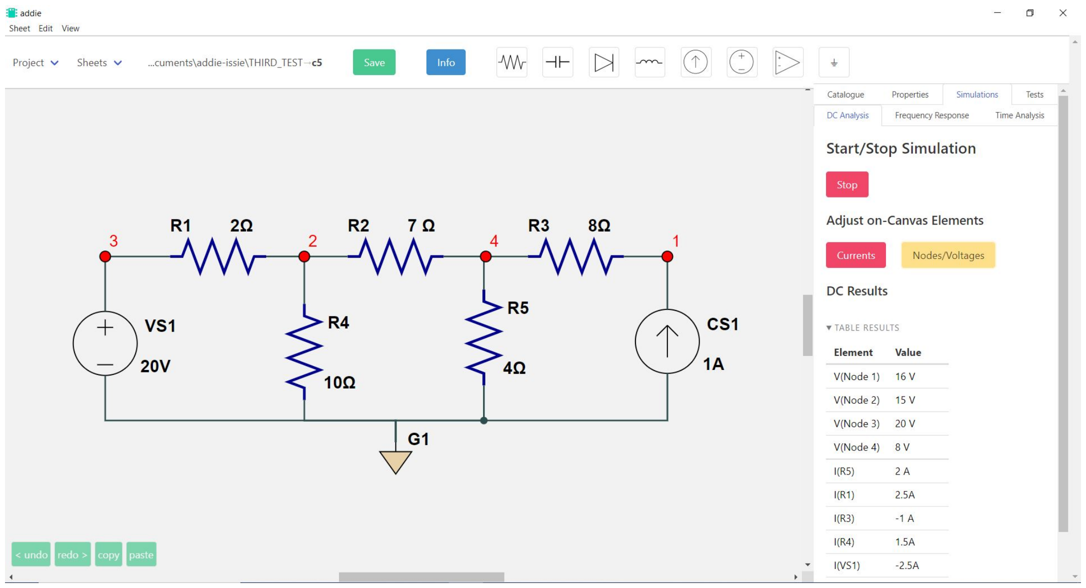
The DC Operating Point Analysis results are visualized using three different means: (i) DC Table, (ii) DC Equations, and (iii) On-canvas results.

The DC Table is a table in the RHS pane which appears below the 'Start' simulation button and summarizes all the node voltages and component currents of the simulated circuits. The DC Equations are a set of equations that formulate the solution. These are created by identifying voltage dividers, short-circuits (inductors), and nodes with constant voltages (e.g. nodes connected at the ends of voltage sources).

The on-canvas results are visualizations of the DC Table values on the canvas. These split into two categories: (i) Nodes/Voltages and (ii) Currents. Nodes are represented on the circuit as red dots (to differentiate from ports) and the user can select whether the node number (Figure 5.2) or the node voltage (Figure 5.3) will be displayed. The node locations are calculated using the function analyzed

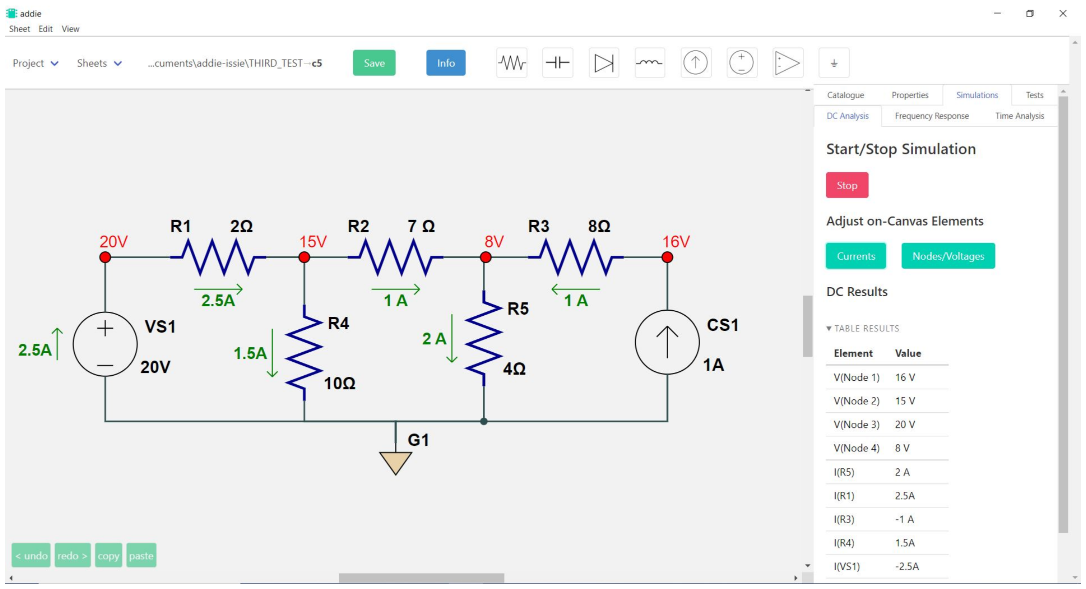
[1]written as part of this project. ISSIE already uses mutable variables in some functions.

in section 5.5.2. The currents flowing through components are represented using arrows drawn next to components. The absolute value of the `ComponentCurrents Map` is drawn next to the arrows. This means that the arrow will always have the direction of the current flow. Following from the preprocessing done in the current calculation function in section 5.6.4 (current flow is assumed to be right-to-left or top-to-bottom), it is straightforward to find the direction of the current arrow based on the sign of the current value.



Element	Value
V(Node 1)	16 V
V(Node 2)	15 V
V(Node 3)	20 V
V(Node 4)	8 V
I(R5)	2 A
I(R1)	2.5A
I(R3)	-1 A
I(R4)	1.5A
I(VS1)	-2.5A

Figure 5.2: DC Simulation Visualizing the Node Locations on the Canvas



Element	Value
V(Node 1)	16 V
V(Node 2)	15 V
V(Node 3)	20 V
V(Node 4)	8 V
I(R5)	2 A
I(R1)	2.5A
I(R3)	-1 A
I(R4)	1.5A
I(VS1)	-2.5A

Figure 5.3: DC Simulation Visualizing the Node Voltages and Branch Current on the Canvas

5.7 AC Analysis

The AC Analysis supported by ADDIE, allows users to select an input Voltage Source and an output node, and view the Magnitude and Phase of the ratio $\frac{\mathrm{V(Node)}}{\mathrm{V(VS)}}$ for a wide range of frequencies. This section analyses the algorithms used for Frequency Response analysis and the way the results are visualized.

Before analyzing the circuit with respect to frequency, there are two necessary modifications that take place, apart from the combination of grounds discussed in section 5.5.1. Initially, the selected input source is converted to a source of Magnitude 1 and Phase 0 (for the purposes of the simulation this is a (DC 1.0) source). Then, all the remaining sources, including the diodes, are converted to (DC 0.0) sources (considered as short circuits). This allows the reusing of the existing code from DC Operating Point Analysis and assures the resulting graph represents the ratio $\frac{\text{V(Node)}}{\text{V(VS)}}$.

After performing the transformations discussed above, the algorithm simply maps a pre-defined list of frequencies (20 points per decade, from 1 rad/s to 10.000.000 rads/s) to a list of complex numbers represented as `ComplexP`, which represent the Magnitude and Phase at each frequency point. The mapping function takes each frequency value and runs MNA on the transformed circuit using this frequency. All the algorithms involved in the creation of the MNA matrix had one extra parameter, `wmega`, which is now used to include the impedance of capacitors and inductors. Using the `MathJS` library, the complex matrix is inverted, and then gets multiplied with the vector **b**, as in DC Analysis. The mapping function then returns the required value which is simply the complex voltage value of the specified - by the user - output node.

5.7.1 High Precision AC Analysis

High Precision AC Analysis is implemented using the algorithm derived in section 4.5 by increasing the frequency points by a factor of 5 around the corner frequencies (100 points per decade). Users can trigger high-precision analysis by selecting the relevant tickbox in the 'Frequency Response' sub-tab before pressing the 'Show' button. The decision to use 100 points per decade around the corner frequencies strikes a balance between precision and simulation speed. This choice offers a sufficiently accurate plot for student experiments while maintaining a fast and responsive simulation experience.

5.7.2 Visualisation Of Results

Simulation results are displayed in the graph area using the Plotly library. The graph includes both the magnitude and the phase plots, which have the same x-axis (frequency), using two different y-axes: one on the right for magnitude values and one on the left for phase values. Phase is by default set to degrees. The Frequency and Magnitude units are configurable from buttons in the ACSim sub-tab and can be Hz or rads/s (for frequency) and dB or non-dB (for magnitude).

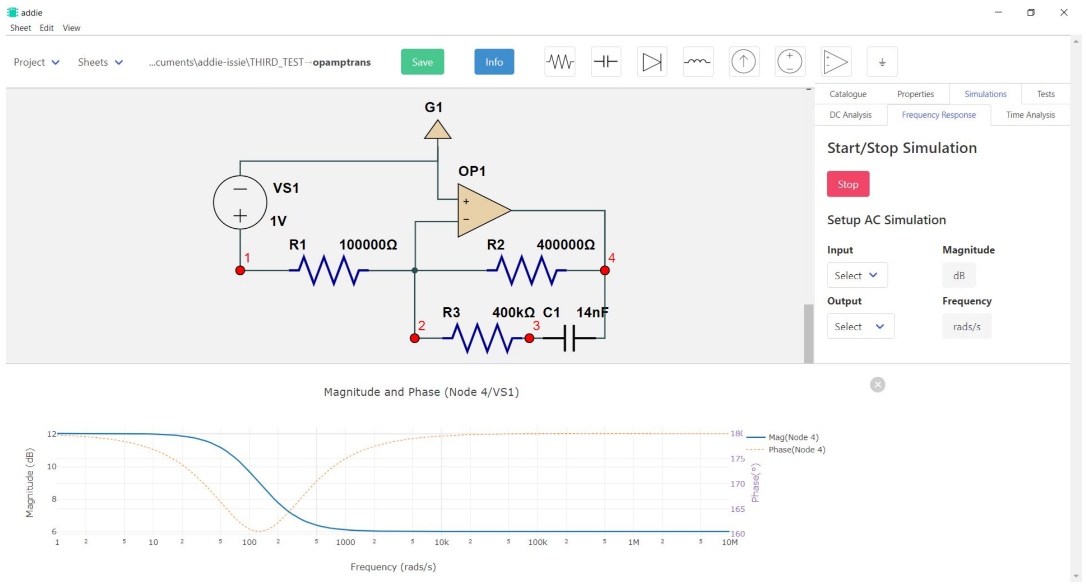


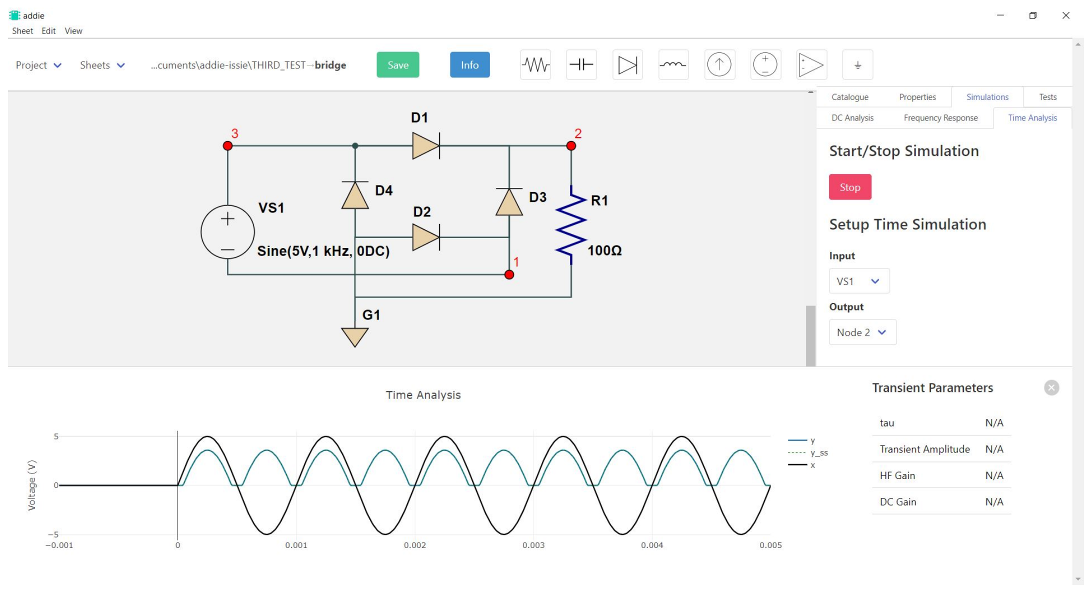
Figure 5.4: Frequency Response Simulation Results

