Clemson University

TigerPrints

All Theses Theses

12-2022

Surrogate Modeling of Nonlinear Components and Circuits

Byron William Byars
Clemson University

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Recommended Citation

Byars, Byron William, "Surrogate Modeling of Nonlinear Components and Circuits" (2022). *All Theses*. 3909.
https://tigerprints.clemson.edu/all_theses/3909

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

Surrogate Modeling of Nonlinear Components and Circuits

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science
Electrical Engineering

---

by
Byron William Byars
December 2022

Accepted by:
Dr. Melissa Smith, Committee Chair
Dr. Richard Groff
Dr. Hassan Raza

Abstract

Surrogate models are simplified approximations of functions that are described by complex equations. Surrogate modeling of physical systems have been used in various fields such as biology, fluid dynamics, climate modeling, and various other engineering disciplines. This data-driven approach is used to decrease computational cost, decrease computation time, or when the output of a system is difficult or impossible to measure, by using a "black-box" method to approximate the output given inputs. In regards to circuit analysis, surrogate models can be used to decrease computation time and computational load.

In this thesis, surrogate modeling is used to model various nonlinear components and circuits in fREEDA, a multi-physics circuit simulator, for the purpose of speeding up transient analysis. Neural networks are used in place of physics-based equations, resulting in a speedup of $5 - 18x$ for the evaluation of the components and $3x$ for the evaluation of entire circuits. The components and circuits tested in this work include: BJT (Bipolar Junction Transistor), MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor), common-emitter amplifier, and common-source amplifier.

Dedication

This work is dedicated to my family and friends who have always been there to support and inspire me.

Acknowledgments

First and foremost, I would like to thank Dr. Smith for her guidance throughout my graduate work. This would not have been possible without your help and support. I would also like to thank Dr. Groff and Dr. Raza for serving on my committee. Watching how you two approached problems in senior design helped me grow as an engineer. Next, I would like to thank the students in the FCTL. I learned so much from everyone in this lab and could not have asked for a better group of people to have been around during graduate school. And lastly, I'd like to thank all the faculty, staff, and students who I have worked with during my time at Clemson University.

Table of Contents

Title Page i

Abstract ii

Dedication iii

Acknowledgments iv

List of Tables vii

List of Figures viii

List of Listings ix

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Contribution 1
- 1.3 Outline 2

2 Background **3**
- 2.1 Circuit Simulation Software 3
- 2.2 Neural Networks 4
- 2.3 Component and Circuits Models 7

3 Related Work **8**
- 3.1 Interpolation 8
- 3.2 Machine Learning 9

4 Research Design and Methods **10**
- 4.1 Data Collection 10
- 4.2 Transistor Models 12
- 4.3 Neural Network Training and Architecture 14
- 4.4 Surrogate Model Implementation 15

5 Results **19**
- 5.1 Surrogate Model Accuracy 19
- 5.2 Surrogate Model Speedup 25

6 Conclusions and Discussion **28**
- 6.1 Conclusion 28
- 6.2 Future Work 28

Bibliography **30**

Appendices **32**
A Code Listings 33

List of Tables

4.1 Component Neural Network Architecture 15
4.2 Circuit Neural Network Architecture 15

5.1 RMSE of BJT Output Variables 20
5.2 RMSE of MOSFET Output Variables 22
5.3 RMSE of Common-Emitter Amplifier Outputs 23
5.4 RMSE of Common-Source Amplifier Outputs 23
5.5 Gummel-Poon NPN BJT Timing 26
5.6 Philips MOS9 N-Channel MOSFET Timing 26
5.7 Common-Emitter Amplifier Timing 27
5.8 Common-Source Amplifier Timing 27

List of Figures

1.1 Overview of Contribution . . . 2

2.1 Network Graph [14] . . . 4
2.2 Fully Connected Neural Network . . . 5
2.3 ReLU Activation Function . . . 6
2.4 Overview of Auto-Keras [7] . . . 7

4.1 Overview of Workflow . . . 11
4.2 Common-Emitter Amplifier . . . 13
4.3 Common-Source Amplifier . . . 14
4.4 Implementation Process . . . 16
4.5 Circuit Implementation . . . 18

5.1 BJT Output Variables . . . 21
5.2 MOSFET Output Variables . . . 21
5.3 Common-Emitter Amplifier Input and Output Voltage . . . 24
5.4 Common-Source Amplifier Input and Output Voltage . . . 24

Listings

4.1 Dense Layer and ReLU in Surrogate Model Implementation 17
1 Diode Element in fREEDA 33
2 fREEDA Netlist for Half-Wave Rectifier 36

Chapter 1

Introduction

1.1 Motivation

Surrogate models are an approximate mapping between input and output of complex systems. Circuit analysis lends itself well to surrogate modeling as it is not so complex that the approximations are not highly accurate, but there is an opportunity to speed up the process of transient analysis. By creating a general, easy-to-use workflow, it would not be difficult for one to replace a suite of commonly used transistors, for example, with surrogate models of those transistors that could be used instead. An additional benefit to using surrogate models, in the case of modeling a circuit specifically, is that it protects a circuit designer's intellectual property while allowing others to perform testing by presenting a "black-box" circuit [13].

1.2 Contribution

This work presents a modular workflow for replacing nonlinear components and circuits in fREEDA with high fidelity surrogate models to decrease the time to complete transient analysis. All that is needed to model a component or circuit is training data. The data is fed into an AutoML Application Programming Interface (API) to find an optimal neural network to replace the physics-based equations that are traditionally used (see Figure 1.1). The resulting speedup for evaluating a component is $5 - 18x$ and for evaluating an entire circuit is $3x$.

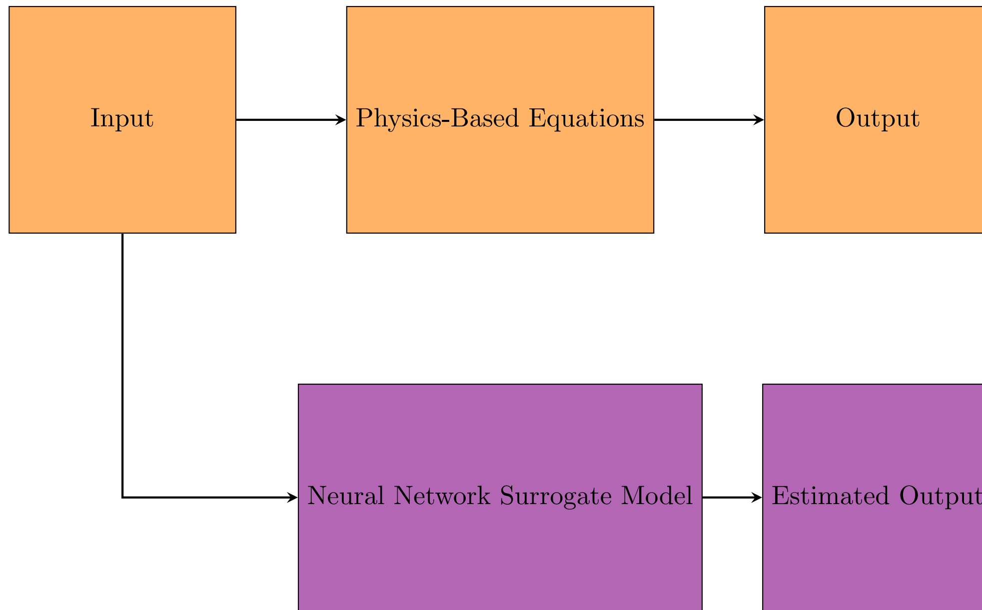

Figure 1.1: Overview of Contribution

1.3 Outline

The rest of this manuscript is organized as follows: Chapter 2 covers background information regarding fREEDA, neural networks, surrogate modeling, and the transistors modeled, Chapter 3 overviews related work on surrogate modeling of circuits, Chapter 4 describes the surrogate modeling workflow designed, Chapter 5 details the results from experiments completed, and Chapter 6 includes the conclusion and possible future work.

Chapter 2

Background

This chapter provides an overview of the components used for modeling, the circuit simulation software used, as well as the relevant machine learning concepts.

