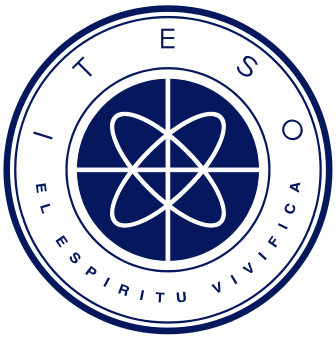
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

MAESTRÍA EN DISEÑO ELECTRÓNICO



IMPROVING HARMONIC BALANCE PERFORMANCE VIA PARALLELIZATION

Tesis que para obtener el grado de

MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: Jorge García Bedoy Torres

Director de tesis: Dr. José Ernesto Rayas Sánchez

Tlaquepaque, Jalisco. Septiembre de 2016.

MAESTRO EN INGENIERIA (2016) ITESO

Maestría en Diseño Electrónico Tlaquepaque, Jal., México

TÍTULO: **Improving Harmonic Balance Performance via Parallelization**

AUTOR: Jorge García Bedoy Torres
Ingeniero en Electrónica (ITESO, México)

DIRECTOR DE TESIS: José Ernesto Rayas Sánchez
Departamento de Electrónica, Sistemas e Informática, ITESO
Ingeniero en Electrónica (ITESO, México)
Maestro en Sistemas Electrónicos (ITESM Campus Monterrey, México)
Doctor en Ingeniería Eléctrica (Universidad McMaster, Canadá)
Senior, IEEE

NÚMERO DE PÁGINAS: viii, 74

Dedicatoria

Dedico esta tesis a mis padres, María Eugenia y Jorge, por su cariño y apoyo incondicional.

Summary

In this thesis, an approach to parallelizing the Harmonic Balance (HB) algorithm for circuit simulation is proposed. Initially, a description of the current state of the art for parallelization, as applied to Electronic Design Automation (EDA), is provided, along with an introduction to the Harmonic Balance algorithm. Previous work on parallelizing the HB algorithm is briefly presented. Next, the necessary netlist parsing infrastructure required for a circuit simulator is described and implemented through the use of regular expressions in Python and subsequently benchmarked against a variety of different circuits. Voltage and current plotting capabilities are also expanded upon at this stage. Afterwards, a more in-depth description of the HB algorithm is provided, explaining step-by-step the generation of the required matrices. Next, it comes a general overview of modern tools and languages used for scientific computing, with a particular focus on Python, culminating in an initial implementation of the HB algorithm in this language. Having developed a baseline circuit simulator implementing the HB algorithm, the different steps in the process are analyzed to identify good parallelization candidates, before making the necessary modifications to enable concurrent evaluation of the non-linear sub-circuit. Finally, a sample circuit with multiple non-linear elements is simulated to evaluate the computational speed-up from the parallelization effort.

Contents

Introduction

In Electronic Design Automation (EDA), the complexity of the circuits that can be simulated accurately is largely limited by computing performance. There are many possible methods and techniques to improve the performance of EDA software, such as employing improved fundamental algorithms, using less accurate but faster models, or making better use of the hardware resources, which in modern systems, typically translates into utilizing multiple compute engines concurrently. This work focuses on the application of parallelization to increase the performance of the Harmonic Balance (HB) algorithm, especially when dealing with circuits with multiple non-linear devices.

Chapter 1 offers an overview of the current state of the art for parallelization efforts as they pertain to EDA, along with an introduction to the Harmonic Balance algorithm. It starts by discussing recent developments in microprocessor design and architecture, focusing on the trend towards using multi- and many-core architectures as a mean to increase performance. It then proceeds to explain how these developments impact EDA efforts. A brief overview of the HB algorithm is presented, stating its benefits in comparison to other circuit simulation methods, elaborating on circuit partitioning into linear and non-linear subcircuits, and summarizing the steps required to obtain the steady-state response of a circuit through this method. It concludes by reviewing prior work on the parallelization of this algorithm.

Chapter 2 is dedicated to the definition, description and implementation of a set of tools meant to simplify the development of a simulator applying the HB algorithm. It uses the SPICE netlist format as a baseline to state the benefits of having a standardized method of describing circuits, before proceeding to elaborate further on the implementation details of a circuit netlist parser in Python, focusing particularly on the requirements to support component model definitions as well as sub-circuits. The parser described in this chapter leverages the Regular Expression (or RegEx) engine available in Python. Regular Expressions are a special type of text patterns that evaluate whether or not an input string meets a particular set of rules. In addition, the netlist parser plotting capabilities are also described and implemented. This chapter concludes by defining some extensions to the SPICE netlist specification to support HB simulation, as well as

some benchmark results to ascertain the performance of the tools previously described.