5.8 Time Analysis

Time Simulation in ADDIE enables users to observe the dynamic behavior of their circuits over time. Due to its inherent complexity, ADDIE supports two specific types of circuits for time simulation: circuits without capacitors or inductors, and first-order circuits (one capacitor or inductor). Although this imposes a limitation, it is important to note that ADDIE still functions effectively for circuits relevant to ADC (Analysis and Design of Circuits) material, where the emphasis is primarily on first-order circuits. Thus, despite this restriction, ADDIE can deliver the expected functionality for the circuits typically encountered in the module.

5.8.1 Circuits without Capacitors or Inductors

When dealing with circuits that are comprised of resistors, diodes, operational amplifiers, and voltage/current sources only, the simulation process is relatively straightforward. It involves performing a DC Analysis at multiple time points, resulting in a time-domain simulation. The algorithm employed follows these steps: firstly, the values of sine wave voltage sources are computed for each time point. Subsequently, modified nodal analysis is executed (as described above) to analyze the circuit's behavior at each specific time. By iterating through these calculations, the simulation accurately captures the circuit's response over time.



Input: VS1

Output: Node 2

Transient Parameters

tau	N/A
Transient Amplitude	N/A
HF Gain	N/A
DC Gain	N/A

Figure 5.5: Time Domain Simulation of Circuit with no Reactive Components

5.8.2 Circuits with one Capacitor or Inductor

Simulating first-order circuits requires a distinct algorithm that differs from the one used for circuits without reactive components. These circuits involve additional calculations specific to their characteristics. The algorithm employed for simulating first-order circuits follows the approach outlined in the analysis of such circuits as described in the ADC module. This specialized algorithm ensures accurate simulation results tailored to the unique properties of first-order circuits, and is summarized below:

- Output is always steady-state + transient, where:
  - Steady-state: same frequency as the input; new amplitude and phase offset are determined using frequency response
  - Transient is always of the form $Ae^{\frac{-t}{\tau}}$
- τ is equal to $R_{th} * C$ or $\frac{L}{R_{th}}$ where R_{th} is the Thevenin resistance seen by C or L

- To find A we solve: $\Delta output = HFgain * \Delta input \implies y_{ss}(0^+) + A = HF * x(0^+)$, as $x(0^-)$ and $y(0^-)$ are force-set to 0

(I) Finding the Thévenin Resistance

According to the Thévenin Theorem: Any two-terminal network consisting of resistors, fixed voltage/current sources and linear dependent sources is externally equivalent to a circuit consisting of a resistor in series with a fixed voltage source. The aim for the purposes of time simulation is to find the Thévenin resistance (R_{th}) seen by the capacitor or the inductor.

A Thévenin equivalent circuit has a straight-line characteristic with the equation $V = R_{th}I + V_{th}$. Given that an implementation of nodal analysis already exists (MNA), the algorithm here finds two points in the Thévenin equivalent circuit, from which it will obtain the Thévenin resistance R_{th} by finding the slope of the line. This is done by substituting the relevant component with a current source I, and obtaining the results for two different current values, $I_1 = 1A$ and $I_2 = 2A$. Performing MNA on these two circuits will produce two different voltage values, V_1 and V_2. Then the Thévenin resistance is equal to:

$$R_{th} = \frac{V_2 - V_1}{I_2 - I_1} = V_2 - V_1, \text{ since } I_2 - I_1 = 1$$

(II) Finding the Steady-State

For the purposes of time simulation, the steady state is stored in the simulation function as a voltage source, which can be then used to find its value at different time points. It is obtained by performing MNA for DC/Step input sources, and AC analysis for sinusoidal inputs. For the former case, y_{ss} is simply a Voltage Source with a DC value equal to the voltage of the output node calculated by MNA. For the latter, y_{ss} is set as a sinusoidal voltage source, with the same frequency f as the input source, and new amplitude $A' = \text{ACAnalysis.Mag} \times A$ and new phase offset $PO' = \text{ACAnalysis.Phase} + PO$.

(III) Finding the High Frequency (HF) Gain

At high frequencies, as the frequency approaches infinity ($f \to \infty$), the impedance of a capacitor becomes approximately zero, while the impedance of an inductor tends to infinity. This behavior arises from the impedance formulas for a capacitor and an inductor ($Z_C = \frac{1}{j\omega C}$ and $Z_L = \frac{1}{j\omega L}$) as ω approaches infinity. To ensure that this behavior is accurately represented in the circuit simulation, the capacitor or inductor is locally transformed into a 0-current or 0-voltage source, respectively. Subsequently, the Modified Nodal Analysis (MNA) is performed on the transformed circuit to calculate the high-frequency gain ($HFgain = \frac{V(\text{output node})}{V(\text{input node})}$). This approach enables the simulation to capture the appropriate behavior of capacitors and inductors at high frequencies.

(IV) Finding the Amplitude (A) of the transient

This is now straightforward as all the necessary variables for the calculation of A have been found. Therefore, using the formula described above, A is calculated as $A = HF * x(0^+) - y_{ss}(0^+)$.

(V) Finding the timesteps

ADDIE does not allow users to set their own start/end time and timesteps, but provides a stable algorithm that covers all the areas of interest that a student might want to investigate. The calculation of timesteps depends on the type of the input source, as explained in table 5.5.

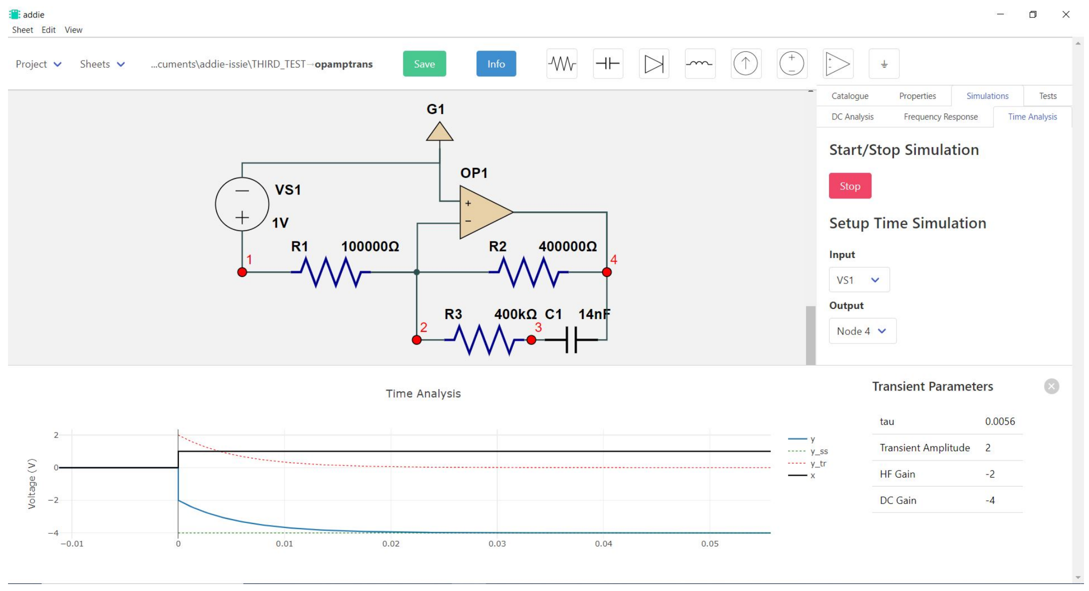
Input Source Type	Start time	End time	TimeStep
Sinusoidal of frequency f	0.0	$5/f$	$1/(40 * f)$
DC/Step with time constant τ	0.0	$10 * \tau$	$\tau/20.$

Table 5.5: Transient Analysis: TimeStep selection

Essentially, the end time is selected so that either the output has converged to the steady-state, or so that the user can see 5 full periods of a sinusoid. The time step is then selected based on the start and end times to assure that in total, there are 200 timesteps.

5.8.3 Visualisation Of Results

Combining all the information obtained above, the `transientAnalysis` function returns three lists of length 200: one containing the samples of y_{ss}, one for y_{tr}, and the timesteps used (dts). The graph displays in total 4 signals against time: (i) $\mathbf{x}$ (input signal), $\mathbf{y}$ (output signal), $\mathbf{y_{ss}}$ and $\mathbf{y_{tr}}$. Both $\mathbf{x}$ and $\mathbf{y}$ are calculated in the graph function, as $\mathbf{y}$ is simply $\mathbf{y_{ss}} + \mathbf{y_{tr}}$, and $\mathbf{x}$ can be derived directly from the circuit.



Input: VS1

Output: Node 4

Transient Parameters

tau	0.0056
Transient Amplitude	2
HF Gain	-2
DC Gain	-4

Figure 5.6: Time Domain Simulation of Circuit with one Reactive Component

Chapter 6

Testing and Results

The final deliverable of this project underwent testing from three different perspectives: correctness, performance, and user experience. To evaluate correctness, the features of ADDIE were examined to confirm that each simulation feature functioned as intended. Additionally, it was ensured that the application remained stable and did not suffer from unexpected crashes. For performance testing, quantitative analysis was performed by measuring the average time required for specific actions and processes. Lastly, the user experience quality was assessed by assigning survey participants a series of tasks within ADDIE, measuring their performance, and gathering their feedback.

6.1 Stability

For a software application to be considered stable, it must not suffer from inexplicable crashes or experience unpredictable behavior while utilizing it. The stability of ADDIE is verified by analyzing all the cases in the codebase that can cause undefined behavior or crashes.

By employing the MVU (Model-View-Update) framework in conjunction with the strongly-typed F# language, the application is already highly safeguarded against any instances of undefined behavior. This is also verified by the use of ISSIE over the past 3 years, where more than 200 students use it each year for numerous tasks, including large-scale projects. However, there are two scenarios that can cause ADDIE to crash.

