FEDERAL UNIVERSITY OF ITAJUBA - UNIFEI

Graduate program in Electrical Engineering

Hanneli Carolina Andreazzi Tavante

Functional Programming Applied to Electrical Engineering

Supervision

Main Supervisor
Prof. Ph.D. Benedito Donizetti Bonatto
Co-Supervisor
Prof. Ph.D. Maurilio P. Coutinho

December 2019
Itajuba - MG - Brazil

2018100494

Acknowledgements

A chain of unexpected events and some unexpected people around the world brought me here. I wouldn't be writing this page if my supervisors, Dr. B. Bonatto and Dr. M. Coutinho hadn't accepted me as their student. I kindly thank them for their endless patience with my hyperactive behaviour towards science. They were the best people to deal with the unusual ideas I proposed during these two years as a MSc student.

I also thank our University, UNIFEI, for being there all the time, during my bachelor's and during my graduate studies. In times when truths and tautologies are threatened, resisting to ignorance is a brave act.

Last but not least, I wouldn't have gone far if my two best friends, A. A. and A. A. (what a coincidence for these abbreviations!) hadn't been supporting me for the last decade or so. I dedicate all the lines of code one will find here to my partner B. C. for being a true enthusiast of learning and of science. And most importantly, to my family, V. R and I. R.

H. C. A. Tavante

“It was not by making yourself heard but by staying sane that you carried on the human heritage.”

George Orwell

Contents

Abstract

Many engineering projects rely on software to execute simulations and analysis of a wide variety of domains. Computer programs are great allies of the engineers when it comes to simulations, including the ones for electromagnetic transient analysis. However, a single programming paradigm (the imperative paradigm) seems to have dominated most of the commercial and academic applications.

This work presents and implements an algorithm to analyse simple electromagnetic transient circuits adopting functional programming. The code uses the nodal analysis found on industry programs like the EMTP (Electromagnetic Transients Program). The results of adopting the Haskell language and functional programming are very favourable to the engineering community: programs with higher chances to have fewer bugs, with concise implementations and with more focus on the mathematical aspects of the algorithm.

Keywords: Functional programming, electromagnetic transient analysis, Haskell, programming languages.

Acronyms and Abbreviations

CDA	Critical Damping Adjustment
CSV	Comma-separated values
ETR-P	Electromagnetic Transient Program
GHC	Glasgow Haskell Compiler
GHCi	Glasgow Haskell Compiler interactive
GUI	Graphical User Interface
I/O	Input/Output
JSON	JavaScript Object Notation
PoC	Proof of Concept
THTA	Trapezoidal History Term Averaging
UNIFEI	Universidade Federal de Itajuba
UBC	University of British Columbia
UI	User Interface

List of Tables

List of Figures

Listings

1 Introduction

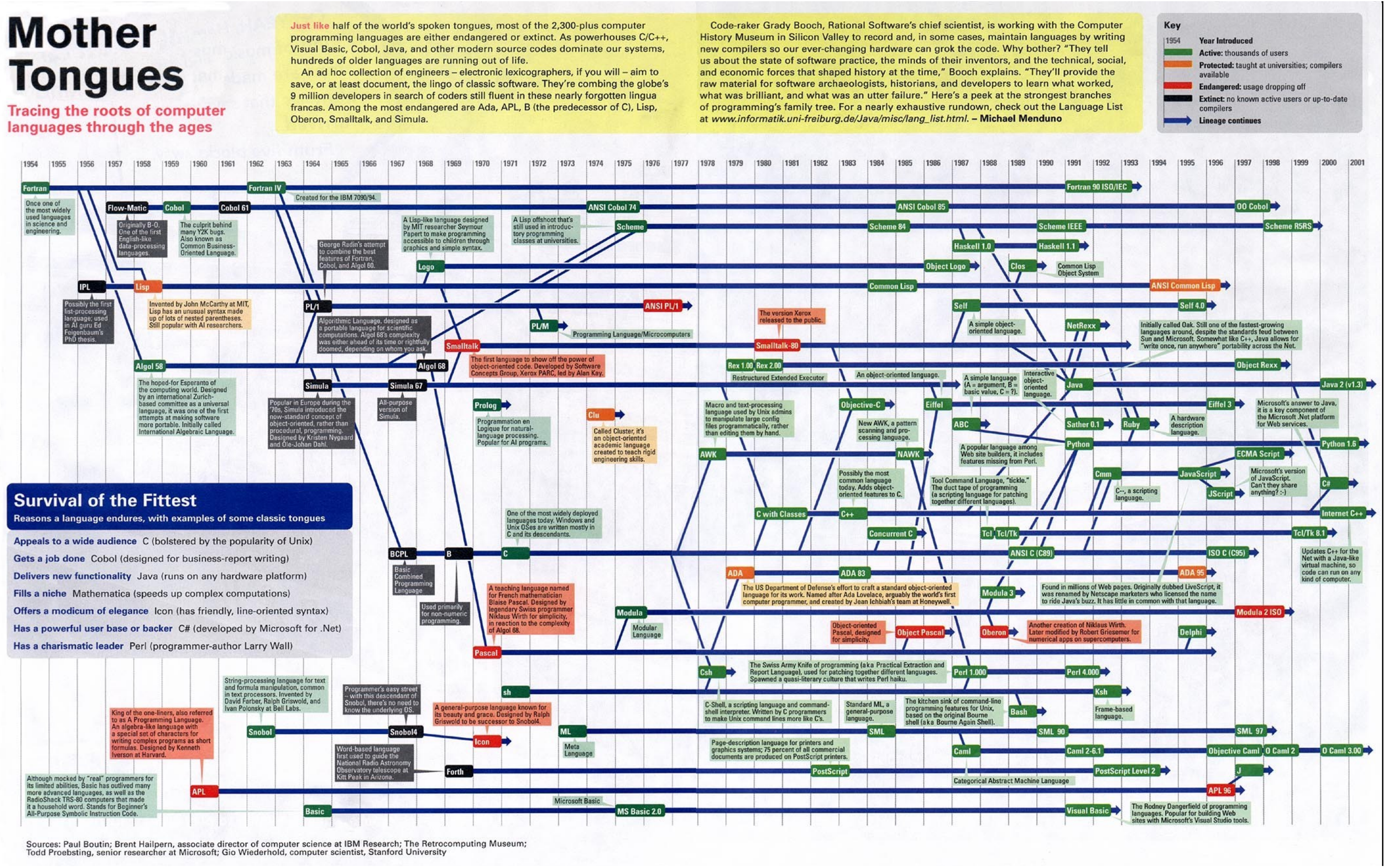
1.1 Motivation

Computer-aided applications play a crucial role in engineering. Since the 1950s, many engineering projects rely on software to execute simulations and analysis of a wide variety of domains. Engineering (Civil, Mechanical, Electrical) has a heavy focus on mathematical models, time optimisations and new applications for modern techniques. Nevertheless, in many situations, engineers of these domains do not spend much time investigating a vital tool at the development of computer software: programming languages.

One of the earliest and most concise definitions of programming languages comes from [1].

> "[A Programming Language] is considered to be a set of characters and rules for combining them which have the following characteristics: (1) machine code knowledge is unnecessary; (2) there is good potential for conversion to other computers; (3) there is an instruction explosion (from one to many); and (4) there is a notation which is closer to the original problem than assembly language would be".

The core of software engineering for industrial applications started with the Assembly language, moving towards more structured languages like Fortran, Algol, Cobol, PL/I, Basic, Pascal, C, Smalltalk, Prolog, C++, Matlab, and more recently, Python, R and Java [2]. However, it is possible to find a much broader spectrum of programming languages as illustrated in Figure 1.1 (source [3]).



Mother Tongues

Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers – electronic lexicographers, if you will – aim to save, or at least document, the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *www.informatik.uni-freiburg.de/Java/misc/lang_list.html.* – **Michael Menduno**

Key

1954	**Year Introduced**
🟩	**Active:** thousands of users
🟧	**Protected:** taught at universities; compilers available
🟥	**Endangered:** usage dropping off
⬛	**Extinct:** no known active users or up-to-date compilers
➡	**Lineage continues**

Survival of the Fittest

Reasons a language endures, with examples of some classic tongues

Appeals to a wide audience C (bolstered by the popularity of Unix)
Gets a job done Cobol (designed for business-report writing)
Delivers new functionality Java (runs on any hardware platform)
Fills a niche Mathematica (speeds up complex computations)
Offers a modicum of elegance Icon (has friendly, line-oriented syntax)
Has a powerful user base or backer C# (developed by Microsoft for .Net)
Has a charismatic leader Perl (programmer-author Larry Wall)

1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
Fortran
Once one of the most widely used languages in science and engineering.
Fortran IV
Created for the IBM 7090/94.
Fortran 90 ISO/IEC
Flow-Matic
Originally B-0. One of the first English-like data-processing languages.
Cobol
The culprit behind many Y2K bugs. Also known as Common Business-Oriented Language.
Cobol 61
ANSI Cobol 74
ANSI Cobol 85
OO Cobol
A Lisp-like language designed by MIT researcher Seymour Papert to make programming accessible to children through graphics and simple syntax.
Logo
Object Logo
A Lisp offshoot that's still used in introductory programming classes at universities.
Scheme
Scheme 84
Scheme IEEE
Scheme R5RS
Haskell 1.0
Haskell 1.1
Clos
Common Lisp Object System
IPL
Possibly the first list-processing language; used in AI guru Ed Feigenbaum's PhD thesis.
Lisp
Invented by John McCarthy at MIT, Lisp has an unusual syntax made up of lots of nested parentheses. Still popular with AI researchers.
Common Lisp
ANSI Common Lisp
George Radin's attempt to combine the best features of Fortran, Cobol, and Algol 60.
PL/1
ANSI PL/1
PL/M
Programming Language/Microcomputers
Self
A simple object-oriented language.
Self 4.0
Algorithmic Language, designed as a portable language for scientific computations. Algol 68's complexity was either ahead of its time or rightfully doomed, depending on whom you ask.
Smalltalk
The first language to show off the power of object-oriented code. Developed by Software Concepts Group, Xerox PARC, led by Alan Kay.
The version Xerox released to the public.
Smalltalk-80
NetRexx
Initially called Oak. Still one of the fastest-growing languages around, despite the standards feud between Sun and Microsoft. Somewhat like C++, Java allows for "write once, run anywhere" portability across the Net.
Algol 58
The hoped-for Esperanto of the computing world. Designed by an international Zurich-based committee as a universal language, it was one of the first attempts at making software more portable. Initially called International Algebraic Language.
Algol 68
Rex 1.00
Rex 2.00
Restructured Extended Executor
Object Rexx
Simula
Popular in Europe during the '70s, Simula introduced the now-standard concept of object-oriented, rather than procedural, programming. Designed by Kristen Nygaard and Ole-Johan Dahl.
Simula 67
All-purpose version of Simula.
An object-oriented language.
A simple language (A = argument, B = basic value, C = ?).
Interactive object-oriented language.
Java
Java 2 (v1.3)
Prolog
Programmation en Logique for natural-language processing. Popular for AI programs.
Clu
Called Cluster, it's an object-oriented academic language created to teach rigid engineering skills.
Macro and text-processing language used by Unix admins to manipulate large config files programmatically, rather than editing them by hand.
Objective-C
Eiffel
Eiffel 3
Microsoft's answer to Java, it is a key component of the Microsoft .Net platform for Web services.
New AWK, a pattern scanning and processing language.
ABC
Sather 0.1
Ruby
A hardware description language.
A popular language among Web site builders, it includes features missing from Perl.
Python
Python 1.6
AWK
NAWK
ECMA Script
Possibly the most common language today. Adds object-oriented features to C.
Tool Command Language, "tickle." The duct tape of programming (a scripting language for patching together different languages).
Cmm
C--, a scripting language.
JavaScript
JScript
Microsoft's version of JavaScript. Can't they share anything? :-)
C#
One of the most widely deployed languages today. Windows and Unix OSes are written mostly in C and its descendants.
C with Classes
C++
Internet C++
Concurrent C
Tcl
Tcl/Tk
Tcl/Tk 8.1
BCPL
Basic Combined Programming Language
B
Used primarily for non-numeric programming.
C
ANSI C (C89)
ISO C (C95)
Updates C++ for the Net with a Java-like virtual machine, so code can run on any kind of computer.
ADA
The US Department of Defense's effort to craft a standard object-oriented language for its work. Named after Ada Lovelace, arguably the world's first computer programmer, and created by Jean Ichbiah's team at Honeywell.
ADA 83
ADA 95
A teaching language named for French mathematician Blaise Pascal. Designed by legendary Swiss programmer Niklaus Wirth for simplicity, in reaction to the complexity of Algol 68.
Modula
Modular Language
Modula 3
Found in millions of Web pages. Originally dubbed LiveScript, it was renamed by Netscape marketers who licensed the name to ride Java's buzz. It has little in common with that language.
Modula 2 ISO
Object-oriented Pascal, designed for simplicity.
Object Pascal
Oberon
Another creation of Niklaus Wirth. Later modified by Robert Griesemer for numerical apps on supercomputers.
Delphi
Pascal
Csh
The Swiss Army Knife of programming (aka Practical Extraction and Report Language), used for patching together different languages. Spawned a quasi-literary culture that writes Perl haiku.
Perl 1.000
Perl 4.000
sh
Ksh
C-Shell, a scripting language and command-shell interpreter. Written by C programmers to make Unix command lines more like C's.
Standard ML, a general-purpose language.
The kitchen sink of command-line programming features for Unix, based on the original Bourne shell (aka Bourne Again Shell).
Bash
Frame-based language.
String-processing language for text and formula manipulation, common in text processors. Invented by David Farber, Ralph Griswold, and Ivan Polonsky at Bell Labs.
Programmer's easy street – with this descendant of Snobol, there's no need to know the underlying OS.
King of the one-liners, also referred to as A Programming Language. An algebra-like language with a special set of characters for writing complex programs as short formulas. Designed by Kenneth Iverson at Harvard.
Snobol
Snobol4
A general-purpose language known for its beauty and grace. Designed by Ralph Griswold to be successor to Snobol4.
Icon
ML
Meta Language
SML
SML 90
SML 97
Page-description language for printers and graphics systems; 75 percent of all commercial documents are produced on PostScript printers.
Caml
Caml 2-6.1
Objective Caml
O Caml 2
O Caml 3.00
Word-based language first used to guide the National Radio Astronomy Observatory telescope at Kitt Peak in Arizona.
Forth
PostScript
PostScript Level 2
Categorical Abstract Machine Language
J
Although mocked by "real" programmers for its limited abilities, Basic has outlived many more advanced languages, as well as the RadioShack TRS-80 computers that made it a household word. Stands for Beginner's All-Purpose Symbolic Instruction Code.
APL
APL 96
Basic
Microsoft Basic
MS Basic 2.0
Visual Basic
The Rodney Dangerfield of programming languages. Popular for building Web sites with Microsoft's Visual Studio tools.

Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

Figure 1.1: Programming languages history summary

Programming languages itself is a dense topic, branching into compilers, static analysis, program synthesis, proofs, concurrency, type theory, logic and many others. Programming languages also have different paradigms [4]. Historically speaking, the industry has been adopting procedural and **imperative** styles during the majority of the time.

Imperative programming is a paradigm that describes computation as statements - they can modify the state of the program. These statements focus on how they should solve the problem proposed in the algorithm, requiring a detailed instruction guide to perform.

There are different programming paradigms apart from the imperative style. Some languages follow the **Functional programming** paradigm - "they are descriptive rather than imperative, have no assignment command and no explicit flow of control - subcomputations are ordered only partially, by data dependency"[5]. The principal component in this alternative paradigm is the application of a function to its arguments, not the computation of statements.

1.2 Research goals

This work proposes the adoption of the functional programming paradigm with a functional programming language to build a simple program reproducing a well-known algorithm for electromagnetic transient analysis of simple electrical circuits. This baseline algorithm uses a didactic program, developed in the MatLab plataform, by students at UNIFEI and UBC (see [6]). It aims to answer the following questions:

1. What are the benefits of using a functional programming language? Will the development process be more intuitive? Will it be possible to apply all the functional programming concepts directly into the application domain?

2. What will be the differences with respect to the code base? Will it be shorter or longer? Will it produce a readable code?

3. What will be the technical challenges? Functional programming is becoming more popular in the industry only in recent years, so there are not many documents and articles available to report challenges during the development process of engineering applications.

4. How and why "functional languages are associated with fewer defects than either procedural or scripting languages"[7]?

The development of the Haskell application will answers the questions previously proposed.

Some of the complementary goals of this project are listed below.

- Create an open-source implementation of a project using a real-world engineering application (electromagnetic transients) applying functional programming.

There are not many projects in electrical engineering using this paradigm and this work can be a basic reference project for future research on similar topics.

- A software developer of electrical engineering applications should care about the tools he/she adopts. The programming language chosen is one of the main tools of the project. The selection of an inappropriate tool leads to bugs and unwanted behaviour. The correctness of the program also relies on the programming language. A professional software engineer will consider this matter when delivering a project.

1.3 Relevance

The future of engineering applications is strictly connected to advances in computer science and software development. Still, these two areas are mostly treated as entirely separate domains. This work aims to build a bridge between programming languages (with a case study on functional programming) and electrical engineering (with electromagnetic transient analysis).

The domain of programming languages research is vast. Analysing the use of functional programming for electrical engineering applications is just a starting point. This work is relevant because it can open doors for several future outputs, such as type analysis focused on the most common engineering models, formal verification of algorithms (increasing the reliability of the delivered software), and so on. Expanding this analysis to a field called Type Theory may lead to exciting results - would it be possible to guarantee that the written algorithm is mathematically equivalent to the engineering model? There are not many publications in this domain yet. Functional programming is the entry point for a more in-depth analysis of this matter.

1.4 Methodology

After a literature review on functional programming, Haskell (the functional language chosen to be the primary tool of this work) and existing applications, a working software containing the proof of concept (PoC) will be delivered and it will be publicly accessible on Github (a Git repository hosting service with free plans).

Once the PoC is done, its results will be compared with the ones produced by Matlab version, validating if the algorithm produces values at least similar to the ones produced by the Matlab version. A comparison between code paradigms will follow the numerical results.

Octave (an open-source implementation of Matlab) is used to run the simulations from the imperative implementation; Stack[8] and Cabal[9] are used with GHC to compile and run the Haskell version.

1.5 Chapters Overview

Chapter 2 gives an overview of the Nodal Analysis in Electromagnetic Transient Analysis, as well as a historical context of the software build for this domain, both in industry and in academia.

Chapter 3 presents the most important concepts of functional programming. It will provide the reader with conceptual examples and possible applications. It uses the actual code from the Haskell program developed in this work.

Chapter 4 applies the functional programming concepts described in the previous chapter to the Haskell language, providing a rich set of practical applications in the language. Haskell is not the only functional language available; there are others like OCaml, SML, Racket. Two separates chapters were kept in order to emphasise that learning functional programming is different from learning Haskell (although it is a necessary to know both for delivering good quality software and the results of this work).

Chapter 5 provides a thorough explanation of the Haskell application. In every section of this chapter, there is a recap of the main algorithm for electromagnetic transient analysis, a walk-through the Haskell code and comparisons with the original didactic program developed in MatLab.

Chapter 6 presents the results of the implementation, numerically comparing the values obtained from complete simulations in both versions, Haskell and Matlab. It then compares the development process for both of the paradigms, functional and imperative.

Chapter 7 answers the questions proposed at 1.2. It also provides an extensive list of future work and a contribution guide for researches interested in the topic.

Future work is also reported on the appendices. They provide the reader with an introduction to Lambda-calculus (A), logic (B) and type analysis (C).

2 Electromagnetic Transients

The simulation and analysis of electromagnetic transients in basic electric circuits are essential to the study of power systems. Electromagnetic transient analysis software can help to determine overcurrent, overvoltage and other unwanted phenomena in power systems. Since the 1970s, computer-aided algorithms have been developed to simulate a wide variety of conditions on complex circuits.

> Despite the powerful numerical techniques, simulation tools and graphical user interfaces currently available, those involved in electromagnetic transients studies, sooner or later, face the limitations of models available in transients packages, the lack of reliable data and conversion procedures for parameter estimation or insufficient studies for validating models ([10])

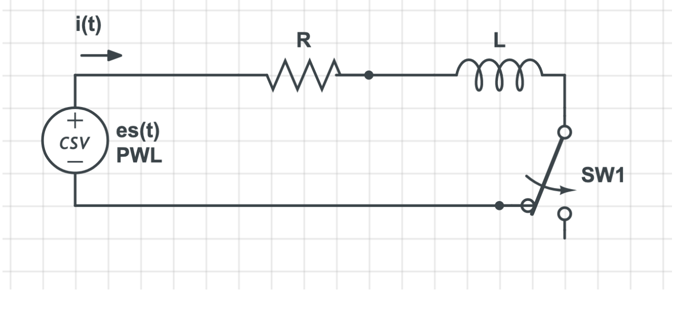
Some of these simulation tools are well-known in industry and academia, such as ATP and EMTP. These two programs operate by creating a representation of the equivalent input circuit and obtaining the simulation results either by state space or nodal analysis. This work is focused on the latter approach.

2.1 Overview of Nodal Analysis

Broadly studied at electromagnetic transients courses and firstly described by H. Dommel in [11], the nodal admittance matrix method is the base algorithm for programs like EMTP and ATP. This representation can deal with both single and multi-phase circuits containing resistors, capacitors, sources and a few other nonlinear elements as well as the discretization of the circuit differential equations through the trapezoidal integration method.

Figure 2.1 illustrates a simple circuit and the continuous time domain first order differential equation and its solution for the current as function of time, considering the time constant (L/R) of the circuit.

Figure 2.1: Transient Analysis - Example Circuit



$$Ri + L\frac{di}{dt} = es(t) \tag{2.1a}$$

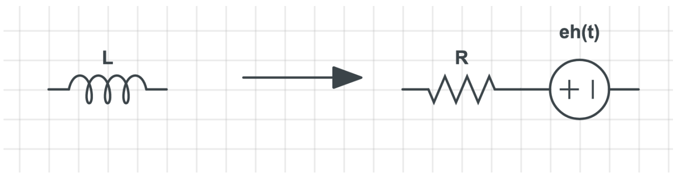
$$i(t) = \frac{E}{R}[1 - e^{\frac{-t}{T}}] \tag{2.1b}$$

$$T = \frac{L}{R} \tag{2.1c}$$

When transforming the equation previously described in an algorithm, it is necessary to adopt discrete intervals of time Δt, since digital computers are not able to simulate the idea of continuous-time.

Assuming the single-phase case, [11] describes the discrete representations for the inductors and the capacitor components, based on the trapezoidal numerical method integration:

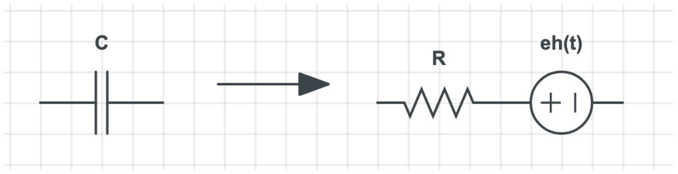
Figure 2.2: Inductor - equivalent discrete representation



$$R = \frac{2L}{\Delta t} \tag{2.2a}$$

$$eh(t) = -V(t - \Delta t) - \frac{2L}{\Delta t}i(t - \Delta t) \tag{2.2b}$$

Figure 2.3: Capacitor - equivalent discrete representation



$$R = \frac{\Delta t}{2C} \tag{2.3a}$$

$$eh(t) = V(t - \Delta t) + \frac{\Delta t}{2C}i(t - \Delta t) \tag{2.3b}$$

Other numerical methods can be taken into consideration. Each distinct numerical method can generate different equivalent values for the limped L and C components. The trapezoidal integration method has been used in academic and commercial EMTP-based programs mainly due to its accuracy and stability properties. Analysing the advantages or disadvantages of each method is out of the scope of this work.

2.2 EMTP algorithm

Into EMTP or ATP software, the discrete models previously described compose a matrix equation in the following format:

Figure 2.4: Nodal matrix model

$$[G][V] = [h] \tag{2.4a}$$

At 2.4, G is the matrix of conductances, V is the vector of unknown voltages and h is the vector of known current sources (independent current sources and the historical equivalent current sources).

