Trinity College Dublin

The University of Dublin

School of Engineering

Department of Electronic and Electrical Engineering

Adding transient simulation of frequency domain devices to the Gnucap circuit simulator

Seán Higginbotham

Supervisor: Assistant Prof. Justin King

April 2024

A Dissertation submitted in partial fulfilment
of the requirements for the degree of
M.A.I. Electronic Engineering

Abstract

Radio frequency design constitutes a dominant element in the development of key communications technologies. Having accurate, robust, and widely accessible simulation methods is critical to ensuring continued advancements in this field, and guaranteeing the associated infrastructural and societal shifts that such technologies enable.

High frequency circuits invariably contain multiple non-linear components which are naturally dealt with via time marching simulation of their time-domain analytic equations. However, including this alongside linear, generally dispersive, devices and effects, which are typically only characterised through a set of frequency-domain data describing the scattering response of an associated port-network, has traditionally been a problem for designers. Frequency-domain methods such as the harmonic balance technique and its successors have dominated radio frequency design for decades. However, such methods exhibit disadvantages in the context of modern circuits which are increasingly non-linear, and which operate with increasingly complicated modulated signals.

Various alternatives have been proposed, though as of yet no universally accepted method has emerged. Though harmonic balance will likely not be replaced, this project seeks to implement one such pure transient technique as an alternative. The proposed technique is based on using the vector-fitting algorithm to produce a model of the frequency response of the linear port-network, and then using a recursive convolution formulation to allow the time-domain response to be efficiently obtained from the port's impulse response. An equivalent circuit companion model is developed from the resulting time-domain power-wave relation. This companion model allows the linear device to be directly included in a transient simulation alongside the analytic non-linear components, by way of providing a manner of computing the voltage and current on the network's ports.

We implement the technique for one-port networks in a circuit driven by baseband signals. It is added to the free, open-source Gnucap circuit simulator as a 'device plugin'. This report details how the implementation was done and provides results illustrating that it works as intended; the plugin can be installed by a user, who simply provides it with a file of frequency-domain data representing the port-network, and the plugin works naturally with the Gnucap transient solver to allow obtaining a transient solution of the overall circuit. A pure transient technique such as this does not require limiting assumptions or approximations on any components in the circuit and they are therefore preferable in certain contexts to frequency-domain methods like harmonic balance.

The project offers a significant contribution towards increasing the accessibility of radio-frequency electronics design and teaching.

Lay Abstract

Advanced technologies like 5G and beyond are enabling significant societal and infrastructural shifts in modern societies, and are emerging as being crucial for expediting social, environmental, and economic change.

Behind such technologies, however, is a great degree of work done by engineers who specialise in designing and validating the electronic circuits which underly and make these technologies possible. The primary manner in which this process is done is through computer-aided simulation of the circuit behaviour. These simulations, though generally quite robust, still offer room for improvement in the context of keeping pace with the ever more demanding advancements required in communications technology. One such area that warrants attention is in more efficiently and accurately being able to understand how circuit behaviour evolves over time. Traditional approaches in this field are arguably not sufficient when it comes to such metrics since they in general only provide us with what the behaviour looks like ‘after a long time’, or are not as applicable to more modern use cases. In any case, compromises on accuracy must be made to use them.

This project entailed developing a prototype for a potential alternative to complement these traditional methods. As a platform, it added the method to a freely available circuit simulation software, known as Gnucap. The technique uses well established algorithms in the field to offer a way in which engineers can obtain an accurate idea of the behaviour evolution of the communications circuit over time in a relatively straightforward manner that naturally fits into the rest of the ecosystem. It does this without needing to compromise on accuracy.

The project work also constitutes a small contribution towards increasing the accessibility of such design methods, in an industry that is dominated by commercial providers.

Acknowledgements

I would like to thank my M.A.I supervisor Dr. Justin King, whose previous work was the basis for this project. He provided invaluable insights and guidance which made the project both possible and an enjoyable experience, instilling curiosity at each discussion.

Relevant academic references are included in the bibliography section.

Acknowledgements of the dependancies used in the project code follow.

Gnucap

Gnucap is the creation of Albert Davis and is developed by him and others. It is provided under the GNU GPLv3, which is also the license that this project code is provided under on the associated GitHub repository.

See `https://www.gnu.org/licenses/gpl-3.0.html`. For the GNU GPLv3 license. Additionally, see the Gnucap repository here `https://savannah.gnu.org/projects/gnucap/`.

LAPACK

LAPACK is a co-creation of The University of Tennessee and The University of Tennessee Research Foundation, The University of California Berkeley, and The University of Colorado Denver. See the user guide here `https://netlib.org/lapack/`.

The LAPACKE C bindings are the creation of Intel Corp.

The relevant licensing files are found within the source code and on the this website.

Should the reader of this report have any questions or suggestions, please feel free to reach out at `higginbs@tcd.ie`, or via other channels such as the project GitHub located at `https://github.com/SHigginbotham/transient-sparam-gnucap`. The project supervisor may also be of interest, available at `justin.king@tcd.ie`.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Project Objectives . . . 2
  - 1.2.1 Updates on Interim Plan . . . 3

2 Literature Review **5**
- 2.1 Scattering Parameters . . . 5
- 2.2 Simulation Techniques . . . 8
  - 2.2.1 Transient Analysis . . . 9
  - 2.2.2 Frequency Domain Analysis and Harmonic Balance . . . 9
    - 2.2.2.1 Harmonic Balance Developments . . . 10
  - 2.2.3 Alternatives and Proposed Scheme . . . 10
- 2.3 Vector Fitting . . . 12
  - 2.3.1 Algorithm Details . . . 12
  - 2.3.2 Implementation Details . . . 14
  - 2.3.3 A Note on Implementation . . . 17
- 2.4 Recursive Convolution . . . 17
- 2.5 Companion Model . . . 19

3 Technical Environment and Methodology **22**
- 3.1 Programming Environment . . . 22
- 3.2 Gnucap . . . 23
  - 3.2.1 Installation . . . 23
  - 3.2.2 Plugins . . . 25
    - 3.2.2.1 Concept . . . 25
    - 3.2.2.2 Device Plugins for Transient Simulations . . . 26
- 3.3 LAPACK . . . 34
  - 3.3.1 Installation . . . 34
  - 3.3.2 Functions Used . . . 36

4 Implementation **41**
- 4.1 Compiling . . . 41
- 4.2 Vector Fitting Implementation . . . 42
  - 4.2.1 Preliminaries . . . 42
  - 4.2.2 Iterative Algorithm . . . 43
- 4.3 Recursive Convolution Model . . . 45
  - 4.3.1 Polynomial Functions . . . 46
  - 4.3.2 Device Parameters . . . 46
  - 4.3.3 Transient Function Overrides . . . 47
- 4.4 Access to Work . . . 49

5 Results and Conclusions **50**
5.1 Results Methodology 50
5.2 Vector Fitting Results 51
5.2.1 RLC Circuit 51
5.2.2 Transmission Line 51
5.2.3 Butterworth Filter 53
5.2.4 Simulating Beyond Trained Frequencies 56
5.3 Recursive Convolution Results 57
5.4 Discussion and Future Work 59

Appendix A1 Example Device Plugin **65**

Appendix A2 Example `tasks.json` for Plugin Compilation with VSCode **68**

Appendix A3 Modified `make.inc` File for Building LAPACK **69**

Appendix A4 `vf.h` File **71**

Appendix A5 `rc_model.cc` File **90**

Appendix A6 `tasks.json` File for Compiling the RC Model with VSCode **102**

Appendix A7 `rlc_z11.txt` Simulation Data **104**

Appendix A8 `tl_s11.txt` Simulation Data **113**

Appendix A9 `butterworth_s11.txt` Simulation Data **117**

Appendix A10 `test.ckt` Gnucap Batch Script **121**

List of Figures

1.1 Conceptual illustration of splitting RF circuits into non-linear and linear parts . . 2

2.1 General 2-port network 5
2.2 Illustration of a terminated TL 6
2.3 2-port network characterised by S-parameter matrix 7
2.4 S-parameter measurement on a 2-port network 7
2.5 Summary of the traditional approach to simulating RF/MW circuits via HB, and the proposed approach implemented by this project 11
2.6 Circuit diagram of the recursive convolution companion model 20

3.1 Simplified Gnucap transient simulation flow 28
3.2 Simple resistive network to illustrate Gnucap plugin 33
3.3 Changes to the C/C++ compilation of LAPACK 35
3.4 Changes to the Fortran compilation of LAPACK 35
3.5 Changing the file extension of the LAPACK libraries 36

5.1 Example of a 1-port network with known analytic expression 51
5.2 Magnitude and phase response of the C++ VF model of a known RLC circuit's input impedance 52
5.3 1-port lossless TL network for VF evaluation 53
5.4 Comparison of the poles and residues of the C++ and MATLAB VF models of series lossless TL circuit 53
5.6 1-port Butterworth LPF network for VF evaluation 53
5.5 Magnitude and phase response of the C++ and MATLAB VF models of a lossless TL circuit's input S-parameter 54
5.7 Comparison of the poles and residues of the C++ and MATLAB VF models of a Butterworth LPF 54
5.8 Magnitude and phase response of the C++ and MATLAB VF models of a Butterworth LPF circuit's input S-parameter 55
5.9 Magnitude and phase response of the C++ and MATLAB VF models of a lossless TL circuit's input S-parameter, where we run beyond trained frequencies 56
5.10 The Gnucap circuit used to test the completed RC model plugin 57
5.11 TS port voltage and current results of the lossless TL circuit, showing results for the MATLAB and Gnucap models, and the 'ground truth' ADS transient data . 58
5.12 Comparing the TS port voltage from Gnucap against the SS value of the port voltage 59

List of Tables

3.1 Table of Gnucap device base class types and their inheritance structure 27
3.2 Traversal sequence of a matrix stored in row-major layout, using both row-major and column-major traversal schemes 38

5.1 C++ VF fit parameters compared against analytic expectations for RLC circuit . 51
5.2 Comparison of the execution time of the VF algorithm in the MATLAB and C++ implementations. 59

Listings

3.1 Expected terminal output from starting Gnucap by invoking the binary executable. 25
3.2 Example code used at the beginning of a device plugin source file. The plugin class must inherit from one of the base classes. 27
3.3 Some of the required virtual function overrides for a device plugin. Notice that they relate to simulation methods because Gnucap needs to know how to deal with them during simulation runs. 28
3.4 Example of a `tr_begin()` override. This function sets up initial and constant values for the TS. 30
3.5 Example of a `do_tr()` override. This function serves to stamp the parameters of the device constitutive relations into the MNA matrix. 30
3.6 Example of adding a device plugin to the Gnucap dispatcher. This must be done at the very end of any plugin file source file. 32
3.7 Example command for compiling a plugin source into a dynamic library file, using `g++`. 33
3.8 Example sequence of terminal commands used to load a plugin, build a netlist with it, and perform a TS. The TS results are printed to the terminal. 33
3.9 Algorithm for row-major traversal of a 2D array. 37
3.10 Algorithm for column-major traversal of a 2D array. 37
3.11 Declaration of the function `LAPACKE_dgelss`. From `lapacke.h`. 38
3.12 Declaration of the function `LAPACKE_dgeev`. From `lapacke.h`. 39
3.13 Declaration of the function `cblas_dgemm`. From `cblas.h`. 40
4.1 Command for compiling `rc_model.cc` into a dynamic library file, `rc.so`, using `g++`. 41
4.2 `vf.h` vector fitting function header. This function is called from within the RC model plugin, with most of its arguments being user provided. 42
4.3 At the end of the VF function, the VF model parameters are used to update the RC model instance's corresponding attributes. These are used in computing the parameters of the RC model. 42

Nomenclature

$V_n^+(s)$	Incident voltage wave at port n
$V_n^-(s)$	Reflected voltage wave at port n
$I_n^+(s)$	Incident current wave at port n
$I_n^-(s)$	Reflected current wave at port n
$A_n(s)$	Incident power wave at port n
$B_n(s)$	Reflected power wave at port n
Z_0	Characteristic impedance of a transmission lines
Z_{ref}	Reference impedance for S-parameter measurements
$\tilde{S}(s)$	VF model for an S-parameter $S(s)$ represented by $\bar{S}(s)$
$\tilde{K}$	VF model remainder term
K	VF auxiliary remainder
G_c	RC model conductance term
$\bar{S}(s)$	Vector of data samples representing $S(s)$, evaluated at s
$\mathbf{s}$	Vector of frequency sample points for $\bar{S}(s)$
$\tilde{\mathbf{r}}$	Vector of VF model residues
$\tilde{\mathbf{p}}$	Vector of VF model poles
$\mathbf{r}$	Vector of VF auxiliary residues
$\mathbf{p}$	Vector of VF auxiliary poles
$\hat{\mathbf{r}}$	Vector of VF scaling function residues
$h[n]$	RC model history term
$d_k[n]$	RC model convolution term
$i_c[n]$	RC model current source term
Γ	Reflection coefficient
$\sigma(s)$	VF scaling function
ADS	Advanced Design System
BLAS	Basic Linear Algebra Subprograms
DAE	Differential-algebraic equation
EDA	Electronic design automation
EM	Electromagnetic
FD	Frequency Domain
GCC	GNU Compiler Collection
Gnucap	GNU Circuit Analysis Package
HB	Harmonic balance
KCL	Kirchoff's Current Law
KVL	Kirchoff's Voltage Law
LAPACK	Linear Algebra PACKage
LPF	Low-pass filter
LSA	Linear systems analysis
LTI	Linear time-invariant
MAE	Mean absolute error
MNA	Modified nodal-analysis

MW	Microwave
NA	Nodal-analysis
OFDM	Orthogonal frequency-division multiplexing
PFE	Partial-fraction expansion
RC	Recursive convolution
RF	Radio-frequency
S-parameters	Scattering parameters
SS	Steady-state
TD	Time-domain
TL	Transmission line
TS	Transient simulation
VF	Vector fitting
VNA	Vector network analyser
VSCode	Visual Studio Code

Chapter 1

Introduction

1.1 Background

Communications technology is a large driver of modern electronics design, with it being a key enabler of many of the societal shifts of the past few decades. The continued coupling of key infrastructure to wireless and mobile networks provides ample evidence of this, with 5G being the typical example of the facilitative effect such technologies can have [1]. In this context, demand for greater bandwidths and performance are inevitable in order to accommodate increasing traffic and this implies an unabated trend towards higher signal frequencies. Furthermore, the underlying hardware continues to operate at greater clock speeds with more intimate device geometries in order to support this.

This presents a problem for designers at all levels of these systems. When dealing with signals of 30 MHz to 4 GHz we are in what is considered the **radio-frequency** (**RF**) band, and above this in the **microwave** (**MW**) band. At these frequencies, traditional circuit analysis based on **Kirchoff's Current Law** (**KCL**) and **Kirchoff's Voltage Law** (**KVL**) begins to fail and we must consider the voltage and current signals as propagating waves [2]. That is, it is no longer accurate to treat signal levels as being only time-varying since the space-varying nature of the **electromagnetic** (**EM**) waves becomes non-neglible at typical circuit scales. Consequently device parameters must be considered as being distributed over unit length, rather than being lumped to some infinitesimal point in space.

Analysis at RF/MW frequencies can therefore be quite difficult and even intractable since we must resort to dealing directly with Maxwell's equations. However, if we are only interested in input-output relationships of a given network then we may abstract it to a single or multi-port network described by one or more transfer functions [3] in the **frequency domain** (**FD**). **Scattering parameters** (**S-parameters**) are the typical such scheme employed in RF/MW systems. The use of **linear systems analysis** (**LSA**) in this way necessarily implies that such an approach restricts our focus to **linear time-invariant** (**LTI**) devices. Most passive RF components are LTI, such as transmission lines, filters, and matching networks.

Nevertheless, highly non-linear components, such as power amplifies, invariably occur in wireless systems, for which there is a need to use a **time-domain** (**TD**) approach. On the basis of this discussion, we may generalise RF/MW circuits as consisting of non-linear components and linear components. The non-linear components are most appropriately analysed in a **transient simulation** (**TS**) because they do not yield to LSA. On the other hand, the linear components can be abstracted as a 'black box' and characterised by S-parameters in the FD.

In the context of circuit simulation and modern **electronic design automation** (**EDA**) flows, the **harmonic balance** (**HB**) technique has traditionally been used to enable a consistent solution to be obtained for the overall circuit. In HB, non-linear components are considered approximately linear in the band of interest. Assuming a periodic or quasi-periodic input, the response of non-linear elements is therefore also assumed periodic or quasi-periodic, and hence solved in the frequency domain using phasor analysis [4]. Actual implementation varies, but HB allows non-linear components to be solved alongside LTI elements as long as these assumptions on the non-linear components are accurate.

1.2 Project Objectives

However, the assumptions of HB may not be entirely justifiable in modern systems, a point which chapter 2 elaborates on. We may alternatively seek to convert the FD S-parameter description of the linear components to the TD such that it can be included in a TS. This would be doubly advantageous since it would allow us to avoid limiting approximations on the non-linear components, while also providing us with an insight into the transient evolution of the circuit, which FD analyses like HB, cannot provide.

Conversion into the TD is not as simple as it sounds, however. The non-linear components generally have robust analytic expressions, but the S-parameters describing LTI devices are usually collected as a data vector or matrix across frequencies ω using either (1) EM simulation via a computer package or simulator, or (2) from physical measurement using a **vector network analyser** (**VNA**). In the FD this does not present an issue since we are simulating across ω regardless, but there appears to be no generally accepted or universal technique for obtaining a TD expression from a data vector. Chapter 2 discusses this further. Additionally, many of the LTI networks or effects that we represent as S-parameters might also lack an analytical TD description entirely, such as when modelling noise. They may be otherwise difficult to deal with due to containing many distributed elements, as is the case with transmission lines or large matching networks. These issues tend to make FD data vectors the only acceptable way to characterise RF/MW LTI devices.

This purpose of this project is to implement in C++ one technique for addressing this problem, adapted from the approach used in [5]. First of all, a polynomial model is fit to the S-parameter data using the **vector fitting** (**VF**) numerical technique, and then we convert it to a TD constitutive relation using **recursive convolution** (**RC**). We end up obtaining the voltage $v_k[n]$ and current $i_k[n]$ at the port k of the S-parameter block. The block can hence be included in the nodal-analysis algorithm of the transient solver alongside the analytic non-linear components.

The preceding concepts are illustrated in Fig. 1.1.

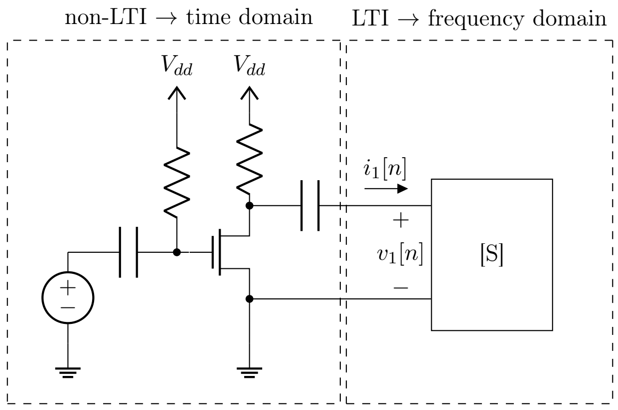

Figure 1.1: Conceptual illustration of project motivation. Shown is a simple case for a 1-port LTI network described by its S-parameter matrix, [**S**]. Note that the non-linear part is exemplified by an arbitrary transistor amplifier stage, to which no importance should be attached; it is simply to emphasise that it is not a black-box like the linear part typically is, and is hence characterised by an analytic TD equation. Such equations are in general differential-algebraic equations.

As a development platform for implementing our proposed technique, this project uses the

GNU Circuit Analysis Package (**Gnucap**), which is a free and open-source circuit simulator distributed under the GNU General Public License. This simulator is implemented in the C++ language and is under active development.

Gnucap is a typical SPICE-class simulator, with the expected analysis options such as transient, DC, and AC (phasor). However, the program is incomplete and lacks many features. Notably it lacks an ability to robustly handle RF/MW circuits in a way that RF/MW engineers are typically interested in. It does, however, possess a relatively mature plugin system by which users can contribute custom features.

Because of its open nature and in-development status, Gnucap is an ideal testbed for implementation of our proposed scheme. The architecture of Gnucap and the project programming environment, to the extent that they are useful for this report, are explained in Chapter 3.

The original **project goal** provided in the project's interim report is as follows:

"To contribute to the development of Gnucap by adding a technique for simulating RF circuits. This will be done via a plugin and consist of adding the capability to include LTI devices described in the frequency domain by S-parameters to an otherwise transient analysis. This will be done via the vector fitting and recursive convolution methods."

We note that commercial simulators and EDA tools already provide quite robust implementations of HB and its variants, and are ubiquitously used by RF/MW engineers. These solutions, however, are in general prohibitively expensive for many who might wish to benefit from their use. The use of Gnucap in this project is, therefore, also motivated by a desire to contribute to the common good by lowering the bar of entry to these kinds of tools. In the future, we might also seek to leverage the open-source nature of Gnucap, and similar softwares, to effectively compare various different methods for RF/MW simulation. Community-driven exercises such as this are not generally possible in an industry which is dominated by closed-source ecosystems, where designers must settle for what is in most situations the only option available.

In any case, this report details the manner in which the project goal as stated above is accomplished. We do not claim that the implementation is robust or completely reliable, but remains as a research exercise and proof of concept, with quite significant scope for further development and improvement. The aim is not to replace proprietary simulators, but to merely offer an alternative and to broaden the scope for community-driven discussion and development.

This report describes how we implement the proposed technique for a 1-port network.

1.2.1 Updates on Interim Plan

Here we reiterate the original objectives of the project and briefly discusses how they have changed, and if they've been achieved or not.

The following blocks contain the original objectives in *italics*, and then a brief note on any updates as required. To the right is a coloured box which is coloured gray if accomplished, and left blank (i.e., same colour as background) otherwise. Note that some of these objectives may only make complete sense once the rest of the report is read.

Objective	Accomplished
(1) Understand the architecture and coding environment of Gnucap, specifically its transient analysis solver, in a way that will allow appropriate addition of the plugin. This will culminate in the creation of various 'test' plugins to evaluate understanding.	▨

Note	The necessary understanding to implement the plugin was attained and will be elucidated in chapter 3 of this report. Evaluation of understanding was done by a continuous debugging approach as opposed to implementing distinct 'test' plugins, however.

(2) Create an isolated, external C++ implementation of the VF algorithm and test it with some S-parameter data of an analytic circuit.	▨
Note	The majority of time spent on the project was in developing this isolated VF algorithm, since it constituted installation and learning of the LAPACK library, in addition to significant debugging. The S-parameters used to evaluate it were measured from a simple transmission line system inside the ADS software.

(3) Create the plugin for Gnucap that will chiefly employ RC and companion modelling to derive TD port values using the outputs of the isolated VF implementation of (2). Evaluate against analytic circuit.	▨
Note	In fact this objective was attained in conjunction with objective (4) below since the VF algorithm was implemented into the plugin prior to the final form of the RC model. The plugin was developed iteratively in a series of proof-of-concept style steps because the unfamiliar nature of Gnucap made careful validation of each addition neccessary before proceeding.

(4) Combine the program from (2) into the Gnucap plugin in order to complete it.	▨
Note	The proposed combination of VF and the RC model proved successful in practice. Combining them was in fact relatively straightforward, indicating the surprising robustness of Gnucap's plugin system.

(5) Comprehensively validate the plugin with both analytic S-parameter data and some 'real' (as in, from an actual circuit of interest) measured or simulated data to emulate practical use of the plugin.	
Note	Though the plugin was verified with S-parameter data, it could certainly not be considered 'comprehensive', and the data was measured via simulation of small, rudimentary circuits inside ADS, not 'real' analytic circuits of significance. Regrettably, time constraints meant that deeper probing of the implementation was not possible, though chapter 5 does provide some preliminary results and suggests lines of future enquiry.

(6) Publish or disseminate plugin to the Gnucap community.	
Note	The plugin code is provided on an associated GitHub repository, but the Gnucap community has not yet been informed as of time of writing. See section 4.4.

(6) Stretch objectives: Compare to commercial solvers; Investigate alternatives to the VF and RC approach.	
Note	This objective was not achieved. A significant amount of validation and refinement is still warranted on the plugin. Unless the project is continued in the future, this objective will remain unfulfilled.

Chapter 2

Literature Review

In this chapter, we explain and discuss relevant theoretical details that are needed to understand the technical work and results in chapters 3, 4, and 5. It is also more convenient to discuss some details of implementation in this section at the points where the related theory is presented. Therefore, this chapter not only serves as a literature review but also elucidate on some of the rationale and nuances of the actual project work itself.

2.1 Scattering Parameters

As mentioned in the introduction, it is common to abstract networks as single and multiport networks, where we only characterise their input-output relationship. We invariably use the passive sign convention, where the port into which the current flows is at the positive voltage. This section will use Ludwig & Bretchko [6] as a reference, specifically the chapter on port analysis [3]. Further details can be found therein.

Typically in port analysis, the transfer functions are ratios of port voltage and current, which are measured by shorting other ports (such that the voltage drop across them is zero) or making them open circuits (such that current entering the port is zero). For example, in the 2-port network shown in Fig. 2.1, the 'impedance parameter' for port 1 is given by $Z_{11} = \frac{V_1}{I_1}\big|_{I_2=0}$, and its 'admittance parameter' would be $Y_{11} = \frac{I_1}{V_1}\big|_{V_2=0}$. We are not concerned with impedance and admittance parameters in this project, we just use them as an example.

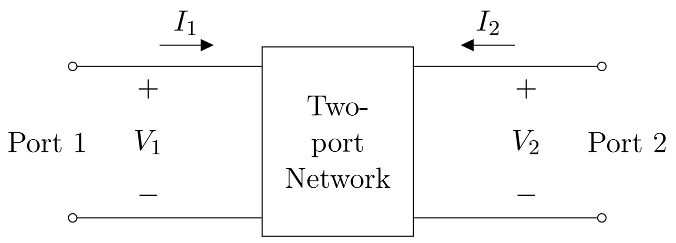

Figure 2.1: General 2-port network with port voltage and current direction specified.

This measurement procedure is only practical for low frequencies. As we reach RF/MW frequencies, the voltage and current signals must now be considered as propagating EM waves, and each wire is now treated as a **transmission line** (**TL**). Taking the approach of [7], we (1) assume lossless TLs[1], (2) assume sinusoidal signals, and (3) assume that wave propagation in space is one-dimensional (i.e., 'straight' TLs). Under these assumptions we may use phasor analysis and write these waves as consisting of both a voltage and current, being

$$V(z) = V_{\mathrm{m}}^{+}e^{-kz} + V_{\mathrm{m}}^{-}e^{+kz} \quad and \quad I(z) = I_{\mathrm{m}}^{+}e^{-kz} + I_{\mathrm{m}}^{-}e^{+kz}, \tag{2.1}$$

[1]Lossless means that Z_0 is real-valued, i.e., $R = G = 0$.

where z is distance along the TL ranging from 0 to d, the TL length, and k is called the propagation constant, being the analogous spatial term to temporal frequency ω. The time-dependant part of these expressions can be ignored because we are using phasors, and are focusing on LTI networks. We can hence write, as shown in (2.1), $V^+(z) = V_\mathrm{m}^+ e^{-kz}$, defined as the **incident voltage wave** entering the positive terminal of the device, and $V^-(z) = V_\mathrm{m}^- e^{+kz}$, defined as the **reflected voltage wave**, which returns back to the source on the negative voltage line, conceptually understood as having been 'reflected' off of the input port. The current term is similarly defined.

We will not be dealing with such expressions in any great detail in this project. However, when it comes to port-analysis we note that RF/MW circuits present a problem because we must now account for the **reflection coefficient** *at* any loads attached to the ports, which, for the same assumptions stated above, is defined as

$$\Gamma = \frac{V^-(z=0)}{V^+(z=0)} = \frac{V_\mathrm{m}^-}{V_\mathrm{m}^+} = \frac{Z_L - Z_0}{Z_L + Z_0}. \tag{2.2}$$

The term Z_L is the terminating load of the TL from the perspective of the source, and Z_0 is called the **characteristic impedance** of the TLs (port wires), which depends chiefly on the geometry and material properties of the wire. We set $z = 0$ because distance is measured from the load. The exponential terms hence reduce to 1 and Γ is simply the ratio of phasor amplitudes of the reflected and incident waves. Figure 2.2 shows a general TL to illustrate the above concepts. The TL in this figure can be seen as representing what is inside the port block of Fig. 2.1, if we were to attach a load V_s onto port 1, and terminate port 2 into Z_L.

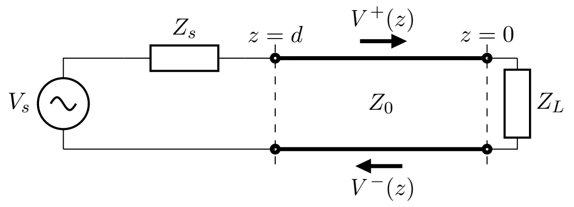

Figure 2.2: Illustration of a TL of length d, with an attached source of impedance Z_s and terminated with a load Z_L. Shown are the forward and reverse travelling voltage waves on the TL.

Returning to how voltage and current are measured, we can see that for short-circuits, $Z_L = 0$, yielding $\Gamma = -1$, which means that $V_\mathrm{m}^- = -V_\mathrm{m}^+$. Likewise, for open-circuits, $Z_L \to \infty$, and $V_\mathrm{m}^- = V_\mathrm{m}^+$. The only case where we get a zero Γ is where $Z_L = Z_0$, which is called 'impedance matching'.

Non-zero Γ are generally unwanted as they prevent our desired signal level from reaching the load, and more nefariously cause interference and unstable oscillations which can damage or destroy a circuit. Another factor which prevents the use of open and short circuits for RF/MW measurement is that such concepts don't make much sense at high frequencies because inductive and capacative effects on the wires will be non-negligible (i.e., because they're considered as TLs).

S-parameters allow characterising port networks in high-frequency circuits by instead measuring so-called 'power waves' instead of voltage and current directly.

Consider Fig. 2.3, which shows a 2-port network characterised by an S-parameter matrix, $[\mathbf{S}]$. V_1, V_2, I_1, and I_2 are denoted the 'port' voltages and currents, and A_1, A_2, B_1, and B_2 are the **incident power wave** and **reflected power wave** on each port, respectively. When dealing with S-parameters, all these parameters are in the FD (functions of $j\omega$) and hence phasors in the case of sinusoidal (single-harmonic) inputs, but can more generally be written as functions of complex frequency s. For a Q-port network, there are $Q \times Q$ defined S-parameters, which are

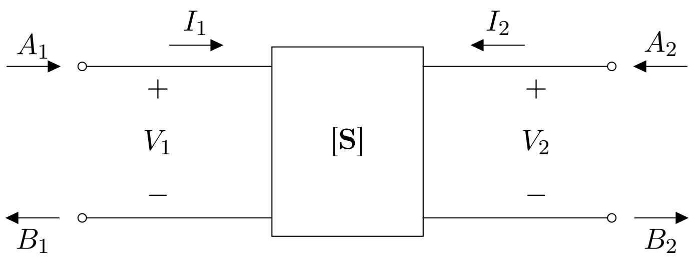

Figure 2.3: 2-port network characterised by S-parameter matrix [S], with incident and reflected power waves indicated. In our case, such networks will always be LTI.

contained in a matrix [**S**] of the same dimensions. S-parameters are defined between any two ports n and m as a transfer function

$$S_{nm} = \left.\frac{B_n}{A_m}\right|_{A_j=0\forall j\in[1,Q],\ j\neq m}, \tag{2.3}$$

i.e., the incident power wave on all ports other than port m *must* be zero. The principle behind S-parameter measurement is the aforementioned impedance matching. The general idea is indicated in Fig. 2.4, which shows the measurement of S_{11} and S_{22} for a 2-port network. Fig. 2.3 is not strictly correct as it implies open circuits on the port terminals. In reality, we must source them or terminate them with a load. For measurement, we enforce the $A_j = 0$ requirement by terminating the appropriate port into a load Z_L which matches the Z_0 of that port's TLs. In practice, as mentioned in the introduction, S-parameters are measured using a VNA or with simulation software. In the case of 1-port S-parameter networks, which is what we focus on in this project, terminating into a matched load can be understood to happen internally.

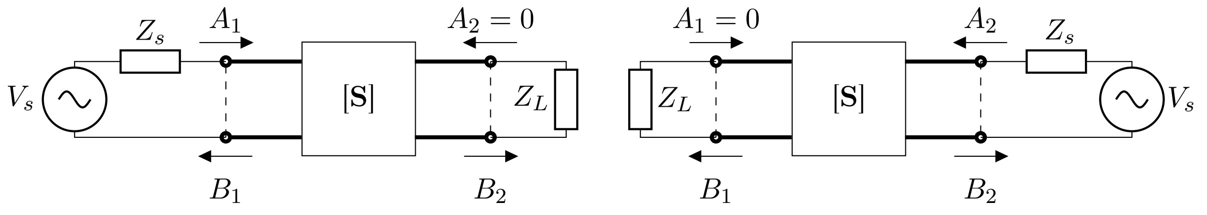

Figure 2.4: Procedure for measurement of the S-parameters S_{11} (left) and S_{22} (right) of a 2-port network; The reference port must be matched such that no incident wave enters it.

Note that the 'port' voltage and currents of Fig. 2.3 are in actuality the FD travelling waves defined in (2.1), at each port. For convenience, we will simply refer to them as port voltage and port current in this report, even though this is strictly not accurate terminology. Using these definitions, the power waves, for a lossless TL, are defined as

$$A_n = \frac{V_n + Z_{\text{ref}} I_n}{2\sqrt{Z_{\text{ref}}}} \quad \textit{and} \quad B_n = \frac{V_n - Z_{\text{ref}} I_n}{2\sqrt{Z_{\text{ref}}}}, \tag{2.4}$$

where Z_{ref} is the characteristic impedance of the measuring lines, denoted as Z_{ref} to avoid confusion with the Z_0 of the ports themselves. More formally, Z_{ref} is the impedance to which the measured S-parameters are referenced. It can be anything, but in practice we choose it to be the same as the characteristic impedance of the ports themselves so as to avoid reflections. This value is typically 50 Ω by convention.

Using (2.4) we can obtain expressions for the port voltage and currents by simple rearranging of terms. S-parameters are technically applicable to any RF/MW device, but are not strictly useful for non-linear elements because they cannot be scrutinised with LSA theory, nor do we generally wish to analyse them in the FD, but rather in the TD. We hence restrict our focus to LTI circuits when using S-parameters. Indeed, because TD treatment of LTI devices at RF/MW frequencies is not generally practical, S-parameters prove to be particularly useful. The necessary background to understand this is elaborated on in the following sections.

2.2 Simulation Techniques

We now move to a brief discussion of circuit simulation as it relates to this project. In the TD, circuit simulators typically operate by constructing a series of KCL equations which represent the circuit. The equations are composed by using the constitutive relations which model current behaviour through the device. Such relations must generally be of the form $i(v(t))$, or more broadly $\frac{\mathrm{d}}{\mathrm{d}t}q(v(t))$. The simulator automatically constructs KCL equations with respect to each node of the circuit when given a netlist referenced to a set of device models, and then uses, in effect, **nodal-analysis** (**NA**) to solve for unknown currents and voltages at nodes for each time step. The technique actually used is called **modified nodal-analysis** (**MNA**) [8], and is universally employed in all simulators, including Gnucap.

On a basic level, MNA operates by 'stamping' each element into a linear system of the form

$$\mathbf{G}\mathbf{x} = \mathbf{z}, \tag{2.5}$$

where $\mathbf{G}$ is a system matrix of element conductances, $\mathbf{z}$ is a vector of known (i.e., independent) current and voltage sources, and $\mathbf{x}$ is the vector of node voltages and currents which we wish to solve for. Note that this is the fundamental MNA formulation for linear, static elements. Extension to dynamic and non-linear elements is not done here, but [9] provides a very robust overview of these cases. [10] also provides a good description of the details of the linear, static case of MNA.

RF/MW circuit analysis can in general be split into TD and FD methods. To understand this, we may generalise each node's KCL equation at some time t as

$$f(v,t) = i(v(t)) + \frac{\mathrm{d}}{\mathrm{d}t}q(v(t)) + \int_{-\infty}^{t} y(t-\tau)v(\tau)\,d\tau + u(t) = 0, \tag{2.6}$$

which is a so-called 'distributed test' formulation of the circuit [11][2]. With respect to a given node, the term $u(t)$ is the sum of independent current sources in the connected branches. The term $i(v(t))$ is the sum of currents from conductive elements, and $q(v(t))$ is the sum of charges from capacitive elements, which again is differentiated to obtain a current term. Because of the formulation of this equation, these first two terms represent the non-linear components of the circuit, specifically non-linear conductances and non-linear capacitances.

The reason for this is that the integral term is a convolution of the node voltage $v(t)$ with an impulse response $y(t)$ of some system. Since convolution only applies for LTI devices, then necessarily we may consider that this impulse response is the combined impulse response of the LTI devices connected to the node, and hence the convolution gives their response.

We see from this equation the rationale behind the splitting of RF/MW circuits into linear and non-linear networks; we could obtain such a KCL for each node and combine them to represent the circuit as a whole. This concept of having distinct linear and non-linear networks will now be discussed further with respect to the appropriate simulation technique for each network.

[2] As noted previously, in RF/MW circuits, LTI devices are generally distributed, which is where this name comes from. This equation does not apply to time-variant circuits, circuits with inductances, or circuits with controlled sources. Since the equation is simply for illustration, this was deemed acceptable.

2.2.1 Transient Analysis

In the TD a device's constitutive relation is generally a **differential-algebraic equation** (**DAE**), which yields well to the time-marching solutions provided by transient analysis in SPICE-class simulators. Transient analysis is robust in the sense that differential terms can be straightforwardly discretised using some finite-difference method such as Euler or trapezoidal approximation. Then, using an optimisation technique such as Newton-Raphson, any non-linear terms can be iteratively linearised around some operating point until convergence [12]. Even complicated non-linear circuits are therefore suitable to transient analysis as long as we have an analytical expression for them in the TD.

However, (2.6) can provide insight into two issues with this when it comes to RF/MW circuits. First of all, the LTI devices are almost ubiquitously components such as matching networks and filters for which we usually want to perform FD analysis since we are most interested in frequency effects, such as how they filter a signal, impedance matching, and so on. The second issue is perhaps more severe; it is that the transient solution for many such devices is not so easily obtained. As alluded to in section 2.1, LTI devices and any effects which can be modelled as such, in RF/MW circuits are typically only known through S-parameter data vectors in the FD, with no known analytical TD equation to describe them. Indeed, though in theory we can transform the S-parameter data to the TD, for LTI devices it is simply more convenient to solve in the FD since this expression does not contain integro-differential terms, and is therefore a relatively trivial algebraic problem.

2.2.2 Frequency Domain Analysis and Harmonic Balance

In RF/MW circuits, then, there is an issue of desiring a TS for the non-linear components, and a FD analysis for the LTI components [13]. The coupling of these two parts of the circuit makes this more difficult since we can't merely solve them separately since their solutions won't match.

Traditionally, the harmonic balance technique has been used for this purpose. Consider again (2.6). We may convert it to the FD by applying the Fourier Transform, yielding the equation [4]

$$F(V(j\omega)) = I(V(j\omega)) + j\omega Q(V(j\omega)) + Y(j\omega)V(j\omega) + U(j\omega) = 0. \tag{2.7}$$

In simulations, this expression can be evaluated across frequencies ω in what is called 'AC analysis'. It can be seen that given some data vector containing the frequency response $Y(jw)$ that describes a linear network, we can straightforwardly obtain the solution of the LTI network, $Y(j\omega)V(j\omega)$.

However, since non-linear equations do not yield well to LSA, dealing with them in the FD is not tractable. The chief reason for this is that the responses to non-linear devices generally produce infinite harmonics of any input frequency, and so there is no linear mapping from input to output.

We may, however, make some assumptions to alleviate this problem. Firstly, we assume that, at the operating point we are currently considering, that the device behaves approximately linearly or only exhibits mild non-linearity, and hence only produces a single output tone for each input tone. This is arguably an acceptable approximation since we invariably linearise non-linear elements in the TD anyway. Secondly we assume that the excitatory signal of the non-linear components is periodic or quasiperiodic.

What these assumptions mean is that if the input can be accurately represented as a finite sum of tones, for instance $\sum_{k=0}^{n} a_k \cos(k\omega_0 t)$ for some small n, then the output will also be a finite sum but simply shifted in phase and scaled in amplitude. Note however that since the linearity assumption is usually mildly relaxed (i.e., we accept mild non-linearity), then the output will consist of a larger number of harmonics than the input. Indeed, HB becomes computationally slow as we increase the number of harmonics, and this is a limiting case for its accuracy when attempting to simulate responses to complicated input signals which need more tones to be approximated well.

In any case, we see that under these assumptions the non-linear response is a sum of simple phasors. We may hence solve the non-linear part in the FD alongside the LTI part.

The specifics of the algorithm will not be described here, and there is in fact a variety of approaches that can be taken. [14] provides quite a thorough discussion of a selection of these across various chapters. One such method mentioned is so-called ‘Harmonic Newton’ method, which converts the phasor approximation of the nodal voltages back to the TD, where it will simply be a sinusoidal sum that can be solved in the usual time-marching manner, and then converted back to the FD to be added with the LTI part. Because HB usually solves the non-linear network in the TD in this manner, it is more correctly considered a mixed-domain method, and not a FD method.

2.2.2.1 Harmonic Balance Developments

We note two distinct aspects of the HB approach. Firstly, since the solution is derived in the FD, it will *only* contain the **steady-state** (**SS**) component. This is a useful characteristic in low-frequency design. However, at RF/MW frequencies we might also be interested in the transient response which cannot be obtained using such FD methods.

