Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Department of Electronic & Electrical Engineering

Including Measured Frequency Domain Data within Time Domain Simulations

Daniel Sloggett
16321200
Trinity College Dublin
sloggetd@tcd.ie
Supervisor: Dr. Justin King

28th of April 2021

A thesis submitted in partial fulfilment
of the requirements for the degree of
MAI (Electronic and Computer Engineering)

Declaration

☐ I agree that this thesis was completed in line with the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

☐ I have completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

☐ I agree that this thesis will not be publicly available but will be available to TCD staff and students in the University's open access institutional repository on the Trinity domain only.

Signed:

Daniel Sloggett

Date

Contents

Lay Abstract

This project investigates using measured experimental data to simulate more complex circuits. We consider a component to be linear when its value remains constant with voltage and/or current. When a component is linear, it can amplify a signal without shifting the signal's frequencies. We can characterise these components by experimental measurements for use in radio frequency applications. However, if a component is not linear then we cannot use this technique. Combining these measured data with non-linear components requires special treatment. In this project, we compare two of such competing methods by both speed and accuracy. A custom simulator was made to allow this comparison to be easily made. It was concluded that the two methods performed well in different situations, and that the treatment of the non-linear parts of the circuit was important for accuracy.

Abstract

This project aims to investigate the usage of measured frequency domain data such as scattering parameters, with non-linear time domain simulations. These comparisons are made by implementing two methods of transforming scattering parameter data for use in the time domain. The methods implemented are the non-uniform discrete time impulse response [NUDTIR] method, and the vector fitting using recursive convolution [VFRC] method. These methods are compared in a custom built circuit simulator. It was found that VFRC has issues with passivity, but is faster than NUDTIR for the same timestep. It was also found that the NUDTIR method behaved better with larger timesteps than the VFRC method. Both methods exhibit issues regarding higher frequency harmonics. It was concluded that NUDTIR and VFRC excel in different situations, and that the harmonics introduced by non-linear elements should be treated with care.

Introduction

Including frequency domain data in time domain simulations is a complex task. Different components lend themselves to representation in either the time domain or the frequency domain. Non-linear components are more suited to time domain simulations, whereas measured data such as scattering parameters [S-parameters] are more easily described and understood in the frequency domain. Difficulty arises when we try to perform analysis of circuits that require both non-linear components, and components better described in the frequency domain. Current simulation packages tend to use an approach called *Vector Fitting*, first described by Gustavsen [1] to transform the measured frequency domain data into the time domain. In general this method is quite fast in simulation time, as it minimises the access time to the previous time steps via a method called *recursive convolution*, which is described by Semlyen and Dabuleanu [2]. This combined method will be referred to as Vector Fitting - Recursive Convolution [VFRC].

In this document an alternative method is compared to this S-parameter recursive convolution method; obtaining a *(non-uniform) discrete time impulse response* [(NU)DTIR] using a forced causal *discrete Fourier transform* [DFT], followed by a non-uniform convolution. This project aims to investigate the advantages and disadvantages of this method. The potential importance of this investigation, is that it could lead to faster and potentially more accurate simulations in certain conditions.

Through the course of this project a general purpose simulator was developed for high frequency circuit simulation. It has the ability to read in a general netlist, with which it can generate and solve a time domain MNA system.

Background

4.1 Discrete Fourier Transform and Causality

One of the key differences between the Discrete Fourier Transform [DFT] and the continuous Fourier transform, is periodicity. The continuous Fourier transform does not require a signal to be periodic. On the other hand, the DFT and/or *inverse discrete Fourier transform* [IDFT] imposes periodicity when they transform from one domain to another. For example, a finite length time domain signal is periodically extended in the frequency domain. Similarly, for finite frequency responses we assume a periodicity in the impulse response. This has implications for how we must interpret the obtained signal from an IDFT in the time domain. Namely, we must ensure that the signal is causal in the time domain.

A *Causal Signal*, in a digital signal processing sense, can generally be defined as when the value of a signal only depends on the previous or current state of the system; i.e. does not depend on future values. A DTIR obtained via an IDFT is said to be non-causal when the value at $t = a$, where a is a negative value, is non-zero. According to this definition, a signal which is periodic, such as that obtained from an IDFT, cannot be causal. As such, a definition of periodic causality will be considered as described by Oppenheim and Schafer [3], this is seen in Equation 4.1. Where $\widetilde{x}_N$ is a periodic signal in the time domain, with N is the number of values in a single period. This equation states that so long as all values in the second half of a period are 0, the signal is considered *periodically causal*.

$$\forall n.(\frac{N}{2} < n < N \rightarrow \widetilde{x}_N[n] = 0) \tag{4.1}$$

This leads to the natural question of what can we do in order to ensure that a signal is periodically causal as we transform it from the frequency domain to the time domain. This is a question that was examined by T. Brazil [4]. Firstly, we know that for a real valued signal in the time domain $F[m] = F^*[N - m]$, where F is the discrete frequency response. Using this fact, we can derive Equation 4.2

$$f[0] = 2 \sum_{m=1}^{N/2-1} \mathbb{R}\{F[m]\} + F[0] + F[N/2] \tag{4.2}$$

Now let us consider a potentially non-causal frequency domain signal. Another way to phrase "being causal", is to say that the signal is finite in the time domain. We know from the above discussion on periodicity, that if the signal is of finite length in the time domain, then it is implied

to be continuously periodic in the frequency domain. This results in Equation 4.3

$$\mathbb{I}\{F[N/2]\} = 0 \tag{4.3}$$

$$S[m] = S_R[m] + jS_I[m] \tag{4.4}$$

$$F[m] = (S[m] - K)e^{-j\omega\tau m} \tag{4.5}$$

Now let us massage our original data as in Equation 4.4 into the form shown in Equation 4.5. This form is strategically chosen as we need some sets of transformations that are easily reversible in the time domain, which will result in a causal signal to suit Equation 4.3. One way to do this, is by multiplying $F[m]$ by an exponential, which is equivalent to a time shift. But in order for this to work, we need to timeshift by a value of $\tau < T_s$ and we need a way to ensure that $f[0] = 0$. Because we have two conditions to satisfy, we need two transforms combined as shown in Equation 4.5. When we undo this transformation in the time domain, it will result in a NUDTIR. This means that the time difference between taps is not constant, and as such we will need a time vector and a value vector to map each tap value to a time. This is particularly important if we wish to test the accuracy of ignoring low magnitude value taps as described by Schutt-Aine in [5].

$$f'[n] = \begin{cases} 0, & \text{if } n = 0 \\ K, & \text{if } n = 1 \\ \mathcal{F}^{-1}\{F\}[n-1], & \text{otherwise} \end{cases} \tag{4.6}$$

$$t'[n] = \begin{cases} -\tau, & \text{if } n = 0 \\ 0, & \text{if } n = 1 \\ T_s(n-1) - \tau, & \text{otherwise} \end{cases} \tag{4.7}$$

4.2 Modified Nodal Analysis

Simulating a circuit is a common task in circuit design. Specific code for an RLC and non-linear time domain circuit can be expressed and solved as a system of differential algebraic equations. However, this is not feasible to do in the general case as when the size of the circuit increases, the circuit becomes increasingly more complex to capture with such a manual model. As such, a systematic way to construct, parse, and solve an arbitrary circuit must be used. The main advantage to this approach, is we can consider the discretisation of elements separately rather than as a complete system. The book Circuit Simulation by Najm [6] examines the foundation for how the theory for such a piece of software is formulated. Najm starts by going through the theory of representing circuits as graphs, and using a matrix to represent the connection between nodes on said graph.

$$\mathbf{A}\mathbf{x} = \mathbf{y} \tag{4.8}$$

$$\mathbf{x} = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix} \tag{4.9}$$

$$\mathbf{y} = \begin{bmatrix} i_1 \\ i_2 \\ \dots \\ i_n \end{bmatrix} \tag{4.10}$$

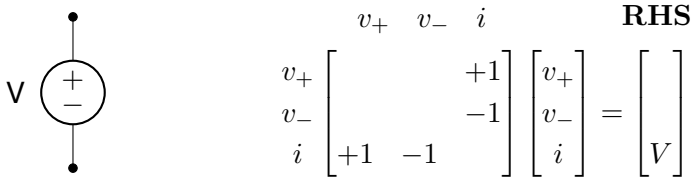
The matrix is constructed as in Equation 4.8, where $\mathbf{A}$ is a matrix constructed to represent the admittance from the node whose index is the column, to the node whose index is the row, excluding the reference node, $\mathbf{x}$ is the nodal voltage and $\mathbf{y}$ is the current vector. This is the vector containing currents flowing into the node, which cannot be accounted for from voltage drops along the element. The circuit can be solved for the nodal voltages by solving for $\mathbf{x}$. Such a construction is known as *Nodal Analysis*.

$$\mathbf{x} = \begin{bmatrix} \mathbf{v} \\ \mathbf{i} \end{bmatrix} \tag{4.11}$$

$$\mathbf{y} = \begin{bmatrix} \mathbf{v} \\ \mathbf{i} \end{bmatrix} \tag{4.12}$$

One shortcoming of this method, is that the system of equations can only account for what Najm refers to as *Group 1* components. Group 2 components such as voltage sources are not

easily captured with this method. To overcome this issue, Najm modifies the nodal analysis to add the ability to have voltage sources in the y vector. Najm modifies $\mathbf{x}$ and $\mathbf{y}$ as seen in Equations 4.11 and 4.12. By doing so, we can represent voltage sources and other group 2 components. This modification is known as *Modified Nodal Analysis*. One major advantage of Nodal Analysis and MNA, is that the matrix for the system can be obtained by summing so called *stamps* for each component. A stamp of a component defines how nodes are connected, and what effects that induce by defining the component's element equations. This is incredibly useful and allows us to easily construct the MNA system for an arbitrary netlist of pre-implemented components. An example of a voltage source and its MNA stamp is seen in Figure 4.1.



***(a)** Voltage Source* ***(b)** Voltage Source MNA stamp*

***Figure 4.1:** Voltage source and its MNA stamp*

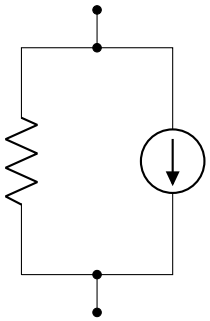


***Figure 4.2:** Norton Equivalent Two Port Companion Model*

We can extend MNA to work with non-linear components too. Najm explains how we can use the Newton-Raphson method to linearise the non-linear model locally using a *companion model*. It is common to replace a complex component with a companion model. Najm often uses a Norton equivalent circuit, as in Figure 4.2, to represent an arbitrary component in the circuit, as it can model the resistance/admittance of the circuit, and it can model current being introduced to the circuit too; that is to say, it directly relates to the components in the admittance and current matrices of the MNA system. This MNA technique can also be expanded to work for transient simulations, where capacitors and inductors are introduced. Najm outlines how one can apply several numerical integrators to find the stamps for capacitors and inductors. The same Newton-Raphson technique can be applied to these dynamic components for non-linear capacitors and inductors.

4.3 Radio Frequency Simulations

4.3.1 Waves and Scattering Parameters

In RF design signals are often abstracted to being thought of as power waves. This is done as a way to avoid working with currents and voltages, and instead focus on the transfer of power. This is particularly relevant when working with scattering parameters [S-parameters]. Ludwig and Bretchko [7] outline how S-parameters use two power waves, the normalised incident power wave a and the normalised reflected power wave b as defined in Equations 4.13 and 4.14. Where u is the voltage across that port, i is the power flowing into the positive terminal of the port, and z_{ref} is the reference impedance of the wave.

$$a = \frac{u + iz_{ref}}{2\sqrt{z_{ref}}} \tag{4.13}$$

$$b = \frac{u - iz_{ref}}{2\sqrt{z_{ref}}} \tag{4.14}$$

$$b(j\omega) = a(j\omega)S(j\omega) \tag{4.15}$$

$$b(t) = (a * s)(t) \tag{4.16}$$

Ludwig and Bretchko show that linear multiport systems can be characterised by taking the quotient of a and b driven at several frequencies, where the non-driven ports are terminated with the reference impedance. A *vector network analyser* uses this principle. The sequence formed by using these measured data with the driven frequency are known as S-parameters. It can be shown that the values of a and b can then be related in the frequency domain as shown in Equation 4.15, where ω is the angular frequency, and S is the frequency domain S-parameter. In the time domain this relation is as in Equation 4.16, where t is the time, and s is the time domain S-parameter.

4.3.2 Simulation

Components which are suited to frequency domain representations must often be converted to a time domain representation in order to perform a transient analysis with components whose ideal representation is in the time domain. This task is not an easy one, and has been the subject of much research in the field of RF design [1][2][8][5].

$$\sum_{i=0}^{N} \frac{c_i}{s - p_i} \tag{4.17}$$

$$\sum_{i=0}^{N} c_i e^{jp_i t} \tag{4.18}$$

One of the key breakthrough pieces of research was performed by Gustavsen [1] [9] [10], and examined the use of a fitted rational approximation of S-parameter data via the least squares method. He notes that through a series of manipulations of equations, one can arrive at an iterative approach to fit a partial fraction expansion to the S-parameters in a *poles and residues* form as in Equation 4.17, where p_i is a pole, and c_i is a residue. This form in particular lends itself to being transformed to the time domain easily, as one can easily perform an inverse Laplace transform to convert the poles and residues to a sum of exponentials as in Equation 4.18. Not only does this make the transform easy to compute, but it also allows easy convolution through a method demonstrated by Semlyen and Dabuleanu called *recursive convolution* [2].

This method, while extremely popular, can be shown to slow down significantly as the complexity of the S-parameters increases [8]. In 2011 Schutt-Aine compares *Order Reduction* methods to different methods of convolution, and it is found that there can be orders of magnitude in the difference between his proposed *fast convolution* method and the popular vector fitting with recursive convolution method. This proposed method examines the IFFT of the frequency domain data, and came to the conclusion that for the vast majority of the time domain impulse response given by S-parameters, the magnitude is sufficiently small that we can ignore their contribution to the simulation, the example given, was that for a 1601-point IFFT of S-parameter data, only 20 points of the data are larger than 1% of the maximum absolute value. This leads to a tremendous speed up over the vector fitting method, but at the expense of potential accuracy losses.

Later, in 2018, Schutt-Aine goes into further detail on how one can actually implement this idea in the context of circuit simulation [5]. He goes on to propose an equivalent model of varying order than can be used in simulations to great effect. For comparisons sake, he shows how this method compares in accuracy to a simulation performed in NGSpice. One can see that the results from the fast convolution are notably smoother, and lack some of the higher frequency detail that is present in the NGSpice simulation. In this paper, Schutt-Aine did not take into account the causality of the DTIR. This will lead to inaccuracies if the signal is non-causal, as discussed in Section 4.1.

4.4 Automatic differentiation

Automatic differentiation is using a way to abstract away the process of determining the numerical value of a derivative at a point. The core mechanism of automatic differentiation is using the predefined rules of differentiation such as the chain rule, product rule, quotient rule, as well as some predefined functions and their derivatives to build up more complex functions whose derivatives can then be calculated. This is often implemented using object oriented programming features such as operator overloading.

