
Virtual Analog modelling with Automated Nodal DK-method in Real-Time

Project Report

Joakim Borg, Rasmus Højsted Kürstein, Daniel Strübig

Aalborg University
Electronics and IT



Electronics and IT

Aalborg University

<http://www.aau.dk>

AALBORG UNIVERSITY

STUDENT REPORT

Title:

Virtual Analog modelling with Automated Nodal DK-method in Real-Time

Theme:

Virtual Analog

Project Period:

Spring Semester 2019

Project Group:

Group 1

Participant(s):

Joakim Borg

Rasmus Højsted Kürstein

Daniel Strübig

Supervisor(s):

Stefania Serafin

Copies: 1

Page Numbers: 45

Date of Completion:

May 28, 2019

Abstract:

Nodal DK is a method used for Virtual Analog modelling, to derive nonlinear state-space systems as physical models of electronic circuits.

Compared to other methods, the Nodal DK method is not modular, as the nonlinear equations of the state-space system have to be derived for a specific circuit. However, the method can be implemented with automation of the derivation of these equations.

Other frameworks using this method have been presented, however the majority are offline implementations. In this project an implementation of a framework using the automated NDK-method in real-time is presented. This consists of a parser for automation, a C++ library for real-time applications as well as an offline testing environment.

To exemplify and evaluate our implementation, a handful of circuits varying in complexity have been modelled to demonstrate the effectiveness of this framework.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

0.1	List of Abbreviations and Terms	vii
0.2	List of Symbols and Math Operations	ix
Preface		xi
1	Introduction	1
1.1	Motivation	2
2	Related Works	3
2.1	SPICE	3
2.2	Wave Digital Filters	4
2.3	State-Space Modelling	4
2.4	Nonlinear Solvers	5
3	Fundamentals	7
3.1	Modified Nodal Analysis	7
3.2	Discretization	8
3.3	Generating an NDK-Model from a Circuit	9
3.4	State-space Model	10
3.5	Solving the System	11
3.6	Nonlinear Components	12
3.6.1	The Diode	12
3.6.2	NPN Transistor	15
3.6.3	Triode	18
3.7	Operational Amplifiers	19
3.8	Real-time Improvements for Potentiometers	20
4	Implementation	21
4.1	Parser	21
4.1.1	Syntax	22
4.1.2	Parsing procedure	23
4.2	C++ Framework	23
4.2.1	Class Structure	24
4.2.2	Extendability of Nonlinear Components and Nonlinear Solvers	25
4.2.3	Interface Design	25
4.3	Matlab	25

5	Evaluation	27
5.1	RLC Lowpass Filter	27
5.2	Diode	28
5.3	Common Emitter Amplifier	29
5.4	Common Cathode Amplifier	31
5.5	Operational Amplifier	32
5.6	Wah-Wah Effect Pedal	33
5.7	Stress Testing	34
6	Discussion	37
6.1	Accuracy	37
6.2	Stability	37
6.3	Computational Cost	38
6.4	Usability	38
7	Conclusion	41
	Bibliography	43
A		45
A.1	C++ Class Diagram	45

0.1 List of Abbreviations and Terms

NDK	Nodal Discrete K
KCL	Kirchhoff's Current Law
MNA	Modifier Nodal Analysis
lhs, rhs	Left hand side, right hand side
SPICE	Simulation Program with Integrated Circuit Emphasis
LTspice	SPICE simulation software
JUCE	C++ cross-platform application framework
VST3	Standard for audio plugins
AU	Standard for audio plugins
JSON	Data-interexchange format
Eigen	Template library for linear algebra in C++
REAPER	A Digital Audio Workstation (DAW)
RLC	An electrical circuit consisting of a resistor, inductor and capacitor
netlist	Text representation of a SPICE circuit

0.2 List of Symbols and Math Operations

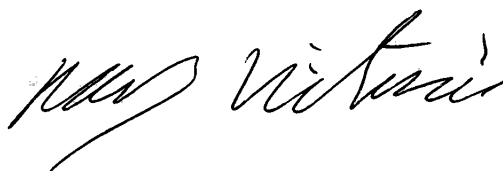
Θ	$\Phi \times \Phi$ conductance matrix (MNA)
ϕ	$\Phi \times 1$ vector of potentials at each node in a circuit
i_s	$n_u \times 1$ vector of source currents
$i_n, i_n[n]$	$n_i \times 1$ vector of nonlinear currents (and its discrete counterpart)
$u, u[n]$	$n_u \times 1$ vector of input voltages
$x, x[n-1]$	$n_x \times 1$ vector of state currents (and its discrete counterpart)
$y[n]$	$n_o \times 1$ discrete vector of output voltages
$v_n, v_n[n]$	$n_n \times 1$ vector of nonlinear voltages (and its discrete counterpart)
M -form,	The set of matrices derived from manual MNA
N -form	The set of matrices derived from algorithmic NDK
N_r, M_r	$n_r \times \Phi$ matrices describing the positions of resistors
N_x, M_x	$n_x \times \Phi$ matrices describing the positions of capacitors and inductors
N_v	$n_v \times 1$ matrix describing the positions of potentiometers
N_u, M_u	$n_u \times \Phi$ matrices describing the positions of source voltages
N_n, M_n	$n_n \times \Phi$ matrices describing the positions of nonlinear components
N_o	$n_o \times \Phi$ matrix describing the positions of output nodes
N_{opaO}	$n_{opa} \times \Phi$ matrix describing the positions of opamps
N_{opaI}	$n_{opa} \times \Phi$ matrix describing the positions of opamps
G_r	$n_r \times n_r$ diagonal matrix containing the conductance of resistors
G_x	$n_x \times n_x$ diagonal matrix containing the discretized conductance of capacitors and inductors
R_v	$n_v \times n_v$ diagonal matrix containing the resistance of potentiometers
Z	$n_x \times n_x$ diagonal matrix containing the sign of the discretization of capacitors (1) and inductors (-1)
Φ	The number of nodes in the circuit
n_r	The number of resistors in the circuit
n_x	The number of capacitors and inductors in the circuit
n_u	The number of source voltages in the circuit
n_n	The number of nonlinear equations derived from the circuit
n_o	The number of outputs in the circuit
n_v	The number of potentiometer resistors (two per potentiometer)
I	The identity matrix (size should be clear from context)
S	$(\Phi + n_u) \times (\Phi + n_u)$ conductance matrix (or $(\Phi + n_u + n_{opa}) \times (\Phi + n_u + n_{opa})$ with opamp extension)
S_M, S_N	The manual MNA and the algorithmic NDK version of S respectively
0	The zero matrix or zero vector (size should be clear from context)
T	The sampling period
f_s	The sampling rate
$\text{diag}(a_1, \dots, a_n)$	A $n \times n$ diagonal matrix with the elements a_1, \dots, a_n along its diagonal
$A, B, C, D, E,$	The final (real-time) state-space matrices
F, G, H, K	
$A_0, B_0, C_0, D_0, E_0,$	The offline computed state-space matrices used to construct $A - K$.
F_0, G_0, H_0, K_0	
$J_f(x)$	The numerical jacobian of the function f with the vector x as input
$(A)_{k,m}$	The entry of matrix A at row k , column m
$(a)_k$	The k :th entry of vector a
$a_{\text{name},k}$	The k :th entry of a vector a_{name}

Preface

Aalborg University, May 28, 2019



Joakim Borg
<jborg18@student.aau.dk>



Rasmus Højsted Kürstein
<rkars15@student.aau.dk>



Daniel Strübig
<dstrub18@student.aau.dk>

Chapter 1

Introduction

In the field of sound synthesis, the idea of physical modelling differs vastly from other methods, as its underlying approach is the approximation of the actual physical behaviour that takes place in a physical system. Oftentimes, this process is represented through a system of differential equations, which, when solved, produces the desired output data. Over the past decades, a multitude of branches within physical modeling has evolved, each with their own level of complexity [20].

Sound synthesis using physical modelling provides significant advantages over e.g. filter-based solutions, such as its flexibility and access to parameters, which allows more complex control of an instrument. Due to the nature of its numerical approximation, physical modeling allows for the creation of instruments that would be logistically expensive or impossible to build otherwise. This flexibility however requires more processing power than sample-based sound synthesis. Depending on the sophistication of the approximation method, the computational cost can vary from model to model.

The branch of virtual analog modelling within the physical modelling domain focuses on the numerical representation and emulation of electronic circuits. The motivation behind virtual analog modeling is among other aspects the accessibility of vintage synthesizers, which may be out of production, particularly expensive or challenging to control [24]. To emulate a circuit in the digital domain, a number of methods can be used to discretize the individual components of a circuit and approximate their influence on the audio signal.

This report describes the development of a framework that provides a set of tools to turn a circuit schematic representation into a audio plug-in. The idea is to take a schematic from a circuit simulation tool (e.g. LTspice), export it in a netlist representation and feed it into an audio development framework. This procedure aims to speed up the development process by creating prototypes quickly and easily. The framework relies heavily on the concept of state-space modeling and the adaptations of these concepts by Martin Holters and Udo Zölzer [10] and as well as David Yeh [22], which will be explained in the following chapters.

1.1 Motivation

Modeling analog audio circuits has been, and still is, an expanding field in modern research. Most notably the works by Yeh, Holters, Zölzer and Mačák have laid the foundation for a systematic derivation of circuit approximations [22][10][13]. Having this collection of research at hand, the motivation for this framework was to create a tool that allows for fast prototyping of circuits, targeted towards plugin developers. Moreover, while real-time implementations of the proposed methods exist [25], our framework provides a multitude of different features for real-time application as well as an offline environment for debugging and allows for fast prototyping. Thus, the motivation of this project was to contribute to the virtual analog domain by combining the recent advances in the field and provide an easy-to-use prototyping framework.

Chapter 2

Related Works

A selection of paradigms for circuit simulation is presented and explained in this chapter, to provide an overview of the state of the art for Virtual Analog modelling. Each category is accompanied by frameworks and solutions that have been developed. Specifically state-space modeling and nodal analysis are emphasized as these are the focal point of this project. Moreover, a brief overview of current challenges for the different methods of Virtual Analog modelling is given with emphasis on the method of iteratively solving nonlinear equations.

2.1 SPICE

To simulate the behaviour of an electronic circuit digitally and possibly use it for Virtual Analog modelling, there is a multitude of tools available. Circuit simulation has been developed for a long time, especially offline circuit simulation, as it has been used by electrical engineerers since the 70s. One such circuit simulator is SPICE [19], which enables the user to place components on a digital drawing board. These components can then be connected and their behaviour simulated. Furthermore, a circuit can be specified as a netlist - This is a text representation of the circuit, which specifies each component, its properties, parameters and values (e.g. capacitance of a capacitor) and most importantly the nodes which they are connected to.

With this list, SPICE sets up (non-)linear equations for a specified circuit and solves them automatically. This produces the time-domain responses of the circuit on a given input signal. To assure convergence and low error rate, SPICE uses high sampling rates in some situations, which lets it achieve precision that exceeds the limits of our audible perception [23]. These high sample rates are one of the reasons why SPICE can only be used for offline simulations. Despite this, it does however provide a great baseline for circuit simulations.

2.2 Wave Digital Filters

One mathematical approach to Virtual Analog modelling is using wave digital filters (WDF). The theory behind WDF has been elaborated by Alfred Fettweis and is based on the concept of computing signals as incoming and outgoing waves between the components of a circuit [6]. Every component acts as a filter, in the sense that they introduce a certain transformation of an incoming wave onto the outgoing wave.

The signals and states stored as wave variables are a conversion of the relationship of the Kirchhoff pair, e.g. voltage and current [15].

On a broader level, WDF components can be described by its ports and terminals for incoming and outgoing waves. Simple components with two terminals, such as resistors, diodes or capacitors, are modelled as one-port elements. The components are then connected through adaptors, the equivalent to nodes in a circuit. With this the circuit can be described as a binary tree and its computation scheme can be derived. One element is chosen as the root - If the circuit has a nonlinear component, this component needs to be the root. The rest of the components are leaves of the binary tree structure. To calculate the waves of the components, the up-going waves are calculated from the leaves to the root. Afterwards, the outgoing wave is calculated at the root, and propagated down to the leaves. This process is then repeated. [15].

A handful of frameworks for Virtual Analog modelling using WDF exist. One of those frameworks is the MATLAB framework by Zölzer et. al, which provides a set of scripts and structs that allows for an implementation of simple circuits [15].

Due to its rigid computational scheme, circuit topologies not allowing a binary-tree structure are not possible to compute. An example of this is a bridge connection. Furthermore, the circuit is thus limited to having only a single one-port nonlinearity. However, different solutions have been suggested for this problem: Schwerdtfeger and Kummert propose the extension to multidimensional Wave Digital Filters. For the approach, slow convergence time becomes issue. A modified approach is however presented as well, where convergence time can be decreased [17].

Werner et. al. present another framework for modelling circuits with multiple nonlinearities through WDF [21]. In the framework, the multiple nonlinearities are combined into a single vector at the root of the binary-tree. The vector system is solved by Modified Nodal Analysis (MNA) and a modified case of the K-Method. However, the derivation of the vector is convoluted and to some degree obscures the modularity of WDF.