The first scenario is when `failwithf` is called in the code. This function is used to handle cases that should never occur when running the application. `failwithf` enables users to print to the console a specific error message that can direct developers to the part of the code that caused the unexpected behavior and the crash. An example of this is demonstrated in listing 6.1 where the `CurrentProject` field of the model is guaranteed to not be `None` if the user is able to trigger a simulation.

```
// part of the DC Operating Point Analysis simulation

match model.CurrentProject with
|Some proj ->
    //simulation code
|None ->
    failwithf "Project cannot be None when running a simulation"
```

Listing 6.1: Example of Option used in the code

The second scenario is when trying to access elements that do not exist. One example is trying to access the 6th element of a list that has only 5 elements. To guarantee that such scenarios do not occur, F# offers safe access functions that return an `Option<T>`, which has the value `Some element` if the element exists, or `None` otherwise. Examples of such functions are `List.tryItem` which provides a safe function to obtain the value at a particular index, and `List.tryFind`: a safe alternative of the `List.find` function. Essentially, direct indexing and the `List.find` function should only be used when the required element to be obtained is guaranteed to exist.

In ADDIE, stability is guaranteed by manually inspecting all non-safe functions and checking that the element being searched is guaranteed to exist. Nevertheless, as ADDIE relies on JavaScript libraries for simulations, it is crucial to account for potential unsuccessful runs (such as when the MNA matrix cannot be inverted). In such cases, it becomes essential to handle appropriately the returned `null` value.

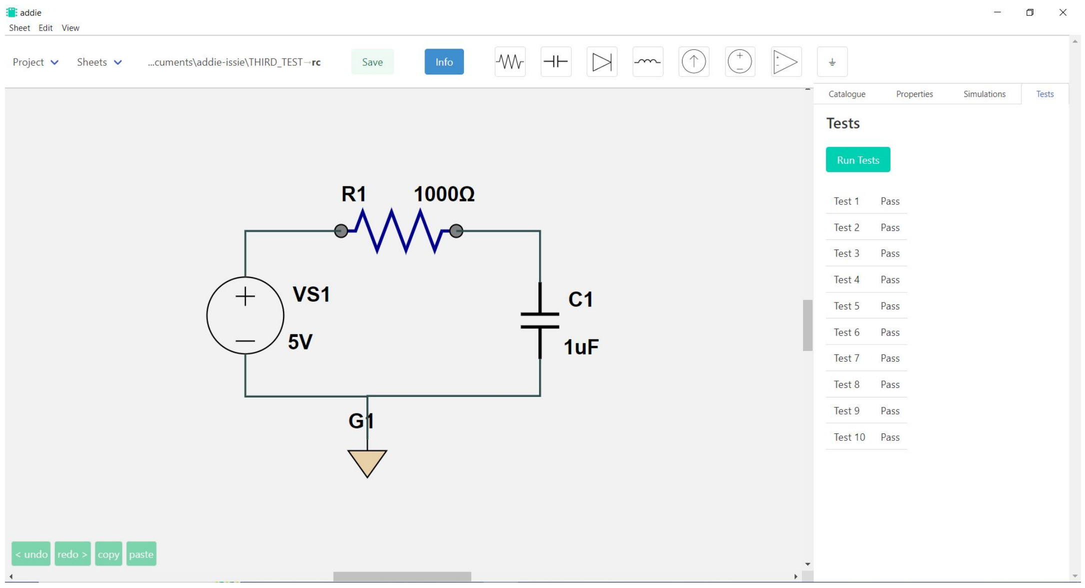
Initially, the canvas state is analyzed to identify common errors such as the absence of ground or short circuits (see section 5.5.3 - Error Checking). This function also takes into account the fact that time simulation can only run with a maximum of one voltage source and one reactive component, forcing the simulation to stop early, to avoid unexpected behavior, and it also informs the user about this limitation.

However, the tests conducted during the error-checking stage, do not guarantee that a solution can be obtained. Specifically, there is a possibility that the circuit does not have a unique solution (MNA matrix is not invertible). The existence of such matrix is checked in advance by calculating the determinant of the matrix, as shown in algorithm 1. If the determinant is 0, the function returns `None`, which is then treated accordingly by the view function to display an 'Unknown error in the circuit' error message. Otherwise, the matrix is invertible, and a solution can be obtained.

6.2 Correctness

The application's correctness is evaluated through an extensive suite of tests, which verify that the simulation performs as expected. Having tests in place is crucial for any software application for two primary reasons. Firstly, they ensure the correctness of the algorithms used in the calculation of results, which can be challenging to ascertain solely by examining the source code. Secondly, tests provide developers with the confidence to modify algorithms while ensuring that existing functionalities remain intact. This type of testing, commonly referred to as regression testing, becomes increasingly important as the codebase grows. Without it, manually ensuring the accuracy of each algorithm after every modification becomes a time-consuming task. Consequently, the application's maintainability and extensibility are greatly improved.

Since the simulation heavily relies on the 'MathJS' JavaScript library, it is not possible to execute the tests under the .NET environment. Consequently, unlike ISSIE, which utilizes Expecto [21] for testing, ADDIE's testing process needs to be carried out from within the application itself. Specifically, when ADDIE is run in development mode, an additional tab labeled 'Tests' is available alongside 'Catalogue', 'Properties', and 'Simulations'. Developers can access this tab and run the tests by clicking a designated button. After the tests are completed, a table summarizing the results, indicating which tests passed and which ones failed, will be displayed below the 'Run Tests' button. Figure 6.1 provides a visual representation of this process. The 10 tests provided cover all the supported simulation features.



Test 1	Pass
Test 2	Pass
Test 3	Pass
Test 4	Pass
Test 5	Pass
Test 6	Pass
Test 7	Pass
Test 8	Pass
Test 9	Pass
Test 10	Pass

Figure 6.1: Tab containing the supported tests

6.3 Quantitative Performance Testing

In this subsection, 6 different circuits, each with different characteristics will be used to assess the performance of the application. Different simulations will be run in each circuit, to assess all the different simulation modes and algorithms. The measurements were performed on a Dell Latitude 5400 series, which boasts an Intel Core i7 (8th gen) CPU with 16GB of RAM. The results are summarized in tables 6.1 and 6.2.

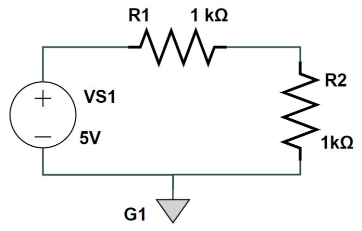
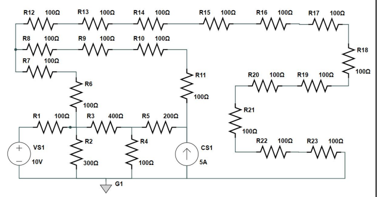
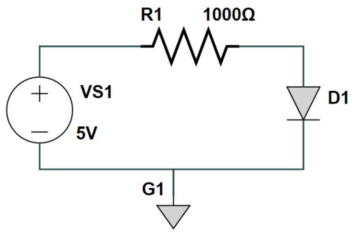
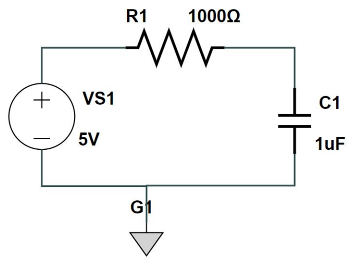
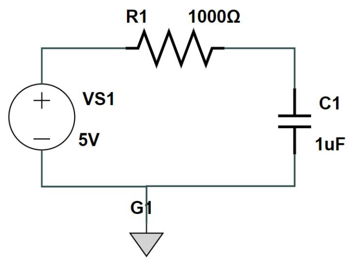
Diagram	Characteristics & Analysis	Execution Time
	This is a simple potential divider consisting of one 5V DC Source and two resistors. The **DC Simulation** execution time will be measured for this circuit, to see the execution time of MNA. The complexity-heavier subtask of MNA is the inversion of the matrix, which is $\mathcal{O}((n+m)^3)$ for n nodes and m voltage sources, and then the matrix completion and multiplication follow. The simulation is completed with an average runtime of 0.52 ms	Trial | Time (ms) 1 | 0.5 2 | 0.5 3 | 0.4 4 | 0.5 5 | 0.7 Avg | 0.52
 R12 100Ω R13 100Ω R14 100Ω R15 100Ω R16 100Ω R17 100Ω R8 100Ω R9 100Ω R10 100Ω R18 100Ω R7 100Ω R11 100Ω R20 100Ω R19 100Ω R6 100Ω R21 100Ω R1 100Ω R3 400Ω R5 200Ω VS1 10V R2 300Ω R4 100Ω CS1 5A R22 100Ω R23 100Ω G1	This is a much larger circuit consisting of 20 nodes and 1 voltage source, to explore how execution times are increased with respect to the number of nodes. Again, **DC Simulation** was investigated, completing with an average runtime of 15.7 ms	Trial | Time (ms) 1 | 19.2 2 | 14.5 3 | 16.4 4 | 14.7 5 | 13.5 Avg | 15.7
	This is a circuit containing a diode, where the aim is to measure the difference the presence of the diode makes in **DC Simulation** compared to a resistor (first circuit of the table). This simulation is expected to take longer as it will first calculate the correct mode of the diode. This circuit's simulation was completed with an average runtime of 1.7 ms	Trial | Time (ms) 1 | 1.3 2 | 1.4 3 | 2.1 4 | 1.6 5 | 1.9 Avg | 1.7
	This is a simple RC circuit where we will explore the performance of **AC Analysis** for a 2-node circuit (3×3 matrix). As discussed in section 5.7, AC analysis uses the same functions as MNA, however, AC Analysis requires 140 (20 points per decade $\times$ 7 decades) executions to obtain the results. For an RC circuit, this is completed with an average time of 21.8 ms	Trial | Time (ms) 1 | 18.9 2 | 32.3 3 | 13.6 4 | 24.6 5 | 19.8 Avg | 21.8
	Again, we use the same RC circuit, this time for **Transient Analysis**. Transient analysis (with one Capacitor/Inductor) runs MNA a total of 5 times (2 to find the Thevenin resistance, 1 for DC gain, 1 for HF gain, and one to find the steady-state). Then it calculates the output steady-state and transient and the input at 200 distinct time samples. For the simple RC circuit, this is completed with an average time of 6.7 ms	Trial | Time (ms) 1 | 5.0 2 | 7.0 3 | 7.1 4 | 8.4 5 | 6.2 Avg | 6.7

Table 6.1: Performance Testing Results (Part I)

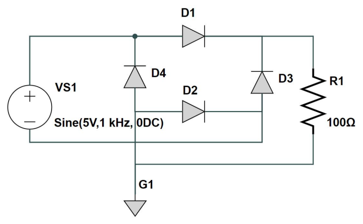
Diagram	Characteristics & Analysis	Execution Time
 D1 D4 D2 D3 R1 VS1 Sine(5V,1 kHz, 0DC) 100Ω G1	Finally, to experiment with a circuit that has a really high complexity, a **bridge rectifier** with a sinusoidal input is tested. This is a very heavy - in terms of computational complexity - circuit as it requires the calculation of the correct diode modes at each time sample (a total of 200). This is a circuit with 3 nodes and 5 (potentially - if all diodes are in conducting mode) voltage sources. Given that there are 4 diodes, a total of 16 different combinations must be tested to find the correct diode modes. The total complexity of this simulation is, therefore, $\mathcal{O}(i * d^2 * (n+m)^3)$ where: i = number of iterations (200), d = number of diodes (4), n = number of nodes (3), and m = number of voltage sources (≤ 5). However, this complexity is significantly improved with the caching of diodes, as the diodes remain in the same mode for a large number of iterations, leading to an amortized time complexity of $\mathcal{O}(i * (n+m)^3)$. In this case, the simulation is completed with an average runtime of 1268.6 ms	Trial | Time (ms) 1 | 1211.2 2 | 1322.0 3 | 1251.3 4 | 1256.6 5 | 1302.1 Avg | 1268.6

Table 6.2: Performance Testing Results (Part II)

6.4 User Testing

The user experience plays a crucial role in determining the success and sustainability of any application or platform that interacts with users. This section presents an overview of the user testing and feedback results obtained from individuals who have interacted with the application. By delving into their perspectives and insights, valuable information about the user experience, usability, and overall satisfaction with the simulation software is obtained. Through this analysis, we can assess the effectiveness of the application in meeting user expectations and identify areas for improvement.