It is possible to state a general algorithm [11] in electromagnetic transients analysis according to the following procedures:

1. Read the input data;
2. Build the steady-state solution matrix;
3. Build the transient solution matrix (G);
4. Find the steady-state solution to initialise histories;
5. Assume $t = \Delta t$;
6. Evaluate current and voltage sources;
7. Solve for node voltages;
8. Update history functions;
9. Increment t: $t = t + \Delta t$;
10. Go back to item 6 if the maximum simulation time hasn't been reached.

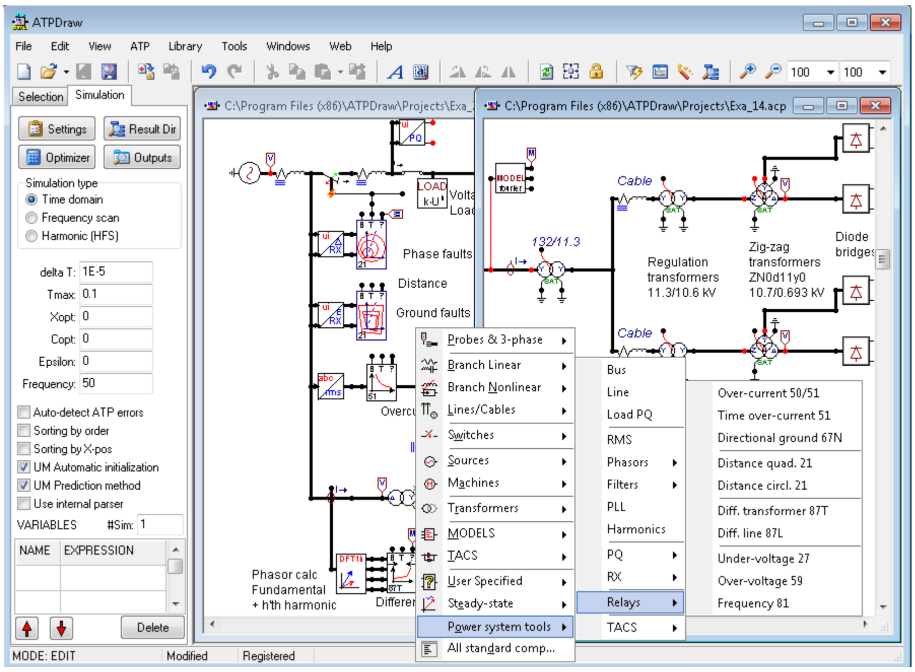
2.3 Existing commercial software

In this section, an overview of the two major programs used for electromagnetic transient analysis is presented. Both are written in imperative languages (Fortran, C and C++). Imperative programming stands for a programming paradigm that uses statements which can change a program's state. Chapter 3 brings a detailed explanation of this matter.

2.3.1 ATP

Written mostly in Fortran language, the Alternative Transient Program (ATP) [12] became popular in the 1970s and 1980s. Originally, the user had to insert the circuit data in a card, as well as the simulation parameters, and then run the algorithm. The code is not open source, and even though it is possible to download the software for free, there are some restrictions on its license:. "Licensing to use ATP is free of all charge for all who have not engaged in EMTP commerce" [13].

Modern versions of ATP support a rich GUI, named ATPDraw. See an example at Figure 2.5.



Simulation type
(● Time domain) Frequency scan Harmonic (HFS)

delta T: 1E-5
Tmax: 0.1
Xopt: 0
Copt: 0
Epsilon: 0
Frequency: 50

☐ Auto-detect ATP errors
☐ Sorting by order
☐ Sorting by X-pos
☑ UM Automatic initialization
☑ UM Prediction method
☐ Use internal parser

VARIABLES #Sim: 1

NAME	EXPRESSION

MODE: EDIT Modified Registered

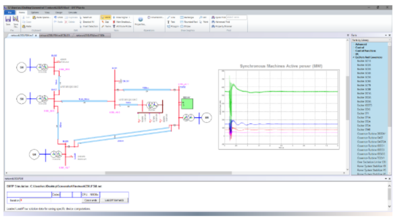
Figure 2.5: ATPDraw

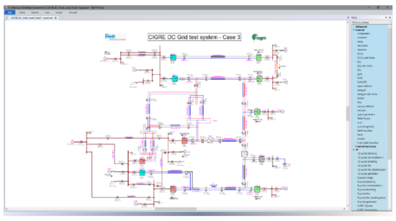
[14]

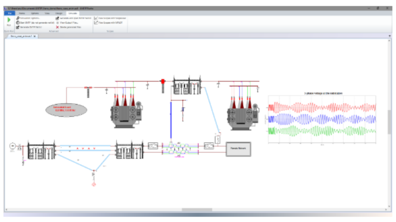
ATP supports a wide variety of nonlinear components, switches, transformers and even support for the creation of customised elements. It can be used for time and frequency domain studies, lightning studies, electrical machines, control systems and power electronics projects.

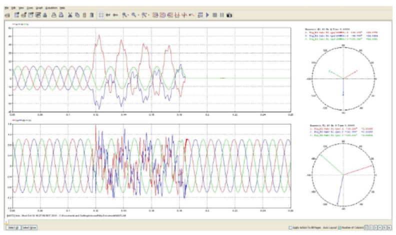
2.3.2 EMTP-RV

EMTP-RV [15] is mostly written in C and C++. It was released on the market a few years after ATP. It is also a closed-source software, and it has a paid license. It is possible to obtain a free trial for a few days. It also comes with a sophisticated GUI, advanced machine models, transformer models which include magnetic core saturation and hysteresis, extensive library of control devices and functions and more.


Simulation of single-phase fault in an unbalanced 230kV Network. Simulation is automatically initialized from EMTP Unbalanced multi-phase load-flow results.


CIGRE DC GRID Test System. See the following reference for details: "EMT simulation of the CIGRE B4 DC Grid test system" S. Dennetière H. Saad RTE France. 2014 CIGRE Canada Conference


Ferroresonance in an industrial plant


ScopeView is an advanced tool for visualization and post-processing of data

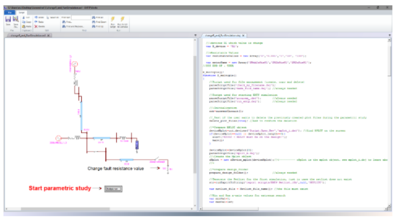

Example of parametric studies performed using EMTPWorks scripting capabilities (JavaScript)

Figure 2.6: EMTP

EMTP implements, among other optimisations, the Critical Damping Adjustment (CDA) algorithm. "The CDA procedure eliminates the numerical oscillations that can occur in transients simulations that use the trapezoidal rule of integration"[16].

2.4 Academic (didactic) software

Commercial versions are suitable for production projects, but for the learning process of undergraduate and graduate students, they might not be ideal, since their source code is often unavailable. Building a didactic prototype helps to understand the ideas behind the algorithms, possible optimisations and problems (numerical oscillations, errors, processing times). It also provides the students with a better understanding of the model and of the approach of nodal analysis method.

2.4.1 ETR-P Matlab

Educational tool developed by a joint work from UNIFEI and UBC, this simplified Matlab/Octave version of an electromagnetic transient analysis program reads the user input from a file (in a similar format as ATP's cards) and executes the simulation. It can plot charts and it offers support to Transmission Lines, Switches and other nonlinear elements. Single-phase analysis only.

This academic project implements the CDA algorithm, as well as an equivalent technique named by the authors of [17] as **"THTA"** - Trapezoidal History Term Averaging.

This project does not have an official name; the students refer to it simply as "THTA", but the nomenclature may lead to misinterpretation of what is the academic tool and what is, in fact, the Trapezoidal History Term Averaging optimisation. For this particular reason, this work refers to UNIFEI's implementation as **ETR-P** (Electromagnetic Transient Program).

ETR-P's source code is open, and it is possible to find it on Github [6]. It is not actively maintained and the code is purely procedural and imperative. It is restricted to the Matlab/Octave environment.

2.4.2 ETR-Py - Python

Inspired by ETRP-P, ETR-Py [18] implements a subset of features from ETR-P using the Python Language, this time using Object Oriented programming, classes, modules and popular libraries such as NumPy and SciPy. The code is also available at Github - [19]. The features for chart plotting, Transmission Lines and Switches are not implemented yet.

When this Python version was proposed, it used to be called "PyTHTA". Once again, to avoid misunderstandings with the term "THTA", this work replaces the former name with **ETR-Py**.

2.4.3 Comparison of the software alternatives

Up to this point, only imperative code has been used to build a program to run an electromagnetic transient analysis simulation. There are other programming paradigms, and this work adopts one of them (**Functional Programming**) to create a new version of ETR-P using an utterly distinct programming style. The elaboration of this other program, named **Haskell ETR-P**, will make a robust comparison between advantages and disadvantages of each programming style possible. The code is also be open source and it is already on Github [20].

The Haskell version of ETR-P aims to provide a solution where students can focus on the algorithm itself, and not on the "language plumbing" (particularities of a programming language that are required to make a program work properly but demand time and expertise to get it right). The existing academic versions mentioned at 2.4.1 and even the more recent 2.4.2 are subjected to the existing problems of the imperative programming paradigm. The new approach with functional programming may help the students to use more time thinking about the algorithm and new ways to improve it.

3 Functional Programming

Functional programming is not an entirely new concept. The nomenclature was solidified in the 1970's by J. Backus [21]. In his work [22] (published in 1990), J. Hughes summarizes the main benefits of this style by comparing pieces of code in Miranda (a functional language) with structural code. The present work will follow a similar approach. The upcoming sections will introduce and compare the main tools of functional programming. In order to fully understand the ideas, it is necessary to put aside the previous knowledge of the imperative style.

> Functional programming is so-called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions ([22])

In other words, "the primary role of the programmer is to construct a function to solve a given problem" [23] and "the primary role of the computer is to act as an evaluator or calculator".

There are many other languages labelled as *functional* languages. Besides Haskell and Miranda, it is possible to mention SML, Hope and, more recently, Scala and Clojure. There are also languages which are popular among the imperative style, but also support some aspects of functional programming. This is the case of Javascript and Python.

Some functional languages are considered to be "pure": "A language is purely functional if (i) it includes every simply typed λ-calculus term, and (ii) its call-by-name, call-by-need, and call-by-value implementations are equivalent (modulo divergence and errors)"[24]. Nevertheless, this definition is not strict. In the group of "purely functional languages", it is possible to cite Haskell and SML.

Even if the definition of "purely function" is not well established, some other topics are well-agreed among the academic community when it comes to characteristics of the functional programming paradigm. They will be listed in the upcoming sections, following the summary proposed by [25].

3.1 First class functions

Passing functions as arguments to other functions or returning functions as the result of an operation are perfectly valid steps in functional programming. Functions are interpreted with the same connotation of mathematical functions. That means it is conceivable to have several definitions for the same function:

$$f(x) = x + x \tag{3.1a}$$

$$f(x) = 2x \tag{3.1b}$$

Listing 3.4 shows an example of a simple function in a functional language.

```haskell
double x = x+x
```

Listing 3.1: A simple function in a functional language

The $=$ operator is not binding any variable; it is actually performing the same mathematical function one finds when declaring a function. In languages like C++ or Python, it is reasonable to have a code like the following:

```
a = 1
a = a + 10
```

Listing 3.2: Mutable references

The code on listing 3.2 changes the value of the variable `a` from 1 to 11. Such a concept is not allowed in a functional language. The `=` operator attaches some value to a variable, and in a functional setting, it becomes immutable. A **value** is a fully simplified expression. Every time a new binding occurs, a new environment for the bound variables is created. A **function** corresponds in some sense to the methods in C++ or Python, but it also holds the mathematical foundation mentioned earlier. Analysing the listing 3.2 under mathematical terms would lead to an inconsistency.

It is also feasible to vary the body of the function according to the domain (or, interpreting it as functional programming, the input parameters) 3.2.

$$f(x) = x, x \geq 0 \tag{3.2a}$$

$$f(x) = -x, x < 0 \tag{3.2b}$$

It is possible to reproduce the exact same structure in functional programming by using **pattern matching**. In the main application of this work, for example, it is necessary to calculate an integer value (`thtactl`) control for determining the final result of a current vector (in the Haskell ETR-P implementation). Simplifying the process to make the usage of pattern matching more explicit, imagine the following scenario:

```haskell
thtaControl :: Int -> [Float]
thtaControl thtactl
  | thtactl <= 0 = [0.0, 0.0]
  | thtactl < 3 = [1.0, 1.0]
  | otherwise = [2.0, 2.0]
```

Listing 3.3: Pattern matching with guards. This does not correspond to the original implementation

The `[Float]` represents a list of Float numbers. In the code at listing 3.3, `thtactl` is the input argument of the `thataControl` function. Depending on its value, the behaviour of the function changes. This structure avoids the confusion of nested `if/else` blocks.

3.2 Strong type system

It is possible to organise values in collections called **types**. There are **primitive** or **base** types (boolean, integer, float, double, etc.), and **compound** or **derived** types, whose values are build from other types (basic or compound) [23]. A function is called with arguments and produces a value. When invoking a certain function (think of it as an operation), it is necessary to guarantee the call with the appropriate types of parameters. For example, calling the function `add` with characters might not be a valid operation. The languages have mechanisms to check the validity of these types. One way to perform this **type check** is to have a compiler that captures mismatched data.

No runtime errors arise from type mismatches. This behaviour originates the label 'strong': "the domain and codomain of each function is either stated in or inferable from the program text, and there is a syntactic discipline which prevents a function from being applied to an inappropriate argument" [26].

3.3 Polymorphic types

Functions also have a type. For example, the listing 3.3 has type `(Int -> [Float ↪ ])`, because it receives an integer and returns a list of float numbers. In some languages, it would be possible to generalise these types. Both integers and floats are numbers, allowing the programmer to determine that the type of the function could also be `(Num -> [Num])`. The study of polymorphic types has several topics, but it is reasonable to think about when it is important to have generalisations. For example, to create a function that measures the `length` of a list, it would be possible to consider the type `([Int] -> Int)`. But then the function `length` would be restricted to lists of `Int` only, which is not ideal. It is feasible to generalise the domain of the function `length` to a **Polymorphic type**: `([a] -> Int)`.

3.4 Algebraic types

When developing a function, it is necessary to think of its return type, the type of its parameters and its body. For example, to build a `simulation` function for Haskell ETR-P, the input would be a list of components. For the output, a Vector `I` together with a Matrix `V` would be the expected:

```haskell
thtaSimulation :: [ComponentData] -> (Vector Double, Matrix Double)
```

Listing 3.4: Haskell ETR-P simulation function

The `[]` denotes a list, as mentioned previously. A List can have any length, but all its elements must have the same type - that is why the type declaration consists of `[ComponentData]`, making it explicit it will be a list of ComponentData. Another structure that requires attention is the **Tuple** - they support different types in the same structure - for example, `(Vector Double, Matrix Double)` holds a Vector and a Matrix.

A function is a value: it will be type-checked and evaluated. Just like in C++ or Python, a function can call itself, being labelled as a **recursive function**. It must contain a base case (which will indicate when to stop the process) and a recursive case (which will propagate the recursive call).

Another crucial concept are `let` expressions. They allow the creation of local bindings, often required by recursive function calls. See listing 3.5.

```haskell
diagonalUpdate :: Int -> Matrix Double -> Double -> Matrix Double
diagonalUpdate d buffer gkmHead =
  let
      updated = (Matrix.getElem d d buffer) + gkmHead
  in
      Matrix.setElem updated (d, d) buffer
```

Listing 3.5: let expression

The `let` expression holds the local binding of `updated`, which computes an intermediate value to be used at the function call triggered after the `in` keyword. `updated` cannot be used outside of the `let` block.

For most of the applications, the basic types (Integer, Double, Float, Boolean, etc) in the standard library are not enough to express the goals and operations of the program. The same way one can define classes in object oriented programming, it is also possible to define custom types in functional languages. It is reasonable to aggregate several **fields** in a **Record** just as in listing 3.6.

```haskell
{first = "Hanneli", last = "Tavante", course="EE"}
```

Listing 3.6: Record

A more structured idea than a record originates a **Datatype** as seen in listing 3.7

```haskell
data ComponentData =
  ComponentData {
    componentType :: ComponentType,
    nodeK :: Int,
    nodeM :: Int,
    magnitude :: Double,
    param1 :: Double,
    param2 :: Double,
    plot :: Int
     }

data ComponentType = Resistor | Capacitor | Inductor | EAC | EDC
```

Listing 3.7: ComponentData datatype

The custom `ComponentData` carries several fields: componentType (whose type is also a custom type), nodeK, nodeM, magnitude, param1, param2 and plot.

Another example of DataType is the `ComponentType`. The constructors do not have any parameters (Resistor, Capacitor, etc). To obtain the component type of a component data, a set of `if`/`else` blocks would be required (when thinking according to the imperative programming style). In a functional setting, there is **Pattern Matching**. The code in listing 3.8 provides another example of this feature, this time using a **case** expression:

```haskell
condutance :: ComponentData -> Double -> Double
condutance component dt =
  case componentType component of
    Resistor -> 1.0 / (magnitude component)
    Capacitor -> (magnitude component) * 0.000001 * 2 / dt
    Inductor -> dt / (2 * 0.001 * (magnitude component))
    _ -> 0.0
```

Listing 3.8: Pattern matching example with case

It is also possible to work with pattern matching when declaring the function's body as seen in listing 3.9:

```haskell
buildIVector :: [ComponentData] -> Vector Double -> Vector Double -> Vector
    ↪ Double
buildIVector [] _ iVector = iVector
buildIVector (component:cs) ih iVector =
-- recursive case omitted
```

Listing 3.9: Pattern matching in functions

`buildIVector` can be pattern matched against its own parameters. Its behaviour changes when it receives an empty list (meaning the end of the recursion) or when it receives a non-empty list, denoted by `(component:cs)`. The underscore means the value does not have any influence on that pattern matching format.

3.5 Modularity

There are systems of modules which can be attached to projects, making the development of larger systems easier and more organised. For the developed Haskell ETR-P, for example, a few modules were required to build the project as shown in listing 3.10.

```haskell
-- vector
import Data.Vector (Vector)
import qualified Data.Vector as Vector

-- Matrix
import Data.Matrix (Matrix)
import qualified Data.Matrix as Matrix

-- HMatrix
import qualified Numeric.LinearAlgebra.Data as HMatrix
import qualified Numeric.LinearAlgebra.HMatrix as HMatrix

-- bytestring
import Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString
```

Listing 3.10: Modules

In listing 3.10, it is possible to find the Vector, Matrix, HMatrix and ByteString modules. It is plausible to import them to the main project assigning an alias. For example, `import qualified Data.Matrix as Matrix` adds the alias `Matrix` to the module `Data.Matrix`.

3.6 Fucntional Programming - summary

"Functional programming can be contrasted with imperative programming either in a negative or a positive sense"[27]. Combining the "pros" and "cons" listed at [27] and at [28], it was possible to built the table 3.1, which aims to be impartial and summarise an objective and concise comparison of the main differences between these two paradigms:

	Imperative Paradigm	**Functional Paradigm**
Main component	Statements	Functions (can be treated as values)
Assignments	The same name may be associated with the same value	No assignments. A name is only associated with one value
Execution order	Crucial importance	Not very important
Mutability	Yes, most of the time the structures are mutable	Usually not present
State changes	Essential building block	No state
Flow control	Loops, if/else, exceptions	Recursion, pattern matching, IO

Table 3.1: Comparison: Imperative and Functional paradigms

4 Haskell

Haskell is a functional language developed in the 1990s. Designed to be lazy, pure and to support type classes [29], the language has been evolving since its creation. This chapter will provide examples of some of the functional programming concepts introduced in Chapter 3 using the Haskell language. This chapter is not a "Haskell Tutorial"; its primary goal is to present the concepts of functional programming that will be used in the practical section and, therefore, will be one of the central objects of study of this work. The fundamental ideas of the next section come from [30], and from [31] (course at TU Delft); those are also good references to get started into Haskell programming.

The development of the Haskell language started in 1987 [29], but its foundations started a few decades before, with Alonzo's Church Lambda Calculus. In the 1970s, John Backus inaugurated the term *Functional style* in his work [21], motivating a meeting of programming language researches ten years later. This group officially started the works on the Haskell language.

The principal features of the Haskell language [30] are listed below:

- Concise programs - Haskell's syntax is designed to be compressed and to have just a few keywords. *List Comprehensions*, for example, are a concise way to navigate on a list, create new ones, filter elements or transform the existing ones. No need for explicit loops (like `while`) or nested if/else blocks.

- Type system - Haskell's type system helps programmers to detect incompatibility errors at compile time (the functions often require the expected input and output types). Haskell supports polymorphism and overloading, providing the developers with a wide range of features.

- Recursive functions - One and probably the most important way to navigate through a collection of elements is using **recursion**. To determine when to stop or to continue, recursion can be combined with **pattern matching**.

- Higher-order functions - Functions can receive other functions as arguments, as well as return them as values. A **higher order function** abstracts the behaviour of a group of functions, wrapping it up in another function.

- Effectful functions - A function is called **pure** when it produces the same output given the same inputs, without any interference. In some circumstances, there may be *side effects*, causing unexpected outputs. Operations that involve files are a good example of impurity. If a function requires a file, it may or may not be present, causing the function to have multiple outputs for the same inputs. It will be an **impure** function. Haskell provides mechanisms to deal with this impurity scenario: *monads* and *applicatives.*

- Lazy evaluation - In Haskell, a computation is not performed until it is required by another function. This mechanism avoids unnecessary computations and allows the usage of infinite structures, such as infinite lists.
- Equational reasoning - Combined with induction, it can be used to transform, execute and demonstrate properties of Haskell programs.

4.1 Function and Types in Haskell

Recall that one of the aspects of functional programming is having a style closer to a mathematical representation. In this context, `f x` in the Haskell language corresponds to the mathematical function application $f(x)$. In other words, an empty, blank space represents a function application.

4.1.1 Basic types

Another topic that arises in functional languages are **types**: they are a collection of related values, as seen in listing 4.1.

```
v::T -- v has type T
e::T -- The evaluation of the expression e will produce a value of type T
```

Listing 4.1: Types for variables and expressions

In Haskell, every expression must have a type. For this language, it is possible to either manually define the types or to have them inferred using the **type inference** process. This step happens before the proper evaluation, so if there is anything that does not match the possible type (for example, adding a string with an integer), the code will not compile. Making types explicit when writing functions is a good practice in Haskell. Mathematically speaking, this is shown in fig. 4.1:

Figure 4.1: Transient Analysis - Example Circuit

$$\frac{f :: A \to B \qquad e :: A}{fe :: B}$$

Figure 4.1 states that if there is a function **f** that maps an argument of type **A** to a result of type **B** and **e** is an expression of type **A**, the application of the function **f** will have the type **B**.

Haskell programs are labelled as *type safe* because the compiler guarantees the absence of type errors during run time. One downside about this feature is that sometimes a valid expression may be misinterpreted during the type inference process, causing it to reject a correct expression.

Some examples of built-in types in Haskell are Int, Float, Double. There are also more sophisticated built-in types in Haskell - Lists, collections of elements of the same types; and Tuples, finite sequences of components, which can be of different types. An example can be found in listing 4.2.

```
[1,2,3] -- A List of Integers
("Hanneli","UNIFEI",2019,True) -- A Tuple of arity four (four elements)
```

Listing 4.2: Lists and Tuples

Conceptually, Lists require a specification of the type they will convey. It is reasonable to have lists of Integers, Floats and Strings, for examples. Generalising, there is the type `List a`, where **a** is a variable to indicate a type. A type (in this case, **List**) that contains one or more type variables (in this case, **a**), is a **polymorphic type**.