Chapter 3 provides an in-depth description of the Harmonic Balance algorithm. It starts by providing some historical background on the methodology and the reasoning behind circuit partitioning into linear and non-linear devices. It then describes the process followed to generate the matrices and equations that describe the behavior of both sub-circuits. Having covered the HB algorithm in-depth, the method employed to perform the AC analysis once the HB equation has been solved is described. It then provides an overview of the usage of Python as a programming language for scientific computing, covering both the advantages that it has over other options, such as MATLAB, R, SciLab, as well as the capabilities and libraries that are more commonly used, such as NumPy and SciPy. Next, the details of the actual Python implementation of the HB solver are mentioned, along with a description of the challenges encountered along the way. Finally, the results obtained from simulating the test circuit are stated and analyzed, evaluating the performance and providing further understanding on the main contributors to execution time, focusing particularly on the time employed to evaluate the behavior of the non-linear subcircuit.

Chapter 4 starts with an overview of the current state of CPU design as it pertains to scientific computing. It presents an analysis of the benefits of parallelization for these types of applications, as well as a brief discussion of the requirements for parallel programming to result in improved performance, such as the elimination or avoidance of data dependencies between tasks and the minimization of overhead due to inter-thread communication and synchronization. It then expands upon the work described in Chapter 3, by looking at the results obtained and identifying which steps of the circuit simulation are good candidates for parallelization, focusing on those where the execution takes a large enough percentage of the total time to increase the likelihood of the effort resulting in performance benefits. Next, those modifications proposed are implemented and a test circuit is simulated, resulting in improved performance as additional execution threads are added.

The author wishes to express his sincere appreciation to Dr. José Ernesto Rayas-Sánchez, tenured Professor and director of research in the Computer-Aided Engineering of Circuits and Systems (CAECAS) group at ITESO, not only for his technical guidance and supervision, but also for his unwavering patience throughout the development of this thesis. Special thanks to Intel Tecnologia de Mexico, for providing financial support during the Masters Program, as well as to Jose María Uruñuela, for his support in this endeavour. Finally, thanks to those who kept

encouraging me to finish my thesis, in alphabetical order, Miguel Ambriz, Jessica Hernández, Antonio Luna, Alejandra Medina, Diego Romo, and Ana Villa, without whose insistence, this document would have remained forever incomplete.

1. The Harmonic Balance Algorithm and Parallelization – A Review of the State of the Art

Historically, advances in manufacturing technology have allowed for smaller, more power-efficient and faster circuits. This meant that companies, particularly CPU manufacturers, could keep increasing the operating frequency on their products to improve performance. Unfortunately, when designs reached the 3 GHz mark, it became evident that thermal as well as power delivery issues would no longer allow this paradigm to go on, and a fundamental change in approach was required. Thus began the era of parallelism, where software needs to adapt and evolve in order to take advantage of additional processing resources.

Scientific computing is particularly impacted by this new approach to writing software, since computational complexity is usually high and performance is critical, and currently developed algorithms may not be easily parallelizable. Thus, a great deal of effort has to be applied to both, identifying potential improvements as well as developing the actual implementations.

This chapter attempts to survey the EDA landscape to identify parallelization efforts, particularly in the area of circuit simulation, and specifically as they apply to the Harmonic Balance (HB) algorithm, which is primarily used for microwave and RF circuits. Additionally, the HB algorithm is briefly described in this report, with the intent of eventually developing a parallel implementation of this type of analysis.

1.1. The Road towards Parallelism

Over the years, scientific advances in numerous fields have been made possible by not only the development and refinement of modeling and simulation algorithms, but also by the increased computational power provided by Moore's Law [Intel-12b]. For decades, transistor scaling enabled CPU manufacturers to utilize increasingly higher frequencies as the primary source of enhanced performance.

However, when general purpose processors approached the 3 GHz mark, it became evident that even using new manufacturing processes and improved micro-architectures, frequency scaling

was no longer the most efficient way to increase performance, largely due to the fact that delivering power to the CPU and removing excess heat from it were no longer practical. Thus, a radical shift to multi-core processors took place, with CPU manufacturers making use of increased transistor budgets to increase the number of execution units on a single die [Ramanathan-06].