The second aspect is that we can in general only expect an accurate solution from HB if the approximation of the input signal as a quasiperiodic signal is a good one. This necessarily restricts it to un-modulated signals because modulated signals can not accurately be represented as a sum of harmonics (i.e., they are not periodic). This is particularly the case for modern modulation schemes such as **orthogonal frequency-division multiplexing** (**OFDM**) which are highly non-periodic.

Hence HB in its original form has become quite insufficient for a field of engineering that is chiefly concerned with communication systems. Indeed, this issue has been known for some time, and significant work has been done to extend the usefulness of HB. Some notable developments have been extending applicability to simple modulation schemes [15], hence enabling passband simulations for signals with low bandwidths, and later to a system-level, multi-carrier systems [16]. Modern HB methods implemented into simulation tools are quite capable, though do become quite slow if we demand too large a number of harmonics. The references provided are a small sampling of the vast literature on HB.

2.2.3 Alternatives and Proposed Scheme

There appears to be disagreement in the literature on the best direction to take for RF/MW simulation. Despite the developments on HB since its original inception, it has been suggested [5, 13] that HB remains unsatisfactory for modern systems, which increasingly employ very high bandwidth, highly complicated modulated signals, alongside novel architectures. HB needs to consistently keep pace with advances in communications engineering to remain useful as a simulation technique.

Given that transient analysis places no assumptions on the nature of a signal or the system it operates in (except for some approximation error due to discretisation and linearising), it emerges as a natural alternative to HB for analysing RF/MW systems. Indeed, because TSs provide both the transient and the SS solution, they are necessary if we wish to understand system level parameters and performance characteristics, such as bit error, settling time, presence of voltage spikes, and so on. Even if HB remains the de-facto technique, there is still value in having an equally viable TD method. Of course, there will always be a need to understand FD behaviour and to obtain a SS solution quickly, so HB is unlikely to be replaced, but it can and should be complemented.

However, to develop a robust TD approach, we must return to the issues discussed above; namely, that we generally only know LTI, dispersive RF/MW components via their S-parameter data vector in the FD. So, if we wish to do a pure TS we must transform this measured data into the TD. Several different schemes have been proposed to do this. The work in [17] utilises a Fourier Series expansion of the tabulated data in the FD and then performs an inverse Fourier Transform. [18] extends this it to work for passband analysis.

One of the frequently cited issues with using such inverse transforms is that they necessarily require performing computationally intense convolution operations to obtain the output of the LTI network, since the number of time-steps in a TS is usually quite large [19]. The work by Brazil in [13] seeks to address this by instead using a 'discrete-time convolution'. An older approach named recursive convolution is presented in [20], which reduces convolution to a simple recursive sum. Indeed, [18] utilises this technique.

Another approach we could take is to derive a rational function approximation of the FD data, which allows us to obtain a very simple TD impulse response. One of the foundational pieces of work in this area is the vector fitting algorithm [19]. The original VF algorithm is applicable only to baseband signals, but work by King and Brazil [5] effectively extends it to allow representation of passband signals. They combine it with the RC technique to derive simple constitutive relations that allow computing the S-parameter block's TD port voltage and current in an efficient manner.

In order to implement TD simulation of an S-parameter black box in Gnucap, this project will use the formulation in [5] as a basis. However, we restrict our focus to the baseband case, i.e., the original VF algorithm presented in [19], and leave extension to passband simulation as future work.

For clarity, Fig. 2.5 provides a visual summary of the traditional HB technique (assuming that the non-linear part is solved in the TD as with the 'Harmonic Newton' method), and then the proposed technique that will be used in this project, and which is largely based on [5].

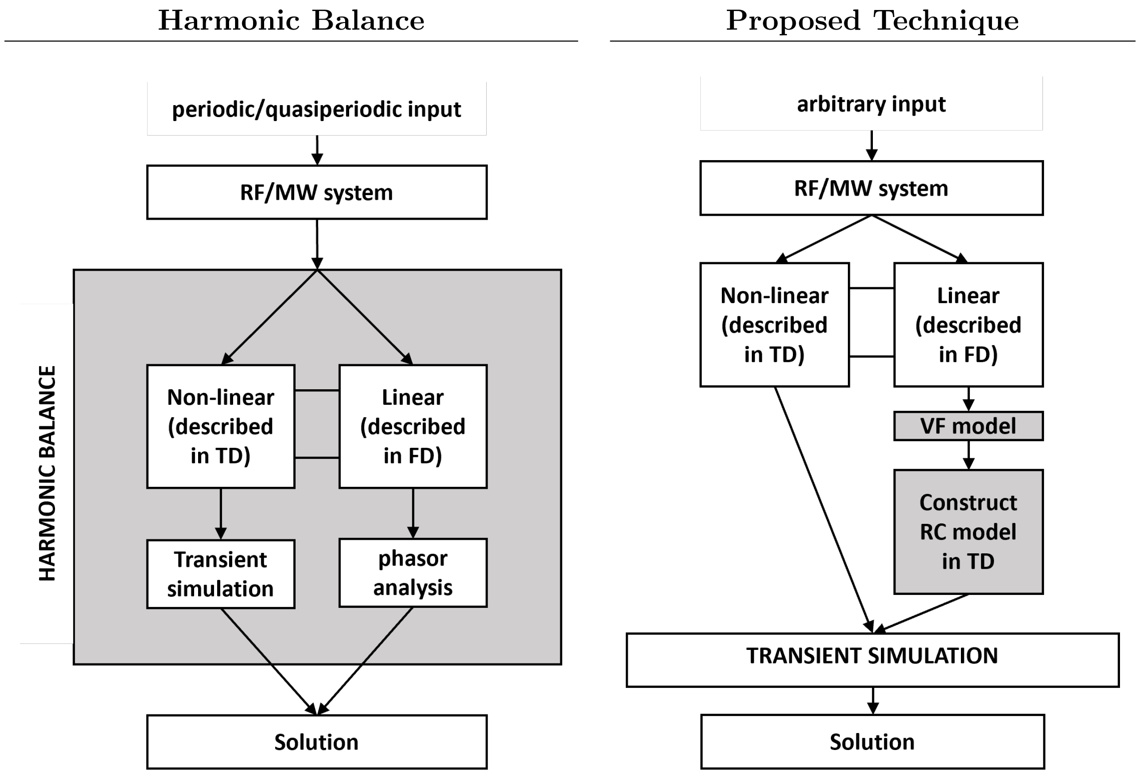

Figure 2.5: Summary of the traditional approach to simulating RF/MW circuits via HB, and the proposed pure transient approach implemented in this project.

The rest of this literature review will discuss in detail the VF and RC procedures, and how they were linked and implemented into Gnucap. It is important to emphasise that the use of rational function approximation is not universally considered appropriate in the context of the accuracy desired of EDA simulations. Hence, we implement in this project just one of many possible approaches. From the perspective of accuracy, VF will provide an inevitable degree of error[3] if the underlying S-parameters are not able to be well approximated by a rational function representation. In general, rational function fitting requires the use of polynomials of large degrees to be accurate for more complicated systems, and so might also become computationally inefficient.

2.3 Vector Fitting

2.3.1 Algorithm Details

Let us consider the context of LTI networks described by S-parameters in more detail. As mentioned before, we do not in general have an analytical expression for a given element $S_{nm}(s)$ of the scattering matrix, $[\mathbf{S}]$. Rather, a tabulated matrix of data across frequencies ω is generated using a VNA, or from simulation, which represent each of the S-parameters. Let us denote this measured data (evaluated at a point s) as $\bar{S}_{nm}(s)$ to distinguish them from the 'ground-truth' S-parameter, $S_{nm}(s)$.

In this project we only focus on implementation of 1-port networks, such as is shown in Fig. 1.1, and so in our case $[\mathbf{S}] = S_{11}(s)$. Therefore, for simplicity we will drop the subscript and use simply $S(s)$. Note that in this case, having only a single S-parameter, the collected data is a vector, not a matrix.

The first step of our proposed scheme involves converting this data vector to an analytic transfer function in the FD. This class of problem typically involves fitting the data to an approximate analytic function, which we shall denote $\tilde{S}(s) \approx \bar{S}(s)$. If we assume that our **collected data**, $\bar{S}(s)$ (note that in practice this is FD data, i.e., $\bar{S}(j\omega)$), is not significantly impacted by measurement error and hence represents the ground-truth $S(s)$ of the LTI system we are trying to model, then $\tilde{S}(s)$ will give us an approximate analytic function for our S-parameter block.

There are many possible models we can use, but based on the fact that the efficient RC technique requires a rational function approximation, Gustavsen and Semlyen [19] presented a well-known procedure for fitting measured FD data to a rational function model.

Consider a general rational transfer function,

$$\tilde{S}(s) = \frac{a_0 + a_1 s + a_2 s^2 + ... + a_N}{b_0 + b_1 s + b_2 s^2 + ... + b_N}. \tag{2.8}$$

We take the polynomial degrees to be both equal to N, such that we have the same number of poles and zeroes in the model. We can of course choose the value of N that we use, with larger N expected to yield greater accuracy in the model.

By factoring and performing a **partial-fraction expansion** (**PFE**) of (2.8), we obtain the 'pole-residue' form of the rational fraction,

$$\tilde{S}(s) = \tilde{K} + \sum_{k=1}^{N} \frac{\tilde{r}_k}{s - \tilde{p}_k} + s\tilde{c}, \tag{2.9}$$

where $\tilde{K}$ is a real, constant remainder term, $\tilde{c}$ is real, constant proportional term, $\tilde{r}_k \in \tilde{\mathbf{r}}$ are the residues of the PFE, and $\tilde{p}_k \in \tilde{\mathbf{p}}$ are its poles. Note that the boldface $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{p}}$ denote N-length

[3]At least in its original formulation. See the VF website in [21] for details on later developments to the algorithm.

vectors of the poles and residues. In order to ensure that the impulse response $\tilde{s}(t)$ is real-valued, then $\tilde{r}_k$ and $\tilde{p}_k$ should be real or occur in complex-conjugate pairs.

The equation in (2.9) is the **model** that VF solves for with respect to the **model coefficients** $\tilde{K}$, $\tilde{\mathbf{r}}$, and $\tilde{\mathbf{p}}$. Note that [19] applies to any general FD data, but in our case we are focusing on S-parameters specifically. Hence, the proportional term $\tilde{c}$ is 0 and we henceforth ignore it. The reason for this is that by definition [22], $|S(s)| < 1$ for passive systems. In other words, $\tilde{c}$ must be 0 as otherwise we would violate this condition as ω increases.

According to [19], solving (2.9) is non-linear, and so VF 'linearises' it and obtains the correct model coefficient values $\tilde{K}$, $\tilde{\mathbf{r}}$, and $\tilde{\mathbf{p}}$ via solving an auxiliary system. That procedure will now be described.

We first introduce an arbitrary **scaling function**, $\sigma(s)$, and then multiply $\sigma(s)$ by our desired model, $\tilde{S}(s)$, yielding two definitions, (2.10a) and (2.10b).

$$(\sigma\tilde{S})(s) = \sum_{k=1}^{N} \frac{r_k}{s-p_k} + K \tag{2.10a}$$

$$\sigma(s) = \sum_{k=1}^{N} \frac{\hat{r}_k}{s-p_k} + 1 \tag{2.10b}$$

The 'scaling' of our model (2.9) by $\sigma(s)$ is defined to be itself, i.e., (2.10a), and $\sigma(s)$ is defined to have the same poles $\mathbf{p}$ as $(\sigma\tilde{S})(s)$, but different residues, $\hat{r}_k \in \hat{\mathbf{r}}$, and a remainder of 1. Note the important distinction that these coefficients *are not* the model coefficients that we wish to obtain (hence the lack of a tilde accent). Let us therefore refer to K as the **auxiliary remainder**, $\mathbf{r}$ as the **auxiliary residues**, and $\hat{\mathbf{r}}$ as the **scaling function residues**.

The authors of [19] formulated these equations in such a way so that by stating that $\tilde{S}(s) \approx \bar{S}(s)$, we can write for a given frequency s,

$$(\sigma\tilde{S}(s)) \approx \sigma(s)\bar{S}(s), \tag{2.11a}$$

$$\sum_{k=1}^{N} \frac{r_k}{s-p_k} + K \approx \bar{S}(s)\sum_{k=1}^{N} \frac{\hat{r}_k}{s-p_k} + \bar{S}(s), \tag{2.11b}$$

$$\implies \sum_{k=1}^{N} \frac{r_k}{s-p_k} + K - \sum_{k=1}^{N} \frac{\bar{S}(s)\cdot\hat{r}_k}{s-p_k} \approx \bar{S}(s). \tag{2.11c}$$

Our data vector $\bar{S}(s)$ is an indexed term of an in-general P-length vector of samples at **frequency points** $s_n \in \mathbf{s}$, where $\mathbf{s}$ is of course also P-length. In other words, the n^{th} data sample is $\bar{S}(s_n)$.

Then, following the formulation in Appendix A of [19], we can use (2.11c) to write P row vectors, $\mathbf{A}_n \in \bar{\mathbf{A}}$ for each s_n, and a column vector $\mathbf{x}$, where

$$\mathbf{A}_n = \begin{bmatrix} \frac{1}{s_n-p_1} & \dots & \frac{1}{s_n-p_N} & 1 & \frac{-\bar{S}(s_n)}{s_n-p_1} & \dots & \frac{-\bar{S}(s_n)}{s_n-p_N} \end{bmatrix}, \tag{2.12a}$$

$$\mathbf{x} = \begin{bmatrix} r_1 & \dots & r_N & K & \hat{r}_1 & \dots & \hat{r}_N \end{bmatrix}^T. \tag{2.12b}$$

Then, with P being the number of data samples, and N being the order of our model (i.e., the number of poles and residues), we can construct a linear system $\bar{\mathbf{A}}\mathbf{x} = \mathbf{b}$, where $\bar{\mathbf{A}}$ is a $P \times (2N+1)$ system matrix consisting of P rows corresponding to row vectors $\mathbf{A}_1$, $\mathbf{A}_2$, …, $\mathbf{A}_n$, …, $\mathbf{A}_P$ for each frequency point s_n, as per (2.12a), and $\mathbf{b}$ is simply our tabulated data evaluated across $\mathbf{s}$, $\bar{S}(s)$. This linear system is solved via least-squares to give $\mathbf{x}$

VF is inherently an iterative procedure, where from an initial guess of the auxiliary poles, $\mathbf{p}^{(0)}$, it converges to optimal values for the coefficients in (2.9) to match our provided data vector[4].

[4]In fact convergence is not always guaranteed. We will not discuss convergence in this report, nor was it investigated in any detail, but [23] gives a good discussion on the topic.

The values of the starting poles must be conjugate complex pairs for VF to work, and [19] recommends linearly distributing the imaginary components across the frequency range $[0, s_P]$. Details on how this is done are provided in the the description of technical work in section 4.2.

The effect of beginning with N complex conjugate pairs as the starting poles, is that there are now in fact $2N$ unique pole terms. The reason for this is that the VF formulation stipulates that, in order to ensure that the corresponding residues also come in conjugate pairs, for each complex pole $p_k = \alpha_k \pm j\beta_k$, we replace the corresponding component $A_{n,k} = \frac{1}{s_n - p_k}$ in (2.12a) with two components, $A_{n,k}$ and $A^{\mathrm{j}}_{n,k}$, where

$$A_{n,k} = \frac{1}{s_n - p_k} + \frac{1}{s_n - p_k^*} \quad \textit{and} \quad A^{\mathrm{j}}_{n,k} = \frac{j}{s_n - p_k} - \frac{j}{s_n - p_k^*} \tag{2.13}$$

Note that any component in $\bar{\mathbf{A}}$ corresponding to real poles still retains the original fraction form as shown in (2.12a). This nuance may be more clear if one refers to the relevant code and associated description in section 4.2.

2.3.2 Implementation Details

The exact coefficients that we use in our model (2.9) are taken at different stages of the algorithm; it is shown in [19] the correct model poles $\tilde{\mathbf{p}}$ are equivalent to the zeroes of the scaling function $\sigma(s)$. Once these are calculated, we then substitute these computed model poles in for the auxiliary poles in (2.10a) and set it equal to $\bar{S}(s)$. Solving this by least-squares, the resulting auxiliary residues and auxiliary remainder are the model residues $\tilde{\mathbf{r}}$ and the model remainder $\tilde{K}$.

To elaborate on this, this subsection now describes how the VF algorithm can be performed, as it was implemented in this project. To make understanding easier it will be described in a series of steps.

STEP ONE : CONSTRUCT MATRICES

Using the initial poles $\mathbf{p}^{(0)}$, then both $\bar{\mathbf{A}}$ and $\mathbf{b}$ consist entirely of known terms, and we can write the system $\bar{\mathbf{A}}^{(0)}\mathbf{x}^{(0)} = \mathbf{b}$. $\bar{\mathbf{A}}^{(0)}$ is given by the matrix (2.14), and the solution vector and output vector are given by (2.15) and (2.16), respectively. In (2.14), for space reasons we use as shorthand the expressions defined in (2.13), where here the k index corresponds to $p_k^{(0)}$.

$$\begin{bmatrix}
A^{(0)}_{1,1} & \cdots & A^{(0)}_{1,N} & A^{\mathrm{j}\,(0)}_{1,1} & \cdots & A^{\mathrm{j}\,(0)}_{1,N} & 1 & -\bar{S}(s_1)\cdot A^{(0)}_{1,1} & \cdots & -\bar{S}(s_1)\cdot A^{(0)}_{1,N} & -\bar{S}(s_1)\cdot A^{\mathrm{j}\,(0)}_{1,1} & \cdots & -\bar{S}(s_1)\cdot A^{\mathrm{j}\,(0)}_{1,N} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
A^{(0)}_{n,1} & \cdots & A^{(0)}_{n,N} & A^{\mathrm{j}\,(0)}_{n,1} & \cdots & A^{\mathrm{j}\,(0)}_{n,N} & 1 & -\bar{S}(s_n)\cdot A^{(0)}_{n,1} & \cdots & -\bar{S}(s_n)\cdot A^{(0)}_{n,N} & -\bar{S}(s_n)\cdot A^{\mathrm{j}\,(0)}_{n,1} & \cdots & -\bar{S}(s_n)\cdot A^{\mathrm{j}\,(0)}_{n,N} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
A^{(0)}_{P,1} & \cdots & A^{(0)}_{P,N} & A^{\mathrm{j}\,(0)}_{P,1} & \cdots & A^{\mathrm{j}\,(0)}_{P,N} & 1 & -\bar{S}(s_P)\cdot A^{(0)}_{P,1} & \cdots & -\bar{S}(s_P)\cdot A^{(0)}_{P,N} & -\bar{S}(s_P)\cdot A^{\mathrm{j}\,(0)}_{P,1} & \cdots & -\bar{S}(s_P)\cdot A^{\mathrm{j}\,(0)}_{P,N}
\end{bmatrix} \tag{2.14}$$

$$\mathbf{x}^{(0)} = \begin{bmatrix} r^{(0)}_{1,\alpha} & \cdots & r^{(0)}_{N,\alpha} & r^{(0)}_{1,\beta} & \cdots & r^{(0)}_{N,\beta} & K^{(0)} & \hat{r}^{(0)}_{1,\alpha} & \cdots & \hat{r}^{(0)}_{N,\alpha} & \hat{r}^{(0)}_{1,\beta} & \cdots & \hat{r}^{(0)}_{N,\beta} \end{bmatrix}^T \tag{2.15}$$

$$\mathbf{b} = \begin{bmatrix} \bar{S}(s_1) & \bar{S}(s_2) & \ldots & \ldots & \bar{S}(s_{P-1}) & \bar{S}(s_P) \end{bmatrix}^T \tag{2.16}$$

Note that we arrange the system matrix $\bar{\mathbf{A}}^{(0)}$ by placing all the $A^{(0)}_{n,k}$ terms first, and then all of the $A^{\mathrm{j}\,(0)}_{n,k}$ terms. We do this on both sides of the '1' that corresponds to the remainder $K^{(0)}$. This choice is not mandatory, and it could also be formulated in the way described in appendix A of [19] where these corresponding terms are interlaced. The only effect is that the order of terms in $\mathbf{x}^{(0)}$ will be different.

Because the system matrix is constructed as in (2.14), it means that $\mathbf{x}^{(0)}$ has all the auxiliary residues $\mathbf{r}^{(0)}$, followed by the auxiliary remainder $K^{(0)}$, followed the scaling function residues $\hat{\mathbf{r}}^{(0)}$. Since $\mathbf{p}^{(0)}$ are all complex, the residues will also be complex, and in $\mathbf{x}^{(0)}$ are split into all their real parts (denoted with an α subscript) first and then all of their imaginary parts (denoted with a β subscript.

The $\mathbf{b}$ output matrix is simply the S-parameter data vector, and so is a constant column vector.

STEP TWO : SEPARATE REAL AND IMAGINARY

Though we can solve the system as is, [19] states that because the frequencies $\mathbf{s}$ which our model is fitted against are positive, that we must split the matrices $\bar{\mathbf{A}}^{(0)}$ and $\mathbf{b}$ into its real and imaginary parts, and concatenate these so as to have all the components be real-valued. This is intended to ensure Hermitian Symmetry in the model $\tilde{S}(s)$, such that the TD impulse response $\tilde{s}(t)$ is real-valued.

This operation is relatively straightforward and it means that the system we solve becomes

$$\bar{\mathbf{A}}^{(0)}_{\text{real}}\mathbf{x}^{(0)} = \mathbf{b}_{\text{real}}, \tag{2.17a}$$

$$\begin{bmatrix} \texttt{real}\left(\bar{\mathbf{A}}^{(0)}\right) \\ \texttt{imag}\left(\bar{\mathbf{A}}^{(0)}\right) \end{bmatrix} \mathbf{x}^{(0)} = \begin{bmatrix} \texttt{real}\,(\mathbf{b}) \\ \texttt{imag}\,(\mathbf{b}) \end{bmatrix}, \tag{2.17b}$$

where `real()` and `imag()` refer to functions that return the real and imaginary parts of the complex input, respectively. In this initial step where we use $\mathbf{p}^{(0)}$, which consists of N fully complex ($\beta \neq 0$) poles, then $\bar{\mathbf{A}}^{(0)}_{\text{real}}$ is $2P \times (2(2N) + 1)$, $\mathbf{b}_{\text{real}}$ is $2P \times 1$, and hence $\mathbf{x}^{(0)}$ is $(2(2N) + 1) \times 1$.

STEP THREE : SOLVE LINEAR SYSTEM

We solve the system (2.17b) via least-squares to obtain $\mathbf{x}^{(0)}$. In this project we use the **Linear Algebra PACKage** (**LAPACK**) library [24] to perform this. This is expounded on in section 3.3.2.

STEP FOUR : EXTRACT ZEROES

At this point we have values for $\mathbf{r}^{(0)}$, $K^{(0)}$, and $\hat{\mathbf{r}}^{(0)}$. Without stating the details here, it is shown in [19] that we obtain our model poles $\tilde{\mathbf{p}}$ by computing the zeroes, $\hat{z}^{(0)}_k \in \hat{\mathbf{z}}^{(0)}$ of the scaling function. These zeroes are obtained by solving for the eigenvalues of the matrix

$$\mathbf{H}^{(0)} = \mathbf{A}^{(0)}_{\text{z}} - \mathbf{b}^{(0)}_{\text{z}}\hat{\mathbf{c}}^{(0)}_{\text{z}}, \tag{2.18}$$

where, $\mathbf{A}^{(0)}_{\text{z}}$ is $2N \times 2N$ matrix consisting of four $N \times N$ diagonal sub-matrices of the real and imaginary parts of the initial poles $\mathbf{p}^{(0)}$, $\mathbf{b}^{(0)}_{\text{z}}$ is a $2N \times 1$ column vector of N twos followed by N zeroes, and $\hat{\mathbf{c}}^{(0)}_{\text{z}}$ is a $1 \times 2N$ row vector of the real and imaginary parts of $\hat{\mathbf{r}}^{(0)}$. Hence, $\mathbf{H}^{(0)}$ is a $2N \times 2N$ matrix.

These matrices are given in (2.19), (2.20), and (2.21), respectively. We again use α to refer to the real part, and β to refer to the imaginary part. Note that this formulation of the $\mathbf{H}^{(0)}$ matrix is different to that given in [19], but this is simply an implementation detail much like how we chose not to have the components of (2.14) be interlaced.

$$\mathbf{A}_\mathrm{z}^{(0)} = \left[\begin{array}{ccc|ccc} p_{1,\alpha}^{(0)} & \ddots & \ddots & p_{1,\beta}^{(0)} & \ddots & \ddots \\ \ddots & p_{2,\alpha}^{(0)} & \ddots & \ddots & p_{2,\beta}^{(0)} & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \hdashline -p_{1,\beta}^{(0)} & \ddots & \ddots & p_{1,\alpha}^{(0)} & \ddots & \ddots \\ \ddots & -p_{2,\beta}^{(0)} & \ddots & \ddots & p_{2,\alpha}^{(0)} & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \end{array}\right] \tag{2.19}$$

$$(\mathbf{b}_\mathrm{z}^{(0)})^T = \begin{bmatrix} 2 & \dots & \dots & 2 & 0 & \dots & \dots & 0 \end{bmatrix} \tag{2.20}$$

$$\hat{\mathbf{c}}_\mathrm{z}^{(0)} = \begin{bmatrix} \hat{r}_{1,\alpha}^{(0)} & \dots & \hat{r}_{N,\alpha}^{(0)} & \hat{r}_{1,\beta}^{(0)} & \dots & \hat{r}_{N,\beta}^{(0)} \end{bmatrix} \tag{2.21}$$

In the case of real poles, the matrices are constructed differently. The corresponding components of $\mathbf{A}_\mathrm{z}^{(0)}$ would simply be the real pole itself (since it has no imaginary parts), and likewise for the real scaling residue that corresponds to the real pole in $\hat{\mathbf{c}}_\mathrm{z}^{(0)}$. Lastly, the corresponding elements in $\mathbf{b}_\mathrm{z}^{(0)}$ would simply be a one.

In practice it is more straightforward to construct a separate matrix for the real and imaginary poles, $\mathbf{H}_\mathrm{r}^{(0)}$ and $\mathbf{H}_\mathrm{c}^{(0)}$. We solve for the eigenvalues of each separately, and concatenate them together to obtain our model poles $\tilde{\mathbf{p}}$.

STEP FIVE : ITERATE

Before we obtain $\tilde{\mathbf{r}}$ and $\tilde{K}$, in order to refine our model poles, we iterate steps one to four, but substitute in the model poles $\tilde{\mathbf{p}}$ as the starting poles. That is, we set $\mathbf{p}^{(1)} = \hat{\mathbf{z}}^{(0)}$. We iterate for a specified number of times M, substituting $\mathbf{p}^{(m)} = \hat{\mathbf{z}}^{(m-1)}$ each time. The computed zeroes at each iteration should converge to the something close to the actual poles of the system that is represented by $\bar{S}(s)$.

STEP SIX: OBTAIN MODEL RESIDUES AND REMAINDER

When we stop on iteration M, we then choose our final model poles $\tilde{\mathbf{p}} = \hat{\mathbf{z}}^{(M)}$. In order to obtain $\tilde{\mathbf{r}}$ and $\tilde{K}$, we construct a system by evaluating

$$\sum_{k=1}^{N} \frac{\tilde{r}_k}{s - \tilde{p}_k} + \tilde{K} = \bar{S}(s) \tag{2.22}$$

across P frequencies s_n as before, and assuming that $\tilde{\mathbf{p}}$ has N elements, to obtain the simpler system

$$\mathbf{A}_\mathrm{f}\mathbf{x}_\mathrm{f} = \mathbf{b}, \tag{2.23}$$

$$\begin{bmatrix} A_{1,1} & \dots & A_{1,N} & A_{1,1}^\mathrm{j} & \dots & A_{1,N}^\mathrm{j} & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n,1} & \dots & A_{n,N} & A_{n,1}^\mathrm{j} & \dots & A_{n,N}^\mathrm{j} & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{P,1} & \dots & A_{P,N} & A_{P,1}^\mathrm{j} & \dots & A_{P,N}^\mathrm{j} & 1 \end{bmatrix} \begin{bmatrix} \tilde{r}_{1,\alpha} \\ \vdots \\ \tilde{r}_{N,\alpha} \\ \tilde{r}_{1,\beta} \\ \vdots \\ \tilde{r}_{N,\beta} \\ \tilde{K} \end{bmatrix} = \begin{bmatrix} \bar{S}(s_1) \\ \bar{S}(s_2) \\ \vdots \\ \vdots \\ \vdots \\ \bar{S}(s_{P-1}) \\ \bar{S}(s_P) \end{bmatrix}. \tag{2.24}$$

This system does not include the scaling function coefficients, and $A_{n,k}$ and $A^{\text{j}}_{n,k}$ index the model poles $\tilde{\mathbf{p}}$. We show the case where all the model poles are fully complex, in which case $\mathbf{A}_{\text{f}}$ is $P \times (2N+1)$, and $\mathbf{x}_{\text{f}}$ is $(2N+1) \times 1$. We simply perform another final least squares to obtain $\tilde{\mathbf{r}}$ and $\tilde{K}$.

After completion of these steps, all the coefficients in the model (2.9) are known and we can use it as an analytic expression for our FD S-parameter block. We are now in a position to move to the TD.

2.3.3 A Note on Implementation

Note some important details. Firstly, even though the computed $\hat{\mathbf{z}}^{(m)}$ remain the same size as the initial poles guess used to compute them (actually, it is twice that size i.e., $2N$ since the initial poles will all occur in conjugate pairs), the actual poles that we use in the next iteration *do not*. This is because some filtering is done to only pass on as poles the zeroes with significant magnitude and which exist at positive frequencies. In practice, we see an approximate halving of the number of subsequent poles relative to the number of σ zeroes by doing this.

What that means is that over iterations, both the number of poles N will change, and additionally, real poles will appear and the number of real and complex poles will fluctuate but then converge as the model poles converge. It is hence worthwhile to differentiate between the number of 'starting' or 'initial' poles, and the number of model poles at the end of the algorithm. In this report, when we refer to the 'number of poles', it means the starting poles. Otherwise, the distinction will be noted where it is important.

In the above formulation we have shown the case for all poles being complex, as is the case at the beginning. But since real poles will be generated as the algorithm proceeds, it becomes more practical to, at various points in the algorithm, split the processing of the real and complex poles and their associated terms into separate matrices. This is evident in chapter 4. The important point is that the above matrix dimensions are illustrative. In the implementation we use variable sized arrays to ensure that the algorithm works properly.

2.4 Recursive Convolution

Now that we have an analytical model, we may convert it to a TD impulse response $s_{nm}(t)$ for use in a TS. Performing an inverse Laplace transform on (2.9) (and remembering that $\tilde{c}$ is 0), we get

$$\tilde{s}(t) = \tilde{K}\delta(t) + \sum_{k=1}^{N} \tilde{r}_k e^{\tilde{p}_k t}. \tag{2.25}$$

This familiar impulse response form, a sum of weighted exponentials, is the reason that VF is formulated as a PFE in pole-residue form. The RC technique presented by Semlyen and Dabuleanu in [20] enables very efficient convolution operations with impulse responses of this form. We will use the general approach taken by [5] and illustrate the RC technique applied to a single-port network.

Recall from section 2.1 the definition of the incident and reflected power waves on a port, A_n and B_n, in (2.4). Since we're only concerned with 1-port blocks, we can drop the subscript. We can convert these to the TD by taking their inverse Laplace transform, and since all the terms are either constant or FD signals themselves, we simply get

$$a(t) = \frac{v(t) + Z_{\text{ref}} i(t)}{2\sqrt{Z_{\text{ref}}}} \quad \textit{and} \quad b(t) = \frac{v(t) - Z_{\text{ref}} i(t)}{2\sqrt{Z_{\text{ref}}}}, \tag{2.26}$$

where $v(t)$ and $i(t)$ are the TD port voltage and current of the S-parameter block. We do not know $b(t)$, but we can assume that there is a general incident wave $a(t)$ associated with any signal entering the S-parameter block via the rest of the circuit, as in Fig. 1.1. Assuming that the input is causal and is presented at time $t = 0$, we obtain the reflected power wave at the port, at time $t \geq 0$ by convolution,

$$\begin{aligned} b(t) &= (\tilde{s} * a)(t), && (2.27a)\\ &= \int_0^t \tilde{K}\delta(\tau) \cdot a(t-\tau)d\tau + \int_0^t \sum_{k=1}^{N} \tilde{r}_k e^{\tilde{p}_k \tau} \cdot a(t-\tau)d\tau, && (2.27b)\\ &= \tilde{K}a(t) + \sum_{k=1}^{N} \tilde{r}_k \cdot \int_0^t e^{\tilde{p}_k \tau} \cdot a(t-\tau)d\tau && (2.27c)\\ &= \tilde{K}a(t) + \sum_{k=1}^{N} \tilde{r}_k \cdot (e^{\tilde{p}_k t} * a(t)). && (2.27d) \end{aligned}$$

From this point on, let us switch to a simpler notation, and drop the 'tilde' accent from the model coefficients. Focusing in on the expression $e^{p_k t} * a(t)$, we observe that convolution is in many circumstances computationally expensive, especially for very long simulations and small differences between the time steps. As noted previously, this has traditionally been a reason for the preference for FD simulation methods in RF/MW design.

Let us denote the k^{th} convolution at time t as $e^{p_k t} * a(t) = d_k(t)$. Let us also denote the time step difference of the simulator, in our case Gnucap, by Δ. It is shown in [20] that if we split this convolution into

$$d_k(t) = \int_0^\Delta e^{p_k \tau} a(t-\tau)d\tau + \int_\Delta^t e^{p_k \tau} a(t-\tau)d\tau, \tag{2.28}$$

then we can use variable substitution of τ to get

$$d_k(t) = e^{p_k \Delta} \cdot d_k(t-\Delta) + \int_0^\Delta e^{p_k \tau} \cdot a(t-\tau)d\tau. \tag{2.29}$$

That is, the convolution at the current time t can be obtained by the sum of a scaled version of the convolution at the previous time step, and a 'short' integral over a single time step. In other words, from a specified initial condition for which $d_k(t) = 0$ (or other known constant), we can straightforwardly update d_k at each time step by keeping track of its 'history' as the simulation proceeds.

At this point in the development it is useful to switch notation to discrete time, since this is the domain in which we will be operating in the computer simulation. Let us approximate $t = n\Delta \implies n = \frac{t}{\Delta}$, where n is the current time step, being separated by Δ seconds from the previous time step. We hence write

$$d_k[n] = e^{p_k \Delta} \cdot d_k[n-1] + \int_0^\Delta e^{p_k \tau} \cdot a(n\Delta - \tau)d\tau. \tag{2.30}$$

In circuit simulators we can usually specify our desired Δ, or at least access it. It is hence a known value.

In appendix 1 of [20] it is further shown that, by using a quadratic approximation for the $a(n\Delta - \tau)$ term in (2.30), and performing the integration, that we obtain (2.31), a final discrete-time expression for computing the convolution at each time step n.

$$d_k[n] = \alpha_k d_k[n-1] + \lambda_{k,2} a[n] + \mu_{k,2} a[n-1] + \nu_{k,2} a[n-2] \tag{2.31}$$

The coefficients are defined as

$$\alpha_k = e^{p_k\Delta}, \tag{2.32a}$$

$$\lambda_{k,2} = \frac{1}{-p_k} \cdot \left(\frac{1-\alpha_k}{(-p_k\Delta)^2} - \frac{3-\alpha_k}{-2p_k\Delta} + 1 \right), \tag{2.32b}$$

$$\mu_{k,2} = \frac{1}{-p_k} \left(-2 \cdot \frac{1-\alpha_k}{(-p_k\Delta)^2} + \frac{2}{-p_k\Delta} - \alpha_k \right), \tag{2.32c}$$

$$\nu_{k,2} = \frac{1}{-p_k} \cdot \left(\frac{1-\alpha_k}{(-p_k\Delta)^2} - \frac{1+\alpha_k}{-2p_k\Delta} \right). \tag{2.32d}$$

The '2' in the subscript specifies that these are the coefficients of the quadratic fit for $a(n\Delta - \tau)$. Even though we must have $a[n-1] = a[n-2] = 0$ for the initial time step (i.e., a causal signal), we cannot, strictly speaking, use a second order fit for $a(n\Delta - \tau)$ for time steps 1 and 2, since the required data points don't technically exist yet. Hence, the first order fits of these same coefficients, also provided by [20], can be used for these time steps. The first order coefficients are

$$\lambda_{k,1} = \frac{1}{-p_k} \cdot \left(1 + \frac{1-\alpha_k}{p_k\Delta} \right), \tag{2.33a}$$

$$\mu_{k,1} = \frac{1}{-p_k} \cdot \left(\frac{\alpha_k - 1}{p_k\Delta} - \alpha_k \right), \tag{2.33b}$$

$$\nu_{k,1} = 0, \tag{2.33c}$$

where a_k remains the same regardless of the fit order.

All of the coefficients (2.32) and (2.33) are functions of model poles $\mathbf{p}$, and time step difference Δ. This means that we should re-compute them at each time step if Δ changes. It also implies that, since each p_k is in general complex-valued, they will be complex numbers also. This needs to be considered with regard to development platforms. For instance, computations with complex numbers in MATLAB are considerably simpler than the same computations in C++. This is particularly true if we consider that we will be interfacing these equations with Gnucap's solver. We might therefore seek to convert to real values if possible.

2.5 Companion Model

For simplicity we will only use the second order coefficients in this section, and hence drop the '1' and '2' subscripts. The procedure for first order coefficients is essentially the same, but how they are actually used is better understood by referring to the discussion on implementation in section 4.3.

Let us substitute (2.31) into the discrete time form of (2.27d), where upon expanding and factoring we can obtain

$$b[n] = a[n] \left(K + \sum_{k=1}^{N} r_k \lambda_k \right) + h[n], \tag{2.34}$$

where $h[n]$ a term containing the aforementioned **history** of the convolution, being

$$h[n] = \sum_{k=1}^{N} r_k \cdot \left(\alpha_k d_k[n-1] + \mu_k a[n-1] + \nu_k a[n-2] \right). \tag{2.35}$$

We can effectively isolate the computation of $h[n]$ from the rest of the response. If we store the values $d_k[n-1]$, $a[n-1]$, and $a[n-2]$ at each time step, then we can easily update $h[n]$ as the MNA solution proceeds.

The simulator, however, operates in terms of voltages and currents at device terminals (i.e., nodes). We can substitute the discrete-time versions of the expressions in (2.26) into (2.34) and rearrange to get it in terms of voltage and current at the S-parameter block's port,

$$\frac{v[n] - Z_{\text{ref}} i[n]}{2\sqrt{Z_{\text{ref}}}} = \frac{v[n] + Z_{\text{ref}} i[n]}{2\sqrt{Z_{\text{ref}}}} \cdot \left(K + \sum_{k=1}^{N} r_k \lambda_k \right) + h[n], \tag{2.36}$$

$$i[n] = \left(\frac{1 - \left(K + \sum_{k=1}^{N} r_k \lambda_k \right)}{Z_{\text{ref}} \left(1 + K + \sum_{k=1}^{N} r_k \lambda_k \right)} \right) \cdot v[n] - \left(\frac{2h[n]}{\sqrt{Z_{\text{ref}}} \left(1 + K + \sum_{k=1}^{N} r_k \lambda_k \right)} \right). \tag{2.37}$$

Notice that the coefficient of $v[n]$ in (2.37) is akin to a conductance, and the term being subtracted on the right-hand side is akin to an independent current source. That is,

$$i[n] = G_{\text{c}} v[n] - i_{\text{c}}[n] \quad \textit{and} \quad v[n] = \frac{1}{G_{\text{c}}} \left(i_{\text{c}}[n] + i[n] \right), \tag{2.38}$$

where

$$G_{\text{c}} = \frac{1 - \left(K + \sum_{k=1}^{N} r_k \lambda_k \right)}{Z_{\text{ref}} \left(1 + K + \sum_{k=1}^{N} r_k \lambda_k \right)} \quad \textit{and} \quad i_{\text{c}}[n] = \frac{2h[n]}{\sqrt{Z_{\text{ref}}} \left(1 + K + \sum_{k=1}^{N} r_k \lambda_k \right)}. \tag{2.39}$$

The equivalent circuit is shown in Fig. 2.6, and is referred to as the **companion model** of the S-parameter block in the TD. More specifically, it is equivalent to the TD RC formulation for a 1-port S-parameter network. The port voltage and current equations in (2.38) give us

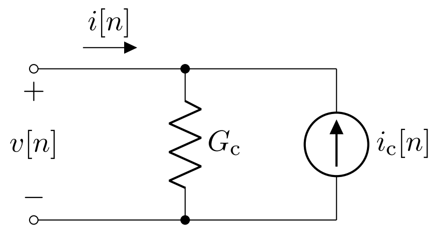

Figure 2.6: Circuit illustrating the equivalent model of the recursive convolution operation, where the independent current source and parallel conductance should be updated at each time step. The direction of the port voltage $v[n]$ and current $i[n]$ are indicated, which are obtained via TS of the RC equivalent model.

constitutive relations that can be used in a device model in order to include the S-parameter block in a TD simulation. Indeed, companion modelling is ubiquitous in circuit simulators as a way to linearise components [12], and so it is a natural manner by which to include the S-parameter network in the simulation. As mentioned previously, Gnucap provides a utility for easily developing custom device models. The second part of this project's objectives consists of developing a model that implements (2.38), and is covered in section 4.3.

The G_{c} and $i_{\text{c}}[n]$ terms use the model coefficients computed in the VF step, $\mathbf{p}$, K, and $\mathbf{r}$, and are updated at each time step along with $h[n]$, λ_k, μ_k, and ν_k.

It should be emphasised that this model only applies to single-port S-parameter networks. If extension to multi-port networks is desired, it would imply computing a VF model for each parameter in $[\mathbf{S}]$, and then developing a similar RC companion model for each. Conceivably this would consist of a cascade of circuits which are similar to the one in Fig. 2.6, one for each port. Nevertheless, it is left as future work.