For an example, take the function for the non-linear capacitors used in Figure 5.3 and Appendix C. We can see that the derivative is comprised of repeated use of the chain rule, which is naturally expressed using a stack as in function calls.

$$C(u) = C_p + C_o(1.0 + \tanh(P_{10} + P_{11}u)) \tag{4.19}$$

$$C(u) = C_p + C_o(1.0 + \tanh(X(u))) \tag{4.20}$$

$$C(u) = C_p + C_o(Y(u)) \tag{4.21}$$

$$\frac{dC(u)}{du} = \frac{dC(u)}{dY(u)}\frac{dY(u)}{dX(u)}\frac{dX(u)}{du} \tag{4.22}$$

$$\frac{dC(u)}{du} = (C_o)(\operatorname{sech}^2(X(u)))(P_{11}) \tag{4.23}$$

The C++ program will leverage the compiler to generate and optimise the complex functions' derivatives. While calculating the derivative, one can calculate the original value of the function at the same point for only a marginal extra computational cost. So if both values are required, we can optimise around this fact. The main advantage is that one can write the function once, and not have to concern themselves with how the derivative for that function is found. An example of automatic differentiation similar to that used in the developed simulator is shown in Listing 4.1. It can be show in `AutoDiffTest.cpp` that this auto-generated code is marginally faster than the analytical version of the same code.

```
1  // This is using the self-developed library for automatic differentiation
2  // variables r1 and r2 being reference 1 and reference 2 respectively
3  // alpha, beta, gamma, delta, xi, lambda, mu, zeta, and Vto are constants
4  namespace AD = AutoDifferentiation;
5
6  using ADT = AD::DiffVar<double, 2>;
7  ADT V_gs(r1, 1, 0);
8  ADT V_ds(r2, 0, 1);
9
10 auto Vgst = V_gs - (1 + beta*beta) * Vto + gamma * V_ds;
```

```
11 auto Veff = 0.5 * (Vgst + AD::sqrt(AD::pow(Vgst, 2) + delta*delta));
12 auto power = lambda / (1 + mu * AD::pow(V_ds, 2) + xi * Veff);
13 auto area = alpha * V_ds * (1 + zeta * Veff);
14 auto f1 = AD::tanh(area);
15 auto Ids_lim = beta * AD::pow(Veff, power);
16 auto Idrain = Ids_lim * f1;
17
18 // Here Idrain[0] is the value of the function
19 // Here Idrain[1] is the value of the derivative wrt r1
20 // Here Idrain[2] is the value of the derivative wrt r2
21 // I_ds is the equivalent current calculated as from the jacobian
22 auto I_ds = Idrain[0] - Idrain[1] * r1 - Idrain[2] * r2;
```

Listing 4.1: *I_{drain} Auto Differentiation example.*

4.5 Computational Principles

4.5.1 Processor Pipelining and Out of Order Execution

In modern *complex instruction set computers* [CISC] each instruction may take several instruction cycles to fetch and execute. This means that if a processor were to execute each instruction in order one after another, then large parts of the CPU would lay idle for multiple CPU cycles until the current instruction finishes executing. As such, a pipeline is employed so that at each clock cycle different aspects of different instructions can be executed at the same time. This leads to faster sequential execution of instructions.

However, in situations where data dependencies occur, the pipeline will be forced to wait on the result of previous instructions. This is known as a *pipeline stall* and results in a *bubble* in the pipeline. In general the biggest culprits for pipeline stalls are memory accesses, conditional jumps/executions and direct memory dependencies. In order to mitigate the effects of this, the compiler will attempt to leave as much space between data dependant instructions as possible, and the CPU will attempt to optimise the order that instructions are carried out, even if that means altering the order that instructions are executed compared to the source machine code. This style of execution is called *out of order execution* [OoOE].

In general, OoOE leads to large speed ups, but as a result it can make optimising instructions at the assembly level difficult due small instruction order changes leading to a different execution queue. It also leads to a desynchronisation between the *program/instruction counter* [PC] and the actual instruction stalling execution. It is for this reason that sampling profiling for assembly must be done with the knowledge that the most commonly sampled instruction may not be the instruction that is causing the pipeline stall, as the sampler will simply check what instruction the PC is pointing at.

4.5.2 Caching

When one is writing high performance software, one cannot ignore the influence of the cache on speed. CPUs in the modern day have approached a plateau in terms of raw frequency increases. As such other optimisations have come into play to increase the speed of programs. In general a large bottleneck for certain applications has always been access to external memory. That includes accessing data stored out in main memory, also known as *RAM*. In order to mitigate this, a local copy of recently used data is stored locally on the CPU in a special region called the cache. When a program tries to pull in data from main memory, we first check if it is in the cache, if it is, the time taken to access the memory can be reduced by potentially several orders of magnitude. This is called a cache hit. However, if we do not find the data, we must go to main memory to fetch it, which is orders of magnitude slower and introduces bubbles to the pipeline. This is called a cache miss. Caches often rely on a principle known as the principle of data locality. That is the assumption that if you access data in memory, that you are more likely to need data that is beside that piece of data in the near future. As such, when we fetch data from main memory, we gather a whole *cache line* worth of data, and thus evicting a previous line. This behaviour rewards well thought out layout of data in memory, and can lead to extreme slow-downs if we do not take caution to avoid *polluting the cache*.

Polluting the cache, occurs when we pay no attention to the order in which we organise and access data in memory, leading to useful data being evicted from the cache in order to accommodate this data. The cause of this is often access of fragmented data. As such, if we need to access a lot of data in a short period of time often, it is best to organise the data in such a way that the machine can sequentially access that data. For example, as an array rather than a linked-list.

4.5.3 Branch Prediction

Conditional jumps and conditional execution are often referred to as a *branch*. The naïve approach to dealing with branches in a pipeline is to stall the entire pipeline until the condition can be resolved before continuing, introducing a bubble to the pipeline. This leads to a performance penalty at every branch taken. As such, this approach is not taken in practice. Instead a portion of dedicated hardware has been developed called a *branch prediction unit* [BPU]. The purpose of this unit, is to store what path of a branch was taken the last n times that it was executed, where n is the history depth. Using that information, the BPU will attempt to predict what branch the processor will take. Using this prediction, the processor will begin the early stages of fetching/executing the instructions of that branch, and in some processor it may even begin executing multiple branches. This means the overhead for a branch which consistently takes the same path is less than that of an unpredictable branch, an example of such a branch is a long for/while loop. However, this logic is not perfect and will sometimes mispredict. If this occurs, and the processor only executes one branch at a time, this will introduce a bubble as the mispredicted branch must be discarded. This is why unpredictable branches should be avoided at all costs in tight control loops.

4.5.4 Programming Paradigms

In computer software development, there are many differing programming paradigms. The goal of programming paradigms is to provide a general rule and direction to aid in designing the architecture of software. The paradigm used is predominantly determined by the language the software is written in. There are two major categories: Imperative, and declarative. Imperative programming is telling the machine how to change its state; whereas declarative programming states the desired outcome, and not how states must change to achieve this.

In this document, I will only be taking a deeper dive into the two subcategories of imperative programming. Procedural, and object-oriented. Procedural is what languages such as C fall in to. They are driven by the idea that instructions are divided into procedures, which can be reused multiple times to save on time developing and debugging. This ultimately leads to packaging functionality based on relations between data-structures, variables and procedures.

Object-oriented programming has a distinct difference to procedural programming. Object oriented programming aims to package functionality up based on externally exposed behaviours and hiding of irrelevant data, rather than a raw data object passed to specific functions.

Each approach to programming comes with its own strengths and weaknesses. Generally speaking procedural programs tend to be focusing at a much lower level due to their focus on data flow. However they suffer when a shared state is needed. Such an example of this would be the C's standard maths libraries, which have mass amounts of hidden state via global variables such as error state, which can lead to program-wide ramifications for actions as simple as calling some floating point operations. It also leads to not being able to utilise these functions in a C++ `constexpr` expression. In general, object-oriented design tends to help with this issue. It keeps all state that is relevant to a *class* in one specific place; that is inside the object itself. This is called encapsulation. The downside to the typical encapsulation models, is that it can still hide state from the user. And more often than not, it leads to more hidden side-effects, though these side-effects may not be as far reaching as global state changes.

The issue arises when one tries to follow these suggestions of software architecture too rigidly. It leads to sub-optimal methods/procedures simply to fall cleanly into a paradigm. The reality for high performance code, is that layers of abstraction can have a profound affect how your code is run, and how efficient it is. In particular, if one is not careful, it begins to pollute the cache. This is because one must follow long chains of pointers through memory to reach the data you actually want. (See Section 4.5.2) To remedy this, there exists another design philosophy called *Data-Oriented Design*. This approach aims to leverage the convenience and packaging of object-oriented design, but instead of focusing on interfaces, we let the design of the objects, and the methods associated with them be sculpted by the resources that are accessed/required for the operation. This may lead to less intuitive interfaces on the exterior, but for the computer it is easier to optimise, and can lead to more performant code. This form

of design sees major development in the game engine industry. This industry shares a lot in common with the high performance simulation industry, due to the vast amount of floating point operations on large sets of data, with a goal of performing them as quickly as possible. A simple example of a data-oriented design can be choosing to flatten a 2D array into a 1D array, with an external interface still allowing a pseudo-2D access, while not sacrificing the cache locality that is afforded by a 1D array. Further optimisation can also be achieved by deciding what way the rows and columns are organised based on how they tend to be accessed. A great resource that details the philosophy of this design pattern is written by Richard Fabian, called *Data-Oriented Design* [11].

4.5.5 Multithreading

Some designs lend themselves heavily to multithreading and software parallelism. Such software tends to be several independent but similar actions that do not overlap in their memory writes. Due to reasons discussed later in this section, the sections are also relatively long to compensate for the cost of creating and destroying threads or adding the correct data to a threadpool.

Unfortunately, the type of software developed for this project falls into the second category of software. Simulations such as those for circuits tend to be highly serialised as they are progressively stepping through the time domain, and every node tends to affect one another. The best potential candidate for multithreading, the convolution of the DTIR, does not see any speed gains due to the cost of starting threads being greater than the cost of waiting to process the taps sequentially. There is the potential that with higher port S-parameter systems it may become worth it, but for the two port systems explored in this project, it only slowed down execution.

In general multithreading can very memory intensive due to the structure of memory in a processor. Most processors do not share all caches between cores, and as such the data must be fetched into cache for each processor independently, and when items are dirtied they must all be updated. This is made worse when we have threads sharing writeable memory, as depending on aliasing, we must employ access control methods such as mutexes. Adding to this issue is the fact that in general threads are expensive to create and destroy, this is because we must make several calls to the operating system to allocate the memory and request the thread. One way around this issue is to utilise a fixed size pool of threads that collect *tasks* to perform, and sit idle otherwise. This is known as a threadpool. Threadpools are complex in their own right, and when a rudimentary threadpool was tested, it was still slower than performing the same task sequentially.

4.5.6 Equation Solving Methods

There are several methods to solving systems of equations. QR decomposition is better suited to rectangular matrices and is useful for least squares fitting problems. An example of where least squares is useful, is in the solving of poles and residues for Vector Fitting. It is not, however, the fastest solver available for solving general square matrices as used in MNA. For this reason LU decomposition was focused on.

$$A = LU \tag{4.24}$$
$$Ax = y \tag{4.25}$$
$$LUx = y \tag{4.26}$$
$$L(Ux) = y \tag{4.27}$$
$$Ux = x' \tag{4.28}$$
$$Lx' = y \tag{4.29}$$

Generally speaking, LU decomposition, and its derivatives with partial pivoting, are fast square matrix decomposers. They allow us to decompose matrices into a pair of upper and lower triangular matrices that we can solve in multiple stages via forward and backward substitution. They are orders of magnitude faster than naïve Gaussian elimination and substitution. This is done by leveraging the fact that L and U are triangular matrices, and therefore substitution can be done in stages, in order, greatly simplifying the algorithm needed to solve each stage. The general flow for solving a system with LU decomposition is to decompose the system into the form seen in Equation 4.26 then we rearrange into the form seen in Equation 4.29. We then solve for x' using substitution. We can then finally solve Equation 4.28 for x.

Design and Implementation

5.1 Design Principles

The general design and architecture followed for this project will be a predominantly data-oriented design for often accessed underlying structures, with extensible aspects such as components following a more object-oriented design. This is done to try maximise use of the cache will still allowing rapid development of components, without changing the underlying flow of the simulator.

As this project is ultimately designed to be a fast running implementation, a general preference is given to speed over memory usage. This may mean keeping local copies of duplicate data in order to avoid polluting the cache when just one value is needed. The cost of this is that the data is harder to keep synchronised, and it leaves the potential for bugs as a result. This is a trade-off that was made

5.2 Design Overview

5.2.1 Intended Simulation Flow

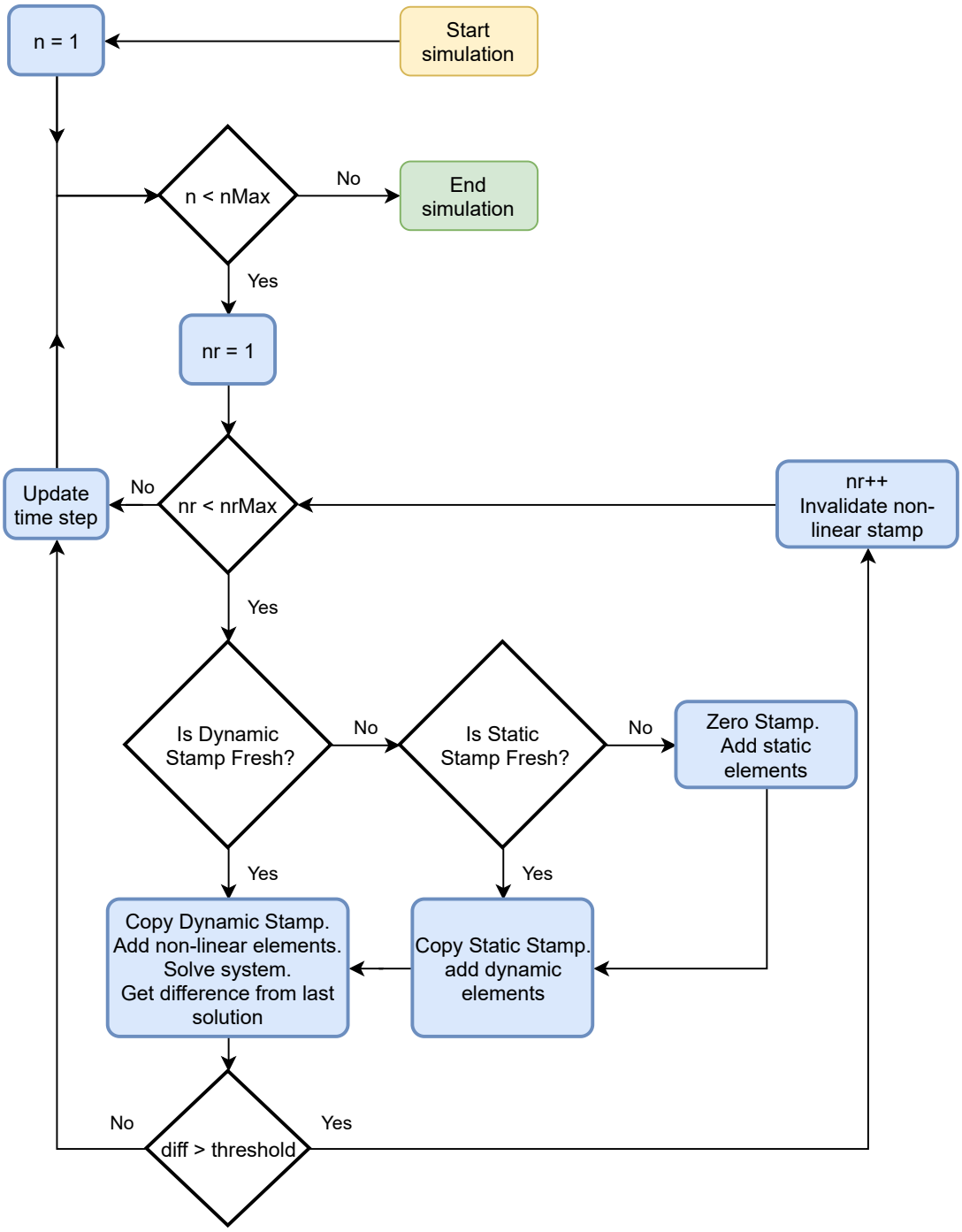


Figure 5.1: *Intended simulation flow*

Figure 5.1 shows the general flow for how the simulation is run after the netlist is read in, and the DC operating point for the starting time has been found. We can see it consists of multiple nested loops with theoretically as few conditionals as possible to avoid branch prediction misses. Furthermore, the Newton-Raphson loop terminates when the solution of the next step has a maximum difference of less than a pre-set threshold. This was found to occur roughly seven steps into the loop for the final simulated benchmark.