User feedback was collected to assess whether the project aim has been fulfilled. In other words, the goal is to evaluate whether the final deliverable provides visualizations and simulation details that aid novice students in the learning of Analogue Electronics, and whether the application is user-friendly, fast, and intuitive.

The user feedback survey included a cohort of 10 electrical engineering students from both Imperial College and other international universities, who possess basic knowledge of analog electronics. This specific selection aimed to ensure that participants would not face any confusion regarding the simulated concepts, allowing them to fully grasp and utilize the application. Also, 5/10 users were first, second, and third-year students of Imperial, implying that they had already used ISSIE and were familiar with the top-level UI of the application. This made their feedback on the intuitiveness of the application slightly less significant, however, it was a necessary compromise, in order to assure that students who were closer (chronologically) to the first-year material would assess the functionalities of the application.

The feedback collection process was divided into two stages. All questions, apart from those asking the completion time of a task, were formulated as statements, on which users had to reflect, ranking them on a scale of 1 to 5, where 1 represents “Strongly Agree” and 5 “Strongly Disagree”.

6.4.1 Stage 1: Circuit Creation

The first stage involved creating a set of circuits and reporting on the intuitiveness, simplicity, and speed of circuit creation, while also comparing those factors with LTSpice. The results are summarized in tables 6.3 and 6.4, which provide insights into circuit creation times and user feedback on the circuit creation user interface.

For the circuit creation speed questionnaire, students have been split into users and non-users. For ADDIE, users refer to students who have used and are familiar with ISSIE, whereas for LTSpice, users are considered students who have used LTSpice for simulations in the past. In any case, students were not given any user guide or further instructions other than an image of the circuit they were required to create.

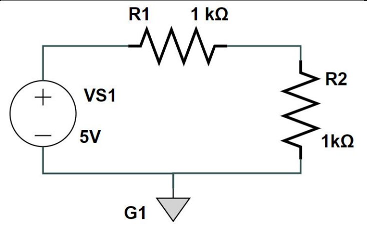
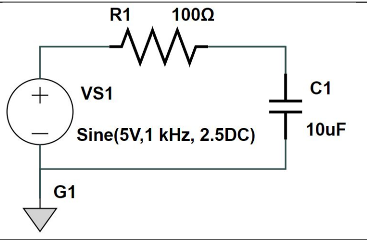
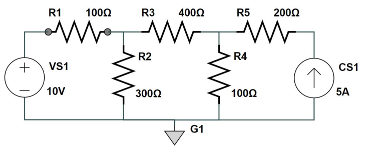
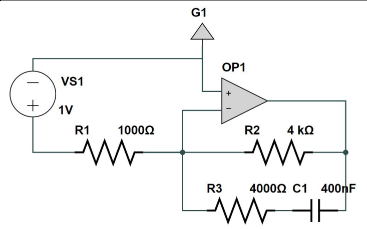
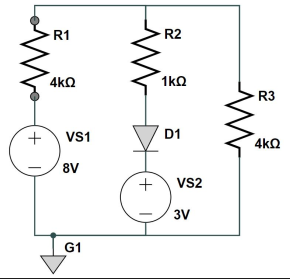
Diagram		Time Taken (Users)			Time Taken (Non-users)	
		ADDIE	LTSpice		ADDIE	LTSpice
	1-3 mins	20%	37.5%	1-3 mins	0%	0%
	3-6 mins	60%	62.5%	3-6 mins	20%	0%
	6-10 mins	20%	0%	6-10 mins	60%	0%
	> 10 mins	0%	0%	> 10 mins	20%	100%
	1-3 mins	80%	37.5%	1-3 mins	20%	0%
	3-6 mins	20%	62.5%	3-6 mins	60%	0%
	6-10 mins	0%	0%	6-10 mins	20%	50%
	> 10 mins	0%	0%	> 10 mins	0%	50%
	1-3 mins	40%	12.5%	1-3 mins	0%	0%
	3-6 mins	60%	87.5%	3-6 mins	100%	0%
	6-10 mins	0%	0%	6-10 mins	0%	100%
	> 10 mins	0%	0%	> 10 mins	0%	0%
	1-3 mins	40%	0%	1-3 mins	20%	0%
	3-6 mins	60%	100%	3-6 mins	80%	0%
	6-10 mins	0%	0%	6-10 mins	0%	0%
	> 10 mins	0%	0%	> 10 mins	0%	100%
	1-3 mins	60%	37.5%	1-3 mins	20%	0%
	3-6 mins	40%	62.5%	3-6 mins	80%	50%
	6-10 mins	0%	0%	6-10 mins	0%	50%
	> 10 mins	0%	0%	> 10 mins	0%	0%

Table 6.3: Time taken by the users to create each of the given circuits

#	**Statement**	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	The UI for adding components to the canvas and altering their position was intuitive	70%	30%	0%	0%	0%
2	The UI for connecting components using wires and formulating a circuit was intuitive	10%	50%	30%	10%	0%
3	I believe ADDIE's UI for creating circuits is simpler, faster, and clearer than LTSpice's	80%	20%	0%	0%	0%
4	I find LTSpice's wiring process time-consuming and prefer the auto-routing offered by ADDIE.	0%	60%	40%	0%	0%
5	I was quickly able to grasp how to navigate around the application	60%	40%	0%	0%	0%
6	I find the UI offered by ADDIE for improving my circuit's appearance (rotation, snapping of wires/components, etc.) easier to use than the equivalent of LTSpice	80%	20%	0%	0%	0%

Table 6.4: User feedback on the circuit creation UI of ADDIE

User Feedback Results

Starting from the first circuit (first row of table 6.3), we see that non-users found ADDIE an easier and faster tool for creating a simple potential divider, where 60% managed to create the full circuit in 6-10 minutes, compared to LTSpice, where novice users had to spend more than 10 minutes or did not manage to create the circuit. Given that this was the first circuit users were asked to create, and thus their first interaction with the two applications, it can be deduced that ADDIE offers a much more intuitive and user-friendly UI as users managed to complete the same task in significantly less time. On the other hand, we notice that users of ISSIE required more time compared to users of LTSpice, but this was expected as, after all, ADDIE has several differences compared to ISSIE, and it was their first time interacting with it.

Comparing the first with the second circuit, the two notable differences are the use of a capacitor instead of a resistor and the use of a Sinusoidal Voltage Source instead of a DC. Given that in both applications the process for adding basic components (resistors, capacitors, etc.) is similar, it can be said that the main difference is the use of a different type of voltage source. Comparing both the users and non-users of each application, it is evident that students managed to configure faster the voltage source in ADDIE. Specifically, a non-user needed more than 10 minutes to identify how to set up a sinusoidal voltage source in LTSpice.

After the first two circuits, we can assume that students are now more or less familiar with the process of setting up a circuit. In all three remaining circuits, students managed to create the schematics faster in ADDIE than in LTSpice. This leads to the assumption that ADDIE offers a much faster UI for creating circuits.

All of the above points are also verified from the set of statements the users were asked to reflect on, after creating all 5 circuits. These statements, along with the questionnaire results, are presented in table 6.4. The main conclusion from the user feedback presented in table 6.4 is that students do find ADDIE a simple and intuitive application, however, they tend to get a bit confused with the wiring algorithm. This is reasonable given that all the other analog simulation tools discussed in section 2.2 allow wires to be connected to both component ports and other wires, compared to ADDIE, which allows only component-to-component connections.

6.4.2 Stage 2: Simulations

The second stage involved running simulations on the created circuits to explore the simulation capabilities and their visualizations, and report on the intuitiveness, meaningfulness, and comprehensibility of the simulation results and the overall simulation process. Specifically, users were asked to run DC Operating Point Analysis on all circuits, and then also run AC Analysis on circuit 2, and Time Analysis on circuit 4, selecting the output of the operational amplifier as the output node of the simulation. The results of the questionnaire are summarized in table 6.5.

#	**Statement**	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	I was able to identify by myself, without spending more than a couple of minutes, how to set up and run a particular simulation.	70%	30%	0%	0%	0%
2	I was quickly able to understand how to enable/disable the on-canvas visualizations (both currents and node voltages or node numbering)	60%	40%	0%	0%	0%
3	Given the option to display each node's location and numbering on the circuit, I find that the DC Table offered by ADDIE is much easier to understand than the equivalent of LTSpice	100%	0%	0%	0%	0%
4	Seeing node voltages and current arrows on the canvas made the reading of simulation results faster	100%	0%	0%	0%	0%
5	The on-canvas visualizations helped me understand the underlying relations of each circuit	30%	50%	20%	0%	0%
6	After being instructed to "Change the value of the resistor and see how the results change", I was able to identify that I can use the slider for this task.	20%	40%	0%	40%	0%
7	I find the set-up process of AC and Time Simulation straightforward.	10%	90%	0%	0%	0%
8	In AC Simulation, I was able to identify myself how to change the graph units from non-dB to dB and from f to ω.	80%	20%	0%	0%	
9	In AC Simulation, I was able to understand what is displayed in the graph area and I find it relevant to the ADC material	12.5%	75%	0%	12.5%	0%
10	I find the time simulation results highly relevant to the ADC material	75%	25%	0%	0%	0%
11	Overall, the error messages of ADDIE helped me towards fixing the issues in my circuit (if applicable)	0%	20%	80%	0%	0%
12	Overall, I believe that the live simulation offered by ADDIE, along with the option to change values via a slider, enabled me to better understand the underlying relations of the circuit and the ADC material in general	20%	60%	20%	0%	0%

Table 6.5: User feedback on the simulation procedures and results of ADDIE

User Feedback Results

By examining the user feedback presented in table 6.5, several positive observations can be made. Firstly, based on statements 1, 2, and 7, it is evident that users perceive the simulation process as intuitive and easy to set up. Additionally, regarding DC simulation, users agree that ADDIE provides results in a clear and straightforward format, allowing them to grasp the information more quickly (statements 3 and 4).

Moving on to the AC and Time simulations, users found the graphed results to be self-explanatory and highly relevant to the ADC material. They were also able to manipulate the graph units without requiring additional guidance (statements 8, 9, and 10). Lastly, a large part of the students found the on-canvas visualizations and the live simulation offered by ADDIE helpful in understanding the underlying relations of each circuit.