Furthermore, using multiple cores not only has the potential to increase performance, but also to reduce the total amount of power required to complete a task. Considering the following formula to calculate the power consumed by a processor [Ditzel-10]:

$$P = C_{dyn} V^2 f + P_{Leakage} \tag{1-1}$$

where C_{dyn} is the dynamic capacitance, V is the voltage, f is the frequency, $P_{Leakage}$ is the power consumption due to leakage (on the order of 33%), and P is the total power consumed by the processor, and taking into account that a processor running at a lower frequency is typically able to operate with a lower voltage, it is evident that dividing a task between multiple execution units running each at a lower frequency and voltage, the overall power consumed to perform it can be dramatically lower (see Fig. 1-1). A clear example is that up to a 73% performance improvement can be obtained on a dual-core processor for traditional tasks while consuming only 2% more power [Ramanathan-06].

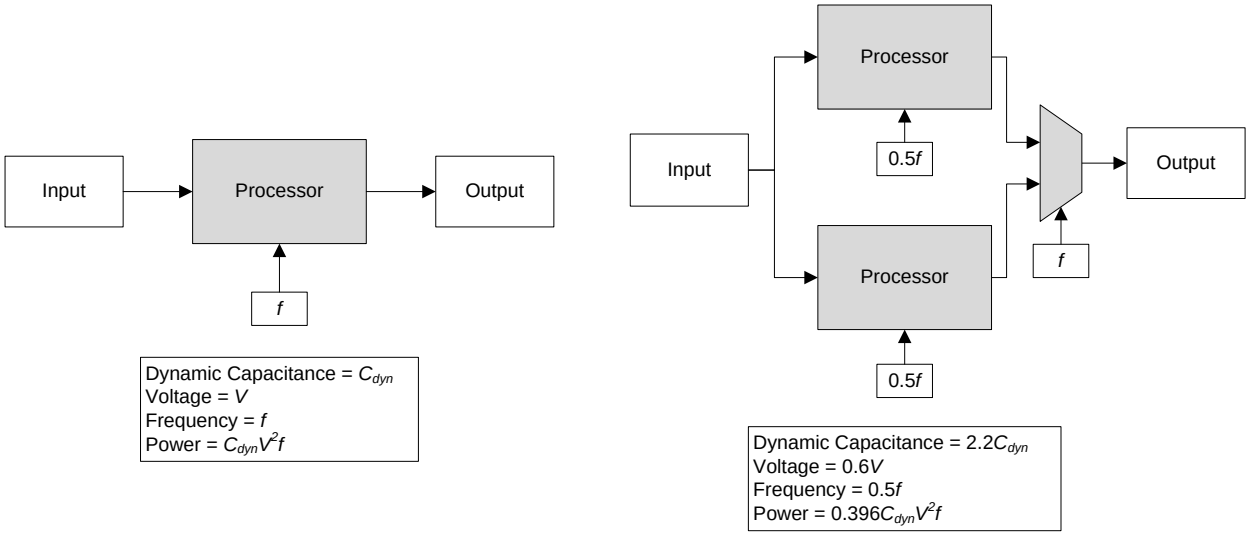


Fig. 1-1 Comparison of power consumption between single and dual processors when performing the same amount of work. Based on Fig. 1.1 from [Munshi-11]

Even though most desktop processors currently have between 2 and 8 cores, some widely available graphics cards can have more than 1,500 execution units[1]. While these "cores" were originally designed with specific workloads in mind (such as pixel shading), it is possible to leverage the existing hardware for other purposes.

1.2. Multi-Core/Many-Core Technology Applied to EDA

One particular area where recent efforts to introduce parallel processing have provided significant speed-ups is in Electronic Design Automation (EDA) [Topaloglu-11], where tasks such as linear algebra operations can benefit from using parallel programming models to exploit the computing power provided by some of these Processing Elements (PE's), regardless of which specific type of architecture is employed [Kapre-09].

The problems that these efforts target are quite varied; some examples include: accelerating the Conjugate Gradient Method to solve linear equation systems for circuit simulation [Maringanti-09], parallelizing fundamental algorithms such as sorting [Edahiro-09], faster logic simulation based on GPUs [Zhang-11], etc.