A **List** like [1,2,3] is equivalent to `(1:(2:(3:[])))`. From this alternative representation it is possible to see that lists are recursively compounded. Initially, there is the empty list `[]`, and then 3, then 2, then 1 are appended. In functional languages, recursion is very important to compose several data structures. Recursion is also the basic looping mechanism. In Java or C++, it is common to find a **for** loop accessing items of collections by indexes. In contrast, Haskell relies on recursion to perform operations across lists. For example, multiplication can be defined as a recursive sum as shown in listing 4.3.

```
mult _ 0 = 0 -- Base case
mult n m = (mult n (m - 1)) + n -- Recursive call
```

Listing 4.3: Recursion

Given that a function can receive other functions as arguments and return other functions, a function is itself a type. That means functions receiving multiple parameters can be rewritten as functions that receive a single parameter and return another function. This is called **currying**. For example, when declaring a function `adds` to add two integers, the function type can be defined as something that receives two integers and returns another integer (`adds :: Int -> Int -> Int`) or as a function that takes only one integer and returns another function that takes the second integer and returns the result: `adds :: Int -> (Int -> Int)`. In the previous example, the parentheses are optional, since the `->` (named 'arrow function') is interpreted as right-associative. So `Int -> Int -> Int -> Int` is the same as `Int`
`↪ -> (Int -> (Int -> Int))`.

However, function application (denoted by a simple space character `" "`) associates to the left. So `adds x y z` means `((adds x)y)z`.

In Haskell, the concept of *class* differs from the idea of the object-oriented languages. A class is a collection of types that supports overloaded operations (*methods*). For example, *Equality* is a class that contains a method `(==)` under the type signature of `(==):: a -> a -> Bool`, that determines if an element of the collection of types `a` is equal to another of the same type, a Bool result must be returned. Ordering (`ord`), `Show`, `Read`, Numerical (`Num`) and `Fractional` are other examples of Haskell classes.

4.1.2 Function structure and resources

Haskell supports `if`/`else` blocks (example: `if n == 0 then 0 else 1`), but it is much more common to use *guard* expressions (see listing 3.3) or *pattern matching* (see listing 4.4):

```
condutance :: ComponentData -> Double -> Double
condutance component dt =
  case componentType component of
    Resistor -> 1.0 / (magnitude component)
    Capacitor -> (magnitude component) * 0.000001 * 2 / dt
    Inductor -> dt / (2 * 0.001 * (magnitude component))
    _ -> 0.0
```

Listing 4.4: Pattern matching with case

It is possible to pattern match against Lists, Tuples and other types.

It is feasible to define auxiliary, nameless functions. They are called **lambda** expressions and a similar concept is present in some object-oriented languages such as Python. In listing 4.5, there is a lambda expression right after `map`. This anonymous function specifies one should be added to the value of each component of a collection. \ indicates the declaration of a lambda expression. The name 'lambda' expression originates from λ-Calculus (visit appendix A for more information).

```
map (\r -> (r + 1)) [1, 2, 3]
```

Listing 4.5: Lambda functions

`map` applies another function to each element of a list. For example, to add 2 to all the elements of `[1, 2, 3]`, one can invoke the `map` function as in `map (+2)[1, 2, 3]`.

4.1.3 Higher order functions

Higher-order functions allow common programming patterns to be wrapped in a function. To build the intuition on this topic, it is reasonable to sketch an example with the `fold` function. "Folding" a list means to compress it following a certain operation. For example, `foldl (+)0 [1, 2, 3]` returns 6; the 0 is the initial buffer. + is the operation, and $0 + 1 + 2 + 3$ is the result of `fold`. Alternatively, `foldl (+)1 ↪ [1, 2, 3]` returns 7, because the initial accumulator is 1.

"`fold` takes an 'initial answer' `acc` and uses [a function] `f` to 'combine' `acc` and the first element of the list, using this as the new 'initial answer' for 'folding' over the rest of the list"[32]. A collection can be folded to the left with `foldl` or to the right with `foldr`. The listing 4.6 describes the implementation of a left fold, named as `foldl2`, since `foldl` is the name of the funciton in the standard implementation of the Haskell language.

```
foldl2 f v [] = v
foldl2 f v (x:xs) = foldl2 f (f v x) xs
```

Listing 4.6: fold left

The type signature of `foldl` is given by listing 4.7:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Listing 4.7: fold left type signature

Functions can be composed the same way they are composed in mathematics. The higher-order composition function (denoted by the operator `.`) allows the function composition $f \circ g$ with the syntax `f . g` .

4.1.4 Composite types

It is possible to define custom types in Haskell by using the keyword **data**. The listing 4.8 is an example of a new data called Shape, which has two possible constructions: it can be a Circle or a Rectangle.

```
data Shape = Circle Float | Rectangle Float Float

square::Float -> Shape
square n = Rectangle n n
```

Listing 4.8: Custom types

Rectangle and **Circle** are constructor functions. They exist for building pieces of data, but they do not carry the defining equations. The logic to build the actual rectangle is in the function **square**. This is consistent with the principles of functional programming: the data is isolated, and the basic method of computation to build anything is the application of functions.

The keyword *type* adds an alias to existing types. For example, `type String ↪ = [Char]`, implies that `String` is just a List of `Char`. Type declarations can be parameterised by other types. For example, `type Pair a = (a, a)`.

4.2 Impure functions

One of the key ideas in functional programming is the concept of *pure functions*. They have no side effects (they do not mutate any state of the program or change non-local variables). These functions also return the same value given the same input arguments, just as a regular mathematical function.

However, I/O operations can have side effects. For example, opening a file may succeed or may throw an error, either because the file does not exist or because it is corrupted. Note that this problem has nothing to do with manipulating the information on a file; it is an external problem. Realistically, it is only possible to have a code that expects the possibility of a side effect. Languages like Java deal with this scenario by throwing run time exceptions.

Haskell represents impurity with the `IO` type. Expressions of this type are named as *actions*.

The use case presented in this work requires, from the very first beginning, an action that will generate side effects - reading CSV files with the input data. A function to read and parse the data can have side effects when (a) the file is not in the specified directory; (b) the file does not exist; (c) the file is empty; (d) the file is corrupted. The type of a parsing function must incorporate the possibility of these side effects, and for that matter, two elements are required:

1. an action to change the context from pure to impure (`return` function in listing 4.9);

2. a sequencing operator to apply the parser and call the result, then reapply the parser, get the result and repeat the process until the end of the file is reached. A sequence of actions can be chained with the `do` notation as seen in listing 4.9.

```haskell
do
  eitherSimulation <- decodeSimulationFromFile "data/simulation.csv"
  componentsList <- decodeItemsFromFile "data/components.csv"
  return (eitherSimulation, componentList)
```

Listing 4.9: do notation; Note: this is just a didactic draft example.

4.3 Monads

Haskell supports even more generic sets of functions - some of them can be generalised over a range of parameterised types (lists, trees, IO). For example, it is possible to generalise the idea of `map` to other structures rather than Lists. In the reference implementation of this work, in many scenarios `map` occurs in a `Vector`. This generalisation is seen on listing 4.10:

```haskell
gkm :: Vector ComponentData -> Double -> Vector Double
gkm components dt =
  Vector.map (\c -> condutance c dt) components
```

Listing 4.10: map in a Vector

There is a class of types supporting the `map` function - this is the `Functor` class, with `fmap` as the generic function that maps another function over each element of a structure [30].

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Listing 4.11: Functor class

4.10 is an example of a Functor. Functors work for single-element functions. It is possible to generalise the idea of Functor itself, making it support any number of parameters, as suggested in fig. 4.2.

```haskell
fmap0 :: a -> f a

fmap1 :: (a -> b) -> f a -> f b

fmap2 :: (a -> b -> c) -> f a -> f b -> f c

fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

Figure 4.2: Generalising functors

[30]

Two basic functions are capable of encoding the desired generalisation. They are built using the idea of currying and are described in listing 4.12:

```haskell
class Functor f=> Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Listing 4.12: Applicatives

This pair of functions is called **Applicative Functor** (or simply **Applicatives**). Examples of its usage can also be found in the implementation of Haskell ETR-P in listing 4.13:

```haskell
instance FromNamedRecord ComponentData where
  parseNamedRecord m =
    ComponentData
      <$> m .: "Element Type"
      <*> m .: "Node K"
      <*> m .: "Node M"
      <*> m .: "Value"
      <*> m .: "Source param 1"
      <*> m .: "Source param 2"
      <*> m .: "Plot"

```

```haskell
instance FromField ComponentType where
  parseField "R" =
    pure Resistor

  parseField "L" =
    pure Inductor

  parseField "C" =
    pure Capacitor

  parseField "EDC" =
    pure EDC

  parseField "EAC" =
    pure EAC

  parseField otherType =
    Other <$> parseField otherType
```

Listing 4.13: Using Applicatives

In Haskell, it is possible to encode a possibility of failure in the function's type. The `Maybe` type wraps a success branch (named `Just a`) or a possible failure, represented by the absence of value (represented by `Nothing`). The **bind** operator `>>=` takes an argument of type "a" and a function of type `a -> b`. Both can fail, the argument and the function. `>>=` returns a result of type "b", which can also be a failure (encoded as a Maybe) as shown in listing 4.14:

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
           Nothing -> Nothing
           Just x -> f x
```

Listing 4.14: The bind operator

It is also necessary to have a function that provides a connection between pure functions and actions (impure). This is the **return** function, with the signature defined in listing 4.15.

```haskell
return :: a -> IO a
```

Listing 4.15: The return

The `return` function defined in listing 4.15 has nothing to do with the "return" statements found in languages like C.

A **Monad** arises from the combination of these two elements, `>>=` and `return`. A Monad is an applicative that supports both of these operations. "Every Monad is an applicative functor"[33] in its generalised form, as proposed in 4.2. This work will not explore in detail the powerful constructions built on the monadic style. A complete example of Monad applications can be found on the work of [34]. The code in listing 4.16 and listing 4.17 describe a Monadic parser with the two principal elements of a Monad: the `return` and bind operator `>>=` :

```
newtype Parser a = Parser (String -> [(a,String)])
```

Listing 4.16: Declaring the type Parser

```
instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f = Parser (\cs -> concat [parse (f a) cs' |
    (a,cs') <- parse p cs])
```

Listing 4.17: Declaring the type Parser

4.4 Advanced Features and fields of study

The previous sections provided the basic concepts required to understand and interpret the code in Chapter 5. However, the Haskell language has been evolving continuously and it has advanced topics which are also out of the scope of this work. Just to give a taste to the reader, some of the active research topics involve (but are not limited to): Deep EDSL [35], verification of Haskell programs with type theory [36], constraint solving [37], Monad transformers [38], verification of Haskell programs in proof interactive theorem provers (like Coq [39] or Liquid Haskell [40], see [41] and [42]), program synthesis in Haskell [43], etc.

5 Haskell in Electromagnetic Transient Analysis - Implementation

This chapter describes the development of the principal analysis tool for this work: a functional version of the original ETR-P software rewritten in Haskell. The upcoming sections do not follow the same order of content creation; instead, they try to build a more intuitive connection between the theory of functional programming and a practical aspect of its application when building a program for electromagnetic transient analysis. Each section methodically follows the same basic structure:

- Logical explanation of the electrical process
- Haskell implementation
- Functional programming resources
- Comparison with the implementation in ETR-P using Matlab/Octave. Additional languages, such as Python and C++, will also be targeted for contrasting the different paradigms.
- Improvements and future work

5.1 The goals for a Haskell version of ETR-P

The main goal of this open-source Haskell program is to compare the style differences between functional and imperative code. This version does not contain all the features supported at the original ETR-P. Its scope was limited in order to demonstrate a proof of concept. Non-linear components, switches, transmission lines are not implemented. Triangular voltage sources and external current sources were put aside for the initial implementation. RLC components, DC and sinusoidal elements are the core of this Haskell program. It is restricted to single-phase circuits.

Optimisation, performance, graphics, UI, charts and user input are popular topics in program development, but they are not the goal of this work either. This project is a comparison of programming paradigms and an analysis of the outcomes.

Following H. W. Dommel's original proposition in [11], this program reads in the circuit data, creates the initial steady-state matrix equations for initialization of equivalent historic sources of digital lumped components. Then it solves the nodal equations $[G][V] = [I_h]$ for every time step (where G is the conductances matrix, and I_h are the historical and external current sources), obtaining the final values for nodal voltages, branch voltages and branch currents after the simulation time.

Translating this goal to Haskell, it is necessary to elaborate a function with the type described in listing 5.1. It follows the functional programming principle that says "Programs in Haskell can be viewed as functions whose input is that of the problem being solved, and whose output is the desired result.[...] The behaviour of the function can be understood as [..] computation by calculation". [44]

```
thtaSimulation :: Vector ComponentData -> SimulationData -> SimulationResults
```

Listing 5.1: ETR-P Haskell goal's function

In listing 5.41, the function **thtaSimulation** receives two parameters: the components representing the circuit (**Vector ComponentData**) and the simulation data (**SimulationData**). It returns a tuple with a the final current vector and the final voltage matrix, represented by the type **SimulationResults**.

In ETR-P (Matlab/Octave), there is no single function isolating this compact representation. Instead, there is a single block of code sequentially declaring the steps of the algorithm.

The following section 5.1.1 will describe data acquisition from a set of files.

5.1.1 Data input

The data describing the simulation parameters and the components of the circuit is stored in external files. This part of the code must handle the data and feed the algorithm with the collected information. Just like the implementation of ETR-Py, this the initial circuit status will be stored in `.csv` files. ETR-P (Matlab/Octave) uses a `.txt` file.

The Haskell implementation requires two different files (one for components and another for the simulation parameters). A well-formed CSV does not mix different sources of information, making it inappropriate to store the Time and other information of the simulation along with the components and nodes information. The original ETR-P and also ETR-Py handle this step improperly mixing these two sources of information in the same input file.

Listing 5.2 and listing 5.3 are examples of the input data split in two files.

```
Element Type,Node K,Node M,Value,Source param 1,Source param 2,Plot
EDC,2,0,10,0,0,0
R,2,1,10,0,0,0
L,1,0,1,0,0,0
```

Listing 5.2: Input data file for components

```
Number of Nodes,Number of Voltages Sources,Step Size,Maximum time for simulation
2,1,0.0001,0.05
```

Listing 5.3: Input data file for time and simulation data

Listing 5.4 shows the input file format of the original ETR-P Matlab program:

```
T 2 1 100E-6 50E-3 0 0 0 0 0
EDC 2 0 10 0 0 0 0 0 5
R 2 1 10 0 0 0 0 0 5
L 1 0 1 0 0 0 0 0 5
NV 1 2 0 0 0 0 0 0 0
```

Listing 5.4: Original input data file for ETR-P Matlab

Headers were also inserted on each CSV file to make their columns self-explanatory (listing 5.2 and listing 5.3). These headers are taken into consideration when parsing the files in the Haskell implementation.

File operations are considered impure (details in section 4.2), given that they can generate several unexpected behaviours to a function. Some of the possible are listed below:

1. File path is incorrect;
2. File is corrupted;
3. File is in the wrong format (not a CSV);
4. A well-formed CSV is expected to have data in all the columns, for all the rows. The file might contain missing data;
5. File is in a different encoding (example: file contains characters from a different alphabet rather than the Latin alphabet);

As described in section 4.2, it will be necessary to use Haskell's specific mechanisms to deal with possible side effects.

The first step is building a parser to take the `.csv` file and turn it into proper types. The implementation in this work uses the library **cassava** for handling the parsing. The implementation starts with the `components.csv` file, describing each component and their corresponding position in the circuit (see listing 5.5).

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module Main where

import qualified Data.ByteString.Lazy as BL
import qualified Data.Vector as V
-- from cassava
import Data.Csv

type ComponentData = (BL.ByteString, Int, Int, Float, Param, Param, Int)

main :: IO ()
```

```haskell
14 main = do
15   csvData <- BL.readFile "data/components.csv"
16   let v = decode NoHeader csvData :: Either String (V.Vector ComponentData)
```

Listing 5.5: Initial implementation of CSV file parsing with cassava

The code at listing 5.5 imports the basic modules (from lines 6 to 9), followed by a basic **type** declaration: `ComponentData`. It matches the structure of each line of the CSV file *components.csv*. `ComponentData` is just a synonym to a tuple structure made of `(BL.ByteString, Int, Int, Float, Param, Param, Int)`. Next, in the main body, the file is read with `BL.readFile` and its content is decoded, ignoring the CSV header. The result of reading a file can be either a well-formed data or an error. That is why the `Either` type is necessary: to carry either a successful output (which is called Right) or an error (wich is called Left, by convention).

Even though the code on listing 5.5 achieves its purpose of reading and parsing a CSV file, it is important to improve the expressiveness of the program types. A data type called `ComponentData` is built, where it is possible to store a corresponding `ComponentType` for each type component the project supports (initially starting with a restricted subset of possible elements: Resistors, Inductors, Capacitors, AC and DC sources; this list will be expanded for more components in the future). The project's goal is to make `ComponentData` be the data type that represents each line of the `components.csv` file, as suggested in listing 5.6.

```haskell
data ComponentData =
  ComponentData {
    componentType :: ComponentType,
    nodeK :: Int,
    nodeM :: Int,
    magnitude :: Double,
    param1 :: Double,
    param2 :: Double,
    plot :: Int
     }
  deriving (Eq, Show)

data ComponentType = Resistor | Capacitor | Inductor | EAC | EDC | Other Text deriving (Eq, Show)
```

Listing 5.6: ComponentData and ComponentType declarations

A more sophisticated version of the code takes into consideration the `csv` header, mapping each item to a `ComponentData` field. The function `parseNamedRecord` from *Cassava* library does the job:

```haskell
instance FromNamedRecord ComponentData where
  parseNamedRecord m =
    ComponentData
      <$> m .: "Element Type"
      <*> m .: "Node K"
      <*> m .: "Node M"
      <*> m .: "Value"
      <*> m .: "Source param 1"
      <*> m .: "Source param 2"
      <*> m .: "Plot"
```

Listing 5.7: parseNamedRecord

Cassava is a mature Haskell library for parsing and encoding CSV files. With the support of monads (see section 4.3), it allows both index and name based conversions from text files to proper Haskell types. It has several other features such as customizable record-conversion instance derivation (with Haskell generics), incremental decoding and encoding API and streaming API for constant-space decoding. The full documentation can be found on [45].

Compare the listing 5.7 with the declared header of the *components.csv*:

```
Element Type,Node K,Node M,Value,Source param 1,Source param 2,Plot
```

Listing 5.8: CAV header components file

The code at listing 5.7 applies (`<*>`) to each element (which receives an alias of `m`, executing the role of a temporary variable in an iteration) to the described sequence of header names ("Element Type", then "Node K", then "Node M", etc). Columns with types defined at Haskell's standard library, such as `Int` or `Double`, are directly converted to the fields of `ComponentData`.

It is necessary then to specify how "Element Type" translates to `ComponentType`. `ComponentType` is not a type defined at Haskell's standard library; it is a proper type. `parseField` function must translate the text of an `m` to this custom type `ComponentType`, as shown in listing 5.9:

```haskell
instance FromField ComponentType where
  parseField "R" =
    pure Resistor

  parseField "L" =
    pure Inductor

  parseField "C" =
    pure Capacitor

  parseField "EDC" =
    pure EDC

  parseField "EAC" =
    pure EAC

  parseField otherType =
    Other <$> parseField otherType
```

Listing 5.9: parseField

In this implementation, a `ComponentType` supports the RLC components, EAC and EDC (as specified in section 5.1). If any other string is specified, it is flexible enough for preventing the program to crash if the user accidentally inputs a non-existing Component Type (for example, "H"). The algorithm will ignore the text type with `Other`. A more strict implementation could throw an error in the presence of a field that cannot be converted with `parseField`. It is important to notice that if there are any parsing errors, there will be an error during runtime execution.

Even if some errors can happen during runtime, the application "shields" the code by warning it about problems that can happen. The `IO` type deals with impure functions. In the following example, combined with `Either`, it throws an exception if something does not work as expected while parsing a `csv` file or lets the code move forward if the outcome is appropriated. Using the functional programming style, functions to read and decode a `csv` file are composed. The functions `decodeItems` and `decodeItemsFromFile` summarise these operations. The code in listing 5.10 brings the proper implementations suggested at the official Cassava's guide:

```haskell
decodeItems :: ByteString -> Either String (Vector ComponentData)
decodeItems =
  fmap snd . Cassava.decodeByName

decodeItemsFromFile :: FilePath -> IO (Either String (Vector ComponentData))
decodeItemsFromFile filePath =
  catchShowIO (ByteString.readFile filePath)
    >>= return . either Left decodeItems

catchShowIO :: IO a -> IO (Either String a)
catchShowIO action =
  fmap Right action
    `catch` handleIOException
  where
    handleIOException :: IOException -> IO (Either String a)
    handleIOException =
      return . Left . show
```

Listing 5.10: Decoding a CSV file

The type signature of `decodeItemsFromFile` shows that it receives a `FilePath` and the return of the function may have side effects (not a pure function). This idea is represented with the type `IO`. The result will be `Either` an error message from an exception (described at `catchShowIO`) or a successful `Vector` of `ComponentData` elements.

To acquire the simulation data from the `simulation.csv` file, the exact same process is used. First, the type declaration, shown in listing 5.11:

```haskell
data SimulationData =
  SimulationData {
    nodes :: Int,
    voltageSources :: Int,
    stepSize :: Double,
    tmax :: Double
    }
  deriving (Eq, Show)
```

Listing 5.11: SimulationData declaration

Then, the conversion from text to the data type using the `parseNamedRecord` from `Cassava` library, listed in the code at listing 5.12:

```
instance FromNamedRecord SimulationData where
  parseNamedRecord m =
    SimulationData
      <$> m .: "Number of Nodes"
      <*> m .: "Number of Voltages Sources"
      <*> m .: "Step Size"
      <*> m .: "Maximum time for simulation"
```

Listing 5.12: parseNamedRecord for simulation data

`SimulationData` only contains basic types form Haskell's standard library (`Int` and `Double`). Conversions with `parseField` are not required.

Lastly, the decoding functions are listed in listing 5.13.

```
decodeSimulation :: ByteString -> Either String (Vector SimulationData)
decodeSimulation =
  fmap snd . Cassava.decodeByName

decodeSimulationFromFile :: FilePath -> IO (Either String (Vector SimulationData)
    ↪ )
decodeSimulationFromFile filePath =
  catchShowIO (ByteString.readFile filePath)
    >>= return . either Left decodeSimulation
```

Listing 5.13: Decoding the CSV simulation file

In the original ETR-P, the data input happens as listed in the code at listing 5.14:

```
FILENAME = input('Enter the input file name *.txt : ', 's');
FID = fopen(FILENAME);
data = textscan(FID, '%s %d %d %f %f %f %f %d %d %d');
fclose(FID);
% data contains the input file organized as a table. Now, each column
% should be put in a separated column vector.
type = data{1}; from = data{2}; to = data{3}; val4 = data{4}; val5 = data{5};
val6 = data{6}; val7 = data{7}; val8 = data{8}; val9 = data{9}; plt = data{10};
% bmax is the number of lines in the input file.
bmax = length(type);
%% Calculating parameters for vector and matrix dimensioning
b = 1;
if strcmp(type(b), 'T')
    % Reading time card data
    % N = Number of nodes
    % M = Number of voltage sources
    % dt = step size
    % tmax = maximum time of simulation
    N = from(b);
    M = to(b);
    dt = val4(b);
    tmax = val5(b);
    Damp = plt(b);
    if Damp == 0
        fprintf('-------------------- THTA Activated --------------------\r');
    end
else
    error('Input file error: the first line is not the Time Card.');
end
```

Listing 5.14: Shortened version of ETR-P Matlab code to read the input data

There are no explicit types, and the context of what is going on in the code relies mostly on variable names and code comments. `if/else` blocks deal with malformed files (for ETR-P, files that do not start with 'T', which is the row holding the simulation's parameters, are considered to be malformed and return an error message to the user). No additional checks are made to see if there are components different from the RLC, transmission lines, voltage sources, current sources or switches. These lines would be ignored.

5.1.2 Improvements to the data input

CSV files are convenient and more structured than a pure `txt` files. However, other configurations force the user to adopt a more restrictive format, such as `XML` or `JSON`, reducing the number of problems generated by malformed or illegal input data. `JSON` files are widely accepted in modern software applications. Ideally, the `components.csv` file could be translated in a JSON file similar to the suggestion in listing 5.15:

```
{
  "circuit": "A circuit",
  "time": {

  },
  "nv": {

  },
  "nodes": [
    {
      "from": 1,
      "to": 1,
      "element_type": "R",
      "element_value": 10.5,
      "power_10": 0,
    }
  ]
}
```

Listing 5.15: Input file as JSON

This feature is listed on the project's issue tracker [20].