2.1 Circuit Simulation Software

fREEDA is an open-source multi-physics circuit simulator that was first presented as an alternative to SPICE or other SPICE-like circuit simulators. It is implemented almost entirely in C++ with an object oriented design approach [14]. Each element has a class that is based on the common base class element. See Listing 1 for an example of the "diode" element. It is here where the reference name of the element, number of terminals, parameters of the specific element (i.e. saturation current, breakdown voltage, etc. for the "diode" element) as well as the init and eval routines resides. The physics-based equations relating the state variables to voltages and current are located within the eval routine [14]. The input to the eval routine are of class AD (automatic differentiation) [9]. Automatic differentiation provides an analytical derivative by repeatedly applying the chain rule to a composition of simple operations to calculate the Jacobian. This AD class, specific to elements, is initialized in the init routine, where the number of derivatives to be calculated is declared. For example, the "diode" element has one state variable and the time derivative of the state variable for a total of two inputs to the physics-based equations whereas the "bjtnpn" element has three state variables with the first and second time derivatives of those state variables for a total of nine inputs to the physics-based equations. Note that these derivatives are

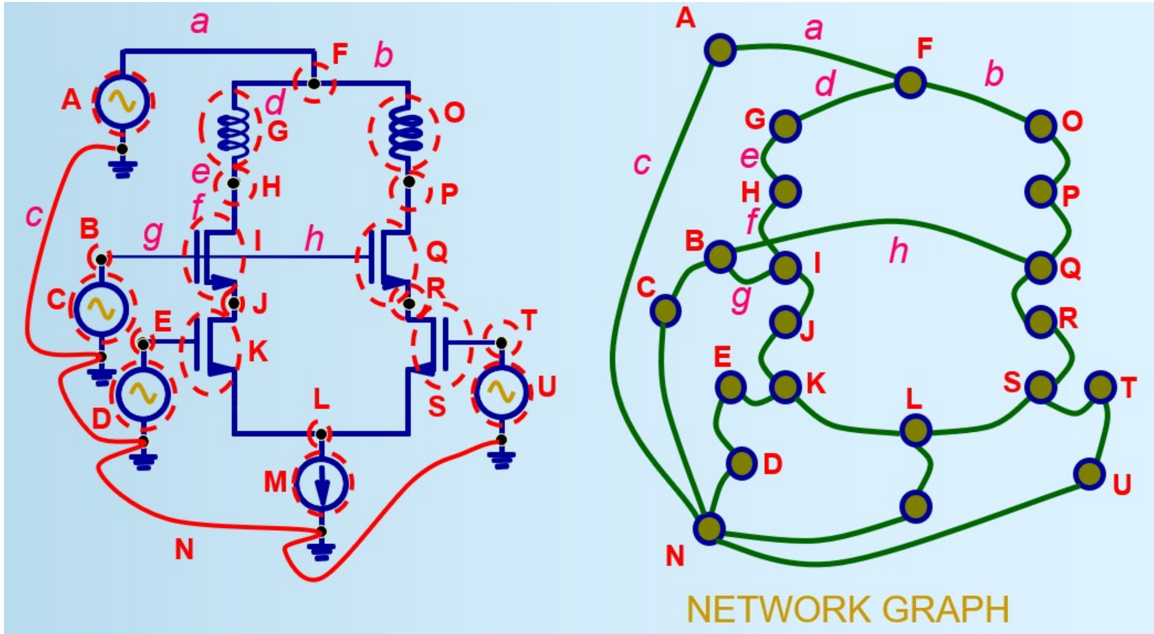

Figure 2.1: Network Graph [14]

calculated automatically each time step outside of the eval routine [14].

Circuits in fREEDA are stored as a network graph in a netlist. The network graph treats both elements and terminals as nodes (see Figure 2.1) and labels them as such in the netlist [14]. See Listing 2 for an example of a netlist for a half-wave rectifier. fREEDA supports the following circuit analysis types: DC, AC, harmonic balance, transient (both fixed and variable time-stepping), and wavelet analysis.

LTspice [2] was used for data collection and verifying the fidelity of the surrogate models. LTspice is a SPICE circuit simulator from Analog Devices that provides a wide variety of different components and models without any additional modifications, some of which were used in this work including NPN BJTs and N-Channel MOSFETs. Additional useful features of LTspice include schematic capture and a graphing tool, both of which are not currently available in fREEDA.

2.2 Neural Networks

The majority of machine learning problems can be categorized as one of the following: supervised learning, unsupervised learning, or reinforcement learning. The problem in this work is

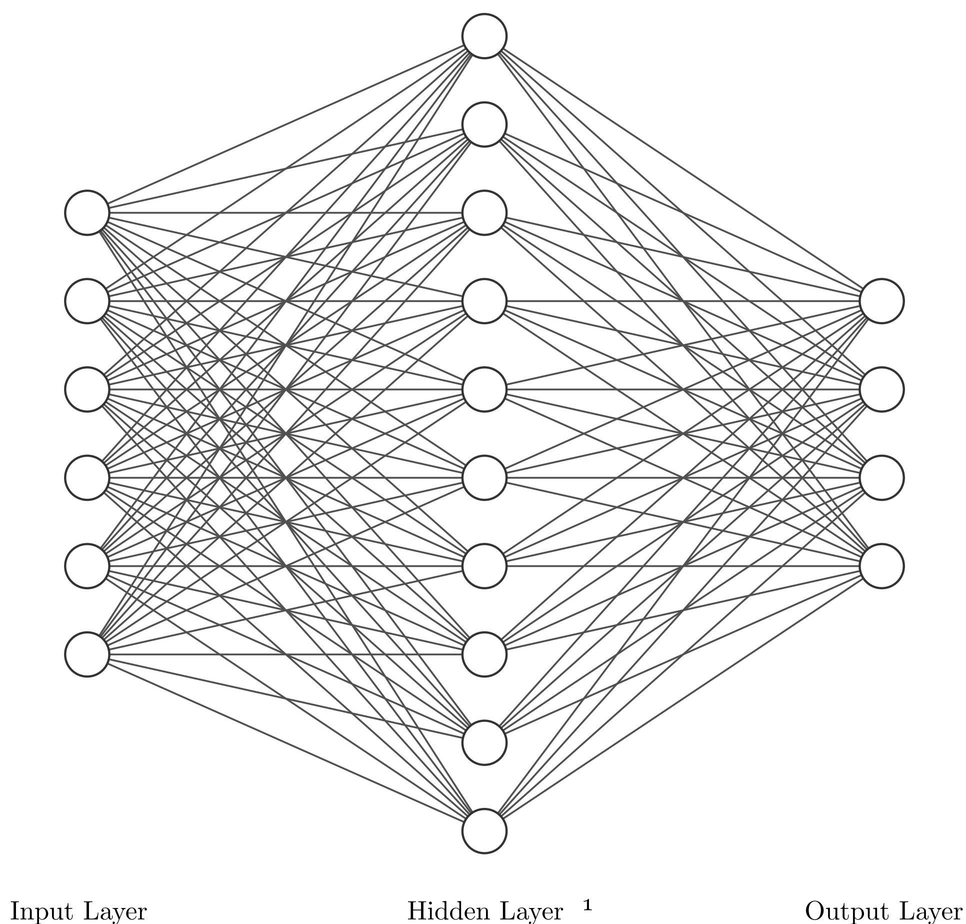

Figure 2.2: Fully Connected Neural Network

a supervised learning problem. The goal of supervised learning is to approximate some function

$$F : X \rightarrow Y$$

where $X \in \mathbb{R}^m$ is the input and $Y \in \mathbb{R}^n$, the output, is some known label given k samples. In this work, it was found that relatively small ($\sim$ 1000 parameters) fully connected neural networks were able to model the nonlinear components and circuits tested with high accuracy. Figure 2.2 is an example of a fully connected neural network, as each neuron in the previous layer is connected to every neuron in the current layer. Using Figure 2.2 as an example (i.e. Input Layer $\in \mathbb{R}^6$, Hidden Layer $\in \mathbb{R}^{10}$, Output Layer $\in \mathbb{R}^4$), the values of any neuron in the hidden layer can be calculated as follows:

$$Neuron_j = (\sum_{i=1}^{6} w_{i,j} * x_i) + b_j \text{ for } j \in [1, 10]$$

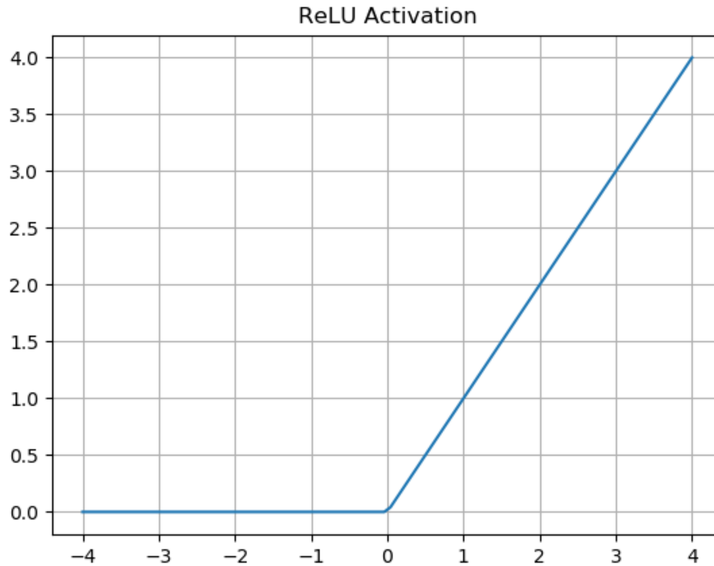

Figure 2.3: ReLU Activation Function

where w and b are the corresponding weight and bias values. This same logic is used to find values of neurons in other hidden layers (if they exist) as well as the output values. Following this calculation at each layer, an activation function is applied to turn the previously linear mapping to a nonlinear mapping. Specifically, this work only involves the activation function ReLU (Rectified Linear Unit), which is defined as $f(x) = max(0, x)$ as shown in Figure 2.3.