A significant finding that emerged from the questionnaire was that not all students were able to recognize the slider's functionality and its potential for meaningful live simulations (statement 6). In response to this feedback, a measure was taken after the user feedback stage to address this issue. A concise text explaining the purpose and usage of the slider was added below the `div` that contains it. This addition aimed to provide clearer instructions and enhance the overall user experience by ensuring that all students understand how to utilize the slider effectively

Furthermore, considering the structure of the questionnaire used, it is worth noting that the majority of students did not encounter any errors in their circuits. This can be attributed to the fact that they were provided with explicit instructions on the circuit they needed to create, which intentionally excluded any errors. This explains the 80% neutral response to statement 11. To gain additional insights, verbal feedback was collected from students who tested the application. From this feedback, it was found that only two students encountered issues during their testing. The issues identified were the absence of a ground connection and the occurrence of duplicate connections (overlapping wires). Therefore, it is important to acknowledge that the assessment of ADDIE's error messages' clarity by students has not been effectively evaluated due to the limited occurrence of errors in the provided circuits.

6.4.3 Comparison of ADDIE and LTSpice

After providing feedback on circuit creation and circuit simulation, students were asked to directly compare ADDIE to LTSpice, and report on the usefulness of ADDIE. This was again completed by providing the 5 statements presented in table 6.6 and asking the students to rank them on a scale of 1 to 5 (1: "Strongly Agree" - 5: "Strongly Disagree").

Overall, students find ADDIE's UI for creating circuits and running simulations simpler and more user-friendly compared to the UI of LTSpice. Also, students agree that the live simulation offered by ADDIE is a valuable tool for an analog circuit simulator. Lastly, 75% of Imperial students (6 out of 8) agree that using a simplified Analog Electronics simulation tool like ADDIE as an extra resource to experiment and test the taught concepts would have improved their ADC learning experience and potentially their performance.

#	**Statement**	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	I find the setting-up of simulations in LTSpice complicated and confusing	20%	40%	20%	20%	0%
2	I struggle to understand error messages in LTSpice	40%	0%	60%	0%	0%
3	I find simulating circuits using ADDIE is easier compared to LTSpice	70%	10%	20%	0%	0%
4	I find the live simulation offered by ADDIE valuable for an analog simulation tool	30%	50%	10%	10%	0%
5	Overall, I feel using a simplified Analog Electronics tool like ADDIE as an extra resource to experiment and test the taught concepts would have improved my ADC learning experience and potentially my performance (Imperial students only)	37.5%	50%	12.5%	0%	0%

Table 6.6: Comparison between ADDIE and LTSpice

Chapter 7

Evaluation

The introduction chapter of this report set forth the project's primary objectives, which were utilized to establish and shape the project's direction. After researching appropriate algorithms, tools, and existing alternatives, in Chapter 3, a precise set of formal requirements was delineated, outlining the specific criteria that must be fulfilled (separated into essential and desirable) for the project to be considered successful. This chapter assesses the final project deliverable in relation to the aforementioned objectives and requirements. This is accomplished by evaluating the final deliverable of the project against (i) the project aim, (ii) ISSIE's Core Principles, and (iii) the project requirements. Lastly, the performance of the application will be evaluated and the feature set of ADDIE will be compared with alternative simulation tools.

7.1 Evaluation against Project Aim

The main aim of this project was to explore novel ways in which visualizations and simulations can help students understand basic analog concepts and combine those under a desktop application that will be capable of designing and simulating analog circuit designs.

The final deliverable has successfully met the project's aim of facilitating student comprehension of basic analog concepts through visualizations and simulations. The desktop application developed as part of this project offers a robust set of tools and functionalities that empower students to not only grasp analog concepts but also apply their knowledge to design and simulate analog circuits. Through the integration of interactive visualizations, students can now visualize abstract concepts and gain a deeper understanding of the underlying relations. Also, it is important to note that the application effectively supports all the concepts of the first part of the ADC module, allowing users to explore and manipulate various parameters of analog circuits and observe the corresponding behaviors in real time.

Starting from DC Operating Point Analysis, students can see the node voltages and branch currents visualized on the canvas, allowing them to read the results faster. At the same time, they can view in real-time, how changing the value of a component (e.g. using the provided slider) affects the voltages and currents of the circuit. This allows the users to better understand the underlying relations of each circuit. For example, students can directly see whether a specific voltage/current is directly proportional or inversely proportional to the selected component's value. Similarly, for AC and Time analysis, the graphs provided, allow users to view how their circuits perform against frequency and time, and again, how the values of the circuit affect the corner frequencies or the transient response. Moreover, users can split the output of the time simulation into the two sub-signals that formulate it: the steady-state and transient signals. Lastly, the DC Operating Point Analysis simulation presents to the users a set of simple equations that are formulated by the circuit, as well as the option to view the Thevenin equivalent seen by a component. These two options, allow users to further understand what leads to the solution of their circuit and provide a simulation highly relevant to the ADC material. These conclusions are also verified by the user feedback questionnaire. Specifically, students were asked how much they agreed with the following statements:

1. *The on-canvas visualizations helped me understand the underlying relations of each circuit*

2. *Seeing node voltages and current arrows on the canvas made the reading of simulation results faster*

3. *I believe that the live simulation offered by ADDIE, along with the option to change values via a slider, enabled me to better understand the underlying relations of the circuit and the ADC material in general*

4. *I feel using a simplified Analog Electronics tool like ADDIE as an extra resource to experiment and test the taught concepts would have improved my ADC learning experience and potentially my performance*

The 4 statements had an average score of 4.1, 5.0, 4.0, and 4.25 respectively, on the scale 1: Strongly Disagree - 5: Strongly Agree. In other words, the majority of students agreed (or strongly agreed) that ADDIE effectively harnesses the power of visualizations and simulations to enhance their comprehension of basic analog concepts.

7.2 Evaluation against ISSIE's Core Principles

In chapter 1, ISSIE's Core Principles were outlined, focussing on the robustness, obviousness, and intuitiveness of the application. In this section, the features of ADDIE will be examined, to assess whether they align with these principles.

7.2.1 Robustness

The application's robustness, which refers to its ability to convey erroneous behavior to the users in an informative manner that aids in the fast resolution of the issues, was partially examined in the user feedback section. Most of the students did not encounter errors in their circuits, and for this reason, could not report on the robustness of ADDIE. However, the two students that encountered errors noted that the error message that appeared helped them fix the issues of their circuits. In general, the robustness of the application is good, but can definitely be improved further. Specifically, currently, when the circuit is not solvable (determinant of MNA matrix is zero), and the cause of the error is not one of the cases covered by the error checking function (see section 5.5.3), users get an error message stating "Unknown error in the circuit. Cannot run simulation". This message, although it informs users about an error, it does not give a proper explanation of the cause or specify the exact error. An ideal error-checking function would be one that finds the nullspace of the MNA matrix and is able to deduct from there which components/connections/nodes make the circuit not solvable, and convey that information to the users.

7.2.2 Obviousness

The principle of obviousness emphasizes the importance of designing an application and its visualizations in a way that conveys their meaning without requiring users to consult a user guide. The user interface of ADDIE has been Carefully crafted to maximize obviousness. Non-trivial operations are either accompanied by concise explanatory text or guided through the use of informative popups. Users agree on the obviousness of ADDIE, and this is confirmed by the student feedback questionnaire analyzed in section 6.4. Specifically, students were able to understand all the visualizations provided by ADDIE (nodes and currents on canvas, graph results), specifying that it was clear what each plot/on-canvas result represented. Moreover, they found the popups and buttons they encountered in the process of performing modifications to their circuits or running simulations self-explanatory and clear. Finally, it is worth noting that while a comprehensive evaluation was not conducted on this aspect, students who encountered issues with their circuits expressed appreciation for the helpfulness of the error messages displayed in the notification boxes. These error messages aided them in identifying and resolving the underlying issues, contributing to a smoother and more effective troubleshooting process.

7.2.3 Intuitiveness

Intuitiveness entails designing the user interface (UI) in a manner that ensures users of all backgrounds can easily understand and utilize the application and its features. Exceptional care was taken during the UI design phase to prioritize this aspect, as it sets ADDIE apart from other Analog Simulation tools in the market. ADDIE provides a user interface that is inherently self-explanatory, simplifying the process of circuit creation and simulation.

Starting from the circuit creation, the available components are presented both as buttons in the File Menu section and as a Catalogue in the Catalogue tab of the RHS pane, following the UI style of ISSIE. When a component button is clicked, a popup appears, prompting students to select a value for the component if necessary. They can then easily add the selected component to their circuit by dragging and dropping it into place. To connect components, users utilize the ports located at the two ends of each component. When the mouse hovers over a component, the ports become visible, indicating that they can be used for establishing connections. Once a connection is initiated, all the ports become visible, allowing users to complete the connection.

In Table 6.4, which presents the results of the user feedback questionnaire on ADDIE's circuit creation UI, we observe that the majority of students strongly agree that navigating the application and adding components is intuitive. However, it is worth noting that some non-ISSIE users mentioned that the wiring procedure, which differs from other tools (wires must start/end in component ports), initially posed a slight challenge for them.

Considering the simulations supported by ADDIE, students also found the process of setting up and running simulations to be straightforward and uncomplicated. However, as previously discussed in Section 6.4.2, it was discovered through the questionnaire that not all students fully grasped the functionality of the slider. To address this issue, a concise and informative text was incorporated below the `div` element containing the slider, explaining its purpose and how it should be used.

7.3 Performance Evaluation

While performance was not a stringent requirement for this project, it is worth noting that the desktop application developed exhibits commendable smoothness and fluidity. Despite not prioritizing performance optimization, the application efficiently executes all simulations within reasonable timeframes which contributes significantly to the overall seamless user experience. Section 6.3 and tables 6.1 and 6.2 contain a detailed analysis of the complexity of each simulation, along with time measurements to allow comparison between the different circuits and simulations.

7.3.1 Software Robustness

The software's robustness, which refers to its ability to function correctly even in exceptional circumstances, was thoroughly examined in section 6.1. The majority of functions in the codebase were found to be exception-free, while those that could throw exceptions had checks and system constraints in place to prevent invalid arguments. Similar verifications were performed for cases involving `failwithf`, and it was concluded that the cases were indeed not possible due to the implemented checks and constraints. Extensive manual and user testing revealed no crashes or undefined behavior, even when subjected to various interactions with the user interface, including malformed inputs. This demonstrated the resilience of ADDIE's error-checking and handling systems.

The decision to develop ADDIE in F# proved to be beneficial for its robustness. The immutability of the application's data and the powerful type inference system ensured that every case was explicitly handled, reducing undefined behavior and enhancing the codebase's robustness. The compiler's analysis caught potential errors, such as unmatched or improperly ordered cases, during the coding phase, significantly reducing the likelihood of bugs during runtime.

7.4 Evaluation against Requirements

In Chapter 3, an elaborate list of requirements was presented, categorized into essential and desirable. These requirements served as a roadmap during the project's implementation phases. This section aims to assess the extent to which the specified requirements have been fulfilled.