5.2 Initial Setup

After collecting and parsing the required input data for the simulation, it is necessary to work on the initial setup. Several parameters must be determined. They are listed in the following subsections.

D: number of unknown voltage sources

The number of unknown voltage sources (D) is given by the formula $D = N - M$, where N and M are the number of nodes in the simulation and the number of voltage sources, respectively. They are both obtained from the `SimulationData`. In the Haskell implementation, this process is represented as described in listing 5.16:

```haskell
(nodes simulation - voltageSources simulation)
```

Listing 5.16: Determining the number of unknown voltage sources D

There is a similar, straightforward formula at ETR-P.

gkm: Conductance for each element

For each type of RLC component, there is a formula to determine its respective conductance. It varies with the `ComponentType` and it depends on the simulation's step size. The respective Haskell function will then receive two parameters: the `ComponentData` and a Double value representing the simulation step size. It will return another Double value representing the appropriate conductance, as declared in listing 5.17.

```haskell
condutance :: ComponentData -> Double -> Double
condutance component dt =
  case componentType component of
    Resistor -> 1.0 / (magnitude component)
    Capacitor -> (magnitude component) * 0.000001 * 2 / dt
    Inductor -> dt / (2 * 0.001 * (magnitude component))
    _ -> 0.0
```

Listing 5.17: Determining the conductance for each element gkm

The `ComponentType` is identified with a **Pattern Matching** `case`. If the `ComponentType` ↪ is not a Resistor, Capacitor or Inductor, the conductance is determined as 0.0. Recall section 4.1.2 for a broader explanation of Pattern matching.

From this function, it is possible to determine the conductance vector for all the elements of the simulation. The parsed `components.csv` originates a `Vector` of `ComponentData` elements (when parsed with no errors). This initial Vector can feed the function listing 5.18:

```haskell
gkm :: Vector ComponentData -> Double -> Vector Double
gkm components dt =
  Vector.map (\c -> condutance c dt) components
```

Listing 5.18: Determining the conductance vector gkm

Listing 5.18 receives a `Vector ComponentData` and a `Double` as input values, maps each element of the `Vector` with the `condutance` function and returns another vector as the final result.

In ETR-P, despite of the several temporary variables, the code for calculating conductances relies on many nested `if`/`else` blocks (see listing 5.19)

```matlab
% gkm = conductance for each element in the input file
gkm = zeros(bmax,1);
[...]
% Calculate branch conductances for each input data row
for b = 2:bmax
    if strcmp(type(b), 'R') || strcmp(type(b), 'S')
        R = val4(b);
        gkm(b) = 1/R;
    elseif strcmp(type(b), 'L')
        L = val4(b)*1e-3; %mH
        gkm(b) = dt/(2*L);
    elseif strcmp(type(b), 'C')
        C = val4(b)*1e-6; %uF
        gkm(b) = 2*C/dt;
    elseif strcmp(type(b), 'TL')
        Zc = val4(b);
        gkm(b) = 1/Zc;
    end
end
```

Listing 5.19: Calculating conductances in Matlab

npoints: number of points in the simulation

Another straightforward value, the number of points in the simulation (`npoints`) is given by the formula $npoints = fix(tmax/dt) + 1$, where $tmax$ and dt are both obtained from the `SimulationData` and correspond to the maximum timestamp for the simulation and the step size, respectively. `fix` (which is called `round` in Haskell) truncates the division to its integer portion.

```haskell
npoints :: SimulationData -> Int
npoints sim =
  round ((tmax sim)/(stepSize sim)) + 1
```

Listing 5.20: Determining the number of points in the simulation npoints

As specified in listing 5.20, the `npoints` function receives the `SimulationData` as a single parameter and returns an `Int` with the number of points. The function `npoints` requires the `SimulationData` as an input. That means it requires a successful parsing of the `simulation.csv` file. It does not make sense to call this function if there were errors to obtain the simulation parameters. This restriction is encoded in the function's signature. Such a feature is not present in the ETR-P Matlab code.

The code in ETR-P is equally direct and for this reason it is not listed in this section.

nh: number of energy storage elements

The number of energy storage elements (`nh`) is the number of Inductors and Capacitors in the circuit. In the Haskell implementation, a `filter` functor (filter in a Vector) is created to collect only `C` or `L` `ComponentTypes` as presented in listing 5.21.

```
filterEnergyStorageComponent :: Vector ComponentData -> Vector ComponentData
filterEnergyStorageComponent =
  Vector.filter (\r -> (componentType r == Capacitor) || (componentType r ==
      ↪ Inductor))
```

Listing 5.21: filtering energy storage elements

The function `filterEnergyStorageComponent` receives a `Vector` of `ComponentData` and selects only the ones which `ComponentType` are `C` or `L`. The function `nh` (shown in listing 5.22) receives a `Vector CmponentData`, then filters it with `filterEnergyStorageComponent` ↪ and returns the length of the filtered elements.

```
nh :: Vector ComponentData -> Int
nh components =
  length $ filterEnergyStorageComponent components
```

Listing 5.22: Determining the number energy storage elements nh

The Matlab ETR-P version relies on `if`/`else` blocks to determine the value of `nh`, as seen in listing 5.23.

```
% nh = number of energy storage elements
nh = 0;
[...]
for b = 2:bmax
    % Set tstart to zero. If we find any source that was active at t<0,
    % set the program to compute the Steady State Solution.
    tstart = 0;
    if strcmp(type(b), 'L') || strcmp(type(b), 'C')
        % finding the number of lumped elements which need history terms
        nh = nh + 1;
        val8(b) = nh;
        [...] % Code to deal with Transmission Lines ommited
```

Listing 5.23: Determining the number energy storage elements nh with ETR-P

G matrix: Conductances matrix

The conductances square matrix `G` holds the conductances calculated with the support of the `gkm` vector. The `G` matrix is built using **Recursion** and with the `let/in` Haskell structures (check listing 5.24).

```haskell
buildGMatrixFromList :: SimulationData -> Matrix Double -> [ComponentData] ->
    ↪ Matrix Double
buildGMatrixFromList _ buffer [] = buffer
buildGMatrixFromList simulation buffer (c:cs) =
  let gkmss = condutance c $ stepSize simulation
      bufferUpdtate = gMatrix (nodeK c, nodeM c) buffer gkmss
  in buildGMatrixFromList simulation bufferUpdtate cs
```

Listing 5.24: Building the G Matrix

The function `buildGMatrixFromList` receives the `SimulationData`, a buffer matrix (which its initial value is set to zero), a list of `ComponentData` (`[ComponentData]`) and returns the `G` matrix. It is a recursive function: if the list of `ComponentData` is empty, it returns the value held in the buffer. Otherwise, it calculates the conductance of the given element, updates the values of the buffer with the function `gMatrix` and holds the value on `bufferUpdtate`.

```haskell
gMatrix :: (Int, Int) -> Matrix Double -> Double -> Matrix Double
gMatrix (k, 0) buffer gkmHead = diagonalUpdate k buffer gkmHead
gMatrix (0, m) buffer gkmHead = diagonalUpdate m buffer gkmHead
gMatrix (k, m) buffer gkmHead =
  let updateK = ((k, k), (Matrix.getElem k k buffer) + gkmHead)
      updateM = ((m, m), (Matrix.getElem m m buffer) + gkmHead)
      updated1 = ((k, m), (Matrix.getElem k m buffer) - gkmHead)
      updated2 = ((m, k), (Matrix.getElem m k buffer) - gkmHead)
  in
    gMatrixUpdate [updateK, updateM, updated1, updated2] buffer

gMatrixUpdate :: [((Int, Int), Double)] -> Matrix Double -> Matrix Double
gMatrixUpdate [] gmatrix = gmatrix
gMatrixUpdate (((k, m), toUpdate):xs) gmatrix =
  gMatrixUpdate xs (Matrix.setElem toUpdate (k, m) gmatrix)

diagonalUpdate :: Int -> Matrix Double -> Double -> Matrix Double
diagonalUpdate d buffer gkmHead =
  let updated = (Matrix.getElem d d buffer) + gkmHead
  in Matrix.setElem updated (d, d) buffer
```

Listing 5.25: Building the G Matrix buffer update

In listing 5.25, the auxiliary function `gMatrix` receives a Tuple of `Int` representing the nodes from the component (`nodeK` and `nodeM`), the buffer (a `Matrix Double`) and the value of the conductance for the element. It patterns matches according to the positions in the tuple (k, m) and calls another auxiliary function, `diagonalUpdate` (when `k` or `m` are zero), or `gMatrixUpdate` (when `k` and `m` have distinct values).

Once again, there isn't a single `if/else` block in this function. Everything is determined and calculated with **Pattern Matching** and **Recursion**. Every step of the recursive call produces a new matrix. There is no risk of mutability-related issues.

The ETR-P code for creating the G Matrix is described in listing 5.26.

```
%% Build and partition the G matrix
% Calculate branch conductances for each input data row
for b = 2:bmax
    if strcmp(type(b), 'R') || strcmp(type(b), 'S')
        R = val4(b);
        gkm(b) = 1/R;
    elseif strcmp(type(b), 'L')
        L = val4(b)*1e-3; %mH
        gkm(b) = dt/(2*L);
    elseif strcmp(type(b), 'C')
        C = val4(b)*1e-6; %uF
        gkm(b) = 2*C/dt;
    end
end

% Build the G matrix
for b = 2:bmax
    k = from(b);
    m = to(b);
    if m == 0
        G(k,k) = G(k,k) + gkm(b);
    elseif k == 0
        G(m,m) = G(m,m) + gkm(b);
    else
        G(k,k) = G(k,k) + gkm(b);
        G(m,m) = G(m,m) + gkm(b);
        if strcmp(type(b), 'TL') == 0
            % If the branch is NOT a transmission line then calculate
            % off-diagonals
            G(k,m) = G(k,m) - gkm(b);
            G(m,k) = G(m,k) - gkm(b);
        end
    end
end
```

Listing 5.26: G Matrix with ETR-P

5.2.1 Refactoring example - GMatrix

The listing 5.24 and its auxiliary functions can be shorter and deserve a refactor. They can be rewritten to be more compact. This enhancement is listed on [20] and this section implements the desired improvement.

The code on listing 5.24 and on listing 5.25 is functional, but verbose. This was the first version of the Haskell implementation and this subsection can be a good place to show the process of refactoring functional code. After building the auxiliary functions on listing 5.25, it became clear that the `GMatrix` could be built only with the conductances (which could be calculated on demand, not necessarily in a `let/in` block)and with pattern matching across the nodes K and M. listing 5.27 brings an alternative version for the implementation, more compact and more direct. In this code, the function `buildGMatrixFromList` is replaced by `buildGMatrixFromVector` (for making the function call simpler), and all the auxiliary functions (`diagonalUpdate`, `gMatrixUpdate` and `gMatrix`) are replaced by a single function, `buildCompactGMatrix`, that receives the step size of the simulation (`dt`), the list of components and a buffer for the `GMatrix`, initially set to zero.

```haskell
buildGMatrixFromVector :: SimulationData -> Vector ComponentData -> Matrix Double
buildGMatrixFromVector simulation components =
  buildCompactGMatrix (stepSize simulation) (Vector.toList components) (Matrix.zero (nodes simulation) (nodes simulation))


buildCompactGMatrix :: Double -> [ComponentData] -> Matrix Double -> Matrix Double
buildCompactGMatrix dt [] buffer = buffer
buildCompactGMatrix dt (component:cs) buffer =
  case (nodeK component, nodeM component) of
        (0, m) -> buildCompactGMatrix dt cs (Matrix.setElem (Matrix.getElem m m buffer + condutance component dt) (m, m) buffer)
        (k, 0) -> buildCompactGMatrix dt cs (Matrix.setElem (Matrix.getElem k k buffer + condutance component dt) (k, k) buffer)
        (k, m) -> buildCompactGMatrix dt cs (Matrix.setElem (Matrix.getElem k k buffer + condutance component dt) (k, k) (Matrix.setElem (Matrix.getElem m m buffer + condutance component dt) (m, m) (Matrix.setElem (Matrix.getElem k m buffer - condutance component dt) (k, m) (Matrix.setElem (Matrix.getElem m k buffer - condutance component dt) (m, k) buffer))))
        (_, _) -> buildCompactGMatrix dt cs buffer
```

Listing 5.27: Building the G Matrix buffer update

In listing 5.27, for the case where the GMatrix needs to be updated outside the diagonal (which requires multiple changes), recursion is used in the `buffer`, producing the effect of multiple sequential updates to each index of the matrix.

5.3 Simulation Loop

With the initial setup completed, the next step is the simulation loop. In terms of functional programming, building this step was one of the most challenging parts of this work. A functional approach for the simulation loop had to be built on top of Recursion and Pattern matching - that means identifying the parameters changing in every iteration under the matrix form of the algorithm. The imperative implementation of ETR-P does not make it clear which parameters belong to the main body of the algorithm on every step of the iteration and which information is just a tool to guarantee the procedural process (for example, the indexes of vectors and matrices).

The first step to build this recursive simulation was identifying what are the fixed parameters, as well as what are the ones that change on every step of the iteration. These discoveries are specified in the function's signatures transcribed in listing 5.28.

```haskell
thtaSimulationStep :: [ComponentData] -> Matrix Double -> SimulationData -> Int
    ↪ -> Int -> Double -> Vector Double -> Matrix Double -> Vector Double ->
    ↪ Vector Double -> SimulationResults
thtaSimulationStep components condutances simulation thtactl n time ih vMatrix
    ↪ vbVector iVector =
 -- implementation
```

Listing 5.28: Building the simulation loop - parameter identification

Every step of the simulation requires the list of `ComponentData`, the `Matrix` of conductances, the `SimulationData`, the `THTACtl` (the core of the THTA algorithm in ETR-P), the simulation step `n`, the time stamp `time` (required for determining `EAC` values), the historical vector `Ih`, the matrix of Voltages `V`, the vector with voltage sources `VB` and the vector of current `I`. The `conductance` of the components is also necessary, but it can be calculated in every step (it is a pure function, which means it will certainly have the same value for the same inputs). This extensive list of parameters produces some intermediate values in every step - they are listed in the `thtaSimulationStep` implementation in listing 5.29.

```haskell
thtaSimulationStep :: [ComponentData] -> Matrix Double -> SimulationData -> Int
    ↪ -> Int -> Double -> Vector Double -> Matrix Double -> Vector Double ->
    ↪ Vector Double -> SimulationResults
thtaSimulationStep _ _ _ _ 1 _ _ vMatrix _ iVector = (iVector, vMatrix)
thtaSimulationStep components condutances simulation thtactl n time ih vMatrix
    ↪ vbVector iVector =
  let (gaa, gab, gba, gbb) = Matrix.splitBlocks ((nodes simulation) - (
      ↪ voltageSources simulation)) ((nodes simulation) - (voltageSources
      ↪ simulation)) condutances
      ihBuffer = buildIhVector (nhComponents components) (stepSize simulation) n
          ↪ (Vector.toList ih) [] vMatrix
      (thta, ihThta, timeThta) = thtaControl thtactl time ihBuffer ih simulation
      vbVec = buildVBVector components timeThta []
      iVec = buildIVector (nhComponents components) (Vector.toList ihThta) (
          ↪ Vector.replicate (nodes simulation) 0)
      (iVecCalc, vVec) = solver (toHMatrixVectorTransformer iVec) (
          ↪ toHMatrixTransformer gaa) (toHMatrixTransformer gab) (
          ↪ toHMatrixTransformer gba) (toHMatrixTransformer gbb) (
          ↪ toHMatrixVectorTransformer vbVec) simulation
      vMatr = Matrix.mapCol (\r _ -> vVec Vector.! (r - 1)) (n-1) vMatrix
  in
      thtaSimulationStep components condutances simulation thta (n-1) timeThta
          ↪ ihThta vMatr vbVec iVecCalc
```

Listing 5.29: Building the simulation loop - let bindings

First, the Tuple `(gaa, gab, gba, gbb)` originates from the `Matrix.split` function, after splitting the matrix of conductances `G` in four sub-matrices. In the model, they are equivalent to the GAA, GAB, GBA and GBB submatrices of the linear system. Next, it is necessary to calculate the current historical values and store the results in `ihBuffer`. It demands a call to an auxiliary function `buildIhVector`. `vbVec` is the `VB` Vector for that step. The `thataControl` function returns a Tuple with the THTA Control and the updated value of the current historic vector, as well as the time step. Next, the values of `iVec` are calculated. `iVec` is the vector of current values, based on the `Ih` (historic current) vector previously calculated. All these intermediate values are passed to the `solver` function, which returns a Tuple with a final value for the `I Vector` and the updated columns of the `V` Matrix. The following binding, `vMatr`, updates the `V` Matrix. Finally, the next step of the recursion is invoked with the newly calculated values.

There is a pattern matching at the beginning of the function `thtaSimulationStep` that forces the stop of the recursive calls if the simulation step is equals to 1. So in the functional approach, the initial step starts the simulation within the largest step `n` and decreases it's value in each step of the iteration. Going from `npoints` to 1 is the same as going from 1 to `npoints`. The `V` Vector will be built from right to left.

The original implementation of ETR-P in Matlab does not aggregate these steps in a single location. It is reasonable, instead, to compare the auxiliary functions `buildIhVector`, `buildVBVector`, `thtaControl`, `buildIVector` and `solver` with the original implementation.

buildIhVector: creating a buffer for the historic current values

The `buildIhVector` function takes the list of components (already filtered for L and C only), the step size parameter (obtained from `SimulationData`), the iteration step, the buffer of the `Ih` Vector (if any), an empty buffer vector (`ihnew`) and the `V` Matrix, returning an updated `Ih` vector buffer. Its values are updated bases on the nodes `K` and `M` and also on the `ComponentType`. That is why there is an extensive pattern matching against the Tuple made of these three pieces of information. The values of the conductances are also necessary and are calculated when needed, as shown listing 5.30.

```haskell
buildIhVector :: [ComponentData] -> Double -> Int -> [Double] -> [Double] ->
    ↪ Matrix Double -> Vector Double
buildIhVector [] _ _ _ ihnew _ = Vector.fromList ihnew
buildIhVector (component:cs) dt n (hold:ihold) ihnew vMatrix =
  case (componentType component, nodeK component, nodeM component) of
      (Inductor, 0, m) -> buildIhVector cs dt n ihold (ihnew ++ [(2*(condutance
          ↪ component dt)*(Matrix.getElem m n vMatrix) + hold)]) vMatrix
      (Inductor, k, 0) -> buildIhVector cs dt n ihold (ihnew ++ [(-2*(condutance
          ↪ component dt)*(Matrix.getElem k n vMatrix) + hold)]) vMatrix
      (Inductor, k, m) -> buildIhVector cs dt n ihold (ihnew ++ [(-2*(condutance
          ↪ component dt)*((Matrix.getElem k n vMatrix) - (Matrix.getElem m n
          ↪ vMatrix)) + hold)]) vMatrix
      (Capacitor, 0, m) -> buildIhVector cs dt n ihold (ihnew ++ [(-2*(condutance
          ↪  component dt)*(Matrix.getElem m n vMatrix) - hold)]) vMatrix
      (Capacitor, k, 0) -> buildIhVector cs dt n ihold (ihnew ++ [(2*(condutance
          ↪ component dt)*(Matrix.getElem k n vMatrix) - hold)]) vMatrix
      (Capacitor, k, m) -> buildIhVector cs dt n ihold (ihnew ++ [(2*(condutance
          ↪ component dt)*((Matrix.getElem k n vMatrix) - (Matrix.getElem m n
          ↪ vMatrix)) - hold)]) vMatrix
      (_, _, _) -> buildIhVector cs dt n ihold ihnew vMatrix
```

Listing 5.30: Building the Ih buffer vector: buildIhVector

Compare the listing 5.30 with the Matlab ETR-P implementation at listing 5.33.

```matlab

%% Calculate the history sources at time t
    % to be used in t+dt for R,L,C elements and in t+Tau for TL
    Ihold = Ih;
    Ihnew = zeros(nh,1);
    for b = 2:bmax
        k = from(b);
        m = to(b);
        idxhist = val8(b);
        idxplt = val9(b);
        if strcmp(type(b), 'L')
            if k == 0
                Ihnew(idxhist) = 2 * gkm(b) * V(m,n) + Ihold(idxhist);
            elseif m == 0
                Ihnew(idxhist) = -2 * gkm(b) * V(k,n) + Ihold(idxhist);
            else
                Ihnew(idxhist) = -2 * gkm(b) * (V(k,n)-V(m,n)) + Ihold(idxhist);
            end
        elseif strcmp(type(b), 'C')
            if k == 0
                Ihnew(idxhist) = -2 * gkm(b) * V(m,n) - Ihold(idxhist);
            elseif m == 0
                Ihnew(idxhist) = 2 * gkm(b) * V(k,n) - Ihold(idxhist);
            else
                Ihnew(idxhist) = 2 * gkm(b) * (V(k,n)-V(m,n)) - Ihold(idxhist);
            end

            % Code for Transmission lines omited.
        end
    end
```

Listing 5.31: Ih vector buffer in Matlab ETR-P

buildVBVector: creating the vector with sources values

The code for the `VB` Vector is pretty straightforward, as seen in listing 5.32.

```haskell
buildVBVector :: [ComponentData] -> Double -> [Double] -> Vector Double
buildVBVector [] _ buffer = Vector.fromList buffer
buildVBVector (c:components) time buffer =
  case (componentType c) of EDC -> buildVBVector components time ((magnitude c) : buffer)
                            EAC -> buildVBVector components time (((magnitude c * cos (2 * pi * param2 c * time + (param1 c * (pi /180))))) : buffer)
                            _ -> buildVBVector components time buffer
```

Listing 5.32: Creating the vector with sources values: buildVBVector

`buildVBVector` receives the list of components, the current time and a temporary buffer with the vector values to be returned after the recursion. It filters the sources (`EAC` and `EDC`) and calculates the voltage vectors accordingly. Triangular sources are not supported yet.

Compare the listing 5.32 with the Matlab ETR-P implementation in listing 5.33:

```matlab
%% Build the VB vector for the time t
    x = 1;
    for b = 2:bmax
        k = from(b);
        m = to(b);
        if strcmp(type(b), 'EDC')
            % DC Voltage Source
            VB(x,1)= val4(b);
            x = x+1;
        elseif strcmp(type(b), 'EAC')
            % AC Voltage Source
            VB(x,1)= val4(b)*cos(2*pi*val6(b)*time + val5(b)*(pi/180) );
            x = x+1;
        end
        % Omitting Triangular sources; not implemented in the Haskell Version
```

Listing 5.33: VB vector buffer in Matlab ETR-P

thataControl: The core of the THTA Algorithm

The next auxiliary function is `thtaControl` in the listing 5.34, which implements the "Trapezoidal History Term Averaging" [17] (THTA) method.

```haskell
thtaControl :: Int -> Double -> Vector Double -> Vector Double -> SimulationData
    ↪ -> (Int, Vector Double, Double)
thtaControl thtactl time ihnew ih simulation
  | thtactl <= 0 = (thtactl, ihnew, (stepSize simulation + time))
  | thtactl < 3 = (thtactl + 1, (Vector.map (\i -> i/2) $ Vector.zipWith (+) ih
      ↪ ihnew), (time + (stepSize simulation/2)))
  | otherwise = (0, ihnew, (stepSize simulation + time))
```

Listing 5.34: Trapezoidal History Term Averaging ETR-P

The function `thtaControl` receives the previous `thtaCtl` value (which is an integer argument that determines the next timestamp), the current timestamp (a Double), the calculated `ihnew` values, the `Ih` that should be updated and finally, the `SimulationData`. It returns the results in a Tuple of three elements: the updated `thtaCtl` integer, the updated `Ih` (which was called `ihnew`) and finally, the timestamp. This function uses **Haskell guards** (`|`) to compare and pattern match against the `thtaCtl` value.