2.3 State-Space Modelling

A different, yet equally viable approach to Virtual Analog is state-space modelling, as it has been subject to extensive research over the past decade. State-space models describe a system with multiple states, inputs and/or outputs as a set of differential equations. It was originally developed as a linear model in the form

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \quad (2.1)$$

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \quad (2.2)$$

where \mathbf{x} denotes the state vector, \mathbf{u} the input vector and \mathbf{y} the output vector, with $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ being matrices of corresponding sizes, but was later extended to handle nonlinear, state-

less elements [13]. Among the more prominent approaches is the K-method, originally developed to solve computability problems in nonlinear acoustic systems [1] which was adapted in [22] for the simulation of analog circuits with memoryless nonlinear components by adding a vector \mathbf{i}_n for the nonlinear currents and extending the system to

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{C}\mathbf{i}_n \quad (2.3)$$

$$\mathbf{i}_n = \mathbf{f}(\mathbf{D}\mathbf{x} + \mathbf{E}\mathbf{u} + \mathbf{F}\mathbf{i}_n) \quad (2.4)$$

$$\mathbf{y} = \mathbf{L}\mathbf{x} + \mathbf{M}\mathbf{u} + \mathbf{N}\mathbf{i}_n \quad (2.5)$$

where the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{L}, \mathbf{M}, \mathbf{N}$ are derived by using modified nodal analysis (see section 3.1) on the given circuit [22]. It was also discovered that solving the nonlinear equation with respect to voltage instead of current, so that equation 2.4 is replaced by

$$\mathbf{G}\mathbf{x} + \mathbf{H}\mathbf{u} + \mathbf{K}\mathbf{i}_n(\mathbf{v}_n) = \mathbf{v}_n, \quad (2.6)$$

has a faster rate of convergence [23]. This resulted in the nodal DK-method (NDK-method), which will be described in detail in chapter 3.

The NDK-method has been used extensively for modelling analog circuits as real-time audio effects. Previous research include models of guitar pedals [10], amplifiers [23] and diode clippers [24], among others. Various improvements have been made to extend the method and to handle some of its computationally more demanding aspects. In [13], the model was extended to handle the operational amplifier, both with a nonlinear model as well as a linear, as the nonlinear behaviour of the opamp rarely is used in audio context. Among the performance improvements, [10] modified the model to handle variable resistors (i.e. potentiometers) separately, thus allowing for input of parameters in real-time (see section 3.8 for details). While these modifications have been made to extend the NDK-model itself, most of the research on computational improvements has been oriented towards ways of solving the nonlinear equations. The current status of this research is described in detail in section 2.4.

Several frameworks for handling NDK systems have been developed, both for algorithmic generation of the NDK matrices from SPICE netlists [25], real-time implementations for solving arbitrary NDK systems in C++ [23], as well as more recent offline implementations for MATLAB [14] and Julia [8]. While the above mentioned framework for generation of NDK matrices and the real-time implementation in C++ integrable with each other, none of the others are based on the same iteration of the NDK-method and there is thus no way to integrate all three aspects (netlist derivation, offline debugging and real-time audio) into a single framework, thus making developing real-time audio applications using either framework significantly more cumbersome.

While the computation of multiple nonlinearities in a circuit is possible, circuit nodes that are connected to only nonlinear elements cannot be computed, which is an issue to this date [8].

2.4 Nonlinear Solvers

Most virtual analog implementations of analog circuits revolve around solving a nonlinear relationship between the current and the voltage over a given component. In state-space

modelling, this issue can be identified in equation 2.6. A commonly-used approach for solving this is the Newton-Raphson method [23], [22], [10], [13], where the solution is found iteratively. However, solving this system using Newton-Raphson requires the jacobian of the system to be inverted multiple times at each time-step, resulting in an excessive computational costs.

There are mainly two approaches to reducing this cost in real-time applications: The first approach is to improve the convergence rate of the algorithm, thus reducing the number of iterations required at each time step, using for example damped Newton-Raphson [7] or homotopy [22]. The second approach is to compute the results of all possible combinations of values for the nonlinear voltages offline and store in a lookup table [23], [13]. The latter solution will however result in very large lookup tables, as their sizes grow exponentially with the number of nonlinear functions [8]. While there are various methods for decreasing the size of the lookup table [13], [9], these will not be addressed in this paper.

Chapter 3

Fundamentals

In this chapter, the necessary fundamentals of state-space modeling are explained to get a thorough understanding of the framework at hand. It provides an in-depth explanation of modified nodal analysis and its discretization methods. Furthermore, the integration of modified nodal analysis into state-space modeling is given. In here, the solving issue with nonlinear components is addressed and demonstrated for three circuits with various nonlinear components. Finally, recent computational improvements for the handling of potentiometers and the integration of operational amplifiers are explained.

3.1 Modified Nodal Analysis

Basic nodal analysis calculates the voltage at each node in a memoryless, linear circuit by using a combination of Kirchhoff's Current Law (KCL) and Kirchhoff's Voltage Law for each node to represent the circuit in a linear matrix form, which can be solved with respect to the voltage at each node [16]. An extension of this circuit, Modified Nodal Analysis (MNA), was developed to handle nonlinear currents and state-based components, and can in its most basic form be expressed as

$$\Theta \boldsymbol{\varphi} = \boldsymbol{c} \quad (3.1)$$

where, given Φ nodes in total, $\Theta \in \mathbb{R}^{\Phi \times \Phi}$ is a conductance matrix, $\boldsymbol{\varphi} \in \mathbb{R}^{\Phi \times 1}$ represent the voltages at each node and $\boldsymbol{c} \in \mathbb{R}^{\Phi \times 1}$ represent the current sources into each node [23]. By separating the currents in \boldsymbol{c} into components depending on input sources, state-based components and nonlinear components, we arrive at the form

$$\Theta \boldsymbol{\varphi} + \boldsymbol{M}_u^T \boldsymbol{i}_s = \boldsymbol{M}_x^T \boldsymbol{x} + \boldsymbol{M}_n^T \boldsymbol{i}_n \quad (3.2)$$

where $\boldsymbol{i}_s \in \mathbb{R}^{n_u \times 1}$ are the currents from the voltage sources, $\boldsymbol{x} \in \mathbb{R}^{n_x \times 1}$ the currents from states, $\boldsymbol{i}_n \in \mathbb{R}^{n_n \times 1}$ the currents from nonlinear components and $\boldsymbol{M}_u \in \mathbb{Z}^{n_u \times \Phi}$, $\boldsymbol{M}_x \in \mathbb{Z}^{n_x \times \Phi}$, $\boldsymbol{M}_n \in \mathbb{Z}^{n_n \times \Phi}$ are incidence matrices, where each column represent a node and each row represent a specific component (in total n_u sources, n_x state-based components and n_n nonlinear components) [10]. The values at the incidence matrices will either be -1 or 1 for each leg of the component connected to a specific node or 0 otherwise. The ground node is not represented in the matrix.

Equation 3.2 can be extended into a state-space compatible form by rewriting the system as

$$S \begin{bmatrix} \varphi \\ i_s \end{bmatrix} = \begin{bmatrix} M_x^T \\ 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ I \end{bmatrix} u + \begin{bmatrix} M_n^T \\ 0 \end{bmatrix} i_n \quad (3.3)$$

where the new conductance matrix $S \in \mathbb{R}^{(\Phi+n_u) \times (\Phi+n_u)}$ (where n_u denotes the number of source voltages) is defined by

$$S = \begin{bmatrix} \Theta & M_u^T \\ M_u & 0 \end{bmatrix} \quad (3.4)$$

and $u \in \mathbb{R}^{n_u \times 1}$ is the source voltages and each of the previously existing matrices have been concatenated with zero matrices of respective sizes [10]. The only difference between equation 3.2 and 3.3 is that the final rows of the expanded system will read

$$\begin{aligned} \varphi_{u_1} &= u_1 \\ &\vdots \\ \varphi_{u_n} &= u_n \end{aligned}$$

with φ_u being the potential of a node connected to a voltage source, thus linking the source voltages of the system to their respective nodes.

3.2 Discretization

To be able to calculate the currents over the state-based elements, which in this case is the capacitor and the inductor, these have to be discretized. This is done as in [10], using the bilinear transform, giving the final state update equations

$$x_C[n] = 2\frac{2C}{T}v_C[n] - x_C[n-1] \quad (3.5a)$$

$$x_L[n] = 2\frac{T}{2L}v_L[n] + x_L[n-1] \quad (3.5b)$$

where $x_C[n]$ and $x_L[n]$ denotes the current stored in a capacitor respectively an inductor at time step n , $v_C[n]$ and $v_L[n]$ the voltage over the component, C the capacitance value in farad, L the inductance in henry and T the sampling period in samples, and the current passing through the component at time step n defined by

$$i_C[n] = \frac{v_C[n]}{R_C} - x_C[n-1], \quad R_C = \frac{T}{2C} \quad (3.6a)$$

$$i_L[n] = \frac{v_L[n]}{R_L} - x_L[n-1], \quad R_L = \frac{2L}{T}. \quad (3.6b)$$

The discretized capacitor/inductor can then be separated into a state-based component and a conductance-based component.

3.3 Generating an NDK-Model from a Circuit

While the method described in section 3.1 works for manually deriving the NDK-model from a circuit, a more rigorous approach is required for automating the generation of models of circuits. This section describes how to generate the relevant NDK-matrices from a netlist.

Based on the method used in [10], a set of incidence matrices are defined:

$$\mathbf{N}_r \in \mathbb{Z}^{n_r \times \Phi} \quad (3.7)$$

$$\mathbf{N}_x \in \mathbb{Z}^{n_x \times \Phi} \quad (3.8)$$

$$\mathbf{N}_v \in \mathbb{Z}^{n_v \times \Phi} \quad (3.9)$$

$$\mathbf{N}_u \in \mathbb{Z}^{n_u \times \Phi} \quad (3.10)$$

$$\mathbf{N}_n \in \mathbb{Z}^{n_n \times \Phi} \quad (3.11)$$

$$\mathbf{N}_o \in \mathbb{Z}^{n_o \times \Phi} \quad (3.12)$$

where n_r correspond to the number of resistors in the circuit, n_x the number of state-based elements, n_v the number of potentiometer-related resistors (each potentiometer counts as two resistors), n_u the number of inputs, n_n the number of nonlinear equations (one component can represent multiple nonlinear functions, see section 3.6) and finally Φ represent the total number of nodes in the system. With the exception of nonlinear components, which are slightly more complicated, these matrices will for each component k have the following values:

$$(\mathbf{N})_{k, \varphi_{\text{from}}} = 1 \quad (3.13)$$

$$(\mathbf{N})_{k, \varphi_{\text{to}}} = -1 \quad (3.14)$$

with all other entries 0, where φ_{from} denote the node index of the first leg represented in the netlist and φ_{to} the second. This behaviour is similar to the \mathbf{M} -matrices in section 3.1, but the sign is not necessarily the same, which will be discussed in-depth in section 3.6.

In addition to these, the conductance of the resistors, state-based elements and the resistance are stored in the diagonal matrices $\mathbf{G}_r \in \mathbb{R}^{n_r \times n_r}$, $\mathbf{G}_x \in \mathbb{R}^{n_x \times n_x}$ and $\mathbf{R}_v \in \mathbb{R}^{n_v \times n_v}$ respectively, i.e.

$$\mathbf{G}_r = \text{diag} \left(\frac{1}{R_1}, \dots, \frac{1}{R_{n_r}} \right) \quad (3.15)$$

$$\mathbf{G}_x = \text{diag} (X_1, \dots, X_{n_x}) \quad (3.16)$$

$$\mathbf{R}_v = \text{diag} (R_1, \dots, R_{n_v}) \quad (3.17)$$

$$(3.18)$$

where X_k for component k is defined as $2C_k/T$ for capacitors and $T/2L_k$ for inductors [10].

Finally, a diagonal matrix $\mathbf{Z} \in \mathbb{Z}^{n_x \times n_x}$ is generated to store the sign of equations 3.5a and 3.5b, i.e. for state-based element index k ,

$$(\mathbf{Z})_{k,k} = 1 \quad \text{if capacitor} \quad (3.19)$$

$$(\mathbf{Z})_{k,k} = -1 \quad \text{if inductor.} \quad (3.20)$$

With the exception of the information required to represent the nonlinear currents, and the content of N_n which will be discussed in section 3.6, all necessary information to represent the circuit is now accounted for. Equation 3.2 can then be written as

$$\Theta \boldsymbol{\varphi} + N_u^T \mathbf{i}_s = N_x^T \mathbf{x} + N_n^T \mathbf{i}_n, \quad (3.21)$$

where

$$\Theta = N_R^T G_R N_R + N_v^T R_v^{-1} N_v + N_x^T G_x N_x. \quad (3.22)$$

From this, the final system, describing the entire nodal analysis can be generated:

$$S \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} = \begin{bmatrix} N_x^T \\ \mathbf{0} \end{bmatrix} \mathbf{x} + \begin{bmatrix} \mathbf{0} \\ I \end{bmatrix} \mathbf{u} + \begin{bmatrix} N_n^T \\ \mathbf{0} \end{bmatrix} \mathbf{i}_n \quad (3.23)$$

where I is the identity matrix ($I \in \mathbb{Z}^{n_u \times n_u}$) and

$$S = \begin{bmatrix} \Theta & N_u^T \\ N_u & \mathbf{0} \end{bmatrix}. \quad (3.24)$$

Once again, note that this is not strictly equivalent to the equations in section 3.1, as the manually calculated matrices M may differ in sign from the algorithmically generated matrices N .