To find a sufficiently good mapping, F, the neural network must be tuned, including parameters such as the size of hidden layers, number of hidden layers, hyperparameter values, etc. To provide a general, easy-to-use workflow, AutoML is used to perform this fine-tuning. AutoML is a rather broad term and can refer to solely hyperparameter tuning or an optimal neural architecture search. For this work, Auto-Keras was selected as it performs both. Auto-Keras uses a Bayesian Optimization algorithm to search a neural architecture search space to minimize a user-defined cost function (or loss function; Auto-Keras uses mean squared error by default) given training data [7]. The Auto-Keras workflow, see Figure 2.4, begins with the Bayesian Optimization search for a neural architecture. Following this, the Graph module builds a neural network from the selected neural architecture. Next, the neural network is trained on the training data provided by the user and lastly the model is saved with results of its performance sent back to the Searcher as feedback. The

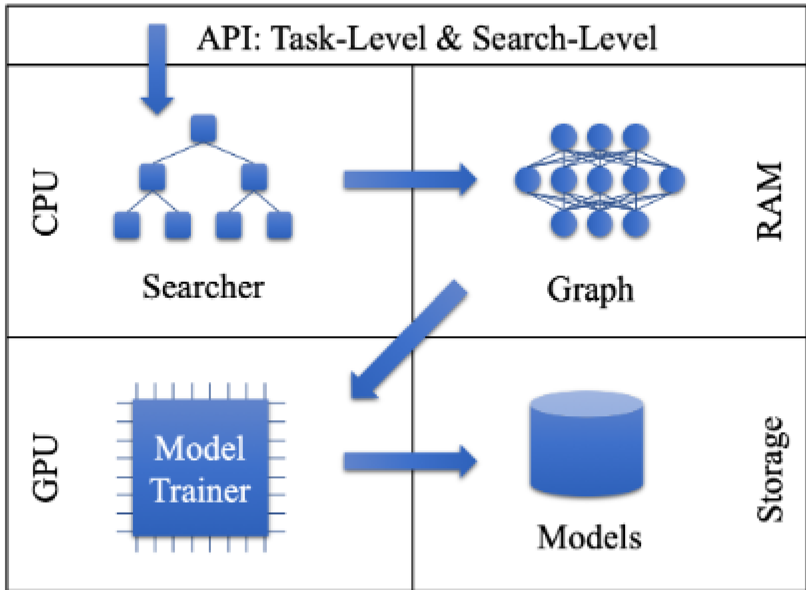

Figure 2.4: Overview of Auto-Keras [7]

model that best minimizes the cost function is then saved for future use.

2.3 Component and Circuits Models

For both BJTs and MOSFETs, there are multiple different models used in circuit simulators that vary in complexity, including the number of parameters that are used to describe the transistor as well as the physics-based equations that are solved at each time step. In general, as complexity increases, so does the accuracy of the model compared to the actual hardware. For this work, the Gummel-Poon model [6] was selected for the BJT and the Philips MOS9 Model [12] for the MOSFET. Both of these are on the lower end in terms of complexity, but are readily available in most popular circuit simulation software. The specific transistors modeled were the 2N2222 NPN BJT and the BSS123 N-Channel MOSFET. The circuits modeled were the common-emitter amplifier and common-source amplifier. Both of these are single-stage amplifiers that are primarily used for voltage amplification. The same BJT and MOSFET that were used for the component-level surrogate models were used in the common-emitter and common-source amplifiers, respectively.

Chapter 3

Related Work

Surrogate models in circuit simulation are used to reduce the time to complete analysis and computational load compared to physics-based equations, as well as protecting the intellectual property of a circuit designer by presenting a high fidelity "black-box" solution rather than a schematic itself. This chapter gives an overview of a few differently types of surrogate models that have been used to model nonlinear circuits, namely interpolation and various machine learning methods.

3.1 Interpolation

Interpolation is the process of making an estimate using information of prior samples. Though a few different interpolation methods have been used for surrogate modeling, the most popular is kriging [16], [8], [5]. Kriging uses known input/output pairs to estimate the output given a new input by combining information of a regression model and the correlation between two sample vectors. It can be thought of as the regression model being a global trend and the correlation being the localized trend [8]. A pitfall of kriging is it requires one to have a relative strong understanding of the data attempting to be interpolated as one must decide on the appropriate regression model and correlation function. As the goal of this work is to make a simple, straightforward workflow to create surrogate models, kriging was excluded.

3.2 Machine Learning

Different machine learning methods such as Support-Vector Machines (SVM) [16], [4] and neural networks [13], [15], [3], [10] have been used to create surrogate models of circuits. The authors in [4] use SVM as a surrogate model for industrial circuits including the Digital-to-Analog Converter and DC-DC Converter. They performed sensitivity analysis to determine which of the input parameters were most important and used those, decreasing the total number of inputs used by over half. They also experimented with the amount of training samples needed to train the surrogate model, ranging from 50 to $1,000$, and showed that even with a relatively small amount of samples, a high fidelity model of complex circuits was achievable.

In [3], the author use a Recurrent Neural Network (RNN) to model an active rail clamp circuit that contained 6 transistors and 2 passive elements. For their experimentation, they implemented the trained RNN in Verilog-A and demonstrated a relatively high accuracy of under 2% root mean squared error (RMSE). The authors in [15] again use a RNN to model circuits. They discuss how though the training data is discrete, the resulting RNN must be a continuous time RNN (CTRNN) as the majority of circuit simulators use an adaptive time step, which is not known prior to simulating. In [13] the authors worked to simulate an aging circuit using a RNN. One of the circuits used in experimentation contained over $1,000$ transistors in an “IP block” (i.e. internals are not known, just the inputs and outputs to the block) and they saw a speedup of $50x$ while maintaining a low normalized root mean squared error (NRMSE) of just 0.74%. Surrogate modeling of circuits is of particular interest in scenarios such as simulating circuit aging where the incremental damage to each transistor is calculated at every time step, which results in a longer time to complete analysis. The majority of previous work done on this topic has focused on the modeling of a few (1-3) various circuits; however, this work attempts to provide a clear and general path for creating surrogate models for any components or circuits.

Chapter 4

Research Design and Methods

This chapter describes the process of training and implementing a surrogate model in place of a component or circuit in fREEDA. See Figure 4.1 for a flowchart outlining this process. Firstly, the data collection process in LTspice [2] as well as the pre-processing of the data is discussed. Next, the selected transistor models are overviewed. Following this, the AutoML process and subsequent neural network architecture is covered. Lastly, a description of how the surrogate model is implemented in place of the physics-based equations in fREEDA.

4.1 Data Collection

For this work, components and circuits implemented in LTspice are being modeled in fREEDA. The first step in this is collecting the data in LTspice. For components, this process consists of creating circuits of interest in the LTspice GUI and performing transient analysis. For both the BJT and MOSFET, a single circuit was selected for data collection. These circuits were the common-emitter amplifier and the common-source amplifier, respectively. Resistance values were varied for the training data, which affected the overall gain of the circuit. The inputs and outputs were selected based on the implementation of the component in fREEDA and the relevant values were selected in LTspice and exported to a *.csv* file. Similarly in data collection for the circuits, the circuit to be model is created with the LTspice GUI, transient analysis was performed, and relevant values were taken as training data. For the circuits modeled in this work, voltages and currents were taken at the input to the circuit of interest and the input to the load, which was directly connected

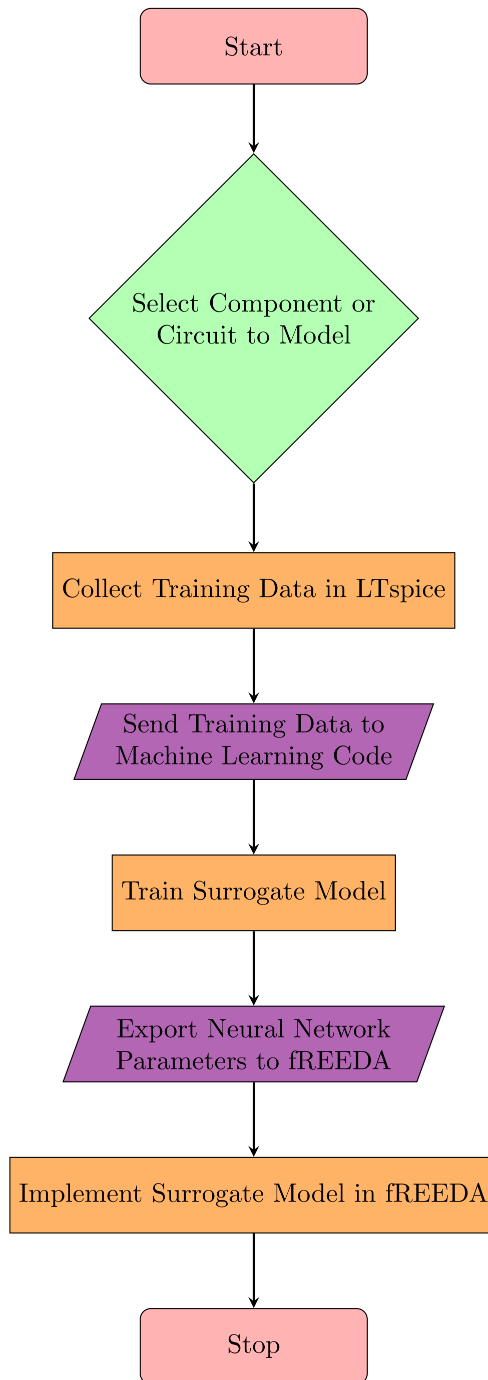

Figure 4.1: Overview of Workflow

to the output of the circuit being modeled. For some of the component modeling, it was necessary to obtain the time derivative of input values as they were used in the training of the surrogate models. The derivatives were approximated in Python via

$$\frac{dx}{dt} \approx \frac{x_{t+\Delta t} - x_t}{\Delta t},$$

where x_t is the value of x at time t and Δt is the time step. It should be noted that the transient analysis in LTspice used a variable time step, so Δt was not constant throughout a simulation. Once the *.csv* contained all the training data needed, it was read into Python for preprocessing and training of the surrogate model.