Compare with the original Matlab implementation at listing 5.35.

```matlab
%% THTA Control Here!
    if Damp == 1
        THTACtl = 0;
    end
    if THTACtl > 0
        if THTACtl == 1
            fprintf('THTA activated at t = %2.50f.\n', time);
        end

        if THTACtl < 3
            fprintf('THTA step %d at t = %2.50f.\n', THTACtl, time);

            Ih = (Ih + Ihnew)/2;

            THTACtl = THTACtl + 1;
            time = time + dt/2;
        else
            Ih = Ihnew;

            THTACtl = 0;
            time = time + dt;
        end
    else
        Ih = Ihnew;
        % Regular operation
        % Increment the time
        time = time + dt;
    end
```

Listing 5.35: THTA Control in Matlab

buildIVector: Calculating the current values for each iteration step

After obtaining the updated values for the `Ih` with the `thtaCtl` function, it is possible to calculate the values for the `I Vector`. The function `buildIVector` receives the list of `ComponentData` elements (already filtered for L and C only), the `Ih` vector and a zero-valued vector for buffering purposes. It returns the calculated `I` Vector, based on **Pattern Matching** a Tuple with the nodes `K` and `M` of each component, along with its `ComponentType`. It is also a recursive function, iterating over all the components of the circuit, as shown in listing 5.36.

```haskell
buildIVector :: [ComponentData] -> [Double] -> Vector Double -> Vector Double
buildIVector [] _ iVector = iVector
buildIVector (component:cs) (ihEl:ih) iVector =
  case (componentType component, nodeK component, nodeM component) of
      (Inductor, k, 0) -> buildIVector cs ih (iVector Vector.// [((k - 1), ((
          ↪ iVector Vector.! (k-1)) + ihEl))])
      (Inductor, 0, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
          ↪ iVector Vector.! (m-1)) - ihEl))])
      (Inductor, k, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
          ↪ iVector Vector.! (m-1)) - ihEl)), ((k-1), ((iVector Vector.! (k-1)) +
          ↪  ihEl))])
      (Capacitor, k, 0) -> buildIVector cs ih (iVector Vector.// [((k - 1), ((
          ↪ iVector Vector.! (k-1)) + ihEl))])
      (Capacitor, 0, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
          ↪ iVector Vector.! (m-1)) - ihEl))])
      (Capacitor, k, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
          ↪ iVector Vector.! (m-1)) - ihEl)), ((k-1), ((iVector Vector.! (k-1)) +
          ↪  ihEl))])
      (_, _, _) -> buildIVector cs ih iVector
```

Listing 5.36: Building I Vector: buildIVector

The original Matlab ETR-P version consists of the code presented at listing 5.37.

```matlab
    %% Clear vector I
    I = zeros(N,1);

    %% Add History Current Sources, evaluated at time t-dt for R,L,C elements
    % or evaluated at t-Tau for TL, into vector I
    %
    for b = 2:bmax
        k = from(b);
        m = to(b);
        % idx = transmission line index for history terms
        % or the lumped element index for history term
        idx = val8(b);
        if strcmp(type(b), 'L') || strcmp(type(b), 'C')
            if m == 0;
                I(k)= I(k) + Ih(idx);
            elseif k == 0;
                I(m)= I(m) - Ih(idx);
            else
```

```
                I(k)= I(k) + Ih(idx);
                I(m)= I(m) - Ih(idx);
            end
        end
    end
```

Listing 5.37: Building the I Vector with Matlab ETR-P

ETR-P Matlab version uses several auxiliary indexes (such as `idx` at listing 5.37) to keep track of Vector indexes in order to calculate output values accordingly. The Haskell version uses recursion and pattern matching, not requiring any sort of temporary index to keep track of the operations.

solver: putting the pieces together for I and V

The last auxiliary function is the `solver`, which receives the following arguments: each submatrix of the G Matrix (split into GAA, GAB, GBA and GBB), the `I` Vector, the `VB` vector and the `SimulationData`. It returns a Tuple with the final value of `I` and the updated column for the `V` Matrix (see listing 5.38).

```

solver :: HMatrix.Vector Double -> HMatrix.Matrix Double -> HMatrix.Matrix Double
    ↪  -> HMatrix.Matrix Double -> HMatrix.Matrix Double -> HMatrix.Vector
    ↪ Double -> SimulationData -> (Vector Double, Vector Double)
solver iVector gaa gab gba gbb vb simulation =
  let ia = HMatrix.subVector 0 ((nodes simulation) - (voltageSources simulation))
      ↪  iVector
      rhsa = ia - (gab HMatrix.#> vb)
      va = gaa HMatrix.<\> rhsa
      ib = (gba HMatrix.#> va) + (gbb HMatrix.#> vb)
      iVec = HMatrix.vjoin [ia, ib]
      vVec = HMatrix.vjoin [va, vb]
  in
    ((fromHMatrixVectorTransformer iVec), (fromHMatrixVectorTransformer vVec))
```

Listing 5.38: Calculating the final values for I and V in each iteration step

To determine the `VA` value, there is a requirement to solve a Linear System Equations `GAA\RHSA`. It would have been possible to solve it manually using Linear Algebra. However, it was easier to use the tools provided by the library **HMatrix**. In order to use the HMatrix, it was necessary to convert the `Matrix` type from the default package to a `HMatrix` Matrix (same situation for `Vector`). They are declared as different types: in Object Oriented Languages, such as C++ or Python, that would be similar to having two different `Matrix` classes (coming from different packages or modules). The type signature of the `solver` function requires a `HMatrix` Matrix and it returns a Tuple of Standard Vectors. For converting these types back and forth, four auxiliary functions were created. They are described at listing 5.39.

```
fromHMatrixTransformer :: HMatrix.Matrix Double -> Matrix Double
```

```haskell
fromHMatrixTransformer matrix =
  Matrix.fromLists $ HMatrix.toLists matrix

toHMatrixTransformer :: Matrix Double -> HMatrix.Matrix Double
toHMatrixTransformer matrix =
  HMatrix.fromLists $ Matrix.toLists matrix

fromHMatrixVectorTransformer :: HMatrix.Vector Double -> Vector Double
fromHMatrixVectorTransformer vec =
  Vector.fromList $ HMatrix.toList vec

toHMatrixVectorTransformer :: Vector Double -> HMatrix.Vector Double
toHMatrixVectorTransformer vec =
  HMatrix.fromList $ Vector.toList vec
```

Listing 5.39: Converting between Matrices and Vectors - HMatrix and standard types

Inside the function `solver`, all the operations are based on the `HMatrix` library. It would have been possible to use a single type of Matrix/Vector; for example the ones from the `HMatrix` library. However, this is a tool that focuses on advanced linear operations which are not required for most parts of the ETR-P Haskell algorithm. The principle of "Less is More" is one baseline of this work, and over-engineering is avoided in order to keep the results as concise and straightforward as possible.

The matrix type comes from the library `Data.Matrix` [46]. It enforces the functional programming behaviour, so it is not possible to mutate an existing Matrix. The matrix type on `HMatrix` [47], on the other hand, allows the user to apply a more imperative approach by implementing mutable structures.

Since the `solver` function returns the updated column for the `V` Matrix, it is necessary to update the matrix itself with the values. Back to the `thtaSimulationStep` ↪ function, the binding `vMatr = Matrix.mapCol (\r _ -> vVec Vector. (r - 1))(n-1)` ↪ `vMatrix` updates the previous column of the `V` Matrix, making it ready for the next step of the iteration.

Compare the `solver` function with the equivalent operations in ETR-P in listing 5.40.

```matlab
    %% Build vector IA for the time t
    IA = I(1:D, 1);

    %% Build the vector RHSA for the time t
    RHSA = IA - GAB*VB;

    %% Solve for vector VA at the time t
    VA = GAA\RHSA;
    IB = GBA*VA+GBB*VB;
    I = [IA; IB];
    %% Build vector V at the time t (i.e., time counter or point n)
    V(:, n) = [VA; VB];
```

Listing 5.40: Calculating the final values for I and V in Matlab ETR-P

5.3.1 Improvements to the simulation loop

There is a lot of space for code refactoring in the scope of the simulation loop functions. Some enhancements are listed below, and their corresponding issues are already tracked on Github:

- Some functions could be treated as internal (or Lambda) functions. They wouldn't need to be declared as a separate function. This work made all the functions to be external functions in order to discuss their signatures and make the algorithm more explicit in terms of types. It is not possible to explicit the function declaration when creating internal functions. It would invalidate several discussions on this chapter.
- Some functions have a long parameter list. It would be possible to rewrite their calls as partial applications.
- Implementing support for external current sources (IAC, IDC) is a desired feature.
- Implementing support for Triangular voltage sources (ETR) and Triangular current sources (ITR) is another desired feature.
- Implementing support for Switches is another upgrade to be developed.
- Supporting for Transmission Lines is a desired functionality.
- Plotting Charts for Current and Voltage information is another feature.

5.4 Running the code

Now that the previous section presented an overview of the main functions of the program, the basic project settings are explained and detailed. This Haskell project is built with **Stack** [8] - it creates a simple way to compile and run the main code together with its dependencies - and **Cabal** [9], a tool to easily manage external libraries as project dependencies. Since this is a short program, all the content was kept in a single file in a single module. The `Main.hs` file has a `main` function, which is called when the command `stack run` is invoked in the project's directory. Detailed build information can be found at the project's *README.md* instruction file [20].

In the `main` function, the functions to parse the `csv` files are invoked, creating the initial setup and moving into the simulation loop (shown on listing 5.41).

```haskell
main :: IO ()
main = do
  eitherSimulation <-
      fmap getSingleSimulationLine
        <$> decodeSimulationFromFile "data/simulation.csv"

  case eitherSimulation of
    Left reason ->
      Exit.die reason

    Right simulation -> do
      components_list <- decodeItemsFromFile "data/components.csv"
      case components_list of
        Left reason -> Exit.die reason
        Right components -> do
          let results = thtaSimulation components simulation
          putStr "Simulation: \n"
          print (results)
```

Listing 5.41: Main function

In the code on listing 5.41, the initial `Either` structure holds the result of a successful parsing of the simulation file or a failure which throws an error and exits the program. If the parsing operation was successful (`Right simulation`), the same process is repeated for the components file. In case of another successful parsing, then the simulation is invoked with `thtaSimulation`. It is necessary to provide the two parameters this function requires: the parsed `ComponentData Vector` and the `SimulationData`.

An improvement that can be implemented is transforming the file names into parameters of the `main` function call. See [20]. This single file could also be split into smaller files. The logic for the nodal analysis could be encapsulated in a module. This is not the goal of this project, but it is interesting to acknowledge the importance of modularity.

The complete code of the Haskell ETR-P is listed on the Appendix D.

The Matlab ETR-P version does not hold these operations in an isolated function. It completes the simulation by procedurally by parsing the input files and running for loops.

5.4.1 Open Source implementation on Github

As a final remark for this chapter, it is important to emphasize that all the code is open source. It is hosted on Github at `https://github.com/hannelita/thtahs`. A broader analysis of the benefits of open source projects is beyond the scope of this work. However, it is possible to list a few key points: benefits for learning and e-learning [48], quick feedback from end-users [49], distributed management [50] and the advantage of easily sharing source code.

6 Results

6.1 Simulation outputs - RL Circuit

In order to demonstrate that the results obtained by the Haskell ETR-P version are the same as the ones in the Matlab version, a simulation using the same parameters in both Matlab and Haskell version is adopted. The chosen circuit configuration, components and simulation values are presented at listing 6.1 and at listing 6.2 (which happen to be the same values used at chapter 5). The circuit is shown in fig. 6.1.

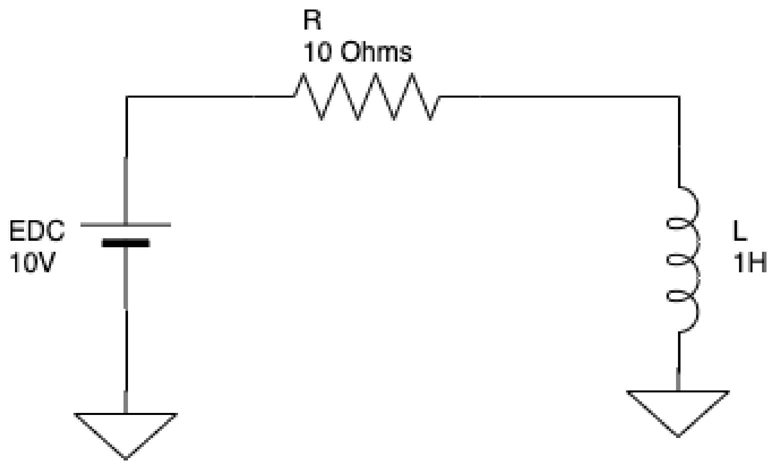


Figure 6.1: DC Circuit

```
Element Type,Node K,Node M,Value,Source param 1,Source param 2,Plot
EDC,2,0,10,0,0,0
R,2,1,10,0,0,0
L,1,0,1,0,0,0
```

Listing 6.1: Input data file for components in the Haskell implementation

```
Number of Nodes,Number of Voltages Sources,Step Size,Maximum time for simulation
2,1,0.0001,0.0005
```

Listing 6.2: Input data file for time

The same setup in ETR-P is given by the file shown at listing 6.3.

```
T 2 1 100E-6 5E-4 0 0 0 0 0
```

```
EDC 2 0 10 0 0 0 0 0 5
R 2 1 10 0 0 0 0 0 5
L 1 0 1 0 0 0 0 0 5
NV 1 2 0 0 0 0 0 0 0
```

Listing 6.3: Original input data file for ETR-P Matlab

6.1.1 Short simulation with DC Voltage

The simulation time is only 0.0005 seconds, which gives us only 6 `npoints` for the step size of 0.0001. In a later section of this chapter, a longer simulation will be tested and executed.

In the Matlab version, it is possible to log the results from each iteration step by removing the semi-colons at the end of each line, or by explicitly calling the command `fprintf`, as the example in listing 6.4.

```
%% Build vector IA for the time t
    IA = I(1:D, 1);

    %% Build the vector RHSA for the time t
    RHSA = IA - GAB*VB;

    %% Solve for vector VA at the time t
    VA = GAA\RHSA;
    IB = GBA*VA+GBB*VB;
    I = [IA; IB];
    %% Build vector V at the time t (i.e., time counter or point n)
    V(:, n) = [VA; VB];

    fprintf('---------------- I final in every step of the loop -----------\n');
    I
    fprintf('---------------- V final in every step of the loop -----------\n');
    V
```

Listing 6.4: Logging the results of V and I at the end of each iteration in Matlab

Logging and debugging is not equally straightforward in Haskell. A log would represent an impurity, since it writes at the standard output. To print out the values of operations in pure functions, it is necessary to use the `Debug.Trace` library. To log the values of `I` and `V` in every iteration, there must be a call to the `Trace.trace` function in some of the let binding in the `thtaSimulationStep` function. Example in listing 6.5.

```
thtaSimulationStep :: [ComponentData] -> Matrix Double -> [Double] ->
    ↪ SimulationData -> Int -> Int -> Double -> Vector Double -> Matrix Double
    ↪ -> Vector Double -> Vector Double -> SimulationResults
thtaSimulationStep _ _ _ _ _ 1 _ _ vMatrix _ iVector = (iVector, vMatrix)
thtaSimulationStep components condutances gkms simulation thtactl n time ih
    ↪ vMatrix vbVector iVector =
let (gaa, gab, gba, gbb) = Matrix.splitBlocks ((nodes simulation) - (
    ↪ voltageSources simulation)) ((nodes simulation) - (voltageSources
    ↪ simulation)) condutances
    ihBuffer = buildIhVector components gkms n ih (Vector.replicate (nh (Vector
        ↪ .fromList components)) 0) vMatrix
    vbVec = buildVBVector components
    (thta, ihThta, timeThta) = thtaControl thtactl time ihBuffer ih
    iVec = buildIVector components ihThta (Vector.replicate (nodes simulation)
        ↪ 0)
    (iVecCalc, vVec) = Trace.trace ("Solver = \n" ++ show (solver (
        ↪ toHMatrixVectorTransformer iVec) (toHMatrixTransformer gaa) (
        ↪ toHMatrixTransformer gab) (toHMatrixTransformer gba) (
        ↪ toHMatrixTransformer gbb) (toHMatrixVectorTransformer vbVec)
        ↪ simulation)) solver (toHMatrixVectorTransformer iVec) (
        ↪ toHMatrixTransformer gaa) (toHMatrixTransformer gab) (
        ↪ toHMatrixTransformer gba) (toHMatrixTransformer gbb) (
        ↪ toHMatrixVectorTransformer vbVec) simulation
    vMatr = Trace.trace ("vMatrix = \n" ++ show (Matrix.mapCol (\r _ -> vVec
        ↪ Vector.! (r - 1)) (n-1) vMatrix)) Matrix.mapCol (\r _ -> vVec Vector
        ↪ .! (r - 1)) (n-1) vMatrix
in
    thtaSimulationStep components condutances gkms simulation thta (n-1) time
        ↪ ihThta vMatr vbVec iVecCalc
```

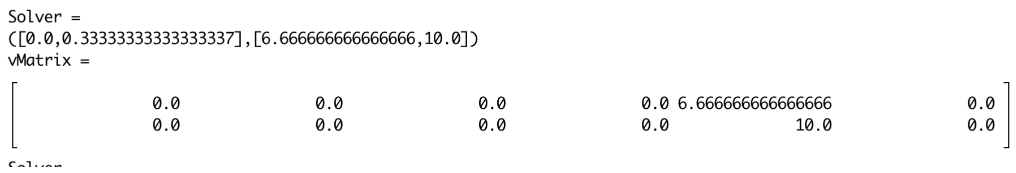
Listing 6.5: Logging the values of I and V at the Haskell implementation

Step 0

Initial step. Everything is set to 0.0.

Step 1

Results in Haskell - fig. 6.2:



```
Solver =
([0.0,0.33333333333333337],[6.666666666666666,10.0])
vMatrix =

[              0.0              0.0              0.0              0.0 6.666666666666666              0.0 ]
[              0.0              0.0              0.0              0.0              10.0              0.0 ]
```

Figure 6.2: Results for step 5 - Haskell (recall that the Haskell loop stars from the highest value of the iteration, which is 5)

Results form Matlab - fig. 6.3:

```
---------------- I final in every step of the loop -----------
I =

    0.0000e+00
  333.3333e-03

---------------- V final in every step of the loop -----------
V =

    0.0000e+00     6.6667e+00     0.0000e+00     0.0000e+00     0.0000e+00     0.0000e+00
    0.0000e+00    10.0000e+00     0.0000e+00     0.0000e+00     0.0000e+00     0.0000e+00
```

Figure 6.3: Results for step 1 - Matlab

Step 2

Results in Haskell - fig. 6.4:

```
Solver =
([-0.3333333333333333,0.5555555555555556],[4.444444444444445,10.0])
vMatrix =
```

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 & 4.444444444444445 & 6.666666666666666 & 0.0 \\ 0.0 & 0.0 & 0.0 & 10.0 & 10.0 & 0.0 \end{bmatrix}$$

Figure 6.4: Results for step 4 - Haskell

Results form Matlab - fig. 6.5:

```
---------------- I final in every step of the loop -----------
I =

  -333.3333e-03
   555.5556e-03

---------------- V final in every step of the loop -----------
V =

    0.0000e+00     6.6667e+00     4.4444e+00     0.0000e+00     0.0000e+00     0.0000e+00
    0.0000e+00    10.0000e+00    10.0000e+00     0.0000e+00     0.0000e+00     0.0000e+00
```

Figure 6.5: Results for step 2 - Matlab

Step 3

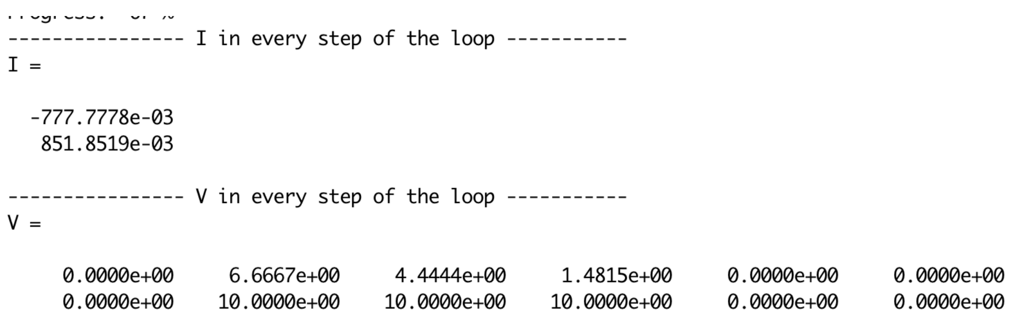
Results in Haskell - fig. 6.6:

```
Solver =
([-0.7777777777777778,0.8518518518518519],[1.4814814814814812,10.0])
vMatrix =
```

$$\begin{bmatrix} 0.0 & 0.0 & 1.4814814814814812 & 4.444444444444445 & 6.666666666666666 & 0.0 \\ 0.0 & 0.0 & 10.0 & 10.0 & 10.0 & 0.0 \end{bmatrix}$$

Figure 6.6: Results for step 3 - Haskell

Results form Matlab - fig. 6.7:

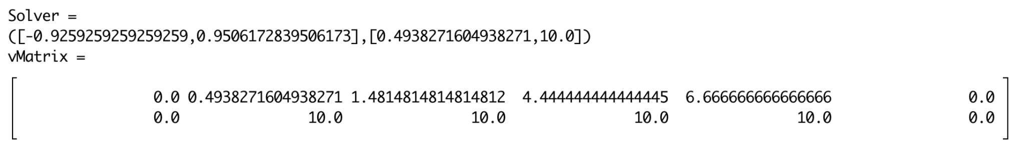


```
---------------- I in every step of the loop -----------
I =

  -777.7778e-03
   851.8519e-03

---------------- V in every step of the loop -----------
V =

     0.0000e+00     6.6667e+00     4.4444e+00     1.4815e+00     0.0000e+00     0.0000e+00
     0.0000e+00    10.0000e+00    10.0000e+00    10.0000e+00     0.0000e+00     0.0000e+00
```

Figure 6.7: Results for step 3 - Matlab

Step 4

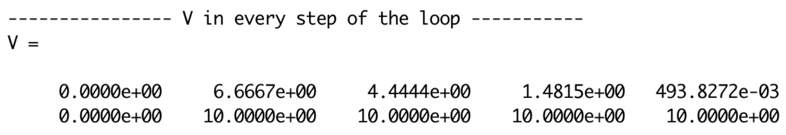
Results in Haskell - fig. 6.8:



```
Solver =
([-0.9259259259259259,0.9506172839506173],[0.4938271604938271,10.0])
vMatrix =
┌                                                                                                    ┐
│              0.0 0.4938271604938271 1.4814814814814812  4.444444444444445  6.666666666666666     0.0 │
│              0.0               10.0               10.0               10.0               10.0     0.0 │
└                                                                                                    ┘
```

Figure 6.8: Results for step 2 - Haskell

Results form Matlab - fig. 6.9:



```
---------------- V in every step of the loop -----------
V =

     0.0000e+00     6.6667e+00     4.4444e+00     1.4815e+00   493.8272e-03
     0.0000e+00    10.0000e+00    10.0000e+00    10.0000e+00    10.0000e+00
```

Figure 6.9: Results for step 4 - Matlab

Step 5 - final step

Figure 6.10 is a plot of both simulations for the voltage outputs of the V Matrix. There are no significant differences between the results and the curves are on the same place.