Technical Work

Chapter 3

Technical Environment and Methodology

Chapters 3 and 4 will describe the actual technical work done during the project.

This project uses two other pieces of software, (1) Gnucap, and (2) LAPACK, which is a Fortran library for matrix and linear algebra computations. This chapter gives necessary technical background on these two softwares in the context of how they are used, i.e., the methodology of the project. We also describe how they can be installed and built from source, should the reader wish to use the plugin.

Chapter 4 will then detail the actual project source code itself, how it can be compiled, and where it can be accessed. These two chapters can therefore be regarded as documentation for the project. This is considered appropriate, given that it is by nature primarily a programming project, and not, strictly speaking, a research project.

In order to aid comprehension, throughout the rest of the report we will use the following conventions for styling text that is intended to represent some object within the development environment itself (i.e., files, commands, and so on). They will all be imposed onto a `gray` box, and the text itself will use

`black`	for code snippets, commands, executables,
`blue`	for directories,
`green`	for (non-directory and non executable) files, and
`orange`	for objects which don't easily fit any of these categories.

As shown, the text will also be stylised as `monospaced`.

3.1 Programming Environment

Gnucap is written in the C/C++ programming language, and from examining its documentation and project structure, is evidently maintained on a Linux operating system. Additionally, Most Linux distributions come 'ready-made' with tools and software which make development significantly easier than it is on Windows. This is particularly the case for C/C++, for which Windows does not provide an easy to use toolchain. Hence, we decide to use Linux, both out of personal preference, and to ensure efficiency, given the limited timeframe of the project.

The project work is completed primarily within an Oracle VirtualBox **virtual machine** (**VM**), hosted locally on a Windows 11 machine. See [25] for information on VirtualBox and its installation. The choice to use VirtualBox is somewhat arbitrary, and mostly based on familiarity on the part of the author. It is also desirable to have a graphical interface to work in.

The development is done on an Ubuntu 22.04.3/4 VM, installed as an `.iso` image. See [26] for a good tutorial on how to use Ubuntu on VirtualBox as a VM. Again, this environment choice is somewhat arbitrary and only noted for completeness sake.

Within the VM itself, development is primarily done using Microsoft's **Visual Studio Code** (**VSCode**) text editor, information on which can found at [27]. VSCode provides many useful features, such as syntax highlighting, 'IntelliSense', comprehensive debugging, and the possibility to extend functionality via extensions. It is chosen over a simpler editor for these reasons; given the relatively obscure nature of Gnucap and its documentation, its use proves extraordinarily helpful in debugging and speeding up development.

The compiler we use is `g++`, though it is may also be required to use the more general `gcc` at times. Both are part of the **GNU Compiler Collection** (**GCC** and can compile C/C++, Fortran, among others. Of course we only use it for C/C++ in this project. These compilers invariably come pre-installed on Linux distributions, or otherwise can very easily be installed. In this project, the code (including the Gnucap and LAPACK dependencies) is compiled using the GCC implementations of the `C++11` and `C11` standards which come pre-installed on the specific version of Ubuntu we use.

The downside of using VSCode is that it requires some initial set up in order to use it for C/C++ development. This *can* be quite involved and platform specific, and will not be discussed here. For Linux, some guidelines can be found at [28]. An important step is correctly modifying include paths to reference Gnucap and LAPACK headers, which need to be specified in the `c_cpp_properties.json` file. The provided link has guidelines on how to do this. Note, however, that it is entirely possible to do command-line compilation, which simply requires some extra switches and options in the compiler command. Throughout this report we will provide such commands, but the equivalent VSCode compilation files, which are named `tasks.json`, are given in the appendices.

In addition to this C/C++ development environment, MATLAB (on Windows 11) is used for processing of results, quick testing, and so on. More importantly, it is also used for comparing the performance (i.e., outputs) of the C/C++ code against a provided MATLAB 'golden model'. We use this golden model to evaluate correctness of the C++ implementation. The specifics of the use of the MATLAB model are described in chapter 5.

It is important to reiterate that these choices are, for the most part, a personal one of the author. It is perfectly possible to repeat this project (and to continue development on it) using a different set up.

3.2 Gnucap

Gnucap is a free and open source circuit simulator created by Albert Davis, and much of its development is done by him, and more recently also by others such as Gennady Serdyuk and Felix Salfelder (see the `AUTHORS` file in the Gnucap source). It is distributed under the GNU General Public License, which can be found here [29].

This section starts by briefly describing how Gnucap can be installed on a Linux machine. It then gives a description of some of the relevant aspects of the software that we use. Specifically, we focus on some of the less intuitive details, for which the available documentation *can* be confusing to readers who are not familiar with the general architecture of circuit simulators. Only details relevant to the project (i.e., the creation of device plugins for transient simulations), will be discussed.

3.2.1 Installation

In this subsection we will describe how the Gnucap source can be installed and built on the described environment.

Gnucap is hosted on the GNU Savannah site. This site has both GNU projects, and non-GNU projects which share a similar philosophy to the GNU project. Gnucap's project page is found here [30], which should be the point of reference for any information on it. Inside the

'Development Tools' box on this page can be found various GIT repositories associated with Gnucap. We are only concerned with the primary one, which can tracked here [31].

To install Gnucap, the best and easiest option is to simply clone the repository directly. Assuming a Linux OS and that GIT is installed, then within a terminal opened in some working directory, the command `git clone https://git.savannah.gnu.org/git/gnucap.git` will install from the HTTPS remote repository. In this project we work with the main version of 'master 2021.01.07'. It is not expected that later development versions will prevent the work in this project from working, though if it does it should be relatively straightforward to fix.

Having downloaded the source code, building is relatively straightforward. Firstly, make sure to `cd` into the top directory (most likely named `gnucap`). Unfortunately it seems that (at the time of writing) some of the documentation available for Gnucap, including within the source files, can be outdated. This is of course excusable given its in-development status, and of course will vary depending on the version. Nevertheless, the `INSTALL` file in the top directory describes how to build Gnucap. For completeness, the steps that are taken in this project are stated here briefly.

(1) In a terminal opened in the top directory, excecute the configuration scription by invoking its executable with `./configure`.

(2) Then, run the command `make install`, or `sudo make install` if that doesn't work. Note that at this step some required C/C++ header `.h` files might be missing depending on platform. The header files that `gcc`/`g++` reference are usually somewhere in `/usr/include` or `/usr/lib`.

(3) Here we note some of the file structure details of Gnucap

   (a) The `lib` directory contains the 'core' functionality of Gnucap which is required for it to run. During building, these files are compiled into the **dynamic library** `libgnucap.so`, which is placed (by default) into the `/usr/local/lib` directory.

   (b) The `apps` directory contains `.cc` model files for various circuit components, implemented as device plugins. These were created by the developers of Gnucap. During building, they are compiled into `libgnucap-default-plugins.so`, a dynamic library file. This is located in `/usr/local/lib/gnucap`.

   (c) The `main` directory contains the `main.cc` file which is compiled into the Gnucap executable binary itself. This executable is located in `/usr/local/bin/gnucap`. Note that this directory (or other if customising) should be added to the `PATH` **environment variable** if it is desired that Gnucap can be invoked from anywhere on the system. In any case, `/usr/local/bin` is typically included in the `PATH` by default. During runtime, the Gnucap binary links to `libgnucap.so` and `libgnucap-default-plugins.so`. To do this, it uses the environment variable `LD_LIBRARY_PATH`, which defines the paths that a linker searches when dynamically linking. It may be necessary to edit `LD_LIBRARY_PATH` (or create it if it doesn't exist) to ensure Gnucap can find these two `.so` files.

   (d) The `Include` directory contains the C `.h` headers that reference the definitions (`.cc` files) in `libs` and `apps`. During compilation they are copied into `/usr/local/include/gnucap`, where they are used during creation of `libgnucap.so` and `libgnucap-default-plugins.so`. These header files must be included in any custom code that uses Gnucap's source, such as in developing plugins.

We can test if Gnucap is properly set up by opening a terminal in any location (or the location of the executable if it has not been added to `PATH`), and simply typing `gnucap` to start it. Something akin to what is shown in listing 3.1 should be displayed.

Listing 3.1: Expected terminal output from starting Gnucap by invoking the binary executable.

```
$ gnucap
Gnucap : The Gnu Circuit Analysis Package
Never trust any version less than 1.0
Copyright 1982-2013, Albert Davis
Gnucap comes with ABSOLUTELY NO WARRANTY
This is free software, and you are welcome
to redistribute it under the terms of
the GNU General Public License, version 3 or later.
See the file "COPYING" for details.
main version: master 2021.01.07
core-lib version: master 2021.01.07
default plugins: master 2021.01.07
gnucap>
```

3.2.2 Plugins

It is critical that an understanding of Gnucap's device plugins be obtained so that they can be reliably used to realise the RC companion model. An attempt is made to understand the system by way of implementing a simple, example plugin. In this section we will endeavour to relay the understanding gained from this example plugin by using it to illustrate Gnucap's transient simulation flow.

Firstly, however, let us discuss some of the important architectural details of Gnucap to enable understanding of the code written in chapter 4. The starting point for the interested reader should be the Gnucap Wiki, located at [32], which is also accessible via the Savannah page [30] under the 'Group Homepage' link.

The important pages here are the 'Snapshot manual' (also called 'Gnucap manual'), and the 'Tech notes' section. Both of these constitute the documentation of Gnucap. These HTML manual pages also contain a tutorial of sorts that can be used to learn Gnucap's command line interface. This is found under 'Examples'. The information on these pages is incorporated, in varying portions, into a PDF file that is also listed on the website. However, the HTML pages seem to be both more up to date and more comprehensive. The PDF manual is still a useful reference, however, and is easier to read at times.

We will primarily focus on the 'Tech notes' page, which provides details on how to develop and contribute to Gnucap. Of interest to this project is the 'Plugins' page, which contains a walk-through of adding various types of functionality to Gnucap. The plugin interface is rather general, and is intuitive and relatively user-friendly in the sense that it abstracts away much of the underlying simulator complexities.

3.2.2.1 Concept

There are a variety of plugin types corresponding to different aspects of Gnucap. These include custom commands (for example, different types of simulation), device models, and other more abstract objects such as languages (for example, Gnucap has support for SPICE, Verilog, and others as modelling and netlisting languages).

Plugins work as dynamic libraries (`.so`) which are dynamically linked into the Gnucap executable during its runtime, just like the core functionality of Gnucap.

The general idea is that we define a custom class in some source code (i.e., a set of `.cc` and `.h` files). This class must be inherited from provided base classes. We then use some 'boilerplate' Gnucap class interfaces and virtual function overrides to implement a desired functionality. The plugin source is then compiled, along with libraries and external dependencies, into a single `.so` file. There are three ways in which this custom `.so` can be used in Gnucap.

(1) The easiest method, and the one we use in this project, is to simply put the compiled `.so` library into a directory that is searched via `LD_LIBRARY_PATH`, and then use the `load` command in Gnucap to link it. For example, if our plugin is named `test.so`, then we'd simply type `load test.so` in Gnucap and it will link the plugin into the runtime.

(2) Another more involved method is to edit the `Make1` file inside the `apps` directory of the Gnucap source folder, and add in the necessary compilation steps such that our plugin source code is compiled into the `libgnucap-default-plugins.so` file. It will then be automatically linked by Gnucap on startup along with the rest of the default devices. Given the error prone nature of this approach, and the fact that debugging would likely be very difficult, it does not seem like a worthwhile approach unless some very large automated toolchain or the like is required, which is not the case in this project.

(3) There is technically a third way, which involves using either the Linux command `chrpath` to 'edit' the Gnucap binary to specify what libraries it should automatically link during runtime, or to augment the Gnucap build steps to use the `-rpath` option to the same effect. Given the nature of this project, such an approach is not useful because it does not yield well to rapid testing.

When we use the `load` command, Gnucap creates a single static object of the plugin class type, and runs its constructor. In most cases we need not define a bespoke constructor for our plugin, and can rely on the constructor of the base class to do the necessary set up. Plugin functionality is instead specified by overriding virtual functions. The Gnucap plugin system is hence quite robust in the sense that we can rely on the core simulator functionality to take care of most of the actual work, and instead just focus on 'telling it' what we want the plugin to do.

3.2.2.2 Device Plugins for Transient Simulations

In this project the only plugin type that we use is the device plugin. Though [32] provides a lot of detail, in this section we discuss some key concepts.

Architectural Details

In Gnucap, the term 'card list' is used to refer to each circuit component in a netlist, with each component being a 'card'.

Specifically, Gnucap uses two classes, `CARD_LIST` to store each component in a linked-list style structure, and `CARD` as the base class of all device models. These classes are declared inside the header files `e_cardlist.h` and `e_card.h`, respectively. Among other things, they contain the virtual functions which can be re-defined in child classes for the purpose of implementing custom DC analysis (the `dc` command in Gnucap), transient analysis (`tr` or `tran`), and AC analysis (`ac`) for that device.

Additionally, there are functions for specifying parameters via the `PARAMETER` class. These are useful for having user-defined values, such Z_{ref} in (2.39). Functions for general details such as ports, number of nodes, and so on are also available.

Not all device models need to implement all possible virtual functions. Unless specific functionality is desired, it is fine to invoke the base class implementation in most cases.

These base classes are shown in Table 3.1, which is adapted from [32]. The arrows show the flow of inheritance. Each base class extends some features of its parent, in addition to having

Table 3.1: Summary of Gnucap device base class types and their inheritance structure. Adapted from [32].

Base class	Description
`STORAGE`	Devices with 'memory', like capacitors and inductors.
↑	
`ELEMENT`	'elementary' devices such as resistors, sources, and TLs.
↑	
`COMPONENT`	Essentially another preliminary class, redefines some functions specifically for devices.
↑	
`CARD`	Base class of all devices and parameters, declares virtual functions.

unique features of their own. The type of base class we use depends on the type of device functionality that we want. For example, recall from section 2.5 that we need to implement a history term $h[n]$ in our model. Therefore, it makes sense to use `STORAGE` as the base class, since it will already contain defined functionality suited to this kind of behaviour[1].

We can illustrate these concepts by implementing a rudimentary model class which is simply an augmented form of a resistor, with the constitutive relation $v[n] = i[n]R^2$. It is defined in a source file `cust_res.cc` and has a class name `CUST_RESISTANCE`. The source code is given in full in appendix A1. Here, we will mention relevant parts via a series of code listings.

Listing 3.2: We set up a device plugin by including the `globals.h` Gnucap header, as well as the header file of the base class that the device inherits from, `e_elemnt.h` in this case. The device class (`CUST_RESISTANCE`) is defined, along with its constructors, and destructor if needed.

Listing 3.2: Example code used at the beginning of a device plugin source file. The plugin class must inherit from one of the base classes.

```cpp
#include "globals.h"
#include "e_elemnt.h"

namespace
{
  class CUST_RESISTANCE : public ELEMENT
  {
  private:
  // copy constructor used to create instances of the device from the static instance
  explicit CUST_RESISTANCE(const CUST_RESISTANCE &p) : ELEMENT(p) {}
  // should use to de-allocate any dynamic memory if create any; otherwise not needed
  // ~CUST_RESISTANCE() {}
  public:
  // default constructor used by dispatcher to create static instance
  explicit CUST_RESISTANCE() : ELEMENT() {}
```

Inside the class definition itself we follow the usual convention of only declaring any member functions, with their definition being done externally. In creating custom models, this will usually consist of listing the virtual functions we want to override, but we can of course also create new functions as desired.

[1] In fact, it turns out that the behaviour indicated in Fig. 2.6 is very similar to the linearised companion model of a capacitor. See the `d_cap.cc` file for details on how Gnucap implements a capacitor.

Listing 3.3: Depending on the base class we inherit from, there are actually a number of virtual function overrides that we are required to have in order for the code to compile. These seem to be the 'essential' functions without which the device cannot be used in the simulator. They perform tasks such as defining how port voltages are obtained (e.g., `tr_involts()`, `tr_involts_limited()`), and behaviour during TSs (e.g., `tr_begin()`, `do_tr()`). As mentioned, in many cases it is perfectly fine to just call the base class version of these functions, and in fact it is a requirement to do so at the beginning of most of them. If we wish to define custom functionality, then we must add our own code. We should also reiterate that there are many possible functions we can re-define, and the reader should consult [32] for more details.

Listing 3.3: Some of the required virtual function overrides for a device plugin. Notice that they relate to simulation methods because Gnucap needs to know how to deal with them during simulation runs.

```
void tr_iwant_matrix() { tr_iwant_matrix_passive(); }
  // obtain port voltages
  double tr_involts() const { return tr_outvolts(); }
  double tr_involts_limited() const { return tr_outvolts_limited(); }
  // same as TR version but for AC
  COMPLEX ac_involts() const { return ac_outvolts(); }
  void ac_iwant_matrix() { tr_iwant_matrix_passive(); }

  // transient analysis functions
  void tr_load() { tr_load_passive(); }     // load amittance matrix and current vector with values calculated during do_tr
  void tr_unload() { tr_unload_passive(); } // removes component from MNA matrix
  void tr_begin();
  bool do_tr();
```

Transient Flow

Almost all of the provided device models implement custom functionality for TSs and most also for AC simulation. Most models seem to use the DC functionality provided by the base class. Here we will present only the functions relevant to TS since that is the focus of the project.

Fig. 3.1 shows an adapted illustration from the Gnucap PDF manual found at [32], showing the TS flow.

```
gnucap> // performing any command involving a device //
  precalc_first();
...
gnucap> tr <start time> <end time> <time step delta>
  tr_begin();
  for each (time step)
  {
    tr_advance();
    for each (iteration)
    {
      do_tr();
      tr_load();
      // solve MNA system //
    }
    tr_accept()
  }
```

Figure 3.1: Simplified Gnucap transient simulation flow, showing only relevant functions. A TS consists of a nested for-loop structure, moving over time steps, and over optimisation iterations within each time step. Adapted from [32].

When we build a netlist, performing any command that relates to a `CARD` in it will trigger the `precalc_first()` function. We will not go over what exactly this function does, but it in general 'sets up' any constants and so on for the device itself. For our purposes in this project, it is useful because it allows us to do some initial function calls; specifically, it is where the VF algorithm is run in the final plugin.

We run TSs in Gnucap by using the `tr` or `tran` command, and specify a start time, end time, and desired time step difference, for which we will reuse the Δ symbol. Once this command is invoked, the `tr_begin()` function is run. This function is similar to `precalc_last()` in that it can be used to set up initial values and so on, but is specific to TSs. It is only run once per TS; immediately after invoking the command. Hence, if we have anything we'd like to do only once within any given TS, we can do it here.

The solver then enters into a for-loop across the specified time steps. Note that by default, the simulator dynamically adjusts Δ as it thinks appropriate at the end of each of these iterations. However, it will only write out results at integer multiples of the Δ we specify in the `tr` call. This is not usually a problem and is typical of circuit simulators, but for some devices, such as capacitors and TLs, Gnucap internally limits the range that Δ can take to ensure accurate results. We will not go into this here, but it will be brought up again in the discussion on the RC model results in section 5.3, since it has ramifications for the history term, $h[n]$, in our model.

The `tr_advance()` function is only invoked once per time step; before the MNA calculations. As such, we can use `tr_advance()` to perform any Δ-dependant computations. This function is used mainly by dynamic devices which might need to maintain or update some state variable during the simulation.

There is a similar function, `tr_accept()`, which can instead be invoked after the MNA calculations for that time step. It is likewise mostly only useful for dynamic and storage elements. We can use it to update any terms that will be used in the next time step. At this point it should be clear that these two functions are critically important to the functioning of the RC model. This is discussed further in section 4.3.

The core of the transient flow is the `do_tr()` function. Note that this is inside another for-loop across 'iterations'. This refers to the fact that the simulator will iteratively seek to find a correct convergence value via some optimisation scheme, such as Newton-Raphson. This is not the focus of the project so we do not investigate it; the important point is that the `do_tr()` function may therefore be executed multiple times within a given time step. Hence, when attempting to create a model that performs iterative updating of terms and keeps track of a history, we should be very careful *not* to place any of the updating code inside `do_tr()`, since it would have unintended effects and give an inaccurate or misleading result.

For our purposes, the `do_tr()` function simply evaluates and assigns values (calculated using intermediary variables which *should be constant for that given time step*) to the parameters of the model's constitutive relations (KCL and KVL equations) and queues them to be stamped into the MNA matrix. This stamping is in fact done separately in `tr_load()`, which supposedly isolates any errors.

Let us return to the `cust_res.cc` example plugin. This model only implements custom behaviour in the `tr_begin()` and `do_tr()` functions, which we now briefly explain.

Listing 3.4: For some virtual functions, it is required that we call the base class' version before we do anything else. This is usually done to prevent issues that could arise if we forget some important step in our custom code. `tr_begin()` requires this.

After this, we define any values that will remain constant during the transient simulation. This is particularly useful in more advanced models where we can use read-in parameters to assign important values. All device models must take in at least one parameter (for example, resistance in a resistor), with many having optional extra parameters. This 'basic', mandatory parameter has its value, predictably, returned by calling the `value()` function. In the case of this model the intent is to take in resistance just like an ordinary resistor, but instead we assign the squared value (`value()* value()`) in the constitutive relations. The constitutive relations are implemented by two objects, `_y[0]` and `_m0`, which are explained under the next heading.

Listing 3.4: Example of a `tr_begin()` override. This function sets up initial and constant values for the TS.

```
void CUST_RESISTANCE::tr_begin()
  {
  // need to call base class virtual first
  ELEMENT::tr_begin();

  // values that will remain constant during sim
  _y[0].f0 = _m0.c0 = 0;
  _y[0].f1 = (value() != 0.) ? value() * value() : OPT::shortckt; // R^2
  _m0.c1 = 1. / _y[0].f1;                                         // 1 / R^2
  }
```

There are other 'housekeeping' measures such as ensuring that the specified resistance does not yield a short circuit, but we will not go into this as it is fairly self explanatory.

Listing 3.5: For the `do_tr()` override, we simply evaluate the non-constant terms of the constitutive relations for this time step. There are additional functions related to the simulation architecture which we will not go into, such as checking if the provided values are appropriate via `conv_check()`, `converged()`, and so on. These checking functions seem to only be a loose requirement (in that the model will compile and operate in Gnucap without them), but it is nevertheless a good idea to have them just to catch any errors, such as divisions by zero.

Listing 3.5: Example of a `do_tr()` override. This function serves to stamp the parameters of the device constitutive relations into the MNA matrix.

```
bool CUST_RESISTANCE::do_tr()
  {
  // (1) assign non-constant FPOLY terms
  _y[0].x = tr_involts_limited() * (1. / _y[0].f1); // i[n]

  // (3) necessary process functions
  set_converged(conv_check()); // check convergence
  store_values();              // push values to previous iteration (i.e., store
     them!)
  q_load();                    // queue for adding to MNA matrix

  // (3) 'convert' to CPOLY values
  _m0.x = tr_involts_limited(); // v[n]

  return converged();
  }
```

Polynomial Approximation

The `_y[0]` and `_m0` entities are objects of a custom Gnucap type that all device models in Gnucap have (they are declared inside the `ELEMENT` base class, from which all devices inherit). They are used to represent the constitutive relations of the device as a first order Taylor series. The class names are `FPOLY1` and `CPOLY1`, respectively, though we will drop the '`1`' for simplicity, and because there are no higher order versions in Gnucap as of yet.

A first order Taylor series expansion of a function $f(x)$ evaluated at a point x_i, expanded about a nearby point a, is in general given by

$$f(x_\mathrm{i}) = f(a) + \frac{\mathrm{d}f(a)}{\mathrm{d}x_\mathrm{i}} \cdot (x_\mathrm{i} - a). \tag{3.1}$$

The `FPOLY` object, `_y[0]`, implements this approximation, where its attributes are

`_y[0].f0` $:= f(a)$,

`_y[0].f1` $:= \frac{\mathrm{d}f(a)}{\mathrm{d}x_\mathrm{i}}$, and

`_y[0].x` $:= x_\mathrm{i}$.

More intuitively, we can say that `FPOLY` represents the function $f(t) = f_0 + f_1 \cdot (t - x) :=$ `_y[0]` = `_y[0].f0 + _y[0].f1 * _y[0].x`, where $(t - x)$ is some small displacement of the value $x :=$`_y[0].x`, for instance, its value at a previous time step. For the purposes of writing code, it is more useful to ignore the t displacement and assume that the MNA solver will do this for us, which indeed it does within the 'iteration' for-loop of Fig. 3.1. We should remember, however, that this means that since `_y[0].x` is being displaced inside the simulator during each time step, that it *should not be assigned something that is constant for a given time step*.

From the perspective of Gnucap, this seems to mean that we should assign `_y[0].x` inside `do_tr()` using some function evaluations, whose return value the simulator can change during each iteration of `do_tr()` in that time step. In practice this implies that we always assign `_y[0].x` to be the voltage across the device terminals at that time step n, $v[n]$, which is returned by `tr_involts()` or `tr_involts_limited()`. Alternatively, we assign it the current through the device at time step n, $i[n]$, which is simply some scaling of the voltage. Even though the notation of $v[n]$ and $i[n]$ imply they remain constant for a given time step, in actuality they are iterated on until they converge, as mentioned previously.

This requirement on `_y[0].x` is seen in listing 3.5, where we assign it to $i[n]$.

The `CPOLY` type is more or less equivalent, except that it is a Maclauran series, i.e., $a = 0$. Mathematically, it is defined as

`_m0.c0` $:= f(0)$,

`_m0.c1` $:= \frac{\mathrm{d}f(0)}{\mathrm{d}x_\mathrm{i}}$, and

`_m0.x` $:= x_\mathrm{i}$.

Or, writing it more intuitively by dropping the subscript on the x attribute[2], $f(x) = c_0 + c_1 \cdot x :=$ `_m0` = `_m0.c0 + _m0.c1 * _m0.x`.

Just like `FPOLY`, however, we must ensure that `_m0.x` can vary with the internal dynamics of the simulator at each time step, and so it is also typically set to $v[n]$ or $i[n]$ or some scaling of them. We again see this in listing 3.5, where it is assigned as $v[n]$.

Unlike the `x` argument, `_y[0].f0`, `_y[0].f1`, `_m0.c0`, and `_m0.c1` are constants for a given time step, and are typically assigned values which are evaluated in `tr_advance()`, `tr_accept()`, or made constant for the TS in `tr_begin()`. In the `cust_res.cc` example, we use the `value()` (i.e., resistance), as shown in listing 3.4.

Gnucap appears to use the convention that `FPOLY` is represents the KVL version of the constitutive relation, and that `CPOLY` represents the KCL from which the MNA matrix obtains the stamp values. However, this is not a rule in any sense, and some devices, such as capacitors and others which inherit from `STORAGE`, have two `FPOLY` objects. It is perfectly reasonable to declare and use multiple `FPOLY` objects to represent any time-step varying relation in our

[2]Remember that the `CPOLY` `x` argument is not the same as the `FPOLY` `x`; they exist as separate attributes in separate types.

model even if they are only used to keep track of intermediary values. These functions need not be explicit functions of $v[n]$ and $i[n]$, either, but of course must be if we wish to stamp them into the MNA matrix. This is what is done in the RC model, described in section 4.3, wherein we use an `FPOLY` to represent the $i_c[n]$ function in (2.39).

In any case, there is invariably only a single `CPOLY`, which defines what should be stamped into the MNA matrix for that device.

The two types do in fact have copy constructors which allow conversion between them (consult the Gnucap source code for details), but it is not necessary to use these, nor does it seem necessary that our formulations of any `CPOLY` and `FPOLY` objects obey the conversion rules indicated by the copy constructors. As such, we can use `FPOLY` and `CPOLY` rather liberally. It is hence fairly straightforward to define device models once we (1) have a well-defined constitutive relation, and (2) understand `FPOLY` and `CPOLY` well enough that we can represent the relation using them.

Referring to listings 3.4 and 3.5, we illustrate now how `cust_res.cc` implements `_y[0]` and `_m0`.

$$\begin{aligned}
v[n] = i[n]R^2 \quad &\rightarrow \texttt{\_y[0].f0} = 0\\
&\rightarrow \texttt{\_y[0].f1} = R^2\\
&\rightarrow \texttt{\_y[0].x} = i[n]\\
i[n] = v[n]\tfrac{1}{R^2} \quad &\rightarrow \texttt{\_m0.c0} = 0\\
&\rightarrow \texttt{\_m0.c1} = \tfrac{1}{R^2}\\
&\rightarrow \texttt{\_m0.x} = v[n]
\end{aligned}$$

That is, we use adapt `_y[0]` to represent the KVL constitutive relation, and `_m0` to represent the KCL whose attributes are stamped into the MNA matrix. An important point is that `_y[0].f0`, `_y[0].f1`, `_y[0].x`, `_m0.c0`, `_m0.c1`, and `_m0.x` are attributes which encapsulate the constitutive relations represented by the `_y[0]` and `_m0` objects; there is hence no need to explicitly 'assign' `_y[0]` and `_m0`. That is, we do not write out the code for $v[n] = i[n]R^2$ or $i[n] = v[n]\frac{1}{R^2}$, but rather we assign $v[n]$, $i[n]$, and R^2 (performing intermediary calculations and function calls if needed) to the attributes of `_y[0]` and `_m0`, and let the Gnucap MNA do the rest. Once this concept is grasped, it should be easy to understand how to use them.

Compiling the Plugin

Listing 3.6: Each plugin source `.cc` file must finish by creating an instance of the defined class, and then 'registering' it with the appropriate `DISPATCHER` object. In Gnucap, `DISPATCHER` is the class responsible for loading the plugin into the Gnucap runtime. For devices, the corresponding `DISPATCHER` object is the `device_dispatcher`. Note that we also include a 'specifier' by which Gnucap references the device. We can use multiple names separated by a '|'. In this case, we use `p|custr`, meaning that when creating a netlist we can use either `p` or `custr` to instantiate this device.

Listing 3.6: Example of adding a device plugin to the Gnucap dispatcher. This must be done at the very end of any plugin file source file.

```
  CUST_RESISTANCE p1;
  DISPATCHER<CARD>::INSTALL d2(&device_dispatcher, "p|custr", &p1);
}
```

In order to use the plugin it must be compiled into a `.so` file. Since we are using VSCode in this project, compiler options are specified inside of a `tasks.json` file. For `cust_res.cc`, this file is given in appendix A2. In listing 3.7 we give the command line equivalent.

Listing 3.7: Example command for compiling a plugin source into a dynamic library file, using `g++`.

```
$ g++ -I/usr/local/include/gnucap -L/usr/local/lib/ -shared -fPIC cust_res.cc -o
    cust_res.so
```

We use the `-I` option to specify the include directory, which points to the `.h` files, and then the `-L` option to specify the link directories, in this case pointing to where the `libgnucap.so` and `gnucap-default-plugins.so` are located. We must use the `-shared` option, which specifies that we want a single resulting file, the `-fPIC` option, which makes the code **position independent** (this is equivalent to producing a dynamic library) and the `-o` option which specifies that we want an object file. Consult [33] for the specifics of these options. The result is that we get a single `.so` file, named `cust_res.so`.

An important point is that Gnucap requires that all custom plugins be compiled from source on each machine. This means that we cannot just distribute a `.so` file, but each user of the plugin needs to build their own `.so` from some provided source.

Example Simulation

We will use the simple circuit in Fig. 3.2 to illustrate use of the model `cust_res.so` in a Gnucap TS. In Fig. 3.2 we indicate the node labels and the component names as they appear in

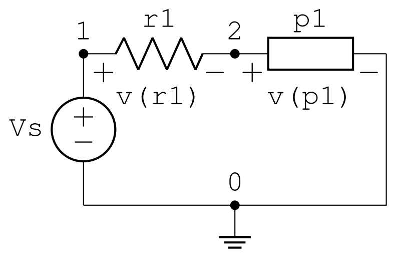

Figure 3.2: Simple resistive network used to illustrate the use of a device plugin instance, `p1`, in a Gnucap netlist. We use Gnucap notation for the nodes and so on.

Gnucap. The corresponding terminal commands to produce this circuit is shown in listing 3.8.

Listing 3.8: Example sequence of terminal commands used to load a plugin, build a netlist with it, and perform a TS. The TS results are printed to the terminal.

```
$ gnucap
...
gnucap> load cust_res.so
gnucap> build
>V1 1 0 5
>r1 1 2 1k
>p1 2 0 1k
>
gnucap> print tr v(r1) v(p1)
gnucap> tr 0 5 1
#Time       v(r1)      v(p1)
 0.         0.004995   4.995
 1.         0.004995   4.995
 2.         0.004995   4.995
 3.         0.004995   4.995
 4.         0.004995   4.995
 5.         0.004995   4.995
gnucap>
```

We will not go into detail on Gnucap commands as it is well documented at [32]. Instead we mention them as we use them in this report.

As noted previously the `load` command links to a specified plugin, in this case our `cust_res.so` (where we assume that this file is somewhere referenced by `LD_LIBRARY_PATH`). We can use the `build` command to build a netlist, or add to an already existing one since Gnucap keeps track of entered elements (we can use `delete` to remove an element, or `clear` to delete the entire netlist). The format for instantiating a device is of the general format `<device identifier> <+ve terminal> <-ve terminal> [parameters]`. The first letter or sequence of letters in the identifier must be the device specifier. For example, `r` is for Gnucap's default resistor, `v` denotes an independent voltage source, and we use `p` for our `cust_res.so`.

The voltage source `Vs` has been given a value of 5 V, and both `r1` and `p1` a value of 1 kΩ. These become the return value of `value()` in the device model, as discussed previously.

We then set up a TS, using the `print` command to indicate what values we want written out at each time step (this can be thought of as a 'measure' command with which we can probe specific nodes and differences), in this case the voltage across `r1` and `p1`. The TS itself is run for 5 s with a Δ of 1 s. Fig. 3.2 is in effect a resistive divider, and so we expect `v(p1)` to be $5 \cdot \frac{1000^2}{1000+1000^2} \approx 4.955$ V, and hence for `v(r1)` to be $5 - 5 \cdot \frac{1000^2}{1000+1000^2} \approx 4.995$ mV. This is indeed what we see from the printed results, indicating that `cust_res.so` does what we want it to do.

3.3 LAPACK

Recall from section 2.3 that the VF algorithm involves matrix operations. Computational packages such as MATLAB can easily perform these without much thought from the programmer since its language abstracts away most of the details.

However, this project implements VF in C++, which lacks a convenient native linear algebra solution at the time of writing[3]. A naive implementation could be attempted, but would likely be error-prone and lack robustness. Furthermore, it would consume a significant amount of time.

We therefore use the LAPACK library to do most of this work. LAPACK provides routines for every operation we need. Indeed, this is the advantage of using LAPACK over similar C/C++ libraries, such as the Eigen library; it provides single function calls that can do involved operations such as linear least squares and eigenvalue calculations, as well as more rudimentary operations such as matrix multiplication[4]. LAPACK is based on an implementation of the well-known **Basic Linear Algebra Subprograms** (**BLAS**) specification, and so we can expect its behaviour to be very robust. [35].

LAPACK allows computation with complex-valued matrices, which is very convenient for us since VF operates in the FD. However, we don't actually need to exploit this functionality because the key computations in VF all occur on real-valued matrices.

3.3.1 Installation

LAPACK is written in the Fortran language, but has C bindings known as the 'LAPACKE C INTERFACE', or simply LAPCKE [36], which is what we use in this project. The LAPACKE interface is provided with the LAPACK source code. This section describes how to build LAPACK (and LAPACKE) such that it can be used in our C/C++ VF code.

Note that because we load the RC model plugin as a `.so` file, then we must ensure that the LAPACK libraries that are linked to when compiling the model are also dynamic libraries.

[3]Interestingly, the upcoming C++ 26 standard will likely include this in some form [34].

[4]Having said that, perhaps in the future it might be worthwhile to look more closely at each of these operations in order to improve the accuracy of the VF model. The discussion on least-squares methods elsewhere in this report alludes to this.

Hence, the procedure we show in this section creates `.so` versions of the LAPACK library files, not the usual `.a` (static library) versions.

Because we use the C bindings, we need to build in the following order: (1) LAPACK itself, (2) LAPACKE, and (3) CBLAS, which is a C interface to the Fortran BLAS implementation which underlies LAPACK. The `README.md` file in the LAPACK top directory (which should be called `lapack-3.12.0` if it has not been renamed) specifies a few options for installation. We simply build with the GCC compilers (`g++`, `gcc`) using the makefiles provided with the LAPACK source.

We do need to make some modifications to these makefiles, which are relatively straightforward. In the top directory, there is a file named `make.inc.example`. This file should be renamed to `make.inc`. Then we make the following modifications to it.

Add in `-DHAVE_LAPACK_CONFIG_H -DLAPACK_COMPLEX_CPP -fPIC` to the end of the `CFLAGS` line, after the optimisation flag. These additional flags specify that we want to have the LAPACKE complex types be C++ complex types (i.e., `std::complex`). The `-fPIC` flag specifies that we want the compiled libraries to be `.so` files. These changes are shown in Fig. 3.3

```
CC = gcc
CFLAGS = -O3
```

$\downarrow$

```
CC = g++
CFLAGS = -O3 -DHAVE_LAPACK_CONFIG_H -DLAPACK_COMPLEX_CPP -fPIC
```

Figure 3.3: Changes to the C/C++ compilation options for LAPACK inside the `make.inc` file. We switch to the `g++` compiler, and change the complex types and library file format.

Additionally, we should add the `-FPIC` flag to the Fortran options, shown in Fig. 3.4.

```
FFLAGS = -O2 -frecursive
FFLAGS_DRV = $(FFLAGS)
FFLAGS_NOOPT = -O0 -frecursive
```

$\downarrow$

```
FFLAGS = -O2 -frecursive -fPIC
FFLAGS_DRV = $(FFLAGS)
FFLAGS_NOOPT = -O0 -frecursive -fPIC
```

Figure 3.4: Changes to the Fortran compilation options for LAPACK inside the `make.inc` file. We simply add the flag for compiling into a dynamic library.

When LAPACK finishes compiling, it produces five libraries that we need to link to when building our VF code. The names and locations of these are also specified in the `make.inc` file. By default these have a `.a` file extension since the default makefile produces static libraries. We need to change the `.a` file extensions to `.so` [5]. Fig. 3.5 shows these changes.
Appendix A3 shows the full code for the modified `make.inc` file, such that it can be easily copied as needed.

After these modifications are made to the makefile, we can build each the three packages that we need. The steps to do this are

[5]Although it is unclear if changing these actually manifests as any changes in the binaries produced, it is better to be sure. It also prevents confusion by ensuring consistency.

```
BLASLIB      = $(TOPSRCDIR)/librefblas.a
CBLASLIB     = $(TOPSRCDIR)/libcblas.a
LAPACKLIB    = $(TOPSRCDIR)/liblapack.a
TMGLIB       = $(TOPSRCDIR)/libtmglib.a
LAPACKELIB   = $(TOPSRCDIR)/liblapacke.a
```

↓

```
BLASLIB      = $(TOPSRCDIR)/librefblas.so
CBLASLIB     = $(TOPSRCDIR)/libcblas.so
LAPACKLIB    = $(TOPSRCDIR)/liblapack.so
TMGLIB       = $(TOPSRCDIR)/libtmglib.so
LAPACKELIB   = $(TOPSRCDIR)/liblapacke.so
```

Figure 3.5: Changing the file extension of the LAPACK libraries inside the `make.inc` file from `.a` to `.so`. This ensures the libraries are dynamic.

(1) From the LAPACK top directory, `lapack-3.12.0`, run the command `make`, ensuring beforehand that the `make.inc` file in this directory is modified as described above,

(2) `cd` into the `LAPACKE` directory and run `make` here also,

(3) and finally, `cd` into the `CBLAS` directory and run `make`.

Note that when each `make` command is run, that the actual build process will take some time because LAPACK also performs automatic tests to ensure everything is working correctly.

Once the builds are finished, the `lapack-3.12.0` top directory should contain five library files: `libcblas.so`, `liblapack.so`, `libtmglib.so`, `liblapacke.so`, and `librefblas.so`. These are the libraries we will link to in our VF code.

3.3.2 Functions Used

LAPACK contains many functions, for which documentation is found here [37]. This documentation describes only the Fortran implementation, but since the LAPACKE bindings are an interface for calling these, it is equally applicable to the C/C++ versions[6].

The identifier for the C versions of all these routines is simply the Fortran routine identifier, but with `LAPACKE_` appended as a suffix. The first letter of the Fortran routine identifier describes the data type it used with; `s` for single precision (`float` type in C/C++), `d` for double precision (`double` type in C/C++), and `c` and `z` for single and double precision complex (`std::complex<float>` or `std::complex<double>`, in our case). In other words, LAPACK provides four different versions of each routine depending on the data type we need. We only use the real double precision case (`d`) in this project.

We now briefly outline the LAPACK functions that we use in the VF code of 4.2, highlighting any important points or rationale for their use as appropriate. For simplicity, we use the Fortran base identifiers.

Matrix Layout

It is appropriate to begin by explaining the difference between row and column major ordering, since LAPACKE functions require us to specify this as the first argument.

[6]The only difference is that some of the function arguments might be different or unique. For instance, the LAPACKE functions require us to specify the layout of the matrices. In LAPACK they are all assumed to be column-major since this is the format used by Fortran. C/C++ on the other hand, uses row-major, and so it seems that LAPACKE, and CBLAS, support both formats.

In C/C++, all matrices are just large arrays which are stored sequentially in memory. Therefore, when performing computer calculations, we need to tell the software how exactly we want the matrix to be structured, so that calculations will give the correct result.

Assume we have a 4×2 2D array (i.e., a 4×2 matrix), which in matrix-notation is

$$\mathbf{V} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}. \tag{3.2}$$

There are two ways we can store or access this. In **row-major**, the matrix would be stored or accessed as `V = {1, 2, 3, 4, 5, 6, 7, 8}`.

In **column-major**, it would be stored or accessed as `V = {1, 3, 5, 7, 2, 4, 6, 8}`.

That is, in row-major we traverse the matrix row-wise, and in column-major we traverse the matrix column-wise.