4.2 Transistor Models

For this work, two specific components and two circuits were selected. The componenets include the 2N2222 NPN BJT and the BSS123 N-Channel MOSFET. The Gummel-Poon model was used for the BJT and the Philips MOS9 MOSFET Model was used for the MOSFET. The circuits modeled were the common-emitter amplifier and the common-source amplifier. The transistors used in the amplifier circuits were the 2N2222 and BSS123, respectively.

The inputs to the physics-based equations for the Gummel-Poon model BJT element in fREEDA are

$$V_{be}, V_{bc}, V_{cjs}, \frac{dV_{be}}{dt}, \frac{dV_{bc}}{dt}, \frac{dV_{cjs}}{dt}, \frac{d^2V_{be}}{dt^2}, \frac{d^2V_{bc}}{dt^2}, \frac{d^2V_{cjs}}{dt^2}$$

with outputs

$$V_{cjs}, V_{bjs}, V_{ejs}, I_c, I_b, I_e$$

where c = collector, b = base, e = emitter, and js = junction substrate. For the Philips MOS9 MOSFET model, the inputs to the physics-based equations include the voltages

$$V_{ds}, V_{gs}, V_{bs}$$

with outputs

$$V_{db}, V_{gb}, V_{sb}, I_d, I_g, I_s$$

where d = drain, g = gate, s = source, and b = bulk.

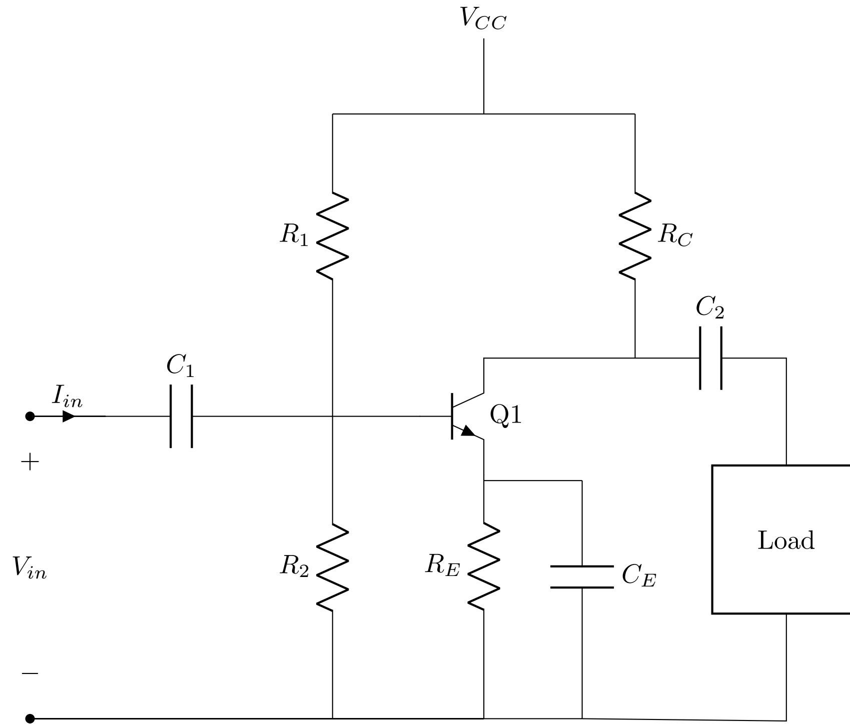

Figure 4.2: Common-Emitter Amplifier

The junction substrate and bulk were both connected to the ground terminal for all simulations. Though used by the physics-based equations for the Gummel-Poon model for the BJT, the second time derivatives were not used as inputs to the neural network as it negatively affected the accuracy. However, using the first time derivatives of the inputs for the MOS9 MOSFET Model helped the accuracy, and were used in the training of the surrogate model. Different transistor models use different inputs to the physics-based equations. As the inputs used for the training of the surrogate models are based on the inputs to the physics-based models, there will likely need to be an effort made to determine which of the inputs to the physics-based model should be used in training to create the highest fidelity surrogate model possible for new transistor models.

For the two circuits tested, the common-emitter amplifier (Figure 4.2) and the common-source amplifier (Figure 4.3), the inputs were selected to be the input voltage and current to the amplifier itself. The outputs were the voltage across the load and the current flowing through it, see Figure 4.5. For all simulations and testing, a passive load of a resistor of varying resistance was used.

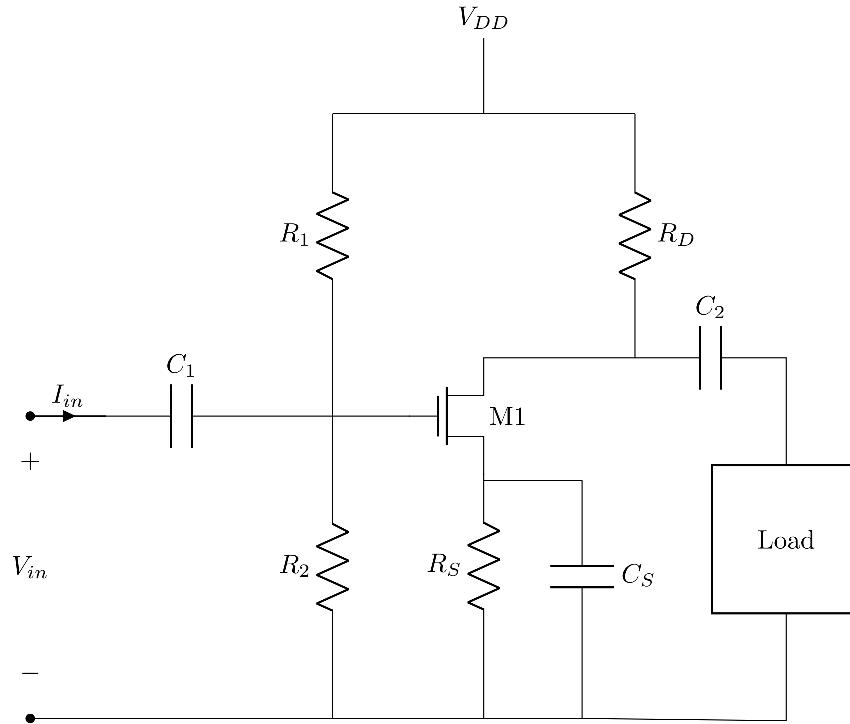

Figure 4.3: Common-Source Amplifier

4.3 Neural Network Training and Architecture

Following training data collection, the next step in building a surrogate model is training a neural network using the Structured Data Regressor model in AutoKeras [7]. When initializing an AutoKeras model, there are parameters that can be set such as maximum number of trials, or number of Keras models tested, loss function, number of epochs, and maximum number of parameters. These values were all left to the default values which were 100 trials, a loss function of mean squared error, 1000 epochs, and the maximum number of parameters was left blank. Training the AutoKeras model consists of a neural architecture search that attempts to find the optimal architecture, or architecture that most minimizes the loss function, given training data. After training, the best model found is exported for future use. As this exhaustive search is time consuming, this process is only conducted once for creating new surrogate models. In addition to of saving the parameters of the best neural network found, AutoKeras also saves the number, type, and size of layers, activation function, the optimizer used, learning rate, if dropout was used, etc. In the case of creating a new surrogate model of a similar complexity to one that has already been trained, it was found that using the

Layer (Type)	Output Shape	Parameters
Input	(None, 32)	224
Dense	(None, 32)	224
Dense	(None, 32)	1056
Dense	(None, 6)	198
Total Parameters		1478

Table 4.1: Component Neural Network Architecture

Layer (Type)	Output Shape	Parameters
Input	(None, 2)	0
Dense	(None, 64)	192
Dense	(None, 2)	130
Total Parameters		322

Table 4.2: Circuit Neural Network Architecture

same architecture was successful. For example, if there was a need to create a surrogate model for a different BJT or a common-emitter amplifier circuit with different passive element values, all that would need to be done is train a network of the same size, activation function, optimizer, and learning rate that was used in the best model found. This process was done for both the components and circuits in this work. The architecture that best modeled the Gummel-Poon BJT was used to train the surrogate model for the Philips MOS9 MOSFET and the architecture found to best model the common-emitter amplifier was used to train the surrogate model for the common-source amplifier. For this work, Tensorflow was used to train the surrogate models once the architecture was known [1]. The architecture used for the components and circuits can be seen in Tables 4.1 and 4.2, respectively. Additionally, the Adam optimizer and a learning rate of 0.001 was used for the training of all neural networks. Lastly, a simple script was written to extract the weights and biases from each layer and save them to a *.txt* file for future implementation of the surrogate models in fREEDA.