3.4 State-space Model

The NDK state-space system contains three main update equations:

$$\mathbf{x}[n] = \mathbf{A}\mathbf{x}[n-1] + \mathbf{B}\mathbf{u}[n] + \mathbf{C}\mathbf{i}_n[n] \quad (3.25a)$$

$$\mathbf{y}[n] = \mathbf{D}\mathbf{x}[n-1] + \mathbf{E}\mathbf{u}[n] + \mathbf{F}\mathbf{i}_n[n] \quad (3.25b)$$

$$\mathbf{v}_n[n] = \mathbf{G}\mathbf{x}[n-1] + \mathbf{H}\mathbf{u}[n] + \mathbf{K}\mathbf{i}_n[n]. \quad (3.25c)$$

Given the full MNA-model from 3.23, the relevant voltages for each of the three components can be extracted using the corresponding N -matrix:

$$\mathbf{v}_x = [N_x \quad \mathbf{0}] \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} = [N_x \quad \mathbf{0}] S^{-1} \left(\begin{bmatrix} N_x^T \\ \mathbf{0} \end{bmatrix} \mathbf{x} + \begin{bmatrix} \mathbf{0} \\ I \end{bmatrix} \mathbf{u} + \begin{bmatrix} N_n^T \\ \mathbf{0} \end{bmatrix} \mathbf{i}_n \right) \quad (3.26a)$$

$$\mathbf{v}_o = [N_o \quad \mathbf{0}] \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} = [N_o \quad \mathbf{0}] S^{-1} \left(\begin{bmatrix} N_x^T \\ \mathbf{0} \end{bmatrix} \mathbf{x} + \begin{bmatrix} \mathbf{0} \\ I \end{bmatrix} \mathbf{u} + \begin{bmatrix} N_n^T \\ \mathbf{0} \end{bmatrix} \mathbf{i}_n \right) \quad (3.26b)$$

$$\mathbf{v}_n = [N_n \quad \mathbf{0}] \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} = [N_n \quad \mathbf{0}] S^{-1} \left(\begin{bmatrix} N_x^T \\ \mathbf{0} \end{bmatrix} \mathbf{x} + \begin{bmatrix} \mathbf{0} \\ I \end{bmatrix} \mathbf{u} + \begin{bmatrix} N_n^T \\ \mathbf{0} \end{bmatrix} \mathbf{i}_n \right) \quad (3.26c)$$

where $\mathbf{v}_x \in \mathbb{R}^{n_x \times 1}$ are the voltages over the state-based components, $\mathbf{v}_o \in \mathbb{R}^{n_o \times 1}$ are the voltages over the outputs and $\mathbf{v}_n \in \mathbb{R}^{n_n \times 1}$ are the voltages over the nonlinear components [10]. Simplifying the discretization from equation 3.5 (here repeated in matrix form)

$$\mathbf{x}[n] = \mathbf{Z} (2\mathbf{G}_x \mathbf{v}_x[n] - \mathbf{x}[n-1]) \quad (3.27)$$

gives an expression for $v_x[n]$ which can be inserted into equation 3.26a:

$$(2\mathbf{Z}\mathbf{G}_x)^{-1} (x[n] + \mathbf{Z}x[n-1]) = [\mathbf{N}_x \quad \mathbf{0}] \mathbf{S}^{-1} \left(\begin{bmatrix} \mathbf{N}_x^T \\ \mathbf{0} \end{bmatrix} x[n-1] + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} u[n] + \begin{bmatrix} \mathbf{N}_n^T \\ \mathbf{0} \end{bmatrix} i_n[n] \right) \quad (3.28)$$

from which the state update equation of 3.25a can be derived

$$x[n] = 2\mathbf{Z}\mathbf{G}_x [\mathbf{N}_x \quad \mathbf{0}] \mathbf{S}^{-1} \left(\begin{bmatrix} \mathbf{N}_x^T \\ \mathbf{0} \end{bmatrix} x[n-1] + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} u[n] + \begin{bmatrix} \mathbf{N}_n^T \\ \mathbf{0} \end{bmatrix} i_n[n] \right) - \mathbf{Z}x[n-1] \quad (3.29)$$

which finally gives the matrices A , B and C :

$$A = 2\mathbf{Z}\mathbf{G}_x [\mathbf{N}_x \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{N}_x^T \\ \mathbf{0} \end{bmatrix} - \mathbf{Z} \quad (3.30a)$$

$$B = 2\mathbf{Z}\mathbf{G}_x [\mathbf{N}_x \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \quad (3.30b)$$

$$C = 2\mathbf{Z}\mathbf{G}_x [\mathbf{N}_x \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{N}_n^T \\ \mathbf{0} \end{bmatrix}. \quad (3.30c)$$

Similarly, the matrices $D - K$ can be derived directly from equations 3.26b (as $y = v_o$) and 3.26c:

$$D = [\mathbf{N}_o \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{N}_x^T \\ \mathbf{0} \end{bmatrix} \quad (3.31a)$$

$$E = [\mathbf{N}_o \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \quad (3.31b)$$

$$F = [\mathbf{N}_o \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{N}_n^T \\ \mathbf{0} \end{bmatrix}. \quad (3.31c)$$

$$G = [\mathbf{N}_n \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{N}_x^T \\ \mathbf{0} \end{bmatrix} \quad (3.31d)$$

$$H = [\mathbf{N}_n \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \quad (3.31e)$$

$$K = [\mathbf{N}_n \quad \mathbf{0}] \mathbf{S}^{-1} \begin{bmatrix} \mathbf{N}_n^T \\ \mathbf{0} \end{bmatrix}. \quad (3.31f)$$

3.5 Solving the System

At time step n , the values of the input vector $u[n]$ and the previous state $x[n]$ are known, as well as all the matrices $A - K$. The only value missing to solve equation 3.25 is the value of $i_n[n]$. The content of i_n depends on which nonlinear components are used, with the main requirement being that it is a function of the nonlinear voltages [23], i.e. $i_n[n] = i_n(v_n[n])$. Equation 3.25c can then be written as

$$\begin{cases} p[n] = \mathbf{G}x[n-1] + \mathbf{H}u[n] \\ v_n[n] = p[n] + \mathbf{K}i_n(v_n[n]) \end{cases} \quad (3.32)$$

where $p[n]$ is a vector of known values at time step n . This can be defined as a function f_n so that

$$f_n(v_n[n]) = p[n] + K i_n(v_n[n]) - v_n[n] = \mathbf{0} \quad (3.33)$$

and its corresponding jacobian J_f :

$$J_f(v_n[n]) = K J_{i_n}(v_n[n]) - I \quad (3.34)$$

where $J_f(v_n[n])$ is the jacobian of $f_n(v_n[n])$, $J_{i_n}(v_n[n])$ is the jacobian of i_n at time step n and I is the identity matrix corresponding to the number of nonlinear functions n_n . This system can be solved using Newton iteration, i.e. for a vector $v = v_n[n]$

$$v_1 = v_0 - J_f^{-1}(v_0) f(v_0) \quad (3.35a)$$

$$v_{i+1} = v_i - J_f^{-1}(v_i) f(v_i) \quad (3.35b)$$

where v_0 at each time step is set to the calculated value at the previous time step, i.e. $v_0 = v_n[n-1]$. This iterative process is repeated until convergence, at which point this approximated voltage is used for $v_n[n]$ to calculate $i_n[n]$ at time step n .

Once $i_n[n]$ has been calculated, the output $y[n]$ is calculated using equation 3.25b and the final step at time step n is to calculate the state update using equation 3.25a.

3.6 Nonlinear Components

The manual MNA analysis described in section 3.1 shows the actual behaviour of the circuit. The forms from equations 3.2 and 3.3 will in this section be denoted as M -form, with the corresponding conductance matrix S_M while the generated matrices from section 3.3, which will be denoted as N -form, with the corresponding conductance matrix S_N . Capacitors and inductors will be discretized using equation 3.6, but will be denoted $x_c = x_c[n-1]$ and $v_c = v_c[n]$. The matrices will in both cases be identical in which indices are nonzero, but the signs may differ. To compensate for this, the direction of the current of each nonlinear component must be adjusted. To do this, a test circuit has to be evaluated both manually and algorithmically. This section will describe this procedure for three nonlinear components: a diode, a NPN transistor and a triode. The circuits used are not necessarily audio circuits, but merely serve to derive the behaviour of the specific nonlinear component present in each circuit.

3.6.1 The Diode

For calculating the current through the diode, Shockley's diode equation will be used:

$$i_D = I_s \left(e^{\frac{v}{V_t}} - 1 \right) \quad (3.36)$$

where I_s denotes the saturation current, V_t the thermal voltage and v the voltage over the diode [18]. The circuit used to compare the M - and N -forms will be a diode clipper, represented in figure 3.1.

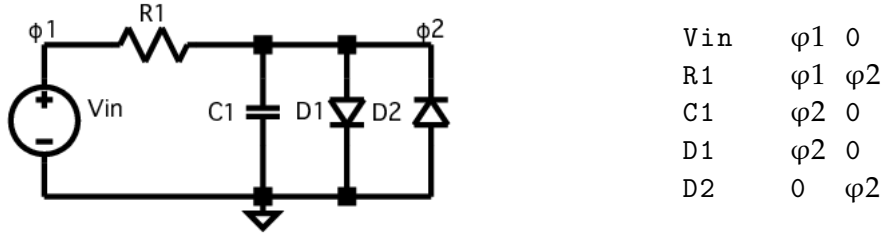


Figure 3.1: Schematic and netlist of a diode clipper

Using KCL at each node gives the equations for the voltages (for simplicity's sake written so that inputs to each node are on the lhs and outputs on the rhs of each equation):

$$\begin{cases} i_{Vin} = \frac{\varphi_1 - \varphi_2}{R_1} \\ \frac{\varphi_1 - \varphi_2}{R_1} + i_{D_2} = i_{C_1} + i_{D_1} \end{cases} \quad (3.37)$$

which when discretized as in section 3.2, is equivalent to

$$\begin{cases} i_{Vin} = \frac{\varphi_1 - \varphi_2}{R_1} \\ \frac{\varphi_1 - \varphi_2}{R_1} + i_{D_2}(\varphi_2 - 0) = \frac{\varphi_2 - 0}{R_{C_1}} - x_{C_1} + i_{D_1}(0 - \varphi_2). \end{cases} \quad (3.38)$$

which can be written according to equation 3.2 as

$$\begin{bmatrix} -G_R & G_R \\ G_R & -G_R - G_x \end{bmatrix} \begin{bmatrix} \varphi_1 \\ \varphi_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} i_s = \begin{bmatrix} 0 \\ -1 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} i_{D_1}(\varphi_2) \\ i_{D_2}(-\varphi_2) \end{bmatrix} \quad (3.39)$$

where $G_R = 1/R_1$ and $G_x = 1/R_{C_1}$ and finally according to equation 3.3

$$\underbrace{\begin{bmatrix} -G_R & G_R & 1 \\ G_R & -G_R - G_x & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{S_M} \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ i_{Vin} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} M_x^T \\ 0 \end{bmatrix}} x + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{\begin{bmatrix} 0 \\ I \end{bmatrix}} u + \underbrace{\begin{bmatrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{bmatrix}}_{\begin{bmatrix} M_n^T \\ 0 \end{bmatrix}} \begin{bmatrix} i_{D_1}(\varphi_2) \\ i_{D_2}(-\varphi_2) \end{bmatrix}. \quad (3.40)$$

Note that in this specific case, current does not actually flow into the node at φ_2 from ground, but will be represented that way to be able to describe the behaviour of i_{D_2} using the Shockley Diode equation.

The N -form will be derived directly from the netlist according to the rules described in section 3.3, where (identically to any other two-legged component)

$$(N_n)_{k, \varphi_{from}} = 1 \quad (3.41)$$

$$(N_n)_{k, \varphi_{to}} = -1 \quad (3.42)$$

for the k :th nonlinear component, which in this specific circuit gives

$$N_R = \begin{bmatrix} 1 & -1 \end{bmatrix} \quad N_x = \begin{bmatrix} 0 & 1 \end{bmatrix} \quad N_u = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad N_o = \begin{bmatrix} 0 & 1 \end{bmatrix} \quad (3.43)$$

$$N_n = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \quad G_R = \begin{bmatrix} \frac{1}{R_1} \end{bmatrix} \quad G_x = \begin{bmatrix} \frac{2C_1}{T} \end{bmatrix} \quad (3.44)$$

which, according to equation 3.23 can be concatenated into

$$\underbrace{\begin{bmatrix} G_R & -G_R & 1 \\ -G_R & G_R + G_x & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{S_N} \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} = \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} N_x^T \\ \mathbf{0} \end{bmatrix}} \mathbf{x} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{\begin{bmatrix} \mathbf{0} \\ I \end{bmatrix}} \mathbf{u} + \underbrace{\begin{bmatrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{bmatrix}}_{\begin{bmatrix} N_n^T \\ \mathbf{0} \end{bmatrix}} \mathbf{i}_n. \quad (3.45)$$

Note that the difference between S_M and S_N is that the first two rows have the opposite signs. Multiplying the two first rows in equation 3.40 by -1 gives

$$\underbrace{\begin{bmatrix} G_R & -G_R & -1 \\ -G_R & G_R + G_x & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{S_M} \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ i_{\text{Vin}} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} M_x^T \\ \mathbf{0} \end{bmatrix}} \mathbf{x} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{\begin{bmatrix} \mathbf{0} \\ I \end{bmatrix}} \mathbf{u} + \underbrace{\begin{bmatrix} 0 & 0 \\ -1 & 1 \\ 0 & 0 \end{bmatrix}}_{\begin{bmatrix} M_n^T \\ \mathbf{0} \end{bmatrix}} \begin{bmatrix} i_{D_1}(\varphi_2) \\ i_{D_2}(-\varphi_2) \end{bmatrix}. \quad (3.46)$$

which, apart from the first row of S_M and the entire M_n is identical to equation 3.45. This can be remedied by defining the currents in 3.45 so that

$$\mathbf{i}_s = [-i_{\text{Vin}}] \quad (3.47)$$

$$\mathbf{i}_n = \begin{bmatrix} -i_{D_1}(\varphi_2) \\ -i_{D_2}(-\varphi_2) \end{bmatrix} \quad (3.48)$$