Let us define a general 2D-array, `matrix`, of dimensions $m \times n$, where m is the number of rows, and n is the number of columns. We can use either row-major access *or* column-major access, regardless of the format used to store it. As long as we are careful with the parameters we specify in any function calls, it should still produce the same sequence of numbers.

To understand this, first note that we can use a nested for loop to traverse the matrix. For row-major traversal we perform the algorithm shown in listing 3.9, and for column-major traversal we use that in listing 3.10

Listing 3.9: Algorithm for row-major traversal of a 2D array.

```
ld = n; // # columns
for (r = 0; r < m; r++)
  for (c = 0; c < n; c++)
    matrix[r * ld + c]
```

Listing 3.10: Algorithm for column-major traversal of a 2D array.

```
ld = m; // # rows
for (c = 0; c < n; c++)
  for (r = 0; r < m; r++)
    matrix[r + c * ld]
```

These traversal algorithms are valid for both the filling of dynamic arrays with data (i.e., storage), and for accessing the data in an array in a consistent manner.

The `ld` parameter is called the **leading dimension** of the matrix, and for row-major it is equal to the number of columns, and for column-major it is equal to the number of rows. Using the same matrix $\mathbf{V}$ above, let us assume that it is stored as row-major. This means that we should specify the number of rows $m = 4$, and number of columns $n = 2$. Running both traversal algorithms on such a system produces the sequence of numbers shown in Table 3.2.

We see that if we want row-major (i.e., row-wise) access of values, then *both* traversal algorithms will produce the correct result. However, this is precisely because we specify the same row and column sizes in both cases; i.e., the number of rows is $m = 4$ and the number of columns is $n = 2$.

If instead we wanted the matrix to be stored in column-major (i.e., column-wise), then we'd need to swap the row and column dimensions. That is, we consider the matrix to be the transpose of that shown in (3.2), with $m = 2$ and $n = 4$. This is equivalent in effect to swapping `r` and `c` in the indexing operations of the matrix, which is equivalent to transposing the matrix from the perspective of accessing its values. If we stored the matrix as row-major, this would of course not change its consecutive layout in memory.

Table 3.2: Traversal sequence of a matrix stored in row-major layout, using both row-major and column-major traversal schemes. Notice that the result is the same if the definitions of number of rows m and number of columns n remains consistent with the layout we want.

row-major	**`r`**	0	0	1	1	2	2	3	3
traversal;	**`c`**	0	1	0	1	0	1	0	1
`ld = n`	`V[r * n + c]`	1	2	3	4	5	6	7	8
column-major	**`c`**	0	0	0	0	1	1	1	1
traversal;	**`r`**	0	1	2	3	0	1	2	3
`ld = m`	`V[r + c * m]`	1	2	3	4	5	6	7	8

So, even though both row-major and column-major traversal leads to the same sequence, this is *only* the case if we are consistent and correct with our definitions of the parameters m (number of rows), n (number of columns), and `ld`. If these parameters are not set correctly in the function call, according to the matrix layout mode that we specify, then the access sequence will be incorrect and hence lead to incorrect results. More nefariously, it can also lead to undefined behaviour. This can result in segmentation faults and unpredictable program crashes. It would, more annoyingly, make debugging very difficult since *sometimes* out of bounds access and assignment seems to be allowed in C/C++ dynamic arrays, depending on the platform and compiler used!

To provide some robustness against users giving incorrect parameters, every LAPACK routine in fact has a series of checks, outputting an error code if they fail and which we can use to identify the problem more easily. If our definition of m, n, and `ld` don't match up with the layout order we specify, then the routine will typically catch this by checking that `ld` is no larger than m (number of rows) in the case of column-major, and no larger than n (number of columns) in the case of row-major.

In the VF code of section 4.2, We use the same algorithms as in listings 3.9 and 3.10 when constructing and accessing matrices, even though strictly speaking this is not required. However, it is done because LAPACK routines use these same structures when performing computations, so it was desired to remain consistent. It is also quite useful as a debugging aid, since the exact index at which an issue arises can be uncovered via a simple if-statement, or by printing out of results to see if it is the correct.

In any case, all LAPACKE routines support both schemes and so it is useful to become familiar with one kind and to use it repeatedly. We use row-major because its more intuitive, and because supposedly this is the way that C/C++ naturally stores arrays in memory.

`dgelss` : linear least squares

Listing 3.11: Declaration of the function `LAPACKE_dgelss`. From `lapacke.h`.

```
lapack_int LAPACKE_dgelss( int matrix_layout, lapack_int m, lapack_int n,
    lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s
    , double rcond, lapack_int* rank );
```

The declaration of the function `LAPACKE_dgelss()` is shown in listing 3.11, which is taken from the `lapacke.h` header file. The `lapack_int` type is defined as either a C `int32_t` or `int64_t` type, and is intended to replace Fortran's integer type. In this project we do not pay much attention to the fine details of data types, or to which are most efficient to use. In later improvements of the project work it would be desirable to do so.

We use this function in the VF code to solve the system (2.17b) (for all iterations, not just the initial). The final step of the VF algorithm uses it again to solve for the model residues $\tilde{\mathbf{r}}$ and model remainder $\tilde{K}$, as per system (2.24).

The documentation for the function is provided at [38]. We note here some important details about the function.

(1) The system matrices $\bar{\mathbf{A}}_{\text{real}}$ and $\mathbf{A}_{\text{f}}$ are generally **rank-deficient**. This is a result of the VF formulation itself. Because the system matrix consists of PFE terms of the general form $\frac{1}{s_n - p_k}$, it means that as $s_n \to p_k$, the nearby rows begin to look identical within machine precision. That is, not all rows are independent. In effect, this means that there is repetition of some data points within the system. For our purposes, rank deficiency implies that there is no unique solution to the VF system. Hence, different algorithms, platforms, and so on may produce different but equally valid solutions. We will touch on this again in chapter 5.

(2) This rank deficiency means that we must be careful about our function choice. Rank-deficient systems have a determinant of 0, which makes computing a least squares solution non-trivial. There are six routines in LAPACK which can find a 'standard' least squares solution, but only three of them allow rank-deficient system matrices. One of these is `dgelss`, though the other two options, `dgelsd` and `dgelsy` might work just as well. The benefits and drawbacks of each are not investigated in this project; we merely desire a function that allows rank deficiency. An additional consideration in function choice is the fact that the VF system matrix is very unlikely to be a square matrix. `dgelss` allows non-square matrices.

(3) The function takes in an argument `a`, being an `lda-by-n` system matrix, and `b` being an `ldb-by-nrhs` output matrix[7]. Note that these two matrices are passed as 1D arrays (i.e., pointers to the first element). This is why it's important to consider row-major and column-major ordering; from the computer's perspective, in a single consecutive data string.

Once the function is finished, the, the `n-by-nrhs` solution vector overwrites the first `n-by-nrhs` elements of `b`. In our case `nrhs` is simply equal to 1, and so we need merely loop over the first `n` values of the overwritten `b` to get the solution vector.

dgeev: eigenvalues of a matrix

Listing 3.12: Declaration of the function LAPACKE_dgeev. From lapacke.h.

```
lapack_int LAPACKE_dgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
    double* a, lapack_int lda, double* wr, double* wi, double* vl, lapack_int ldvl,
    double* vr, lapack_int ldvr );
```

The declaration of the `LAPACKE_dgeev()` function is in listing 3.12. The documentation of this function is at [39]. This function is used in the VF code to compute the eigenvalues of the matrix **H** as given in (2.18), for each iteration of the algorithm.

This function takes in an `n-by-n` array `a`, with `lda=n` in our case. We also provide two `n`-arrays, `wr` and `wi` which store the real and imaginary components of the computed eigenvalues upon completion of the function. The other arguments are used to optionally compute the eigenvectors, which we don't need so we will not use.

[7]Here `lda` and `ldb` are the leading dimensions of the two matrices. More generally, the leading dimension can be smaller than the 'full' dimensions of a matrix, allowing us to access just a section of the matrix. In our case we do not do this and use the full matrix every time, and so `lda` and `ldb` are assigned as per listings 3.9 and 3.10. We also use row-major layout in calls to `dgelss`. Hence `lda` = `m`, the number of rows of `a`, and `ldb` = `m` because `m` also gives the number of rows of `b`, since it must necessarily be equal to the number of rows of `a`.

In the VF code, we use this function on the matrix $\mathbf{H}_\mathrm{r}$, which is constructed using the real pole guesses, and $\mathbf{H}_\mathrm{c}$, which is constructed using the complex pole guesses. We then simply concatenate the eigenvalues from each and assign them to the poles guess of the next VF iteration after some filtering operation to remove the negative frequency poles[8].

`dgemm`: matrix multiplication

Listing 3.13: Declaration of the function `cblas_dgemm`. From `cblas.h`.

```
void cblas_dgemm(CBLAS_LAYOUT layout, CBLAS_TRANSPOSE TransA, CBLAS_TRANSPOSE
    TransB, const CBLAS_INT M, const CBLAS_INT N, const CBLAS_INT K, const double
    alpha, const double *A, const CBLAS_INT lda, const double *B, const CBLAS_INT
    ldb, const double beta, double *C, const CBLAS_INT ldc);
```

LAPACK does not in fact have routines for 'rudimentary' matrix operations such multiplication, but the accompanying BLAS routines on which LAPACK's are based are also accessible to us via the CBLAS interface. Here we use one such function, `dgemm`. To use the C interface, we append `cblas_` to the front as a suffix. This function, for our purposes, performs matrix multiplication. We use it to compute the column-vector times row-vector operation $\mathbf{b}_\mathrm{z}\hat{\mathbf{c}}_\mathrm{z}$ as per (2.18).

The declaration of this routine is given in listing 3.13, with corresponding Fortran documentation available here [40]. We again see some custom types; `CBLAS_LAYOUT`, and `CBLAS_TRANSPOSE`, which are simply C enumerations (i.e., integer types), and `CBLAS_INT`, which is defined as the C `int32_t` type much like `lapack_int`. The macros for these are in `cblas.h`.

The `TransA` and `TransB` arguments specify if we are using transposed matrices, which we are not. We pass in an `M-by-K` array `A` and the `K-by-N` array `B`, which are multiplied to give the `M-by-N` matrix `c` upon the function's completion. In our case `K=1` always since we are using column and row vectors for `A` and `B`.

The `alpha` and `beta` parameters are optional scalers for the inputs, which we set to 1 and 0, respectively. See [40] for details.

[8]This results in removing the negative conjugate pairs of the complex σ zeroes, and hence the poles guess for our next iteration is roughly half the number of these zeroes. See section 4.2 for further details as it relates to implementation.

Chapter 4

Implementation

Having provided the necessary technical elaboration in the previous chapters, this chapter will detail the implementation code itself. New concepts are only be introduced where it is thought more appropriate to do so here than in the previous chapters.

The implementation code is split between two files, (1) `vf.h`, which implements the VF algorithm and which is given in full in appendix A4, and (2) `rc_model.cc`, which implements the RC companion model of Fig. 2.6 as a Gnucap device plugin. Its code is given in full in appendix A5. In general, only the distinct functional blocks of the code are discussed here, with reference to how they relate to previous developments in chapters 2 and 3. For more fine-grained details, the reader is referred to the code files themselves, and also to Gnucap's site, [32],

4.1 Compiling

Firstly, we briefly state how to compile the given code. We compile `rc_model.cc`, which includes `vf.h` as a header. VSCode is used for development in this project, but here we again state the command line equivalent, shown in listing 4.1, of the `tasks.json` file. This file is given in appendix A6.

We of course assume that LAPACK and Gnucap have been installed correctly. We also assume that this compilation command is being invoked in the same directory that contains `rc_model.cc` and `vf.h`, and that LAPACK is installed in a directory `~/code/`.

Listing 4.1: Command for compiling `rc_model.cc` into a dynamic library file, `rc.so`, using `g++`.

```
$ g++ -shared -fPIC rc_model.cc ~/code/lapack-3.12.0/liblapack.so ~/code/lapack
    -3.12.0/libtmglib.so ~/code/lapack-3.12.0/liblapacke.so ~/code/lapack-3.12.0/
    librefblas.so ~/code/lapack-3.12.0/libcblas.so -o rc.so -I~/code/lapack-3.12.0/
    LAPACKE/include -I~/code/lapack-3.12.0/CBLAS/include -I/usr/local/include/gnucap
     -L~/code/lapack-3.12.0 -L/usr/local/lib/ -l lapack -l lapacke -l gfortran -l
    cblas -l tmglib -l refblas
```

We use the `-shared` and `-fPIC` options to create a single `.so` file, named `rc.so`, which is what is loaded into Gnucap. Since we are using the LAPACK libraries within this `.so`, we should also specify them as 'source' files to the compiler; it is not enough to merely link to them[1]. We then specify the include directories for the Gnucap header files, in addition to the LAPACKE and CBLAS header files. The `-L` option specifies directories in which to search for linking targets. The `-l` option tells the linker that we want the LAPACK libraries specifically. The GCC Fortran compiler, `gfortran`, also seems to be required.

[1]At least it appears that this is the case. If we do not have the LAPACK `.so` files listed as so, then the compilation will still complete but the plugin will not work. It is very possible that a naive mistake is being made here since the GCC is not understood in any great detail by the author.

4.2 Vector Fitting Implementation

VF is implemented as a function, `do_vector_fitting()`, whose header is given in listing 4.2. `rc_model.cc` calls this function, providing the required arguments. The first three arguments are pass-by-reference, `&p`, `&r`, and `&rem`. These are declared in the caller and filled in with the model poles, model residues, and model remainder at the end of the VF function as shown in listing 4.3. Note that the function returns 0 on failure and 1 on success. If a 0 is received, it causes the TS and Gnucap to terminate. `nump` and `numi` are the number of initial poles and number of iterations to use in the algorithm. These are user provided.

Listing 4.2: `vf.h` vector fitting function header. This function is called from within the RC model plugin, with most of its arguments being user provided.

```
bool do_vector_fitting(std::vector<std::complex<double>> &p, std::vector<std::
    complex<double>> &r, double &rem, int nump, int numi, bool vflog)
```

Listing 4.3: At the end of the VF function, the VF model parameters are used to update the RC model instance's corresponding attributes. These are used in computing the parameters of the RC model.

```
  p = poles_guess;
  r = residues;
  rem = remainder;

  return 1;
```

The flag `vflog` denotes whether we want log data to be outputted to the run directory by the simulation. If true, then the files `vf_log.txt` (gives data on each iteration), `results.txt` (gives the final model poles, model residues, and model remainder), `residual_log.txt` (provides useful characterising information such as the residual[2] and system rank), and `pole_guess.txt` (gives the poles of each iteration) are created. Note that these are replaced each time a transient simulation is run. The specific contents of these files will not be detailed here since they are likely to change in the near future, and it is not important for the purposes of the report.

Because much of the VF code is quite rudimentary or dense, it is not be copied here. Instead, we split this section into paragraph headings, each of which give the line ranges of `vf.h` as in appendix A4 which correspond to the functionality or operations discussed therein.

4.2.1 Preliminaries

The first part of the file performs preliminary setup for the algorithm. These steps now follow.

Read in S-parameter data ; lines `102–146`

This is a rudimentary parsing loop which reads in, line-by-line, a data file which is formatted with a header starting with `!`, and fields of frequency sample points, and then the real and imaginary part of the data vector at that point. The three fields should be separated by whitespace. See the data files in the appendices for reference. An attempt is made to make the parser robust to variations, but it is likely not perfect.

The data file should be named `s_param_data.txt`. It is intended that in the near future the RC model will allow specifying a specific filename, but this is not implemented at time of writing. Note that despite the name, this data can be any frequency response, not just S-parameters, as will be seen in the next chapter. An important point is that the code expects

[2]Note that at the time of writing the validity of the residual computation is dubious.

the frequency field to be in Hz (not rad/s), so this should be ensured. Note also that it expects baseband data; the frequency field should start at 0 Hz and monotonically increase up to some `max_freq`.

Transform the frequencies ; lines `157–189`

In order to make computation more numerically robust, the code first transforms the sample frequencies to rad/s, and then normalises them to be between 0 and 1 by dividing by `max_freq`. Since we need frequencies to be complex to use them in the VF formulations, they are then turned into a `std::complex<double>` type, with a real part of 0, and the frequency (rad/s normalised between 0 and 1) as the imaginary part. In the code, this 'final form' of the frequencies is named `freqs_comp` and it is used in the rest of the code wherever the data frequencies are used in the VF algorithm.

To obtain our initial poles, we follow [19] in distributing them evenly across the frequency range. Since our range is in 0 to 1, we simply divide this into `nump` increments and multiply across by 2π. If `nump` is N, then the resulting N-vector, `freq_range`, with kth element `x`, is used to construct the poles as $p_k = -0.01$ `x` $+j$ `x`. These poles are called `poles_guess` in the code, and are updated at each iteration with the improved poles calculated from the σ zeroes.

4.2.2 Iterative Algorithm

The body of `vf.h` consists of formulating the required matrices and calling the LAPACKE (or CBLAS) functions to do the required calculations. Within the iteration loop, we first do some preliminary declarations. Afterwards, the actual VF algorithm itself is done, which has been split into eight steps, (i) through (viii), in order to provide some structure to the code. We specify these in the paragraph headings that follow.

Declarations and Definitions; lines `305–402`

Because the actual size of `poles_guess` varies over iterations, the dimensions of the various matrices in VF also vary. Indeed, the input data will of course not always be the same size. Hence, to make the code robust we start off each iteration by defining standard sizes for each of our matrices, and other required parameters, in terms of some quantities that remain constant for that iteration. Namely, they are defined in terms of

`num_freqs`	the number of frequency points in `s_param_data.txt`,
`num_rp`	the size of `real_poles`, which stores the real (zero imaginary coefficient) poles of `poles_guess` at that iteration,
`num_cp`	the size of `comp_poles`, which stores the complex (non-zero imaginary coefficient) poles of `poles_guess` at that iteration,
`num_sp`	the number of data points in `s_param_data.txt` (note this is the same as `num_freqs`), and
`num_poles`	the size of `poles_guess` at that iteration (note this can vary over iterations, and so is different from `nump`, which remains constant).

The rationale for splitting up the real and complex poles is ease of computation; quantities calculated from real poles are easier to deal with from a programming perspective. Furthermore, the VF algorithm itself makes a distinction between real and complex poles in how matrices are constructed, as described in section 2.3. Splitting calculations in this way is hence more intuitive.

Required matrices for the iteration are dynamically allocated (using the C++ `new` operator). These arrays are deleted at the end of each iteration. Though this is arguably inefficient, doing it this way makes development much simpler. The effect is that we start each iteration with 'fresh' sets of data and therefore avoid errors. Indeed, the use of dynamic arrays themselves instead of C++ containers is primarily done to make debugging easier, since it is straightforward to examine what exactly each of these arrays are doing. Use of 'black box' containers like `std::vectors` would be preferable from a robustness and programmability point of view, but may prove difficult to debug in this context.

(i), (ii), (iii) : Build Linear System ; lines `425–603`

This part of the code corresponds to **STEP ONE** and **STEP TWO** of section 2.3.2. We begin by creating the system matrix, $\bar{\mathbf{A}}$, called `a_matrix` in the code, by filling it in a piece-wise manner. If we abstract the PFE terms of (2.14) by the form of the pole used, (`real` for `real_poles`, and `comp1`, `comp2` for `comp_poles` since each has two associated PFE terms (see (2.13)), then we can generalise each row of `a_matrix` as consisting of:

$$\left[\begin{array}{ccccccc} \leftarrow \texttt{num\_rp} \rightarrow & \leftarrow \texttt{num\_cp} \rightarrow & \leftarrow \texttt{num\_cp} \rightarrow & \leftarrow \texttt{1} \rightarrow & \leftarrow \texttt{num\_rp} \rightarrow & \leftarrow \texttt{num\_cp} \rightarrow & \leftarrow \texttt{num\_cp} \rightarrow \\ \{\,\texttt{real}\,\} & \{\,\texttt{comp1}\,\} & \{\,\texttt{comp2}\,\} & \texttt{1} & \{\,\texttt{real}\,\} & \{\,\texttt{comp1}\,\} & \{\,\texttt{comp2}\,\} \end{array}\right]$$

That is, there are in general 2 * (`num_rp` + 2 * `num_cp`) + 1 columns in `a_matrix`. Such generalisations are enabled by defining standard dimensions. They make constructing the VF matrices in C++ straightforward since we can put bounds on the indexing and assignment operations.

Similar approaches are hence used for the solution matrix **b**, named `b_matrix`. Note that `b_matrix` is constructed as being split into real and imaginary coefficients (as per (2.17b)). We similarly transform `a_matrix` into `a_matrix_real` by doubling the number of rows, and splitting the original `a_matrix` into its real parts (which are placed in the upper `num_freqs` rows) and imaginary parts (which are placed in the bottom `num_freqs` rows).

(iv) : Solve Linear System ; lines `611–637`

This part corresponds to **STEP THREE** of section 2.3.2. The `dgelss` function writes over `b_matrix` with the solution vector **x**, which is placed in the first `a_matrix_real_c` (the number of columns of `a_matrix_real`) elements. We can index into these to obtain the auxiliary residues, auxiliary remainder, and σ residues for that iteration as desired (though we do not actually do this in the code).

(v) : Extract Zeroes ; lines `642–781`

This part corresponds to **STEP FOUR** of section 2.3.2. The iteration for-loop is in fact one iteration extra; i.e, it loops `numi+1` times. We do this because the first `numi` loops perform extraction of the σ zeroes and assigns them into `poles_guess` for the next iteration, but the final `numi+1` iteration does not. Instead, the `poles_guess` from iteration `numi` is taken as the the model poles, and the model residues and model remainder are calculated from these.

We have a separate **H** matrix for the `real_poles`, `H_real`. It is calculated as `Az_real` − `bz_real` ∗ `c_real`, where the matrix multiplication is performed using `dgemm`. Similarly, for `comp_poles` we construct `H_comp`, being `Az_comp` − `bz_comp` ∗ `c_comp`.

Note that in the provided code we use column-major ordering in the `dgemm` function[3], but

[3]The reason for this stems from confusion around the documentation for this function, which seems to assume column-major ordering. This of course makes sense since the original BLAS routines require column-

the CBLAS interfaces should work just fine with row-major ordering, though this has not been well tested; we merely know that it works as given in appendix A4.

In any case, because `vf.h` uses row-major indexing throughout, this means that when constructing the **H** matrices that we must be aware that the output of the `dgemm` function will be the transpose of what we want (i.e., elements will be stored column-wise not row-wise). Instead of transposing it again, we simply swap the loop indices when accessing its values[4].

The eigenvalues of the **H** matrices are obtained using `dgeev`. These eigenvalues are our zeroes, and we concatenate the zeroes corresponding to the real and complex poles into a single vector, `all_zeroes`. This vector remains of the same size throughout the algorithm. It is of length $2 *$ `nump`, since all of our initial poles are complex poles and hence have a corresponding conjugate.

This is a rather important point to note. These zeroes represent all of the underlying poles of our system. However, in our formulation of the VF algorithm we decide to only keep the poles that exist in the positive half of the imaginary axis (i.e., the VF model will only exist in positive frequencies). Hence, even though the size of `all_zeroes` remains the same throughout, the actual number of auxiliary poles, the size of `poles_guess`, *will* vary. The final number of model poles will therefore likely be different to the user-specified `nump`.

(vii), (viii) : Calculate Model Parameters and return to RC Model ; lines 784–886

After we're done all of the iterations, having hopefully obtained a converged set of model poles stored in `poles_guess`, we perform a last least-squares on the system (2.24). This part corresponds to **STEP SIX** of section 2.3.2.

The solution $\mathbf{x}_\mathrm{f}$ of this system stores our model residues and model remainder, which we obtain by indexing into the appropriate elements. At the end of the function, the model parameters will be stored in `poles_guess`, `residues`, and `remainder`, which are passed by reference back to `rc_model.cc` as per listing 4.3.

4.3 Recursive Convolution Model

In this section the functionality of the `rc_model.cc` plugin is presented as it is given in appendix A5.

The model extends the `STORAGE` base class, since this is the natural choice for a device that needs to maintain some concept of history. Additionally, this class has an additional `FPOLY` object, `_i[0]`, which we use to store the $i_c[n]$ expression.

We name the device class `RC_MODEL` and define the specifier as `p` or `rcm`, which is used to instantiate the model into a netlist. Referring back to the formulations in sections 2.4 and 2.5, there are some important aspects of the model that should be mentioned.

(1) The time step delta, Δ, in general varies and so we must compute α_k, λ_k, μ_k, ν_k, and consequently $h[n]$, $i_c[n]$, and G_c at each time step in order to remain robust.

(2) Looking at (2.39), we can see that the expression $K + \sum_{k=1}^{N} r_k\lambda_k$ repeats multiple times, and so it would be efficient to compute this separately. We call this expression `factor` in the code.

major, though row-major can be indirectly used by specifying the matrices as transpose. See section 3.3.2 for an explanation on this. In any case, we are not using BLAS directly, so this is not of much interest to us.

[4] i.e., we do `bc_real[j * bc_real_c + k]` instead of `bc_real[k * bc_real_c + j]` (and the same for `bc_comp`).

(3) The reference impedance Z_{ref}, named `Zref` in the code, is a constant and so we choose to take it in as a the `value()` parameter when instantiating the model. That is, for example, typing `p1 1 0 50` in Gnucap instantiates the `rc_model.so` between nodes 1 and 0, with `Zref` =50.

(4) Referring to (2.35), it is evident that at each time step n, we need to have access to the two previous values of $a[n]$, and the previous value of $d_k[n]$. To do this in the code, we maintain two class attributes, one of type `std::vector<std::complex>>`, `dk_store`, for storing $d_k[n-1]$, and one 2-element array, `a_store`, in which we store $a[n-1]$ as `a_store[0]` and $a[n-2]$ as `a_store[1]`.

We need these values at time step n but obviously can only compute them at previous time steps. Hence, even though the companion model relations don't explicitly use $a[n]$ or $d_k[n]$ at any point, we still need to compute them and then push them back into `dk_store` and `a_store` for use in subsequent time steps. This is done in the `tr_accept()` function (see Fig. 3.1).

(5) Recall the coefficients defined in (2.32) and (2.33). In the code, the second order coefficients are implemented as the variables `lambda_k`, `mu_k`, and `nu_k`, and the first order as `lamdba_k_1` and `mu_k_1`. We don't need a variable for $\nu_{k,1}$ since it is 0. The variable for α_k is `alpha_k`.

(6) We use the variable `time_step` as an integer counter to keep track of the current time step in the TS. Note that the counter starts from 0, and so time step $n = 1$ will correspond to `time_step` $= 0$.

4.3.1 Polynomial Functions

The plugin uses three polynomials to keep track of the RC constituent relations. The $v[n]$ and $i[n]$ relations, from (2.38), are `FPOLY` (`_y[0]`) and `CPOLY` (`_m0`) respectively, and are computed in `do_tr()` at each time step. We use the `FPOLY` `_i[0]` for the $i_c[n]$ term, but this is a stylistic choice mostly since it is not actually used in the MNA matrix directly.

We have the following KVL and KCL for the S-parameter block:

$$\begin{array}{lll}
v[n] = \frac{1}{G_c} i_c[n] + \frac{1}{G_c} i[n] & \rightarrow \texttt{\_y[0].f0} & = \frac{1}{G_c} i[n] \\
 & \rightarrow \texttt{\_y[0].f1} & = \frac{1}{G_c} \\
 & \rightarrow \texttt{\_y[0].x} & = i[n] = G_c \cdot \texttt{tr\_involts\_limited()} - i_c[n] \\
i[n] = -i_c[n] + G_c v[n] & \rightarrow \texttt{\_m0.c0} & = -i_c[n] \\
 & \rightarrow \texttt{\_m0.c1} & = G_c \\
 & \rightarrow \texttt{\_m0.x} & = v[n] = \texttt{tr\_involts\_limited()}
\end{array}$$

Recall that `tr_involts_limited()` returns the voltage across the device at the current time step.

4.3.2 Device Parameters

Gnucap allows us to specify multiple parameters beyond just `value()`. At the time of writing there are currently four extra parameters that can be specified. These are:

`nump`	The number of initial poles for our fit, which is passed to `do_vector_fitting()`. It has a default value of 35 if not specified.
`numi`	The number of iterations to use in the VF algorithm., which is passed to `do_vector_fitting()`. It has a default value of 450 if not specified.

`vflog`	A Boolean flag for if we want to print logs during the VF run, as described in section 4.2. By default, it is 0.
`rclog`	A similar Boolean flag as `vflog`, but for the Gnucap TS itself. If it is 1, a file `tr_write.txt` is created in the run directory, which is written with the values of various variables during the TS. It is primarily for debugging and so may not be of interest to most users.

Much like the TS functions, we also override functions to define custom parameters for our device plugin. We will not state the functions nor their details here. Refer to [32]. It is intended that additional functionality be added in the future via extra parameters. For example, being able to customise the input data format in some way would be quite useful.

4.3.3 Transient Function Overrides

The RC companion model is implemented by overriding five specific transient analysis functions, which we now describe as is relevant. Like in section 4.2, we will state a range of line numbers of the code in appendix A5, rather than copying the code here.

`precalc_first()` ; lines `267-280`

This function is rather short and simple. It first evaluates the value of the parameters in order to assign or override the default values, and then calls VF via `do_vector_fitting()`. We use the C++ `assert()` function, in line with the rest of the Gnucap source code, to force a termination in the case that VF fails.

The fact that we run VF in `precalc_first()` means that (1) it runs every time we perform a Gnucap command that involves the device, even if it is not a TS, and (2) that in such cases there is a brief pause after executing the command while VF runs. The time this takes of course vary on the system size and the values of `nump` and `numi`. For the default case we provide some preliminary execution time statistics in chapter 5.

`tr_begin()` ; lines `283-325`

Another short function, here is where we open `tr_write.txt` if applicable, and assign any initial and constant values as appropriate. Here is where we also set `time_step` to 0 to mark the beginning of the TS, i.e., the first time step is indexed by `time_step` = 0, the second by `time_step` = 1, and so on.

An important note is that we initialise $i_c[n]$ (implemented as the `i_c` attribute) to zero, and set all the history terms `h_n` (which implements $h[n]$), `dk_store`, and `a_store` to 0.

This is in fact a critical step since we are considering the input signal as being causal; i.e., it is a value of 0 for all time steps less than `time_step` 0. The reasoning is that it seems that in order for (2.35) to be valid, that we must restrict ourselves to step-like inputs, whose values $a[n]$ are 0 for $n < 0$. This is typically the case in reality since we invariably need to 'turn-on' a circuit at some point, so it will go from 0 input to some known input at $n = 1$. In this case, we assume it is also 0 at `time_step` 0, but it could of course also be any other known value.

`do_tr()` ; lines `455-533`

Ordinarily, `do_tr()` would just calculate the constitutive relations and stamp them into the MNA matrix. That is, it would only do what is shown on lines `519-532`, and leave the actual term updating for that `time_step` to `tr_advance()`. However, it seems that the flow presented in Fig. 3.1 is not strictly correct, because `tr_advance()` is not in fact executed for `time_step` 0. This means that we must do the updating inside `do_tr()` `if (time_step == 0)`. This turns out to be fine despite the fact that `do_tr()` is repeated multiple times per time step, since at `time_step == 0`, we have not actually updated the history terms nor `Gc` and `i_c` yet and they remain as 0.

In general, this updating consists of calculating the coefficients `alpha_k`, `lambda_k`, `mu_k`, and `nu_k` (or their first order counterparts) using the Δ between this and the previous time steps. We use these to get values for `factor`, `Gc`, `h_n`, and subsequently `i_c`.

The initial updating done in `do_tr()` is a bit different:

(1) Since this is the first time step, we do not actually have Δ, which is ordinarily accessible via the Gnucap `_dt` global variable. We hence use a custom `delta` variable instead[5].

(2) `dk_store` and `a_store` are 0, and so the history term `h_n` is 0, as per (2.35). The `i_c` term will also be 0 as per (2.39).

(3) Since it is the first time step, we must use the first order coefficients given in (2.33). Even though the `RC_MODEL` class has attributes for both the first and second order coefficients, here we simply use locally defined variables. The only coefficients we use in `do_tr()` are α_k and λ_k, since the others are all multiplied by 0 at some point.

(4) The only constitutive relation parameter we need to compute is `Gc`, for which we need to compute `factor` using the first order λ_k. The term $\sum_{k=1}^{N} r_k\lambda_k$ is a dot-product, and hence we can compute it by simply summing the N products $r_k\lambda_k$. We do this by looping over the model poles and residues. The resulting value is stored in `res_lambda_sum`.

One last important detail is worth mentioning here. In (2.39), notice that the model residues and poles, as they are obtained from `vf.h`, are complex numbers which occur in conjugate pairs. This means that, for example, the `res_lambda_sum` term describes a conjugate pair term. In order to have $v[n]$ and $i[n]$ be real-valued, we can convert them to real-values by doing `2 * res_lambda_sum.real()`. The primary motivation for this is that the Gnucap MNA solver expects real-values (i.e., `double` types, in general). Furthermore, a complex-valued voltage or current is not that intuitive and is unlikely to interface well with any attached circuitry, whose voltages and currents exists in the real domain.

It is critical that this transformation is only done *after* all prior calculations. That is, we do it on the final `res_lambda_sum`, but *not* the residues `r` and `lambda` values used to compute it!

`tr_advance()` ; lines 329–452

This function does essentially the same updating as that described for `do_tr()` immediately above, but for `time_step` ≥ 1. The only real difference is that the RC coefficients can now be second order and so the calculations are slightly longer. Additionally, `h_n` and `i_c` will be non-zero and so are also computed. To keep things concise, we only note some key differences.

(1) We have access to the actual simulation Δ via `_dt` since we're no longer on the initial time step.

[5] This code is written with the series TL data of appendix A8 in mind, and so we use a `delta` of `0.1e-3`. This is in fact 100 ns from the perspective of the TS test given in appendix A10. See section 5.3 for details on this. The RC cold is therefore inaccurate if we desire to use a separate Δ in our TS, since the erroneous initial coefficient values will likely distort the rest of the simulation. Though it would be relatively straightforward to add the ability to specify an initial Δ of the user's choosing, it is not implemented at time of writing.

(2) For `time_step` $= 1$, we still use first order coefficients, where in this function we call them `lambda_k_1` and `mu_k_1`. We use these to compute `h_n` and hence `i_c` since at this time step we still technically only have two data points, $a[n]$ and $a[n-1]$. However, since the conductance `Gc` is not dependant on the input signal, we can use the second order `lambda_k`, which is accomplished by having a distinct `factor_1` term for use in calculating `i_c`.

    We could also do this for `time_step` 0, but it is technically more correct to have G_c be linear in that case.

(3) For `time_step` $= 2$ and above, we use the second order coefficients for all calculations since from this time step onwards we have access to $a[n]$, $a[n-1]$, and $a[n-2]$. This means that `h_n` will be the complete sum shown in (2.35) (i.e., ν_k is not zero).

Just like for `res_lambda_sum`, we calculate (2.35) as a dot product, which is stored in `hist_sum`. This is also a complex conjugate pair and we convert it to a real value in the same way, by doing `h_n = 2 * hist_sum.real()`. Note, then, that the `h_n` term referred to throughout this section is this real value, which is what we use in calculating `i_c`.

The last part of `tr_advance()` calculates `Gc` and `i_c` for that time step, which are used in assigning the KCL and KVL in `do_tr()`.

`tr_accept()`; lines 536-620

This is where we calculate the history terms and propagate them to the next `time_step`. Note that it appears that the `tr_accept()` function is entered twice per time step, which is undesirable since it will lead to propagating the history terms twice per time step. We use a flag, `accepted`, as an ad-hoc solution to prevent re-calculation of the history terms. It would of course be desirable to instead understand why the function is entered twice, and to design with it rather than around it.

The function begins by calculating $a[n]$ using the discrete-time form of $a(t)$ given in (2.26), which is put into `a_n`. It then calculate $d_k[n]$ as per (2.31) and puts it into `dk_n`. Much like `Gc`, `h_n`, and `i_c`, `dk_n` must also be calculated differently for `time_step` 0 and `time_step` 1.

By referring to (2.31) this function's code is fairly self-explanatory. Its important purpose is to store these values into `dk_store` and `a_store` for use with the subsequent time step (i.e., they are not used in the time step in which they are computed). This is done at the end of the function.

4.4 Access to Work

The code for this project is available as a GitHub repository, which is located at [41]. It is chosen to license it under the GNU GPL in line with the author's personal philosophy, and to be consistent with Gnucap itself. The reader is welcome to use the code in any manner they see fit, as long as it falls under the remit of the GPLv3, which is given on the GitHub under `LICENSE`.

The primary intent of this project is to add useful functionality to Gnucap. It is hoped that the code will be developed further and added to in the future, both by the author and, hopefully, by others who come across it and have the desire to use it. It is therefore expected that the details presented in this report and its appendices will be to some degree outdated (i.e., no longer valid) in the future.

Chapter 5

Results and Conclusions

5.1 Results Methodology

In order to effectively evaluate the correctness of the implementations, there is a need to have some reference against which to compare the results of the implementation. For this, we use a provided MATLAB 'golden model' implementation, which is known to be relatively robust based on previous work by the project supervisor.

This MATLAB code implements, in effect, the same procedures as our C++ code described in chapter 4. That is, it contains (1) a 1-port VF algorithm for the baseband S-parameter case (or any other FD data) discussed in section 2.3 and (2) an implementation of the same RC TD companion model discussed in sections 2.4 and 2.5. It also contains rudimentary TS capabilities via a small MNA system, which uses a programmatically generated sinusoidal source attached to the S-parameter model as the test circuit. We of course use Gnucap's transient simulator, but we use the same test circuit such that comparison with the MATLAB results are possible. This test circuit is described in section 5.3.

Indeed, our implementation in this project is largely informed by this MATLAB implementation. In any case, given that MATLAB provides a very robust and 'user-friendly' manner by which to perform computations with large matrices, it is an ideal reference point for the potentially naive C++ implementation in our Gnucap plugin. This is especially pertinent since we use LAPACK routines, which are not the simplest of functions to debug.

In this chapter, we perform the following analyses:

(1) Evaluate the C++ VF model's correctness against analytic data from a parallel RLC circuit.

(2) Compare the computed model poles, residues, and remainder of the MATLAB and C++ VF models for a series TL circuit and a Butterworth LPF circuit.

(3) Discuss how well the VF models match the original FD data vectors, by showing the computed frequency response for the given port-network parameter.

(4) Investigate how well the MATLAB and C++ (Gnucap) RC model implementations match the expected transient behaviour for a simple circuit consisting of a sinusoidal source attached across the device ports.

(5) Provide preliminary performance characterisation by way of execution time and reporting of the error between the MATLAB and C++ models with respect to the source data.

We conclude the chapter by providing a brief discussion of the project achievements, point out issues, and suggest some of the many potential future works and improvements that can be done on the plugin.

5.2 Vector Fitting Results

5.2.1 RLC Circuit

The VF alogorithm can be applied to any frequency domain data, not just S-parameters. In order to illustrate this, and to demonstrate that the implemented `vf.h` works, we present here the results of running the algorithm on a set of data points, given in appendix A7, which measure the input impedance of the RLC circuit that is shown in Fig. 5.1. This data vector was collected using the Keysight **Advanced Design System** (**ADS**) software. It is provided across ω values from 0 to 50 radians.

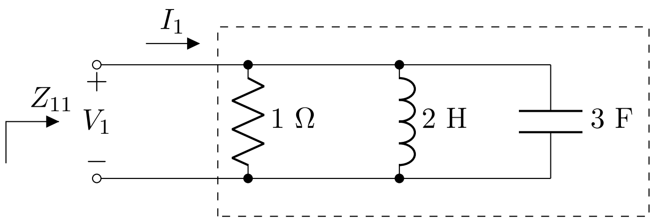

Figure 5.1: Example 1-port network, being an RLC with a known analytic expression for input impedance Z_{11}. Here the '11' essentially denotes the input port of the 1-port network.

Computing the impedance analytically, we get $Z_{11}(s) = \frac{s}{3s^2+s+0.5}$. We can factor the denominator to obtain the poles, and then use the PFE $\frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} = \frac{s}{3s^2+s+0.5}$ to compute the expected residues. The results are given in the middle column of Table 5.1. We run the C++ VF algorithm[1], with `nump=2` and `numi=100`, with the resulting poles, residues, and remainder shown in the right-hand column of Table 5.1. Only one of the poles seem to match analytical computations. This is likely because in `vf.h`, at the end of each iteration, we only keep zeroes with a positive imaginary part. By doing this, we might be causing the VF algorithm to adjust the other poles and the residues. Of course, the remainder term also captures deviations of the `vf.h` model. Nevertheless, if we plot $\tilde{Z}_{11}(j\omega)$, we see that the magnitude response is very close to that indicated by the data, though the phase response does appear to be worse. These results, along with the **mean absolute error** (**MAE**) with respect to the data, are shown in Fig. 5.2. The plots only show up to frequencies of 10 rad/s so as to highlight the RLC resonance.

Table 5.1: C++ VF fit parameters compared against analytic expectations for RLC circuit.

Parameter	$Z_{11}(j\omega)$	$\tilde{Z}_{11}(j\omega)$
poles	$-\frac{1}{6} + \frac{\sqrt{5}}{6}i$, $-\frac{1}{6} - \frac{\sqrt{5}}{6}i$	$-8.577 \times 10^{-5} + 0.351i$, $-0.167 + 0.373i$
residues	$\frac{1}{2} - \frac{\sqrt{5}}{10}i$, $\frac{1}{2} + \frac{\sqrt{5}}{10}i$	$-6.465 \times 10^{-5} - 5.641 \times 10^{-5}i$, $0.167 + 0.075i$
remainder	—	7.874×10^{-10}

5.2.2 Transmission Line

Throughout development of the code, input S-parameter data from the circuit shown in Fig. 5.3, a series connection of lossless TLs terminated by a load resistor, is used as a basis for evaluating correctness. This evaluation is done with reference to the MATLAB golden model, using the same data vector. The data was collected from ADS simulation of the circuit.