4.4 Surrogate Model Implementation

4.4.1 Component Implementation

Once a sufficiently good neural network is found, it is ready to be implemented in fREEDA in place of the physic-based equations. For components, the surrogate model was implemented alongside the physics-based equations in the element's eval routine with a user set flag deciding which

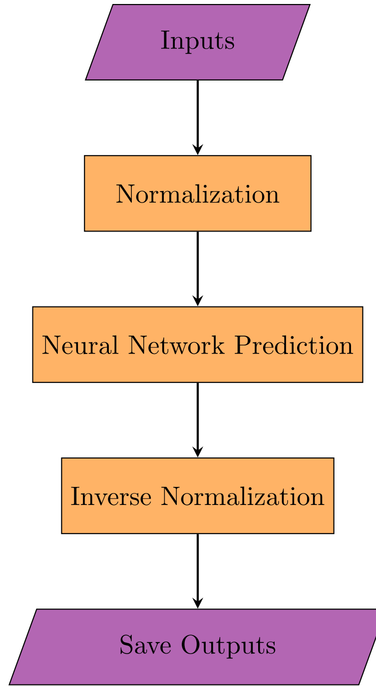

Figure 4.4: Implementation Process

method to be used for calculations. The weights and biases of the neural network are flattened and saved to an element specific *.txt* file. These weights and biases are then read in during the init routine, which is only called once at the beginning analysis, and are saved to 1-D vectors corresponding to its respective layer in the neural network architecture. Then at each time step, the input is normalized based on the training data by (see Figure 4.4):

$$input_{i,normalized} = \frac{input_i}{max(abs(input_{i,training}))}.$$

Note that these maximum values used to normalize the input, and eventually inversely normalize the output of the neural network, were hardcoded in fREEDA. Though in practice, these values may not normalize the inputs and outputs of the neural network exactly between $[-1, 1]$, it will only be slightly above or below this range. This is sufficient as it will keep the inputs and predicted outputs of the implemented surrogate model on a common scale relative to the training data. Following this normalization, the inputs go through the neural network and a prediction is made. Listing 4.1 is an example of a dense layer and the subsequent ReLU operation implementation that resides in the eval routine of the component in fREEDA. As described in the Neural Network Training and Architecture section, the implemented surrogate model is a fully connected neural

network so the rest of the layers and operations are as shown in Listing 4.1. Once the output has been predicted, it must be reversely normalized by:

$$output_j = max(abs(output_{j,training})) \times output_{j,normalized}.$$

Lastly, these outputs are set to the effort (voltage) and flow (current) for the element at time step n.

Listing 4.1: Dense Layer and ReLU in Surrogate Model Implementation

```
//Dense1
for (int i = 0; i < 6; i++) {
    for (int j = 0; j < 32; j++) {
        xdense1[j+k] = dense1weights[j+k]*xnorm[i];
        xrelu1[j] += xdense1[j+k];
    }
    k += 32;
}

for (int i = 0; i < 32; i++) {
    xrelu1[i] += dense1biases[i];
}

//Relu1
for (int i = 0; i < 32; i++) {
    if (xrelu1[i] < 0) {
        xrelu1[i] = 0;
    }
}
```

4.4.2 Circuit Implementation

A NPort element is used to house the surrogate model for circuits in fREEDA. A user can define the number of input and output terminals of the NPort in the netlist of the circuit to be simulated. In this work, the circuits tested were the common-emitter amplifier and the common-source amplifier. For both, the NPort had two input terminals (connected to V_{source}) and two output terminals (connected to R_{load}) (see Figure 4.5).

So the netlist for the common-emitter amplifier and common-source amplifier contained

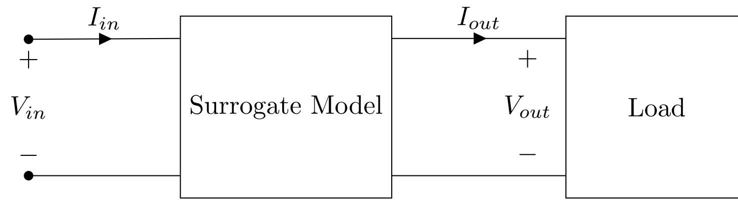

Figure 4.5: Circuit Implementation

three total elements: the voltage source, the NPort element, and the load (in this work just a single resistor). The neural network was implemented similarly to how it was done for individual components. The weights and biases of the neural network were read in and saved during the init routine of the NPort element and then the process shown in Figure 4.4 is completed at each time step.

Chapter 5

Results

The two measurements of interest for the surrogate models created are the accuracy and speedup compared to physics-based equations. This chapter covers both the metrics used and the results that gauged the validity of these models.

5.1 Surrogate Model Accuracy

5.1.1 Components

As discussed in the Research Design and Methods chapter, training data was collected exclusively from LTspice simulations. Therefore, the fidelity of the surrogate models were compared to the traditional physics-based equations in LTspice and not in fREEDA. The BJT and MOSFET components modeled both had six outputs (three voltage and three current values). A plot of the surrogate model's estimate versus ground truth can be seen in Figures 5.1 and 5.2 for the BJT and MOSFET, respectively. The greatest error in prediction tended to occur near the beginning of simulation, during the transient response; however, once steady state was reached the surrogate models were highly accurate. To quantify the accuracy of the models, the metric RMSE (root mean squared error) was used. RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

where n is the number of samples, $\hat{y}_i$ is the neural network prediction at sample i, and y_i is the ground truth at sample i. The scikit-learn package was used for this calculation [11]. The resulting

Output	**RMSE**
V_{cjs}	3.1 $[mV]$
V_{bjs}	2.0 $[mV]$
V_{ejs}	1.3 $[mV]$
I_c	87.4 $[\mu A]$
I_b	170.1 $[\mu A]$
I_e	3.0 $[\mu A]$

Table 5.1: RMSE of BJT Output Variables

RMSE of the BJT and MOSFET components can be seen in Tables 5.1 and 5.2, respectively. For all runs, the RMSE of the voltage values was always orders in magnitude larger than the RMSE for the current values, which is in part explained by the actual value of the voltage and current; the voltage values were on the order of volts whereas the current values were on the order of milliamps or microamps. It can be observed in general, using information from both the plots and tables that describe the surrogate model of components, that the models created were of high accuracy and would be suitable as replacements to the traditional physics-based models in simulation.