Now the two systems are identical. Note that \mathbf{i}_n will in practice not be a function of node voltages $\boldsymbol{\varphi}$ but declare its own nonlinear voltages \mathbf{v}_n which is described in equation 3.25c. To verify that the nodes used in equation 3.38 voltages are the same as the nonlinear voltages declared, we simply check the left hand side of 3.25c:

$$\begin{aligned} \mathbf{v}_n &= [N_n \quad \mathbf{0}] \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ -i_{\text{Vin}} \end{bmatrix} \\ &= \begin{bmatrix} \varphi_2 \\ -\varphi_2 \end{bmatrix} \end{aligned} \quad (3.49)$$

which indeed gives the correct voltages over the respective components. The final equations for the nonlinear currents in the diode clipper circuit are thus

$$\mathbf{i}_n(\mathbf{v}_n) = \begin{bmatrix} -I_s \left(e^{\frac{v_{n,1}}{V_t}} - 1 \right) \\ -I_s \left(e^{\frac{v_{n,2}}{V_t}} - 1 \right) \end{bmatrix}, \quad \begin{bmatrix} v_{n,1} \\ v_{n,2} \end{bmatrix} = \begin{bmatrix} \varphi_2 \\ -\varphi_2 \end{bmatrix} \quad (3.50)$$

From this, the general behaviour for a diode at nonlinear component index k with its corresponding nonlinear voltage $v_{n,m}$ can be described by

$$i_{n,k}(v_{n,m}) = -I_s \left(e^{\frac{v_{n,m}}{V_t}} - 1 \right) \quad (3.51)$$

and the jacobian J_{i_n} described in equation 3.34 will have the partial derivative of $i_{n,k}$ added, i.e.

$$(J_{i_n})_{k,m} = (J_{i_n})_{k,m} + \frac{\partial i_{n,k}}{\partial v_{n,m}}. \quad (3.52)$$

3.6.2 NPN Transistor

For calculating the current through the NPN transistor, the Ebers-Moll equations [18] will be used, but handled differently than in [10]:

$$i_B = I_s \left[\frac{1}{\beta_F} \left(e^{\frac{v_{BE}}{V_t}} - 1 \right) + \frac{1}{\beta_R} \left(e^{\frac{v_{BC}}{V_t}} - 1 \right) \right] \quad (3.53a)$$

$$i_E = I_s \left[\left(e^{\frac{v_{BE}}{V_t}} - e^{\frac{v_{BC}}{V_t}} \right) + \frac{1}{\beta_F} \left(e^{\frac{v_{BE}}{V_t}} - 1 \right) \right] \quad (3.53b)$$

$$i_C = i_E - i_B \quad (3.53c)$$

where i_B, i_E, i_C denote the currents through the base, emitter and collector respectively while β_F, β_R, I_s and V_t are parameters of the transistor and v_{BE} and v_{BC} are the base-emitter voltages and base-collector voltages respectively.

The circuit used to comparing the M - and N -forms will be a simple NPN test circuit, shown in figure 3.2. This circuit does not have any meaningful audio purpose and is only used for its relative simplicity, as once the behaviour of a component in a specific circuit is derived, that model can be used in any circuit.

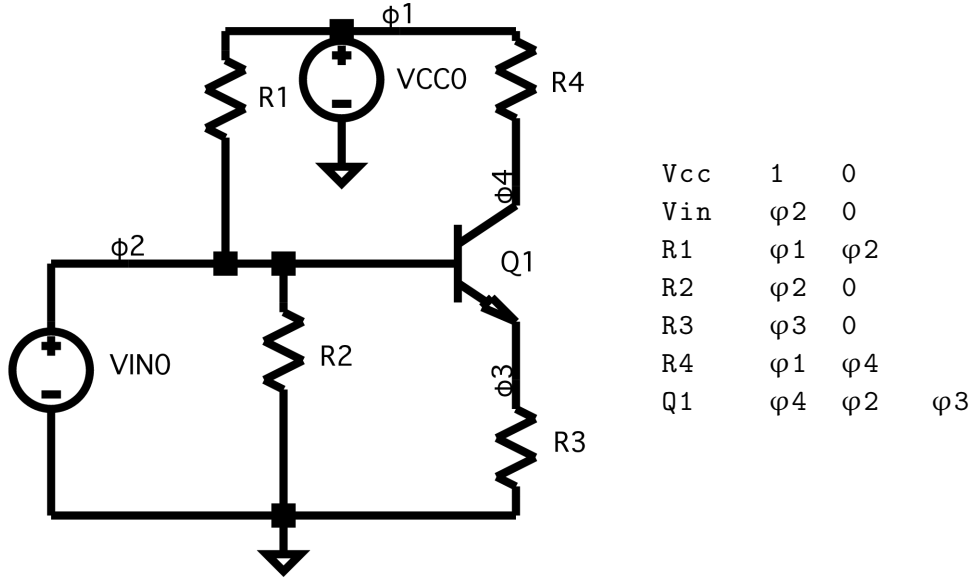


Figure 3.2: Schematic and netlist of an NPN test circuit

KCL gives four main equations in this case:

$$\begin{cases} i_{V_{cc}} = \frac{\varphi_1 - \varphi_2}{R_1} + \frac{\varphi_1 - \varphi_4}{R_4} \\ i_{V_{in}} + \frac{\varphi_1 - \varphi_2}{R_1} = \frac{\varphi_2}{R_2} + i_B \\ i_E = \frac{\varphi_3}{R_3} \\ \frac{\varphi_1 - \varphi_4}{R_4} = i_C \end{cases} \quad (3.54)$$

which in the M -form results in the following system

$$S_M \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \\ i_{V_{in}} \\ i_{V_{cc}} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_{in} \\ v_{cc} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ -1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_B \\ i_E \end{bmatrix} \quad (3.55)$$

where

$$S_M = \begin{bmatrix} -G_{R_1} - G_{R_4} & G_{R_1} & 0 & G_{R_4} & 0 & 1 \\ G_{R_1} & -G_{R_1} - G_{R_2} & 0 & 0 & 1 & 0 \\ 0 & 0 & G_{R_3} & 0 & 0 & 0 \\ G_{R_4} & 0 & 0 & -G_{R_4} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \quad (3.56)$$

The process of deriving the N -form of the circuit is identical to the steps in section 3.6.1, with the extension that for the three-legged NPN transistor, there are two separate equations over two pairs of legs, and the indices in the N_n matrix are for a NPN transistor as the k :th nonlinear component set to:

$$\begin{aligned} (N_n)_{k, \varphi_C} &= 1 \\ (N_n)_{k, \varphi_B} &= -1 \\ (N_n)_{k+1, \varphi_C} &= 1 \\ (N_n)_{k+1, \varphi_E} &= -1 \end{aligned} \quad (3.57)$$

where $\varphi_C, \varphi_B, \varphi_E$ denote the nodes where the collector, base and emitter respectively are connect to. This is analogous to defining two nonlinear voltages, one from collector to base (v_{CB}) and one from collector to emitter (v_{CE}). Deriving the N -form from the netlist gives

the following system:

$$S_N \begin{bmatrix} \boldsymbol{\varphi} \\ \mathbf{i}_s \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{u} + \begin{bmatrix} 0 & 0 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \mathbf{i}_n, \quad (3.58)$$

$$S_N = \begin{bmatrix} G_{R_1} + G_{R_4} & -G_{R_1} & 0 & -G_{R_4} & 0 & -1 \\ -G_{R_1} & G_{R_1} + G_{R_2} & 0 & 0 & -1 & 0 \\ 0 & 0 & G_{R_3} & 0 & 0 & 0 \\ -G_{R_4} & 0 & 0 & G_{R_4} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

which means that the first, second and fourth row of the M -form must be multiplied with -1, resulting in identical systems apart from

$$M_n = \begin{bmatrix} 0 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad N_n = \begin{bmatrix} 0 & 0 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (3.59)$$

and the currents in \mathbf{i}_s , which can be resolved by defining

$$\mathbf{i}_n = \begin{bmatrix} i_B \\ -i_E \end{bmatrix}, \quad \mathbf{i}_s = \begin{bmatrix} -i_{V_{in}} \\ -i_{V_{cc}} \end{bmatrix}. \quad (3.60)$$

As in equation 3.49 in section 3.6.1, equation 3.57 defines over which nodes the voltage is taken. The first two equations define a nonlinear voltage between the collector and base, and the second two a nonlinear voltage between the collector and emitter. More formally, this means that for a nonlinear NPN transistor at index k ,

$$v_{n,k} = v_{CB} \quad (3.61)$$

$$v_{n,k+1} = v_{CE} \quad (3.62)$$

which means that equation 3.53a must be redefined in terms of the relevant voltages:

$$v_{BE} = v_{CE} - v_{CB} = v_2 - v_1 \quad (3.63)$$

$$v_{BC} = -v_{CB} = -v_1 \quad (3.64)$$

which in this circuit gives the final equations

$$i_{n,1} = I_s \left[\frac{1}{\beta_F} \left(e^{\frac{v_2 - v_1}{V_t}} - 1 \right) + \frac{1}{\beta_R} \left(e^{\frac{-v_1}{V_t}} - 1 \right) \right] \quad (3.65)$$

$$i_{n,2} = -I_s \left[\left(e^{\frac{v_2 - v_1}{V_t}} - e^{\frac{-v_1}{V_t}} \right) + \frac{1}{\beta_F} \left(e^{\frac{v_2 - v_1}{V_t}} - 1 \right) \right] \quad (3.66)$$

and for a general NPN transistor at index k with its respective voltages $v_{n,m}, v_{n,m+1}$

$$i_{n,k} = I_s \left[\frac{1}{\beta_F} \left(e^{\frac{v_{m+1}-v_m}{V_t}} - 1 \right) + \frac{1}{\beta_R} \left(e^{\frac{-v_m}{V_t}} - 1 \right) \right] \quad (3.67)$$

$$i_{n,k+1} = -I_s \left[\left(e^{\frac{v_{m+1}-v_m}{V_t}} - e^{\frac{-v_m}{V_t}} \right) + \frac{1}{\beta_F} \left(e^{\frac{v_{m+1}-v_m}{V_t}} - 1 \right) \right] \quad (3.68)$$

and in total four entries for the jacobian:

$$(J_{i_n})_{k,m} = (J_{i_n})_{k,m} + \frac{\partial i_{n,k}}{\partial v_{n,m}} \quad (3.69)$$

$$(J_{i_n})_{k,m+1} = (J_{i_n})_{k,m+1} + \frac{\partial i_{n,k}}{\partial v_{n,m+1}} \quad (3.70)$$

$$(J_{i_n})_{k+1,m} = (J_{i_n})_{k+1,m} + \frac{\partial i_{n,k+1}}{\partial v_{n,m}} \quad (3.71)$$

$$(J_{i_n})_{k+1,m+1} = (J_{i_n})_{k+1,m+1} + \frac{\partial i_{n,k+1}}{\partial v_{n,m+1}}. \quad (3.72)$$

3.6.3 Triode

The third and final nonlinear component derived and implemented in this framework is the Triode. Built on the physical triode model described in [3], the triode's behaviour can be described by

$$i_G = G_G \left[\log(1 + \exp(C_G v_{GK})) \frac{1}{C_G} \right]^{\xi} + I_{G_0} \quad (3.73)$$

$$i_K = G \left[\log \left(1 + \exp \left(C \left(\frac{1}{\mu} v_{AK} + v_{GK} \right) \right) \right) \frac{1}{C} \right]^{\gamma} \quad (3.74)$$

$$i_A = i_K - i_G \quad (3.75)$$

where i_K, i_G, i_A are the currents through the cathode, grid and anode respectively, v_{AK}, v_{GK} are the voltages over the anode to cathode and grid to cathode respectively, and $G, G_G, C, C_G, \mu, \gamma, \xi, I_{G_0}$ are parameters of the triode.