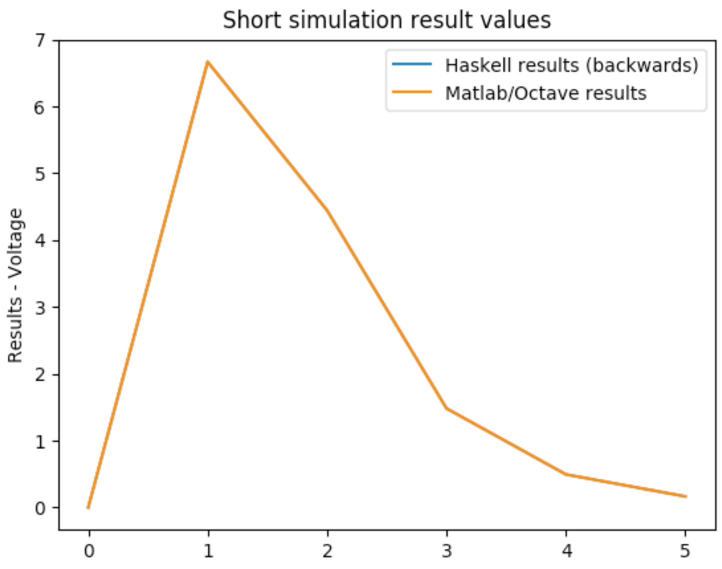


Figure 6.10: Short simulation results chart - V

Figure 6.11 and fig. 6.12 are the chars for IA and IB results, respectively:

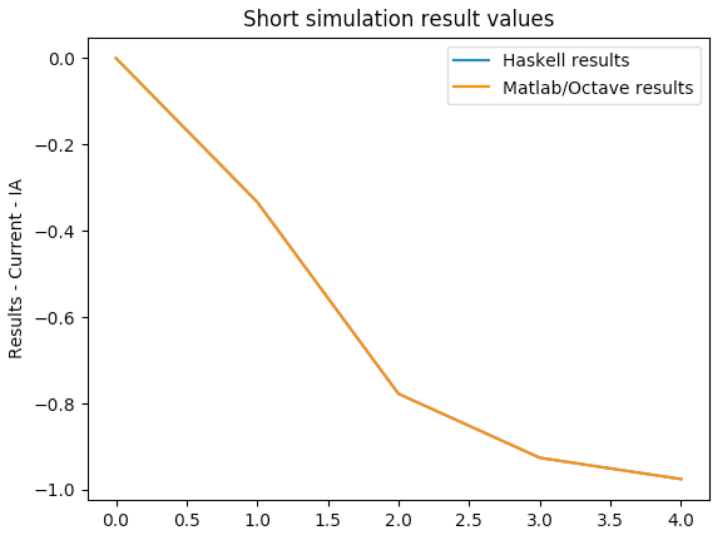


Figure 6.11: Short simulation results chart - IA

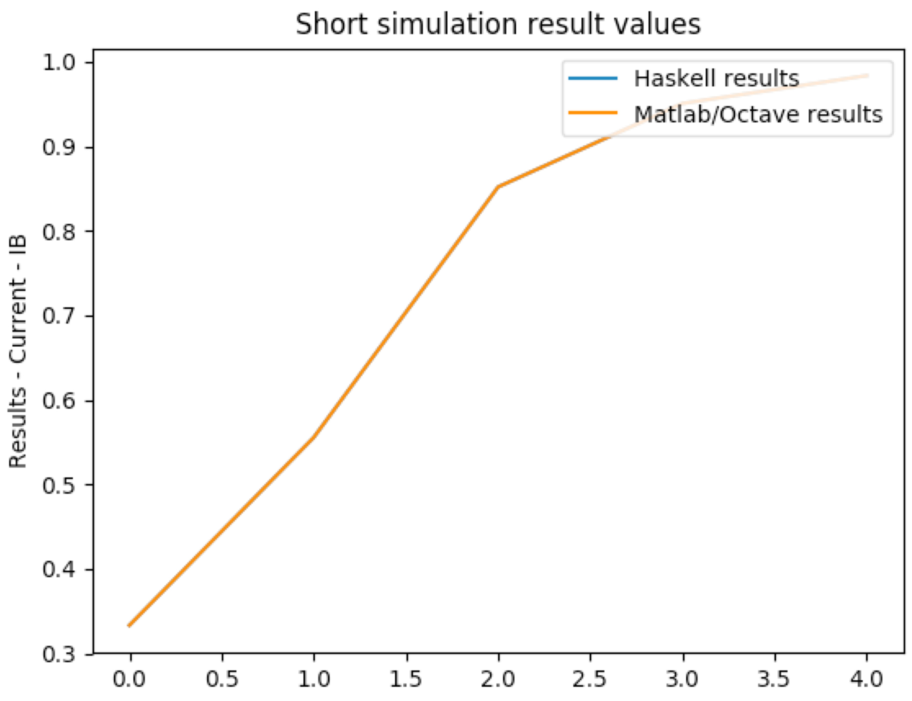


Figure 6.12: Short simulation results chart - IB

Note: The charts were plotted using Python to avoid bias towards the two languages during the comparison process.

The final V Matrix is described in table 6.1.

Step 5	Step 4	Step 3	Step 2	Step 1	Step 0
0.16460905349794236	0.4938271604938271	1.4814814814814812	4.444444444444445	6.666666666666666	0.0
10.0	10.0	10.0	10.0	10.0	0.0

Table 6.1: V Matrix calculated for the Haskell implementation. The results go from right to left. Each column represents one step of the iteration

The final V Matrix from the Matlab implementation is listed in table 6.2.

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5
0.0000e+00	6.6667e+00	4.4444e+00	1.4815e+00	493.8272e-03	164.6091e-03
0.0000e+00	10.0000e+00	10.0000e+00	10.0000e+00	10.0000e+00	10.0000e+00

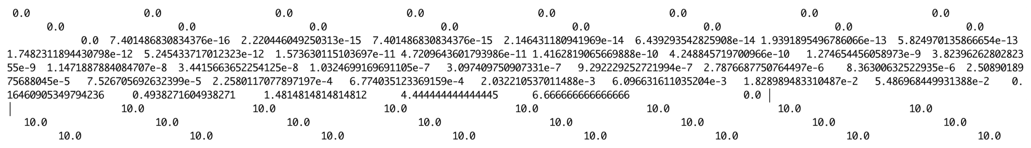
Table 6.2: V Matrix calculated for the Matlab implementation. The results go from left to right.

It is important to also compare the results from the I Vector. For the Haskell implementation, the final values are $[-0.9753086419753086, 0.9835390946502057]$ and for the Matlab ETR-P, the values are $[-975.3086e-03, 983.5391e-03]$. As expected, those are very close values (numerically).

6.1.2 Long simulation with DC Voltage

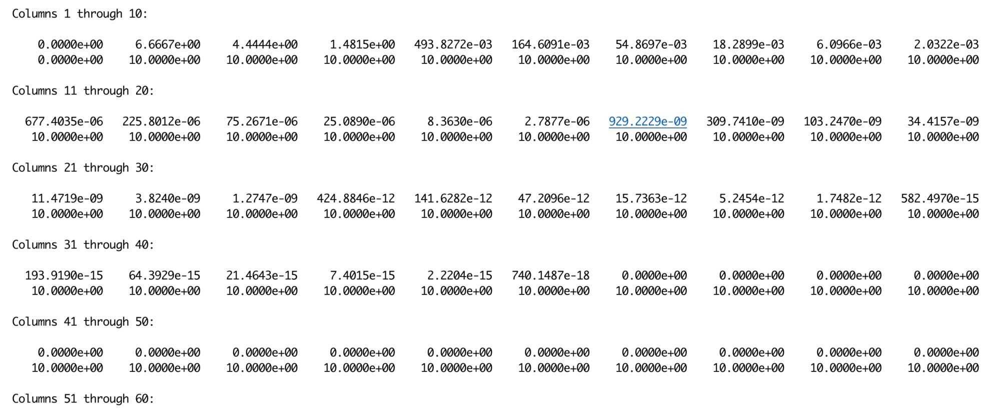
With some initial guarantees of a successful short simulation, it becomes reasonable to run a longer simulation (setting up `tmax` for 0.05) and then compare the results.

When analysing the final `V` Matrices of this longer simulation, it is possible to notice they will be both square and triangular matrices. Several values will be either 0.0 or 10.0 (EDC Voltage source). The transient, calculated values are compared in 6.13.



```
 0.0               0.0               0.0               0.0               0.0               0.0               0.0               0.0
     0.0               0.0               0.0               0.0               0.0               0.0               0.0               0.0
         0.0  7.401486830834376e-16  2.220446049250313e-15  7.401486830834376e-15  2.146431180941969e-14  6.439293542825908e-14 1.9391895496786066e-13  5.824970135866654e-13
1.7482311894430798e-12  5.245433717012323e-12  1.573630115103697e-11 4.7209643601793986e-11 1.4162819065669888e-10  4.248845719700966e-10   1.274654456058973e-9  3.82396262802823
55e-9  1.1471887884084707e-8  3.4415663652254125e-8  1.0324699169691105e-7   3.097409750907331e-7   9.292229252721994e-7  2.7876687750764497e-6   8.36300632522935e-6  2.50890189
75688045e-5   7.526705692632399e-5  2.2580117077897197e-4   6.774035123369159e-4   2.032210537011488e-3   6.096631611035204e-3   1.828989483310487e-2   5.486968449931388e-2    0.
16460905349794236     0.4938271604938271     1.4814814814814812     4.444444444444445     6.666666666666666                         0.0 |
|                10.0              10.0              10.0              10.0              10.0              10.0              10.0              10.0
  10.0              10.0              10.0              10.0              10.0              10.0              10.0              10.0
      10.0              10.0              10.0              10.0              10.0              10.0              10.0              10.0
```

Figure 6.13: Results for the triangular Matrix V in a broader simulation with Haskell



```
Columns 1 through 10:

   0.0000e+00    6.6667e+00    4.4444e+00    1.4815e+00  493.8272e-03  164.6091e-03   54.8697e-03   18.2899e-03    6.0966e-03    2.0322e-03
   0.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00

Columns 11 through 20:

 677.4035e-06  225.8012e-06   75.2671e-06   25.0890e-06    8.3630e-06    2.7877e-06  929.2229e-09  309.7410e-09  103.2470e-09   34.4157e-09
  10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00

Columns 21 through 30:

  11.4719e-09    3.8240e-09    1.2747e-09  424.8846e-12  141.6282e-12   47.2096e-12   15.7363e-12    5.2454e-12    1.7482e-12  582.4970e-15
  10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00

Columns 31 through 40:

 193.9190e-15   64.3929e-15   21.4643e-15    7.4015e-15    2.2204e-15  740.1487e-18    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00

Columns 41 through 50:

   0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
  10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00

Columns 51 through 60:
```

Figure 6.14: Results for the triangular Matrix V in a broader simulation with THTA Matlab

Compare the first value of the figure 6.13 (7.401486830834376e-16) with the last result which is not 0 at 6.14. By contrasting first and last pairs, it is possible to notice the values are very similar (numerically). It is then demonstrated that the program works as expected according to the ETR-P algorithm.

Figure 6.15 is a plot of both simulations for the voltage outputs of the V Matrix in this longer simulation. There are no significant differences between the results. Only the transient values appear in this plot to allow the value comparison.

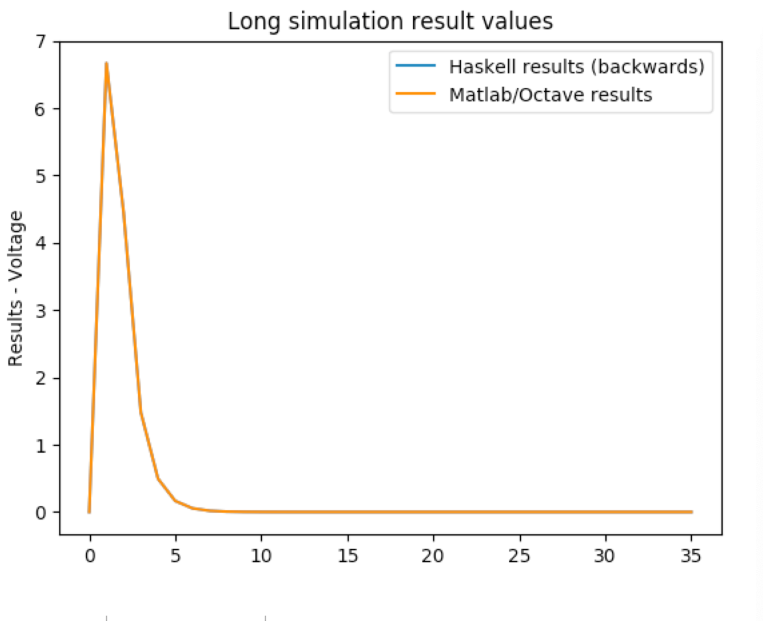


Figure 6.15: Long simulation results chart - V

Note: The chart was plotted using Python to avoid bias towards the two languages during the comparison process.

Once again the results are pretty accurate and result in two overlapping curves.

Next, fig. 6.16 and fig. 6.17 shows the results for the voltage outputs when setting up `tmax` for 0.01s and the step size of 0.000001s.

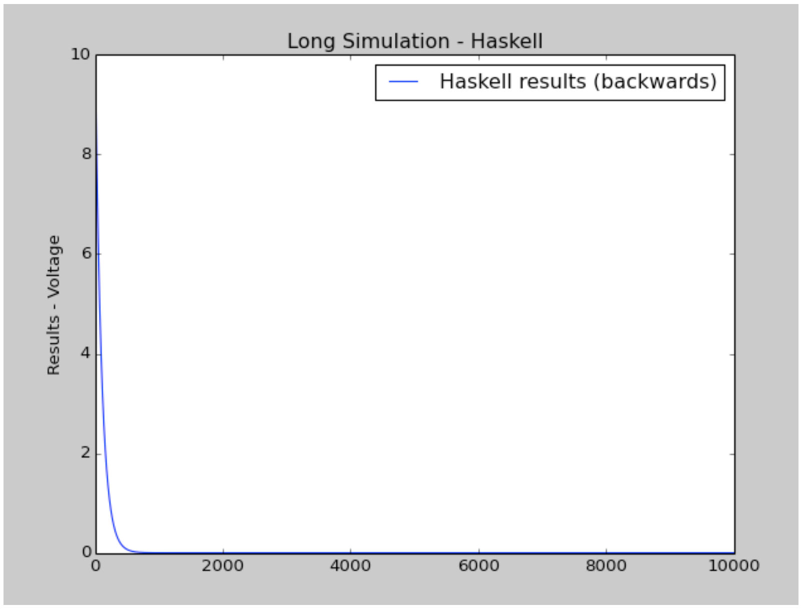


Figure 6.16: DC Circuit, Haskell implementation, step size of 0.000001s

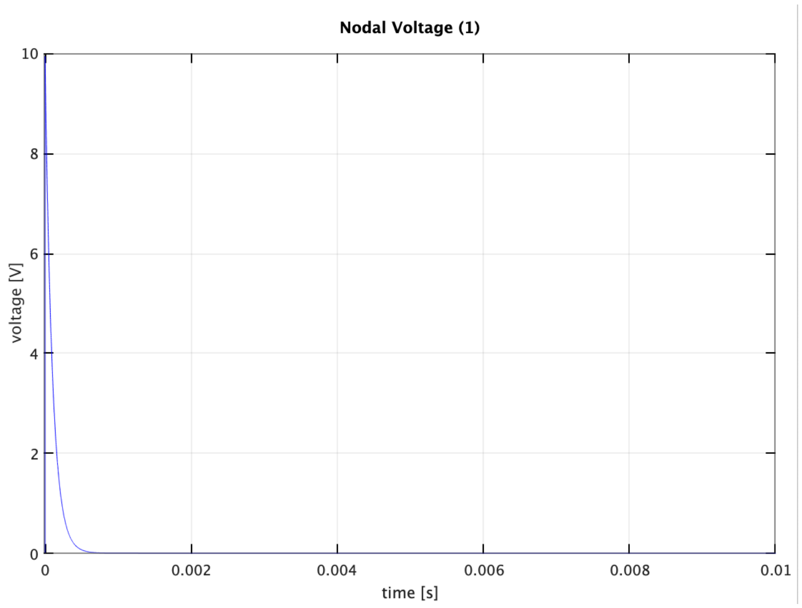


Figure 6.17: DC Circuit, Matlab implementation, step size of 0.000001s

6.1.3 Short simulation with AC Voltage

The results for the short AC simulation (with an EAC source component) are similar for both the implementations. Listing 6.6 is the configuration for the Matlab version, and listing 6.7 for the Haskell version. The circuit is described in fig. 6.18.

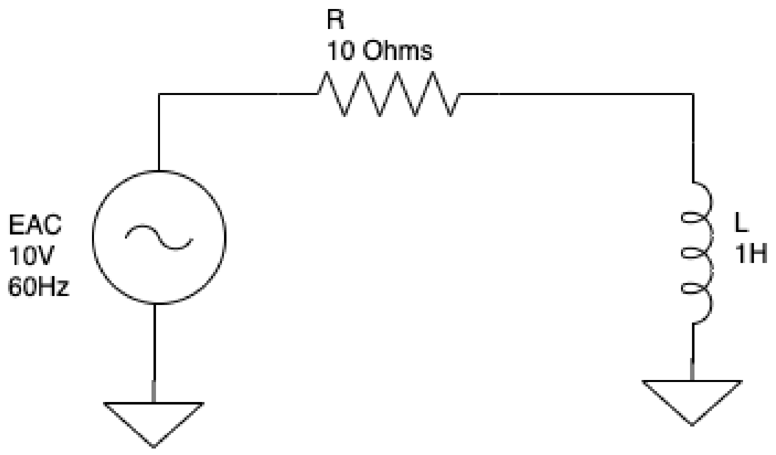


Figure 6.18: AC Circuit

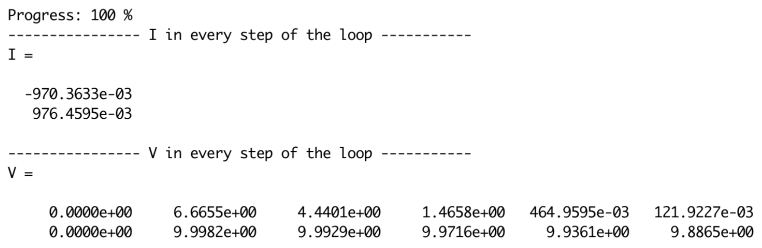
```
T 2 1 100E-6 5E-4 0 0 0 0 0
EAC 2 0 10 0 60 0 0 0 5
R 2 1 10 0 0 0 0 0 5
L 1 0 1 0 0 0 0 0 5
NV 1 2 0 0 0 0 0 0 0
```

Listing 6.6: Original input data file for ETR-P Matlab

```
Element Type,Node K,Node M,Value,Source param 1,Source param 2,Plot
EAC,2,0,10,0,60,0
R,2,1,10,0,0,0
L,1,0,1,0,0,0
```

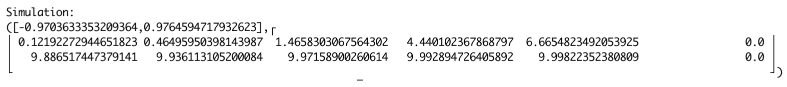
Listing 6.7: Input data file for components in the Haskell implementation

The results for the `I` Vector with this configuration in Matlab are `[-970.3633e ↪ -03, 976.4595e-03]` and `[-0.9703633353209364,0.9764594717932623]` for the Haskell implementation. 6.19 and 6.20 bring a comparison for the Voltage matrices in both implementations.



```
Progress: 100 %
---------------- I in every step of the loop -----------
I =

  -970.3633e-03
   976.4595e-03

---------------- V in every step of the loop -----------
V =

     0.0000e+00     6.6655e+00     4.4401e+00     1.4658e+00   464.9595e-03   121.9227e-03
     0.0000e+00     9.9982e+00     9.9929e+00     9.9716e+00     9.9361e+00     9.8865e+00
```

Figure 6.19: Matlab implementation - AC Voltage Source



```
Simulation:
([-0.9703633353209364,0.9764594717932623],┌
│ 0.12192272944651823 0.46495950398143987  1.4658303067564302   4.440102367868797  6.6654823492053925                0.0 │
│   9.886517447379141   9.936113105200084    9.97158900260614   9.992894726405892    9.99822352380809                0.0 │
└                                                                                                                          ┘)
```

Figure 6.20: Haskell implementation - AC Voltage Source

6.2 Simulation outputs - RC Circuit

6.2.1 Short simulation with DC Voltage

Running a short simulation for a RC Circuit with the setup specified on listing 6.8 and listing 6.9, it is possible to obtain the results listed on fig. 6.21. Once again the voltage values for the node 1 are the same in both Matlab (see fig. 6.22) and Haskell (see fig. 6.23) implementations.

```
Element Type,Node K,Node M,Value,Source param 1,Source param 2,Plot
EDC,2,0,10,0,60,0
R,2,1,10,0,0,0
C,1,0,1,0,0,0
```

Listing 6.8: Input data file for components in the Haskell implementation

```
Number of Nodes,Number of Voltages Sources,Step Size,Maximum time for simulation
2,1,0.0001,0.0005
```

Listing 6.9: Input data file for components in the Haskell implementation

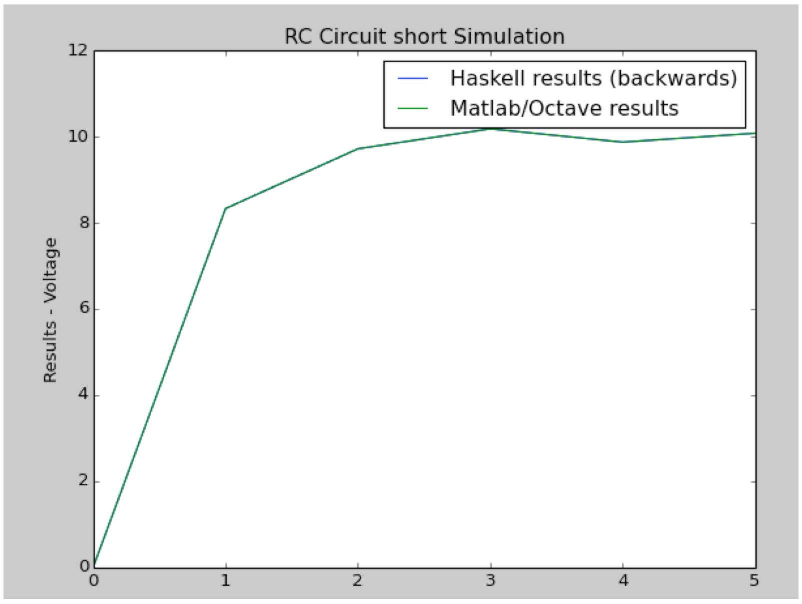
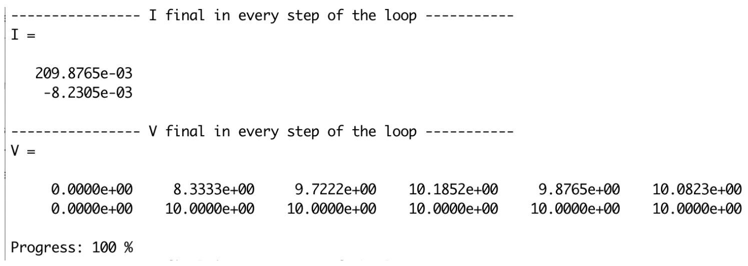
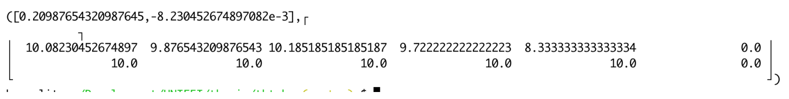


Figure 6.21: Comparison - DC Voltage Source for a RC Circuit



```
--------------- I final in every step of the loop ----------
I =

   209.8765e-03
    -8.2305e-03

--------------- V final in every step of the loop ----------
V =

     0.0000e+00    8.3333e+00    9.7222e+00   10.1852e+00    9.8765e+00   10.0823e+00
     0.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00   10.0000e+00

Progress: 100 %
```

Figure 6.22: Matlab results - DC Voltage Source for a RC Circuit



```
([0.20987654320987645,-8.230452674897082e-3],
 10.08230452674897  9.876543209876543 10.185185185185187  9.722222222222223  8.333333333333334                  0.0
              10.0               10.0               10.0               10.0               10.0                  0.0  )
```

Figure 6.23: Haskell results - DC Voltage Source for a RC Circuit

6.2.2 Short simulation with AC Voltage

Running a short simulation for a RC Circuit with an AC source according to the setup specified on listing 6.10 and listing 6.11, it is possible to obtain the results listed on fig. 6.24. Once again the voltage values for the node 1 are the same in both Matlab (see fig. 6.25) and Haskell (see fig. 6.26) implementations.

```
Element Type,Node K,Node M,Value,Source param 1,Source param 2,Plot
EAC,2,0,10,0,60,0
R,2,1,10,0,0,0
```

```
C,1,0,1,0,0,0
```