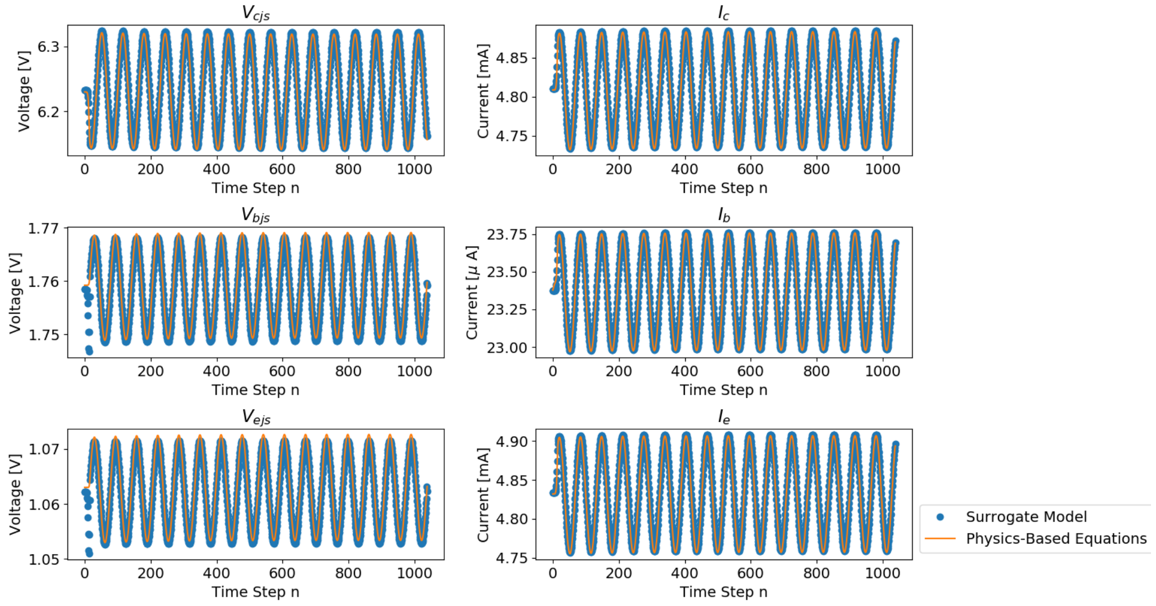

Figure 5.1: BJT Output Variables

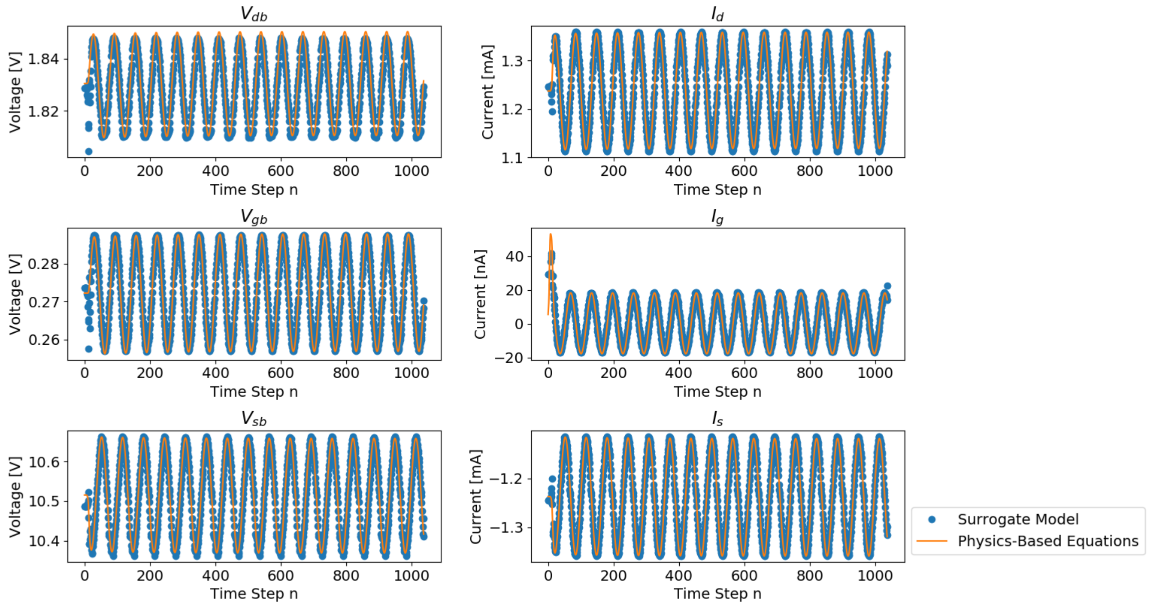

Figure 5.2: MOSFET Output Variables

Output	**RMSE**
V_{db}	4.1 $[mV]$
V_{gb}	1.1 $[mV]$
V_{sb}	12.5 $[mV]$
I_d	6.0 $[\mu A]$
I_g	4.1 $[nA]$
I_s	4.9 $[\mu A]$

Table 5.2: RMSE of MOSFET Output Variables

5.1.2 Circuits

The two circuits in this work that were replaced with surrogate models and tested were the common-emitter amplifier and the common-source amplifier. Similarly to testing the accuracy of the components, the surrogate models of the circuits were compared to data from LTspice using RMSE as the quantifying metric. The outputs for both circuits were v_{out} and i_{out}, where v_{out} is the voltage across the load and i_{out} is the current flowing through the load. Using Figures 4.2 and 4.3 as reference, the specific parameters of the common-emitter and common-source amplifier circuits used in the results section were as follows:

$$R_1 = 24\,k\Omega,\ R_2 = 2\,k\Omega,\ R_c = 9.4\,k\Omega,\ R_e = 470\,\Omega,\ R_{load} = 20\,k\Omega,$$
$$C_1, C_2 = 0.1\,\mu F\ (Coupling\ Capacitors),\ C_s = 0.1\,\mu F\ (Bypass\ Capacitor),$$
$$V_{in}\ (AC) = 100\,mV\ at\ 10kHz,\ V_{cc}\ (DC) = 12\,V$$

for the common-emitter amplifier and

$$R_1 = 200\,k\Omega,\ R_2 = 100\,k\Omega,\ R_d = 960\,k\Omega,\ R_s = 470\,\Omega,\ R_{load} = 50\,k\Omega,$$
$$C_1, C_2 = 0.1\,\mu F\ (Coupling\ Capacitors),\ C_s = 6\,\mu F\ (Bypass\ Capacitor),$$
$$V_{in}\ (AC) = 100\,mV\ at\ 1kHz,\ V_{dd}\ (DC) = 18\,V$$

Output	RMSE
i_{out}	10.9 $[\mu A]$
v_{out}	0.2 $[V]$

Table 5.3: RMSE of Common-Emitter Amplifier Outputs

Output	RMSE
i_{out}	647.3 $[nA]$
v_{out}	25.9 $[mV]$

Table 5.4: RMSE of Common-Source Amplifier Outputs

for the common-source amplifier.

Plots showing the input and output voltages of the amplifier circuits can be seen in Figures 5.3 and 5.4. Note that the plots of the input voltage is in millivolts whereas the output voltage plots are in volts. It is clear from these plots that the surrogate model is able to accurately predict v_{out}, in particular once steady-state is reached. It is worth noting that the reason behind the predictions not being at fixed time distances apart and more clumped together in certain regions (the peaks and troughs) is the adaptive time step that the simulator in LTspice uses. RMSE was also used to verify the accuracy of the surrogate models of the circuits, and was implemented as described in the components section. Tables 5.3 and 5.4 show the resulting RMSE for the common-emitter amplifier and common-source amplifier, respectively. Similarly to the component surrogate models, it should be noted v_{out} is of order volts and i_{out} is of order microamps, which in part explains the multiple orders of magnitude difference in RMSE values.