Using the same procedure as in section 3.6.2 on the circuit and netlist of figure 3.3, with the N_n matrix indices defined as follows for a triode as the k :th nonlinear component:

$$(N_n)_{k,\varphi_A} = 1 \quad (3.76)$$

$$(N_n)_{k,\varphi_G} = -1$$

$$(N_n)_{k+1,\varphi_A} = 1$$

$$(N_n)_{k+1,\varphi_G} = -1$$

where $\varphi_A, \varphi_G, \varphi_K$ denote the nodes which the anode, grid and cathode respectively are connected to, the final representation of this circuit requires the following definitions:

$$\mathbf{i}_n = \begin{bmatrix} -i_G \\ -i_K \end{bmatrix}, \quad \mathbf{i}_s = \begin{bmatrix} -i_{Vin} \\ -i_{Vcc} \end{bmatrix} \quad (3.77)$$

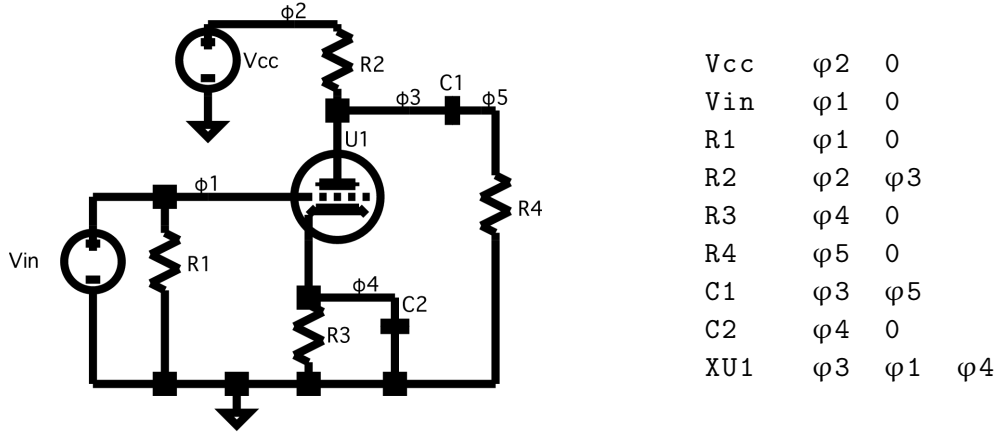


Figure 3.3: Schematic and netlist of a common cathode amplifier

with

$$\mathbf{v}_n = \begin{bmatrix} v_{AG} \\ v_{AK} \end{bmatrix}, \quad v_{GK} = v_{AK} - v_{AG} \quad (3.78)$$

where v_{AG} is the voltage over the anode to grid. This gives the final (and generalized) equations for a triode as nonlinear component at index k and corresponding voltages $v_{n,m} = v_{AG}, v_{n,m+1} = v_{AK}$:

$$i_{n,k} = -G_G \left[\log(1 + \exp(C_G(v_{n,m+1} - v_{n,m}))) \frac{1}{C_G} \right]^\xi + I_{G_0} \quad (3.79)$$

$$i_{n,k+1} = -G \left[\log \left(1 + \exp \left(C \left(\frac{1}{\mu} v_{n,m+1} + v_{n,m+1} - v_{n,m} \right) \right) \right) \frac{1}{C} \right]^\gamma \quad (3.80)$$

with the same four updates of the jacobian as in equation 3.69.

3.7 Operational Amplifiers

In [13], it is described how an operational amplifier can be implemented in the automated NDK-framework as a linear component. To do this, the NDK-framework is extended with two additional incidence matrices: $N_{\text{opaO}} \in \mathbb{Z}^{n_{\text{opa}} \times \Phi}$, which describes the output of the operational amplifiers, where n_{opa} is the total number of opamps in the circuit. N_{opaO} only has a single non-zero value per row, which specifies the node that the output is connected to, i.e.

$$(N_{\text{opaO}})_{k, \varphi_{\text{out}}} = 1 \quad (3.81)$$

where φ_{out} denotes the index of the node to which the opamp's output is connected. The second incidence matrix is $N_{\text{opaI}} \in \mathbb{Z}^{n_{\text{opa}} \times \Phi}$, which specifies the inputs nodes of the operational amplifier. This matrix has two nonzero indices per row,

$$(N_{\text{opaI}})_{k, \varphi_I} = -1 \quad (3.82)$$

$$(N_{\text{opaI}})_{k, \varphi_{NI}} = 1 \quad (3.83)$$

where φ_I and φ_{NI} denote the indices of the nodes to which the opamp's inverting and noninverting input are connected to

A linear operational amplifier is modelled as voltage sources. This means that the matrices are concatenated into the S matrix, similar to other voltage sources. In other words, this redefines S so that

$$S = \begin{bmatrix} \Theta & N_u^T & N_{\text{opaO}}^T \\ N_u & 0 & 0 \\ N_{\text{opaO}} + \alpha N_{\text{opaI}} & 0 & 0 \end{bmatrix}, \quad (3.84)$$

so that the redefined $S \in \mathbb{R}^{(\Phi+n_u+n_{\text{opa}}) \times (\Phi+n_u+n_{\text{opa}})}$, and α is the open-load amplification factor.

This also requires zero-padding of all the matrices in equation 3.23 to match the size of the new system, as well as the u vector. The final behaviour of the linear operational amplifier in the NDK-framework is

$$\varphi_{\text{out}} = \alpha(\varphi_{NI} - \varphi_I). \quad (3.85)$$

3.8 Real-time Improvements for Potentiometers

The NDK-model defined so far has two stages: offline computation of matrices $A - K$ and real-time solving of equation 3.25. This becomes a problem when using any type of variable resistor, as changing a value in R_v would require recomputing S and hence all the matrices. Performance-wise, the main issue is that this would require inversion of S , which as one may recall is a $(\Phi + n_u) \times (\Phi + n_u)$ matrix (or with the operational amplifier extension, $(\Phi + n_u + n_{\text{opa}}) \times (\Phi + n_u + n_{\text{opa}})$) and hence the inversion will be very expensive for any but the smallest of circuits. This issue is solved in [10] by decomposing S so that

$$S = S_0 + \begin{bmatrix} N_v^T \\ 0 \end{bmatrix} R_v^{-1} \begin{bmatrix} N_v & 0 \end{bmatrix} \quad (3.86)$$

where S_0 now can be used to compute constant matrices $A_0 - K_0$ independent of R_v , and when a variable resistor is changed, the only inversion needed is that of R_v , which is significantly smaller than S , followed by real-time, low cost updates of $A - K$ based on $A_0 - K_0$ and the inverted R_v .

The calculations of each of the offline matrices and the real-time update equations will not be described here as this procedure is thoroughly described in [10].

Chapter 4

Implementation

The implementation described in this chapter can be seen as a framework for rapid prototyping of audio circuits using the NDK-method. The environment is built around three main components:

- The parser: a Python script which converts SPICE netlists into the matrices needed for the NDK-model,
- A C++ library, which can be imported into JUCE to export real-time VST3/AU plugins directly from a parsed netlist,
- A set of MATLAB functions for more intricate, offline debugging of a circuit described by a parsed netlist.

In its current state, any circuit consisting of components mentioned in chapter 3 can be implemented using the environment, but new models, new components and even new nonlinear solvers can be implemented without modifying any of the core content of the C++ library due to its modular design.

This chapter will describe the procedure of generating an audio plugin step by step, from parsing the netlist to the final VST3/AU plugin.

4.1 Parser

The parser is implemented as a Python script which converts a netlist into a JSON-file containing all the matrices N, G, R, Z , the vector u described in section 3.3, the nonlinear model names, as well as the values needed to calculate G_x . Since these matrices can be calculated independent of export platform, the sampling rate is not set at this point, which is why the values of G_x cannot be calculated in the parser. The parser does not parse any parameters for nonlinear components (i.e. no `.model` directives), only the model name. The behaviour for each model is then implemented directly in the MATLAB and C++.

The syntax of the parser is built on a subset of SPICE's syntax for netlists with a few extensions to simplify parsing and adapt it to the NDK-framework. In a netlist, each component in the circuit is defined by a text string consisting of a, for each type of component, unique character (for example R for resistors), an optional name and a set of parameters

describing to which nodes it is connected and its values [2]. The entire syntax supported by the parser will be described in section 4.1.1.

4.1.1 Syntax

The basic building blocks for a linear audio circuit are the resistor, capacitor, inductor, voltage input and voltage supply. The difference between voltage input and voltage supply is that a voltage supply is treated as a constant stored inside the NDK-model, while input is any signal that can be fed into it. The syntax for these components is

```

Resistor      := R#<NAME>    N_FROM N_TO VALUE
Capacitor     := C#<NAME>    N_FROM N_TO VALUE
Inductor      := L#<NAME>    N_FROM N_TO VALUE
Voltage Input := VIN#<NAME>  N_FROM N_TO
Voltage Supply := VCC#<NAME> N_FROM N_TO <"DC"> VALUE

```

where # denotes a number (multiple digits are allowed), <·> denotes optional content, "X" denotes the actual string X and N_FROM and N_TO denotes the names of the nodes the component is connected from and to. The voltage input does not take a value, as this will be overwritten by the audio signal fed into it. The voltage input is also required to be 0-indexed with consecutive indexing for multiple inputs, as these values will be the positions that input signals are written to. All values support the suffixes T, G, Meg, k, m, u, μ , n as well as their corresponding units ohms, ohm, Farad, F, Henry, H, V, Volts. If a line is matched up to the entire syntax described here, any additional information, for example input directives such as SINE(0 1 20), will be discarded and the line will be successfully parsed.

Another linear component that isn't natively defined in the SPICE syntax is the potentiometer. A potentiometer model can be created by connecting two resistors r_1 and r_2 in series so that for a given total resistance of R_p , a parameter $\lambda \in [0, 1]$ can be used to control the two potentiometers so that

$$\begin{cases} r_1 = \lambda R_p \\ r_2 = (1 - \lambda) R_p \end{cases} \quad (4.1)$$

For the parser to be able to treat two resistors as a potentiometer, they must to be named

```

Potentiometer Top := RT#<NAME> N_FROM N_TO VALUE
Potentiometer Btm := RB#<NAME> N_FROM N_TO ,

```

where the value of the bottom potentiometer thus will be discarded while the value of the top potentiometer will be used as the the maximum value for the potentiometer. The indices # should begin from zero and must be matched between top and bottom potentiometer (e.g. pairs of RT0, RB0, RT1, RB1, ...) and the indexing must be consecutive, as these indices will be used in the C++ implementation for setting the values of the corresponding potentiometers.

Any alphanumeric node name is accepted as valid, but two nodes are treated as special:

```

Voltage Out := VOUT#<NAME>
Ground      := 0

```

where the output voltages must be consecutively numbered and zero-indexed, as these are the output indices that can be used to access output from the final NDK-model.

Finally, the nonlinear components (and the opamp) are defined by

```
Diode   := D#<NAME>      N_FROM N_TO MODEL
NPN     := Q#<NAME>      NC NB NE NS MODEL
Triode  := XU#<NAME>     NA NG NC MODEL
Opamp   := XUOPA#<NAME> NI I VCC VEE OUT MODEL .
```

Note that the NPN transistor has (and requires) four nodes: collector, base, emitter and substrate, but the substrate node will not be used in the framework.

4.1.2 Parsing procedure

The Python script takes two arguments: the path of the netlist input file and the path of the JSON output file. Each line in the netlist is parsed separately, storing the results as a list of components. This list is iterated over twice. The first time to count the number of each component type, get a set of all node names used in the circuit, which will be used for indexing and constructing the matrices. The ground node is removed from the set and the matrix N_o is generated based on the nodes named according to the Voltage Out naming rule. The second iteration is performed to populate the matrices according to the rules presented in section 3.3 and 3.7. An initial source vector $\mathbf{u} \in \mathbb{R}^{n_u \times 1}$ is also constructed by concatenating the voltage inputs, voltage supplies and opamp voltages (for the latter, refer back to 3.7), so that

$$(\mathbf{u})_k = \begin{cases} 0 & \text{if } k \in [0, \text{num.inputs}) \\ v_{cc,m} & \text{if } k \in [\text{num.inputs}, \text{num.supplies}) \\ 0 & \text{if } k \in [\text{num.supplies}, \text{num.opamps}) \end{cases}, \quad (4.2)$$

where $v_{cc,m}$ denotes the voltage of voltage supply index m and all the zeros only work as placeholders for the actual values that will be updated at each time step when running the model. Finally, all the matrices are added to a dictionary, which is written to the specified JSON-file. The JSON file can then be imported directly into the C++ framework described in section 4.2 to build an audio plugin, or to the MATLAB framework described in section 4.3 for more intricate debugging.

4.2 C++ Framework

The C++ library is designed as a set of classes that can be imported into a JUCE project and directly exported as an audio plugin. It can be separated into two main parts: offline computations, where the code is written to be as clear as possible to read, and real-time computations, which have been optimized for performance. For representing all the matrices, the open source linear algebra library Eigen was used [5].

While the C++ library is implemented as an extension to the JUCE framework, it does not depend on any JUCE internals, and could easily be used in any other environments.

4.2.1 Class Structure

Appendix A.1 contains a class diagram for the C++ library. The main classes in the diagram are `StateSpaceProcessor`, `CircuitParser`, `NonlinearComponentParser`, `ConstantMatrices` and `NonlinearSolver`. These will all be described in detail in the following two sections. At the center of the library is the `StateSpaceProcessor` class, which handles all other objects, matrices as well as the runtime simulation, and is thus the only class the end user interacts with.

Initialization - Offline processing

The `StateSpaceProcessor` is initialized in the `prepareToPlay`-function of the `PluginProcessor`, by passing the sampling rate and a path to the JSON-file for the circuit. The latter is passed on to the `CircuitParser`, which parses the N , G , R and Z matrices as well as the raw input vector u and nonlinear model names from the JSON-file into matrices.

These matrices are passed to the `ConstantMatrices` class, which calculates the matrices $A_0 - K_0$, as well as the other matrices referred to in section 3.8 that are needed to calculate the final matrices $A - K$.

The `NonlinearComponentParser` takes the list of nonlinear component models and generates a list of `NonlinearComponent` objects. The former is a separate class to simplify the process of adding new components to the library, as this is the only class that needs to be modified to parse new models.

Currently, three components have been implemented; The 2N2222 NPN transistor, the 1N914 diode and the 12AX7 tube triode. All of them extend the `NonlinearComponent` interface, thus implementing the functions `calculateCurrent` and `calculateJacobian`, but with different rules for each component, based on the rules described in section 3.6.

Finally, the initial values of R_v are set. Note that 0.001 is added to each value along the diagonal of R_v so that the matrix is guaranteed to be invertible even when a potentiometer is turned all the way in one direction. This is also the behaviour one would expect from an analog potentiometer, as the resistance would never be exactly 0. Once R_v is set, the matrices $A - K$ can be calculated and real-time processing can begin.