The data vector $\bar{S}_{11}(j\omega)$ is given in appendix A8. This system is a good example of how VF can be used to obtain a TD expression for a system that is otherwise awkward to deal with in

[1]Note that since the data vector has frequencies in terms of $2\pi f$, they were first divided by 2π for use with `vf.h`, which expects frequencies in terms of Hz. This is what is shown in appendix A7.

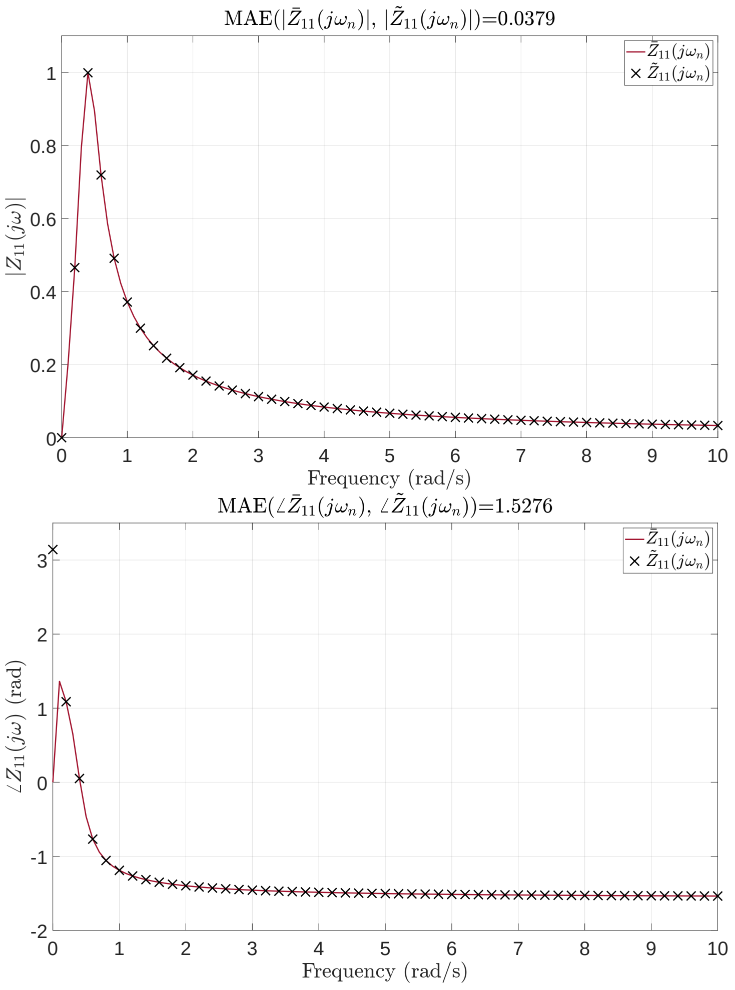

Figure 5.2: Magnitude (top) and phase (bottom) response of the C++ VF model, $\tilde{Z}_{11}(j\omega)$, of a known RLC circuit's input impedance, compared against measured data, $\bar{Z}_{11}(j\omega)$. On top of the plots is the MAE between them.

the TD. However, note that TLs in fact have infinite poles. One way to understand this is that TL analysis consists of splitting the TL into a finite number of finite length segments consisting of a series R and L, and a shunt G and C [2]. In reality however, the full behaviour of the TL is only captured if there are infinite such sections of infinitesimal length, but this is generally approximated in order to enable simulation.

In our VF run we assume that the system is well approximated by 35 poles, which we assume exist in the range of 0 GHz to 100 GHz. If this assumption is too liberal, then we expect that the associated residues for the 'extra' poles will be close to zero.

Fig. 5.4 shows plots of the poles and residues of the MATLAB and C++ VF solutions for this system, where the algorithm is run with `numi` = 450.

We see that the critical points differ quite a lot, though curiously for the lower frequencies the poles seem to match almost exactly. It is not clear why this is the case, though it's possible that this indicates that the higher frequency poles are not significant. Indeed, we see a cluster of residues at zero, indicating that we have an excess of poles. Regardless, even though the models are distinct, both the C++ and the MATLAB VF solutions produce very good representations of the underlying data, as is shown in Fig. 5.5. Also shown is the MAE of each model with respect to the data, $\bar{S}_{11}(j\omega)$.

Both the C++ and MATLAB models produce identical fits down to at least the fourth decimal place, despite having quite different model parameters. This is a good illustration of

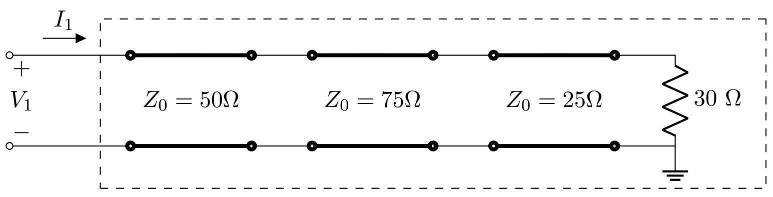

Figure 5.3: 1-port lossless TL network for VF evaluation. We wish to model the input S-parameter of this circuit.

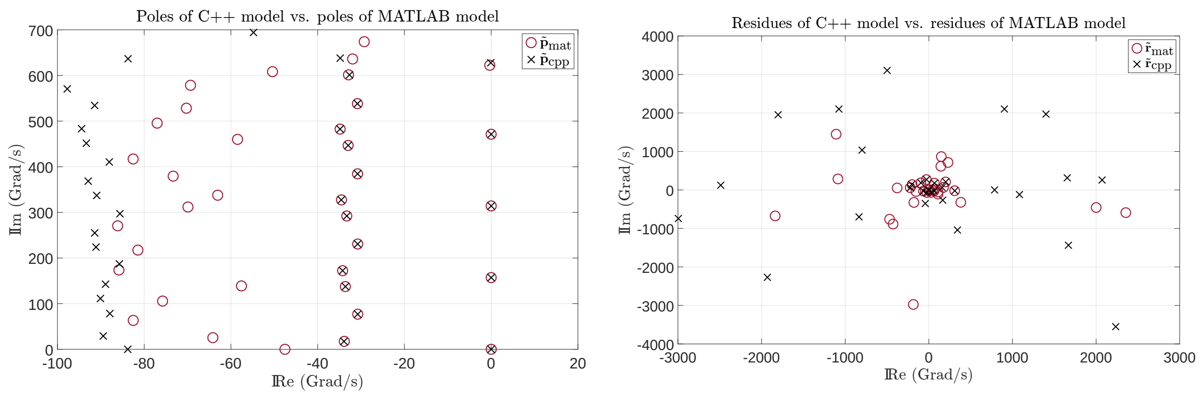

Figure 5.4: Comparison of the poles (left) and residues (right) of the C++ and MATLAB VF models of the input S-parameter for a series lossless TL circuit. Note these are in 'Grad/s', since the data samples are in terms of GHz.

the discussion in section 3.3.2. That is, because the VF algorithm in general produces a rank deficient system matrix, it means that there is no unique solution, i.e., no unique VF model for the data. Hence, despite having quite different transfer functions, the actual result of the two models is pretty much the same.

5.2.3 Butterworth Filter

The final circuit we test the VF implementation with is a Butterworth **low-pass filter** (**LPF**), specified to have a pass-band frequency of 1 GHz, being the 3 dB frequency, and a stop-band frequency of 1.2 GHz, at which the gain should be 20 dB. The filter is also terminated into a 50 Ω impedance. Such a circuit is illustrated in Fig. 5.6.

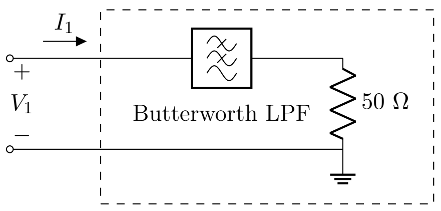

Figure 5.6: 1-port Butterworth LPF network for VF evaluation. We wish to model the input S-parameter of this circuit.

The 1-port (i.e., input) S-parameter data vector of such a filter was collected in ADS, as before, and is given in appendix A9. Note that ADS instantiates the filter as a 'black box' component, and so we do not readily know what its order is, nor its analytic transfer function. VF therefore, is useful for such a case. The S-parameter data covers 0 MHz to 2 GHz, in deltas of 10 MHz.

The poles and residues of the MATLAB and C++ VF solutions are given in Fig. 5.7. The algorithm is run for a `numi` of 450, and `nump` of 35.

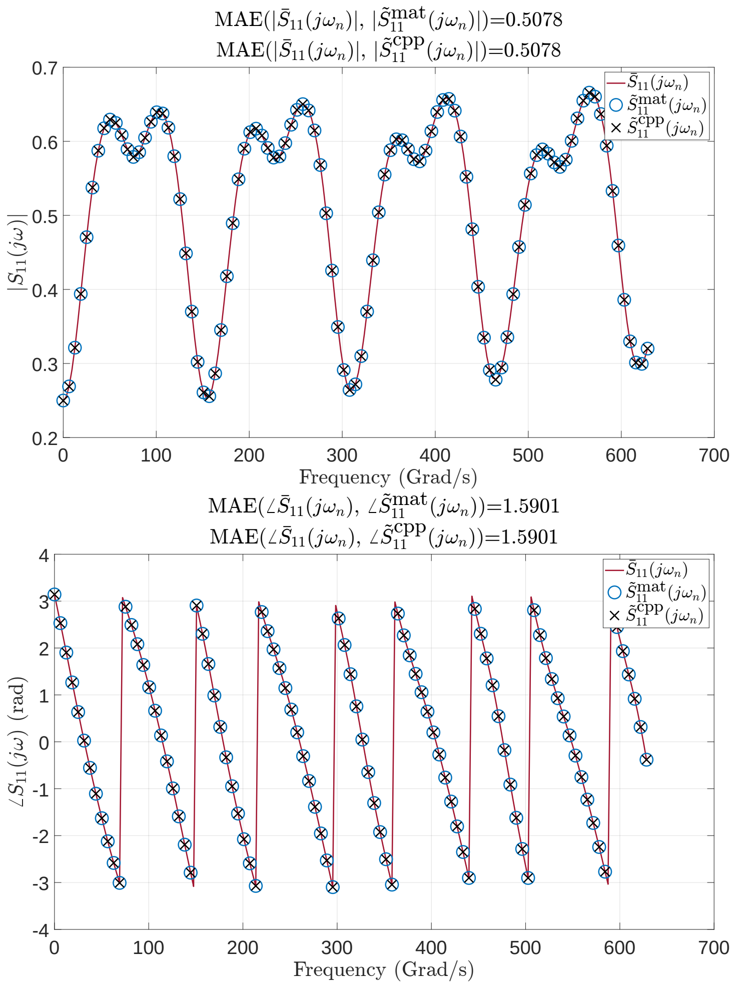

Figure 5.5: Magnitude (top) and phase (bottom) response of the C++ and MATLAB VF models, $\tilde{S}_{11}^{\text{mat}}(j\omega)$ and $\tilde{S}_{11}^{\text{cpp}}(j\omega)$, of a lossless TL circuit's input S-parameter, compared against measured data, $\bar{S}_{11}(j\omega)$. On top of the plots is the MAE between each.

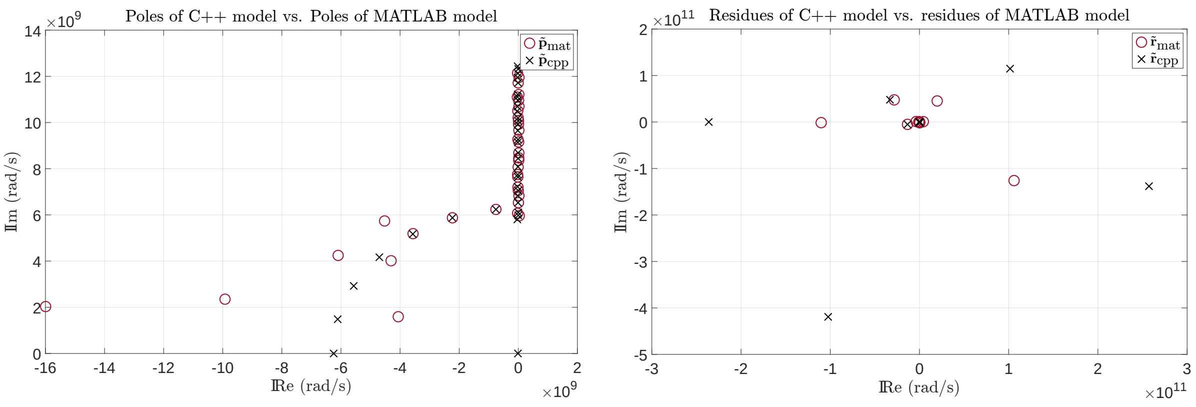

Figure 5.7: Comparison of the poles (left) and residues (right) of the C++ and MATLAB VF models of the input S-parameter for a Butterworth LPF circuit.

We see that 35 poles is far too great of a fit order for this system, with most of the poles and residues being approximately zero as a results. We could hence re-run the fit but with lower order. Interestingly, however, the C++ solution appears to much better capture the expected unit-circle pole distribution of a Butterworth filter. We can try to understand this by again considering the issue of rank deficiency in the system matrix. The MATLAB implementation (`x=A\b`) uses QR factorisation to compute the least squares solution, whereas the `dgelss` function used in the C++ implementation uses the the pseudoinverse. The result is that the C++ implementation is *expected* to find the minimum-norm of the possible solutions.

Again we see that even though the MATLAB and C++ models differ in parameters, that they nevertheless both produce good fits to the data. The models themselves are plotted in Fig. 5.8, against the data samples. We also show the MAE with respect to the data.

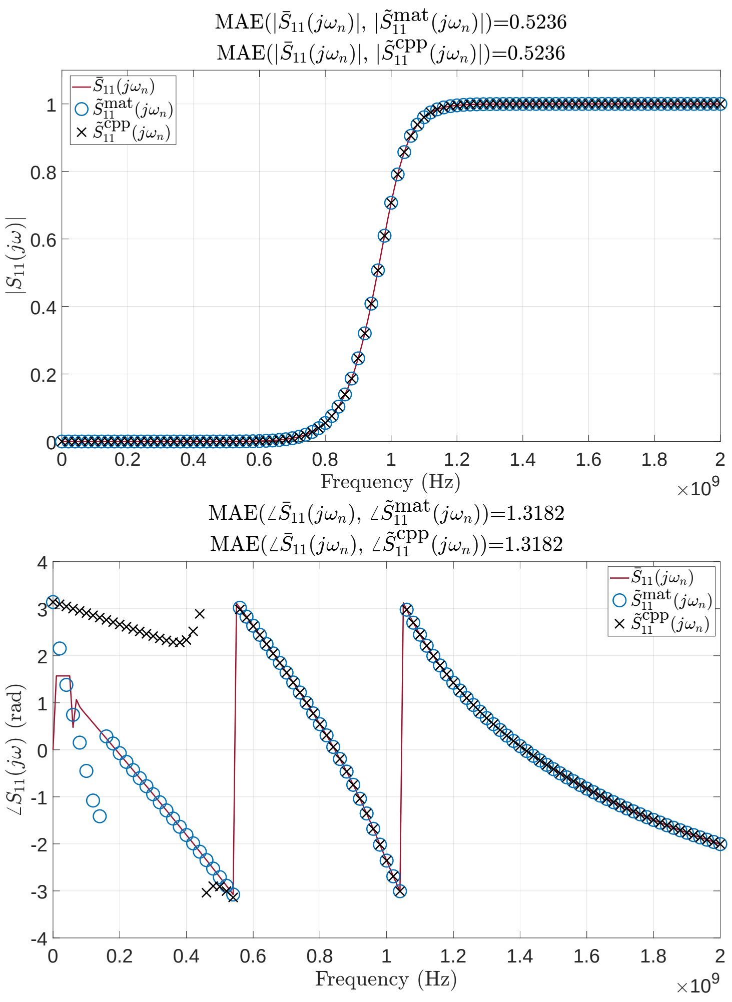

Figure 5.8: Magnitude (top) and phase (bottom) response of the C++ and MATLAB VF models, $\tilde{S}_{11}^{\mathrm{mat}}(j\omega)$ and $\tilde{S}_{11}^{\mathrm{cpp}}(j\omega)$, of a Butterworth LPF circuit's input S-parameter, compared against measured data, $\bar{S}_{11}(j\omega)$. On top of the plots is the MAE between each.

Despite the points about the pseudoinverse being potentially more accurate, we, in all the results of this chapter, see remarkably little difference between the MATLAB and C++ implementations, as is evident from the MAE values; it is possible that the benefits of the minimum-norm solution is only significant at higher decimal places.

Having said that, we see that the phase response of both VF models, but especially the C++ model, deviate quite significantly from the data for the lower frequencies. It is unclear why this occurs, but it could be because the fit order is too high, or maybe the 'noisy' nature of the low frequency samples is to blame. Another probable explanation is that the very small magnitude at these frequencies (which is on the order of 10^{-5} or so) means that the corresponding phase is not well defined.

It should be emphasised that we are modelling the S-parameter of the filter, *not* its transfer function. Hence, the response curves of Fig. 5.8 show the ratio of reflected to incident power waves on the port. We therefore see what is to be expected for a LPF; the signal is passed through up until the passband edge, at which point it begins to be blocked and hence reflected back to the input. That is, $S_{11}(jw)$ goes from approximately 0 in the passband, to 1 in the stopband. In terms of incident and reflected signals, the S-parameter block behaves like a high-pass filter.

5.2.4 Simulating Beyond Trained Frequencies

As a final analysis of the VF implementation, Fig. 5.9 shows the results of using the C++ and MATLAB VF models for the series TL system described in section 5.2.2, but for frequencies up to 250 GHz, even though the models are only trained from 0 Hz to 100 GHz.

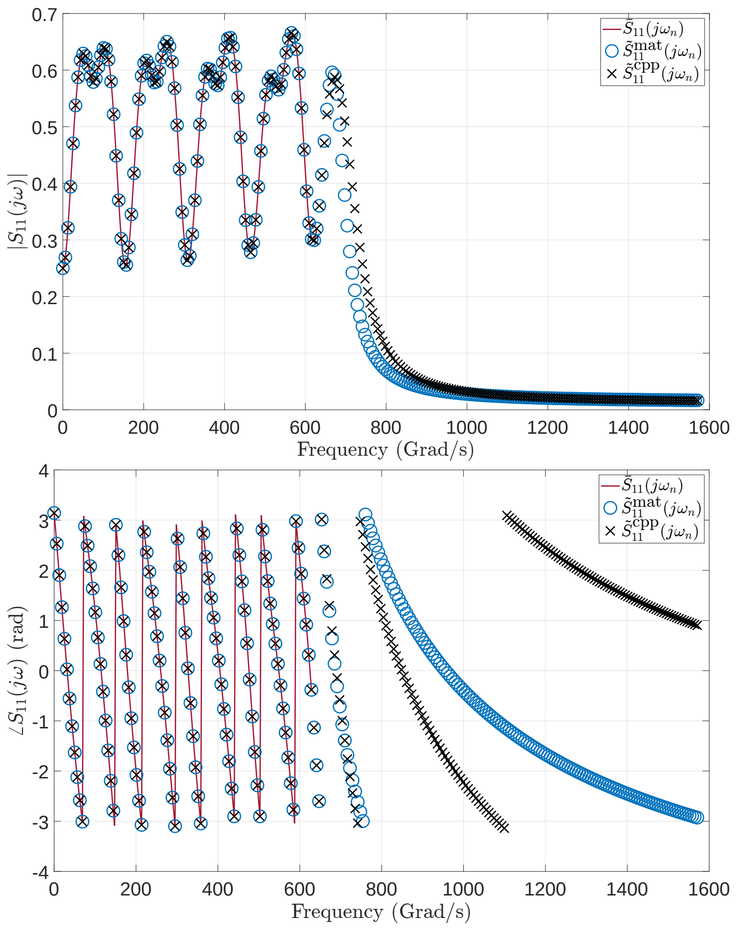

Figure 5.9: Magnitude (top) and phase (bottom) response of the C++ and MATLAB VF models, $\tilde{S}^{\mathrm{mat}}_{11}(j\omega)$ and $\tilde{S}^{\mathrm{cpp}}_{11}(j\omega)$, of the input S-parameter of the same TL system as section 5.2.2, compared against the measured data, $\bar{S}_{11}(j\omega)$. Here we evaluate the models beyond the trained frequencies to examine the behaviour.

The results are broadly as expected, and emphasises that these schemes are only applicable within the measured bandwidth. Following from the above discussions, however, we might expect the C++ implementation to be 'better' since it uses the minimum-norm solution. It is unclear if this is the case from these results. It is notable that in the phase response, the models seem to break down more dramatically as we move to higher frequencies, than it does in the magnitude response which mostly just goes flat and moves towards zero. The C++ implementation appears to be possibly more consistent in this regard. But again, this is just observation.

This analysis is motivated by the fact that our input signal is, in reality, not a perfect sinusoid. Since it is technically 'cut off' at the beginning and the end of the simulation, it is in fact a piece-wise function with very abrupt transitions at the boundaries. This might not be much of an issue with simple inputs like this, but for more complicated, realistic signals like an OFDM signal, we might expect significant frequency components at the higher frequencies, beyond the scope of our data. It is therefore important to test the robustness of the implementation to such behaviour, but we leave this as future work.

5.3 Recursive Convolution Results

The RC model plugin is tested using the Gnucap batch script given in appendix A10, `test.ckt`. This script loads in the RC model, and creates the netlist for the circuit shown in Fig. 5.10, where `p1` refers to an instance of the `rc.so` device model. The RC model is attached in series with a sinusoidal source of frequency 2 GHz, `Vs`, and a source impedance of 50 Ω, `r1`. In these results, the `p1` model uses the TL circuit data given in appendix A8, and runs the VF algorithm for 450 iterations with 35 starting poles. Also note that the reference impedance Z_{ref} is set to 50 Ω. Due to time constraints, we only present the transient results for this series TL circuit here. In any case, it is sufficient to illustrate that the Gnucap plugin works. The reader is invited to test the plugin on other circuits and to explore the code on the project GitHub themselves, if interested.

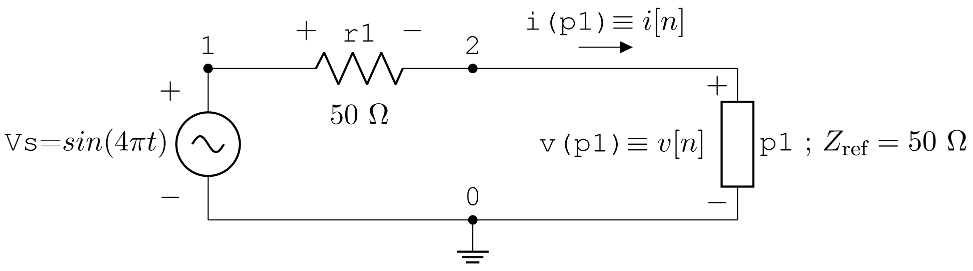

Figure 5.10: The Gnucap circuit used to test the completed RC model plugin. We wish to obtain $v[n]$ and $i[n]$ from a provided set of S-parameter data, as per the project objective.

Some important points about this simulation should be noted. Firstly, because the TL data is in terms of GHz, the TD port voltage and current, $v[n]$ and $i[n]$, which are produced by the RC model are referenced to nanoseconds. This is hence our reference unit, with the transient simulation being run from 0ns to 5 ns, with a Δ of 0.1 ps. This is why the `Vs` source has a frequency set to 2 Hz (4π rad/s); in reality this represents 2 GHz.

Another important point is that we use the `dtmin` and `dtratio` options in the `tr` command to *enforce* a Δ of 0.1 ps in the TS. This is firstly done in order to ensure a consistent Δ in the solution such that comparison to both the MATLAB transient solution, and some ADS transient data collected from the same circuit, can be done.

More nefariously, however, it turns out that the current implementation only works reliably *if* we enforce a consistent Δ in this way. Otherwise the Gnucap solver chooses a very small Δ (on the order of zepto-seconds!) and the simulation takes an unreasonably long time. It is unclear why this occurs exactly, but it is very likely related to the fact that the RC model is tightly coupled to Δ via the coefficients. This could investigated further in the future, but as of this project, enforcing Δ is required.

Fig. 5.11 plots the port voltage $v[n]$ and port current $i[n]$ of the TL circuit (`p1` in Fig. 5.10) computed by the Gnucap transient solver. The voltage $v[n]$ is plotted against both the MATLAB and ADS transient data. For the current $i[n]$, it is only plotted against the MATLAB solution since no current data from ADS is available for this circuit.

For the MATLAB implementation, these transient results are obtained via a small MNA system which uses pre-computed data points and a fixed MNA matrix for the circuit of Fig. 5.10.

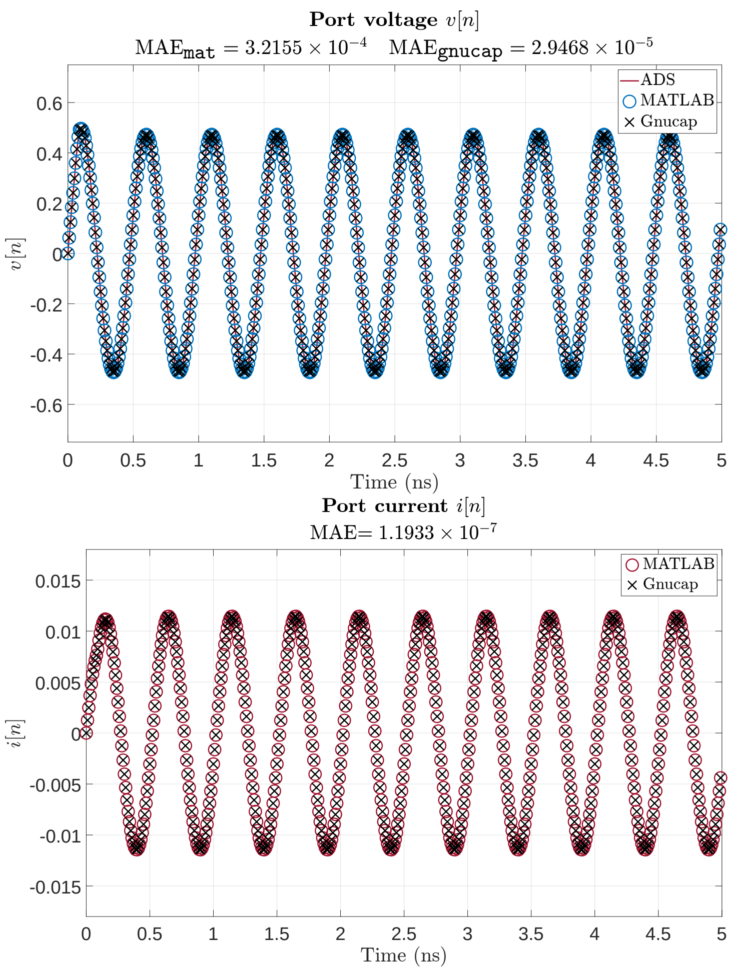

Figure 5.11: TS port voltage (top) and current (bottom) results of the lossless TL circuit, showing results for the MATLAB model, Gnucap plugin, and ADS transient data. In the case of $i[n]$, only the MATLAB and Gnucap is compared. We see that the Gnucap plugin works successfully.

We show the MAE in both plots. The C++ (Gnucap) implementation seems to produce more accurate results than the MATLAB, which of course is no surprise given that Gnucap is an actual circuit simulator.
In any case, the main takeaway is that the Gnucap plugin does precisely what we want it to do; we provide it with an S-parameter data file representing some 1-port network, and it produces the TD port voltage and current of this network.

Fig. 5.12 also illustrates the ability of the Gnucap model to capture transience, which is something not captured by SS methods such as HB. Indeed, this is proof that the plugin *is* actually obtaining the correct TD signals for the port network. This SS data is obtained from a simple phasor analysis of the TL circuit in MATLAB.

Though not strictly part of the plugin testing, one last interesting point of comparison is to compare the MATLAB VF model with the C++ VF model in terms of the time it takes the algorithm to complete. Table 5.2 shows this, where the mean time-taken is obtained over five runs. We collect the C++ data from the isolated form of the VF implementation, i.e., without using Gnucap. Nevertheless, it gives a good insight into the length of the initial latency seen when using the Gnucap plugin.

These results are of course very rough, since they will vary with resource usage on the given platform running the models. Nevertheless, they provide an interesting if not somewhat unexpected insight; the C++ implementation of VF is not much faster than the MATLAB. Both

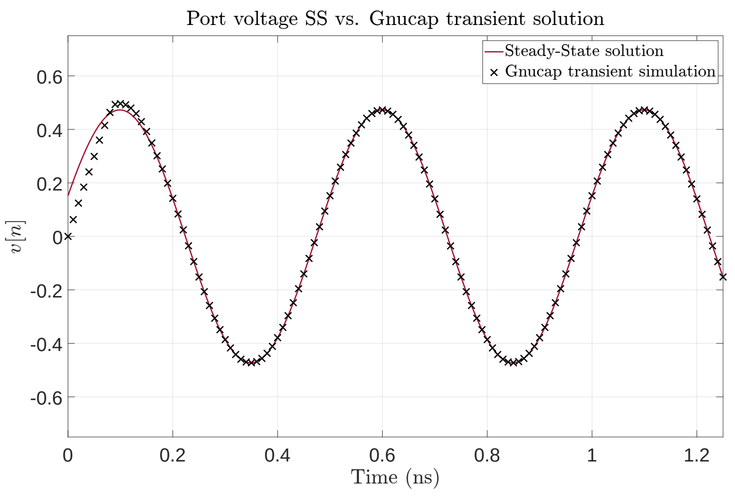

Figure 5.12: Comparing the TS port voltage from Gnucap against the SS value of the port voltage. We see that the Gnucap plugin is effectively able to capture transient effects.

Table 5.2: Comparison of the execution time of the VF algorithm in the MATLAB and C++ implementations, for the series TL circuit. We see a marginal advantage in the C++ case. Note that the C++ data is collected independant of Gnucap.

Model	**Run Time (s)**					**Mean (s)**
	1	**2**	**3**	**4**	**5**	
MATLAB	6.38	7.25	7.80	7.81	7.75	7.34
Gnucap	6.34	5.78	5.72	5.85	5.83	5.96

are, however, relatively naive implementations and could be improved upon. Indeed, the circuit we are testing is quite simple. Testing over more circuits (i.e., different sets of FD data) and across `numi` and `nump` would provide better insights.

There is no use in comparing the speed of the transient solver since the MATLAB model uses pre-computed values, and the Gnucap solver computes them in real-time. Though, a possible future point of investigation could be to more comprehensively compare Gnucap to commercial EDA software, which regrettably is not possible in this project due to time constraints.

5.4 Discussion and Future Work

Though appearing like a relaying of documentation, chapters 2 and 3 in fact represent the majority of the time spent on this project. Given the relatively unusual nature of the project, as a programming rather than research project, it was sought to become quite familiar with the necessary background first. We believe that this was satisfactorily achieved, and the detailed relaying in these chapters is evidence of that.

Indeed, we hope that the plugin example in section 3.2.2.2, and in particular the discussion on Gnucap's polynomial system therein, helps any reader on the topic, which we found to be some of the more difficult concepts to understand during the course of this project. Indeed, attaining an understanding and a verification of knowledge via this simple plugin was critical in being able to confidently develop the much more nuanced `rc_model.so` plugin.

From the project work point of view, the primary conclusion is that the plugin works according to our original project goal as stated in section 1.2; we can instantiate the plugin in Gnucap and give it a FD data file, and it will reliably produce the port voltage and current in the TD. We originally intended to implement the plugin for a 1-port network and this was achieved. We may therefore state that the project was a success.

As of writing, the implementation is still very much so in an elementary state, and has been primarily a proof-of-concept work. However, by its very nature there is significant potential for improvements and expansions beyond what has been thus far implemented. We note some below which are in line with the original motivations and discussions around the project.

(1) It would be beneficial to make the plugin more customisable, mainly by way of adding more parameters to the device. For instance, it would be ideal to be able to specify the name of the data file to use. Being able to specify an initial time step delta is also an important addition. The data file parser could be made more robust, and we should be explicit on the format of the input data that the plugin accepts.

(2) We hope to make the plugin more user-friendly in the future. This would primarily take the form of automating the build flow via a makefile or similar. This is especially important since Gnucap requires each user to build the plugin from source, and so for the plugin to truly disseminate in the RF/MW community it must become more accessible. Since the likely users of this plugin may not be intimately familiar with programming or build processes (indeed, we should not expect them to be!), this addition is crucial if we truly wish to enable alternatives to proprietary EDA softwares. Removing the initial set-up that much of this report details is key to aligning with the project motivation.

(3) On that note, we should also make the code more concise and robust (e.g., by switching to C++ containers instead of dynamic arrays). Since we can use the current state as a 'known', then firming up the code should be fairly straightforward. Such changes would also be beneficial for making it more intuitive. An important addition would be to make the code more error-proof by, for example, adding in try-catch statements with appropriate error messages and so on.

    One element of this process would be to more closely consider the details of each calculation we perform. Significant efficiency and accuracy gains could be made by adding nuance to the algorithms. For example, we could dynamically remove columns in the VF system matrix if the residues for its PFE terms are tending towards 0 (this would indicate that the associated pole is not significant).

(4) Testing the plugin with a broader range of circuits and data is critically important to uncovering more issues and ensuring reliability. We might seek to test with actual, physically measured data, or otherwise use simulated samples of more realistic signals, such as OFDM. We should also compare its results with EDA softwares which implement some form of HB or its successors. Such comparisons can be evaluated both in terms of accuracy and speed.

    Because it was intended as a proof-of-concept. The results in this report only scratched the surface of potential use cases. This process was also expedited with a MATLAB golden model which may reach its own limitations at some point. We might therefore also require a more robust reference point.

(5) Extending the plugin functionality is also an interesting path to take. For instance, we might seek to have it work with greater than 1-port networks. Extending to work with passband FD data, as in [5], would also be a worthwhile exercise. The VF implemented in this project assumes causal inputs, and so it needs to be augmented to work with passband data.

    From the perspective of Gnucap, its plugin systems offers much greater customisability than what has been used in this project. It would be worthwhile to further explore what it can offer.

Nevertheless, we cannot ignore the fact that VF is quite robustly implemented elsewhere. We might therefore use this project as a basis for exploring more advanced techniques and alternatives to VF, such as those discussed in section 2.2.3.

There is a high likelihood of there being many bugs in the code that need to be sorted out. The code as it stands is likely inefficient and could be improved. The regrettably limited scope of the verification done in this project means that the plugin will likely break once it is more rigorously tested. Indeed, there was a significant amount of debugging done throughout which is not mentioned in this report. A significant degree of more testing is required to be confident in the plugin. The good news is that we know it works in this basic prototype state for the provided test cases. We note here some issues found during development that are of interest:

(1) It was found that the VF code does not work if we specify `nump` = 1. This is because in such a case, there are divisions by zero when constructing the initial `poles_guess`. This is of course not desirable since many systems do indeed have a single pole. It is not envisioned that fixing this would be too involved, however.

(2) It was initially desired to execute the VF function within the `tr_begin()` function in `rc_model.cc`, but this appeared to cause issues with the accuracy of the transient simulations. The reason is not clear, but may be related to unintended interference with Gnucap's control of the internal time step Δ that is caused by the initial latency from the VF algorithm. Doing the VF step outside of `precalc_first()` is desirable as it is quite wasteful of both time and compute to have VF run every time that the model instance is affected in some way. We ideally would like it to only run at the beginning of a transient simulation. Alternatively, we could perhaps split the code out and perform it separately to the plugin. The user would then point the plugin to files containing the model parameters, perhaps something akin to the `results.txt` file outputted by `vf.h`.

(3) In retrospect, since the RC companion model parameter $i_c[n]$ is not actually described by a constitutive relation (i.e., a KCL or KVL) but rather is a constant of the model parameters, like G_c, using a `FPOLY` to store it in `rc_model.cc` is probably quite wasteful since we never actually convert it to any other relation, nor do we stamp it into the MNA matrix. The impact is likely quite small, but in the interest of conciseness it should be changed.

To finish off, we reiterate the fundamental motivation of this project; the desire to contribute to the common good. Adding a capability such as this, if even just in the current rudimentary state, is quite a beneficial step towards opening up the EDA industry to more individuals. We hope that the work is appreciated by those who come across it, and that others are encouraged to develop upon it and explore its use as they see fit, or are otherwise inspired to undertake similar projects across the spectrum of electronics. We believe that should more momentum gather behind such open, community-minded incentives, then both electronics design and teaching, and also therefore the larger societal and environmental contexts which they influence so profoundly, would reap great benefits.

Bibliography

[1] "Ericsson Mobility Report November 2023," (Date last accessed: 28 March 2024). [Online]. Available: www.ericsson.com/en/reports-and-papers/mobility-report/reports/november-2023

[2] R. Ludwig and P. Bretchko, *RF Circuit Design.* Prentice Hall, 2000, ch. 1. Introduction.

[3] R. Ludwig and P. Bretchko, *RF Circuit Design.* Prentice Hall, 2000, ch. 4. Single- and Multiport Networks.

[4] K. Kundert, J. White, and A. Sangiovanni-Vincentelli, *Steady-state methods for simulating analog and microwave circuits.* Springer Science+Business Media Dordrecht, 1990, ch. 5. Harmonic Balance Theory.

[5] J. B. King and T. J. Brazil, "Time-domain simulation of passband S-parameter networks using complex baseband vector fitting," in *2017 Integrated Nonlinear Microwave and Millimetre-wave Circuits Workshop (INMMiC)*, 2017, pp. 1–4.

[6] R. Ludwig and P. Bretchko, *RF Circuit Design.* Prentice Hall, 2000.

[7] R. Ludwig and P. Bretchko, *RF Circuit Design.* Prentice Hall, 2000, ch. 2. Transmission Line Analysis.

[8] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, no. 6, pp. 504–509, 1975.

[9] F. N. Najim, *Circuit Simulation.* John Wiley & Sons, Inc., 2010.

[10] V. Litovski and M. Zwolinski, *VLSI Circuit Simulation and Optimization.* Chapman & Hall, 1997, ch. 1. Linear Resistive Circuit Analysis.

[11] K. Kundert, J. White, and A. Sangiovanni-Vincentelli, *Steady-state methods for simulating analog and microwave circuits.* Springer Science+Business Media Dordrecht, 1990, ch. 3. Background.

[12] F. N. Najim, *Circuit Simulation.* John Wiley & Sons, Inc., 2010, ch. 1. Introduction.

[13] T. J. Brazil, "Nonlinear, transient simulation of distributed RF circuits using discrete-time convolution," in *2007 IEEE International Symposium on Circuits and Systems*, 2007, pp. 505–508.

[14] K. Kundert, J. White, and A. Sangiovanni-Vincentelli, *Steady-state methods for simulating analog and microwave circuits.* Springer Science+Business Media Dordrecht, 1990.

[15] E. Ngoya and R. Larcheveque, "Envelop transient analysis: a new method for the transient and steady state analysis of microwave communication circuits and systems," in *1996 IEEE MTT-S International Microwave Symposium Digest*, vol. 3, 1996, pp. 1365–1368 vol.3.

[16] N. Carvalho, J. Pedro, W. Jang, and M. Steer, “Nonlinear RF circuits and systems simulation when driven by several modulated signals,” *IEEE Transactions on Microwave Theory and Techniques*, vol. 54, no. 2, pp. 572–579, 2006.

[17] M. Condon, R. Ivanov, and C. Brennan, “A causal model for linear RF systems developed from frequency-domain measured data,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, no. 8, pp. 457–460, 2005.

[18] J. B. King, “Time-domain representation of passband scattering parameters,” in *2020 IEEE Asia-Pacific Microwave Conference (APMC)*, Dec 2020, pp. 790–791.

[19] B. Gustavsen and A. Semlyen, “Rational approximation of frequency domain responses by vector fitting,” *IEEE Transactions on Power Delivery*, vol. 14, no. 3, pp. 1052–1061, 1999.

[20] A. Semlyen and A. Dabuleanu, “Fast and accurate switching transient calculations on transmission lines with ground return using recursive convolutions,” *IEEE Transactions on Power Apparatus and Systems*, vol. 94, no. 2, pp. 561–571, 1975.

[21] “Vector Fitting - SINTEF,” (Date last accessed: 28 March 2024). [Online]. Available: www.sintef.no/en/software/vector-fitting

[22] R. Ludwig and P. Bretchko, *RF Circuit Design.* Prentice Hall, 2000, ch. 3. The Smith Chart.

[23] Lefteriu, Sanda and Antoulas, Athanasios C., “On the Convergence of the Vector-Fitting Algorithm,” *IEEE Transactions on Microwave Theory and Techniques*, vol. 61, no. 4, pp. 1435–1443, 2013.