Requirement	Comments	Completed?
E1.1	Utilising the top-level UI of ISSIE, ADDIE contains a graphical interface that allows users to easily design complex analog circuits containing any number of components.	Yes
E1.2	All the required new components (Voltage/Current Source, Resistor, Capacitor, Inductor, Diode, Operational Amplifier) have been added and can be used to create schematics.	Yes
E1.3	The ISSIE codebase has been successfully altered to support connections from/to any component port (no output-to-input constraint).	Yes
E1.4	ADDIE runs a set of pre-defined checks to ensure that the circuit does not contain any of the common errors that can cause the simulation to break (e.g. absence of ground). However, this requirement is marked as partially completed as the full implementation would include a function that explores the nullspace of the MNA matrix to give a definite answer on the components/connections that cause the simulation to break.	Partially
E1.4.1	The components/connections that are identified as erroneous by the error checking function are highlighted, as required, to maximize the amount of guidance given to the users.	Yes
E1.5	After successfully parsing the circuit, the locations of the identified nodes are visualized on the canvas as red dots, along with their number. This numbering is then used to read the DC results and set up the AC & Time Simulations.	Yes
E1.6	Modified Nodal Analysis is successfully implemented with the required modifications in place to support capacitors, inductors, and linearized diodes. The results are displayed in a table in the 'DC Simulation' sub-tab in the RHS area of the application.	Yes
E1.6.1	A stable algorithm was derived to obtain the branch currents' values along with their direction.	Yes
E1.6.2	All node voltages and branch currents can be visualized on the circuit. The voltages are presented above the node locations (see requirement **E1.5**) whereas currents are presented using arrows to demonstrate the direction of current.	Yes
E1.7	A slider is included in the 'Properties' tab when a component is selected. The slider allows the fast change of the component's value. The simulation is triggered whenever there is a change on the `CanvasState` (including value change), which creates the effect of a live simulation.	Yes

Table 7.1: Evaluation Against Application Requirements (Part I)

Requirement	Comments	Completed?
E1.8	Frequency response is successfully implemented using the same functions as MNA for a wide range of frequencies.	Yes
E1.8.1	Frequency Response results (Magnitude & Phase) are plotted against frequency (logarithmic scale) in the same graph using solid line for Mangnitude and dotted for Phase, and two distinct y-axes.	Yes
D1.1	Time domain simulation is partially implemented, allowing only constrained circuits to be simulated (see below).	Partially
D1.1.1	Time domain simulation successfully runs for circuits containing any number of non-reactive components. Also, the optimization introduced in section 5.6.3 (caching diode modes) achieves a much faster execution time.	Yes
D1.1.2	Time domain simulation successfully runs for circuits containing one reactive component, using the algorithm proposed in the ADC module. This also allows the transient parameters (time constant τ, transient amplitude, HF & DC gain) to be obtained and presented to the users.	Yes
D1.1.3	Due to the reasons analyzed in section 4.3, a complete time domain simulation is not supported by ADDIE.	No
D1.2	Several graph-area improvements were implemented under requirement **D1.2** (see below).	Partially
D1.2.1	ADDIE allows users to change the units of the axes in the Frequency Response plots ($\omega \leftrightarrow$ f & non-dB $\leftrightarrow$ dB).	Yes
D1.2.2	Detached windows for graph results were not implemented as part of this project. The use of detached windows is a requested feature shared by both ISSIE and ADDIE (see the relevant GitHub Issue).	No
D1.2.3	Only actual frequency response is available in AC Simulation (not straight-line approximation).	No
D1.2.4	High Precision AC Simulation is implemented using the algorithm derived in section 4.5, allowing users to select whether they want to run normal or high-precision frequency response.	Yes
D1.3	Support for real diodes (non-linear) is currently not implemented. However, the algorithm derived in section 4.4 presents a stable way in which it can be implemented in the future.	Partially
D1.4	Users can select from the 'DC Simulation' sub-tab a component and view the Thevenin/Norton equivalent circuit the component 'sees' by viewing the Thevenin/Norton parameters.	Yes
D1.5	Users can view a set of straightforward equations that can be obtained from the circuit in the 'DC Simulation' sub-tab.	Yes
D1.6	Embedded exercises relevant to the ADC material for students have not been implemented.	No
D1.7	Symbolic simulation of circuits (for example to display transfer functions) is not supported by ADDIE because of the reasons analyzed in section 4.3.	No

Table 7.2: Evaluation Against Application Requirements (Part II)

Requirement	Comments	Completed?
E2.1	The project successfully produced bug-free and performant code that aligns with ISSIE's Coding Guidelines and the MVU Coding Principles. The performance of the code was assessed through quantitative testing, while its accuracy and robustness against failures were also confirmed.	Yes
E2.2	The code has been developed with a focus on maintainability. Special attention has been given to utilizing standard library data structures and functions whenever feasible, and any new types or processes that have been introduced are thoroughly documented within the code.	Yes
E2.3	All functions in the provided code have been accompanied by XML comments, along with additional inline comments that clarify the functionality of important sections. These efforts have been made to enhance the maintainability of the codebase for future developers.	Yes
E2.4	The non-trivial algorithms used in ADDIE have not been analyzed in the ADDIE Wiki yet, but, this report provides an in-detail analysis of all the utilized algorithms and data structures.	Not yet
E2.5	By combining the contents of this report with the information provided in the project README, future developers have all the necessary information to further develop ADDIE.	Yes
D2.1	A static website has not been created to demonstrate the features of the application yet, but, it will be created in due course.	Not yet

Table 7.3: Evaluation Against Software Quality Requirements

7.5 Comparison with Existing Tools

In order to be adopted and used by Imperial students, ADDIE must be preferable over the currently used alternatives. In this section, ADDIE will be compared against the two most commonly used Analog Simulation Tools (LTSpice & CircuitLab), in terms of the features each tool supports.

Feature	LTspice[32]	CircuitLab[42]	**ADDIE**
Diagram Editor	✓	✓	✓
Support of Simple Components (Sources, Passive components, Opamps, Linearized Diodes)	✓	✓	✓
Support of Non-Linear Diodes	✓	✓	✗ [a]
Support of Advanced Components (BJTs, MOSFETs, transformers, etc.)	✓	✓	✗
Automatic Wire Routing	✗	✗	✓
Teaching Oriented	✗	✓	✓
Unclattered User Interface	✗	✓	✓
DC Operating Point Analysis	✓	✓	✓
AC Analysis	✓	✓	✓
Time Simulation	✓	✓	✓/✗ [b]
Live Simulation	✗	✗	✓
On-canvas Visualisation of results	✗	✗	✓
Thevenin Equivalent Circuit	✗	✗	✓
DC Equations	✗	✗	✓
Fast Value change using slider	✗	✗	✓
Change of units in AC Simulation	✓/✗ [c]	✗	✓
Time Analysis: Both steady-state and transient solution	✗	✗	✓
Detailed errors and errors displayed on circuit	✗	✗	✓/✗ [d]
Open-Source	✓	✗	✓
Extensible by EE students	✗	✗	✓
User Guide	✓	✓	✓
Actively maintained	✓	✓	✓
Fully cross-platform	Windows, MacOS	All (Web-app)	All

Table 7.4: Comparison of Features with Existing Analog Simulation Tools

[a] *Support for non-linear components (diodes) is partially implemented following the algorithm described in 4.4*

[b] *Time Simulation in ADDIE is restricted as described in section 5.8*

[c] *y-axis units are changeable; x-axis is always in Hz*

[d] *Not fully implemented (as analyzed in 7.4)*

Chapter 8

Conclusion

In conclusion, this project successfully presents an innovative analog circuit simulator desktop application targeted to 1st-year Engineering students. The application not only provides an interactive platform for students to explore and experiment with various circuits, but also offers valuable insights and real-time feedback to facilitate their understanding. These conclusions were also verified by collecting feedback from users who are familiar with LTSpice, which is the primarily used alternative by Imperial Students.

Throughout the course of this project, numerous distinctive and intriguing challenges were encountered, yet the journey of overcoming them has proven to be exceptionally gratifying. One of the initial challenges I encountered was devising a suitable data structure to represent the circuits effectively and then designing a stable algorithm to parse the circuit. This was a very important step of the implementation process as I had to think proactively about what sort of input the (not finalized at the time) simulation algorithms would require. In the end, the selected data structure: A list of components connected to each node, along with information about the connection port, covered all possible needs, and the designed algorithm (discussed in section 5.5.2) parses effectively the `CanvasState` to produce the required data.

Writing the simulation algorithms, along with making the necessary modifications to the `CanvasState`, was also a challenging task. All three simulation types had their own demanding aspects, but, in the final deliverable, all simulations have been implemented successfully, using stable and extensible algorithms.

Lastly, as discussed in section 7.4, all the essential requirements described in chapter 3 have been fully met. Also, most of the desirable features were implemented as well, leading to the successful completion of this project.

8.1 Further Work

As already mentioned in chapter 4, creating an application with the capabilities of LTSpice (which has been actively developed since 1999) exceeds both the scope and the time constraints of a final-year project. The goal for the current project, which has been met, was to create a strong basis of an ISSIE-style LTSpice equivalent, which will be further developed over the coming years, to create a robust, obvious, and intuitive analog circuit simulation tool, matching the capabilities of LTSpice.

The future development process is expected to follow a pattern similar to ISSIE, which has been actively developed over the past three years (since its inception in 2020), substantially increasing its capabilities and value. Specifically, tasks like designing and implementing a robust and stable algorithm to support non-linear components, or implementing a time domain simulation algorithm that works irrespective of the input circuit can be completed as part of two different final-year projects in the coming years. On the other hand, tasks that do not require that much research and/or implementation time, like improvements to the UI, can be completed as part of UROPs[51] or as part of project work for the HLP[16] module. Table 8.1 provides a set of features that I find to be the most impactful considering the purpose of the application, and which can be completed in the future, to aid in the creation of an ISSIE-style LTSpice-capable simulator.

Feature	**Estimated Time**	**Benefit**
Real Diode Implementation using the algorithm described in section 4.4, and by extension other non-linear components (e.g. BJTs) as well	High	9
Improved error checking algorithm which provides feedback on errors by analyzing the nullspace of the MNA matrix	Medium	8
Using a tool like PicoScope[52], create a clean simulated scope interface that has X & Y axis controls and a trigger level that lets students practice scope control in an ideal simulation environment	Medium-High	7
Incorporation of symbolic math operations library to allow calculation of corner frequencies and transfer functions	High	6
Implement a stable time domain simulation algorithm that operates regardless of the number and type of components used	High	5
Implement embedded exercises as part of the application (e.g. straight-line approximation drawing in the graph area)	Medium-High	4
Support simulation of multiple nodes in AC Simulation, along with the option to view the straight-line approximation plots	Low	3
Include a draggable divider between the canvas and the graph that can resize the graph area. Also, allow the graph to be detached in a different window to improve readability	Medium	2
Allow current-to-current and current-to-voltage relations to be simulated in the graph	Medium	1

Table 8.1: Potential improvements to the application, ranked by benefit

8.2 Final Remarks

On a final note, developing an analog circuit simulation application proved to be an immensely demanding and fulfilling endeavor. The project's final result has brought me great satisfaction as it allowed me to showcase the engineering expertise I acquired throughout my degree.

Bibliography

[1] *1. What Is React? - What React Is and Why It Matters [Book].* ISBN: 9781491996737. URL: https://www.oreilly.com/library/view/what-react-is/9781491996744/ch01.html (visited on 01/31/2023).

[2] Admin. *Studies Confirm the Power of Visuals to Engage Your Audience in eLearning.* URL: https://www.shiftelearning.com/blog/bid/350326/studies-confirm-the-power-of-visuals-in-elearning (visited on 01/31/2023).