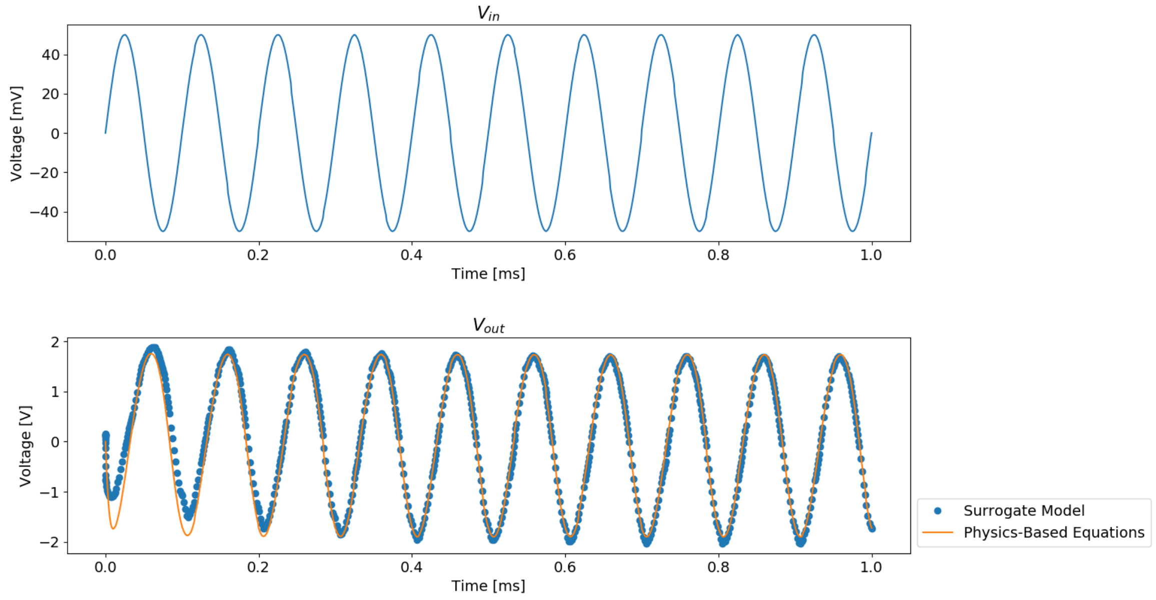

Figure 5.3: Common-Emitter Amplifier Input and Output Voltage

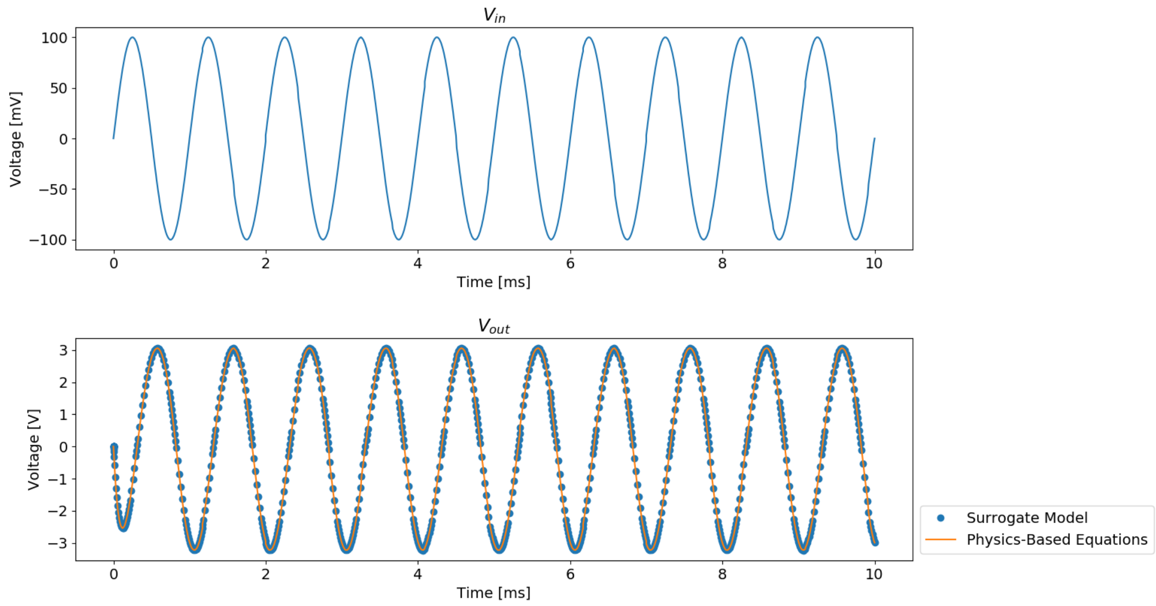

Figure 5.4: Common-Source Amplifier Input and Output Voltage

5.2 Surrogate Model Speedup

5.2.1 Components

The relative speedup of the surrogate models was compared to the traditional methods of circuit analysis in fREEDA. To measure the time difference between the physics-based equations and the surrogate model, the C library *time.h* was used. This measurement took place in the eval routine of the component, which is called at each time step of analysis. For the physics-based equations, the timer started at the first equation and ended when the outputs were saved. For the surrogate model, the timer started at the normalization of the inputs and ended with the outputs being saved. Note that this time difference does not include the reading and saving of the parameters of the neural network but this is only done once, during the init routine of the element. Further, the process of reading and saving the parameters only occurs once, even if there are multiples of the element in the circuit, i.e. if the circuit being analyzed had tens of transistors that were being simulated by surrogate models. The Average Run Time was the average time to complete the physics-based equations and surrogate model over a few different runs, rounded to the nearest microsecond. Speedup was simply calculated as

$$Speedup = \frac{Time_{Physics}}{Time_{NN}}.$$

Results for the two components tested can be seen in Tables 5.5 and 5.6. As the same size neural networks were used for both the surrogate model of the BJT and MOSFET, it follows that the average run time for their respective eval routine are the same. The reasoning for the difference in run time of the representative physics-based equations for the Gummel-Poon BJT and Philips MOS9 MOSFET models is simply the complexity of the models. In the fREEDA implementation of these models, the Gummel-Poon model has 50 parameters and the eval routine is 141 lines whereas the Philips MOS9 model has 125 parameters and the eval routine is 550 lines. As there are multiple transistor models that are more complex than either used here, one would see an even greater speedup as the complexity of the physics-based equations used to model the component increased.

	Average Run Time	Speedup
fREEDA Physics-Based Equations	40 [μs]	-
NN Surrogate Model	8 [μs]	Average: 5x, Max: 10x

Table 5.5: Gummel-Poon NPN BJT Timing

	Average Run Time	Speedup
fREEDA Physics-Based Equations	146 [μs]	-
NN Surrogate Model	8 [μs]	Average: 18x, Max: 30x

Table 5.6: Philips MOS9 N-Channel MOSFET Timing

5.2.2 Circuits

The time difference between circuits and their respective surrogate models was measured in the code for analysis, in this case transient analysis. There is a main time loop in this transient analysis code that runs *number of time steps* times. It is in this loop where the timer was placed, starting at the top of the loop and ending at the bottom of the loop. Tables 5.7 and 5.8 show the resulting speedup of the surrogate models of the common-emitter amplifier and common-source amplifier, respectively.

	Average Run Time	Speedup
fREEDA Physics-Based Equations	320 [μs]	-
NN Surrogate Model	113 [μs]	Average: 3x, Max: 5x

Table 5.7: Common-Emitter Amplifier Timing

	Average Run Time	Speedup
fREEDA Physics-Based Equations	388 [μs]	-
NN Surrogate Model	113 [μs]	Average: 3x, Max: 7x

Table 5.8: Common-Source Amplifier Timing

Average Run Time and Speedup were calculated as described in the components section. The speedup measurement for the circuit surrogate models was not a direct time comparison between the time it takes the neural network to make a prediction and the physics-based equations, as was the case with the individual components. The *run time* measured included everything that had to be done by fREEDA at each time step including updating the effort and flow values at each terminal, the NOX solver [9], and saving any user-defined values. Because of this, the speedup shown in Tables 5.7 and 5.8 is not as large as the individual components.

Chapter 6

Conclusions and Discussion

6.1 Conclusion

This work presents an easy-to-use workflow for creating high-fidelity surrogate models of nonlinear components and circuits. It covers in detail the entire process of creating a surrogate model of either a component or circuit in LTspice and how to implement the surrogate model in fREEDA. The purpose is to decrease the amount of time it takes to complete transient analysis compared to traditional physics-based equations. Also, by using a surrogate model of a circuit, the model can be distributed for use in testing while preserving the intellectual property of the underlying circuit. The components and circuits tested in this work include the NPN BJT, the N-Channel MOSFET, the common-emitter amplifier, and the common-source amplifier. It was shown that that the surrogate models for all the examples speedup the process of transient analysis while maintaining a high accuracy.

6.2 Future Work

There are multiple avenues that could be investigated to further this work. The first step would be to create a library of surrogate models of popular elements and circuits in fREEDA. For example, as it currently stands, if a user wanted to test a different transistor in surrogate model of a common-emitter amplifier, they would need to repeat the whole process to create an entirely new surrogate model. If this varied library is constructed, surrogate models would be more likely used

by fREEDA users.

Secondly, it would be of value to model more complex element models such as the HICUM Level 2 BJT Model. Even if the surrogate model is highly accurate, it is still adding another level of abstraction from the hardware itself. The more accurate the physics-based equations are in modeling the hardware, the more accurate the surrogate models will be, with respect to true hardware implementation. Additionally, as the complexity of the element models increase, so does the time to evaluate the physics-based equations. It stands to reason that surrogate models of the more complex element models could achieve an even greater speedup compared to this work, and would be worthwhile investigating further.

Lastly, there are some parts of the current codebase that could be automated to simplify the process for the user.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Analog Devices. Ltspice xviii, 1998–2022.

[3] Zaichen Chen, Maxim Raginsky, and Elyse Rosenbaum. Verilog-A compatible recurrent neural network model for transient circuit simulation. In *2017 IEEE 26th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, pages 1–3, October 2017. ISSN: 2165-4115.

[4] Angelo Ciccazzo, Gianni Di Pillo, and Vittorio Latorre. Support vector machines for surrogate modeling of electronic circuits. *Neural Computing Applications*, 24, 01 2014.

[5] Dirk Gorissen, Luciano De Tommasi, Wouter Hendrickx, Jeroen Croon, and Tom Dhaene. Rf circuit block modeling via kriging surrogates. In *MIKON 2008 - 17th International Conference on Microwaves, Radar and Wireless Communications*, pages 1–4, 2008.

[6] H. K. Gummel and H. C. Poon. An integral charge control model of bipolar transistors. *The Bell System Technical Journal*, 49(5):827–852, 1970.

[7] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system.

[8] Piotr Kurgan. Efficient Surrogate Modeling and Design Optimization of Compact Integrated On-Chip Inductors Based on Multi-Fidelity EM Simulation Models. *Micromachines*, 12(11):1341, October 2021.

[9] Kevin Long, Raymond Tuminaro, Roscoe Bartlett, Robert Hoekstra, Eric Phipps, Tamara Kolda, Richard Lehoucq, Heidi Thornquist, Jonathan Hu, Alan Williams, Andrew Salinger, Victoria Howle, Roger Pawlowski, James Willenbring, and Michael Heroux. An overview of Trilinos. Technical Report SAND2003-2927, 918383, August 2003.

[10] B. Mutnury, M. Swaminthan, M. Cases, N. Pham, D.N. de Araujo, and E. Matoglu. Macromodeling of nonlinear transistor-level receiver circuits. *IEEE Transactions on Advanced Packaging*, 29(1):55–66, February 2006. Conference Name: IEEE Transactions on Advanced Packaging.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] F.M. Klaassen R.M.D.A. Velghe, D.B.M. Klaassen. Mos model 9, unclassified report nl-ur 003/94. 1994.

[13] E. Rosenbaum, J. Xiong, A. Yang, Z. Chen, and M. Raginsky. Machine Learning for Circuit Aging Simulation. In *2020 IEEE International Electron Devices Meeting (IEDM)*, pages 39.1.1–39.1.4, December 2020. ISSN: 2156-017X.

[14] Michael B. Steer, Nikhil M. Kriplani, Sonali Luniya, Frank Hart, Justin Lowry, and Carlos E. Christoffersen. fREEDA: An Open Source Circuit Simulator. In *2006 International Workshop on Integrated Nonlinear Microwave and Millimeter-Wave Circuits*, pages 112–115, January 2006.

[15] Jie Xiong, Alan S. Yang, Maxim Raginsky, and Elyse Rosenbaum. Neural networks for transient modeling of circuits : Invited paper. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–7, 2021.

[16] M. B. Yelten, Ting Zhu, S. Koziel, P. D. Franzon, and M. B. Steer. Demystifying Surrogate Modeling for Circuits and Systems. *IEEE Circuits and Systems Magazine*, 12(1):45–63, 2012.