Each aspect of the stamp is only generated once when needed then cached for later use. Only when the stamp is marked as dirty is it regenerated. The net effect of this is that we minimise the calculations for dynamic and static elements. This is particularly important for the DTIR S-Parameter blocks as their stamps are very intensive to compute.

5.2.2 Main Class and Data-structure Overview

SimulationEnvironment

This class retains all of the relevant simulation data needed to run the simulation as well as the methods to actually run the main loop of the simulation as detailed in Figure 5.1. Primarily it contains preallocated space for use in the simulation to avoid excess allocation and deallocation of data. It also keeps track of the number of nodes, currents and extra currents needed for DC analysis. It contains the struct `CircuitElements` to keep track of the netlist.

CircuitElements

This struct contains each of the *components*, which are separated into three major category based on what type of element they are: `Static`, `Dynamic`, and `NonLinear`. Each element type has its own vector (collection) to avoid excess calls to empty virtual functions, which as [11] explains is an expensive operation that pollutes the cache. This struct also contains space to cache generated stamps for each of these three component types, as well as a flag to state if they are dirtied or not.

Component

This struct is an abstract base class from which all implemented components derive from. It contains functions to add the components `Static`, `Dynamic`, and `NonLinear` contributions to a stamp. They also contain functions that allow the component to update stored state at the end of each time-step. This functionality combined with the stamp caching in `CircuitElements` simplify the flow of generating stamps to that shown in Figure 5.1.

Github Repository

For full implementation details of the developed circuit simulator consult the project's Github repository [12].

5.2.3 Component Development Flow

The circuit simulator has been designed to allow one to easily add components into it. To do so, one must create a new struct deriving from the base class of `Component`. Then one must override the relevant stamp construction functions, timestep update function, and netlist parsing function. Following this, one must add the component to the header containing `CircuitElements`, and update the parsing function in `SimulationEnvironment` to allow it to call the netlist parsing function. From here the component should be ready to use.

5.2.4 Inputs to the Program

The program, when started on Windows, will prompt the user for a netlist to parse. Otherwise the user must provide a netlist via the first argument to calling the program. This netlist, in general, will contain a unique component prefix, a designator. Following that it will most commonly expect the node numbers for the terminals of the device, and any other defining values. The netlist should also contain the `.transient` directive, which will provide a starting time, an ending time, and a timestep. Optionally one can provide the directive `.graph` to request for the program to graph the output of the simulation, providing that the Python, matplotlib and numpy environments were installed for compilation.

5.3 Combining S-parameters and MNA

5.3.1 Non-Uniform Discrete Time Impulse Response

One of the disadvantages of the NUDTIR method is that we require keeping track of all previous voltages and currents, or alternatively power waves, for the entire length of the NUDTIR. This information is required to both perform the convolution of the incoming waves at the ports. In order to avoid having to store the information in multiple places, and potentially polluting the cache too much while accessing it, the choice was made to consider an S-Parameter port as a group two element, i.e a Thevenin Equivalent circuit, with current and voltage controlled voltage sources. This method is described in Appendix A. The main advantage of this is that as part of running the simulation the current will also be tracked for each port. The disadvantage of this method, is that the time taken to solve the full circuit's system will be longer, as the size of the matrix will increase by one for each port introduced.

In order to cater for the non-uniform nature of the NUDTIR method, it is necessary to either perform the simulation in steps of a common divisor with each tap, or to implement an interpolation method. For this project, a linear interpolation was employed for points which lie between the discrete simulation time points. This comes at a cost of needing to access two previous simulation time points to generate the value for a single tap of the NUDTIR.

5.3.2 Vector Fitting - Recursive Convolution

In a similar fashion, the Vector Fitting - Recursive Convolution [VFRC] approach also requires tracking of previous state, but it is less than that of a NUDTIR block. As such, a circular buffer could be used to store previous state for the S-parameter block. However for the sake of consistency of comparison between the speeds of the methods rather than the speeds of the matrix solver, the VFRC S-parameter block was implemented as a group two element, i.e a Thevenin Equivalent circuit, with current and voltage controlled voltage sources. The details of this approach is documented in Appendix B.

5.4 Interfacing other languages with C++

5.4.1 Interfacing Matlab with C++

Implementing vector fitting model extraction in native C++ code would be a task that was deemed too intensive for a non-performance critical perspective, after taking into account the variety of matrix manipulation techniques that would be required to make it function. As such, the alternative of interfacing with Matlab via its API was preferred. This allowed the program to use external scripts to generate the poles and residues of the model fitting the provided S-parameters. This allowed the infamous trade-off for development time vs run time to be made. In this case I feel the trade-off is justified. This allowed my circuit simulator to utilise the `Vectfit3` toolkit [1] [9] [10] by Gustavsen et al.

5.4.2 Interfacing Python with C++

In this project bindings for python were utilised to help speed up data analysis. The main library that was called was *matplotlib* to output useful graphs to be able to visualise the output of the simulator. The bindings used are a lightly modified version of the single header library `matplotlib-cpp` [13]. This was easy to use, and very effective.

5.5 Assessing the Design

5.5.1 Metrics

With software there are two main ways to measure how long a program takes to execute: wall time, and CPU time. Wall time is the measure of the time taken as if a human were to use a stop watch starting and stopping at the boundaries of execution. CPU time on the other hand, takes

into account the fact that on a modern computer running an operating system, the program will not necessarily get to run for 100% of the time between the start and end of the program. So we only measure the time where the program is actually executing on the CPU. Measuring CPU time in quite difficult, due to kernel calls related to memory management and other factors. As such, in this project the `<chronos>` library is used with `high_resolution_clock` to measure wall time.

Another interesting metric to note is memory usage. In particular, when memory usage exceeds the physical memory of the computer. This can lead to a cascade of page faults that force a program to grind to a halt. The most memory intensive aspect of this program will in general be the storage of the solution in memory. For double precision simulation this is approximately equivalent to $8MN$ where N is the number of time steps, M is the number of nodes (including currents), and 8 comes from the width of the double data-type.

In order to determine the most expensive parts of the simulation one can utilise a sampling profiler to statistically determine which instructions the program spends the most time executing. This can give indications as to what is the slowest part of the program. For a program like this, any memory intensive sections will be the main limiting factor. As such, the non-uniform convolution would be expected to be one of the slowest sections. This can be somewhat offset by truncation of lower valued taps, however this comes at the expense of accuracy

5.5.2 Test Bench Circuit

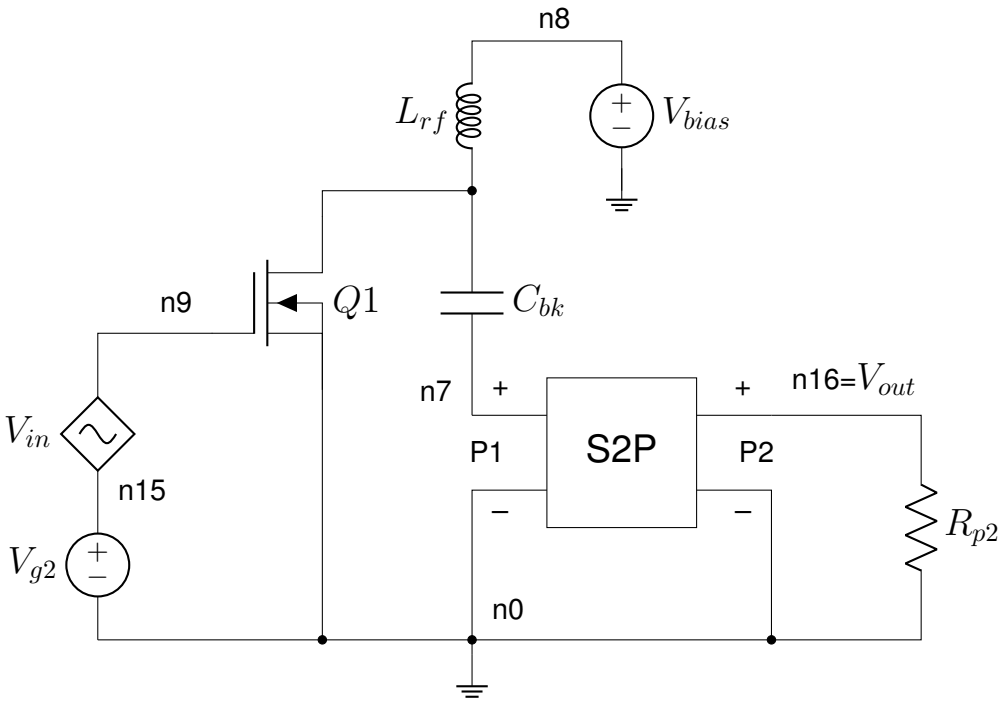


Figure 5.2: *High level class-F amplifier model*

Figure 5.2 shows a high level overview of the test bench circuit described in Figure 5.3 and Appendix C. This model is based on a class-F amplifier and was chosen as the final test bench circuit as it introduces several non-linear elements, and utilises an S-parameter block. The transistor $Q1$ is based on a modified version of the COBRA model for transistors described by Cojocaru and Brazil in 1997 [14].