[24] “LAPACK - Linear Algebra PACKage,” (Date last accessed: 28 March 2024). [Online]. Available: https://netlib.org/lapack

[25] “Oracle VirtualBox,” (Date last accessed: 28 March 2024). [Online]. Available: www.virtualbox.org

[26] “Ubuntu VirtualBox tutorial,” (Date last accessed: 28 March 2024). [Online]. Available: https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox

[27] “Visual Studio Code,” (Date last accessed: 28 March 2024). [Online]. Available: https://code.visualstudio.com

[28] “Using C++ on Linux in VS Code,” (Date last accessed: 28 March 2024). [Online]. Available: https://code.visualstudio.com/docs/cpp/config-linux

[29] “GNU General Public License,” Date last accessed: 28 March 2024. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.html

[30] “Gnucap project page,” (Date last accessed: 28 March 2024). [Online]. Available: https://savannah.gnu.org/projects/gnucap

[31] “Gnucap GIT repository,” Date last accessed: 28 March 2024. [Online]. Available: https://git.savannah.gnu.org/cgit/gnucap.git

[32] “Gnucap Wiki,” (Date last accessed: 28 March 2024). [Online]. Available: http://gnucap.org/dokuwiki/doku.php/gnucap:start

[33] “GCC Command Options Summary,” Date last accessed: 30 March 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

[34] "ISO C++ CppCon 2023 Blog Post," Date last accessed: 31 March 2024. [Online]. Available: https://isocpp.org/blog/2023/09/cppcon-2023-stdlinalg-linear-algebra-coming-to-standard-cpp-mark-hoemmen

[35] "The BLAS as the Key to Portability," (Date last accessed: 31 March 2024). [Online]. Available: https://www.netlib.org/lapack/lug/node65.html

[36] "The LAPACKE C Interface to LAPACK," (Date last accessed: 31 March 2024). [Online]. Available: https://netlib.org/lapack/lapacke.html

[37] "LAPACK Documentation," Date last accessed: 31 March 2024. [Online]. Available: https://www.netlib.org/lapack/explore-html/

[38] "LAPACK dgelss documentation," Date last accessed: 31 March 2024. [Online]. Available: https://www.netlib.org/lapack/explore-html/da/d55/group__gelss_gac6159de3953ae0386c2799294745ac90.html#gac6159de3953ae0386c2799294745ac90

[39] "LAPACK dgeev documentation," Date last accessed: 31 March 2024. [Online]. Available: https://www.netlib.org/lapack/explore-html/d4/d68/group__geev_ga7d8afe93d23c5862e238626905ee145e.html#ga7d8afe93d23c5862e238626905ee145e

[40] "LAPACK dgemm documentation," Date last accessed: 31 March 2024. [Online]. Available: https://www.netlib.org/lapack/explore-html/dd/d09/group__gemm_ga1e899f8453bcbfde78e91a86a2dab984.html#ga1e899f8453bcbfde78e91a86a2dab984

[41] "GitHub repository for project code," Date last accessed: 5 April 2024. [Online]. Available: https://github.com/SHigginbotham/transient-sparam-gnucap

Appendix A1

Example Device Plugin

This appendix provides the full source code in the `cust_res.cc` file, which is discussed as an example in section 3.2.2.2 on developing device plugins with transient simulation capability.

```
/*
  SIMPLE TEST DEVICE MODEL PLUGIN FOR GNUCAP
  IMPLEMENTS V = I * R*R

  Created by:     Sean Higginbotham for M.A.I project
                  Supervisor: Dr. Justin King
                  Department of Electronic and Electrical Engineering,
                  Trinity College Dublin, Ireland, September 2023 - April 2024
*/

#include "globals.h"
#include "e_elemnt.h"

namespace
{
  class CUST_RESISTANCE : public ELEMENT
  {
  private:
  // copy constructor used to create instances of the device from the static instance
  explicit CUST_RESISTANCE(const CUST_RESISTANCE &p) : ELEMENT(p) {}
  // should use to de-allocate any dynamic memory if create any; otherwise not needed
  // ~CUST_RESISTANCE() {}
  public:
  // default constructor used by dispatcher to create static instance
  explicit CUST_RESISTANCE() : ELEMENT() {}

  private:
  // parser functions
  char id_letter() const { return 'a'; }                // id used by parser
  std::string value_name() const { return "r"; }       // value 'name'
  std::string dev_type() const { return "cust_res"; } // model 'type'
  int max_nodes() const { return 2; }                   // max # of ports parser will allow
  int min_nodes() const { return 2; }                   // min # of ports parser will allow
  // allocation functions
  int net_nodes() const { return 2; }     // actual # of ports for MNA matrix
  int matrix_nodes() const { return 2; } // the # of nodes that will be stamped into matrix
  // port functions
  std::string port_name(int i) const
  {
    // must be 0 (+ve) or 1 (-ve) terminal
    assert(i >= 0);
    assert(i < 2);
    static std::string names[] = {"p", "n"};
    return names[i];
  }

  // used to copy this copy this instance
  CARD *clone() const { return new CUST_RESISTANCE(*this); }

  // for ELEMENT derived classes, need to override the following functions
  // relating to TR and AC analysis:
  // -----------------------------------------------------------------
  // notifies the admittance matrix and LU matrix of the nodes used by this device.
  // Call *_passive(), *_active(), or *_extended() depending on device type
  void tr_iwant_matrix() { tr_iwant_matrix_passive(); }
  // obtain port voltages
  double tr_involts() const { return tr_outvolts(); }
  double tr_involts_limited() const { return tr_outvolts_limited(); }
  // same as TR version but for AC
  COMPLEX ac_involts() const { return ac_outvolts(); }
  void ac_iwant_matrix() { tr_iwant_matrix_passive(); }

  // transient analysis functions
  void tr_load() { tr_load_passive(); }     // load amittance matrix and current
      vector with values calculated during do_tr
  void tr_unload() { tr_unload_passive(); } // removes component from MNA matrix
  void tr_begin();
  bool do_tr();
  // there are many other functions (for TR analysis) but they are evidently
  // not required to be overriden unless explicitly required.
  };

  // use the following notation
  /*
  use y.x = i, y.f0 = 0, y.f1 = ohms^2
  use m.x = v, m.c0 = 0, m.c1 = mhos^2
  */

  // is called at beginning of analysis; sets up params and such.
  // main purpose is to just initialise the FPOLY and CPOLY
  void CUST_RESISTANCE::tr_begin()
  {
  // need to call base class virtual first
  ELEMENT::tr_begin();

  // values that will remain constant during sim
  _y[0].f0 = _m0.c0 = 0;
  _y[0].f1 = (value() != 0.) ? value() * value() : OPT::shortckt; // R^2
  _m0.c1 = 1. / _y[0].f1;                                         // 1 / R^2
  }

  // does most of the 'real work'. Optionally calls tr_eval() to do the stuff
  // Can forego and do own stuff too!
  bool CUST_RESISTANCE::do_tr()
  {
  // (1) assign non-constant FPOLY terms
  _y[0].x = tr_involts_limited() * (1. / _y[0].f1); // i[n]

  // (3) necessary process functions
  set_converged(conv_check()); // check convergence
  store_values();              // push values to previous iteration (i.e., store
      them!)
  q_load();                    // queue for adding to MNA matrix

  // (3) 'convert' to CPOLY values
  _m0.x = tr_involts_limited(); // v[n]

  return converged();
  }

  CUST_RESISTANCE p1;
  DISPATCHER<CARD>::INSTALL d2(&device_dispatcher, "p|custr", &p1);
}
```

Appendix A2

Example **tasks.json** for Plugin Compilation with VSCode

This appendix provides the `tasks.json` file, used in the Visual Studio Code project to compile the `cust_res.cc` device plugin. It is in effect equivalent to the command given in listing 3.7.

```
{
 "tasks": [
  {
   "type": "cppbuild",
   "label": "C/C++: g++ build active file",
   "command": "g++",
   "args": [
    "-I/usr/local/include/gnucap",
    "-L/usr/local/lib/",
    "-shared",
    "-fPIC",
    "${file}",
    "-o",
    "${fileDirname}/cust_res.so"
   ],
   "options": {
    "cwd": "${fileDirname}"
   },
   "problemMatcher": [
    "$gcc"
   ],
   "group": {
    "kind": "build",
    "isDefault": true
   },
   "detail": "Task generated by Debugger."
  }
 ],
 "version": "2.0.0"
}
```

Appendix A3

Modified **`make.inc`** File for Building LAPACK

This appendix provides the full code of the `make.inc` file that should be used to build the LAPACK library. Note this is the only makefile we need to modify in the LAPACK compilation. The only parts that are modified are the parts mentioned in section 3.3.1.

```
####################################################################
#  LAPACK make include file.                                       #
####################################################################

SHELL = /bin/sh

#  CC is the C compiler, normally invoked with options CFLAGS.
#
CC = g++
CFLAGS = -O3 -DHAVE_LAPACK_CONFIG_H -DLAPACK_COMPLEX_CPP -fPIC

#  Modify the FC and FFLAGS definitions to the desired compiler
#  and desired compiler options for your machine.  NOOPT refers to
#  the compiler options desired when NO OPTIMIZATION is selected.
#
#  Note: During a regular execution, LAPACK might create NaN and Inf
#  and handle these quantities appropriately. As a consequence, one
#  should not compile LAPACK with flags such as -ffpe-trap=overflow.
#
FC = gfortran
FFLAGS = -O2 -frecursive -fPIC
FFLAGS_DRV = $(FFLAGS)
FFLAGS_NOOPT = -O0 -frecursive -fPIC

#  Define LDFLAGS to the desired linker options for your machine.
#
LDFLAGS =

#  The archiver and the flag(s) to use when building an archive
#  (library).  If your system has no ranlib, set RANLIB = echo.
#
AR = ar
ARFLAGS = cr
RANLIB = ranlib

#  Timer for the SECOND and DSECND routines
#
#  Default:  SECOND and DSECND will use a call to the
#  EXTERNAL FUNCTION ETIME
#TIMER = EXT_ETIME
```

```
#  For RS6K:  SECOND and DSECND will use a call to the
#  EXTERNAL FUNCTION ETIME_
#TIMER = EXT_ETIME_
#  For gfortran compiler:  SECOND and DSECND will use a call to the
#  INTERNAL FUNCTION ETIME
TIMER = INT_ETIME
#  If your Fortran compiler does not provide etime (like Nag Fortran
#  Compiler, etc...) SECOND and DSECND will use a call to the
#  INTERNAL FUNCTION CPU_TIME
#TIMER = INT_CPU_TIME
#  If none of these work, you can use the NONE value.
#  In that case, SECOND and DSECND will always return 0.
#TIMER = NONE

#  Uncomment the following line to include deprecated routines in
#  the LAPACK library.
#
#BUILD_DEPRECATED = Yes

#  LAPACKE has the interface to some routines from tmglib.
#  If LAPACKE_WITH_TMG is defined, add those routines to LAPACKE.
#
#LAPACKE_WITH_TMG = Yes

#  Location of the extended-precision BLAS (XBLAS) Fortran library
#  used for building and testing extended-precision routines.  The
#  relevant routines will be compiled and XBLAS will be linked only
#  if USEXBLAS is defined.
#
#USEXBLAS = Yes
#XBLASLIB = -lxblas

#  The location of the libraries to which you will link.  (The
#  machine-specific, optimized BLAS library should be used whenever
#  possible.)
#
BLASLIB      = $(TOPSRCDIR)/librefblas.so
CBLASLIB     = $(TOPSRCDIR)/libcblas.so
LAPACKLIB    = $(TOPSRCDIR)/liblapack.so
TMGLIB       = $(TOPSRCDIR)/libtmglib.so
LAPACKELIB   = $(TOPSRCDIR)/liblapacke.so

#  DOCUMENTATION DIRECTORY
# If you generate html pages (make html), documentation will be placed in $(DOCSDIR
    )/explore-html
# If you generate man pages (make man), documentation will be placed in $(DOCSDIR)/
    man
DOCSDIR       = $(TOPSRCDIR)/DOCS
```

Appendix A4

`vf.h` File

This appendix gives the code for the header file `vf.h` which implements the VF algorithm for use inside the RC model plugin, `rc_model.cc`. Both files are provided on the GitHub page for this project at [41].

```
/*
  RECURSIVE CONVOLUTION COMPANION MODEL FOR GNUCAP
  USES VECTOR FITTING ALGORITHM TO FIT S-PARAMETER DATA
  AND ALLOW TRANSIENT SIMULATION OF 1-PORT, LTI S-PARAMETER BLOCKS

  THIS IS THE VECTOR FITTING ALGORITHM HEADER FILE
  IT EMPLOYS THE LAPACK LINEAR ALGEBRA LIBRARY

  Created by: Sean Higginbotham for M.A.I project
              Supervisor: Dr. Justin King
              Department of Electronic and Electrical Engineering,
              Trinity College Dublin, Ireland, September 2023 - April 2024

  Copyright (C) 2024 Sean Higginbotham
  Author: Sean Higginbotham <higginbs@tcd.ie>

  This program is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#pragma once

#define _USE_MATH_DEFINES

#include <iostream>
#include <iomanip>
#include <cmath>
#include <complex>
#include <fstream>
#include <vector>
#include <stdlib.h>
#include <string>
#include <algorithm>
```

```cpp
#include <functional>
#include <stdlib.h>
#include <filesystem>
#include <chrono>
#include <ctime>

// LAPACK
#include <lapacke.h>
#include <cblas.h>

// OUTPUT OF THIS FUNCTION SHOULD BE 0 IF FAILED, AND 1 IF SUCCESSFUL

using namespace std::complex_literals;

std::complex<double> one(1.0, 0);
std::complex<double> imag = 1i;

bool do_vector_fitting(std::vector<std::complex<double>> &p, std::vector<std::
    complex<double>> &r, double &rem, int nump, int numi, bool vflog)
{
  auto start = std::chrono::system_clock::now(); // start time of VF run

  // set up log file for VF
  if (vflog)
  {
    try
    {
      std::filesystem::remove("vf_log.txt");
    }
    catch (...)
    {
    }
  }
  std::ofstream log_file;
  if (vflog)
    log_file.open("vf_log.txt", std::ios::app | std::ios::out);
  std::time_t in_time = std::chrono::system_clock::to_time_t(start);
  if (log_file)
    log_file << "VECTOR FITTING LOG FILE FOR TR RUN AT : " << std::ctime(&in_time)
        << "\t====================================\n";

  ////////////////
  // Definitions //
  ////////////////

  // the frequencies at which the data points occur (in Hz)
  std::vector<double> freqs_hz;
  // the data points, are a complex number (cause is freq domain)
  std::vector<std::complex<double>> sp_points;
  // name of the S-param data file
  std::string sp_file = "s_param_data.txt";

  /////////////////////
  // Get S-param data //
  /////////////////////

  in_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
  if (log_file)
    log_file << std::ctime(&in_time) << "\tReading in S-param data...\n";
  std::ifstream s_param_file(sp_file);
  if (s_param_file.is_open())
  {
    std::string row;
    while (getline(s_param_file, row))
    {
      if (row[0] == '!')
        ;
      else
      {
        char c;
        int8_t i = 0;
        size_t p;
        double real = 0., imag = 0.;
        while (c = row[0])
        {
          if (isspace(c) || c == ',')
            row.erase(0, 1);
          else
          {
            i++; // 1 = freqs, 2 == real data, 3 == imag data
            p = row.find_first_of("\t\v\f\n\r ");
            std::string t = row.substr(0, p);
            if (i == 1)
              freqs_hz.push_back(atof(t.c_str()));
            else if (i == 2)
              real = atof(t.c_str());
            else if (i == 3)
              imag = atof(t.c_str());
            row.erase(0, t.length());
          }
        }
        std::complex<double> z_temp(real, imag);
        sp_points.push_back(z_temp);
      }
    }
    s_param_file.close();
  }
  else
  {
    if (log_file)
      log_file << "\tS-param data not available! Make sure there is a file named 's_param_data.txt' in the same directory...\n\tExiting run...\n";
    if (log_file)
      log_file.close();
    return 0;
  }

  //////////////////////////
  // Vector Fitting setup //
  //////////////////////////

  int iters = numi;     // # of iterations of the fitting alg
  int num_poles = nump; // # of poles to use in the fit

  // transform to be in terms of omega (2 * pi * f) and
  // normalise to range of 0 to 1
  std::vector<double> freqs_hz_omega;
  for (int j = 0; j < freqs_hz.size(); j++)
  {
    freqs_hz_omega.push_back(freqs_hz[j] * 2 * M_PI);
  }
  double max_freq = *(std::max_element(freqs_hz_omega.begin(), freqs_hz_omega.end()));
  for (int j = 0; j < freqs_hz_omega.size(); j++)
  {
    freqs_hz_omega[j] = freqs_hz_omega[j] / max_freq;
  }

  // distribute poles across 2PI range in S-plane as per section 3.2 of paper
  // this acts as our initial pole guess
  double increment = (freqs_hz_omega.back() - freqs_hz_omega[1]) / (num_poles - 1);
  std::vector<double> freq_range; // frequencies at which (initial) poles should
      occur (this is DISTINCT from our s-param freqs, which are given by
      freqs_hz_omega)
  for (int j = 0; j < num_poles; j++)
  {
    freq_range.push_back(2 * M_PI * (freqs_hz_omega[1] + increment * (double)j));
  }
  std::vector<std::complex<double>> poles_guess; // need complex poles, not just
      real
  for (auto x : freq_range)
  {
    double real = -0.01 * x;
    double imag = x;
    std::complex<double> temp(real, imag);
    poles_guess.push_back(temp);
  }
  std::vector<std::complex<double>> freqs_comp; // wants freqs to be COMPLEX
      NUMBERS with freq on imag part
  for (auto x : freqs_hz_omega)
  {
    std::complex<double> temp(0, x);
    freqs_comp.push_back(temp);
  }

  ///////////////////////////////
  // Vector Fitting Algorithm //
  ///////////////////////////////

  // the purpose of the algorithm is to get a good fit, and
  // then we can simply extract the poles, residues, and remainder of this fit
  // to use in the RC companion model

  // open file for outputting poles of algorithm, and insert initial guess

  // delete if exists
  if (vflog)
  {
    try
    {
      std::filesystem::remove("pole_guess.txt");
    }
    catch (...)
    {
    }
  }

  std::ofstream pole_write;
  if (vflog)
    pole_write.open("pole_guess.txt", std::ios::app | std::ios::out);
  if (pole_write)
  {
    pole_write << "iteration, poles in 2pif (note these should be multiplied by
        max_freq to get full scale values; max_freq = " << max_freq << ")\ninit";
    for (auto x : poles_guess)
    {
      pole_write << ", " << x.real() << '+' << x.imag() << 'i';
    }
    pole_write << '\n';
  }

  std::vector<std::complex<double>> real_poles;                           // real
      poles
  std::vector<std::complex<double>> comp_poles;                           // complex
       poles
  std::vector<std::complex<double>> all_zeroes;                           // zeroes
      computed from sigma function residues
  std::vector<std::complex<double>> keep_zeroes;                          // zeroes
      computed from sigma function residues that are used for next pole iteration
  std::vector<std::complex<double>> all_zeroes_prev(num_poles * 2, 0.0); // for
      storing sigma zeroes (== poles) of previous iteration

  double remainder;                           // remainder of the VF fit
  std::vector<std::complex<double>> residues; // residues of the VF fit

  in_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
  if (log_file)
    log_file << std::ctime(&in_time) << "\tRunning Vector Fitting algorithm with "
        << num_poles << " starting poles for " << iters << " iterations...\n";

  // PERFORM ITERATIONS TO OBTAIN OPTIMUM FIT

  if (vflog)
  {
    try
    {
      std::filesystem::remove("residual_log.txt");
    }
    catch (...)
    {
    }
  }
  std::ofstream resid_log;
  if (vflog)
    resid_log.open("residual_log.txt", std::ios::app | std::ios::out);
  // std::time_t in_time = std::chrono::system_clock::to_time_t(start);
  if (resid_log)
    resid_log << "Output of VF residuals over iterations;\t[iteration, #rows A_real
        , #cols A_real, rank, VF residual, max pole diff]\n";

  for (int iteration = 0; iteration <= iters; iteration++)
  {
    // here we use our poles guess, the read-in S-param data, and the frequencies
        they occur at
    // we form the Ax = B linear system, and solve for the residues and remainder
        using LAPACKE

    // Steps are as follows:

    //  (i)      form A matrix as per appendix A of VF paper
    //  (ii)     B matrix is just s-param data (sp_points) separated into real and
        imag parts
    //  (iii)    separate out real and imag parts of A (also as per app. A)
    //  (iv)     solve Ax=B with LAPACKE
    //  (v)      extract zeroes (i.e., poles of next iteration and of final fit)
        from this solution with LAPACKE, these act as poles of the next iteration
    //  (vi)     iterate steps (1) to (5) for specified # iterations or until
        convergence
    //  (vii)    when iterations are done, run least squares a final time to get the
         residues and remainder
    //           corresponding to final set of poles
    //  (viii)  return to RC model with the correct poles, residues, and remainder
        describing the fit

    // (1) PRELIMINARIES

    // poles are updated on each iteration, so make sure the updated
    // ones remain stable (-ve real part)
    for (int k = 0; k < poles_guess.size(); k++)
    {
      if (poles_guess[k].real() > 0)
      {
        std::complex<double> temp(poles_guess[k].real() * -1.0, poles_guess[k].imag
            ());
        poles_guess[k] = temp;
      }
    }
    // divide into real and complex poles
    for (auto x : poles_guess)
    {
      if (x.imag() == 0)
        real_poles.push_back(x);
      else
        comp_poles.push_back(x);
    }

    // here we define sizes and create arrays. This needs to be done each iteration
         because the number of poles can CHANGE
    // with each iteration, so we cannot use a fixed set up
    // NOTE
    // s-param data is given by sp_points
    // (scaled to 2PIf) frequencies for sp_points is given by freqs_comp
    // poles are given by poles_guess

    // FREQUENTLY USED SIZES AND THINGS
    // -------------------------------
    std::size_t num_freqs = freqs_comp.size(); // # S-param frequencies
    std::size_t num_rp = real_poles.size();    // # real poles
    std::size_t num_cp = comp_poles.size();    // # complex poles
    std::size_t num_sp = sp_points.size();     // # S-param points
    std::size_t num_poles = poles_guess.size();
    int pole_iter = 0;
    double residual = 0;

    // VARIABLE ARRAY SIZES
    // --------------------

    std::size_t a_matrix_r = num_freqs, a_matrix_c = 2 * (num_rp + 2 * num_cp) + 1;
    std::size_t b_matrix_r = num_sp * 2 - 1, b_matrix_c = 1;
    std::size_t a_matrix_real_r = 2 * num_freqs - 1, a_matrix_real_c = 2 * (num_rp
        + 2 * num_cp) + 1;

    std::size_t Az_real_r = num_rp, Az_real_c = num_rp;
    std::size_t bz_real_r = num_rp, bz_real_c = 1;
    std::size_t c_real_r = 1, c_real_c = num_rp;
    std::size_t bc_real_r = num_rp, bc_real_c = num_rp;
    std::size_t H_real_r = num_rp, H_real_c = num_rp;
    std::size_t real_zeroes_real_d = num_rp, real_zeroes_imag_d = num_rp;

    std::size_t Az_comp_r = 2 * num_cp, Az_comp_c = 2 * num_cp;
    std::size_t bz_comp_r = 2 * num_cp, bz_comp_c = 1;
    std::size_t c_comp_r = 1, c_comp_c = 2 * num_cp;
    std::size_t bc_comp_r = 2 * num_cp, bc_comp_c = 2 * num_cp;
    std::size_t H_comp_r = 2 * num_cp, H_comp_c = 2 * num_cp;
    std::size_t comp_zeroes_real_d = 2 * num_cp, comp_zeroes_imag_d = 2 * num_cp;

    // for least squares solution (dgelss)
    lapack_int m_soln = a_matrix_real_r, // # rows of A
        n_soln = a_matrix_real_c,        // # columns of A
        nrhs_soln = 1,                   // # columns of B
        lda_soln = n_soln,               // leading dimension of A (== # of rows
            for COL-MAJOR, == # cols for ROW-MAJOR)
        ldb_soln = b_matrix_c,           // leading dimension of B (== # of rows
            for COL-MAJOR, == # rows for ROW-MAJOR)
        info_soln;                       // returns '0' if is successful
    double singval[n_soln];
    int jpvt[n_soln];
    int rank;

    // for real zeroes matrix multiplication (dgemm)
    lapack_int m_mult_r = bz_real_r, // # rows of bz_real (column vector)
        n_mult_r = c_real_c,         // # columns of c_real (row vector)
        k_mult_r = bz_real_c,        // # cols bz_real == # rows c_real
        lda_mult_r = m_mult_r,
               ldb_mult_r = k_mult_r,
               ldc_mult_r = num_rp; // 'first dimension' of output matrix

    // for real zeroes eigenvalues (dgeev)
    lapack_int info_eig_r,
        n_eig_r = num_rp, // order of matrix
        lda_eig_r = n_eig_r;

    // for comp zeroes matrix multiplication (dgemm)
    lapack_int m_mult_c = bz_comp_r, // # rows of bz_comp (column vector)
        n_mult_c = c_comp_c,         // # columns of c_comp (row vector)
        k_mult_c = bz_comp_c,        // # cols bz_comp == # rows c_comp
        lda_mult_c = m_mult_c,
               ldb_mult_c = k_mult_c,
               ldc_mult_c = 2 * num_cp; // first dimension of output matrix

    // for comp zeroes eigenvalues (dgeev)
    lapack_int info_eig_c,
        n_eig_c = 2 * num_cp, // order of matrix (eigenvalues only defined for
            square matrices, so order is equivalent to #rows == #cols of H matrix)
        lda_eig_c = n_eig_c;  // leading dimension of H matrix (== max(1, N))

    // VARIABLE ARRAY DECLARATIONS
    // note they're defined as just 1D array so that we don't need to use pointers
        to pointers
    // ---------------------------

    std::complex<double> *a_matrix = new std::complex<double>[a_matrix_r *
        a_matrix_c];
    double *b_matrix = new double[b_matrix_r * b_matrix_c];
    double *b_matrix_final = new double[(b_matrix_r + 1) * b_matrix_c]; // for
        final extraction; we don't ignore DC frequency here so add + 1;
    double *a_matrix_real = new double[a_matrix_real_r * a_matrix_real_c];

    double *Az_real = new double[Az_real_r * Az_real_c]; // should be zero
        initialised
    double *bz_real = new double[bz_real_r * bz_real_c];
    double *c_real = new double[c_real_r * c_real_c];
    double *bc_real = new double[bc_real_r * bc_real_c];
    double *H_real = new double[H_real_r * H_real_c];
    double *real_zeroes_real = new double[real_zeroes_real_d];
    double *real_zeroes_imag = new double[real_zeroes_imag_d];

    double *Az_comp = new double[Az_comp_r * Az_comp_c]; // should be zero
        initialised
    double *bz_comp = new double[bz_comp_r * bz_comp_c]; // should be zero
        initialised
    double *c_comp = new double[c_comp_r * c_comp_c];
    double *bc_comp = new double[bc_comp_r * bc_comp_c];
    double *H_comp = new double[H_comp_r * H_comp_c];
    double *comp_zeroes_real = new double[comp_zeroes_real_d];
    double *comp_zeroes_imag = new double[comp_zeroes_imag_d];

    // ZERO-INITIALISE APPROPRIATE ARRAYS
    for (int j = 0; j < Az_real_r * Az_real_c; j++)
      Az_real[j] = 0;
    for (int j = 0; j < Az_comp_r * Az_comp_c; j++)
      Az_comp[j] = 0;
    for (int j = 0; j < bz_comp_r * bz_comp_c; j++)
      bz_comp[j] = 0;

    // (2) PERFORM VECTOR FITTING ITSELF

    in_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now()
        );
    if (log_file)
      log_file << std::ctime(&in_time) << "\tIteration " << iteration << '\t' <<
          num_rp << '\t' << num_cp << '\n'; // FOR DEBUGGING

    // FILL IN ARRAYS
    // --------------

    // (i) a_matrix
    //  A matrix will be P X 2N + 1
    //  P rows corresponding to frequencies at which s-params were collected (so
        same size as B matrix)
    //  N columns each side of the '1' term corresponding to specified poles
    //
    //    -left half of A matrix is of form:
    //        1/(s-p) for real poles
    //        1/(s-p) + 1/(s-p'), j/(s-p) - j/(s-p') FOR EACH complex pole (
        conjugate pair; i.e., there are TWO entries for each complex pole)
    //        All of the first entries come first, then all of the second entries
    //    -right half of A matrix is the same form, but just scaled by the s-param
        data in the numerator of the partial fractions
    //    -in the middle is a '1' corresponding to the remainder term; note we
        leave out the proportional term cause supposedly
    //     this is 0 for S-parameters...
    for (int j = 0; j < num_freqs; j++) // rows
    {
      pole_iter = 0;
      for (int k = 0; k < 2 * (num_rp + 2 * num_cp) + 1; k++) // cols
      {
        if (k >= 0 && k < num_rp) // LHS real entries
        {
          a_matrix[j * a_matrix_c + k] = one / (freqs_comp[j] - real_poles[
              pole_iter]);

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(
              a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j *
              a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].
              imag()))
          {
            if (log_file)
```

```
              log_file << "inf or NaN in index " << j << ", " << k << " in LHS real\nExiting run..." << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter++;
        }
        else if (k >= num_rp && k < num_rp + num_cp) // LHS comp poles, corresponding to real part of residue for the pair
        {
          a_matrix[j * a_matrix_c + k] = one / (freqs_comp[j] - comp_poles[pole_iter - num_rp]) + one / (freqs_comp[j] - std::conj(comp_poles[pole_iter - num_rp]));

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in LHS comp 1 with pole_iter of " << pole_iter << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter++;
        }
        else if (k >= num_rp + num_cp && k < num_rp + 2 * num_cp) // LHS comp poles, corresponding to imag part of residue for the pair
        {
          a_matrix[j * a_matrix_c + k] = imag * ((one / (freqs_comp[j] - comp_poles[pole_iter - num_rp - num_cp])) - (one / (freqs_comp[j] - std::conj(comp_poles[pole_iter - num_rp - num_cp]))));

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in LHS comp 2 with pole_iter of" << pole_iter << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter++;
        }
        else if (k == num_rp + 2 * num_cp) // insert '1' for remainder
        {
          a_matrix[j * a_matrix_c + k] = one;

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in '1' column" << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter = 0; // reset pole_iter for RHS
        }
        else if (k > num_rp + 2 * num_cp && k <= num_rp + 2 * num_cp + num_rp) // RHS real poles
        {
          a_matrix[j * a_matrix_c + k] = -sp_points[j] / (freqs_comp[j] - real_poles[pole_iter]);

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in RHS real " << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter++;
        }
        else if (k > num_rp + 2 * num_cp + num_rp && k <= num_rp + 2 * num_cp + num_rp + num_cp) // RHS comp poles, corresponding to real part of residue for the pair
        {
          a_matrix[j * a_matrix_c + k] = -sp_points[j] / (freqs_comp[j] - comp_poles[pole_iter - num_rp]) + (-sp_points[j]) / (freqs_comp[j] - std::conj(comp_poles[pole_iter - num_rp]));

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in RHS comp 1 with pole_iter of " << pole_iter << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter++;
        }
        else if (k > 2 * num_rp + 3 * num_cp && k <= num_rp + 2 * num_cp + num_rp + 2 * num_cp) // RHS comp poles, corresponding to imag part of residue for the pair
        {
          a_matrix[j * a_matrix_c + k] = -(imag * sp_points[j]) / (freqs_comp[j] - comp_poles[pole_iter - num_rp - num_cp]) - (imag * (-sp_points[j])) / (freqs_comp[j] - std::conj(comp_poles[pole_iter - num_rp - num_cp]));

          // check for issues
          if (std::isinf(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].real()) || std::isnan(a_matrix[j * a_matrix_c + k].imag()) || std::isinf(a_matrix[j * a_matrix_c + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in RHS comp 2 with pole_iter of " << pole_iter << std::endl;
            if (log_file)
              log_file.close();
            if (pole_write)
              pole_write.close();
            return 0;
          }
          pole_iter++;
        }
      }
    }

    // (ii) b_matrix
    for (int k = 0; k < num_sp; k++)
    {
      b_matrix[k * b_matrix_c + 0] = sp_points[k].real();
      b_matrix_final[k * b_matrix_c + 0] = sp_points[k].real();
      b_matrix_final[(num_sp + k) * b_matrix_c + 0] = sp_points[k].imag(); // for final don't skip DC

      // check for issues
      if (std::isinf(b_matrix[k * b_matrix_c + 0]) || std::isnan(b_matrix[k * b_matrix_c + 0]))
      {
        if (log_file)
          log_file << "inf or NaN in index " << k << " in B matrix\nExiting run...";
        if (log_file)
          log_file.close();
        if (pole_write)
          pole_write.close();
        return 0;
      }
    }
    for (int k = 0; k < (num_sp - 1); k++)
    {
      b_matrix[(num_sp + k) * b_matrix_c + 0] = sp_points[1 + k].imag(); // skip DC component for imag part

      // check for issues
      if (std::isinf(b_matrix[(num_sp + k) * b_matrix_c + 0]) || std::isnan(b_matrix[(num_sp + k) * b_matrix_c + 0]))
      {
        if (log_file)
          log_file << "inf or NaN in index " << k << " in B matrix\nExiting run...";
        if (log_file)
          log_file.close();
        if (pole_write)
          pole_write.close();
        return 0;
      }
    }

    // (iii) a_matrix_real
    // system matrix A decomposed into real and imag parts, minus 1 in rows cause
        we skip DC component for imag half
    for (int j = 0; j < num_freqs; j++) // rows
    {
      for (int k = 0; k < 2 * (num_rp + 2 * num_cp) + 1; k++) // cols
      {
        a_matrix_real[j * a_matrix_real_c + k] = a_matrix[j * a_matrix_c + k].real
            ();
      }
    }
    for (int j = 0; j < num_freqs - 1; j++) // rows (-1 cause skipping DC component
        )
    {
      for (int k = 0; k < 2 * (num_rp + 2 * num_cp) + 1; k++) // cols
      {
        a_matrix_real[(j + num_freqs) * a_matrix_real_c + k] = a_matrix[(1 + j) *
            a_matrix_c + k].imag(); // skip DC component for rows
      }
    }

    // (iv) solve Ax = B using LAPACKE (should probably move this and all above
        into first iteration loop?)

    // this function (dgelss) solves a linear system where system matrix (A) may be
         rank deficient
    // solution matrix will have #rows = #cols of A and #cols = #cols of B, and be
        stored in 'b_matrix'
    // residual of the solution is the sum of squares of the n + 1 : m elements in
        b_matrix
    // but this is supposedly only 'valid' if m_soln > n_soln && rank == n_soln...?
         We output the residual but it is best not to draw conclusions from it as of
         now....
    info_soln = LAPACKE_dgelss(LAPACK_ROW_MAJOR, m_soln, n_soln, nrhs_soln,
        a_matrix_real, lda_soln, b_matrix, ldb_soln, singval, -1, &rank);

    if (info_soln != 0)
    {
      if (log_file)
        log_file << "LAPACKE_dgelss failed on iteration << " << iteration << "with
            return value: " << info_soln << "\nExiting run...\n";
      if (pole_write)
        pole_write.close();
      if (log_file)
        log_file.close();
      return 0;
    }

    // THIS GIVES (SUM OF SQUARES) RESIDUALS OF THE VF SYSTEM, CAN USE FOR
        COMPARING TO ACTUAL DATA IF DESIRED (though its validity is dubious!)
    // if (m_soln > n_soln && rank == n_soln)
    // {
    for (int j = a_matrix_real_c; j < a_matrix_real_r; j++)
    {
      for (int k = 0; k < b_matrix_c; k++)
      {
        residual += std::pow(b_matrix[j * b_matrix_c + k], 2);
      }
    }
    // if (resid_log) resid_log << iteration << ", " << m_soln << ", " << n_soln <<
        ", " << rank << ',' << residual << '\n';
    // }
    /*else */ if (resid_log)
      resid_log << iteration << ", " << m_soln << ", " << n_soln << ", " << rank <<
          ", " << residual << ", ";

    // (v) EXTRACT ZEROES FROM RESIDUES OF THE SIGMA FUNCTION

    // because C is 0 indexed, we use the final iteration to get the final fit (
        step (vii))
    if (iteration < iters)
    {

      // We construct the H matrix for real poles/residues and solve for
          eigenvalues (real zeroes of sigma function)
      // to get poles of next iteration

      for (int k = 0; k < num_rp; k++)
      {
        Az_real[k * Az_real_c + k] = real_poles[k].real(); // diagonal matrix of
            poles
        // check for issues
        if (std::isnan(Az_real[k * Az_real_c + k]) || std::isinf(Az_real[k *
            Az_real_c + k]))
        {
          if (log_file)
            log_file << "inf or Nan at index (" << k << ',' << k << ") of Az_real!\
                n";
          if (pole_write)
            pole_write.close();
          if (log_file)
            log_file.close();
          return 0;
        }

        bz_real[k * bz_real_c + 0] = 1; // column vector of 1s

        c_real[0 * c_real_c + k] = b_matrix[(num_rp + 2 * num_cp + 1 + k) *
            b_matrix_c + 0]; // column vector of sigma residues
        // check for issues
        if (std::isnan(c_real[k]) || std::isinf(c_real[k]))
        {
          if (log_file)
            log_file << "inf or Nan at index " << k << " of c_real!\n";
          if (pole_write)
            pole_write.close();
          if (log_file)
            log_file.close();
          return 0;
        }
      }
      if (num_rp > 0) // some params of cblas_dgemm must be at least 1, so if no
          real poles exist then this operation won't be valid and should skip it
      {
        // matrix multiplication to obtain bz * c, and then elementwise subtraction
            to get the H matrix
        cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m_mult_r, n_mult_r,
            k_mult_r, 1.0, bz_real, lda_mult_r, c_real, ldb_mult_r, 0, bc_real,
            ldc_mult_r);
        for (int k = 0; k < num_rp; k++)
          for (int j = 0; j < num_rp; j++)
            H_real[k * H_real_c + j] = Az_real[k * Az_real_c + j] - bc_real[j *
                bc_real_c + k]; // bc_real is stored as transpose of what we want,
                so swap row and col indices when accessing to maintain correct
                operations

        // GET EIGENVALUES for real poles; real parts are stored in
            real_zeroes_real, and imag parts in real_zeroes_imag
        info_eig_r = LAPACKE_dgeev(LAPACK_ROW_MAJOR, 'N', 'N', n_eig_r, H_real,
            lda_eig_r, real_zeroes_real, real_zeroes_imag, NULL, 1, NULL, 1);
        if (info_eig_r != 0)
        {
          if (log_file)
            log_file << "LAPACKE_dgeev failed with real poles!\nExiting run...\n";
          if (pole_write)
            pole_write.close();
          if (log_file)
            log_file.close();
          return 0;
        }
      }

      // construct H matrix for complex poles/residues and solve for eigenvalues (
          complex zeroes of sigma function)

      for (int k = 0; k < num_cp; k++)
      {
        Az_comp[k * Az_comp_c + k] = comp_poles[k].real();
                                                                    // top left
            sub matrix (real coeffs)
        Az_comp[k * Az_comp_c + (num_cp + k)] = comp_poles[k].imag();
                                                         // top right sub matrix (
            imag coeffs)
        Az_comp[(num_cp + k) * Az_comp_c + k] = -comp_poles[k].imag();
                                                        // bottom left sub matrix
            (-ve of imag coeffs)
        Az_comp[(num_cp + k) * Az_comp_c + (num_cp + k)] = comp_poles[k].real();
                                              // bottom right sub matrix (real
            coeffs)
        bz_comp[k * bz_comp_c + 0] = 2;

            // first half is a bunch of 2s, rest is 0s
        c_comp[0 * c_comp_c + k] = b_matrix[(2 * num_rp + 2 * num_cp + 1 + k) *
            b_matrix_c + 0];                    // 1st half is real part of
            residues
        c_comp[0 * c_comp_c + (num_cp + k)] = b_matrix[(2 * num_rp + 2 * num_cp + 1
             + num_cp + k) * b_matrix_c + 0]; // 2nd half is imag part of residues
      }
      if (num_cp > 0)
      {
        // matrix multiplication to obtain bz * c, and then elementwise subtraction
             to get the H matrix
        cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m_mult_c, n_mult_c,
            k_mult_c, 1.0, bz_comp, lda_mult_c, c_comp, ldb_mult_c, 0, bc_comp,
            ldc_mult_c);
        for (int k = 0; k < 2 * num_cp; k++)
        {
          for (int j = 0; j < 2 * num_cp; j++)
          {
            H_comp[k * H_comp_c + j] = Az_comp[k * Az_comp_c + j] - bc_comp[j *
                bc_comp_c + k]; // bc_comp is stored as the transpose of what we
```

```
              want, so swap row and col indices when accessing
      }
    }

    // GET EIGENVALUES for complex poles; real parts are stored in
       comp_zeroes_real, and imag parts in comp_zeroes_imag
    info_eig_c = LAPACKE_dgeev(LAPACK_ROW_MAJOR, 'N', 'N', n_eig_c, H_comp,
       lda_eig_c, comp_zeroes_real, comp_zeroes_imag, NULL, 1, NULL, 1);
    if (info_eig_c != 0)
    {
      if (log_file)
        log_file << "LAPACKE_dgeev failed with complex poles!\nExiting run...\n
           ";
      if (pole_write)
        pole_write.close();
      if (log_file)
        log_file.close();
      return 0;
    }
  }

  // combine real and complex parts of zeroes;
  for (int k = 0; k < num_rp; k++)
  {
    std::complex<double> temp(real_zeroes_real[k], real_zeroes_imag[k]);
    all_zeroes.push_back(temp);
  }
  for (int k = 0; k < 2 * num_cp; k++)
  {
    std::complex<double> temp(comp_zeroes_real[k], comp_zeroes_imag[k]);
    all_zeroes.push_back(temp);
  }

  // compute max magnitude difference in pole guess
  double max_diff = -1;
  if (all_zeroes.size() == all_zeroes_prev.size()) // we expect the size to
     remain constant throughout the alg, but just in case...
  {
    for (int j = 0; j < all_zeroes.size(); j++)
    {
      double mag_diff = std::abs(all_zeroes[j] - all_zeroes_prev[j]);
      if (mag_diff > max_diff)
        max_diff = mag_diff;
    }
  }
  all_zeroes_prev = all_zeroes;
  if (resid_log)
    resid_log << max_diff << '\n';

  // only keep zeroes with positive imag part (one of the complex conj pairs)
     in 'keep zeroes', and also filter out those with very small magnitudes (
     meaning they don't contribute to the fit)
  for (auto x : all_zeroes)
  {
    if (x.imag() >= 0)
    {
      if (std::abs(x) > 1E-10)
      {
        keep_zeroes.push_back(x);
      }
    }
  }