Appendices

Appendix A Code Listings

Listing 1: Diode Element in fREEDA

```
#include <freeda/elements/Nonlinear/Diode.h>
#include <freeda_core/constants.h>

// Static members
const unsigned Diode::n_par = 20;

// Element information
ItemInfo Diode::einfo =
{
    "diode",
    "Microwave diode",
    "Carlos E. Christoffersen",
    DEFAULT_ADDRESS"category:diode",
    "2000_06_15"
};

// Parameter information
ParmInfo Diode::pinfo[] =
{
    {"js", "Saturation current (A)", TR_DOUBLE, false},
    {"alfa", "Slope factor of conduction current (1/V)", TR_DOUBLE, false},
    {"jb", "Breakdown saturation current (A)", TR_DOUBLE, false},
    {"vb", "Breakdown voltage (V)", TR_DOUBLE, false},
    {"e", "Power-law parameter of breakdown current", TR_DOUBLE, false},
    {"ct0", "Zero-bias depletion capacitance (F)", TR_DOUBLE, false},
    {"fi", "Built-in barrier potential (V)", TR_DOUBLE, false},
    {"gama", "Capacitance power-law parameter", TR_DOUBLE, false},
    {"cd0", "Zero-bias diffusion capacitance (F)", TR_DOUBLE, false},
    {"afac", "Slope factor of diffusion capacitance (1/V)", TR_DOUBLE, false},
    {"r0", "Bias-dependent part of series resistance in forward-bias (Ohms)",
    TR_DOUBLE, false},
    {"t", "Intrinsic time constant of depletion layer (s)", TR_DOUBLE, false},
    {"area", "Area multiplier", TR_DOUBLE, false},
    {"imax", "Maximum forward and reverse current (A)", TR_DOUBLE, false},
    {"eg", "Barrier height at 0 K (eV)", TR_DOUBLE, false},
    {"m", "Grading coefficient", TR_DOUBLE, false},
    {"aro", "r0 linear temperature coefficient (1/K)", TR_DOUBLE, false},
    {"bro", "r0 quadratic temperature coefficient (1/K^2)", TR_DOUBLE, false},
    {"afag", "Temperature-related coefficient", TR_DOUBLE, false},
    {"xti", "Js temperature exponent", TR_DOUBLE, false}
};


Diode::Diode(const string & iname) : ADInterface(&einfo, pinfo, n_par, iname)
{
    // Set default parameter values
    paramvalue[0] = &(js = 1e-16);
    paramvalue[1] = &(alfa = 38.696);
    paramvalue[2] = &(jb = 1e-5);
    paramvalue[3] = &(vb = -1e20);
    paramvalue[4] = &(e = 10.);
    paramvalue[5] = &(ct0 = freeda::Constants::zero);
    paramvalue[6] = &(fi = .8);
    paramvalue[7] = &(gama = .5);
    paramvalue[8] = &(cd0 = freeda::Constants::zero);
    paramvalue[9] = &(afac = 38.696);
    paramvalue[10] = &(r0 = freeda::Constants::zero);
    paramvalue[11] = &(t = freeda::Constants::zero);
    paramvalue[12] = &(area = freeda::Constants::one);
    paramvalue[13] = &(imax = freeda::Constants::zero);
```

```
60     paramvalue[14] = &(eg = .8);
61     paramvalue[15] = &(m = .5);
62     paramvalue[16] = &(aro = freeda::Constants::zero);
63     paramvalue[17] = &(bro = freeda::Constants::zero);
64     paramvalue[18] = &(afag = freeda::Constants::one);
65     paramvalue[19] = &(xti = 2.);
66 
67     // Set the number of terminals
68     setNumTerms(2);
69 
70     // Set flags
71     setFlags(NONLINEAR | ONE_REF | TR_TIME_DOMAIN);
72 
73     // Set number of states
74     setNumberOfStates(1);
75 }
76 
77 void Diode::init() throw(string &)
78 {
79     v1 = log(5e8 / alfa) / alfa; // normal is .5e9
80     k1 = ct0 / pow(.2, gama);
81     k2 = fi * .8;
82     k3 = exp(alfa * v1);
83     k4 = - ct0 * fi / (freeda::Constants::one - gama);
84     k5 = k4 * (.2 * k1 / ct0 - freeda::Constants::one);
85     k6 = cd0 / afac;
86 
87     // initialize automatic differentiation
88     DenseIntVector var(1);
89     initializeAD(var, var);
90 }
91 
92 void Diode::eval(AD * x, AD * effort, AD * flow)
93 {
94     // x[0]: state variable
95     // x[1]: time derivative of x[0]
96 
97     AD vj, dvj_dx, cj, rs, itmp;
98 
99     if (v1 > x[0])
100     {
101         vj = x[0] + freeda::Constants::zero;
102         dvj_dx = freeda::Constants::one;
103     }
104     else
105     {
106         vj = v1 + log(freeda::Constants::one + alfa * (x[0] - v1)) / alfa;
107         dvj_dx = freeda::Constants::one / (freeda::Constants::one + alfa * (x[0] - v1
    ));
108     }
109 
110     // Calculate the junction capacitance (experimental)
111     // Use a modified function with continous first and second deriv.
112     cj = freeda::Constants::zero;
113     if (isSet(5)) // if parameter 5 "ct0" is set
114     {
115         AD exp1 = exp(10. * (vj - k2));
116         const double k14 = ct0 * gama / fi;
117         const double k15 = k4 * (gama - freeda::Constants::one) / fi;
118         if (vj < freeda::Constants::zero)
119         {
120             cj = ct0 / pow(freeda::Constants::one - vj / fi, gama);
121         }
122         else
```

```
123             cj = (ct0 + vj * (k14 + k15 * vj)) / (freeda::Constants::one + exp1) +
124                   k1 * exp1 / (freeda::Constants::one + exp1);
125     }
126 
127     // add depletion capacitance
128     if (isSet(8)) // is parameter 8 "cd0" is set
129     {
130         cj += cd0 * exp(afac * vj);
131     }
132 
133     // Now calculate the current through the capacitor.
134     // Using the chain rule:
135     // dq/dt = cj(vj) * dvj/dx * dx/dt
136     // x[1] is dx/dt
137     flow[0] = cj * dvj_dx * x[1];
138 
139     // Now use the state variable again to calculate the total
140     // current.  This way, we save some exp() calls.  The total
141     // current is the current through the capacitor plus the ideal
142     // diode current.
143     if (v1 > x[0])
144     {
145         itmp = js * (exp(alfa * x[0]) - freeda::Constants::one);
146     }
147     else
148     {
149         itmp = js * k3 * (freeda::Constants::one + alfa * (x[0] - v1)) - js;
150     }
151     flow[0] += itmp;
152 
153     // subtract the breakdown current
154     if (vj - vb > freeda::Constants::one)
155     {
156         itmp = freeda::Constants::zero;
157     }
158     else
159     {
160         itmp = jb * pow(freeda::Constants::one + vb - vj, e);
161     }
162     flow[0] -= itmp;
163 
164     // Calculate Rs
165     if (cj != freeda::Constants::zero)
166     {
167         if (t / cj > r0)
168         {
169             rs = freeda::Constants::zero;
170         }
171         else
172         {
173             rs = r0 - t / cj;
174         }
175     }
176     else
177     {
178         rs = r0;
179     }
180 
181     effort[0] = vj + flow[0] * rs;
182 
183     // scale the current according to area. All the calculations were made
184     // for a unit area diode.
185     flow[0] *= area;
186 }
```

Listing 2: fREEDA Netlist for Half-Wave Rectifier

```
<netlist>

<!-- This is a comment -->

  <element name="vsource">
    <terminals> 1 0 </terminals>
    <instance> v1 </instance>
    <parameter name="vdc"> 1 </parameter>
    <parameter name="vac"> 1 </parameter>
    <parameter name="frequency"> 1e6 </parameter>
  </element>

  <element name="resistor">
    <terminals> 2 0 </terminals>
    <instance> r1 </instance>
    <parameter name="res"> 2.2e3 </parameter>
  </element>

  <element name="diode">
    <instance> d1 </instance>
    <terminals> 1 2 </terminals>
    <parameter name="charge"> 0 </parameter>
  </element>

  <analysis name="SVTran2">
    <parameter name="tstep"> delta_t </parameter>
    <parameter name="tstop"> 1e-5 </parameter>
    <parameter name="deriv"> deriv_type </parameter>
  </analysis>

  <output>
    <effort>
      <label> v2 </label>
      <terminal> 2 </terminal>
      <domain> time </domain>
      <filename> vdiode.dat </filename>
      <stream> false </stream>
    </effort>
    <effort>
      <label> v1 </label>
      <terminal> 1 </terminal>
      <domain> time </domain>
      <filename> vinput.dat </filename>
      <stream> false </stream>
    </effort>
    <flow>
      <label> idiode </label>
      <element> diode </element>
      <instance> d1 </instance>
      <terminal> 0 </terminal>
      <domain> time </domain>
      <filename> diode_current.dat </filename>
      <stream> false </stream>
    </flow>
    <flow>
      <label> ivsrc </label>
      <element> vsource </element>
      <instance> v1 </instance>
      <terminal> 1 </terminal>
      <domain> time </domain>
      <filename> vsource_current.dat </filename>
      <stream> false </stream>
    </flow>
```

```
64   </output>
65
66   <options>
67     <parameter name="delta_t"> 1e-8 </parameter>
68     <parameter name="deriv_type"> 1 </parameter>
69   </options>
70
71 </netlist>
```