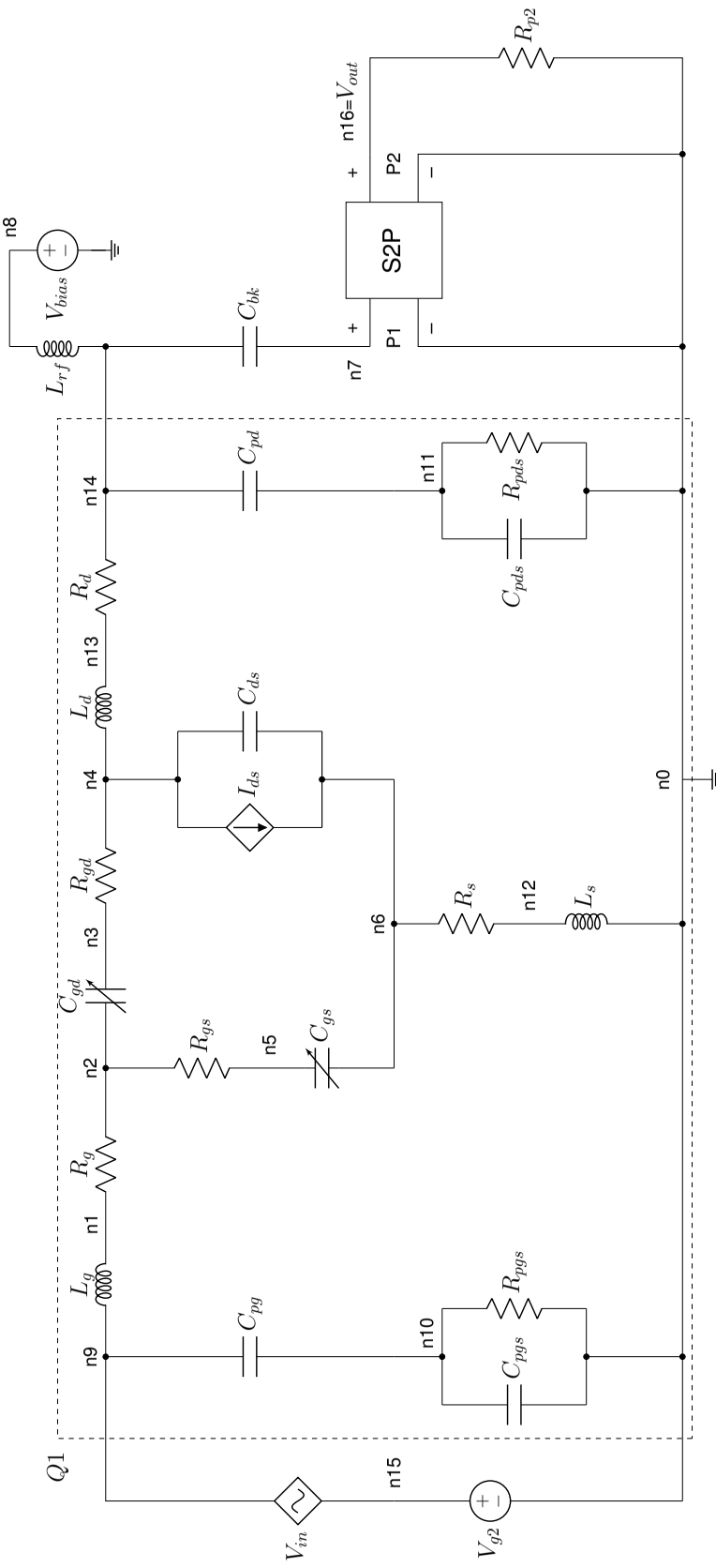


Figure 5.3: *Final class F amplifier model*

Results

6.1 Test bench harmonics

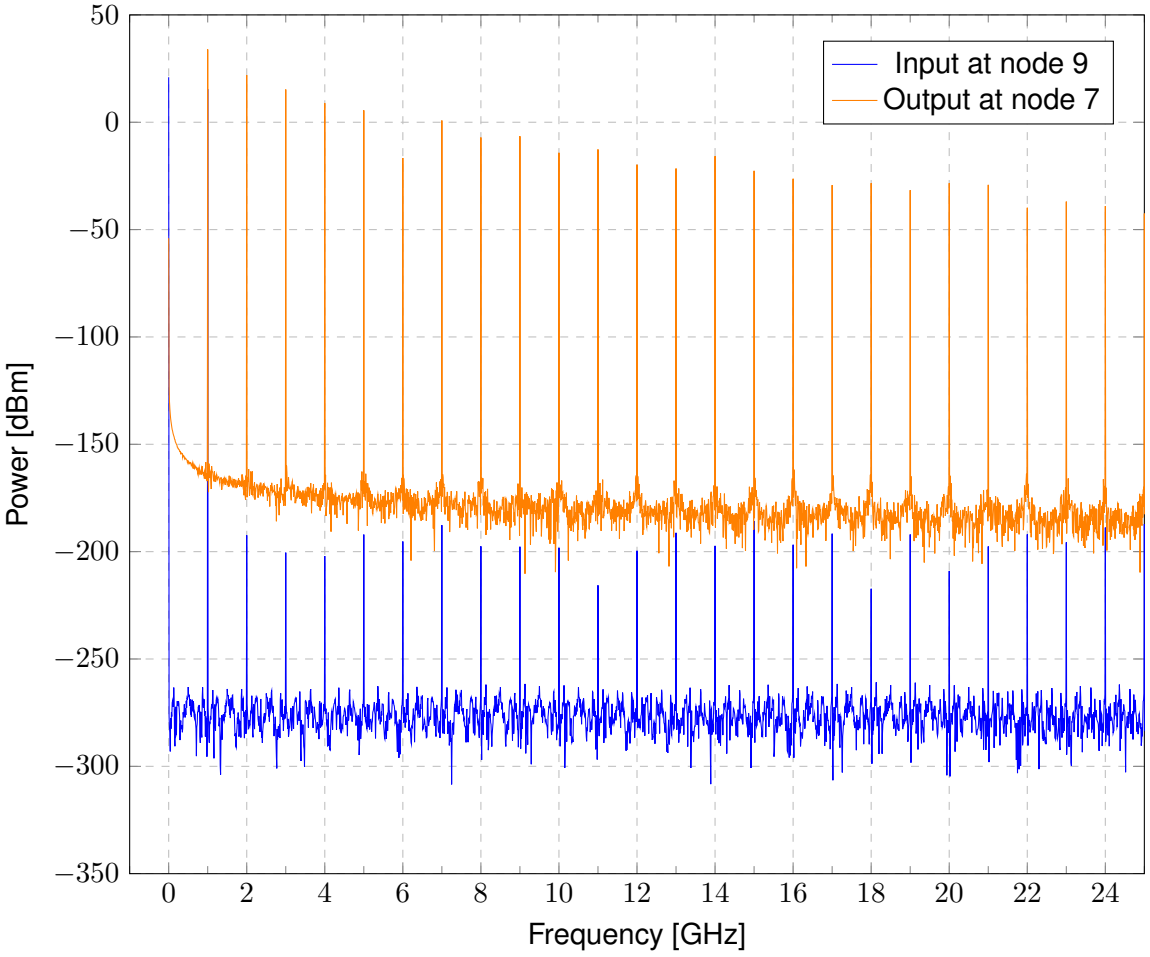


Figure 6.1: *FFT of the output from a 1 GHz signal while terminated with a* 50Ω *resistance, demonstrating the introduction of higher frequency harmonics*

We can see from Figure 6.1 that the class-F amplifier circuit outlined in Figure 5.3 introduces higher frequency harmonics to the signal passing through it. This can be an issue, as for the frequencies outside of 10 GHz the matching S-parameters are undefined. As such, the result is technically undefined. This effect was described by Brazil [4]. In this paper, Brazil describes how the S-parameters tend to behave relatively well outside of the range, however he does recommend having them extend out to the highest spectral components that has a significant

amount of power. To this effect, the higher frequencies introduced by the transistor model have been deemed too weak to have an effect, providing the extrapolation remains passive. This issue applies to both VFRC and NUDTIR.

6.2 Method Comparison

6.2.1 Direct Comparison

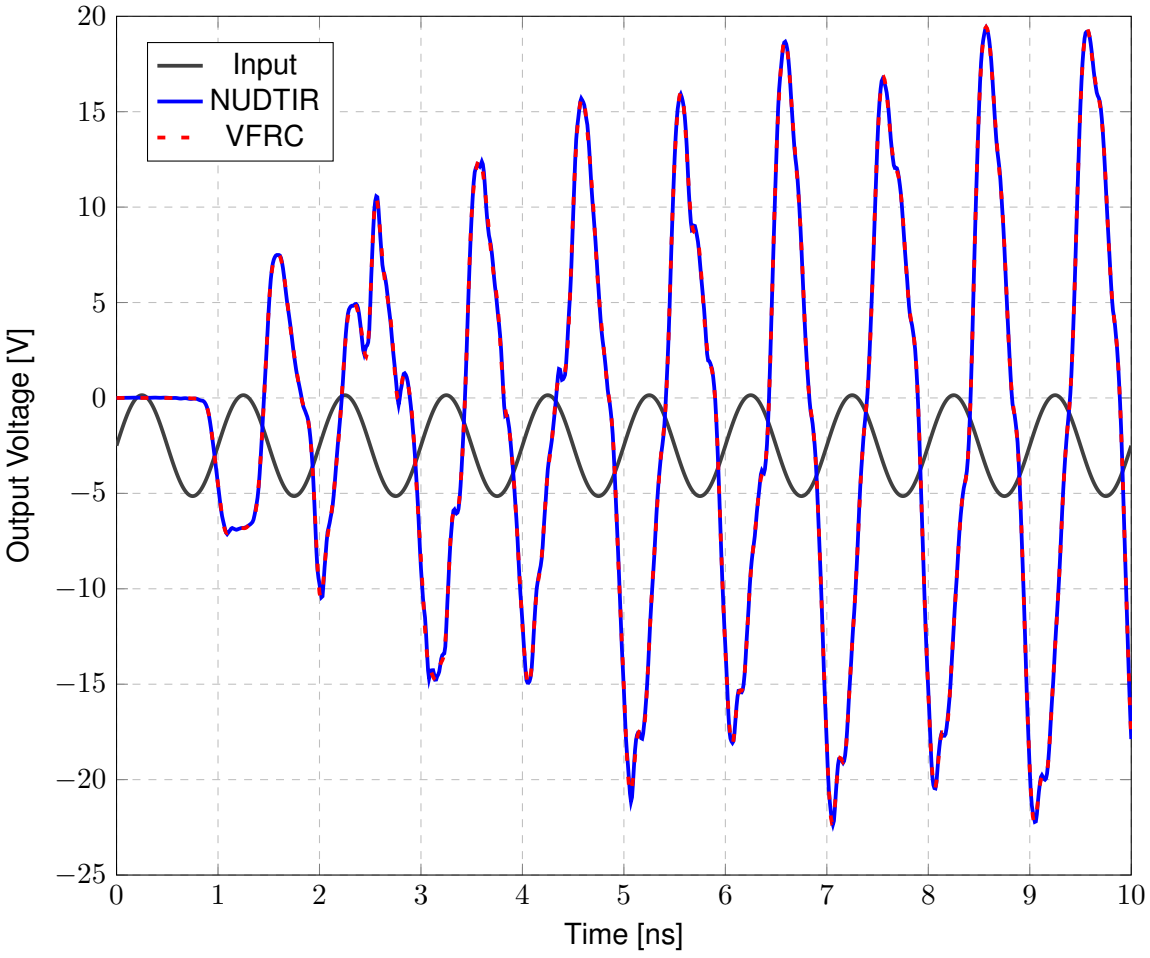


Figure 6.2: *The output from the simulation given the transistor model with a 1 GHz Sine Wave*

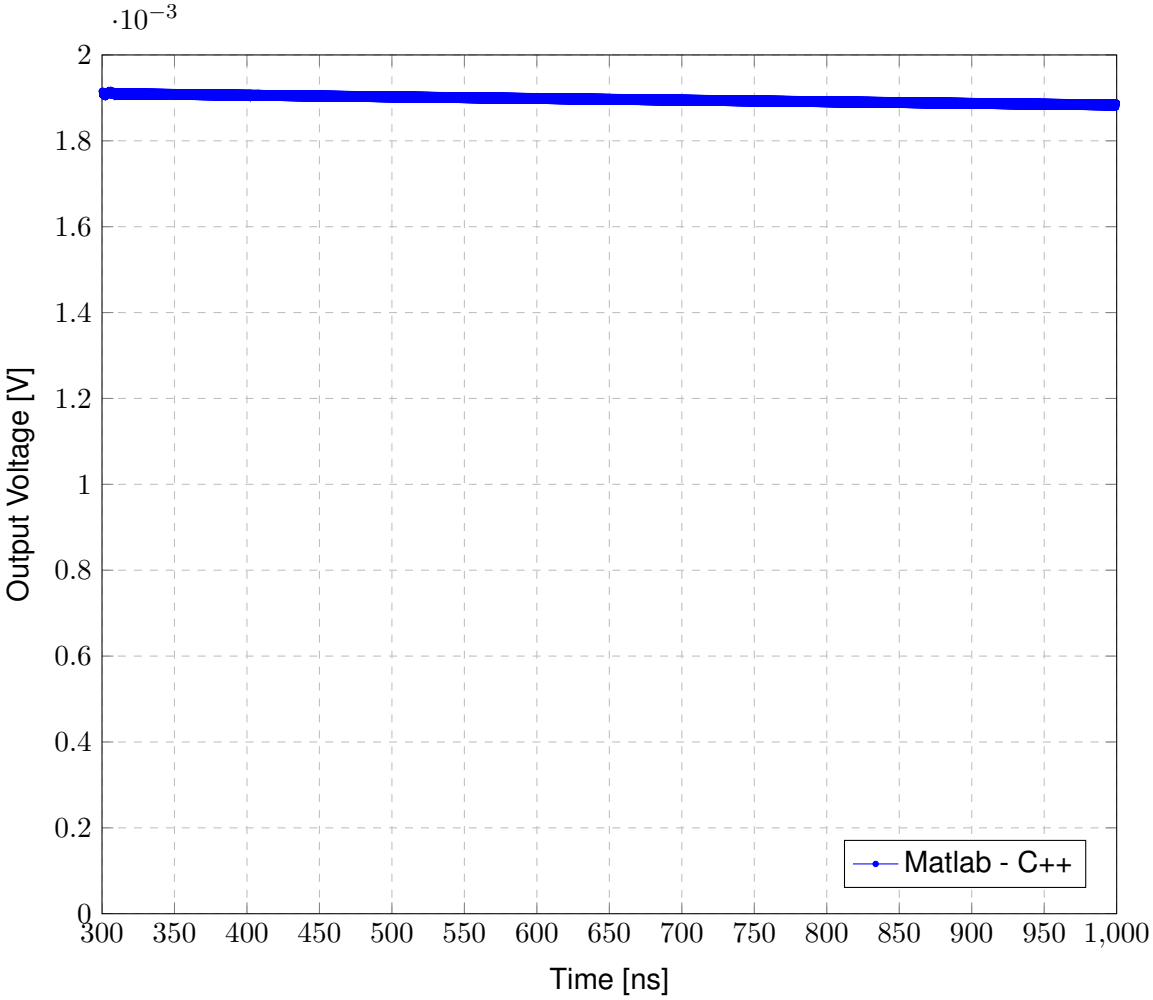


Figure 6.3: *The difference of voltage on port 1 of a S1P block in Matlab vs C++*

In Figure 6.2 we can see that the VFRC and NUDTIR methods of S-parameter simulations yield very similar results, even during the initial transient periods. This indicates that the models are likely performing in a way that is consistent with an accurate model. These results are also consistent with the original provided Matlab code as shown in Figure 6.3.

In terms of speed, the NUDTIR method is shown in Figure 6.4 to take approximately two to three times longer than the VFRC method. And the original provided prototype code for the NUDTIR method written in Matlab takes approximately twenty times longer to execute. These data are based on an AMD Ryzen 7 3700x running at stock settings with 32GB of 3200MHz DDR5 RAM.

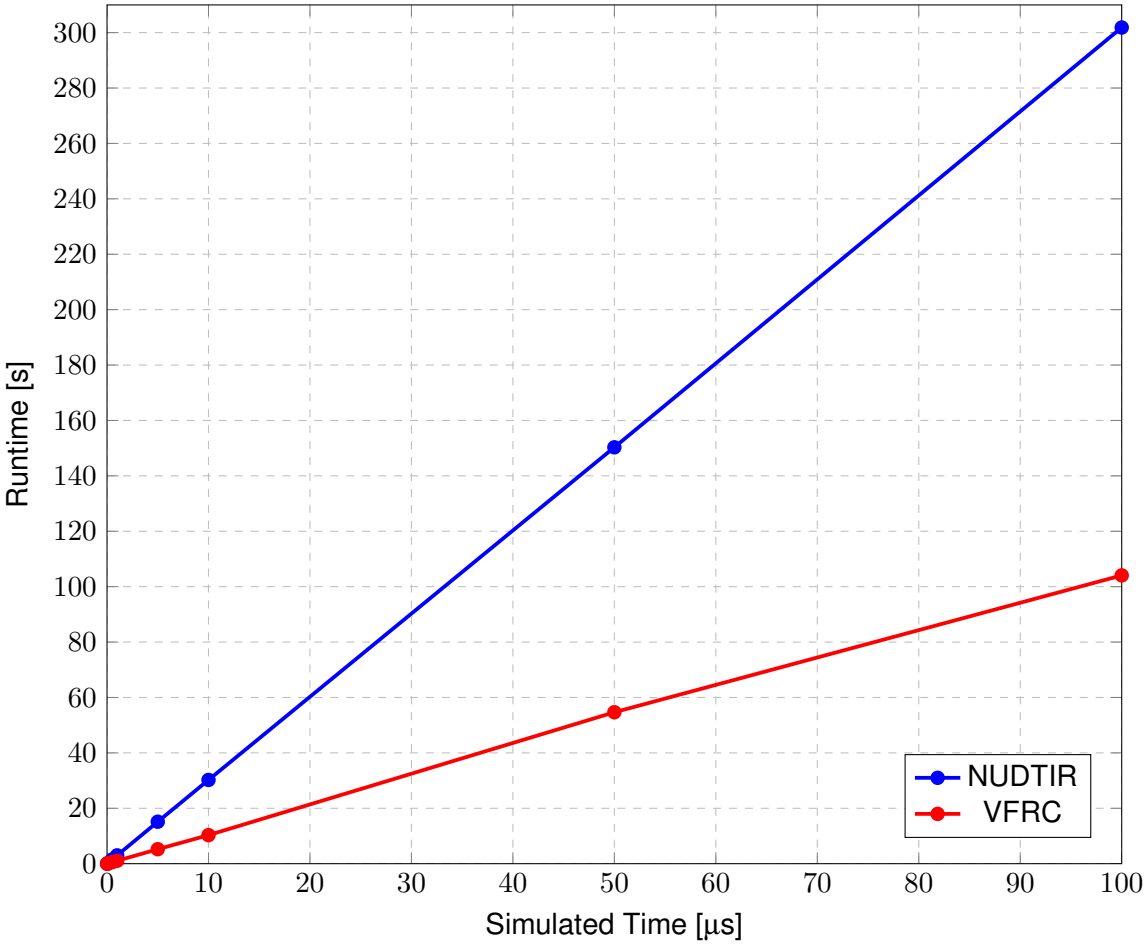


Figure 6.4: *Plot of simulated time vs elapsed time for the NUDTIR method and the VFRC method*

6.2.2 Code Hotspots Found

By utilising Microsoft Visual Studio's Diagnostic Tool's CPU sampling profiler, it was found that the largest bottleneck of the NUDTIR method is line 7 of Listing 6.1. The program spends over half of the whole simulation fetching values from memory and calculating the interpolated `aWaveConvValue` values. This is a result of needing to access a lot of fragmented data in quick succession that does not already exist in the cache, and therefore we start thrashing the cache. This result is confirmed when cross-checked with AMD uProf. This result contrasts the results of the VFRC method, which showed the most time spent being performing the LU decomposition to solve the system.

```
 1 T V_p(size_t p, const Matrix<T> & solutionMatrix,
 2       const size_t n, T simulationTimestep, size_t sizeG_A) const {
 3    // V_p = beta * sum of ports ( history of port )
 4    T toRet = 0;
 5    for (size_t c = 0; c < port.size(); c++) {
 6       for (size_t k = 1; k < s.length(p, c); k++) {
 7 52%>> toRet += aWaveConvValue(c, solutionMatrix, n, s.time(p, c, k),
 8                               simulationTimestep, sizeG_A) *
 9                 s.data(p, c, k);
10       }
11    }
12    return port[p].beta * toRet;
13 }
```

Listing 6.1: *Hotspot in NUDTIR code*

It can be seen in the generated assembly that there is a strong data dependence, which means that the whole processing pipeline must stall until the result is calculated. This means that uProf is inaccurate in its determination of which exact assembly instruction is the largest bottleneck.