```

```
      // SUB IN ZEROES AS POLES OF NEXT ITERATION

      poles_guess.clear(); // make sure vector is empty (probably doesn't make a
          difference...)
      poles_guess = keep_zeroes;
    }
    // (vii) after done iterating, take out the appropriate stuff
    //       we neeed to just get the fit residues and remainder that correspond to
        the last set of poles used to solve the Ax=B system (otherwise our fit
        wouldn't match up!)
    else if (iteration == iters)
    {
      // set up
      // since we only do this final iteration once it is MORE efficient to define
          everthing down here rather than with
      // the PRELIMINARIES above
      int a_matrix_final_r = num_freqs;
      int a_matrix_final_c = 2 * num_poles + 1;
      int a_matrix_final_real_r = 2 * num_freqs;
      int a_matrix_final_real_c = 2 * num_poles + 1;
      std::complex<double> *a_matrix_final = new std::complex<double>[
          a_matrix_final_r * a_matrix_final_c];
      double *a_matrix_final_real = new double[a_matrix_final_real_r *
          a_matrix_final_real_c];

      int m_final = a_matrix_final_real_r;
      int n_final = a_matrix_final_real_c;
      int nrhs_final = b_matrix_c;
      int lda_final = n_final;
      int ldb_final = nrhs_final;

      double singval_final[n_final];
      int jpvt_final[n_final];
      int rank_final;

      std::vector<double> residues_real;
      std::vector<double> residues_imag;

      // create a_matrix for final extraction
      // note that since we only care about the fit residues, we don't need the RHS
          of the system matrix
      for (int j = 0; j < a_matrix_final_r; j++) // rows
      {
        for (int k = 0; k < num_poles; k++) // cols (only need to loop over num
            poles)
        {
          a_matrix_final[j * (2 * num_poles + 1) + k] = one / (freqs_comp[j] -
              poles_guess[k]) + one / (freqs_comp[j] - std::conj(poles_guess[k]));
          a_matrix_final[j * (2 * num_poles + 1) + (num_poles + k)] = imag * (one /
              (freqs_comp[j] - poles_guess[k]) - one / (freqs_comp[j] - std::conj(
              poles_guess[k])));

          // check for issues
          if (std::isinf(a_matrix_final[j * (2 * num_poles + 1) + k].real()) || std
              ::isnan(a_matrix_final[j * (2 * num_poles + 1) + k].real()) || std::
              isnan(a_matrix_final[j * (2 * num_poles + 1) + k].imag()) || std::
              isinf(a_matrix_final[j * (2 * num_poles + 1) + k].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << k << " in
                  a_matrix_final norm!\n Exiting run...\n";
            if (pole_write)
              pole_write.close();
            if (log_file)
              log_file.close();
            return 0;
          }
          if (std::isinf(a_matrix_final[j * (2 * num_poles + 1) + (num_poles + k)].real()) || std::isnan(a_matrix_final[j * (2 * num_poles + 1) + (num_poles + k)].real()) || std::isnan(a_matrix_final[j * (2 * num_poles + 1) + (num_poles + k)].imag()) || std::isinf(a_matrix_final[j * (2 * num_poles + 1) + (num_poles + k)].imag()))
          {
            if (log_file)
              log_file << "inf or NaN in index " << j << ", " << num_poles + k << " in a_matrix_final conj!\n Exiting run...\n";
            if (pole_write)
              pole_write.close();
            if (log_file)
              log_file.close();
            return 0;
          }
        }
        a_matrix_final[j * (a_matrix_final_c) + (a_matrix_final_c - 1)] = one; // 1 for remainder term
      }
      // turn into real matrix
      for (int j = 0; j < a_matrix_final_r; j++)
      {
        for (int k = 0; k < a_matrix_final_real_c; k++)
        {
          a_matrix_final_real[j * a_matrix_final_real_c + k] = a_matrix_final[j * a_matrix_final_c + k].real();
          a_matrix_final_real[(j + a_matrix_final_real_r / 2) * a_matrix_final_real_c + k] = a_matrix_final[j * a_matrix_final_c + k].imag();
        }
      }

      // perform a last least squares
      info_soln = LAPACKE_dgelss(LAPACK_ROW_MAJOR, m_final, n_final, nrhs_final, a_matrix_final_real, lda_final, b_matrix_final, ldb_final, singval_final, -1, &rank);
      if (info_soln != 0)
      {
        if (log_file)
          log_file << "\nLAPACKE_dgelss for final (iteration " << iteration << ") failed: " << info_soln << "\nExiting run...\n";
        if (pole_write)
          pole_write.close();
        if (log_file)
          log_file.close();
        return 0;
      }

      // get residues and remainder

      // solution vector is of dimensions n_final x nrhs_final
      // first num_poles terms should be real parts of residues, second num_poles terms should be imag parts
      // final term should be the remainder

      remainder = b_matrix_final[n_final - 1];
      for (int k = 0; k < num_poles; k++)
      {
        residues_real.push_back(b_matrix_final[k]);
```

```
        residues_imag.push_back(b_matrix_final[num_poles + k]);
        std::complex<double> temp(b_matrix_final[k], b_matrix_final[num_poles + k])
            ;
        residues.push_back(temp);
      }

      // poles are kept from final iteration; that is, the final computed zeroes
          become the poles of our final fit

      // delete stuff...
      delete[] a_matrix_final;
      delete[] a_matrix_final_real;
      delete[] b_matrix_final;
    }

    // print poles to file
    if (pole_write)
    {
      pole_write << "Iteration " << iteration;
      for (auto x : poles_guess)
      {
        pole_write << ", " << x.real() << '+' << x.imag() << 'i';
      }
      pole_write << '\n';
    }

    // clear stuff for next iteration (if there is one)
    real_poles.clear();
    comp_poles.clear();
    all_zeroes.clear();
    keep_zeroes.clear();

    // FREE DYNAMIC MEMORY FOR NEXT LOOP
    delete[] a_matrix;
    delete[] b_matrix;
    delete[] a_matrix_real;
    delete[] Az_real;
    delete[] bz_real;
    delete[] c_real;
    delete[] bc_real;
    delete[] H_real;
    delete[] real_zeroes_real;
    delete[] real_zeroes_imag;
    delete[] Az_comp;
    delete[] bz_comp;
    delete[] c_comp;
    delete[] bc_comp;
    delete[] H_comp;
    delete[] comp_zeroes_real;
    delete[] comp_zeroes_imag;
  }

  in_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
  if (log_file)
    log_file << std::ctime(&in_time) << "\tFinished Vector Fitting Algorithm!
        Writing out results...\n";

  // open file for outputting poles of final results (for testing; can include in a
      flag later on whether to do this or not...)
  if (vflog)
  {
    try
    {
```

```cpp
      std::filesystem::remove("results.txt");
    }
    catch (...)
    {
    }
  }
  std::ofstream results_write;
  if (vflog)
    results_write.open("results.txt", std::ios::app | std::ios::out);

  // get final stuff to take back to RC model
  // need to scale poles and residues since the algorithm was normalised to range of 0 to 1 for frequency.
  // poles and residues will be in terms of 2 * PI * f (angular frequency), not Hz
  if (results_write)
    results_write << "\n\t=========================\tPOLES OF FIT (RESCALED by " << max_freq << ", in 2PIf); # poles: " << poles_guess.size() << "\t=============\n";
  for (int k = 0; k < poles_guess.size(); k++)
  {
    poles_guess[k] = poles_guess[k] * max_freq;
    if (results_write)
      results_write << poles_guess[k].real() << " + " << poles_guess[k].imag() << " i\n";
  }
  if (results_write)
    results_write << "\n\t============================\tRESIDUES OF FIT (RESCALED, in 2PIf); # residues: " << residues.size() << "\t==============\n";
  for (int k = 0; k < residues.size(); k++)
  {
    residues[k] = residues[k] * max_freq;
    if (results_write)
      results_write << residues[k].real() << " + " << residues[k].imag() << "i\n";
  }
  if (results_write)
    results_write << "\n\t==============================\tREMAINDER OF FIT:\t======================\n"
                  << remainder << '\n';

  // finalise and close files
  auto end = std::chrono::system_clock::now();
  std::chrono::duration<double> time_to_run = end - start;
  in_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
  if (log_file)
    log_file << std::ctime(&in_time) << "\tdo_vector_fitting() finished, it took " << time_to_run.count() << " seconds to run!\n\nClosing log file...\n";
  if (results_write)
    results_write.close();
  if (pole_write)
    pole_write.close();
  if (log_file)
    log_file.close();
  if (resid_log)
    resid_log.close();

  // (viii) return fit to RC model via pass by reference and exit VF
  p = poles_guess;
  r = residues;
  rem = remainder;

  return 1;
}
```

Appendix A5

rc_model.cc File

This appendix gives the code for the Gnucap RC model plugin itself, which is compiled into the `rc.so` dynamic library for use with Gnucap as described in this report. It, along with the associated `vf.h` header, is available on the project GitHub at [41].

```
/*
  RECURSIVE CONVOLUTION COMPANION MODEL FOR GNUCAP
  USES VECTOR FITTING ALGORITHM TO FIT S-PARAMETER DATA
  AND ALLOW TRANSIENT SIMULATION OF 1-PORT, LTI S-PARAMETER BLOCKS

  Created by: Sean Higginbotham for M.A.I project
              Supervisor: Dr. Justin King
              Department of Electronic and Electrical Engineering,
              Trinity College Dublin, Ireland, September 2023 - April 2024

  Copyright (C) 2024 Sean Higginbotham
  Author: Sean Higginbotham <higginbs@tcd.ie>

  This program is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// for RC model
#include "globals.h"
#include "e_storag.h"

// for VF
#include "vf.h"

namespace
{

  // ================================================
  // = RECURSIVE CONVOLUTION COMPANION MODEL PLUGIN =
  // ================================================

  // THE FPOLY(s) AND CPOLY FOR RC MODEL AND HOW THEY'RE CONSTRUCTED AS TAYLOR
     SERIES
```

```

// y = (1/Gc) * i_c + (1/Gc) * i[n]                              Impements the port voltage constitutive relation v[n], is set in do_tr()
//      y.f0 = (1/Gc)*i_c
//      y.f1 = 1/Gc
//      y.x = i[n] = Gc * involts() - i_c

// m = -i_c + Gc * v[n]                                          Implements the port current constitutive relation i[n], is set in do_tr()
//      m.c0 = -i_c
//      m.c1 = Gc
//      m.x = v[n] = involts()

// i = i_c = (2 * h_n) / (sqrt(Zref) * (1 + factor))            Implements the i_c RC model internal current, is set in tr_advance()
//      i.f0 = 0
//      i.f1 = 2 / (sqrt(Zref) * (1 + factor))
//      i.x = h_n

class RC_MODEL : public STORAGE
{
private:
  PARAMETER<int> nump;   // # initial poles to use in VF
  PARAMETER<int> numi;   // # iterations to use in VF
  PARAMETER<bool> vflog; // flag for if want to print logs during the VF (logs are put into 'vf_log.txt', and VF results are put into 'results.txt', and each pole iteration is put into 'pole_guess.txt'')
  PARAMETER<bool> rclog; // flag for if want to print logs during the transient sims (logs are put into 'tr_write.txt')
protected:
  explicit RC_MODEL(const RC_MODEL &p) : STORAGE(p) {} // copy constructor
public:
  explicit RC_MODEL() : STORAGE() {} // constructor

  // parameter function overrides
  int param_count() const { return (4 + STORAGE::param_count()); } // # params for this device
  bool param_is_printable(int) const;
  std::string param_name(int) const;
  std::string param_name(int, int) const;
  std::string param_value(int) const;
  void set_param_by_index(int i, std::string &, int offset);

protected:
  char id_letter() const { return 'p'; }
  std::string value_name() const { return "Zref"; } // pass in Zref
  std::string dev_type() const { return "RCmodel"; }
  int max_nodes() const { return 2; }
  int min_nodes() const { return 2; }
  int matrix_nodes() const { return 2; }
  int net_nodes() const { return 2; }
  bool has_iv_probe() const { return 2; }

  // device will be passive cause in this case we're only considering
  // linear, passive (R,L,C) elements in the S-param block

  void tr_iwant_matrix() { tr_iwant_matrix_passive(); }
  void ac_iwant_matrix() { ac_iwant_matrix_passive(); }
  void tr_load() { tr_load_passive(); }
  void tr_unload() { tr_unload_passive(); }

  double tr_involts() const { return tr_outvolts(); }
  double tr_involts_limited() const { return tr_outvolts_limited(); }
  COMPLEX ac_involts() const { return ac_outvolts(); }

  // functions which explicitly implement the RC companion model
  bool do_tr();
  void tr_advance();    // update i_c[n] here using _time, _dt, etc...
  void tr_begin();      // for initialisations
  void precalc_first(); // For doing vector fitting and evaluating parameters
  void tr_accept();     // do storage of history terms here

  // not currently implemented, see how it's done in d_cap.cc
  // can seemingly add ability to probe specific internal states of the device
  // during sims; could use to return i_c[n], etc...
  // double tr_probe_num(const std::string& x) const;

  std::string port_name(int i) const
  {
    assert(i >= 0);
    assert(i < 2);
    static std::string names[] = {"p", "n"};
    return names[i];
  }

  CARD *clone() const { return new RC_MODEL(*this); }

  // FOR STORING DATA FROM VF ALGORITHM

  std::vector<std::complex<double>> vf_poles;
  std::vector<std::complex<double>> vf_residues;
  double vf_remainder;

  // default parameter values (not including Zref)
  const int nump_default = 35;
  const int numi_default = 450;
  const bool vflog_default = 0;
  const bool rclog_default = 0;

  // actual values evaluated
  int num_p;
  int num_i;
  bool vf_log;
  bool rc_log;

  // RC MODEL RELATED PARAMS

  double Zref = 50;                           // Impedance the block is referenced to (Note this is NOT the characteristic impedance of the port TLs)
  double Gc;                                  // parallel conductance of the RC model
  double i_c;                                 // independent current source of the RC model
  double factor;                              // the (K + 2 * real(sum(rk * lambdak))) term that is repeated in the RC constitutive relations (in Gc and i_c)
  double h_n;                                 // history term ( is actually 2* real(h[n]), where h[n] == hist_sum )
  std::vector<std::complex<double>> dk_store; // values of the dk[n-1] history terms (must store as vector since will have an entry for each pole)
  double a_store[2] = {0, 0};                 // for storing a[n-1] and a[n-2]
  double a_n;                                 // value of incident wave a[n] at current time step;
  std::vector<std::complex<double>> dk_n;     // value of history term dk[n] at current time step

  std::vector<std::complex<double>> alpha_k, lambda_k, mu_k, nu_k; // coeficients for 2nd order approx (for time steps > 2)
  std::vector<std::complex<double>> lambda_k_1, mu_k_1;            // 1st order approx (for time steps 1 and 2)
  std::complex<double> res_lambda_sum;                             // dot product of residues and 2nd order lambda
  std::complex<double> res_lambda1_sum;                            // dot product of residues and 1st order lambda
  std::complex<double> hist_sum;                                   // dot product of residues with dk_store and a_store (i.e., the h[n] history term)

  std::ofstream tr_data; // debugging file for RC model
  bool accepted;         // to prevent tr_accept pushing history terms TWICE in a given time step (which messes up the model, obviously!)

  int time_step; // current time step (indexed from 0, so time_step = 0 means the 1st time step of the transient run)
};

// =======================================================
// = SET UP PARAMETERS WANNA READ IN WHEN INSTANTIATING =
// ========================================================

bool RC_MODEL::param_is_printable(int I) const
{
  switch (RC_MODEL::param_count() - 1 - I)
  {
  case 0:
    return nump.has_hard_value();
  case 1:
    return numi.has_hard_value();
  case 2:
    return vflog.has_hard_value();
  case 3:
    return rclog.has_hard_value();
  default:
    return STORAGE::param_is_printable(I);
  }
}
std::string RC_MODEL::param_name(int I) const
{
  switch (RC_MODEL::param_count() - 1 - I)
  {
  case 0:
    return "nump";
  case 1:
    return "numi";
  case 2:
    return "vflog";
  case 3:
    return "rclog";
  default:
    return STORAGE::param_name(I);
  }
}
std::string RC_MODEL::param_name(int I, int j) const
{
  if (j == 0)
    return param_name(I);
  else if (I >= STORAGE::param_count())
  {
    switch (RC_MODEL::param_count() - 1 - I)
    {
    case 0:
      return (j == 1) ? "nump" : "";
    case 1:
      return (j == 1) ? "numi" : "";
    case 2:
      return (j == 1) ? "vflog" : "";
    case 3:
      return (j == 1) ? "rclog" : "";
    default:
      return "";
    }
  }
  else
    return STORAGE::param_name(I, j);
}
std::string RC_MODEL::param_value(int I) const
{
  switch (RC_MODEL::param_count() - 1 - I)
  {
  case 0:
    return nump.string();
  case 1:
    return numi.string();
  case 2:
    return vflog.string();
  case 3:
    return rclog.string();
  default:
    return STORAGE::param_value(I);
  }
}
void RC_MODEL::set_param_by_index(int I, std::string &Value, int Offset)
{
  switch (RC_MODEL::param_count() - 1 - I)
  {
  case 0:
    nump = Value;
    break;
  case 1:
    numi = Value;
    break;
  case 2:
    vflog = Value;
    break;
  case 3:
    rclog = Value;
    break;
  default:
    STORAGE::set_param_by_index(I, Value, Offset);
    break;
  }
}

// ===================================
// = OVVERIDE FUNCTIONS FOR RC MODEL =
// ===================================

// is run when performing any command on the device
void RC_MODEL::precalc_first()
{
  STORAGE::precalc_first(); // won't work if don't call base class version first!

  // we must first evaluate values of parameters provided with the instance, as
      otherwise they'd stay as defaults
  num_p = nump.e_val(nump_default, scope());
  num_i = numi.e_val(numi_default, scope());
  vf_log = vflog.e_val(vflog_default, scope());
  rc_log = rclog.e_val(rclog_default, scope());

  // having vector fitting in here seems to prevent issues (i.e., incorrect
      computatations) during transient sims
  // probably means that the model (device in circuit) needs to be replaced if we
       change the input s_param_data.txt
  assert(do_vector_fitting(vf_poles, vf_residues, vf_remainder, num_p, num_i,
      vf_log));
}

// initialise relevant values and set up; is run once at beginning of transient
    sim
void RC_MODEL::tr_begin()
{

  STORAGE::tr_begin();

  // open log file for debugging
  try
  {
    std::filesystem::remove("tr_write.txt");
  }
  catch (...)
  {
  }
  if (rc_log && !tr_data)
    tr_data.open("tr_write.txt", std::ios::app | std::ios::out);
  if (tr_data)
    tr_data << "Opening file in tr_begin()...\n";

  // =====================
  // = INITIALIASE VALUES =
  // =====================

  Zref = value();
  i_c = 0;
  time_step = 0;
  accepted = 0; // for some reason tr_accept() is entered twice, but wanna
      prevent this!

  _i[0].f0 = 0.;
  _i[0].f1 = 0.;
  _i[0].x = 0.;
  _y[0].f0 = 0.;
  _m0.c0 = 0.;

  // initialise the history terms
  h_n = 0.;
  std::vector<std::complex<double>> zeroes(vf_poles.size(), (0., 0.));
  dk_store = zeroes; // initialise to zeroes
  a_store[0] = 0.;   // a[n-1]
  a_store[1] = 0.;   // a[n-2]

  if (tr_data)
    tr_data << "Exiting tr_begin()...\n";
}

```

```
// is done at start of each time step; here is where the time-step dependant
    parameters are computed
// (except for the first time step which is done in do_tr())
void RC_MODEL::tr_advance()
{
  // we close tr_data at the end of each time step, so re-open again if needed.
  if (rc_log && !tr_data)
    tr_data.open("tr_write.txt", std::ios::app | std::ios::out);
  if (tr_data)
    tr_data << "Entered tr_advance()...\n";
  time_step++;
  STORAGE::tr_advance();

  // clear vectors and things for this iteration
  alpha_k.clear();
  lambda_k.clear();
  mu_k.clear();
  nu_k.clear();
  if (time_step == 1)
  {
    lambda_k_1.clear();
    mu_k_1.clear();
  }                           // 2nd time step (time_step == 1)
  res_lambda_sum = 0. * one; // set to 0
  if (time_step == 1)
    res_lambda1_sum = 0. * one;
  hist_sum = 0. * one; // set to 0

  // make sure VF solution is actually usable!
  assert(vf_poles.size() != 0);
  assert(vf_poles.size() == vf_residues.size());

  // compute alpha, lambda, mu, and nu for this time step
  for (int i = 0; i < vf_poles.size(); i++) // # poles == # residues
  {
    std::complex<double> x = vf_poles[i];    // pole pk
    std::complex<double> r = vf_residues[i]; // residue rk

    std::complex<double> lambda_1, mu_1;
    std::complex<double> alpha = std::exp(x * _dt);
    if (time_step == 1)
    {
      lambda_1 = -(one / x) * (one + (one - alpha) / (x * _dt));
      mu_1 = -(one / x) * ((alpha - one) / (x * _dt) - alpha);
    }
    std::complex<double> lambda = -(one / x) * ((one - alpha) / std::pow((-x *
        _dt), 2) - (3. * one - alpha) / (-2. * x * _dt) + one);
    std::complex<double> mu = -(one / x) * ((-2. * one) * (one - alpha) / std::
        pow(-x * _dt, 2) + 2. * one / (-x * _dt) - alpha);
    std::complex<double> nu = -(one / x) * ((one - alpha) / std::pow((-x * _dt),
        2) - (one + alpha) / (-2. * x * _dt));

    res_lambda_sum += r * lambda;
    res_lambda1_sum += r * lambda_1;

    // write out log data
    if (time_step == 1)
    {
      tr_data << "\np[" << i + 1 << "] = " << x;
      tr_data << "\nr[" << i + 1 << "] =" << r;
      tr_data << "\nalpha[" << i + 1 << "] = " << alpha;
      tr_data << "\nlambda1[" << i + 1 << "] = " << lambda_1;
      tr_data << "\nmu1[" << i + 1 << "] = " << mu_1;
```

```
        if (tr_data)
          tr_data << "\nhist_sum[" << time_step + 1 << "] = " << hist_sum << " + (" << r << " * ( " << alpha << " * " << dk_store[i] << " + " << mu_1 << " * " << a_store[0] << " )) = ";
        hist_sum += (r * (alpha * dk_store[i] + mu_1 * a_store[0]));
        if (tr_data)
          tr_data << hist_sum;
      }
      else
      {
        tr_data << "\np[" << i + 1 << "] = " << x;
        tr_data << "\nr[" << i + 1 << "] =" << r;
        tr_data << "\nalpha[" << i + 1 << "] = " << alpha;
        tr_data << "\nlambda2[" << i + 1 << "] = " << lambda;
        tr_data << "\nmu2[" << i + 1 << "] = " << mu;
        tr_data << "\nnu2[" << i + 1 << "] = " << nu;
        if (tr_data)
          tr_data << "\nhist_sum[" << time_step + 1 << "] = " << hist_sum << " + (" << r << " * ( " << alpha << " * " << dk_store[i] << " + " << mu << " * " << a_store[0] << " + " << nu << " * " << a_store[1] << " )) = ";
        hist_sum += (r * (alpha * dk_store[i] + mu * a_store[0] + nu * a_store[1]));
        if (tr_data)
          tr_data << hist_sum << "\n";
      }
      alpha_k.push_back(alpha);
      lambda_k.push_back(lambda);
      if (time_step == 1)
      {
        lambda_k_1.push_back(lambda_1);
        mu_k_1.push_back(mu_1);
      }
      mu_k.push_back(mu);
      nu_k.push_back(nu);
    }

    // ensure we don't end up dividing by 0
    assert(res_lambda_sum.real() != 0);
    if (time_step == 1)
      assert(res_lambda1_sum.real() != 0);
    assert(vf_remainder != 0);

    factor = (vf_remainder + 2 * (res_lambda_sum.real()));
    double factor_1 = (vf_remainder + 2 * (res_lambda1_sum.real())); // 1st order for 2nd time step

    // calculate Gc term
    Gc = (1. - factor) / (Zref * (1. + factor));

    // - calculate h[n] term
    h_n = (2 * (hist_sum.real()));

    // calculate i_c term
    if (time_step == 1)
      _i[0].f1 = 2 / (std::sqrt(Zref) * (1. + factor_1));
    else
      _i[0].f1 = 2 / (std::sqrt(Zref) * (1. + factor));
    _i[0].x = h_n;
    i_c = _i[0].f1 * _i[0].x;
    if (tr_data && time_step == 1)
    {
      tr_data << "Time step " << time_step + 1 << '\n';
      tr_data << "\tfactor = " << factor_1 << "\n\tGc = " << Gc << "\n\th_n = " <<
          h_n << "\n\ti_c = " << i_c << '\n';
    }
    else if (tr_data)
    {
      tr_data << "Time step " << time_step + 1 << '\n';
      tr_data << "\tfactor = " << factor << "\n\tGc = " << Gc << "\n\th_n = " <<
          h_n << "\n\ti_c = " << i_c << '\n';
    }

    if (tr_data)
      tr_data << "Exiting tr_advance()...\n";
  }

  // do_tr is repeated multiple times till reach satisfied convergence in MNA
      solver
  bool RC_MODEL::do_tr()
  {

    if (tr_data)
      tr_data << "Entering do_tr()....\n";
    accepted = 0; // reset for this time step

    assert(_y[0] == _y[0]);

    // for the first time step, tr_advance() is not entered, so need to do the
        updating here
    if (time_step == 0) // first time step (time_step == 0)
    {
      double delta = 0.1e-3; // assign some arbitrary initial time step of 100ns
          since initial time step can't be accessed on first step?

      // clear vectors and things for this iteration (note all these will be filled
           with 1st order stuff as appropriate so no need to discern like in
          tr_advance())
      alpha_k.clear();
      lambda_k.clear();
      mu_k.clear();
      nu_k.clear();
      res_lambda_sum = 0. * one; // set to 0
      hist_sum = 0. * one;       // set to 0

      assert(vf_poles.size() != 0);
      assert(vf_poles.size() == vf_residues.size());

      // compute alpha, lambda first time step; will be 1st order. Note that on
          first time step only lambda is used (mu and nu are not)
      for (int i = 0; i < vf_poles.size(); i++) // # poles == # residues
      {
        std::complex<double> x = vf_poles[i];    // pole pk
        std::complex<double> r = vf_residues[i]; // residue rk

        std::complex<double> alpha = std::exp(x * delta);
        std::complex<double> lambda = -(one / x) * (one + (one - alpha) / (x *
            delta)); // 1st order

        res_lambda_sum += r * lambda;

        hist_sum += 0.;

        alpha_k.push_back(alpha);
        lambda_k.push_back(lambda);
      }

      assert(res_lambda_sum.real() != 0);
      assert(vf_remainder != 0);

      factor = (vf_remainder + 2 * (res_lambda_sum.real()));

      // calculate Gc term
      Gc = (1. - factor) / (Zref * (1. + factor));

      // calculate h[n] term; will be 0 on first time step regardless
      h_n = 0.;

      // i_c term is also hence zero on first time step so no need to update from
          default value

      if (tr_data)
      {
        tr_data << "Time step " << time_step + 1 << '\n';
        tr_data << "\tfactor = " << factor << "\n\tGc = " << Gc << "\n\th_n = " <<
            h_n << "\n\ti_c = " << i_c << '\n';
      }
    }

    // calculate constitutive relations for MNA solver

    _y[0].f1 = (1 / Gc);
    _y[0].f0 = (1 / Gc) * i_c;
    _y[0].x = Gc * tr_involts_limited() - i_c;

    assert(converged());
    store_values();
    q_load();

    _m0.c1 = 1 / _y[0].f1;
    _m0.c0 = -i_c;
    _m0.x = tr_involts_limited();

    q_accept(); // queue the tr_accept() function
    return converged();
  }

  // here is where we calculate the terms that make up the history term and then
      propagate to next time step
  void RC_MODEL::tr_accept()
  {
    if (!accepted)
    {
      if (tr_data)
        tr_data << "Entered tr_accept()...\n";

      dk_n.clear();

      // these SHOULD be the solutions to the current time step
      double vp = tr_involts_limited();            // port voltage from MNA soln (I
           hope...)
      double ip = Gc * tr_involts_limited() - i_c; // port current entering block (
          I hope...)

      // calculate incident wave a[n]
      a_n = (vp + Zref * ip) / (2 * std::sqrt(Zref));

      // reflected wave b[n] not required, but could be computed here out of
          interest, if desired!

      // print a[n], a[n-1], and a[n-2] for debugging
      if (tr_data)
      {
        tr_data << "Time step " << time_step + 1 << "\n\t";
        tr_data << "a[" << time_step + 1 << "] = " << a_n << "\ta[" << time_step <<
             "] = " << a_store[0] << "\ta[" << time_step - 1 << "] = " << a_store[1]
             << '\n';
      }

      if (time_step == 0) // 1st time step
      {
        if (tr_data)
          tr_data << "\tdk[" << time_step + 1 << "] = \n";

        // calculate current dk[n] term
        for (int i = 0; i < vf_poles.size(); i++)
        {
          std::complex<double> dk = lambda_k[i] * a_n; // should be 1st order
              lambda_k since hasn't been cleared since 1st do_tr()
          dk_n.push_back(dk);

          if (tr_data)
            tr_data << "\tdk[" << time_step + 1 << "](" << i << ") = " << dk << '\n
               ';
        }
      }
      else if (time_step == 1) // 2nd time step
      {
        if (tr_data)
          tr_data << "\tdk[" << time_step + 1 << "] = \n";

        // calculate current dk[n] term
        for (int i = 0; i < vf_poles.size(); i++)
        {
          std::complex<double> dk;
          dk = alpha_k[i] * dk_store[i] + lambda_k_1[i] * a_n + mu_k_1[i] * a_store
              [0];
          dk_n.push_back(dk);

          if (tr_data)
            tr_data << "\tdk[" << time_step + 1 << "](" << i << ") = " << dk << '\n
               ';
        }
      }
      else if (time_step > 1) // all other time steps
      {
        if (tr_data)
          tr_data << "\tdk[" << time_step + 1 << "] = \n";

        // calculate current dk[n] term
        for (int i = 0; i < vf_poles.size(); i++)
        {
          std::complex<double> dk;
          dk = alpha_k[i] * dk_store[i] + lambda_k[i] * a_n + mu_k[i] * a_store[0]
              + nu_k[i] * a_store[1];
          dk_n.push_back(dk);

          if (tr_data)
            tr_data << "\tdk[" << time_step + 1 << "](" << i << ") = " << dk << '\n
               ';
        }
      }

      // store a[n] and dk[n] into previous ones to use for next iteration in
         tr_advance()
      dk_store = dk_n;          // dk[n]   ----->  dk[n-1]
      a_store[1] = a_store[0]; // a[n-1]  ----->  a[n-2]
      a_store[0] = a_n;        // a[n]    ----->  a[n-1]

      if (tr_data)
        tr_data << "Exiting tr_accept()...\n";
    }
    accepted = 1;
    if (tr_data)
      tr_data.close();
  }

  RC_MODEL p1;
  DISPATCHER<CARD>::INSTALL d1(&device_dispatcher, "p|rcm", &p1);
}
```

Appendix A6

`tasks.json` File for Compiling the RC Model with VSCode

This appendix provides the equivalent Visual Studio Code compilation file for the command in listing 4.1. It compiles the `rc_model.cc` file.

```
{
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: g++ build active file",
      "command": "g++",
      "args": [
        "-shared",
        "-fPIC",
        "${file}",
        "~/code/lapack-3.12.0/liblapack.so",
        "~/code/lapack-3.12.0/libtmglib.so",
        "~/code/lapack-3.12.0/liblapacke.so",
        "~/code/lapack-3.12.0/librefblas.so",
        "~/code/lapack-3.12.0/libcblas.so",
        "-o",
        "${fileDirname}/rc.so",
        "-I/usr/local/include/gnucap",
        "-I", "~/code/lapack-3.12.0/LAPACKE/include",
        "-I", "~/code/lapack-3.12.0/CBLAS/include",
        "-L/usr/local/lib/",
        "-L", "~/code/lapack-3.12.0",
        "-l", "lapack",
        "-l", "lapacke",
        "-l", "gfortran",
        "-l", "cblas",
        "-l", "tmglib",
        "-l", "refblas"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "detail": "Task generated by Debugger."
    }
  ],
  "version": "2.0.0"
}
```