Process - Real-time processing

The process function of `StateSpaceProcessor` is the update function equivalent to equation 3.25. The function takes the current audio input as argument and stores it in the correct index of u . Firstly, the nonlinear currents are calculated by calling the `NonlinearSolver`'s `solve` function, which solves equation 3.25c. Currently, only the Newton-Raphson method is implemented as a nonlinear solver. This class uses the `calculateCurrent` and `calculateJacobian` functions of the `NonlinearComponent` classes to iteratively calculate the nonlinear voltages, and returns the currents once solved. Once the nonlinear currents have been calculated, the only remaining steps in the `solve` function are to compute the outputs according to equation 3.25b, the new state (which is stored in `StateSpaceProcessor`) and finally return the output.

For updating values of potentiometers in real-time, the `StateSpaceProcessor` has the function `updatePotValues`, which takes an index, corresponding to the potentiometer index from the parser, and a value $\lambda \in [0, 1]$ to change the values of R_v . Every time this

function is called, `StateSpaceProcessor` recalculates the matrices $A - K$ according to 3.8 with a minimal cost.

4.2.2 Extendability of Nonlinear Components and Nonlinear Solvers

The modularity of the library has been realized by implementing `NonlinearComponent` and `NonlinearSolver` as virtual classes. This gives the user the option to add new components or models by implementing the former, or new solvers by implementing the latter.

To implement a new model of one of the already implemented types of nonlinear component, i.e. a different model of a diode, NPN transistor, triode, only the constants of the existing classes need to be altered in the new class. If the user however wants to implement a new type of nonlinear component, a new class with implementations of `calculateCurrent` and `calculateJacobian` in the same form as the existing derivations of the `NonlinearComponent` needs to be constructed.

While the `NewtonRaphson` class is the only implemented nonlinear solver at the moment, the modular design of the `NonlinearSolver` makes it simple to implement and integrate another solver into the framework, to improve for example performance, convergence rate or stability. This could be used to implement for example a damped Newton-Raphson solver, as mentioned in 2.4, or even a solution for lookup tables, which could use the `NewtonRaphson` class to calculate solutions offline.

4.2.3 Interface Design

To implement a new plugin from a generated JSON-file, the following steps should be performed:

1. Create a new JUCE Audio Plugin and import the library described here.
2. In `PluginProcessor::prepareToPlay`, initialize the `StateSpaceProcessor` with the sampling rate and path to the JSON.
3. In `PluginProcessor::process`, call `StateSpaceProcessor::process` for each sample inside the main loop.
4. If using potentiometers, create a slider for each potentiometer in the `PluginEditor` and from there call the `StateSpaceProcessor::updatePotValues` with the corresponding index for each potentiometer.

The final plugin can now be exported as a VST3 or AU.

4.3 Matlab

The final part of the framework is a set of MATLAB functions designed to simplify debugging of a parsed model, as this is very cumbersome to do in C++. The MATLAB framework consists of three main public functions: `jsonToNdk`, `ndkSolve` and `ndkPlot`.

The `jsonToNdk` parses the circuit from the JSON-file into MATLAB matrices, stored in a MATLAB struct. This enables easier inspection of matrices from the parser, as well as creates a way to run a circuit from the same JSON in both MATLAB and C++.

The `ndkSolve` function takes a user-defined function for the current, a user-defined function for the jacobian and the matrices stored in the `struct`, solves the system using the Newton-Raphson method, and returns all the internal values at each time step. In other words, this gives access to values such as nonlinear currents, nonlinear voltages, state currents and output voltages. This method is mainly designed for debugging functions for the current and functions for the jacobians when implementing new components, so that custom functions and their jacobians can be entered to evaluate convergence.

The last function is the `ndkPlot` which simply is a set of convenience function calls to plot the data returned from the `ndkSolve` in a meaningful way.

As specified in section 3.5, the initial value for the nonlinear voltage vector v is set to the value at the previous time step. At time step $n = 0$, v is initialized to the zero vector, which did not cause any problems in any of the evaluated circuits. Still, there might be circuits with multiple voltage sources connected directly to various nonlinear components, which might cause convergence issues at time step 0. If that is the case, the MATLAB built-in function `fsolve` can be used to calculate the value for the first time step, as it is much more robust than a simple Newton-Raphson solver. This value can then be exported into the C++ library for such a circuit.

Chapter 5

Evaluation

In this section, the evaluation procedure will be described for the developed framework. The evaluation will focus on three main aspects: precision of the simulation (in comparison to SPICE), computational cost, and to some extent, stability. These aspects will be evaluated for 8 different circuits that were designed to cover all the components described in section 3.6, as well as different types of audio effects and varying computational load.

Each circuit was constructed in LTspice, from which a netlist was exported and through the parser loaded into both the MATLAB and C++ frameworks. For each circuit, a set of single channel, 24-bit wav-files with a sample rate of 44.1kHz were generated (content is described in each subsection) and run through LTspice using the `wavefile` command for input and `.wave` directive for output, to ensure that the correct sampling rate was used. Note however, that the `.wave` export function only allows for voltages between -1 and 1 volt. All values outside of this range are being clipped, which would result in effects, that are not caused by component behaviour. To accommodate this, the amplitude are chosen to be low. Furthermore, certain resistances in amplifying circuits are modified to achieve the desired saturation effects without having digital clipping artifacts. Moreover, the Matlab files were exported using the `audiowrite` function. The C++ framework was compiled as a VST3 using the `-O2` optimization flag and the audio files were exported through REAPER.

When evaluating performance, each file was run repeatedly through the VST3 plugin in REAPER on a Macbook Pro 2018 13" with 16 GB RAM and a 2.3 GHz Intel Core i5 (Quad Core) over a period of 2 minutes while the CPU usage of REAPER was exported to a text file using the built-in BSD command `ps` 10 times per second on the REAPER process. This procedure was also run once on each file without the plugin loaded, to calculate REAPER's idle CPU usage, which mean was subtracted from each point of measured performance. From this, the mean and maximum CPU usages were calculated, as well as the standard deviation. Note that this data is presented as percentage of usage of a single core.

5.1 RLC Lowpass Filter

The RLC lowpass filter of figure 5.1 consists of only linear components, hence no Newton iteration is required to compute the output of the model. The circuit was evaluated using 1 second of white noise for the spectrogram and a logarithmic sine sweep from 20Hz to

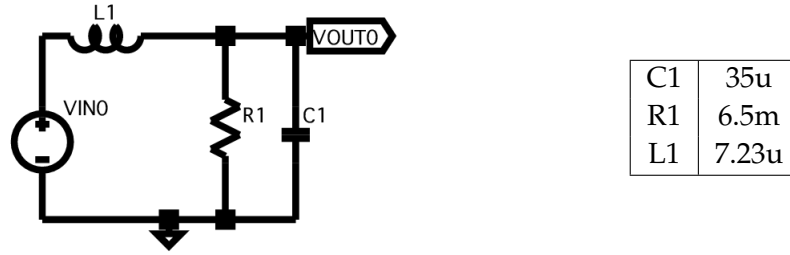


Figure 5.1: Circuit of the evaluated RLC lowpass filter

20kHz over 10 seconds for the frequency response. As shown in figure 5.2, there are no

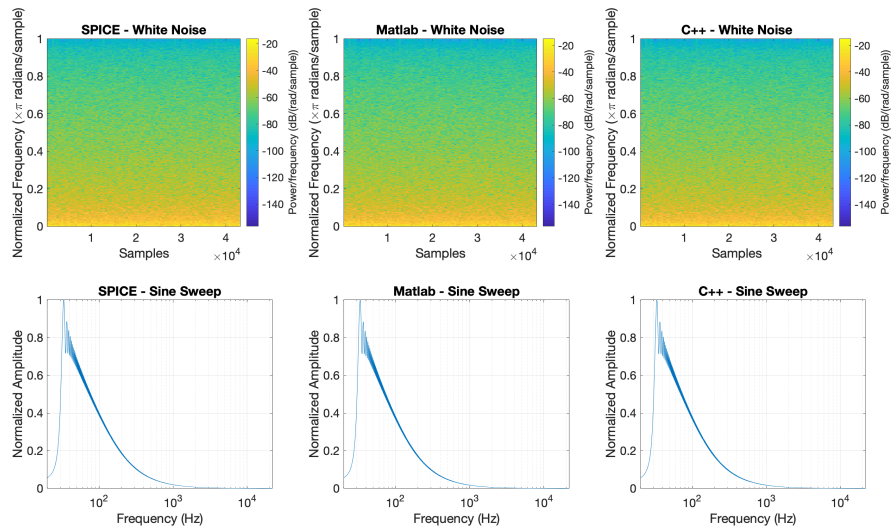


Figure 5.2: Comparison of SPICE, MATLAB and C++ for the RLC Lowpass filter

perceivable differences between the SPICE, MATLAB and C++ simulations. The model is properly filtering out the higher frequencies.

The mean CPU usage for this model was 1.14% with a standard deviation of 0.50 percentage points and a maximum peak of 3.53%. As this model is very simple and only consists of two nodes without any nonlinearities, this is indeed expected to be computationally light with little deviation from SPICE.

5.2 Diode

The diode clipper shown in figure 5.3 contains two diodes. Newton iteration is thus required at each time step to compute the output of the model, with in total 2 nonlinear equations. The circuit was evaluated using four sine waves with a frequency of 440Hz

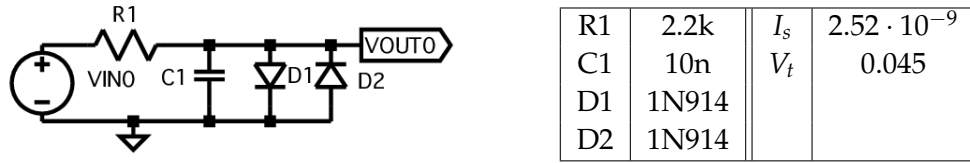


Figure 5.3: Circuit of the evaluated diode clipper

and amplitudes 0.2V, 0.5V, 0.7V, and 1V respectively. As shown in figure 5.4 there are no

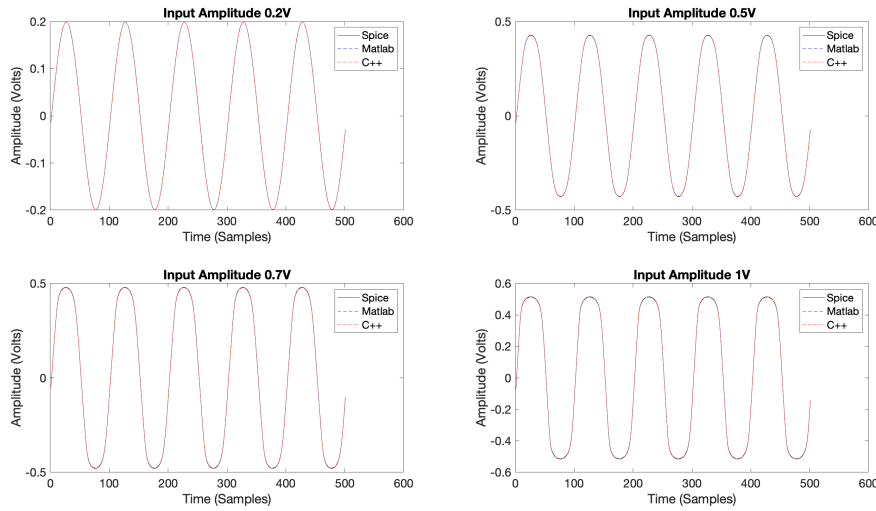


Figure 5.4: Comparison of SPICE, MATLAB and C++ for the diode clipper

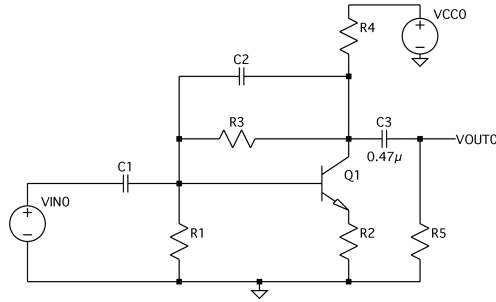
perceivable differences between the SPICE, MATLAB and C++ models. The model is increasing the amount of clipping with increased amplitude of the voltage input. The mean CPU usage for this model was 12.76% with a standard deviation of 2.47 percentage points and a maximum peak of 17.33%. While the number of nodes are the same as in the RLC lowpass filter, this model is significantly slower due to its nonlinearities.

5.3 Common Emitter Amplifier

The circuit shown in figure 5.5 contains a single 2N2222 transistor, which is comprised of two nonlinear equations for base and collector current.

The circuit is designed from a Common Emitter Amplifier, however it is actually attenuating the signal. The purpose of this circuit is saturation of the signal, to evaluate the nonlinear behaviour.

To obtain this behaviour, the output resistor R5 has been set to a relatively low value. Increasing this resistor would turn the model into an amplifier. This circuit was evaluated



R1	100k	C1	47n
R2	22	C2	0.25n
R3	470k	C3	470n
R4	10k	Q1	2N222
R5	700	VCC0	9V
I_s	$1.16 \cdot 10^{-14}$	V_t	0.025
β_F	200	β_R	3

Figure 5.5: Circuit of the evaluated common emitter amplifier

using four sine waves with a frequency of 50 Hz and amplitudes 0.2V, 0.5V, 0.7V, and 1V respectively.