6.2.3 Non-Uniform Discrete Time Convolution Speed-ups

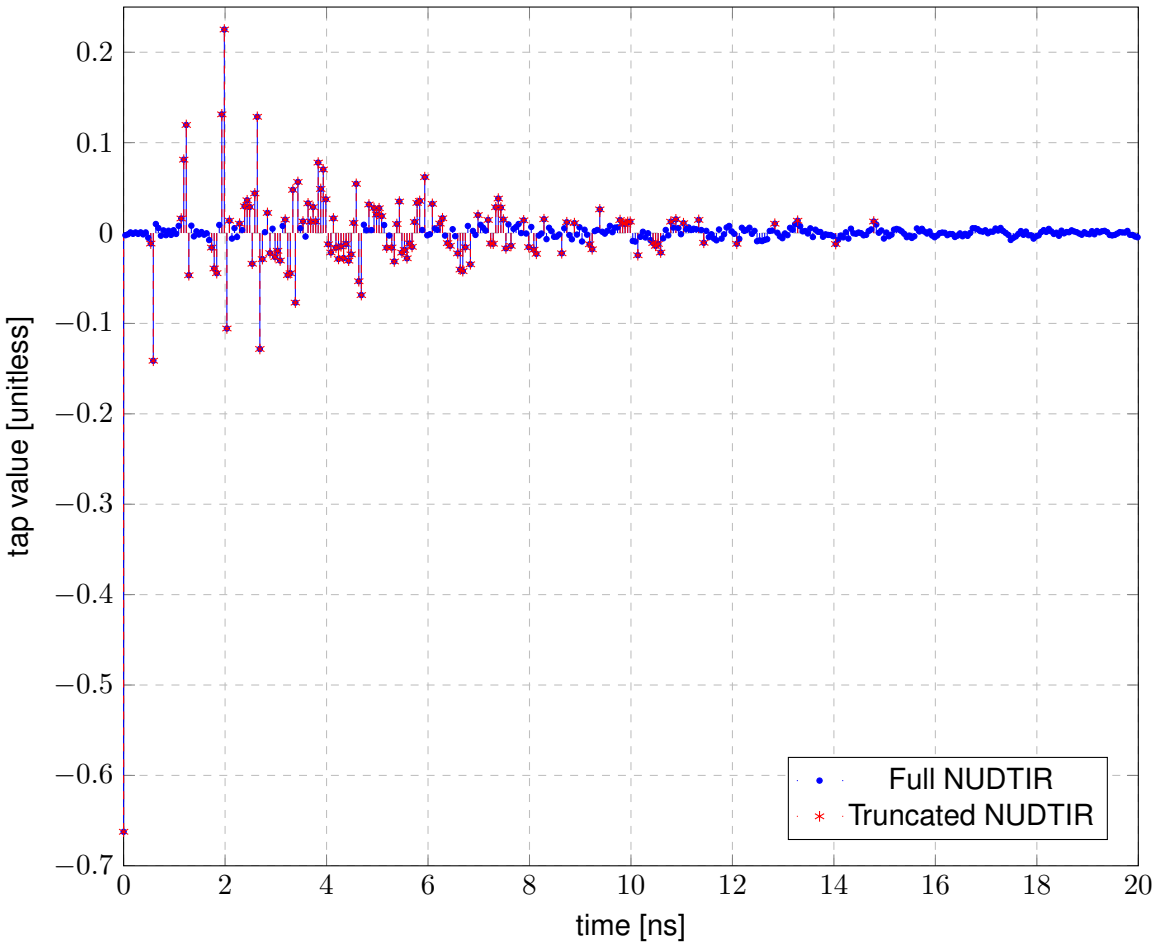


Figure 6.5: *Comparison of the NUDTIR with and without truncation below 1%, where the Truncated NUDTIR is a subset of the full NUDTIR*

Figure 6.5 shows the beginning of a NUDTIR for the impedance matching block $S_{2,2}$. Of the 2000 taps generated for the NUDTIR only 127 of the taps were above on percent of the maximum value. Pruning these values from the NUDTIR response helps to decrease random memory accesses and reduce pollution of the cache, as well as requiring less multiply and accumulates.

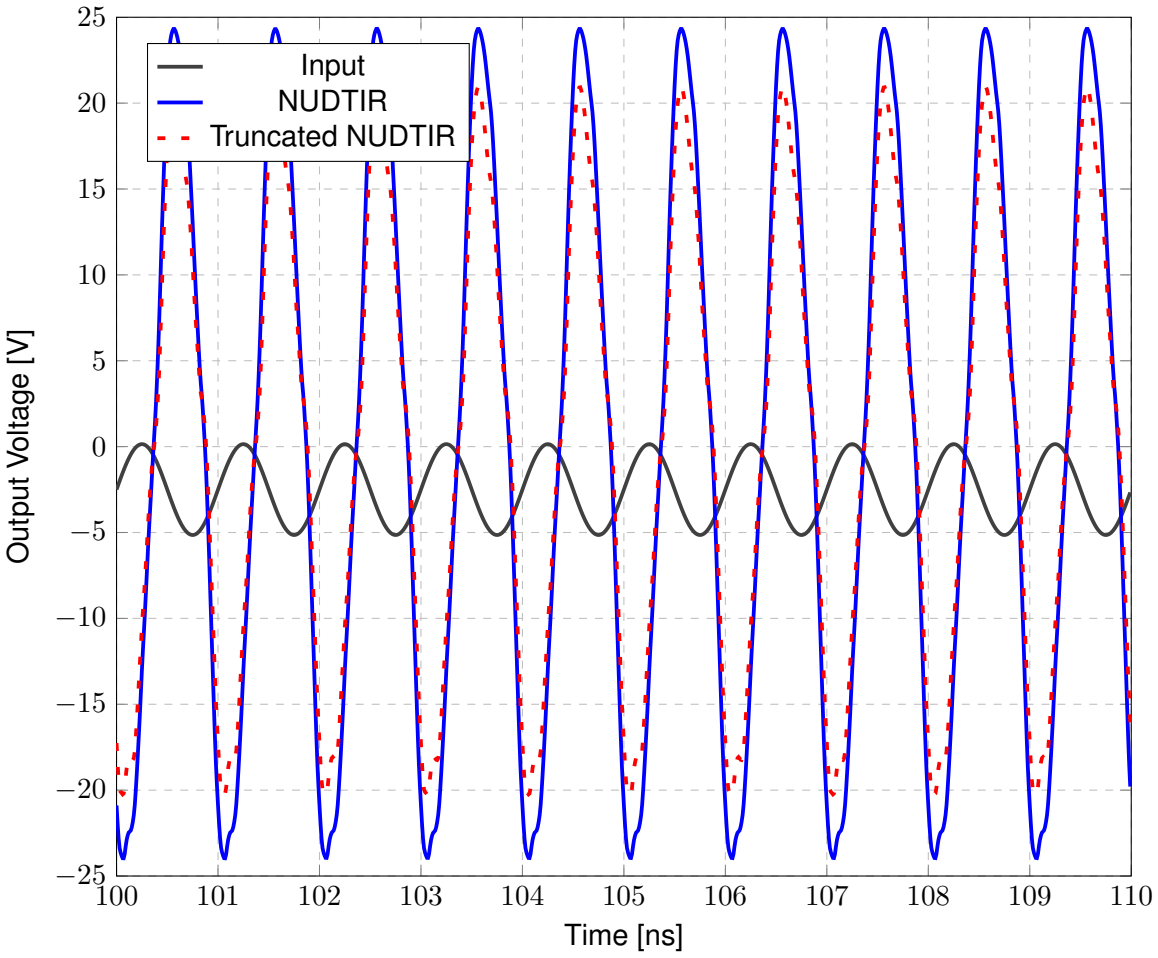


Figure 6.6: *Comparison of the output of the simulation with and without truncation below 1%*

The results of a simulation after applying this pruning criterion is shown in Figure 6.6. We can see from these results that the lower frequency components are preserved rather well, as the dominant shapes are still visible, however some of the higher frequency components show discrepancies. This is because small changes of the higher frequency components can have profound affects on the output. One must weigh up if the speed ups are beneficial over the accuracy losses to be able to justify pruning in this way.

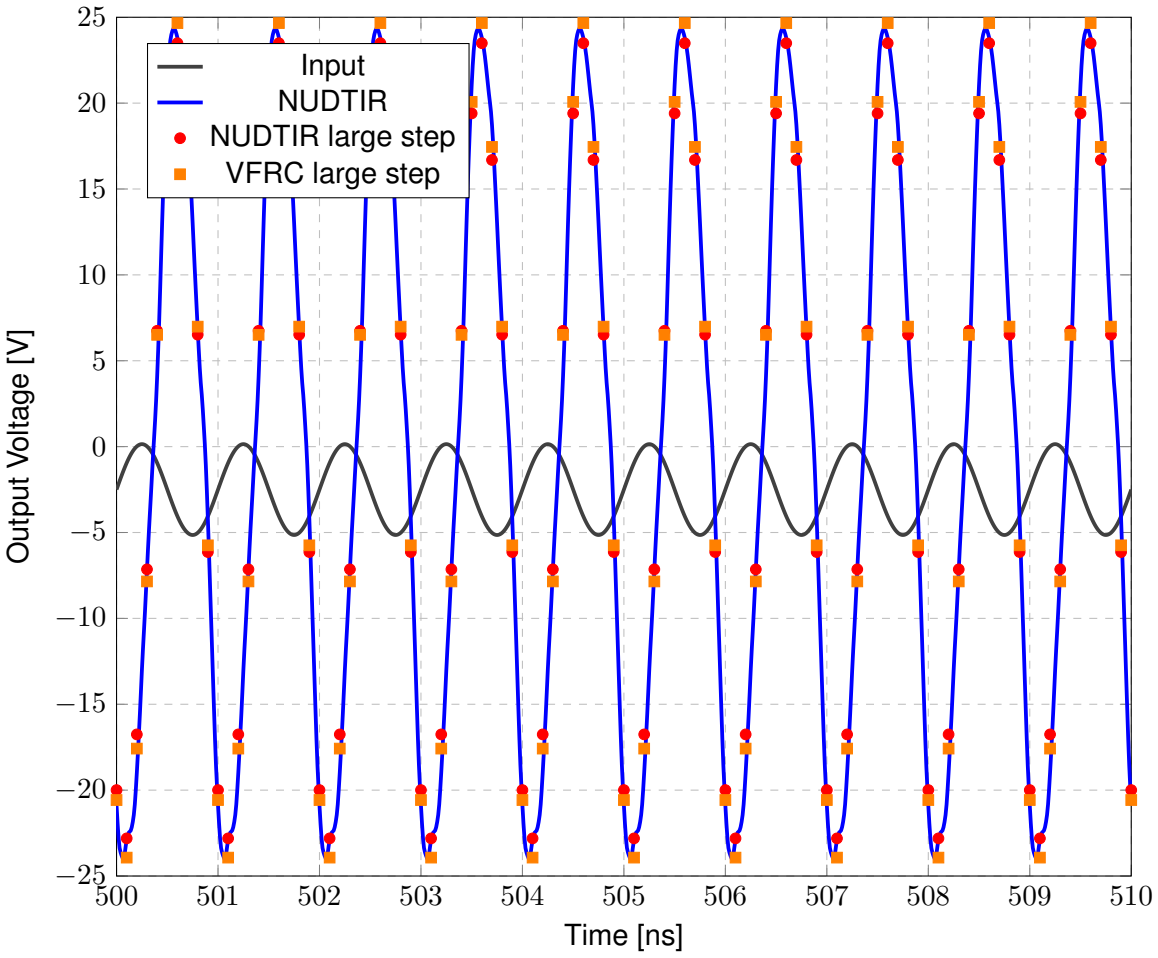


Figure 6.7: *NUDTIR step size comparison*

Generally speaking, it seems that step size can be larger with NUDTIR method than VF method. This is the best shot of speeding up the simulation without sacrificing as much of the accuracy. Note that the VFRC result is mostly giving more extreme values as the step size increases than the NUDTIR approach. This is inherent to the approximations taken via the VFRC method, which is a numerical approximation of integration. There is only so much of an advantage this can provide however, as the input signal's characteristics will dictate what step size we can use before information is lost.