Listing 6.10: Input data file for components in the Haskell implementation

```
Number of Nodes,Number of Voltages Sources,Step Size,Maximum time for simulation
2,1,0.0001,0.0005
```

Listing 6.11: Input data file for components in the Haskell implementation

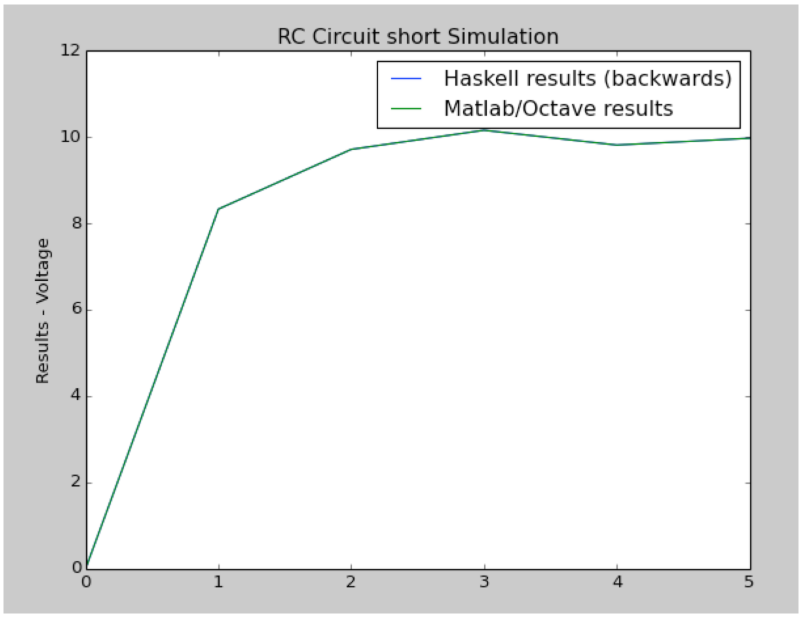
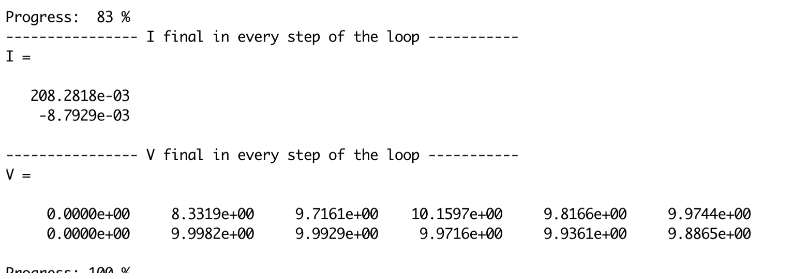
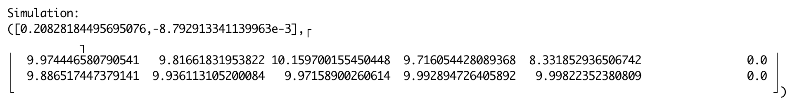


Figure 6.24: Comparison - AC Voltage Source for a RC Circuit



```
Progress:  83 %
---------------- I final in every step of the loop -----------
I =

   208.2818e-03
    -8.7929e-03

---------------- V final in every step of the loop -----------
V =

     0.0000e+00     8.3319e+00     9.7161e+00    10.1597e+00     9.8166e+00     9.9744e+00
     0.0000e+00     9.9982e+00     9.9929e+00     9.9716e+00     9.9361e+00     9.8865e+00

Progress: 100 %
```

Figure 6.25: Matlab results - AC Voltage Source for a RC Circuit



```
Simulation:
([0.20828184495695076,-8.792913341139963e-3],┌
                                      ┐
│  9.974446580790541   9.81661831953822 10.159700155450448  9.716054428089368  8.331852936506742              0.0 │
│  9.886517447379141  9.936113105200084   9.97158900260614  9.992894726405892   9.99822352380809              0.0 │
└                                                                                                                  ┘)
```

Figure 6.26: Haskell results - AC Voltage Source for a RC Circuit

6.3 Recursion problems

One of the downsides of the Haskell language was the performance of the recursive calls. Even though optimisation and execution time analysis are not the main focus of this work, it is important to mention that the overall execution time for the Haskell version of the ETR-P was slower, especially for the long simulations such as the one listed at section 6.1.2 for the step size of 0.000001s.

The Haskell compiler has some optimisations for dealing with recursion, such as *Tail Recursion*. By building the project with special optimisation flags, it is possible to obtain slower execution times for long simulations. `stack build --ghc-options -O2` results in longer compiler times, but in general it is possible to see a drop in the execution time. More details of other compiler flags con be found in the Haskell's GHC compiler documentation at [51].

6.4 Using the functional approach - summary

So far, the development of a fully functional engineering application under the functional paradigm was a challenging process especially because of the lack of references. Most of the existing implementations in the domain of electrical engineering follow the imperative style.

Choosing the appropriate types for every function was also challenging, but a careful selection guaranteed accurate results most of the time. In contrast, when developing using an imperative language, several bugs show up in runtime execution. With Haskell, most of the problems came during compile time.

Abandoning the idea of states and mutability allowed a more mathematical approach to the problem. When writing the functions, most of the time the main concern was building a meaningful and correct function chain to achieve the desired results. It was analogous to the process of developing a mathematical model to an experiment, but in the level of computer science.

The table 6.3 brings a comparison between the principal differences found when writing code in both of the paradigms.

Imperative Paradigm	**Functional Paradigm**
if/else blocks	Pattern Matching
for and while loops	Recursive calls
Focus on order	Focus on type
Most errors are runtime errors	Most errors are compile time errors
Temporary variables and binding	let/in construction

Table 6.3: Code comparison: Imperative and Functional paradigms

7 Conclusions and Future work

This work demonstrated an implementation of an electromagnetic transient analysis program using the functional programming paradigm. At chapter 1, a few questions were proposed. After completing this project, it has become possible to answer some of them or at least suggest a few discussions related to the matter. The upcoming sections bring a summary of this analysis.

What are the benefits of using a functional programming language?

For the test case adopted in this work, there were several benefits to this functional approach:

- During most of the time, when developing the necessary functions, it was possible to focus on the algorithm, on what the function needs as input data and what it should return as output values. It was not necessary to focus on technical aspects of the language that were not part of the role of the function, such as worrying about vector indexes, clearing temporary values, etc.
- There is not a single `if`/`else` block in the Haskell implementation.
- It was possible to approach the development of functions much more mathematically than programmatically. To compose a new function, it was necessary to determine the input parameters (equivalent to thinking of the **domain** of the function) and its output (like the **image** of the function).
- Treating software as a mathematical object is a strategy to seek and prove the correctness of the program.

What are the differences to the code base?

The code base size turned out to be concise - less than 300 lines of code were enough to implement the ETR-P Haskell version. The Matlab ETR-P version has more than 1000 lines of code (it supports more features than the Haskell version, but the number of features is not proportional).

There are many libraries written in Haskell. It is not necessary to write code from zero. There are libraries for web development, machine learning, control systems, matrices and a wide variety of other functions. Most of these libraries are open source and can be found at [52].

Compiling, building and running the project was a straightforward task with Stack [8], a cross-platform program for developing Haskell projects. A simple command like `stack run` would compile and run the project. The compile process took a few seconds (about 12 seconds).

What are the technical challenges?

There is decent support for Haskell at online forums, an essential resource when developing an application. Executing a Google search with an error message usually led to the right answer for the problem. Bleeding-edge technologies and languages may lack this resource. Haskell, on the other hand, is a mature language, and its online knowledge base is broad. Obtaining online information was not a problem. There are also several blog posts, articles from the language maintainers, videos, tutorials and publications in academic conferences.

However, there is a lack of examples for large software applications in the domain of engineering. Most of the material available is restricted to limited, fictional examples (like the ones used in Chapter 4 about polygons and shapes). The guides for real-world, large applications are not abundant.

The real technical challenge came during the development of the code. Sometimes, when writing functions, it was difficult to picture them under the ideas of functional programming principles, and the first draft result was very similar to the imperative style, requiring a massive refactoring work to achieve the functional style. Switching paradigms is not a completely obvious task.

Building the appropriate Data Types was also a challenge. Composing information in a structure that maximises not only the readability of the code but that also forms the right algebraic operations required several iterations of code refactoring.

Making the program compile when a new function was attached was more challenging than finding runtime errors. Most of the times when the program compiled, the output of the program was accurate. Strong and statically typed languages such as Haskell provide a crucial ally for the software engineer - the compiler. In this project, the Haskell compiler caught a vast number of what would have been bugs in runtime execution. This fact confirms the idea that "functional languages are associated with fewer defects than either procedural or scripting languages"[7].

Open source software

Creating an additional implementation of one of the algorithms to simulate electromagnetic transients that is open source might help future students to understand this approach in practice. They will have access to the code, being able to make changes on it, fix bugs and even implement their own features. Researches from other institutions can do the same: they will be able to use the code base for their projects, report issues and even contribute to the original project.

Open-source software is an excellent way to show the practical aspects of this work. Feedback from other engineers comes fast and concisely via Github, which also contains an issue tracker (updated to the actual state of this project, check it out at `https://github.com/hannelita/thtahs/issues`).

Notes on building software

The techniques to build software change from time to time. Not only the technologies but the paradigms, tools, organisations. Writing code responsibly is more than using bleeding-edge technology. The "Software Engineering Code of Ethics" [53] lists a few important principles that should be considered when building programs: "act consistently with the public interest", "managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance", "participate lifelong learning", "ensure their products and related modifications meet the highest professional standards possible". Breaking the monopoly of a single approach is acting ethically when the new projects reach higher standards.

Defining the meaning of "good software" is out of the scope of this work, but certainly, it involves innovation, combining appropriate techniques and ethics. This project took these principles into consideration.

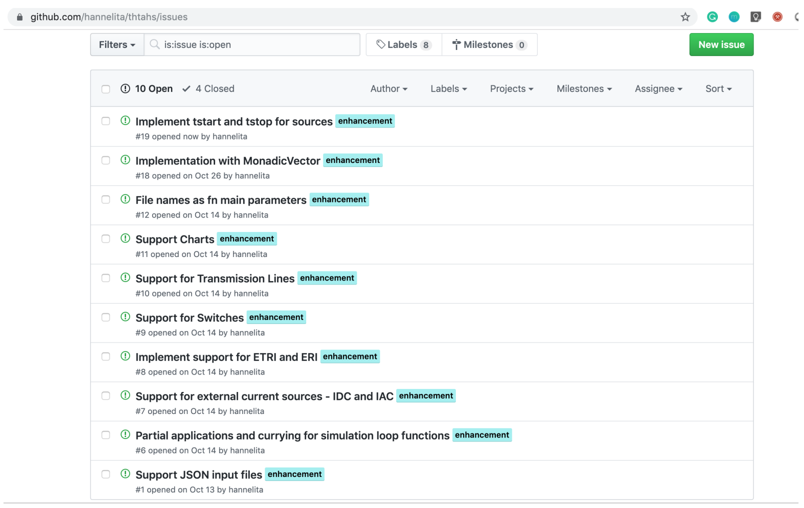
7.1 Future work

Functional programming is one of the pillars of an area so-called **Type Theory**. Together with Logic (Appendix B), λ-calculus (Appendix A) and Constructive Mathematics [25], it helps mathematicians and computer science to reason about code (Appendix C). Some open questions and insights were left out of the scope of this work, but it is important to encourage the readers to think about them:

- The mathematical models in engineering are based on the scientific method. There are experiments, results, approximations and the creation of mathematical representations. In real-world applications, the mechanics of these models is dumped into code written in programming languages. What are the guarantees that the code still respects the proposed model?
- Code Correctness is a topic that arises from Logic. How can one determine if the code is correct? Which type of Logic should be adopted to evaluate code correctness?
- Is it possible to perform an analysis on correctness using any programming language?
- How can one guarantee that their code is mathematically equivalent to the physical phenomena it represents?
- In Engineering, the same technique is used over and over again to solve different problems. For example, linearisation, differential equations, etc. Given that the mathematical models are the same, would it be possible to automate the code generation for these matters?

- When talking about correctness, it becomes essential to have proofs that determine the validity of the code. Ideally, these proofs also need to reflect the physical phenomena involved with the mathematical model it represents.

There is the possibility of implementing nonlinear components, switches, transmission lines and other elements which are present into ATP or EMTP. Charts and graphical interface for the end-user are other enhancements which can be added to the current program. In fig. 7.1 there is a complete list of open issues and features to be implemented in the future.


github.com/hannelita/thtahs/issues

Filters is:issue is:open Labels 8 Milestones 0 New issue

10 Open ✓ 4 Closed Author Labels Projects Milestones Assignee Sort

Implement tstart and tstop for sources enhancement
#19 opened now by hannelita

Implementation with MonadicVector enhancement
#18 opened on Oct 26 by hannelita

File names as fn main parameters enhancement
#12 opened on Oct 14 by hannelita

Support Charts enhancement
#11 opened on Oct 14 by hannelita

Support for Transmission Lines enhancement
#10 opened on Oct 14 by hannelita

Support for Switches enhancement
#9 opened on Oct 14 by hannelita

Implement support for ETRI and ERI enhancement
#8 opened on Oct 14 by hannelita

Support for external current sources - IDC and IAC enhancement
#7 opened on Oct 14 by hannelita

Partial applications and currying for simulation loop functions enhancement
#6 opened on Oct 14 by hannelita

Support JSON input files enhancement
#1 opened on Oct 13 by hannelita

Figure 7.1: Issue tracker on Github for the Haskell ETR-P project.

7.2 Final thoughts

This work was deeply related with abstraction and types. In fact, abstracting the code and abstracting the engineering model of the nodal equations were both joint into the implementation of this Haskell code. By finding ways to abstract the code, it is possible to find more concise ways to represent and explain the physical model. From the equation $[G][V] = [Ih]$ in chapter 2 to the main Haskell function signature in chapter 5 `thtaSimulation :: Vector ComponentData -> SimulationData -> ↪ SimulationResults` a lot of abstration was built. Are these two representations equivalent? This work does not provide any proof of it. But it provides the reader with the formal and technical tools to give an intuition on how to investigate this matter.

Academic works

Publications

Tavante HC, Bonatto BD, Coutinho MP. Open Source Implementations of Electromagnetic Transient Algorithms. In 2018 13th IEEE International Conference on Industry Applications (INDUSCON) 2018 Nov 12 (pp. 825-828). IEEE.

Participation in Conferences

- Programming Languages Mentoring Workshop (PLMW) at ICFP 2018 `https://icfp18.sigplan.org/track/PLMW-ICFP-2018`
- Student Volunteer at Principles of Programming Languages (POPL) 2019 `https://popl19.sigplan.org`
- Student Volunteer at European Conference on Object-Oriented Programming (ECOOP) 2019 `https://2019.ecoop.org`
- Student Volunteer at International Conference of Functional Programming (ICFP) 2019 `19.sigplan.org`

A Lambda Calculus And History of Functional Programming

A.1 Lambda Calculus basics

The roots of functional programming are strictly connected with mathematical logic (see Appendix B for more information). The formalisation of the predicate calculus allowed functions to generalize operations since this type of logic allowed non-logical values (like numbers or strings) in its predicates. Quantifiers ($\exists$ and $\forall$) granted the analysis of sequences, essential for the formalisation of lists and other functional data structures (see [54]). With the advances in number theory (especially the ones proposed by G. Peano), it was possible to see the relationship between induction and recursion. As [28] describes in his work, Russel and Withehead, then Hilbert, then Gödel pushed logics and mathematical foundations forward, giving the first steps to the **theory of computability**.

Three distinct formal approaches to the theory of computability were proposed in the 1930s: *Turing machines*, by A. Turing (see [55] for further details), *recursive function theory*, by S. Kleene and the *Lambda calculus* (or λ-calculus), by A. Church. All these three approaches are equivalent to each other. They are all able to generalise von Neumann's machines (digital computers). Turing machines are based on assignments and time-ordered evaluation. The other techniques treat computations as structured function application.

In the 1960s, the LISP language is created, inspired by the λ calculus. Even though LISP is a straightforward language, it built up the idea of translating high-level functional languages (such as Haskell) in a notation equally simple, based on the lambda-calculus [56]. The λ calculus is based on function abstractions (which generalise expressions) and on function application (which perform the evaluation of the expressions). The name "calculus" is not in vain - it has properties and rules to describe programming languages. The abstraction of λ can "be treated as a universal machine code for programming languages" [28]. A.1 brings an example of the notation in the λ calculus.

```
( λ x . + x 1)
```

Listing A.1: Lambda abstraction

In A.1, the λ indicates a function, with one variable (`x`), just as the mathematical expression $f(x)$ represents a single-variable function. The dot `.` represents the beginning of the body of the function, just as the $=$ in maths. Then an expression in prefix form (operator, operands) shows up; in this case, `+ x 1`, meaning that this function adds one to the variable `x`. `x` is called **bound variable**; it works like a place-holder and will be replaced by an argument when this expression is evaluated. In listing A.2, 4 is the argument. It is also possible to see another component, `y`, which is called **free variable**.

```
( λ x . + x y) 4
```

Listing A.2: Free variable: y; Bound Variable: x

Beta-reduction (Or β-reduction) is the operation of applying an argument to an expression (A.3).

```
( λ x . + x 1) 4 → (+ 4 1)
```

Listing A.3: Beta reduction

The β-reduction can be used "backwards", composing a β-abstraction A.4. The set of β reduction/abstraction is know as β-conversion.

```
(+ 4 1) ← ( λ x . + x 1) 4
```

Listing A.4: Beta abstraction

Intuitively, there is also an α-conversion. This operation refers to variable name substitution, keeping the operations equivalent A.5.

```
( λ x . + x 1) ⟷α ( λ y . + y 1)
```

Listing A.5: Alpha conversion

Expressions can behave equivalently with slightly different bodies when applied to the same arguments. The Eta-conversion (or η-conversion) is the rule describing this scenario A.6.

```
( λ x . + x 1) ⟷η ( + 1)
```

Listing A.6: Eta conversion

There are other rules, but this Appendix will be restricted to these three, which are the most essential for the λ-calculus. After applying these rules, if an expression does not contain any other sub-expressions to be reduced, this expression is in the **Normal Form**, or NF, for short. There are several strategies to reduce an expression. Only some paths lead to normal form. A guaranteed strategy to reach the NF of an expression of the **Normal Order Reduction**. It specifies that "the leftmost, outermost redex should be reduced first" [56]. An outermost evaluation may require extra steps, but it will terminate if there is any path that leads the program to terminate.

There are multiple ways to analyse a function in terms of its semantics, listed in A.1 [57].

1. Interpreting the function as a sequence of operations in time, analysing it under operational aspects. This is a 'dynamic' view of the function, called **operational semantics**. This type of semantics specifies an abstract machine. A state relates to a term, and a behaviour relates to the transition function.

2. Interpret the function 'statically' as a set of arguments and values to be determined. This is called **Denotational semantics**. The meaning of a term is a mathematical object. It assigns a value to every expression in the language.

3. **Axiomatic semantics** states that the meaning of a term is what can be proved about it.

The λ-calculus can be taken into operational and denotational definitions. Denotational semantics is essential because it allows reasoning about the termination of evaluation (formally represented by $\perp$).

When a function needs the value of its argument, it is said to be *strict*. The opposite, the function that does not request the value of its arguments, is called *lazy*.

Lambda-Calculus is a vast topic and its content is much more extensive than the ones listed in A.1. See [58] and [59] for further references.

A.2 Translating programs into Lambda Calculus

In section A.1, the basic tools and principles of Lambda Calculus were presented in a summarised format. It is possible to translate codes written using the functional paradigm into Lambda-calculus. This is the first step to analyse code under a mathematical aspect. [56] suggests two approaches for the translation:

1. Perform successive transformations from one functional program to another, with a simplification in each step. The final simplified version of each function could, in theory, be translated to pure Lambda-Calculus. This approach focuses on syntax

2. Start by translating the program into an 'enriched' version of the Lambda-Calculus (a version of λ-calculus that includes the original one and additional features). From this enriched version, perform simplifications until reaching the simple version of the λ-calculus. This approach focuses on semantics.

This work will not describe these techniques in-depth, nor implement such translation.

B Logic and Mathematics

In order to analyse the functional code as a mathematical object, it is important to think about an appropriate Logic. In classical mathematics, **Propositional Logic** uses connectives (and, or, not, etc) to validate tautologies through proofs (or derivations). A complete guide to all propositional logic domain (such as signed formulas, construction of tableaux, logical consequence analysis, axioms, correctness, completeness and compactness) can be found at [60]. This type of logic is limited to a proposition and connectives, which (for a well formed formula) will be either true or false.

Predicate Logic, or First Order Logic, adds two important components to the analysis - the existential quantifiers $\forall$ and $\exists$ and the introduction of variables to the predicates (example: $\exists xGx$). This type of logic has some similarities with the λ-Calculus (see A), such as the occurrence of free and bound variables and substitution rules.

Are there any other types of logic which would be more suitable for analysing programs? It is necessary to dig into mathematics to try to answer this question. It might be necessary to take some of the foundations of mathematics into consideration. Frege made a few, unsuccessful attempts to create an 'arithmeticization' of analysis ([61]). For Frege, logic is the study of truth-preserving inferences and concepts are ontological counterparts of predicative expressions. Concepts are functions from objects to truth values. A concepts can be categorized as:

- **First-level concepts** - Obtained by removing a name from a declarative sentence, originating a *first level predicate.*
- **Recognition statements** - since any expression can be part of infinitely many sentences, a recognition statement is a concept acting as a wild-card for this case.
- **Equivalence relations** - Frege's logicism aims to build gap-free demonstrations by partitioning the universe in clusters by equinumerosity. This analysis is a recursive characterization of natural numbers (which are clusters themselves). These clusters are equivalence relations.

Frege's model main concern arises of the need of logical objects. How to define these logical objects? Frege creates another concept, the **extensions**. By proposing that every object has an extension in an axiom, Frege runs into a paradox where some sets would be member of themselves and some wouldn't, creating an inconsistency in his general proposal. This is widely known as **Russel's paradox**.

Russel and Withehead investigated this matter and proposed an alternative, the *Ramified Theory of Types*, which does not rely on sets, but on propositional functions. In this theory, all objects are classified into a hierarchy of types. Unfortunately, this approach bans unharmful sets (paradox-free). Most mathematicians prefer to use alternative theories of sets which are free of paradox and do not require the classification of all entities into types. The most popular model following this proposals is the ZFC (Zermelo-Frankel and the Axiom of Choice model). Note that ZFC is only concerned about sets, whereas Russel's main goal is to show how logic can be applied to general statements. ZFC is a formal axiomatic theory: there is a language to write statements. There is no guarantee that ZFC is paradox-free (no proof it is consistent).

These models presented previously somehow rely on the idea of a set. But a set might have issues according to realism. In philosophy, *realism* states that some propositions might be nor true of false: there is no known answer. The *Law of the Excluded Middle* (wither S is true or not-S is true) might not be valid. The "anti-realist" conception of mathematics is called **constructivism**. **Intuitionism** is one type of constructivism, which suggests that mathematical reality is not fully fixed. Infinitude is potential. Intuitionism contasts with classical mathematics and logic for not accepting infinites (see [62]), nor requiring a true/false result.

Intuitionistic mathematics has its own rules, and some of the axioms of the classical logic become invalid under the intuitionistic perspective. When analysing programs, this is one type of logic that could be a potential candidate for adoption. In practical terms, the classical mathematician has a fixed mathematical reality and an actual infinite. The intuitionistic determines a potential infinite. Translating these differences to proofs, the intuitionistic model has the advantage of establishing finite proofs (which are fairly desirable when aiming to termination). Intuitionism imposes constraints to the universal quantifier $\forall$. In intuitionism, the double negation elimination is not valid:

$$\frac{\neg\neg X}{X}$$

In contrast, there is an intuitionistic version which is valid:

$$\frac{X \vee \neg X \qquad \neg\neg X}{X}$$

This chapter does not brings the topic of constructive mathematics in detail, but when combined with functional programming, λ-calculus and logic, it becomes the baseline for Type Theory, quickly described in C. Using the words of [25] to summarise the relevance of this appendix, "the short discussion of constructive mathematics introduced the idea that proofs should have computational content; [...] to achieve this goal, the underlying logic of the system needed to be changed to one in which we only assert the validity of a proposition when we have a proof of the proposition."