[3] *An Algorithm for Modified Nodal Analysis.* URL: http://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA3.html (visited on 05/20/2023).

[4] *Beginning Elm.* Elm Programming. URL: http://elmprogramming.com (visited on 01/25/2023).

[5] Kenneth M. Brow and William B. Gearhart. "Deflation techniques for the calculation of further solutions of a nonlinear system". In: *Numerische Mathematik* 16.4 (Jan. 1, 1971), pp. 334–342. ISSN: 0945-3245. DOI: 10.1007/BF02165004. URL: https://doi.org/10.1007/BF02165004 (visited on 06/19/2023).

[6] *Build cross-platform desktop apps with JavaScript, HTML, and CSS — Electron.* URL: https://electronjs.org/ (visited on 01/19/2023).

[7] *Bulma: Free, open source, and modern CSS framework based on Flexbox.* URL: https://bulma.io/ (visited on 01/25/2023).

[8] cartermp. *F# for Web Development.* URL: https://learn.microsoft.com/en-us/dotnet/fsharp/scenarios/web-development (visited on 01/17/2023).

[9] cartermp. *What is F#.* URL: https://learn.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp (visited on 01/16/2023).

[10] Oliver Caviglioli. *Six ways visuals help learning.* My College. URL: https://my.chartered.college/impact_article/six-ways-visuals-help-learning/ (visited on 06/14/2023).

[11] *Chart.js.* URL: https://www.chartjs.org/ (visited on 01/31/2023).

[12] *Chromium.* URL: https://www.chromium.org/Home/ (visited on 01/19/2023).

[13] Tom Clarke. *Issie - an Interactive Schematic Simulator with Integrated Editor.* original-date: 2020-06-27T16:00:12Z. Nov. 5, 2022. URL: https://github.com/tomcl/issie (visited on 11/22/2022).

[14] *Coding guidelines for ISSIE · tomcl/issie Wiki · GitHub.* URL: https://github.com/tomcl/issie/wiki/Coding-guidelines-for-ISSIE (visited on 12/21/2022).

[15] *Configuring an AC Analysis in Multisim - NI.* URL: https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000YH7bCAG&l=en-GB (visited on 01/24/2023).

[16] *ELEC60015 High Level Programming.* URL: http://intranet.ee.ic.ac.uk/electricalengineering/eecourses_t4/course_content.asp?c=ELEC60015&s=J3#start (visited on 12/20/2022).

[17] *Elmish · Elmish.* URL: https://elmish.github.io/elmish/ (visited on 11/23/2022).

[18] *Fable · Call JS from Fable.* URL: https://fable.io/docs/communicate/js-from-fable.html (visited on 02/04/2023).

[19] *Fable · Packages.* URL: https://fable.io/packages/#/package/Fable.React (visited on 01/19/2023).

[20] *Fable.React.* original-date: 2016-08-16T15:30:17Z. Jan. 24, 2023. URL: https://github.com/fable-compiler/fable-react (visited on 01/25/2023).

[21] Henrik Feldt. *Expecto.* original-date: 2016-10-22T11:49:30Z. June 11, 2023. URL: https://github.com/haf/expecto (visited on 06/14/2023).

[22] *Feliz.Plotly.* URL: https://shmew.github.io/Feliz.Plotly/ (visited on 06/03/2023).

[23] *Fulma.* URL: https://fulma.github.io/Fulma/ (visited on 01/25/2023).

[24] D.G. Haigh, T.J.W. Clarke, and P.M. Radmore. "Symbolic Framework for Linear Active Circuits Based on Port Equivalence Using Limit Variables". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 53.9 (Sept. 2006). Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 2011–2024. ISSN: 1558-0806. DOI: 10.1109/TCSI.2006.882815.

[25] Stanisław Hałgas. "A SPICE-Oriented Method for Finding Multiple DC Solutions in Nonlinear Circuits". In: *Applied Sciences* 13.4 (Jan. 2023). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, p. 2369. ISSN: 2076-3417. DOI: 10.3390/app13042369. URL: https://www.mdpi.com/2076-3417/13/4/2369 (visited on 06/19/2023).

[26] *iCircuit - The Realtime Circuit Simulator and Editor.* URL: https://icircuitapp.com/ (visited on 11/23/2022).

[27] *Integration Methods.* URL: https://ltwiki.org/files/LTspiceHelp.chm/html/integration_method_issues.htm (visited on 06/08/2023).

[28] *Introduction · An Introduction to Elm.* URL: https://guide.elm-lang.org/ (visited on 01/25/2023).

[29] *Introduction to the 'Why use F#' series — F# for fun and profit.* URL: https://fsharpforfunandprofit.com/posts/why-use-fsharp-intro/ (visited on 01/16/2023).

[30] *Kirchhoff's Current Law (KCL) — Divider Circuits And Kirchhoff's Laws — Electronics Textbook.* URL: https://www.allaboutcircuits.com/textbook/direct-current/chpt-6/kirchhoffs-current-law-kcl/ (visited on 01/22/2023).

[31] *Learning F Sharp by Example.* URL: https://www.fincher.org/tips/Languages/fsharp.shtml (visited on 01/17/2023).

[32] *LTspice Simulator — Analog Devices.* URL: https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html (visited on 11/23/2022).

[33] *math.js — an extensive math library for JavaScript and Node.js.* URL: https://mathjs.org/ (visited on 01/31/2023).

[34] *ml-matrix.* npm. Nov. 5, 2022. URL: https://www.npmjs.com/package/ml-matrix (visited on 01/31/2023).

[35] *Modified Nodal Analysis - Intro.* URL: http://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA2.html (visited on 05/20/2023).

[36] *Ngspice, the open source Spice circuit simulator - Intro.* URL: https://ngspice.sourceforge.io/ (visited on 11/23/2022).

[37] Node.js. *Node.js.* Node.js. URL: https://nodejs.org/en/ (visited on 01/19/2023).

[38] *Non-Linear DC Analysis.* URL: http://www.ecircuitcenter.com/SpiceTopics/Non-Linear%20Analysis/Non-Linear%20Analysis.htm (visited on 01/31/2023).

[39] *Nonlinear equations.* URL: https://www.isical.ac.in/~arnabc/numana/nonlin1.html (visited on 01/31/2023).

[40] *Not Helpful Feedback.* URL: https://nidx.co1.qualtrics.com/jfe/form/SV_6nWUkjClCUvhSTz?Q_Language=EN&NI_Referrer=https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000YH8KCAW&l=en-US (visited on 01/22/2023).

[41] *numbers.js: JavaScript's most popular math library.* URL: http://numbers.github.io/ (visited on 06/14/2023).

[42] *Online circuit simulator & schematic editor.* CircuitLab. URL: https://www.circuitlab.com/ (visited on 11/23/2022).

[43] *Overview of types in F# — F# for fun and profit.* URL: https://fsharpforfunandprofit.com/posts/overview-of-types-in-fsharp/ (visited on 01/16/2023).

[44] pavelsavara. *Run .NET from JavaScript.* Nov. 10, 2022. URL: https://learn.microsoft.com/en-us/aspnet/core/client-side/dotnet-interop (visited on 06/08/2023).

[45] *Plotly.* URL: https://plotly.com/nodejs/ (visited on 01/31/2023).

[46] *Qucs project: Quite Universal Circuit Simulator.* URL: https://qucs.sourceforge.net/index.html (visited on 11/23/2022).

[47] *Reconciliation – React.* URL: https://reactjs.org/docs/reconciliation.html (visited on 01/25/2023).

[48] *Rendering Elements – React.* URL: https://reactjs.org/docs/rendering-elements.html (visited on 01/25/2023).

[49] *The Elm Architecture · An Introduction to Elm.* URL: `https://guide.elm-lang.org/architecture/` (visited on 01/25/2023).

[50] *ts2fable.* original-date: 2017-03-25T23:21:25Z. May 28, 2023. URL: `https://github.com/fable-compiler/ts2fable` (visited on 06/03/2023).

[51] *Undergraduate Research Opportunities Programme — Imperial students — Imperial College London.* URL: `https://www.imperial.ac.uk/urop/` (visited on 06/21/2023).

[52] *USB oscilloscopes & mixed signal oscilloscopes.* URL: `https://www.picotech.com/oscilloscope/2000/picoscope-2000-overview` (visited on 06/21/2023).

[53] *Using the type system to ensure correct code — F# for fun and profit.* URL: `https://fsharpforfunandprofit.com/posts/correctness-type-checking/` (visited on 06/14/2023).

[54] *Vulnerability in Electron-based Application: Unintentionally Giving Malicious Code Room to Run - Blog - CertiK Security Leaderboard.* URL: `https://www.certik.com/resources/blog/vulnerability-electron-based-application-malicious-code-execution` (visited on 01/19/2023).

[55] Yusuf@TWC. *Best Free Circuit Simulation software for Windows 10.* The Windows Club. Apr. 3, 2021. URL: `https://www.thewindowsclub.com/best-free-circuit-simulation-software-for-windows-10` (visited on 01/17/2023).

Appendix A

Screenshots of ADDIE

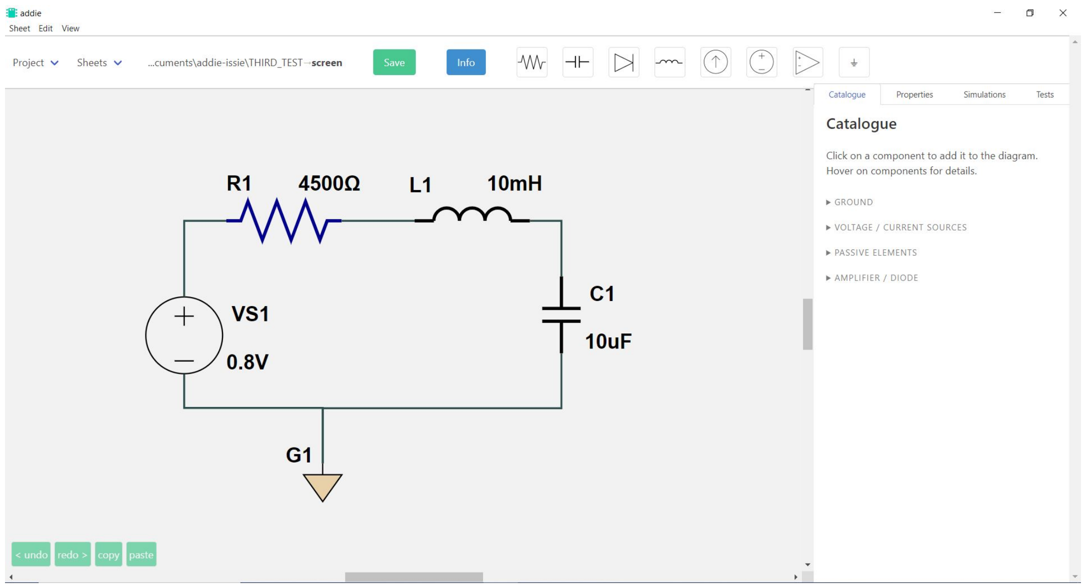
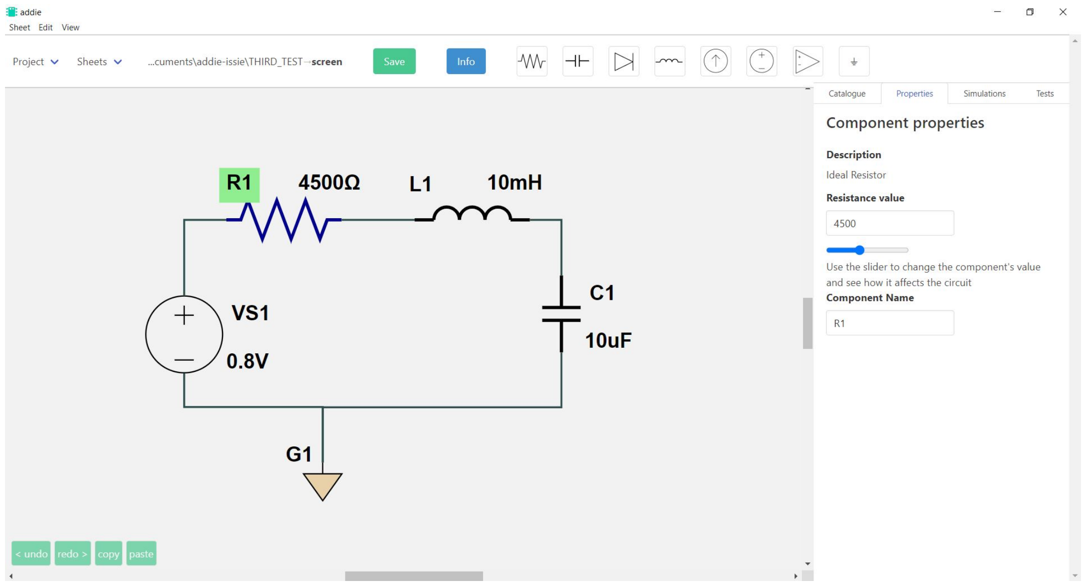