6.3 Effects of timestep of NUDTIR

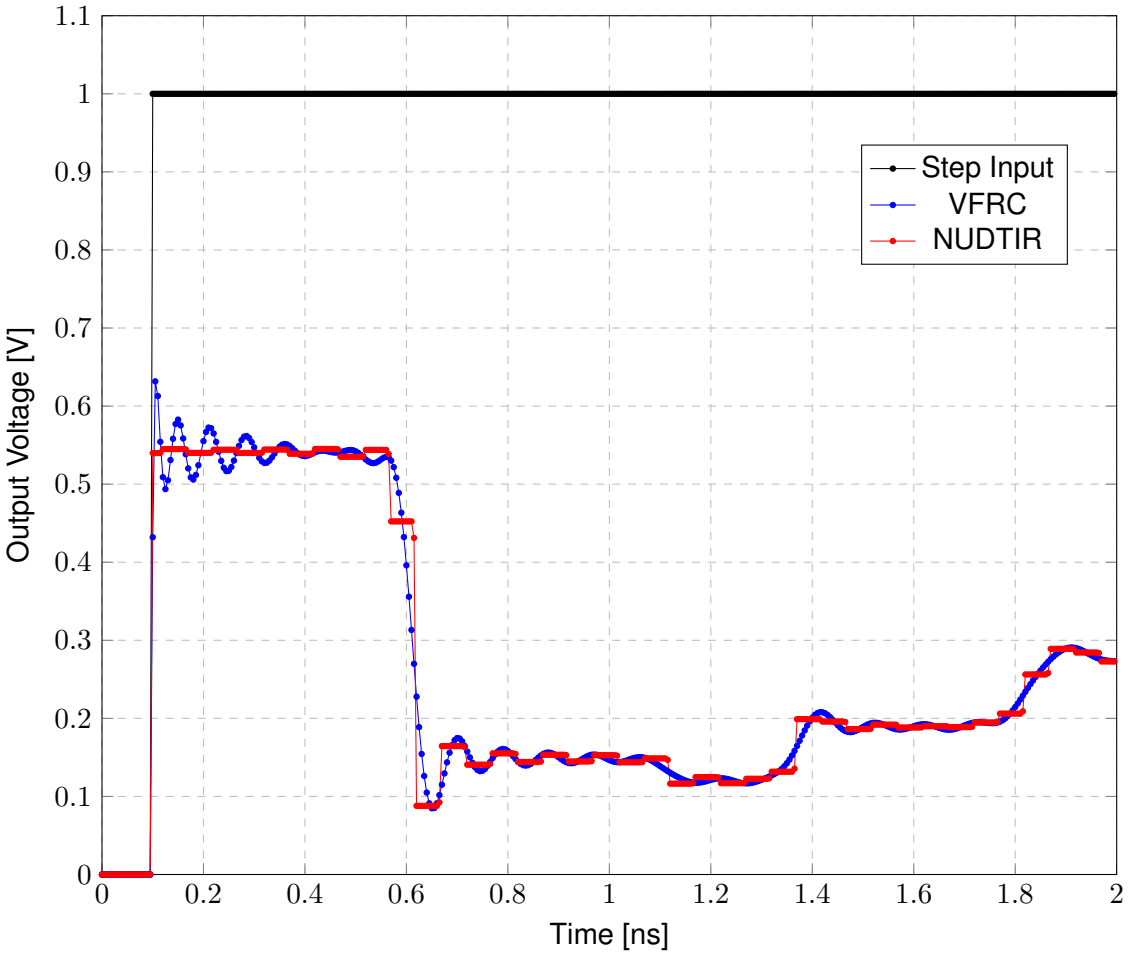


Figure 6.8: *Demonstration of limited step-size due to NUDTIR step size*

One effect that was noticed during the development of this simulator, is the step-like behaviour that is induced due to the discrete nature of the NUDTIR. This behaviour is demonstrated in Figure 6.8. It is shown that the simulation only updates changes at discrete intervals. In practice, the input signal is constantly changing, so there is merit the extra resolution, but this response may warrant further investigation

This step response is similar to the output of a sample and hold ADC, because in essence the NUDTIR is doing exactly that for a step input. As the step never changes in value after rising, the taps of the NUDTIR only change value when it progresses into the *high* region of the step response. The difference between the VFRC response and the NUDTIR response can be explained by the fact that a step response has an infinite maximum frequency, which is undefined outside of the frequency response outlined by the S-parameters. As such it is not

possible to know which output is the correct in an accuracy sense without knowing the original circuit the S-parameters are derived from.

6.4 Vector Fitting Passivity

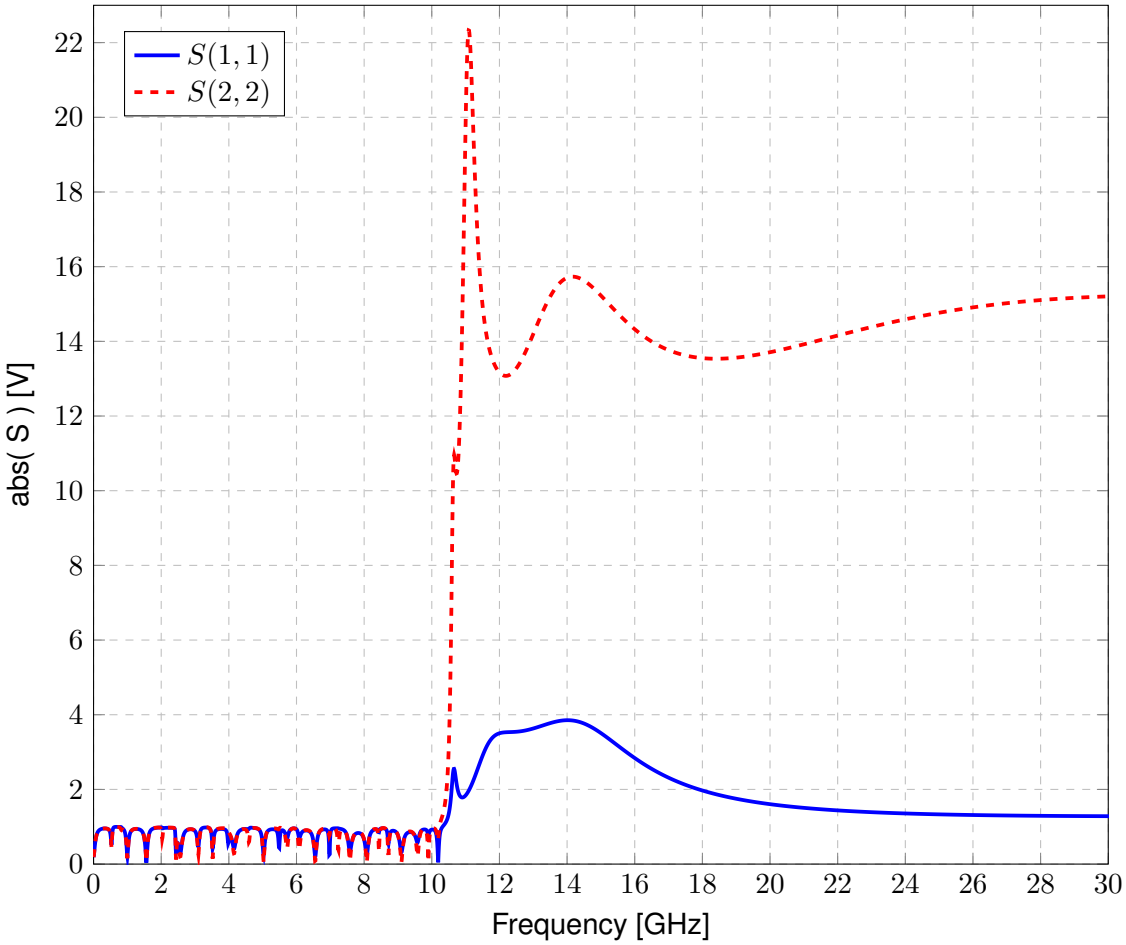


Figure 6.9: *Extrapolation past the original 10 GHz with 100 poles causing non-passivity*

One must be careful not to try fit too many poles to the response. If one does this, the value of the residues can become very large in an attempt to fit the response better. This is predominantly an issue with VFRC, as the fit can become non-passive outside of the fitting region. An example of this is shown in Figure 6.9, where an absolute value greater than 1 indicated a non-passive response for that frequency. If the simulator invokes higher frequency components, the response can cause a positive feedback loop which will make the response diverge.

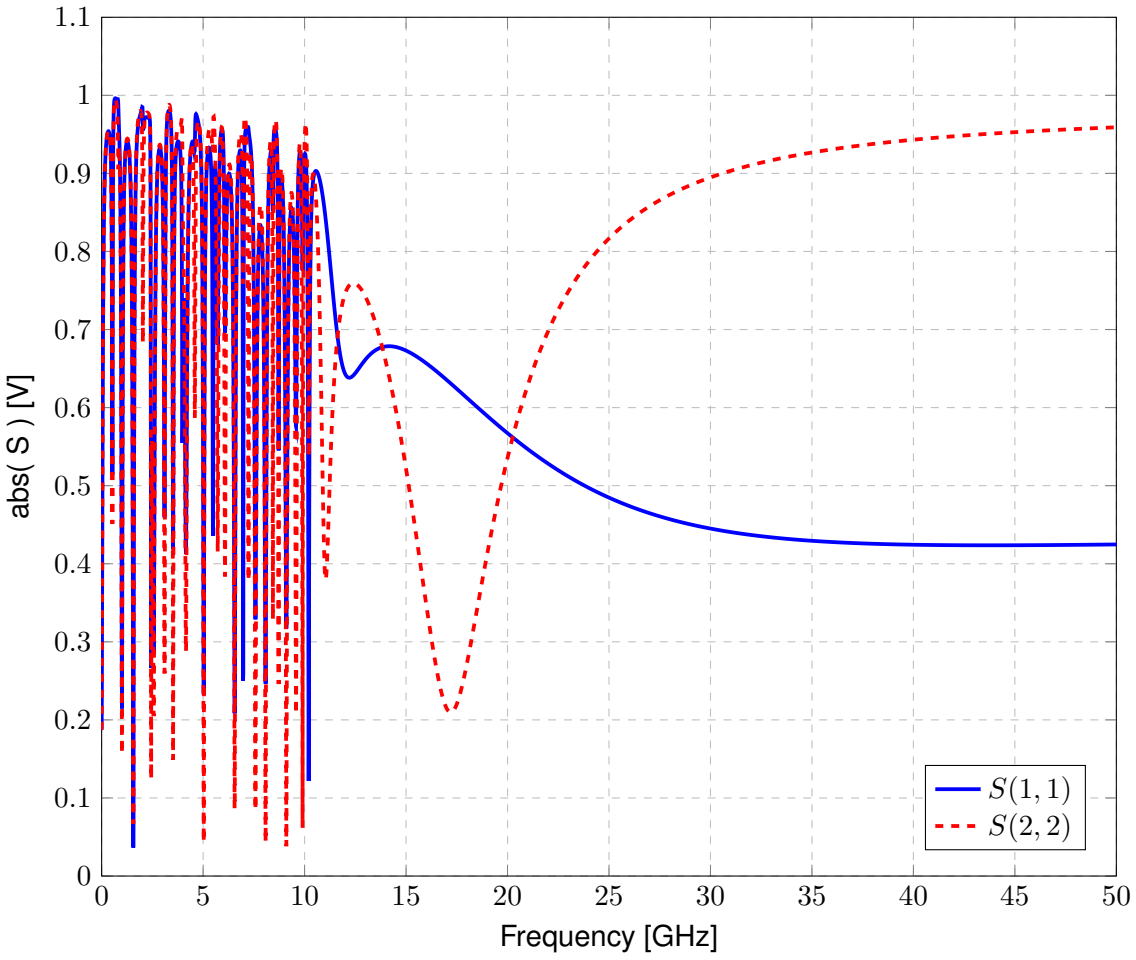


Figure 6.10: *A passive extrapolation past 10 GHz with 78 poles*

In order to combat this in the final implementation of vector fitting, the simulator attempts to fit with 100 poles by default, and reduces the order until there is no non-passive response for a large extension of the frequency range. This resulted in a fit with 78 poles as shown in Figure 6.10. This allows a high accuracy fit, without allowing the result to become non-passive.

6.5 5G Signal Simulation

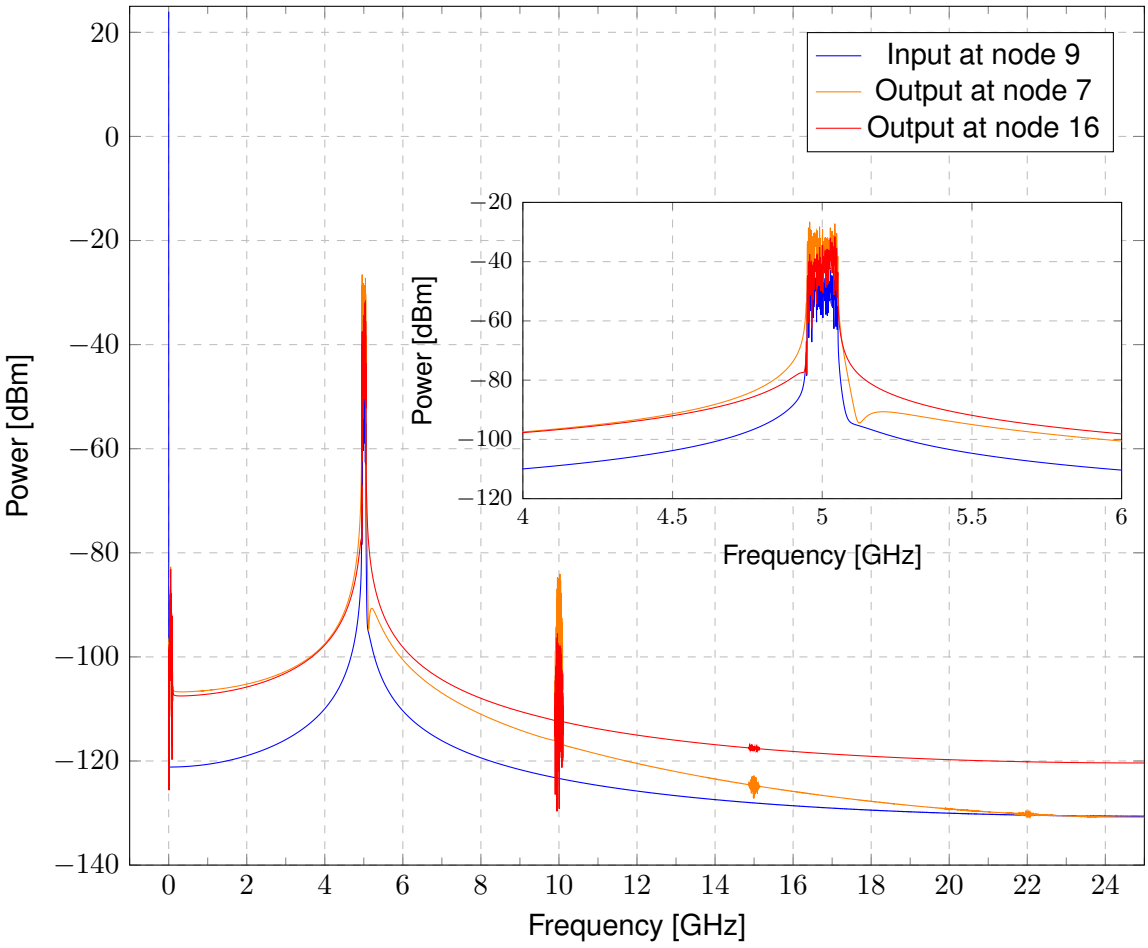


Figure 6.11: *The frequency spectrum of a 5G signal having gone through the full transistor/S-parameter model*

Figure 6.11 showcases the resultant harmonics introduced for a 5G signal passed through the test bench circuit described in Appendix 5.3. Here it can be seen that there is some of the 5G spectrum shown in Figure 6.12 replicated at 10 GHz and 15 GHz. What we can see however, is that the power of these signals are significantly less than that of the original signal around 5GHz. The power diminishes very quickly as the harmonics tend towards higher frequencies.