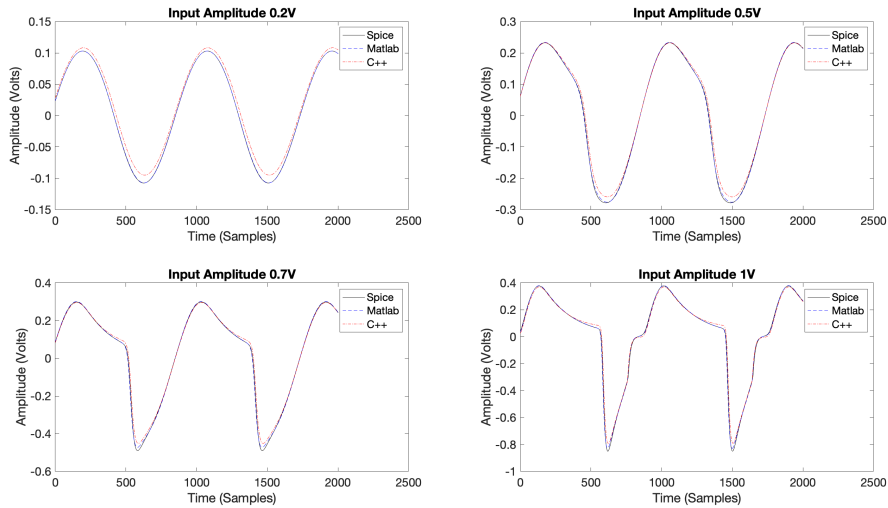


Figure 5.6: Comparison of SPICE, MATLAB and C++ for the common emitter amplifier

The accuracy for this circuit is high, both for the Matlab and C++ implementation, however the C++ implementation deviates a bit more from the SPICE model than the Matlab model does.

The reason for this deviation between the Matlab and the C++ model is most likely because of floating point precision, as the equations set up for the Matlab and C++ framework are exactly the same, with the only difference being value types.

The mean CPU usage for this model was 13.25%, with a standard deviation of 2.36% percentage points and a maximum peak of 17.63%.

This is relatively fast, compared to the diode clipper. The model performs almost as well, even though the nonlinear equation should be more computationally demanding for the transistor currents, as there are more transcendental functions.

One issue with this circuit, which is not visible in our evaluation data, is that the model is unstable. The Newton-Raphson does not converge for higher frequencies, while the amplitude is also high. This is the reason why the evaluation data for this circuit is carried out using a 50Hz sine, to enable us to show that the nonlinear behaviour of the transistor model is correct, despite the stability issue.

5.4 Common Cathode Amplifier

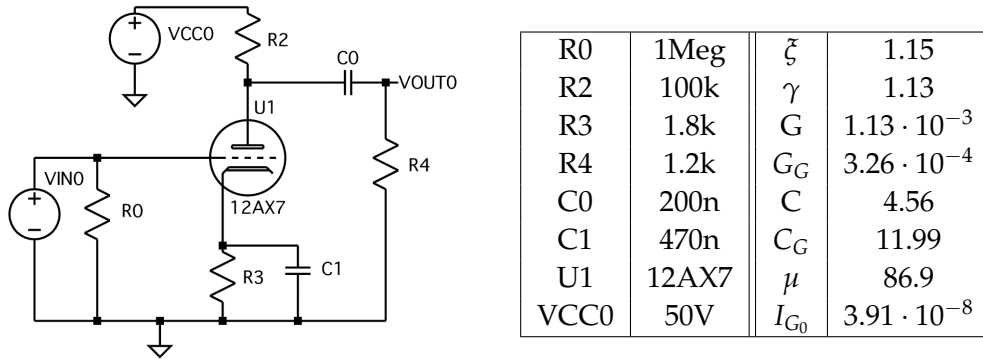


Figure 5.7: Circuit of the evaluated common cathode amplifier

This circuit includes a 12AX7 triode, as can be seen in figure 5.7. The currents for this nonlinear component is solved through two equations. Like the Common Emitter Amplifier in section 5.3, this circuit is also actually attenuating the signal, and again saturating the signal, to enable evaluation of the nonlinear behaviour.

This circuit needs a low output, which leads to a low resistance for the output resistor R4. For this circuit to clip, the VCC0 also needs to be low, which causes saturation to appear with smaller input voltages.

The circuit was evaluated using four sine waves with a frequency of 440Hz and amplitudes 0.2V, 0.5V, 0.7V, and 1V respectively. As shown in figure 5.8, this model deviates quite a bit from the SPICE model, however tubes are quite imprecise. Even though the model is the same, the values might vary quite drastically. This can be seen in [3], where Dempwolf et.al. provide the values they have measured for three different 12ax7 tubes. The nonlinear behaviour we see is however similar to the SPICE model, as well as similar to the results presented in [3].

The mean CPU usage for this model was 15.74%, with a standard deviation of 3.51% percentage points and a maximum peak of 22.13%.

This circuit is slightly slower than the emitter circuit, but this is to be expected as the equations for the currents should be more computationally heavy, as there are many transcendental functions.

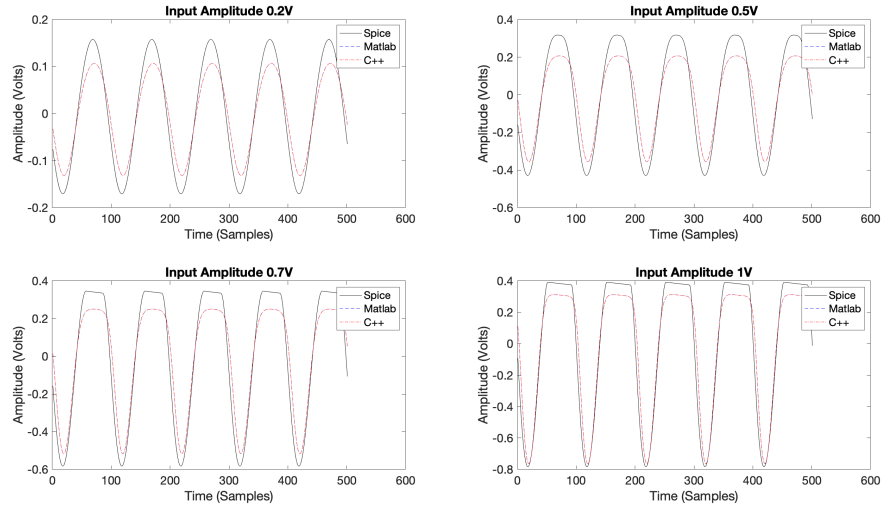


Figure 5.8: Comparison of SPICE, MATLAB and C++ for the common cathode amplifier

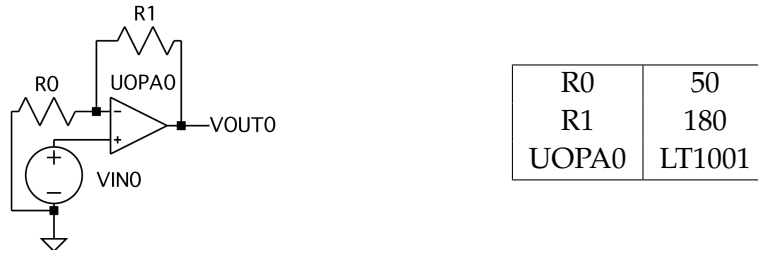


Figure 5.9: Circuit of the evaluated Opamp circuit

5.5 Operational Amplifier

As can be seen in figure 5.9, this circuit does not contain any nonlinear components. It is designed with component values, so that the operational amplifier should operate in its linear range. The circuit simply amplifies the input signal.

As this circuit produces a high and offset output, this evaluation does not include a SPICE model, because the wave export function of SPICE only allows signals between 1V and -1V to be exported [12]. This is not an audio circuit, the intention is only to confirm the linear amplification of the circuit.

This circuit was evaluated using one sine wave with a frequency of 440Hz, an amplitude of 0.2V and an offset of 0.4V. The results presented in figure 5.10 are as could be expected, and the same results would be found for the SPICE model, with the same values for both components and input, as this is in the linear range of the operation amplifier modelled in SPICE.

The mean CPU usage for this model was 0.88%, with a standard deviation of 0.50% percentage points and a maximum peak of 2.23%.

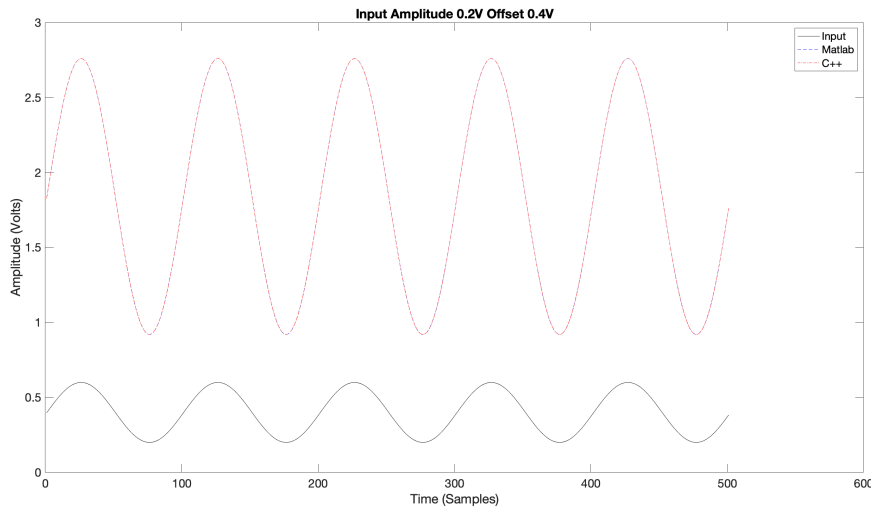


Figure 5.10: Comparison of input signal and output signals of Matlab and C++ implementation of the Operational Amplifier

As there are neither nonlinear components or reactive components, it is expected to be computationally light.

5.6 Wah-Wah Effect Pedal

As seen in figure 5.11, this circuit contains two NPN transistors of the model 2N2222, which is also used in the Common Emitter Amplifier in section 5.5.

The behaviour of this circuit has been evaluated as a static bandpass filter, as the potentiometer is locked in one position for this evaluation.

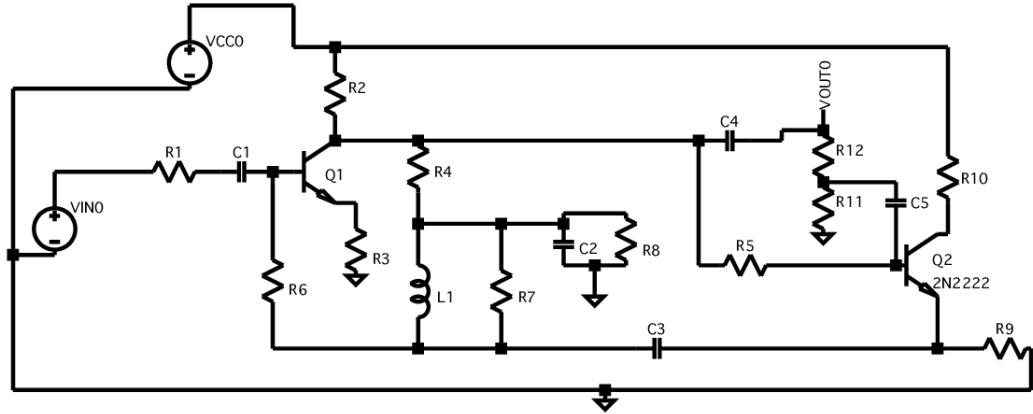
The circuit was evaluated using 1 second of white noise for the spectrogram and a logarithmic sine sweep from 20Hz to 20kHz over 10 seconds for the frequency response. The Matlab implementation results are virtually identically to the SPICE model, as seen in figure 5.12. The C++ model does however deviate slightly. There is a peak in the lower frequencies, and the passband is slightly wider than the SPICE and Matlab model.

As with the emitter amplifier, it is most likely caused by a floating point precision issue, specifically evident for circuits using the NPN transistor model.

The mean CPU usage for this model was 21.03%, with a standard deviation of 4.74% percentage points and a maximum peak of 31.63%.

This was expected this to be the slowest model, as it has four nonlinear equations, compared to a maximum of two for the rest of the circuits.

The potentiometer of this circuit has a stability issue. If R12, which is the top resistor of the virtual potentiometer, goes below 96K the Newton-Raphson solver does not converge. Moving the potentiometer in the working range does however produce a small Wah effect.