C Types and Proofs

A type system is a tool for reasoning about programs. When built using the appropriate logic system, it can avoid paradoxes such as Russel's paradox (B). Type systems help to eliminate run-time errors, by catching inconsistencies (paradoxes) in compile time. In order to analyse functions at this level, it is necessary to see programs as proofs. This Appendix will briefly explore the duality between propositions and types, proofs and elements. A proof of a proposition T is isomorphic to the type T. "a proof by induction is nothing other than a proof object defined using recursion" [25]. A program then becomes a set of terms built by the elements in a certain grammar. From there, it is possible to derive the necessary inference rules. It is with the aid of λ-calculus that the syntax of the language is analysed and transformed into an abstract syntax tree (AST).

This process of formalizing types is detailed described in [57]. As an example, the (simply-typed) λ-Calculus over the type Bool if described as C.1.

$$T ::= \tag{C.1a}$$

$$Bool \tag{C.1b}$$

$$T \rightarrow T \tag{C.1c}$$

This process is extended to all of the base types in the language (such as Integer, Float, Char, etc), as well as collections (Lists, Tuples, etc). Impure functions and exceptions also have corresponding types. The relation between types is also analysed, such as subtyping and polymorphism.

This appendix could bring an extensive buzzword list (recursive types, equi-recursive types, type reconstruction, Universal types, existential types, higher order polymorphism, dependent types). Explaining these concepts in detail is out of the scope of this work, but asking which system would best describe (in terms of paradox-free and expressiveness) the implemented software is a pertinent question. A wider comparison between implementations using the formalism of type theory is a future work. To conclude this brief appendix, it is possible to quote [25]: "There is still much to be done in making type theory a usable and attractive system which supports programming in the large, but I am certain that languages based on type theory will be as popular in a few years as contemporary functional languages".

D Haskell ETR-P source code

Source code for the `main.hs` file.

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module Main where
import qualified Data.Foldable as Foldable

-- vector
import Data.Vector (Vector)
import qualified Data.Vector as Vector

-- Matrix
import Data.Matrix (Matrix)
import qualified Data.Matrix as Matrix

-- HMatrix
import qualified Numeric.LinearAlgebra.Data as HMatrix
import qualified Numeric.LinearAlgebra.HMatrix as HMatrix

-- bytestring
import Data.ByteString.Lazy (ByteString)
import qualified Data.ByteString.Lazy as ByteString

-- cassava
import Data.Csv
  ( DefaultOrdered(headerOrder)
  , FromField(parseField)
  , FromNamedRecord(parseNamedRecord)
  , Header
  , ToField(toField)
  , ToNamedRecord(toNamedRecord)
  , (.:)
  , (.=)
  )
import qualified Data.Csv as Cassava

-- text
import Data.Text (Text)
import qualified Data.Text.Encoding as Text

-- Exception
import Control.Exception

-- base
import qualified Control.Monad as Monad
import qualified System.Exit as Exit
import qualified Debug.Trace as Trace



data ComponentData =
  ComponentData {
    componentType :: ComponentType,
    nodeK :: Int,
    nodeM :: Int,
    magnitude :: Double,
    param1 :: Double,
    param2 :: Double,
    plot :: Int
     }
  deriving (Eq, Show)


data SimulationData =
  SimulationData {
    nodes :: Int,
    voltageSources :: Int,
    stepSize :: Double,
    tmax :: Double
     }
  deriving (Eq, Show)

data ComponentType = Resistor | Capacitor | Inductor | EAC | EDC | Other Text
    ↪ deriving (Eq, Show)
type SimulationResults = (Vector Double, Matrix Double)

instance FromNamedRecord SimulationData where
  parseNamedRecord m =
    SimulationData
      <$> m .: "Number of Nodes"
      <*> m .: "Number of Voltages Sources"
      <*> m .: "Step Size"
      <*> m .: "Maximum time for simulation"


instance FromNamedRecord ComponentData where
  parseNamedRecord m =
    ComponentData
      <$> m .: "Element Type"
      <*> m .: "Node K"
      <*> m .: "Node M"
      <*> m .: "Value"
      <*> m .: "Source param 1"
      <*> m .: "Source param 2"
      <*> m .: "Plot"

instance FromField ComponentType where
  parseField "R" =
    pure Resistor

  parseField "L" =
    pure Inductor

  parseField "C" =
    pure Capacitor

  parseField "EDC" =
    pure EDC

  parseField "EAC" =
    pure EAC

  parseField otherType =
    Other <$> parseField otherType


decodeItems :: ByteString -> Either String (Vector ComponentData)
decodeItems =
  fmap snd . Cassava.decodeByName

decodeItemsFromFile :: FilePath -> IO (Either String (Vector ComponentData))
decodeItemsFromFile filePath =
  catchShowIO (ByteString.readFile filePath)
    >>= return . either Left decodeItems

decodeSimulation :: ByteString -> Either String (Vector SimulationData)
decodeSimulation =
  fmap snd . Cassava.decodeByName

decodeSimulationFromFile :: FilePath -> IO (Either String (Vector SimulationData)
    ↪ )
decodeSimulationFromFile filePath =
  catchShowIO (ByteString.readFile filePath)
    >>= return . either Left decodeSimulation

getSingleSimulationLine :: Vector SimulationData -> SimulationData
getSingleSimulationLine =
    Vector.head


nhComponents :: [ComponentData] -> [ComponentData]
nhComponents =
  filter (\r -> (componentType r == Capacitor) || (componentType r == Inductor))

filterEnergyStorageComponent :: Vector ComponentData -> Vector ComponentData
filterEnergyStorageComponent =
  Vector.filter (\r -> (componentType r == Capacitor) || (componentType r ==
      ↪ Inductor))

nh :: Vector ComponentData -> Int
nh components =
  length $ filterEnergyStorageComponent components

```

```haskell
filterSources :: Vector ComponentData -> Vector ComponentData
filterSources =
  Vector.filter (\r -> (componentType r == EDC) || (componentType r == EAC))

condutance :: ComponentData -> Double -> Double
condutance component dt =
  case componentType component of
    Resistor -> 1.0 / (magnitude component)
    Capacitor -> (magnitude component) * 0.000001 * 2 / dt
    Inductor -> dt / (2 * 0.001 * (magnitude component))
    _ -> 0.0

gkm :: Vector ComponentData -> Double -> Vector Double
gkm components dt =
  Vector.map (\c -> condutance c dt) components

buildCompactGMatrix :: Double -> [ComponentData] -> Matrix Double -> Matrix
    ↪ Double
buildCompactGMatrix dt [] buffer = buffer
buildCompactGMatrix dt (component:cs) buffer =
  case (nodeK component, nodeM component) of
    (0, m) -> buildCompactGMatrix dt cs (Matrix.setElem (Matrix.getElem m m
        ↪ buffer + condutance component dt) (m, m) buffer)
    (k, 0) -> buildCompactGMatrix dt cs (Matrix.setElem (Matrix.getElem k k
        ↪ buffer + condutance component dt) (k, k) buffer)
    (k, m) -> buildCompactGMatrix dt cs (Matrix.setElem (Matrix.getElem k k
        ↪ buffer + condutance component dt) (k, k) (Matrix.setElem (Matrix.
        ↪ getElem m m buffer + condutance component dt) (m, m) (Matrix.setElem (
        ↪ Matrix.getElem k m buffer - condutance component dt) (k, m) (Matrix.
        ↪ setElem (Matrix.getElem m k buffer - condutance component dt) (m, k)
        ↪ buffer))))
    (_, _) -> buildCompactGMatrix dt cs buffer


buildGMatrixFromVector :: SimulationData -> Vector ComponentData -> Matrix Double
buildGMatrixFromVector simulation components =
  buildCompactGMatrix (stepSize simulation) (Vector.toList components) (Matrix.
      ↪ zero (nodes simulation) (nodes simulation))


buildIhVector :: [ComponentData] -> Double -> Int -> [Double] -> [Double] ->
    ↪ Matrix Double -> Vector Double
buildIhVector [] _ _ _ ihnew _ = Vector.fromList ihnew
buildIhVector (component:cs) dt n (hold:ihold) ihnew vMatrix =
  case (componentType component, nodeK component, nodeM component) of
    (Inductor, 0, m) -> buildIhVector cs dt n ihold (ihnew ++ [(2*(condutance
        ↪ component dt)*(Matrix.getElem m n vMatrix) + hold)]) vMatrix
    (Inductor, k, 0) -> buildIhVector cs dt n ihold (ihnew ++ [(-2*(condutance
        ↪ component dt)*(Matrix.getElem k n vMatrix) + hold)]) vMatrix
    (Inductor, k, m) -> buildIhVector cs dt n ihold (ihnew ++ [(-2*(condutance
        ↪ component dt)*((Matrix.getElem k n vMatrix) - (Matrix.getElem m n
        ↪ vMatrix)) + hold)]) vMatrix
    (Capacitor, 0, m) -> buildIhVector cs dt n ihold (ihnew ++ [(-2*(condutance
        ↪ component dt)*(Matrix.getElem m n vMatrix) - hold)]) vMatrix
    (Capacitor, k, 0) -> buildIhVector cs dt n ihold (ihnew ++ [(2*(condutance
        ↪ component dt)*(Matrix.getElem k n vMatrix) - hold)]) vMatrix
    (Capacitor, k, m) -> buildIhVector cs dt n ihold (ihnew ++ [(2*(condutance
        ↪ component dt)*((Matrix.getElem k n vMatrix) - (Matrix.getElem m n
        ↪ vMatrix)) - hold)]) vMatrix
    (_, _, _) -> buildIhVector cs dt n ihold ihnew vMatrix



buildVBVector :: [ComponentData] -> Double -> [Double] -> Vector Double
buildVBVector [] _ buffer = Vector.fromList buffer
buildVBVector (c:components) time buffer =
  case (componentType c) of EDC -> buildVBVector components time ((magnitude c) :
      ↪  buffer)
                           EAC -> buildVBVector components time (((magnitude c *
                               ↪ cos (2 * pi * param2 c * time + (param1 c * (pi
                               ↪ /180))))) : buffer)
                           _ -> buildVBVector components time buffer


buildIVector :: [ComponentData] -> [Double] -> Vector Double -> Vector Double
buildIVector [] _ iVector = iVector
buildIVector (component:cs) (ihEl:ih) iVector =
  case (componentType component, nodeK component, nodeM component) of
    (Inductor, k, 0) -> buildIVector cs ih (iVector Vector.// [((k - 1), ((
        ↪ iVector Vector.! (k-1)) + ihEl))])
    (Inductor, 0, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
        ↪ iVector Vector.! (m-1)) - ihEl))])
    (Inductor, k, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
        ↪ iVector Vector.! (m-1)) - ihEl)), ((k-1), ((iVector Vector.! (k-1)) +
        ↪ ihEl))])
    (Capacitor, k, 0) -> buildIVector cs ih (iVector Vector.// [((k - 1), ((
        ↪ iVector Vector.! (k-1)) + ihEl))])
    (Capacitor, 0, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
        ↪ iVector Vector.! (m-1)) - ihEl))])
    (Capacitor, k, m) -> buildIVector cs ih (iVector Vector.// [((m - 1), ((
        ↪ iVector Vector.! (m-1)) - ihEl)), ((k-1), ((iVector Vector.! (k-1)) +
        ↪ ihEl))])
    (_, _, _) -> buildIVector cs ih iVector


thtaControl :: Int -> Double -> Vector Double -> Vector Double -> SimulationData
    ↪ -> (Int, Vector Double, Double)
thtaControl thtactl time ihnew ih simulation
  | thtactl <= 0 = (thtactl, ihnew, (stepSize simulation + time))
  | thtactl < 3 = (thtactl + 1, (Vector.map (\i -> i/2) $ Vector.zipWith (+) ih
      ↪ ihnew), (time + (stepSize simulation/2)))
  | otherwise = (0, ihnew, (stepSize simulation + time))

fromHMatrixTransformer :: HMatrix.Matrix Double -> Matrix Double
fromHMatrixTransformer matrix =
  Matrix.fromLists $ HMatrix.toLists matrix

toHMatrixTransformer :: Matrix Double -> HMatrix.Matrix Double
```

```haskell
toHMatrixTransformer matrix =
  HMatrix.fromLists $ Matrix.toLists matrix

fromHMatrixVectorTransformer :: HMatrix.Vector Double -> Vector Double
fromHMatrixVectorTransformer vec =
  Vector.fromList $ HMatrix.toList vec

toHMatrixVectorTransformer :: Vector Double -> HMatrix.Vector Double
toHMatrixVectorTransformer vec =
  HMatrix.fromList $ Vector.toList vec

solver :: HMatrix.Vector Double -> HMatrix.Matrix Double -> HMatrix.Matrix Double -> HMatrix.Matrix Double -> HMatrix.Matrix Double -> HMatrix.Vector Double -> SimulationData -> (Vector Double, Vector Double)
solver iVector gaa gab gba gbb vb simulation =
  let ia = HMatrix.subVector 0 ((nodes simulation) - (voltageSources simulation)) iVector
      rhsa = ia - (gab HMatrix.#> vb)
      va = gaa HMatrix.<\> rhsa
      ib = (gba HMatrix.#> va) + (gbb HMatrix.#> vb)
      iVec = HMatrix.vjoin [ia, ib]
      vVec = HMatrix.vjoin [va, vb]
  in
    ((fromHMatrixVectorTransformer iVec), (fromHMatrixVectorTransformer vVec))


thtaSimulationStep :: [ComponentData] -> Matrix Double -> SimulationData -> Int -> Int -> Double -> Vector Double -> Matrix Double -> Vector Double -> Vector Double -> SimulationResults
thtaSimulationStep _ _ _ _ 1 _ _ vMatrix _ iVector = (iVector, vMatrix)
thtaSimulationStep components condutances simulation thtactl n time ih vMatrix vbVector iVector =
  let (gaa, gab, gba, gbb) = Matrix.splitBlocks (nodes simulation - voltageSources simulation) (nodes simulation - voltageSources simulation ) condutances
      ihBuffer = buildIhVector (nhComponents components) (stepSize simulation) n (Vector.toList ih) [] vMatrix
      (thta, ihThta, timeThta) = thtaControl thtactl time ihBuffer ih simulation
      vbVec = buildVBVector components timeThta []
      iVec = buildIVector (nhComponents components) (Vector.toList ihThta) (Vector.replicate (nodes simulation) 0)
      (iVecCalc, vVec) = solver (toHMatrixVectorTransformer iVec) (toHMatrixTransformer gaa) (toHMatrixTransformer gab) (toHMatrixTransformer gba) (toHMatrixTransformer gbb) (toHMatrixVectorTransformer vbVec) simulation
      vMatr = Matrix.mapCol (\r _ -> vVec Vector.! (r - 1)) (n-1) vMatrix
  in
      thtaSimulationStep components condutances simulation thta (n-1) timeThta ihThta vMatr vbVec iVecCalc


thtaSimulation :: Vector ComponentData -> SimulationData -> SimulationResults
thtaSimulation components simulation =
  thtaSimulationStep (Vector.toList components) (buildGMatrixFromVector
    ↪ simulation components) simulation 1 (npoints simulation) 0.0 (Vector.
    ↪ replicate (nh components) 0) (Matrix.zero (nodes simulation) (npoints
    ↪ simulation)) (Vector.replicate (voltageSources simulation) 0) (Vector.
    ↪ replicate (nodes simulation) 0)


npoints :: SimulationData -> Int
npoints sim =
  round ((tmax sim)/(stepSize sim)) + 1

catchShowIO :: IO a -> IO (Either String a)
catchShowIO action =
  fmap Right action
    `catch` handleIOException
  where
    handleIOException :: IOException -> IO (Either String a)
    handleIOException =
      return . Left . show

main :: IO ()
main = do

  eitherSimulation <-
      fmap getSingleSimulationLine
        <$> decodeSimulationFromFile "data/simulation.csv"

  case eitherSimulation of
    Left reason ->
      Exit.die reason

    Right simulation -> do
      components_list <- decodeItemsFromFile "data/components.csv"
      case components_list of
        Left reason -> Exit.die reason
        Right components -> do
          let gmt = buildGMatrixFromVector simulation components
          putStr "GMatrix: \n"
          print (gmt)
          let results = thtaSimulation components simulation
          putStr "Simulation: \n"
          print (results)
```

Listing D.1: Main.hs file code

Bibliography

[1] J. E. Sammet, "Programming languages: history and future," *Communications of the ACM*, vol. 15, no. 7, pp. 601–610, 1972.

[2] K. R. Parker and B. Davey, "The history of computer language selection," pp. 166–179, 2012.

[3] IBM, "Programming languages history." Accessed: 2019-12-01.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to design programs: an introduction to programming and computing.* MIT Press, 2018.

[5] D. Turner, "Church's thesis and functional programming," *Church's Thesis after*, vol. 70, pp. 518–544, 2006.

[6] B. D. Bonatto *et al.*, "Thta octave." `https://github.com/aptis/THTA`, 2018.

[7] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 155–165, ACM, 2014.

[8] "Haskell stack." `https://docs.haskellstack.org/en/stable/README/`. Accessed: 2019-12-01.

[9] "Haskell cabal." `https://www.haskell.org/cabal/`. Accessed: 2019-12-01.

[10] J. A. Martinez-Velasco, "Introduction to electromagnetic transient analysis of power systems," *Transient Analysis of Power Systems: Solution Techniques, Tools and Applications*, pp. 1–8, 2015.

[11] H. W. Dommel, "Digital computer solution of electromagnetic transients in single-and multiphase networks," *IEEE transactions on power apparatus and systems*, no. 4, pp. 388–399, 1969.

[12] ATP, "Atp." `https://www.eeug.org/index.php/about-eeug/about-atp`. Accessed: 2019-12-01.

[13] ATP, "Atp license." `https://www.eeug.org/index.php/form-europe`. Accessed: 2019-12-01.

[14] A. Draw, "Atpdraw." `http://www.atpdraw.net`. Accessed: 2019-12-01.

[15] EMTP, "Emtp-rv." `https://www.emtp-software.com/en/products/emtp`. Accessed: 2019-12-01.

[16] J. Lin and J. R. Marti, "Implementation of the cda procedure in the emtp," *IEEE Transactions on Power Systems*, vol. 5, no. 2, pp. 394–402, 1990.

[17] L. Ferreira, B. Bonatto, J. Cogo, N. de Jesus, H. Dommel, and J. Martí, "Comparative solutions of numerical oscillations in the trapezoidal method used by emtp-based programs," *IPST, Cavtat, Croatia*, pp. 147–153, 2015.

[18] H. Tavante, B. Bonatto, and M. Coutinho, "Open source implementations of electromagnetic transient algorithms," in *2018 13th IEEE International Conference on Industry Applications (INDUSCON)*, pp. 825–828, IEEE, 2018.

[19] H. C. A. Tavante, B. D. Bonatto, *et al.*, "Thta octave." `https://github.com/hannelita/PyTHTA`, 2018.

[20] H. C. A. Tavante, "Thta haskell." `https://github.com/hannelita/thtahs`, 2019.

[21] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its," *Communications*, 1978.

[22] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989.

[23] R. Bird and P. Wadler, *An Introduction to Functional Programming.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988.

[24] A. Sabry, "What is a purely functional language?," *Journal of Functional Programming*, vol. 8, no. 1, pp. 1–22, 1998.

[25] S. Thompson, *Type theory and functional programming.* Addison Wesley, 1991.

[26] D. A. Turner, "Elementary strong functional programming," in *International Symposium on Functional Programming Languages in Education*, pp. 1–13, Springer, 1995.

[27] J. Harrison, "Introduction to functional programming," *Lecture Notes, Cam*, 1997.

[28] G. Michaelson, *An introduction to functional programming through lambda calculus.* Courier Corporation, 2011.

[29] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1, ACM, 2007.

[30] G. Hutton, *Programming in Haskell.* Cambridge University Press, 2016.

[31] E. Meijer, "Introduction to functional programming e. meijer." `https://ocw.tudelft.nl/courses/introduction-to-functional-programming/`. Accessed: 2019-12-01.

[32] D. Grossman, "Programming languages, part a." `https://www.coursera.org/learn/programming-languages`. Accessed: 2019-12-01.

[33] M. Lipovaca, *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[34] G. Hutton and E. Meijer, "Monadic parsing in haskell," *Journal of functional programming*, vol. 8, no. 4, pp. 437–444, 1998.

[35] A. Gill, "Domain-specific languages and code synthesis using haskell," *Communications of the ACM*, vol. 57, no. 6, pp. 42–49, 2014.

[36] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell, "Verifying haskell programs using constructive type theory," in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pp. 62–73, ACM, 2005.

[37] W. T. Hallahan, A. Xue, and R. Piskac, "G2q: Haskell constraint solving," in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pp. 44–57, ACM, 2019.

[38] T. Schrijvers, M. Piróg, N. Wu, and M. Jaskelioff, "Monad transformers and modular algebraic effects: What binds them together," *CW Reports*, 2016.

[39] "Coq." `https://coq.inria.fr/tutorial-nahas`. Accessed: 2019-12-01.

[40] "Liquid haskell." `https://ucsd-progsys.github.io/liquidhaskell-blog/`. Accessed: 2019-12-01.

[41] J. Christiansen, S. Dylus, and N. Bunkenburg, "Verifying effectful haskell programs in coq," in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, (New York, NY, USA), pp. 125–138, ACM, 2019.

[42] N. Vazou, *Liquid Haskell: Haskell as a theorem prover*. PhD thesis, UC San Diego, 2016.

[43] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Synthesizing functional reactive programs," in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, (New York, NY, USA), pp. 162–175, ACM, 2019.

[44] P. Hudak, *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, 2000.

[45] "cassava library." `http://hackage.haskell.org/package/cassava`. Accessed: 2019-12-01.

[46] "Data.matrix library." `https://hackage.haskell.org/package/matrix-0.2.2/docs/Data-Matrix.html`. Accessed: 2019-12-01.

[47] "Hmatrix library." `http://hackage.haskell.org/package/hmatrix`. Accessed: 2019-12-01.

[48] A. Koohang and K. Harman, "Open source: A metaphor for e-learning.," *Informing Science*, vol. 8, 2005.

[49] R. T. Watson, M.-C. Boudreau, P. T. York, M. E. Greiner, and D. Wynn Jr, "The business of open source," *Communications of the ACM*, vol. 51, no. 4, pp. 41–46, 2008.

[50] J. Lerner and J. Tirole, "The open source movement: Key research questions," *European economic review*, vol. 45, no. 4-6, pp. 819–826, 2001.

[51] "Haskell ghc compiler." `https://downloads.haskell.org/ghc/latest/docs/html/users_guide/`. Accessed: 2019-12-01.

[52] "Haskell hackage." `http://hackage.haskell.org/packages/browse`. Accessed: 2019-12-01.

[53] D. Gotterbarn, K. Miller, and S. Rogerson, "Software engineering code of ethics is approved," *Communications of the ACM*, vol. 42, no. 10, pp. 102–107, 1999.

[54] C. Okasaki, *Purely functional data structures.* Cambridge University Press, 1999.

[55] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey, *Computability and logic.* Cambridge university press, 2002.

[56] S. L. Peyton Jones, *The implementation of functional programming languages (prentice-hall international series in computer science).* Prentice-Hall, Inc., 1987.

[57] B. C. Pierce and C. Benjamin, *Types and programming languages.* MIT press, 2002.

[58] H. Barendregt, W. Dekkers, and R. Statman, *Lambda calculus with types.* Cambridge University Press, 2013.

[59] J. R. Hindley and J. P. Seldin, *Lambda-calculus and Combinators, an Introduction*, vol. 13. Cambridge University Press Cambridge, 2008.

[60] R. M. Smullyan, *A beginner's guide to mathematical logic.* Courier Corporation, 2014.

[61] A. George and D. Velleman, "Philosophies of mathematics," *AMC*, vol. 10, p. 12, 2002.

[62] H. Weyl, *Levels of infinity: selected writings on mathematics and philosophy.* Courier Corporation, 2013.