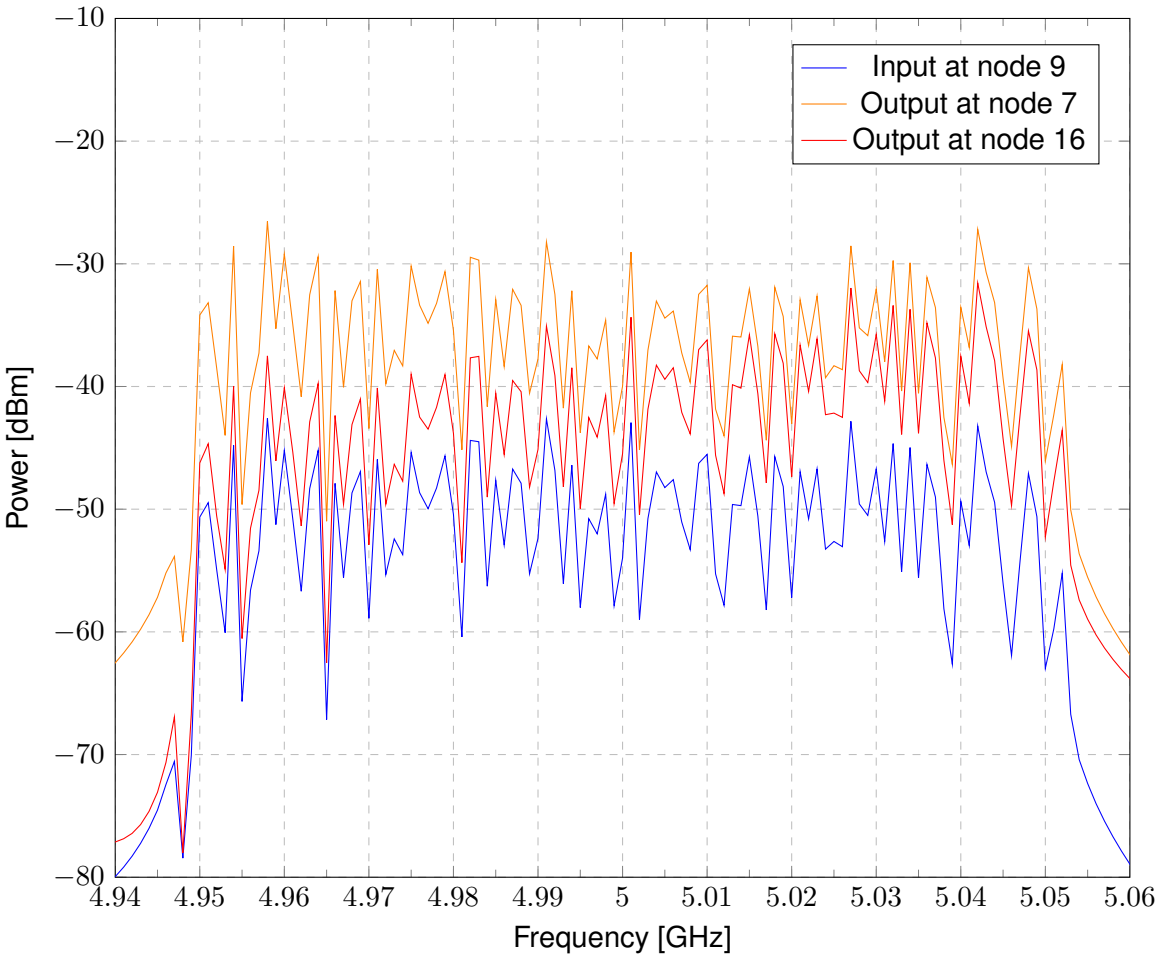


Figure 6.12: *The frequency spectrum of a 5G signal having gone through the full transistor/S-param model, zoomed in around 5GHz*

The zoomed in Figure 6.12 shows how the overall circuit is acting as an amplifier for the signal present at the transistor's input. The output at the second port is less than that of the first port, as the S-parameter block is acting as a low pass filter

Conclusions

7.1 Superior method

It is apparent from the benchmarks carried out, that VFRC is a faster method than NUDTIR for the same simulation time step. This is due to the fact that in general VFRC requires less random access than the NUDTIR equivalent. This is consistent with what would be expected of high speed sequential simulation.

There are two options for speeding up the NUDTIR method. The first is by pruning low magnitude NUDTIR taps, and the second is by increasing the step size of the simulation. It was found that a one percent threshold was required to gain speed to rival the VFRC approach. This lead to a noticeable decrease in accuracy.

The latter method was found to be more effective, so long as the input signal was not of a low enough frequency to allow the step size to be increased. This option lead to substantially faster simulation times as there were less steps to reach the final time, even if each step is more expensive. It is important to note however, that this still introduces inaccuracies.

7.2 Including Measured Frequency Domain Data Within Time Domain Simulations

A key takeaway from this project is that the introduction of non-linear elements introduces higher frequency harmonics that can affect the outcome of the simulation greatly if care is not taken. This is because these harmonics can induce energy into frequencies outside of the defined range of the S-parameters for the block. Ideally one would not let this happen by having a higher frequency set of S-parameters, but the non-linear elements will almost always introduce harmonics that go up to infinity.

7.3 Future Work

Potential explorations of this topic could be attempts to optimise the data access of the past a-wave values for the NUDTIR method. An adaptive step size could also see substantial boosts to speed. The main bottleneck of the VFRC method is the LU-solver. As this is self-developed, substantial speed-ups could be seen by switching to a more mature matrix library. This would also allow an implementation of VFRC without the need to use Matlab scripts.

Bibliography

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

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

[3] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. USA: Prentice Hall Press, 3rd ed., 2009.

[4] T. J. Brazil, "Accurate and efficient incorporation of frequency-domain data within linear and non-linear time-domain transient simulation," in *IEEE MTT-S International Microwave Symposium Digest, 2005.*, pp. 797–800, 2005.

[5] J. Schutt-Ainé, "Circuit synthesis of blackbox macromodels from s-parameter representation," in *2018 IEEE 22nd Workshop on Signal and Power Integrity (SPI)*, pp. 1–3, 2018.

[6] F. Najm, *Circuit simulation*. Hoboken, N.J: Wiley, 2010.

[7] R. Ludwig, *RF circuit design : theory and applications*. Upper Saddle River, NJ: Prentice-Hall, 2009.

[8] J. E. Schutt-Aine, P. Goh, Y. Mekonnen, J. Tan, F. Al-Hawari, P. Liu, and W. Dai, "Comparative study of convolution and order reduction techniques for blackbox macromodeling using scattering parameters," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 1, no. 10, pp. 1642–1650, 2011.

[9] B. Gustavsen, "Improving the pole relocating properties of vector fitting," *IEEE Transactions on Power Delivery*, vol. 21, no. 3, pp. 1587–1592, 2006.

[10] D. Deschrijver, M. Mrozowski, T. Dhaene, and D. De Zutter, "Macromodeling of multiport systems using a fast implementation of the vector fitting method," *IEEE Microwave and Wireless Components Letters*, vol. 18, no. 6, pp. 383–385, 2008.

[11] R. Fabian, *Data-oriented design: software engineering for limited resources and short schedules*. Richard Fabian, 2018. OCLC: 1100990710, ISBN: 9781916478701, URL: https://www.dataorienteddesign.com/dodbook/.

[12] D. Sloggett, "CircuitSimulator_MAI_JUK1." `https://github.com/dslogget/CircuitSimulator_MAI_JUK1/`, 2021.

[13] B. Evers, A. Dewar, A. Sakai, G. Jacquenot, F. Eich, J. Gacon, T. Keller, Chachay, C. Harasyn, M. Zins, W. Liu, K. Mohta, T. Berset, R. Aldridge, tomotakaeru, Y. Mihira, M. Ferrari, R. Taylor-Davies, M. Davi, K. Tanihara, Florian, J. Brinsfield, Q. Tùng, Belre-Yucho, M. Gianolio, NancyLi1013, Y. Feng, JBPennington, kesha787898, G. Lenz, T. Hisada, P. Fors, R. Luies, S. Sanjeet, W. Lee, thesmallcreeper, O. Croquette, Christos, H. HUANG, V. Magnago, adiba, O. Kermorgant, P. Jurczak, M. Mosley, F. Furlan, André, A. Tahmasbi, A. Patel, T. Kneiphof, B. Phung, B. BALP, A. Schuh, Alex, J. Nazario, and A. Gentilini, "matplotlib-cpp." `https://github.com/lava/matplotlib-cpp/`, 2020. commit:70d508fcb7febc66535ba923eac1b1a4e571e4d1.

[14] V. Cojocaru and T. Brazil, "A scalable general-purpose model for microwave fets including dc/ac dispersion effects," *IEEE Transactions on Microwave Theory and Techniques*, vol. 45, no. 12, pp. 2248–2255, 1997.

Non-Uniform Discrete Time Impulse Response Stamp

Starting with the definitions for power waves a and b.

$$u_{\sigma,n} = \text{voltage across at port } \sigma \text{ at time } n \tag{A.1}$$

$$i_{\sigma,n} = \text{current entering port } \sigma \text{ at time } n \tag{A.2}$$

$$a_{\sigma,n} = \text{incident power wave at port } \sigma \text{ at time } n \tag{A.3}$$

$$b_{\sigma,n} = \text{reflected power wave at port } \sigma \text{ at time } n \tag{A.4}$$

$$s_{\sigma,c,k} = \text{NUDTIR value for port } \sigma \text{ from port } c \text{ for time } k \tag{A.5}$$

$$z_{ref} = \text{Reference impedance for the measured S-parameter data} \tag{A.6}$$

$$a_{\sigma,n} = \frac{u_{\sigma,n} + i_{\sigma,n} z_{ref}}{2\sqrt{z_{ref}}} \tag{A.7}$$

$$b_{\sigma,n} = \frac{u_{\sigma,n} - i_{\sigma,n} z_{ref}}{2\sqrt{z_{ref}}} \tag{A.8}$$

$$b_{\sigma,n} = \sum_{c \in P} (a_c * s_{\sigma,c})_n \tag{A.9}$$

$$\text{let } h_{\sigma,n} = \sum_{c \in P} \sum_{k=1}^{n} (u_{\sigma,n-k} + i_{\sigma,n-k} z_{ref}) s_{\sigma,c,k} \tag{A.10}$$

$$b_{\sigma,n} = \sum_{c \in P} a_{c,n} s_{\sigma,c,0} + \frac{h_{\sigma,n}}{2\sqrt{z_r ef}} \tag{A.11}$$

$$b_{\sigma,n} = a_{\sigma,n} s_{\sigma,\sigma,0} + \sum_{c \neq \sigma} a_{c,n} s_{\sigma,c,0} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{A.12}$$

$$b_{\sigma,n} - a_{\sigma,n} s_{\sigma,\sigma,0} = \sum_{c \neq \sigma} a_{c,n} s_{\sigma,c,0} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{A.13}$$

$$\frac{(1 - s_{\sigma,\sigma,0}) u_{\sigma,n} - z_{ref}(1 + s_{\sigma,\sigma,0}) i_{\sigma,n}}{2\sqrt{z_{ref}}} = \sum_{c \neq \sigma} a_{c,n} s_{\sigma,c,0} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{A.14}$$

$$(1 - s_{\sigma,\sigma,0}) u_{\sigma,n} - z_{ref}(1 + s_{\sigma,\sigma,0}) i_{\sigma,n} = 2\sqrt{z_{ref}} \sum_{c \neq \sigma} a_{c,n} s_{\sigma,c,0} + h_{\sigma,n} \tag{A.15}$$

$$(1 - s_{\sigma,\sigma,0}) u_{\sigma,n} = z_{ref}(1 + s_{\sigma,\sigma,0}) i_{\sigma,n} + 2\sqrt{z_{ref}} \sum_{c \neq \sigma} a_{c,n} s_{\sigma,c,0} + h_{\sigma,n} \tag{A.16}$$

$$u_{\sigma,n} = z_{ref} \frac{1 + s_{\sigma,\sigma,0}}{1 - s_{\sigma,\sigma,0}} i_{\sigma,n} + \frac{2\sqrt{z_{ref}}}{1 - s_{\sigma,\sigma,0}} \sum_{c \neq \sigma} a_{c,n} s_{\sigma,c,0} + \frac{1}{1 - s_{\sigma,\sigma,0}} h_{\sigma,n} \tag{A.17}$$

$$u_{\sigma,n} = z_{ref} \frac{1 + s_{\sigma,\sigma,0}}{1 - s_{\sigma,\sigma,0}} i_{\sigma,n} + \sum_{c \neq \sigma} \frac{s_{\sigma,c,0}}{1 - s_{\sigma,\sigma,0}} (u_{c,n} + i_{c,n} z_{ref}) + \frac{1}{1 - s_{\sigma,\sigma,0}} h_{\sigma,n} \tag{A.18}$$

$$\beta_\sigma = \frac{1}{1 - s_{\sigma,\sigma,0}} \tag{A.19}$$

$$V_\sigma = \beta_\sigma \sum_{c \in P} \sum_{k=1}^{n} (u_{c,n-k} + i_{c,n-k} z_{ref}) s_{\sigma,c,k} \tag{A.20}$$

$$\alpha_{\sigma,c} = \beta_\sigma s_{\sigma,c,0} \tag{A.21}$$

$$R_\sigma = \beta_\sigma z_{ref}(1 + s_{\sigma,\sigma,0}) \tag{A.22}$$

$$u_{\sigma,n} = R_\sigma i_{\sigma,n} + \sum_{c \in P, c \neq \sigma} \alpha_{\sigma,c} (u_{c,n} + i_{c,n} z_{ref}) + V_\sigma \tag{A.23}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & -\alpha_{12} & \alpha_{12} & -R_1 & -z_{ref}\alpha_{12} \\ -\alpha_{21} & \alpha_{21} & 1 & -1 & -z_{ref}\alpha_{21} & -R_2 \end{bmatrix} \begin{bmatrix} v_{1+} \\ v_{1-} \\ v_{2+} \\ v_{2-} \\ i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ V_1 \\ V_2 \end{bmatrix} \tag{A.24}$$