R1	68k	R12	96k	β_F	200
R2	22k	C1	10n	β_R	3
R3	390	C2	4.7u	V_t	0.025
R4	470k	C3	10n		
R5	470k	C4	220n		
R6	1.5k	C5	220n		
R7	33k	L1	500m		
R8	82k	Q1	2N2222		
R9	10k	Q2	2N2222		
R10	1k	VCC0	9V		
R11	4k	I_s	$1.16 \cdot 10^{-14}$		

Figure 5.11: Circuit of the evaluated Wah-Wah Effect Pedal

5.7 Stress Testing

As all of the evaluated circuits computation-wise performed within a single-core handleable range, two more circuits were constructed to evaluate boundaries for C++ framework computation-wise. The purpose of the first circuit was to evaluate the maximum number of linear components supported, and was based on the RLC lowpass filter in figure 5.1. Here, the node VOUT0 instead was chained into a new identical lowpass filter, which in turn was chained into another lowpass filter, etc., up to in total 64 lowpass filters (192 linear components) in the same circuit. The mean CPU usage for this model was 61.12% with a standard deviation of 0.91 percentage points and a maximum peak of 65.83%. The value 64 was arrived at by experimentally doubling the number of lowpass filters in the circuit on each iteration. While 64 filters did not clip with its 65.83% peak, 128 filters did. Hence, more than 192 linear components are technically possible, but the limit

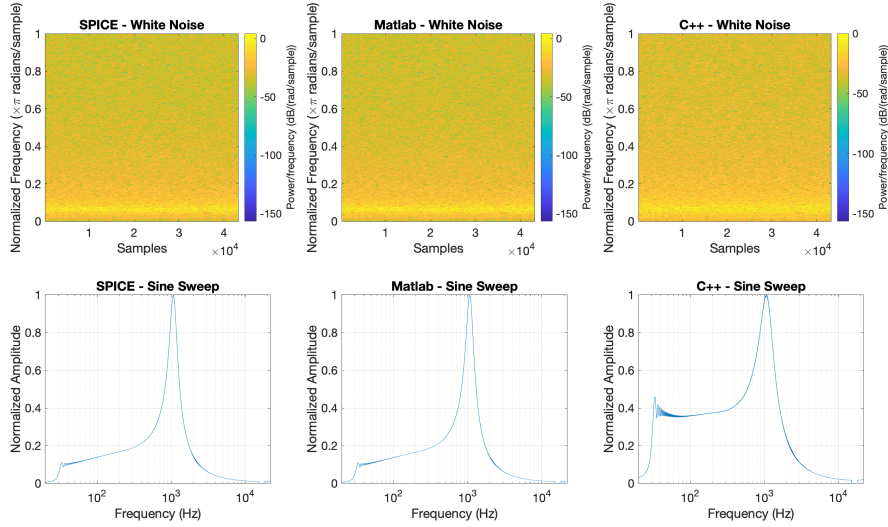


Figure 5.12: Comparison of SPICE, MATLAB and C++ for the Wah-Wah

is somewhere in-between these two values.

A similar test was performed for nonlinear components by connecting multiple diodes in series in the same diode clipper, hence requiring an inversion of a $k \times k$ jacobian in each iteration for k diodes. For a model consisting of 13 diodes, the mean CPU usage was 63.50% with a standard deviation of 10.42 percentage points and a maximum peak of 84.23%. Note that these values were calculated according to the procedure described in section 5, so the mean CPU usage of REAPER when run without plugins was subtracted from the results, meaning that with a peak of 84.23% CPU usage, the core on which REAPER was running was at some point overloaded, resulting in clipping (in total twice during a 2 minutes run).

As the performance depends both on the size of the linear part of the system, the size and complexity of the jacobian, the convergence rate of the individual nonlinear functions, as well as the amplitude and variation in the voltages over the nonlinear components, this result is insufficient to declare a definite boundary for the framework. Still, it gives a pointer towards how high of a computational load that can be estimated to be handled by the framework.

Chapter 6

Discussion

In this chapter, the results of the evaluation, as well as the further steps that could be taken in regard to the development of the framework implemented will be discussed.

6.1 Accuracy

As shown in chapter 5, most of the models implemented for the evaluation are fairly accurate to the same circuits modelled in LTspice. It is to be expected that the LTspice simulations have slightly different results, as the modelling method used in SPICE differs from the NDK-method.

The deviation between the MATLAB and C++ implementations was not expected. As was mentioned in the evaluation, this might be because of floating point precision in C++, resulting in round off errors. For further development of the framework, a formal investigation of this behaviour could be carried out to determine if this is what causes the deviation.

Another way to evaluate circuits is to carry out measurements of physical circuits and compare models implemented with this framework to their physical counterparts. With this method, artifacts of physical circuits not simulated in this framework might appear as well. Finally, to evaluate the sound quality of modelled circuits, user testing could be carried out, to investigate if the differences between our modelled circuits and the physical circuits are conceptually distinguishable. While neither of the two latter were done in this project, it would be beneficial to do this once the framework is more mature.

6.2 Stability

Generally, investigating stability issues has not been the main focus of this project. In the evaluation, however, some circuits for the evaluation did prove to be unstable. Specifically, as was presented in chapter 5, the Common Emitter Amplifier had issues with higher frequencies as shown in section 5.3, and the Wah-Wah model had stability issues when the value of the potentiometer was under a certain threshold, as mentioned in section 5.6.

Some possibilities for solving stability issues would be to extent the framework with other nonlinear solvers that are not as prone to convergence issues. Examples of this are the damped Newton-Raphson method or using homotopy, as was presented in section 2.4. As mentioned, the focus has not been on implementing different nonlinear solvers that could stability, but because the framework has been implemented with modularity and extendability in mind, it would be simple to include this into the framework.

Another method that could improve the stability, especially when the issue is with higher frequency content, is upsampling. While this may help with convergence, it will also drastically increase the computational cost and make any solution not based on lookup table inviable.

6.3 Computational Cost

Computational cost was not a problem in any of the models evaluated in chapter 5 apart from the ones specifically designed for maximizing the CPU usage. From a prototyping perspective, the framework is in its current state sufficient for implementing small to medium sized circuits, but limited for real-time evaluation of larger circuits. For production, a CPU usage of above 12% for a single diode clipper is also quite high.

One explanation for this performance is the use of the Eigen library for linear algebra. The results in [4] suggests that an alternative library for matrix inversion, such as LAPACK [11] could offer better performance. This would require primitive arrays with sizes known at compile time, which, in the current implementation is not supported, but could easily be implemented by using tokens in the C++ classes that can be regex-replaced by the parser when parsing the circuit. This would of course make the development process more convoluted, as the `StateSpaceProcessor` and all its subclasses would need to be rewritten each time a circuit is modified.

Another solution would be to implement lookup tables. As there are various approaches to decreasing the size of the lookup tables [13], [9], this could possible be used to increase the performance for larger circuits. While this is out of the scope for the current project, this could easily be implemented in the C++ by writing a class realizing the `NonlinearSolver` interface, which could use the `NewtonRaphson` class internally for the offline solving.

6.4 Usability

In terms of the usability of the presented framework, a multitude of different measures can be taken to increase its ease of use.

A general consideration is setting up and maintaining a documentation for the MATLAB and C++ libraries. As this framework is designed around a streamlined workflow, a proper readme file could convey the intended way of using it. On top of that, a tutorial with the implementation of a simple nonlinear circuit could make the framework more accessible to developers and serve as an introduction to the paradigms of state-space modeling with emphasis on Virtual Analog emulation.

As a furthering annotation, the MATLAB environment in its current state does not provide a granular set of classes as the C++ implementation does. Thus, one has to derive

the jacobian as well as set necessary currents for nonlinear components manually. With an increased amount of nonlinear components, this process is both cumbersome and prone to produce careless mistakes. Having a class system in MATLAB that follows the object-oriented paradigm that has been applied in C++ would make the testing environment in MATLAB easier to use.

The parser script, which derives the NDK matrices from the SPICE schematics, does not check the validity of the circuit in its current state. However, a set of rules has to be followed to in order for the parsing to work correctly. This includes having at least one ground node, the correct naming of voltage sources and other basic checks. Having a set of error messages and warnings, that the parser would throw whenever the syntax of the netlist does not hold up to certain sanity checks, could provide helpful insights for the developer.

Considering the number of nonlinear components that are provided in C++ with the framework, one is able to implement basic circuits that produce desired audio effects, such as filtering and saturation stages. However, bearing the intended modularity of the framework in mind, it is desirable that developers implement their custom models of e.g. triodes and transistors. Having a proper step-by-step guide to this would contribute greatly to the framework.

While usability testing of the project could lead to interesting findings, there is still a range of features that would need be to implemented before the proper intended usage of the framework can be carried out, as has been discussed in this chapter.

Chapter 7

Conclusion

As the framework described in this report has implemented the NDK-method as an automated derivation of the nonlinear equations of a state-space system, the modularity issue does no longer exist, when using our framework to create physical models of electronic circuits. A set of auxiliary tools for MATLAB have been developed to simplify the debugging and fine-tuning of circuits. With this, designing circuits and fine-tuning them can be done offline in MATLAB before it is implemented in real-time using the same set of data.

Moreover, the C++ library allows real-time prototyping of the modelled electronic circuits up to a certain size. Larger circuits do yet necessitate optimization of the framework, with its most critical processing block being the solver. The circuits implemented for the evaluation have demonstrated the accuracy of the framework and the nonlinear behaviour of the modelled components. However, in certain scenarios, it has also highlighted that measures to ensure stability of the physical models need to be investigated. Since the C++ library has been implemented with the purpose of being extendable, a range of methods that can optimize the computational cost as well as improve upon the stability of modelled circuits can easily be incorporated into the framework.

While the effectiveness of the framework has been clearly shown in the evaluation of the test circuits, further steps needs to be taken to extend the usability of the framework. As a general consideration, user tests with developers and engineers would prove useful in this endeavour.

Bibliography

- [1] Gianpaolo Borin, Giovanni De Poli, and Davide Rocchesso. "Elimination of delay-free loops in discrete-time models of nonlinear acoustic systems". In: *IEEE Transactions on Speech and Audio Processing* 8.5 (2000), pp. 597–605.
- [2] *bwrc.eecs.berkeley.edu*. http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/UserGuide/elements_fr.html. Accessed: 24-05-2019.
- [3] Kristjan Dempwolf and Udo Zölzer. "A physically-motivated triode model for circuit simulations". In: *14th international conference on Digital Audio Effects DAFx*. Vol. 11. 2011.
- [4] Felix Eichas et al. "Physical Modeling of the MXR Phase 90 Guitar Effect Pedal." In: *DAFx*. 2014, pp. 153–158.
- [5] *Eigen.tuxfamily.org*. http://eigen.tuxfamily.org/index.php?title=Main_Page. Accessed: 21-05-2019.
- [6] A. Fettweis. "Wave digital filters: Theory and practice". In: *Proceedings of the IEEE* 74.2 (1986), pp. 270–327. ISSN: 0018-9219. DOI: 10.1109/PROC.1986.13458.
- [7] Ben Holmes and Maarten van Walstijn. "Improving the robustness of the iterative solver in state-space modelling of guitar distortion circuitry". In: *Proc. 18th Int. Conference on Digital Audio Effects (DAFx-15), Trondheim, Norway*. 2015.
- [8] Martin Holters and Udo Zölzer. "A generalized method for the derivation of non-linear state-space models from circuit schematics". In: *2015 23rd European Signal Processing Conference (EUSIPCO)*. IEEE. 2015, pp. 1073–1077.
- [9] Martin Holters and Udo Zölzer. "A kd tree based solution cache for the non-linear equation of circuit simulations". In: *2016 24th European Signal Processing Conference (EUSIPCO)*. IEEE. 2016, pp. 1028–1032.
- [10] Martin Holters and Udo Zölzer. "Physical Modelling of a Wah-wah Effect Pedal as a Case Study for Application of the Nodal DK Method to Circuits with Variable Parts". In: Sept. 2011.
- [11] *LAPACK*. <http://www.netlib.org/lapack>. Accessed: 25-05-2019.

- [12] *LTspice Manual*. http://ltwiki.org/LTspiceHelp/LTspiceHelp/_WAVE_Write_selected_nodes_to_a_wav_file_.htm. Accessed: 27-05-2019.
- [13] Jaromír Mačák. “Real-time digital simulation of guitar amplifiers as audio effects”. PhD thesis. Ph. D. thesis, Brno University of Technology, Brno, 2012.
- [14] *NodalDKFramework*. <https://github.com/jardamacak/NodalDKFramework>. Accessed: 22-05-2019.
- [15] Jonathan S. Abel et al. *DAFX - Digital Audio Effects*. Apr. 2011. ISBN: 978-0470665992.
- [16] Matthew NO Sadiku and Charles K Alexander. *Fundamentals of electric circuits*. McGraw-Hill Higher Education, 2007.
- [17] Tim Schwerdtfeger and Anton Kummert. “A Multidimensional Approach to Wave Digital Filters with Multiple Nonlinearities”. In: *Proceedings - 22nd European Signal Processing Conference* (Sept. 2014).
- [18] Adel S Sedra and Kenneth Carless Smith. *Microelectronic circuits*. New York: Oxford University Press, 1998.
- [19] Andrei Vladimirescu. *The Spice Book*. New York: Wiley, 1994.
- [20] Vesa Välimäki and Tapio Takala. “Virtual musical instruments - Natural sound using physical models”. In: *Organised Sound* 1 (Aug. 1996), pp. 75 – 86. DOI: 10.1017/S1355771896000039.
- [21] Kurt Werner et al. “Resolving Wave Digital Filters with Multiple/Multiport Nonlinearities”. In: *Proceedings - 18th International Conference on Digital Audio Effects, DAFx 2015* (Nov. 2015).
- [22] David Yeh. “Digital Implementation of Musical Distortion Circuits by Analysis and Simulation”. PhD thesis. June 2009.
- [23] David Yeh, Jonathan S Abel, and Julius Smith. “Automated Physical Modeling of Nonlinear Audio Circuits For Real-Time Audio Effects—Part I: Theoretical Development”. In: *Audio, Speech, and Language Processing, IEEE Transactions on* 18 (June 2010), pp. 728 –737. DOI: 10.1109/TASL.2009.2033978.
- [24] David Yeh and Julius Smith. “Simulating guitar distortion circuits using wave digital and nonlinear state-space formulations”. In: *Proceedings - 11th International Conference on Digital Audio Effects, DAFx 2008* (Sept. 2008).
- [25] David T Yeh. “Automated physical modeling of nonlinear audio circuits for real-time audio effects—Part II: BJT and vacuum tube examples”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 20.4 (2011), pp. 1207–1216.

Appendix A

A.1 C++ Class Diagram