Figure A.1: ADDIE's User Interface (Catalogue tab)



Component properties

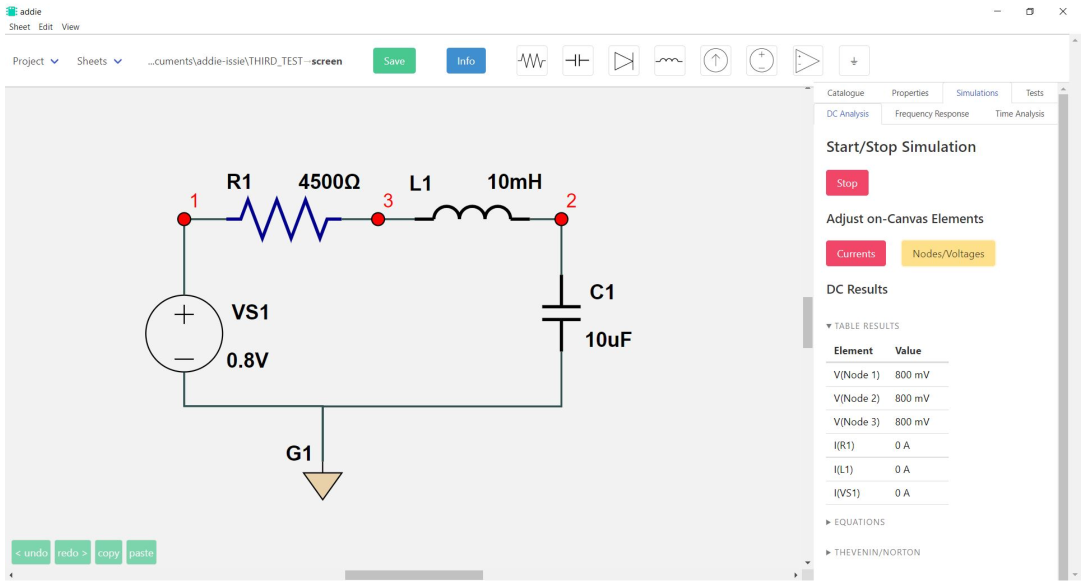
Description
Ideal Resistor

Resistance value
4500

Use the slider to change the component's value and see how it affects the circuit

Component Name
R1

Figure A.2: ADDIE's User Interface (Properties tab)



Start/Stop Simulation

Stop

Adjust on-Canvas Elements

Currents
Nodes/Voltages

DC Results

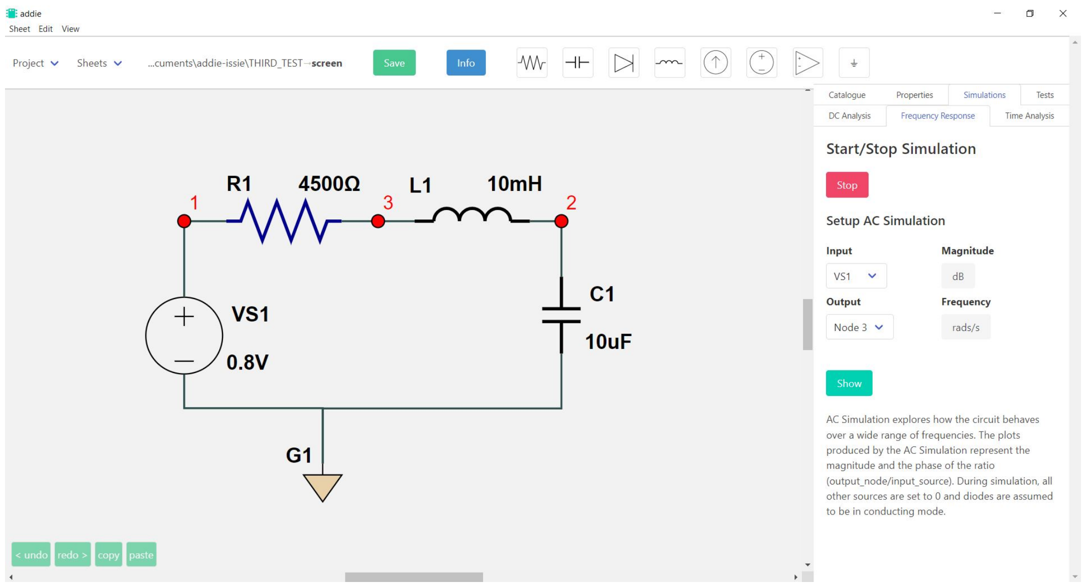
▾ TABLE RESULTS

Element	Value
V(Node 1)	800 mV
V(Node 2)	800 mV
V(Node 3)	800 mV
I(R1)	0 A
I(L1)	0 A
I(VS1)	0 A

▸ EQUATIONS

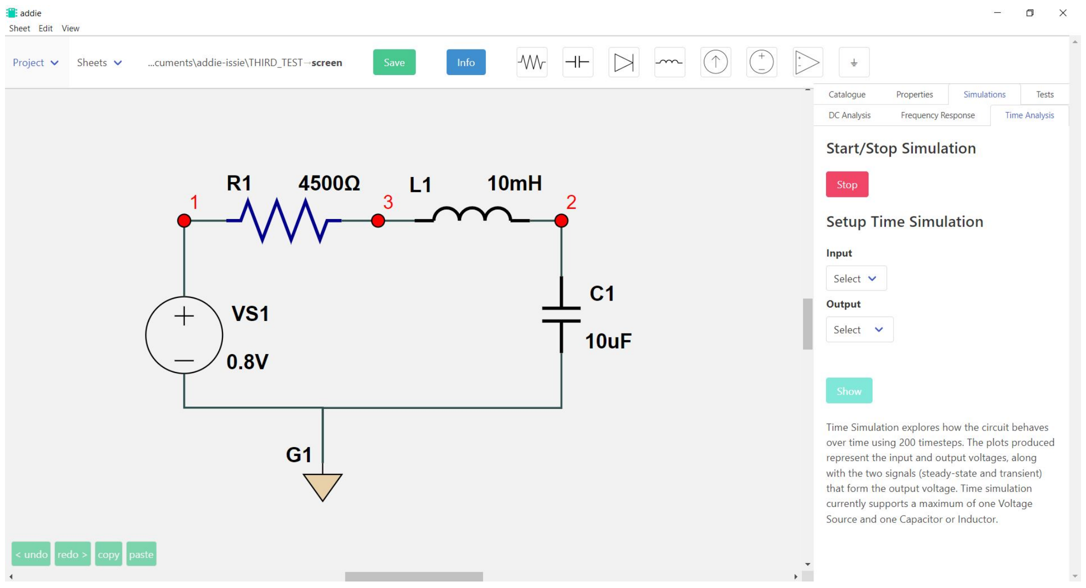
▸ THEVENIN/NORTON

Figure A.3: ADDIE's User Interface (DC Simulation tab)



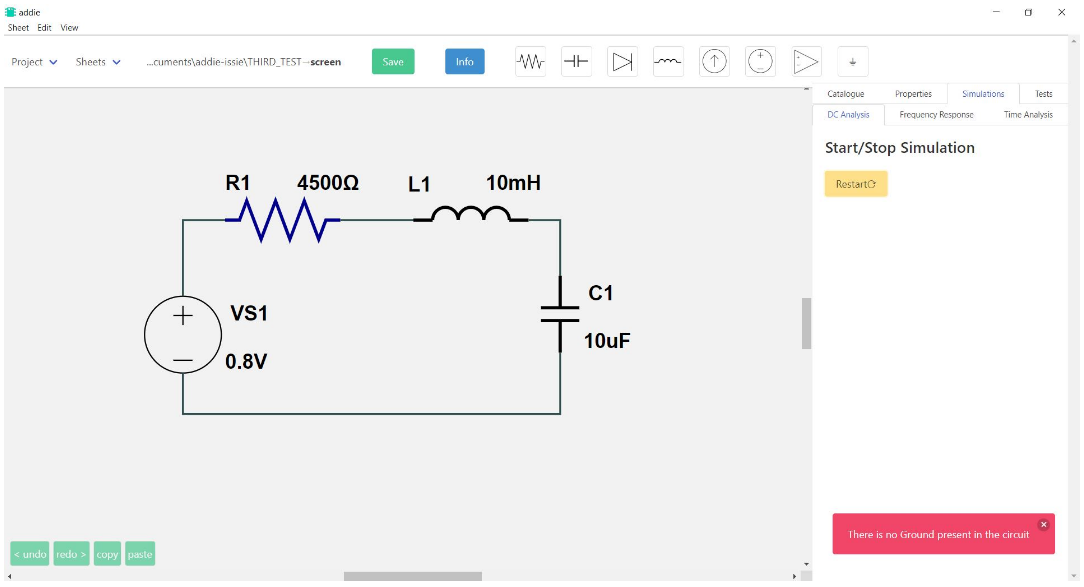
AC Simulation explores how the circuit behaves over a wide range of frequencies. The plots produced by the AC Simulation represent the magnitude and the phase of the ratio (output_node/input_source). During simulation, all other sources are set to 0 and diodes are assumed to be in conducting mode.

Figure A.4: ADDIE's User Interface (AC Simulation tab)



Time Simulation explores how the circuit behaves over time using 200 timesteps. The plots produced represent the input and output voltages, along with the two signals (steady-state and transient) that form the output voltage. Time simulation currently supports a maximum of one Voltage Source and one Capacitor or Inductor.

Figure A.5: ADDIE's User Interface (Time Simulation tab)



There is no Ground present in the circuit

Figure A.6: Example of Error Message displayed to the users