2 port S-parameter equivalent circuit with terms defined above

$$b_{\sigma,n} = \sum_{c \in P} (a_c * s_{\sigma,c})_n \tag{A.25}$$

$$b_\sigma = \sum_{k=0}^{n} s_{\sigma,\sigma,k} a_\sigma + \sum_{c \neq \sigma} \sum_{k=0}^{n} s_{\sigma,c,k} a_c \tag{A.26}$$

$$b_\sigma - \sum_{k=0}^{n} s_{\sigma,\sigma,k} a_\sigma = \sum_{c \neq \sigma} \sum_{k=0}^{n} s_{\sigma,c,k} a_c \tag{A.27}$$

$$(1 - \sum_{k=0}^{n} s_{\sigma,\sigma,k}) u_\sigma = z_{ref}(1 + \sum_{k=0}^{n} s_{\sigma,\sigma,k}) i_\sigma + \sum_{c \neq \sigma} \sum_{k=0}^{n} s_{\sigma,c,k}(u_\sigma + i_\sigma z_{ref}) \tag{A.28}$$

$$u_\sigma = z_{ref} \frac{1 + \sum_{k=0}^{n} s_{\sigma,\sigma,k}}{1 - \sum_{k=0}^{n} s_{\sigma,\sigma,k}} i_\sigma + \sum_{c \neq \sigma} \frac{\sum_{k=0}^{n} s_{\sigma,c,k}}{1 - \sum_{k=0}^{n} s_{\sigma,\sigma,k}} (u_c + i_c z_{ref}) \tag{A.29}$$

$$R_\sigma = z_{ref} \frac{1 + \sum_{k=0}^{n} s_{\sigma,\sigma,k}}{1 - \sum_{k=0}^{n} s_{\sigma,\sigma,k}} \tag{A.30}$$

$$\alpha_{\sigma,c} = \frac{\sum_{k=0}^{n} s_{\sigma,c,k}}{1 - \sum_{k=0}^{n} s_{\sigma,\sigma,k}} \tag{A.31}$$

$$V_\sigma = 0 \tag{A.32}$$

$$u_\sigma = R_\sigma i_\sigma + \sum_{c \neq \sigma} \alpha_{\sigma,c}(u_c + i_c z_{ref}) \tag{A.33}$$

Vector Fitting - Recursive Convolution Stamp

$$u_{\sigma,n} = \text{voltage across at port } \sigma \text{ at time } n \tag{B.1}$$

$$i_{\sigma,n} = \text{current entering port } \sigma \text{ at time } n \tag{B.2}$$

$$a_{\sigma,n} = \text{incident power wave at port } \sigma \text{ at time } n \tag{B.3}$$

$$b_{\sigma,n} = \text{reflected power wave at port } \sigma \text{ at time } n \tag{B.4}$$

$$residue_{\sigma,c,p} = p^{\text{th}} \text{ residue value for port } \sigma \text{ from port } c \tag{B.5}$$

$$pole_{\sigma,c,p} = p^{\text{th}} \text{ pole value for port } \sigma \text{ from port } c \tag{B.6}$$

$$remainder_{\sigma,c} = \text{remainder value for port } \sigma \text{ from port } c \tag{B.7}$$

$$z_{ref} = \text{Reference impedance for the measured S-parameter data} \tag{B.8}$$

$$t_s = \text{timestep} \tag{B.9}$$

$$\gamma_{\sigma,c,p} = pole_{\sigma,c,p} t_s \tag{B.10}$$

$$\delta_{\sigma,c,p} = e^{\gamma_{\sigma,c}} \tag{B.11}$$

first order (B.12)

$$\lambda_{\sigma,c,p} = -\frac{residue_{\sigma,c,p}}{pole_{\sigma,c,p}}(1 + \frac{1-\delta_{\sigma,c,p}}{\gamma_{\sigma,c,p}}) \tag{B.13}$$

$$\mu_{\sigma,c,p} = -\frac{residue_{\sigma,c,p}}{pole_{\sigma,c,p}}(\frac{\delta_{\sigma,c,p}-1}{\gamma_{\sigma,c,p}} - \delta_{\alpha,c,p}) \tag{B.14}$$

$$\nu_{\sigma,c,p} = 0 \tag{B.15}$$

second order (B.16)

$$\lambda_{\sigma,c,p} = -\frac{residue_{\sigma,c,p}}{pole_{\sigma,c,p}}(\frac{1-\delta_{\sigma,c,p}}{\gamma^2_{\sigma,c,p}} + \frac{3-\delta_{\sigma,c,p}}{2\gamma_{\sigma,c,p}} + 1) \tag{B.17}$$

$$\mu_{\sigma,c,p} = -\frac{residue_{\sigma,c,p}}{pole_{\sigma,c,p}}(-2\frac{1-\delta_{\sigma,c,p}}{\gamma^2_{\sigma,c,p}} - \frac{2}{\gamma_{\sigma,c,p}} - \delta_{\sigma,c,p}) \tag{B.18}$$

$$\nu_{\sigma,c,p} = -\frac{residue_{\sigma,c,p}}{pole_{\sigma,c,p}}(\frac{1-\delta_{\sigma,c,p}}{\gamma^2_{\sigma,c,p}} + \frac{1+\delta_{\sigma,c,p}}{2\gamma_{\sigma,c,p}}) \tag{B.19}$$

$$\lambda_{\sigma,c} = \sum_p \lambda_{\sigma,c,p} \tag{B.20}$$

$$\mu_{\sigma,c} = \sum_p \mu_{\sigma,c,p} \tag{B.21}$$

$$\nu_{\sigma,c} = \sum_p \nu_{\sigma,c,p} \tag{B.22}$$

$$x_{\sigma,p,n} = \delta_{\sigma,\sigma,p} x_{\sigma,p,n-1} + \sum_c \left(\lambda_{\sigma,c,p} a_{c,n} + \mu_{\sigma,c,p} a_{c,n-1} + \nu_{\sigma,c,p} a_{c,n-2} \right) \tag{B.23}$$

$$b_{\sigma,n} = \sum_c (\sum_p (x_{\sigma,p,n}) + rem_{\sigma,c} a_{c,n}) \tag{B.24}$$

$$b_{\sigma,n} = \sum_c (\sum_p (\delta_{\sigma,c,p} x_{\sigma,p,n-1} + \lambda_{\sigma,c,p} a_{c,n} + \mu_{\sigma,c,p} a_{c,n-1} + \nu_{\sigma,c,p} a_{c,n-2}) + rem_{\sigma,c} a_{c,n}) \tag{B.25}$$

$$b_{\sigma,n} = \sum_c (\sum_p (\delta_{\sigma,c,p} x_{\sigma,p,n-1}) + \lambda_{\sigma,c} a_{c,n} + \mu_{\sigma,c} a_{c,n-1} + \nu_{\sigma,c} a_{c,n-2} + rem_{\sigma,c} a_{c,n}) \tag{B.26}$$

$$h_{\sigma,n} = 2\sqrt{z_{ref}} \sum_c (\sum_p (\delta_{\sigma,c,p} * x_{\sigma,p,n-1}) + \mu_{\sigma,c} a_{c,n-1} + \nu_{\sigma,c} a_{c,n-2}) \tag{B.27}$$

$$b_{\sigma,n} = \sum_c (\lambda_{\sigma,c} a_{c,n} + rem_{\sigma,c} a_{c,n}) + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{B.28}$$

$$b_{\sigma,n} = \sum_{c}(\lambda_{\sigma,c} + rem_{\sigma,c})a_{c,n} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{B.29}$$

$$b_{\sigma,n} = (\lambda_{\sigma,\sigma} + rem_{\sigma,\sigma})a_{\sigma,n} + \sum_{c\neq\sigma}(\lambda_{\sigma,c} + rem_{\sigma,c})a_{c,n} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{B.30}$$

$$b_{\sigma,n} - (\lambda_{\sigma,\sigma} + rem_{\sigma,\sigma})a_{\sigma,n} = \sum_{c\neq\sigma}(\lambda_{\sigma,c} + rem_{\sigma,c})a_{c,n} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{B.31}$$

$$\frac{(1 - \lambda_{\sigma,\sigma} - rem_{\sigma,\sigma})u_{\sigma,n} - (1 + \lambda_{\sigma,\sigma} + rem_{\sigma,\sigma})i_{\sigma,n}z_{ref}}{2\sqrt{z_{ref}}} = \sum_{c\neq\sigma}(\lambda_{\sigma,c} + rem_{\sigma,c})a_{c,n} + \frac{h_{\sigma,n}}{2\sqrt{z_{ref}}} \tag{B.32}$$

$$R_\sigma = z_{ref}\frac{1 + \lambda_{\sigma,\sigma} + rem_{\sigma,\sigma}}{1 - \lambda_{\sigma,\sigma} - rem_{\sigma,\sigma}} \tag{B.33}$$

$$\alpha_{\sigma,c} = \frac{\lambda_{\sigma,c} + rem_{\sigma,c}}{1 - \lambda_{\sigma,\sigma} - rem_{\sigma,\sigma}} \tag{B.34}$$

$$V_{\sigma,n} = \frac{h_{\sigma,n}}{1 - \lambda_{\sigma,\sigma} - rem_{\sigma,\sigma}} \tag{B.35}$$

$$u_{\sigma,n} = R_\sigma i_{\sigma,n} + \sum_{c\neq\sigma}\alpha_{\sigma,c}(u_{c,n} + i_{c,n}z_{ref}) + V_{\sigma,n} \tag{B.36}$$

DC. Starting with Equation B.23 (B.37)

$$a_{\sigma,n} = a_\sigma \forall n \tag{B.38}$$

$$b_{\sigma,n} = b_\sigma \forall n \tag{B.39}$$

$$\text{let } x'_{\sigma,c,p} = \sum_{k=0}^{\infty} (\delta^k_{\sigma,c,p}(\lambda_{\sigma,c,p} + \mu_{\sigma,c,p} + \nu_{\sigma,c,p}))a_c \tag{B.40}$$

$$x'_{\sigma,c,p} = -\frac{\lambda_{\sigma,c,p} + \mu_{\sigma,c,p} + \nu_{\sigma,c,p}}{\delta_{\sigma,c,p} - 1} a_c \tag{B.41}$$

$$\text{let } \psi_{\sigma,c,p} = -\frac{\lambda_{\sigma,c,p} + \mu_{\sigma,c,p} + \nu_{\sigma,c,p}}{\delta_{\sigma,c,p} - 1} \tag{B.42}$$

$$b_\sigma = \sum_c (\sum_p (x'_{\sigma,c,p}) + rem_{\sigma,c} a_c) \tag{B.43}$$

$$b_\sigma = \sum_c (\sum_p (\psi_{\sigma,c,p}) a_c + rem_{\sigma,c} a_c) \tag{B.44}$$

$$b_\sigma = \sum_c ((\sum_p (\psi_{\sigma,c,p}) + rem_{\sigma,c}) a_c) \tag{B.45}$$

$$\text{let } \phi_\sigma = \sum_p (\psi_{\sigma,\sigma,p}) + rem_{\sigma,\sigma} = -\sum_p (\frac{\lambda_{\sigma,\sigma,p} + \mu_{\sigma,\sigma,p} + \nu_{\sigma,\sigma,p}}{\delta_{\sigma,\sigma,p} - 1}) + rem_{\sigma,\sigma} \tag{B.46}$$

$$b_\sigma = \phi_\sigma a_\sigma + \sum_{c \neq p} ((\sum_p (\psi_{\sigma,c,p}) + rem_{\sigma,c}) a_c) \tag{B.47}$$

$$R_\sigma = z_{ref} \frac{1 + \phi_\sigma}{1 - \phi_\sigma} \tag{B.48}$$

$$\alpha_{\sigma,c} = \frac{\phi_\sigma}{1 - \phi_\sigma} \tag{B.49}$$

$$V_\sigma = 0 \tag{B.50}$$

$$u_\sigma = R_\sigma i_\sigma + \sum_{c \neq \sigma} \alpha_{\sigma,c}(u_c + i_c z_{ref}) + V_\sigma \tag{B.51}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & -\alpha_{12} & \alpha_{12} & -R_1 & -z_{ref}\alpha_{12} \\ -\alpha_{21} & \alpha_{21} & 1 & -1 & -z_{ref}\alpha_{21} & -R_2 \end{bmatrix} \begin{bmatrix} v_{1+} \\ v_{1-} \\ v_{2+} \\ v_{2-} \\ i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ V_1 \\ V_2 \end{bmatrix} \tag{B.52}$$

2 port S-parameter equivalent circuit with terms defined above

Class-F Amplifier Circuit

```
% This is the netlist for the Provided F-Class Amplifier
% with COBRA based Transistor Model

% ========== Capacitors ==========
% Biasing capacitor
Cbk 14 7 1000

% Internal Capacitors
Cds 4 6 1.000000000000000e-04
CNgd 2 3 8.414491353653375e-05 0.823819882094039 -3.972818823528234
    ↪ 0.0267363621772036
CNgs 5 6 6.121484546594651e-04 6.840699799958159e-04 4.416304358274649
    ↪ 2.125317001188324

% Parasitic Capacitors
Cpds 11 0 4.000009800000000e-05
Cpd 14 11 0.001000500000000
Cpg 9 10 6.2005e-04
Cpgs 10 0 5.5563e-04

% ========== Resistors ==========
% Terminal Series Resistors
Rd 13 14 2.688
Rg 1 2 1.745
Rs 12 6 0.833
Rp2 16 0 50

% Internal Resistors
Rgd 3 4 100
Rgs 2 5 100

% Parasitic Resistors
% Rpds 11 0 infinity
Rpgs 10 0 397

% ========== Inductors ==========
```

```
35 % Terminal Inductors
36 Ls 0 12 0.008582
37 Ld 4 13 0.1522
38 Lg 9 1 0.1479
39
40 % Biasing Inductor
41 Lrf 8 14 100
42
43 % ========== Current Source ==========
44 % Determined by model params. Needs to also know the four places for
       ↪ reference
45 % i.e V_gs = n5 - n6 (across Cgs)
46 % and V_ds = n4 - n6
47 INds 4 6 5 6 4 6
48
49 % ========== Voltage Sources ==========
50 % Bias
51 Vdsq 8 0 28
52 % Gate
53 VSg 9 15 2.645 1 0
54 %VTg1 9 15 1E-3 TimeSeries/10G_TimeDomain_Data_Dirichlet.txt
55 Vg2 15 0 -2.5
56
57 % ========== S-Parameter Block ==========
58 % This is a NUDTIR block
59 S1 0 2 7 0 16 0 S-Parameters/matchingSparam.s2p
60
61
62 % ========== Directives ==========
63 .transient( 0, 10000, 0.01 )
64 .graph( 7 )
65 .graph( 9 )
66 .outputFile( "Datadumps/MOSNUDTIR.txt" )
67 %.nodc
```

Listing C.1: *class-F amplifier netlist*

Listing C.1 outlines the netlist used to represent the class-F amplifier circuit shown in Figure 5.3.

$$C(u) = C_p + C_o(1.0 + \tanh(P_{10} + P_{11}u)) \tag{C.1}$$

CN is defined as a non-linear capacitor with a capacitance shown in Equation C.1. Where u is the voltage across the capacitor. IN is defined as in Listing 4.1 as `Idrain`, where `r1` and `r2` are v_{gs} and v_{ds} respectively