Appendix A7

`rlc_z11.txt` Simulation Data

This appendix gives the frequency domain data vector, $\bar{Z}_{11}(j\omega)$, for the RLC circuit in Fig. 5.1. It was collected using an ADS simulation.

```
! RLC circuit Z11(jw) (freqs in Hz, real and imag part of S11(jw))
0 0 0
0.0159 0.043308792 0.203551321
0.0318 0.21691974 0.412147505
0.0477 0.629811057 0.482855143
0.0637 0.997506234 0.049875312
0.0796 0.8 -0.4
0.0955 0.516944285 -0.499712809
0.1114 0.34244182 -0.474526522
0.1273 0.240927571 -0.427646439
0.1432 0.178614743 -0.383029394
0.1592 0.137931034 -0.344827586
0.1751 0.109931043 -0.312803787
0.191 0.089818118 -0.28592101
0.2069 0.074861904 -0.263168386
0.2228 0.06342139 -0.243719341
0.2387 0.054462935 -0.226928896
0.2546 0.047308935 -0.212298845
0.2706 0.041499794 -0.199443127
0.2865 0.036714547 -0.188060067
0.3024 0.032723314 -0.177911491
0.3183 0.029357798 -0.168807339
0.3342 0.02649239 -0.160594343
0.3501 0.02403173 -0.15314766
0.3661 0.021902322 -0.146364649
0.382 0.020046748 -0.140160179
0.3979 0.018419598 -0.134463069
0.4138 0.016984566 -0.129213353
0.4297 0.015712328 -0.124360164
0.4456 0.014578984 -0.119860072
0.4615 0.01356489 -0.115675771
0.4775 0.012653779 -0.111775044
0.4934 0.011832077 -0.108129917
0.5093 0.01108839 -0.104715988
0.5252 0.010413092 -0.101511868
0.5411 0.009798002 -0.098498736
0.557 0.009236134 -0.095659959
0.573 0.008721491 -0.092980788
0.5889 0.008248902 -0.090448091
0.6048 0.007813885 -0.088050146
0.6207 0.007412545 -0.085776447
0.6366 0.007041479 -0.08361756
0.6525 0.006697705 -0.081564979
0.6685 0.006378602 -0.079611024
```

```
0.6844 0.006081855 -0.077748736
0.7003 0.005805417 -0.075971799
0.7162 0.00554747 -0.074274463
0.7321 0.005306396 -0.072651486
0.748 0.00508075 -0.071098076
0.7639 0.00486924 -0.069609846
0.7799 0.004670706 -0.06818277
0.7958 0.004484104 -0.066813147
0.8117 0.004308495 -0.065497571
0.8276 0.00414303 -0.064232898
0.8435 0.00398694 -0.063016226
0.8594 0.00383953 -0.061844867
0.8754 0.003700164 -0.060716333
0.8913 0.003568268 -0.059628315
0.9072 0.003443316 -0.058578665
0.9231 0.003324828 -0.057565385
0.939 0.003212364 -0.056586616
0.9549 0.003105523 -0.055640622
0.9708 0.003003935 -0.054725781
0.9868 0.00290726 -0.053840576
1.0027 0.002815186 -0.052983591
1.0186 0.002727426 -0.052153493
1.0345 0.002643713 -0.051349037
1.0504 0.002563802 -0.05056905
1.0663 0.002487466 -0.049812432
1.0823 0.002414494 -0.049078147
1.0982 0.002344692 -0.048365219
1.1141 0.002277878 -0.047672729
1.13 0.002213883 -0.04699981
1.1459 0.002152552 -0.046345643
1.1618 0.002093738 -0.045709454
1.1777 0.002037305 -0.045090511
1.1937 0.001983126 -0.044488122
1.2096 0.001931082 -0.043901631
1.2255 0.001881063 -0.043330416
1.2414 0.001832965 -0.042773887
1.2573 0.001786691 -0.042231485
1.2732 0.001742148 -0.041702678
1.2892 0.001699253 -0.041186961
1.3051 0.001657925 -0.040683854
1.321 0.001618087 -0.040192898
1.3369 0.00157967 -0.03971366
1.3528 0.001542606 -0.039245724
1.3687 0.001506833 -0.038788695
1.3846 0.001472292 -0.038342195
1.4006 0.001438925 -0.037905864
1.4165 0.001406681 -0.03747936
1.4324 0.00137551 -0.037062352
1.4483 0.001345364 -0.036654528
1.4642 0.0013162 -0.036255587
1.4801 0.001287974 -0.035865242
1.4961 0.001260648 -0.035483218
1.512 0.001234183 -0.035109252
1.5279 0.001208543 -0.03474309
1.5438 0.001183694 -0.034384491
1.5597 0.001159605 -0.034033224
1.5756 0.001136244 -0.033689065
1.5915 0.001113583 -0.0333518
1.6075 0.001091593 -0.033021225
1.6234 0.001070249 -0.032697142
1.6393 0.001049525 -0.032379362
1.6552 0.001029397 -0.032067703
1.6711 0.001009844 -0.031761988
```

```
1.687 0.000990842 -0.031462051
1.703 0.000972373 -0.031167728
1.7189 0.000954415 -0.030878863
1.7348 0.000936951 -0.030595306
1.7507 0.000919961 -0.030316912
1.7666 0.00090343 -0.03004354
1.7825 0.000887341 -0.029775057
1.7985 0.000871679 -0.029511331
1.8144 0.000856427 -0.029252239
1.8303 0.000841572 -0.028997658
1.8462 0.000827101 -0.028747472
1.8621 0.000813 -0.028501567
1.878 0.000799257 -0.028259836
1.8939 0.00078586 -0.028022172
1.9099 0.000772796 -0.027788474
1.9258 0.000760056 -0.027558643
1.9417 0.000747629 -0.027332584
1.9576 0.000735504 -0.027110204
1.9735 0.000723672 -0.026891416
1.9894 0.000712123 -0.026676131
2.0054 0.000700849 -0.026464268
2.0213 0.00068984 -0.026255744
2.0372 0.000679089 -0.026050482
2.0531 0.000668587 -0.025848405
2.069 0.000658327 -0.025649441
2.0849 0.000648302 -0.025453517
2.1008 0.000638504 -0.025260564
2.1168 0.000628926 -0.025070516
2.1327 0.000619563 -0.024883306
2.1486 0.000610407 -0.024698873
2.1645 0.000601453 -0.024517155
2.1804 0.000592694 -0.024338091
2.1963 0.000584125 -0.024161626
2.2123 0.000575741 -0.023987701
2.2282 0.000567536 -0.023816263
2.2441 0.000559506 -0.023647259
2.26 0.000551645 -0.023480637
2.2759 0.000543948 -0.023316348
2.2918 0.000536411 -0.023154342
2.3077 0.00052903 -0.022994572
2.3237 0.000521801 -0.022836993
2.3396 0.000514718 -0.022681559
2.3555 0.000507779 -0.022528228
2.3714 0.000500979 -0.022376956
2.3873 0.000494315 -0.022227702
2.4032 0.000487783 -0.022080427
2.4192 0.00048138 -0.021935091
2.4351 0.000475102 -0.021791657
2.451 0.000468946 -0.021650086
2.4669 0.000462909 -0.021510344
2.4828 0.000456988 -0.021372394
2.4987 0.00045118 -0.021236202
2.5146 0.000445482 -0.021101736
2.5306 0.000439891 -0.020968962
2.5465 0.000434405 -0.02083785
2.5624 0.00042902 -0.020708366
2.5783 0.000423736 -0.020580483
2.5942 0.000418548 -0.020454169
2.6101 0.000413455 -0.020329397
2.6261 0.000408455 -0.020206138
2.642 0.000403545 -0.020084366
2.6579 0.000398722 -0.019964052
2.6738 0.000393986 -0.019845172
```

```
2.6897 0.000389334 -0.019727699
2.7056 0.000384763 -0.019611609
2.7215 0.000380273 -0.019496877
2.7375 0.000375861 -0.019383481
2.7534 0.000371525 -0.019271396
2.7693 0.000367263 -0.0191606
2.7852 0.000363075 -0.019051071
2.8011 0.000358958 -0.018942788
2.817 0.000354911 -0.018835728
2.833 0.000350931 -0.018729872
2.8489 0.000347018 -0.0186252
2.8648 0.000343171 -0.018521691
2.8807 0.000339387 -0.018419327
2.8966 0.000335665 -0.018318088
2.9125 0.000332004 -0.018217955
2.9285 0.000328403 -0.018118912
2.9444 0.00032486 -0.01802094
2.9603 0.000321374 -0.017924022
2.9762 0.000317944 -0.017828142
2.9921 0.000314568 -0.017733281
3.008 0.000311246 -0.017639425
3.0239 0.000307977 -0.017546557
3.0399 0.000304758 -0.017454663
3.0558 0.00030159 -0.017363726
3.0717 0.000298471 -0.017273731
3.0876 0.0002954 -0.017184665
3.1035 0.000292376 -0.017096513
3.1194 0.000289399 -0.01700926
3.1354 0.000286466 -0.016922894
3.1513 0.000283578 -0.016837401
3.1672 0.000280734 -0.016752767
3.1831 0.000277932 -0.01666898
3.199 0.000275172 -0.016586027
3.2149 0.000272453 -0.016503895
3.2308 0.000269774 -0.016422573
3.2468 0.000267134 -0.016342049
3.2627 0.000264533 -0.016262311
3.2786 0.000261969 -0.016183347
3.2945 0.000259443 -0.016105146
3.3104 0.000256953 -0.016027698
3.3263 0.000254499 -0.01595099
3.3423 0.00025208 -0.015875014
3.3582 0.000249695 -0.015799758
3.3741 0.000247343 -0.015725213
3.39 0.000245025 -0.015651367
3.4059 0.00024274 -0.015578212
3.4218 0.000240486 -0.015505738
3.4377 0.000238263 -0.015433935
3.4537 0.000236071 -0.015362794
3.4696 0.000233909 -0.015292306
3.4855 0.000231777 -0.015222462
3.5014 0.000229674 -0.015153253
3.5173 0.000227599 -0.015084671
3.5332 0.000225552 -0.015016707
3.5492 0.000223533 -0.014949352
3.5651 0.000221541 -0.014882599
3.581 0.000219575 -0.01481644
3.5969 0.000217635 -0.014750866
3.6128 0.000215721 -0.01468587
3.6287 0.000213832 -0.014621445
3.6446 0.000211968 -0.014557582
3.6606 0.000210128 -0.014494275
3.6765 0.000208312 -0.014431516
```

```
3.6924 0.000206519 -0.014369298
3.7083 0.00020475 -0.014307615
3.7242 0.000203003 -0.014246459
3.7401 0.000201278 -0.014185823
3.7561 0.000199575 -0.014125702
3.772 0.000197894 -0.014066088
3.7879 0.000196234 -0.014006975
3.8038 0.000194595 -0.013948357
3.8197 0.000192976 -0.013890228
3.8356 0.000191377 -0.013832581
3.8515 0.000189798 -0.013775411
3.8675 0.000188238 -0.013718711
3.8834 0.000186698 -0.013662476
3.8993 0.000185177 -0.013606701
3.9152 0.000183674 -0.013551379
3.9311 0.000182189 -0.013496505
3.947 0.000180722 -0.013442074
3.963 0.000179273 -0.01338808
3.9789 0.000177841 -0.013334518
3.9948 0.000176426 -0.013281383
4.0107 0.000175028 -0.01322867
4.0266 0.000173647 -0.013176374
4.0425 0.000172282 -0.013124489
4.0585 0.000170933 -0.013073012
4.0744 0.0001696 -0.013021937
4.0903 0.000168282 -0.012971259
4.1062 0.000166979 -0.012920975
4.1221 0.000165692 -0.012871078
4.138 0.00016442 -0.012821566
4.1539 0.000163162 -0.012772433
4.1699 0.000161918 -0.012723676
4.1858 0.000160689 -0.012675289
4.2017 0.000159473 -0.012627269
4.2176 0.000158272 -0.012579611
4.2335 0.000157084 -0.012532312
4.2494 0.000155909 -0.012485367
4.2654 0.000154747 -0.012438773
4.2813 0.000153598 -0.012392525
4.2972 0.000152462 -0.012346619
4.3131 0.000151339 -0.012301053
4.329 0.000150228 -0.012255822
4.3449 0.000149129 -0.012210922
4.3608 0.000148042 -0.01216635
4.3768 0.000146967 -0.012122102
4.3927 0.000145904 -0.012078175
4.4086 0.000144852 -0.012034565
4.4245 0.000143811 -0.011991269
4.4404 0.000142782 -0.011948284
4.4563 0.000141764 -0.011905605
4.4723 0.000140756 -0.011863231
4.4882 0.000139759 -0.011821156
4.5041 0.000138773 -0.01177938
4.52 0.000137797 -0.011737897
4.5359 0.000136832 -0.011696706
4.5518 0.000135876 -0.011655803
4.5677 0.000134931 -0.011615185
4.5837 0.000133995 -0.011574849
4.5996 0.000133069 -0.011534792
4.6155 0.000132153 -0.011495012
4.6314 0.000131246 -0.011455505
4.6473 0.000130348 -0.011416269
4.6632 0.00012946 -0.0113773
4.6792 0.00012858 -0.011338597
```

```
4.6951 0.00012771 -0.011300156
4.711 0.000126848 -0.011261975
4.7269 0.000125995 -0.011224051
4.7428 0.000125151 -0.011186382
4.7587 0.000124315 -0.011148965
4.7746 0.000123487 -0.011111797
4.7906 0.000122668 -0.011074876
4.8065 0.000121857 -0.0110382
4.8224 0.000121054 -0.011001766
4.8383 0.000120258 -0.010965571
4.8542 0.000119471 -0.010929614
4.8701 0.000118691 -0.010893892
4.8861 0.000117919 -0.010858403
4.902 0.000117154 -0.010823144
4.9179 0.000116397 -0.010788114
4.9338 0.000115647 -0.01075331
4.9497 0.000114904 -0.010718729
4.9656 0.000114169 -0.01068437
4.9815 0.00011344 -0.010650231
4.9975 0.000112719 -0.010616309
5.0134 0.000112004 -0.010582603
5.0293 0.000111296 -0.01054911
5.0452 0.000110595 -0.010515828
5.0611 0.0001099 -0.010482756
5.077 0.000109212 -0.010449891
5.093 0.00010853 -0.010417232
5.1089 0.000107855 -0.010384776
5.1248 0.000107186 -0.010352521
5.1407 0.000106523 -0.010320467
5.1566 0.000105867 -0.01028861
5.1725 0.000105216 -0.01025695
5.1885 0.000104571 -0.010225483
5.2044 0.000103933 -0.010194209
5.2203 0.0001033 -0.010163126
5.2362 0.000102673 -0.010132232
5.2521 0.000102051 -0.010101525
5.268 0.000101435 -0.010071004
5.2839 0.000100825 -0.010040667
5.2999 0.00010022 -0.010010511
5.3158 9.96E-05 -0.009980537
5.3317 9.90E-05 -0.009950741
5.3476 9.84E-05 -0.009921123
5.3635 9.79E-05 -0.009891681
5.3794 9.73E-05 -0.009862412
5.3954 9.67E-05 -0.009833317
5.4113 9.61E-05 -0.009804393
5.4272 9.56E-05 -0.009775638
5.4431 9.50E-05 -0.009747052
5.459 9.45E-05 -0.009718632
5.4749 9.39E-05 -0.009690377
5.4908 9.34E-05 -0.009662287
5.5068 9.28E-05 -0.009634358
5.5227 9.23E-05 -0.009606591
5.5386 9.18E-05 -0.009578983
5.5545 9.12E-05 -0.009551534
5.5704 9.07E-05 -0.009524241
5.5863 9.02E-05 -0.009497104
5.6023 8.97E-05 -0.009470121
5.6182 8.92E-05 -0.009443292
5.6341 8.87E-05 -0.009416613
5.65 8.82E-05 -0.009390085
5.6659 8.77E-05 -0.009363706
5.6818 8.72E-05 -0.009337475
```

```
5.6977 8.67E-05 -0.00931139
5.7137 8.62E-05 -0.009285451
5.7296 8.57E-05 -0.009259656
5.7455 8.53E-05 -0.009234004
5.7614 8.48E-05 -0.009208493
5.7773 8.43E-05 -0.009183124
5.7932 8.39E-05 -0.009157893
5.8092 8.34E-05 -0.009132801
5.8251 8.30E-05 -0.009107846
5.841 8.25E-05 -0.009083027
5.8569 8.21E-05 -0.009058343
5.8728 8.16E-05 -0.009033792
5.8887 8.12E-05 -0.009009375
5.9046 8.07E-05 -0.008985089
5.9206 8.03E-05 -0.008960933
5.9365 7.99E-05 -0.008936907
5.9524 7.94E-05 -0.00891301
5.9683 7.90E-05 -0.00888924
5.9842 7.86E-05 -0.008865597
6.0001 7.82E-05 -0.008842079
6.0161 7.78E-05 -0.008818685
6.032 7.74E-05 -0.008795415
6.0479 7.70E-05 -0.008772267
6.0638 7.66E-05 -0.008749241
6.0797 7.62E-05 -0.008726336
6.0956 7.58E-05 -0.00870355
6.1115 7.54E-05 -0.008680883
6.1275 7.50E-05 -0.008658333
6.1434 7.46E-05 -0.008635901
6.1593 7.42E-05 -0.008613584
6.1752 7.38E-05 -0.008591382
6.1911 7.34E-05 -0.008569295
6.207 7.31E-05 -0.008547321
6.223 7.27E-05 -0.008525459
6.2389 7.23E-05 -0.008503709
6.2548 7.20E-05 -0.008482069
6.2707 7.16E-05 -0.00846054
6.2866 7.12E-05 -0.008439119
6.3025 7.09E-05 -0.008417807
6.3185 7.05E-05 -0.008396602
6.3344 7.02E-05 -0.008375503
6.3503 6.98E-05 -0.00835451
6.3662 6.95E-05 -0.008333623
6.3821 6.91E-05 -0.008312839
6.398 6.88E-05 -0.008292159
6.4139 6.84E-05 -0.008271581
6.4299 6.81E-05 -0.008251106
6.4458 6.77E-05 -0.008230731
6.4617 6.74E-05 -0.008210457
6.4776 6.71E-05 -0.008190283
6.4935 6.68E-05 -0.008170207
6.5094 6.64E-05 -0.00815023
6.5254 6.61E-05 -0.00813035
6.5413 6.58E-05 -0.008110567
6.5572 6.55E-05 -0.00809088
6.5731 6.51E-05 -0.008071288
6.589 6.48E-05 -0.008051791
6.6049 6.45E-05 -0.008032388
6.6208 6.42E-05 -0.008013078
6.6368 6.39E-05 -0.00799386
6.6527 6.36E-05 -0.007974735
6.6686 6.33E-05 -0.007955701
6.6845 6.30E-05 -0.007936758
```

```
6.7004 6.27E-05 -0.007917905
6.7163 6.24E-05 -0.007899141
6.7323 6.21E-05 -0.007880465
6.7482 6.18E-05 -0.007861878
6.7641 6.15E-05 -0.007843378
6.78 6.12E-05 -0.007824966
6.7959 6.09E-05 -0.007806639
6.8118 6.07E-05 -0.007788398
6.8277 6.04E-05 -0.007770242
6.8437 6.01E-05 -0.007752171
6.8596 5.98E-05 -0.007734183
6.8755 5.95E-05 -0.007716279
6.8914 5.93E-05 -0.007698457
6.9073 5.90E-05 -0.007680718
6.9232 5.87E-05 -0.00766306
6.9392 5.85E-05 -0.007645483
6.9551 5.82E-05 -0.007627987
6.971 5.79E-05 -0.00761057
6.9869 5.77E-05 -0.007593233
7.0028 5.74E-05 -0.007575975
7.0187 5.71E-05 -0.007558795
7.0346 5.69E-05 -0.007541693
7.0506 5.66E-05 -0.007524667
7.0665 5.64E-05 -0.007507719
7.0824 5.61E-05 -0.007490847
7.0983 5.59E-05 -0.00747405
7.1142 5.56E-05 -0.007457329
7.1301 5.54E-05 -0.007440682
7.1461 5.51E-05 -0.00742411
7.162 5.49E-05 -0.007407611
7.1779 5.46E-05 -0.007391185
7.1938 5.44E-05 -0.007374832
7.2097 5.42E-05 -0.007358551
7.2256 5.39E-05 -0.007342342
7.2415 5.37E-05 -0.007326204
7.2575 5.34E-05 -0.007310137
7.2734 5.32E-05 -0.00729414
7.2893 5.30E-05 -0.007278213
7.3052 5.27E-05 -0.007262356
7.3211 5.25E-05 -0.007246567
7.337 5.23E-05 -0.007230847
7.353 5.21E-05 -0.007215195
7.3689 5.18E-05 -0.007199611
7.3848 5.16E-05 -0.007184093
7.4007 5.14E-05 -0.007168643
7.4166 5.12E-05 -0.007153259
7.4325 5.10E-05 -0.007137941
7.4485 5.07E-05 -0.007122688
7.4644 5.05E-05 -0.0071075
7.4803 5.03E-05 -0.007092377
7.4962 5.01E-05 -0.007077318
7.5121 4.99E-05 -0.007062323
7.528 4.97E-05 -0.007047391
7.5439 4.95E-05 -0.007032523
7.5599 4.93E-05 -0.007017717
7.5758 4.90E-05 -0.007002973
7.5917 4.88E-05 -0.006988291
7.6076 4.86E-05 -0.00697367
7.6235 4.84E-05 -0.006959111
7.6394 4.82E-05 -0.006944612
7.6554 4.80E-05 -0.006930173
7.6713 4.78E-05 -0.006915795
7.6872 4.76E-05 -0.006901476
```

```
486  7.7031 4.74E-05 -0.006887216
487  7.719 4.72E-05 -0.006873015
488  7.7349 4.70E-05 -0.006858872
489  7.7508 4.69E-05 -0.006844787
490  7.7668 4.67E-05 -0.00683076
491  7.7827 4.65E-05 -0.006816791
492  7.7986 4.63E-05 -0.006802878
493  7.8145 4.61E-05 -0.006789023
494  7.8304 4.59E-05 -0.006775223
495  7.8463 4.57E-05 -0.00676148
496  7.8623 4.55E-05 -0.006747792
497  7.8782 4.54E-05 -0.006734159
498  7.8941 4.52E-05 -0.006720582
499  7.91 4.50E-05 -0.006707059
500  7.9259 4.48E-05 -0.00669359
501  7.9418 4.46E-05 -0.006680176
502  7.9577 4.44E-05 -0.006666815
```

Appendix A8

tl_s11.txt Simulation Data

This appendix gives the frequency domain data vector, $\bar{S}_{11}(j\omega)$, for the lossless TL circuit of Fig. 5.3. The data was collected using an ADS simulation.

```
! Series of lossless TLs S11(jw) (frequencies in GHz)
0.0000000000000000e+00 -2.4999526569078478e-01 0.0000000000000000e+00
5.0000000000000000e-01 -2.4316607111139632e-01 7.6265797810698324e-02
1.0000000000000000e+00 -2.2040264831234435e-01 1.5432503931685707e-01
1.5000000000000000e+00 -1.7616641550716816e-01 2.3251103635564246e-01
2.0000000000000000e+00 -1.0488973501332344e-01 3.0374154086912170e-01
2.5000000000000000e+00 -4.9576163772463433e-03 3.5606337816704653e-01
3.0000000000000000e+00 1.1832401656618563e-01 3.7571469612940450e-01
3.5000000000000000e+00 2.5219848702755465e-01 3.5157624467796655e-01
4.0000000000000000e+00 3.7873474297564580e-01 2.7917264870648850e-01
4.5000000000000000e+00 4.7898034784218946e-01 1.6261119686261366e-01
5.0000000000000000e+00 5.3731729762092217e-01 1.3893947801994386e-02
5.5000000000000000e+00 5.4454201828808135e-01 -1.4979312661636271e-01
6.0000000000000000e+00 4.9902036041575215e-01 -3.0960119491009946e-01
6.5000000000000000e+00 4.0610839195017201e-01 -4.4823543434614166e-01
7.0000000000000000e+00 2.7647192047984315e-01 -5.5224311554024463e-01
7.5000000000000000e+00 1.2396168907995819e-01 -6.1323849860516844e-01
8.0000000000000000e+00 -3.6496759955048752e-02 -6.2821765599423640e-01
8.5000000000000000e+00 -1.9076315026721691e-01 -5.9919491905164779e-01
9.0000000000000000e+00 -3.2700142661088427e-01 -5.3234697101747708e-01
9.5000000000000000e+00 -4.3680767343142468e-01 -4.3677278528509406e-01
1.0000000000000000e+01 -5.1585712007907036e-01 -3.2293937251564836e-01
1.0500000000000000e+01 -5.6391323346227751e-01 -2.0093428747899603e-01
1.1000000000000000e+01 -5.8403599042619847e-01 -7.8798736523452137e-02
1.1500000000000000e+01 -5.8098561951433902e-01 3.8597457774066479e-02
1.2000000000000000e+01 -5.5917510267718207e-01 1.4963159618436003e-01
1.2500000000000000e+01 -5.2091643573054114e-01 2.5473773668987004e-01
1.3000000000000000e+01 -4.6578571931806978e-01 3.5434191867607018e-01
1.3500000000000000e+01 -3.9147056371564048e-01 4.4686232786084434e-01
1.4000000000000000e+01 -2.9569007512388334e-01 5.2772920993732897e-01
1.4500000000000000e+01 -1.7826103636817925e-01 5.8980627203376734e-01
1.5000000000000000e+01 -4.2466689855886974e-02 6.2488871005997171e-01
1.5500000000000000e+01 1.0462227140226199e-01 6.2559847359185272e-01
1.6000000000000000e+01 2.5275089052369548e-01 5.8708271434164827e-01
1.6500000000000000e+01 3.8967532975461827e-01 5.0820961193573411e-01
1.7000000000000000e+01 5.0264565780180481e-01 3.9219882610474255e-01
1.7500000000000000e+01 5.7998794667301223e-01 2.4672645772695900e-01
1.8000000000000000e+01 6.1268743384532276e-01 8.3533124719550933e-02
1.8500000000000000e+01 5.9588704883863097e-01 -8.2494231575854685e-02
1.9000000000000000e+01 5.3014612658599014e-01 -2.3483469352884973e-01
1.9500000000000000e+01 4.2217867056511849e-01 -3.5746640430628102e-01
2.0000000000000000e+01 2.8467443195000564e-01 -4.3747529908393062e-01
2.0500000000000000e+01 1.3481669266862939e-01 -4.6768670217943542e-01
2.1000000000000000e+01 -8.6129146448813421e-03 -4.4852627834498660e-01
```

```
2.1500000000000000e+01 -1.2906629415273951e-01 -3.8819257636010440e-01
2.2000000000000000e+01 -2.1583861303003871e-01 -3.0060263233500067e-01
2.2500000000000000e+01 -2.6599711081593425e-01 -2.0149203961250284e-01
2.3000000000000000e+01 -2.8367237192388539e-01 -1.0406814467314639e-01
2.3500000000000000e+01 -2.7694486744070668e-01 -1.6014486191138246e-02
2.4000000000000000e+01 -2.5372827866005598e-01 6.0942952192958588e-02
2.4500000000000000e+01 -2.1846742910223083e-01 1.2891645584411945e-01
2.5000000000000000e+01 -1.7099090755086543e-01 1.9029078065037835e-01
2.5500000000000000e+01 -1.0780603713771908e-01 2.4436486739623972e-01
2.6000000000000000e+01 -2.5409331627889165e-02 2.8549163164610392e-01
2.6500000000000000e+01 7.6028960139252533e-02 3.0380502643072171e-01
2.7000000000000000e+01 1.8988745172414556e-01 2.8821154612255284e-01
2.7500000000000000e+01 3.0274910910894937e-01 2.3074460207439984e-01
2.8000000000000000e+01 3.9685057709801574e-01 1.3047849772522027e-01
2.8500000000000000e+01 4.5451340205339807e-01 -4.6800394632342084e-03
2.9000000000000000e+01 4.6282068176242319e-01 -1.5907551059755845e-01
2.9500000000000000e+01 4.1679727035404368e-01 -3.1271131914532080e-01
3.0000000000000000e+01 3.2020746180482540e-01 -4.4555188797004930e-01
3.0500000000000000e+01 1.8417208913255090e-01 -5.4125819779980300e-01
3.1000000000000000e+01 2.4492022653344048e-02 -5.8949934478349775e-01
3.1500000000000000e+01 -1.4134477077982233e-01 -5.8670324570531263e-01
3.2000000000000000e+01 -2.9679050871363954e-01 -5.3557330282728433e-01
3.2500000000000000e+01 -4.2819040616592441e-01 -4.4383259005027303e-01
3.3000000000000000e+01 -5.2601390939732395e-01 -3.2258034117742213e-01
3.3500000000000000e+01 -5.8542366323161477e-01 -1.8450612924170512e-01
3.4000000000000000e+01 -6.0623390134650612e-01 -4.2107405594650087e-02
3.4500000000000000e+01 -5.9226014923056591e-01 9.3956412062016020e-02
3.5000000000000000e+01 -5.5005157449465480e-01 2.1616485128789598e-01
3.5500000000000000e+01 -4.8710304597680554e-01 3.2077723615465004e-01
3.6000000000000000e+01 -4.0989163475742685e-01 4.0748694200565522e-01
3.6500000000000000e+01 -3.2236443250702995e-01 4.7799545976566377e-01
3.7000000000000000e+01 -2.2556129973747385e-01 5.3390561299393879e-01
3.7500000000000000e+01 -1.1866125867856736e-01 5.7481509185785362e-01
3.8000000000000000e+01 -1.0180060841246119e-03 5.9753005485335242e-01
3.8500000000000000e+01 1.2579413136999662e-01 5.9676169656758427e-01
3.9000000000000000e+01 2.5683847263612725e-01 5.6689811673904966e-01
3.9500000000000000e+01 3.8392299731916912e-01 5.0402094593378555e-01
4.0000000000000000e+01 4.9670780843937812e-01 4.0745144922620852e-01
4.0500000000000000e+01 5.8430361622656690e-01 2.8051097190056723e-01
4.1000000000000000e+01 6.3691760677291165e-01 1.3052738214613113e-01
4.1500000000000000e+01 6.4729758664401804e-01 -3.1733534633557665e-02
4.2000000000000000e+01 6.1188231903584578e-01 -1.9304582710544160e-01
4.2500000000000000e+01 5.3162551992935003e-01 -3.3901888236459360e-01
4.3000000000000000e+01 4.1243474145040104e-01 -4.5567640026195810e-01
4.3500000000000000e+01 2.6509336896598068e-01 -5.3127778573821305e-01
4.4000000000000000e+01 1.0446670342023268e-01 -5.5824572647088766e-01
4.4500000000000000e+01 -5.2196539223364868e-02 -5.3488174934344213e-01
4.5000000000000000e+01 -1.8788270971427901e-01 -4.6634053993406177e-01
4.5500000000000000e+01 -2.8898507794041195e-01 -3.6424045579360359e-01
4.6000000000000000e+01 -3.4816927230342709e-01 -2.4451745985771509e-01
4.6500000000000000e+01 -3.6577753240918531e-01 -1.2378652321709296e-01
4.7000000000000000e+01 -3.4890655326219133e-01 -1.5333616325461747e-02
4.7500000000000000e+01 -3.0825670830575702e-01 7.3648573213207558e-02
4.8000000000000000e+01 -2.5400262427751596e-01 1.4235250079044606e-01
4.8500000000000000e+01 -1.9249549709188563e-01 1.9410360649723601e-01
4.9000000000000000e+01 -1.2518405975494640e-01 2.3270360924016315e-01
4.9500000000000000e+01 -5.0062053036385112e-02 2.5906391138164692e-01
5.0000000000000000e+01 3.5109390248993178e-02 2.6956967498135526e-01
5.0500000000000000e+01 1.2927908812831101e-01 2.5683553120286873e-01
5.1000000000000000e+01 2.2554252272145869e-01 2.1269439160742706e-01
5.1500000000000000e+01 3.1110330612046933e-01 1.3227606468189088e-01
5.2000000000000000e+01 3.6971554421947883e-01 1.7668520765609940e-02
5.2500000000000000e+01 3.8613843236631729e-01 -1.2050565871565623e-01
```

```
5.3000000000000000e+01 3.5087200485487169e-01 -2.6443329443696939e-01
5.3500000000000000e+01 2.6332775460505453e-01 -3.9296677896267695e-01
5.4000000000000000e+01 1.3230695653353552e-01 -4.8649809241621939e-01
5.4500000000000000e+01 -2.6109791757519263e-02 -5.3109619104820138e-01
5.5000000000000000e+01 -1.9218890605727357e-01 -5.2075855831503404e-01
5.5500000000000000e+01 -3.4636049949514103e-01 -4.5757530723897016e-01
5.6000000000000000e+01 -4.7225911119861819e-01 -3.5031552728775522e-01
5.6500000000000000e+01 -5.5858736136631881e-01 -2.1218822572964788e-01
5.7000000000000000e+01 -5.9983828786701709e-01 -5.8414542247863466e-02
5.7500000000000000e+01 -5.9610949586173145e-01 9.5997824704268167e-02
5.8000000000000000e+01 -5.5225288827148622e-01 2.3809174100104896e-01
5.8500000000000000e+01 -4.7653532170487212e-01 3.5831747287607907e-01
5.9000000000000000e+01 -3.7893689776052708e-01 4.5130928088896349e-01
5.9500000000000000e+01 -2.6925353784921557e-01 5.1593051039927762e-01
6.0000000000000000e+01 -1.5531707138790241e-01 5.5444229925171296e-01
6.0500000000000000e+01 -4.1824362313233276e-02 5.7082724392254058e-01
6.1000000000000000e+01 6.9710820333587220e-02 5.6867213315096432e-01
6.1500000000000000e+01 1.7969770759641701e-01 5.4940028004097852e-01
6.2000000000000000e+01 2.8834842164070240e-01 5.1168637973664965e-01
6.2500000000000000e+01 3.9367756270554044e-01 4.5237831445214693e-01
6.3000000000000000e+01 4.9056811433779757e-01 3.6846378316475425e-01
6.3500000000000000e+01 5.7124158251655488e-01 2.5913463481934002e-01
6.4000000000000000e+01 6.2676972786688623e-01 1.2712644779087207e-01
6.4500000000000000e+01 6.4894195806116195e-01 -2.0979753517041042e-02
6.5000000000000000e+01 6.3191276316093981e-01 -1.7534337440828321e-01
6.5500000000000000e+01 5.7334928161528187e-01 -3.2399662447904559e-01
6.6000000000000000e+01 4.7504030350836457e-01 -4.5422100862449488e-01
6.6500000000000000e+01 3.4302950852965730e-01 -5.5398886984923668e-01
6.7000000000000000e+01 1.8732618824205938e-01 -6.1338973929727447e-01
6.7500000000000000e+01 2.1187859762979100e-02 -6.2599070546622149e-01
6.8000000000000000e+01 -1.4008304813652350e-01 -5.9003497716714526e-01
6.8500000000000000e+01 -2.8088268033234898e-01 -5.0927884338921015e-01
6.9000000000000000e+01 -3.8753681161956199e-01 -3.9314593567221634e-01
6.9500000000000000e+01 -4.5072908109921783e-01 -2.5582567795802164e-01
7.0000000000000000e+01 -4.6751371799391761e-01 -1.1408900623342517e-01
7.0500000000000000e+01 -4.4207449819092737e-01 1.5964552703077003e-02
7.1000000000000000e+01 -3.8454754501216770e-01 1.2236879317415031e-01
7.1500000000000000e+01 -3.0793110547172264e-01 1.9963185857609686e-01
7.2000000000000000e+01 -2.2411239432098851e-01 2.4881829787435208e-01
7.2500000000000000e+01 -1.4068790187012747e-01 2.7520758290758529e-01
7.3000000000000000e+01 -5.9982162982666565e-02 2.8458774733801817e-01
7.3500000000000000e+01 1.9404119175411294e-02 2.7990918467113501e-01
7.4000000000000000e+01 9.9361824980132019e-02 2.5976455338389492e-01
7.4500000000000000e+01 1.7884505878268508e-01 2.1927506376209022e-01
7.5000000000000000e+01 2.5170084922906244e-01 1.5300641923172328e-01
7.5500000000000000e+01 3.0675689516938398e-01 5.8729926179353700e-02
7.6000000000000000e+01 3.3026974199868731e-01 -5.9206147437088123e-02
7.6500000000000000e+01 3.1022879045279006e-01 -1.8870983968575641e-01
7.7000000000000000e+01 2.4085450229971128e-01 -3.1139942743952426e-01
7.7500000000000000e+01 1.2560714276503049e-01 -4.0664120893891992e-01
7.8000000000000000e+01 -2.2598803849426252e-02 -4.5669274581667318e-01
7.8500000000000000e+01 -1.8404625310746392e-01 -4.5106843529481361e-01
7.9000000000000000e+01 -3.3661908445704469e-01 -3.8868945614423334e-01
7.9500000000000000e+01 -4.6033486033525062e-01 -2.7745459498115999e-01
8.0000000000000000e+01 -5.4070335765215449e-01 -1.3186444120712978e-01
8.0500000000000000e+01 -5.7035927510120144e-01 3.0230010231923503e-02
8.1000000000000000e+01 -5.4909345421107236e-01 1.9080363847933085e-01
8.1500000000000000e+01 -4.8274728468921091e-01 3.3419354982576494e-01
8.2000000000000000e+01 -3.8146504644974810e-01 4.4879467508144433e-01
8.2500000000000000e+01 -2.5767554415424743e-01 5.2792118364264629e-01
8.3000000000000000e+01 -1.2404618030411141e-01 5.6989428079852533e-01
8.3500000000000000e+01 8.3944111966050006e-03 5.7739250355497052e-01
8.4000000000000000e+01 1.3171830060170708e-01 5.5609778782792874e-01
```

```
8.4500000000000000e+01 2.4180864838924299e-01 5.1278213358557190e-01
8.5000000000000000e+01 3.3799288171594366e-01 4.5323376092144263e-01
8.5500000000000000e+01 4.2153778018586596e-01 3.8069802806448255e-01
8.6000000000000000e+01 4.9346528636822495e-01 2.9553314194066083e-01
8.6500000000000000e+01 5.5262366419664399e-01 1.9633752291090992e-01
8.7000000000000000e+01 5.9494283399412184e-01 8.2055499609274940e-02
8.7500000000000000e+01 6.1419892217483385e-01 -4.5955897882827994e-02
8.8000000000000000e+01 6.0383390420486660e-01 -1.8280632675991501e-01
8.8500000000000000e+01 5.5898175877189682e-01 -3.2020676202012105e-01
8.9000000000000000e+01 4.7799933329463951e-01 -4.4758088273807634e-01
8.9500000000000000e+01 3.6321229202613869e-01 -5.5363416253632469e-01
9.0000000000000000e+01 2.2092955455430974e-01 -6.2794323648788286e-01
9.0500000000000000e+01 6.0920029263274333e-02 -6.6232963486777396e-01
9.1000000000000000e+01 -1.0447175963667366e-01 -6.5194313447506813e-01
9.1500000000000000e+01 -2.6147277062225693e-01 -5.9604645285862856e-01
9.2000000000000000e+01 -3.9625936728587918e-01 -4.9847720610781049e-01
9.2500000000000000e+01 -4.9654855775672035e-01 -3.6769875226223236e-01
9.3000000000000000e+01 -5.5332483096347107e-01 -2.1627939623093395e-01
9.3500000000000000e+01 -5.6248240840102293e-01 -5.9614849561926489e-02
9.4000000000000000e+01 -5.2596762292008381e-01 8.6192883985319491e-02
9.4500000000000000e+01 -4.5186282899288843e-01 2.0710075170464790e-01
9.5000000000000000e+01 -3.5294553117453265e-01 2.9389149612978016e-01
9.5500000000000000e+01 -2.4374191301927195e-01 3.4392428376019996e-01
9.6000000000000000e+01 -1.3688378295037484e-01 3.6092015213143014e-01
9.6500000000000000e+01 -4.0208058525110557e-02 3.5257113972133253e-01
9.7000000000000000e+01 4.4065944869647211e-02 3.2687636623316907e-01
9.7500000000000000e+01 1.1769872944583315e-01 2.8885954552049348e-01
9.8000000000000000e+01 1.8317265782465886e-01 2.3915288005021151e-01
9.8500000000000000e+01 2.4036992603181773e-01 1.7499053754726773e-01
9.9000000000000000e+01 2.8464622822536345e-01 9.3084586721219581e-02
9.9500000000000000e+01 3.0711015444673517e-01 -6.8256514773133862e-03
1.0000000000000000e+02 2.9711948662161913e-01 -1.1913593143837091e-01
```

Appendix A9

`butterworth_s11.txt` Simulation Data

This appendix gives the frequency domain data vector, $\bar{S}_{11}(j\omega)$, for the Butterworth filter circuit of Fig. 5.6. The data was collected via using an ADS simulation.

```
! Butterworth LPF S11(jw) (frequencies are in Hz)
0                  0                  0
10000000                   0     9.96560534e-27
20000000                   0     8.07948152e-23
30000000                   0      1.5451875e-20
40000000                   0       6.344619e-19
50000000                   0     1.11712025e-17
60000000    2.22044605e-16     1.14744806e-16
70000000     4.4408921e-16     8.09877579e-16
80000000    3.33066907e-15     4.32889825e-15
90000000    1.73194792e-14     1.86444328e-14
100000000    7.39408534e-14     6.74457595e-14
110000000    2.73114864e-13     2.10832911e-13
120000000    8.98614516e-13     5.80622678e-13
130000000    2.67208478e-12     1.42596462e-12
140000000    7.28950234e-12     3.13996894e-12
150000000    1.84545712e-11     6.18046743e-12
160000000    4.37503367e-11      1.0683818e-11
170000000    9.78466197e-11     1.53658663e-11
180000000     2.0769142e-10     1.49640247e-11
190000000    4.20499413e-10    -5.05625089e-12
200000000    8.15421286e-10     -7.8591703e-11
210000000    1.51969348e-09    -2.76953558e-10
220000000    2.72969691e-09    -7.39531672e-10
230000000    4.73649386e-09    -1.72362723e-09
240000000    7.95383603e-09    -3.68234631e-09
250000000    1.29440463e-08    -7.38215908e-09
260000000     2.0433192e-08    -1.40746806e-08
270000000    3.13011879e-08    -2.57401631e-08
280000000     4.6524447e-08     -4.5422695e-08
290000000    6.70379694e-08    -7.76785751e-08
300000000    9.34699329e-08    -1.29158935e-07
310000000    1.25684591e-07    -2.09344313e-07
320000000    1.62048477e-07    -3.31441157e-07
330000000    1.98310678e-07    -5.13436401e-07
340000000    2.25960841e-07    -7.79284375e-07
350000000     2.2989961e-07    -1.16016799e-06
360000000    1.85227186e-07    -1.69573098e-06
370000000    5.29291346e-08    -2.43511711e-06
380000000   -2.25781851e-07    -3.43757368e-06
390000000   -7.36719159e-07    -4.77227693e-06
400000000   -1.60181393e-06    -6.51691563e-06
```

```
410000000   -2.99025681e-06    -8.75442408e-06
420000000   -5.13183593e-06     -1.1567088e-05
430000000   -8.33250733e-06    -1.50270585e-05
440000000   -1.29920684e-05    -1.91821042e-05
450000000   -1.96235691e-05    -2.40352196e-05
460000000   -2.88737767e-05    -2.95164963e-05
470000000   -4.15435936e-05    -3.54454745e-05
480000000   -5.86067978e-05    -4.14820409e-05
490000000   -8.12248248e-05    -4.70638571e-05
500000000   -0.000110754524    -5.13283232e-05
510000000   -0.000148744897    -5.30172476e-05
520000000   -0.000196917779    -5.03627508e-05
530000000   -0.000257126219    -4.09535461e-05
540000000   -0.000331283071    -2.15816645e-05
550000000   -0.000421250924     1.19289875e-05
560000000   -0.000528683196     6.49090674e-05
570000000   -0.000654804928     0.000144039733
580000000   -0.000800120739     0.000257549615
590000000    -0.00096403668     0.000415400726
600000000    -0.00114438245     0.000629446752
610000000    -0.00133682095      0.00091354199
620000000    -0.00153413357       0.0012835731
630000000    -0.00172537251        0.001757379
640000000    -0.00189487555      0.00235451664
650000000    -0.00202114563      0.00309582221
660000000     -0.0020756062      0.00400270877
670000000    -0.00202125568      0.00509613285
680000000    -0.00181126008      0.00639515428
690000000    -0.00138754269       0.0079150068
700000000   -0.000679454967      0.00966459179
710000000    0.000397356928       0.0116433058
720000000     0.00194173819       0.0138371149
730000000      0.0040672683       0.0162137999
740000000     0.00690171291       0.0187173114
750000000      0.0105853759       0.0212612099
760000000      0.0152679733       0.0237212101
770000000      0.0211035718       0.0259269243
780000000      0.0282430478       0.0276530016
790000000      0.0368234317       0.0286100109
800000000      0.0469534125       0.0284356256
810000000       0.058694206       0.0266869575
820000000      0.0720349559       0.0228352926
830000000       0.086861897       0.0162650213
840000000       0.102920726      0.00627927159
850000000       0.119772136     -0.00788434476
860000000       0.136741451      -0.0270233963
870000000       0.152865013      -0.0519201216
880000000       0.166838753      -0.0832643102
890000000       0.176978527       -0.121543158
900000000       0.181207432       -0.166893622
910000000       0.177091916       -0.218918252
920000000       0.161954299        -0.27647629
930000000       0.133090594       -0.337479353
940000000       0.088113139       -0.398744092
950000000      0.0254107504         -0.4559765
960000000     -0.0553282291       -0.503970405
970000000      -0.152643995       -0.537077472
980000000      -0.263043835         -0.5499351
990000000       -0.38110371       -0.538332599
1e+09      -0.500000009       -0.499999996
1.01e+09      -0.612391974       -0.435080322
1.02e+09      -0.711435539       -0.346130761
1.03e+09      -0.791656698       -0.237659942
```

```
1.04e+09       -0.849478026       -0.115363573
1.05e+09       -0.883332944       0.0147092803
1.06e+09        -0.89343837        0.146859347
1.07e+09       -0.881369261        0.276203276
1.08e+09       -0.849579695         0.39889224
1.09e+09       -0.800972575        0.512134067
1.1e+09      -0.738567564        0.614090912
1.11e+09       -0.665275911        0.703717547
1.12e+09       -0.583767616        0.780586544
1.13e+09       -0.496407833        0.844727221
1.14e+09       -0.405239786        0.896490418
1.15e+09       -0.311995809        0.936441718
1.16e+09       -0.218123311        0.965280836
1.17e+09       -0.124817019        0.983782809
1.18e+09      -0.0330522728        0.992756334
1.19e+09        0.056383463        0.993015016
1.2e+09       0.142862211        0.985358077
1.21e+09        0.225890656        0.970557831
1.22e+09        0.305088824        0.949351916
1.23e+09        0.380171975        0.922438823
1.24e+09        0.450935297        0.890475687
1.25e+09        0.517241024        0.854077582
1.26e+09        0.579007566        0.813817843
1.27e+09        0.636200346        0.770229041
1.28e+09        0.688824044        0.723804371
1.29e+09        0.736916014        0.674999304
1.3e+09       0.780540682        0.624233381
1.31e+09        0.819784766         0.57189207
1.32e+09        0.854753177        0.518328648
1.33e+09        0.885565509         0.46386607
1.34e+09        0.912353017        0.408798793
1.35e+09        0.935256026        0.353394554
1.36e+09        0.954421687        0.297896088
1.37e+09        0.970002062        0.242522771
1.38e+09        0.982152471        0.187472198
1.39e+09        0.991030073         0.13292169
1.4e+09       0.996792654       0.0790297196
1.41e+09        0.999597598       0.0259372708
1.42e+09         0.99960101      -0.0262308756
1.43e+09        0.996956975      -0.0773649286
1.44e+09        0.991816944       -0.127368941
1.45e+09        0.984329219        -0.17615973
1.46e+09        0.974638534       -0.223665864
1.47e+09        0.962885717       -0.269826713
1.48e+09        0.949207422        -0.31459155
1.49e+09        0.933735921       -0.357918726
1.5e+09       0.916598955       -0.39977488
1.51e+09        0.897919628       -0.440134218
1.52e+09        0.877816341       -0.478977831
1.53e+09        0.856402764       -0.516293065
1.54e+09        0.833787838       -0.552072933
1.55e+09        0.810075797        -0.58631557
1.56e+09        0.785366221       -0.619023728
1.57e+09        0.759754102       -0.650204306
1.58e+09        0.733329924       -0.679867918
1.59e+09        0.706179758       -0.708028492
1.6e+09       0.678385372      -0.734702903
1.61e+09        0.650024341       -0.759910629
1.62e+09        0.621170166       -0.783673437
1.63e+09        0.591892403         -0.8060151
1.64e+09        0.562256793       -0.826961126
1.65e+09        0.532325388        -0.84653852
1.66e+09        0.502156689       -0.864775558
```

```
1.67e+09        0.471805777        -0.881701587
1.68e+09        0.441324446        -0.897346838
1.69e+09        0.410761335        -0.911742254
1.7e+09        0.380162061        -0.924919342
1.71e+09        0.349569343        -0.936910027
1.72e+09        0.319023133        -0.947746532
1.73e+09        0.288560736        -0.957461255
1.74e+09        0.258216932        -0.966086673
1.75e+09        0.228024094        -0.973655243
1.76e+09        0.198012299        -0.980199324
1.77e+09        0.168209444        -0.985751097
1.78e+09        0.138641346        -0.990342501
1.79e+09        0.109331853        -0.994005171
1.8e+09       0.0803029361        -0.996770389
1.81e+09       0.0515747933        -0.998669035
1.82e+09       0.0231659361        -0.999731547
1.83e+09    -0.00490671857        -0.999987887
1.84e+09     -0.0326277638        -0.999467508
1.85e+09     -0.0599832204        -0.998199329
1.86e+09     -0.0869604589        -0.996211715
1.87e+09      -0.113548124        -0.993532454
1.88e+09      -0.139736062        -0.990188749
1.89e+09      -0.165515254        -0.986207197
1.9e+09      -0.190877748        -0.981613788
1.91e+09      -0.215816599        -0.976433892
1.92e+09      -0.240325806         -0.97069226
1.93e+09      -0.264400257        -0.964413017
1.94e+09      -0.288035675        -0.957619662
1.95e+09      -0.311228565        -0.950335073
1.96e+09      -0.333976168        -0.942581505
1.97e+09      -0.356276409        -0.934380596
1.98e+09      -0.378127859         -0.92575337
1.99e+09      -0.399529688        -0.916720247
2e+09      -0.420481626        -0.907301046
```

Appendix A10

test.ckt Gnucap Batch Script

This is the Gnucap batch script which was used to easily test the RC model plugin. It should be run from the directory which contains the `s_param_data.txt` data vector file, using `gnucap -b test.ckt`.

```
# SIMPLE TEST FOR RC_MODEL; RUN FROM COMMAND LINE USING 'gnucap -b test.ckt'
.load rc.so
.gen freq=2
.gen amplitude=1
Vs 1 0 generator(1)
Rs 1 2 50
p1 2 0 50 rclog=1 vflog=1 numi=450 nump=35
.options dtmin 0.1e-3
.options dtratio 1
.print tr v(p1) i(p1) dt(p1)
.tr 0 5 0.1e-3 trace all > sim_out.txt
.end
```