1.3. A Brief Introduction to Harmonic Balance

Harmonic Balance (HB) is a method used to determine the steady-state response of nonlinear microwave and RF circuits. It is particularly well suited for circuits that have a mix of short and long time constants [Maas-03]. Some of the limitations of time-domain transient techniques that make HB a great analysis method in certain situations are the following:

a) It is hard to analyze certain types of devices, such as dispersive transmission lines or multi-port networks described by Y or S parameters.
b) Circuits with long time constants can require an inordinate amount of simulation cycles when using transient time-domain techniques.
c) With every new linear or non-linear reactive element, an additional differential

[1] GeForce GTX 680 | Specifications | GeForce – Nvidia Corporation (2012), Jun. 24, 2012 http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications

equation needs to be added to the analysis.

Based on this, HB methods are a great choice for most microwave circuits excited with sinusoidal signals, or with other periodic waveforms.

In order to apply HB to a circuit, the first step is usually to partition it into linear and nonlinear sub-circuits (see Fig. 1-2). Additionally, a source is connected to the input and output ports. The linear sub-circuit can be treated as a multiport device and described by a multiport matrix representation (such as S or Y parameters), while the non-linear devices are modeled by their *I/V* or *Q/V* characteristics and are analyzed in the time-domain. This leaves us with an *N*+2 port network with a non-linear device connected to each of the *N* ports, where *N* is the number of ports which have a non-linear element connected.

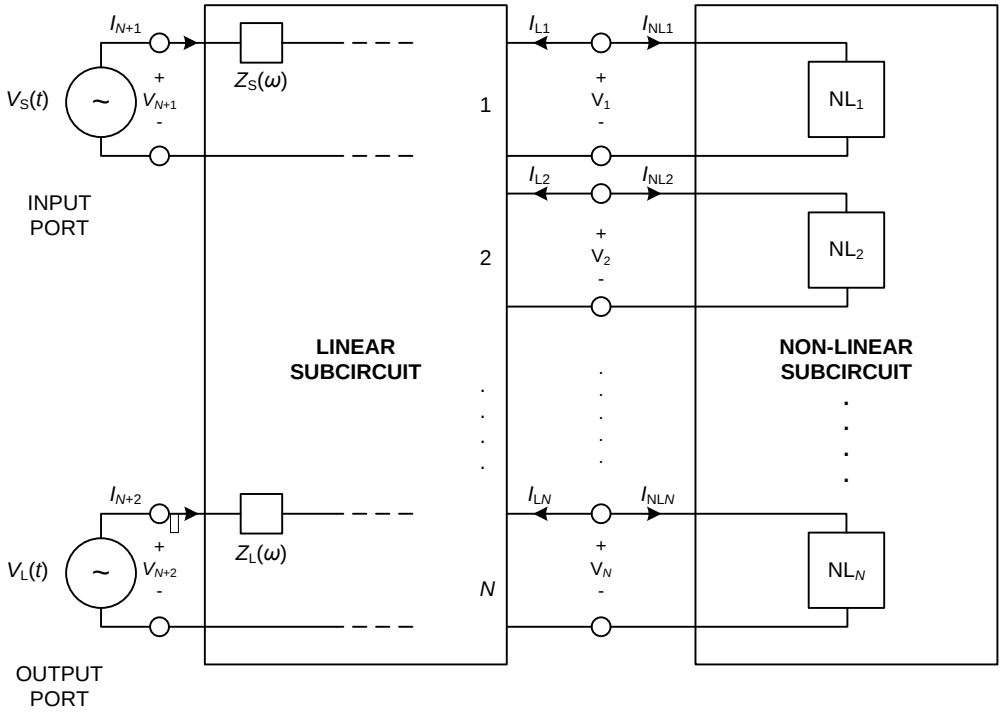


Fig. 1-2 Circuit partitioning for Harmonic Balance. Diagram based on Fig 3.3 from [Maas-03]

The presence of non-linear components in the circuit means that an infinite number of harmonics will exist at each of the ports. However, selecting only the DC component and the first *K* harmonics and ignoring the rest will allow us to describe the circuit operation with a high enough

degree of accuracy while considerably reducing analysis time and complexity.

The circuit is considered as solved or analyzed when the voltage at each of the ports is known, or to put it in another way, when the sum of the currents at each of the ports is 0 (or an acceptably small value). The objective in HB is to find a set of port voltages that result in the same port current for both the linear and non-linear equations (i.e. satisfy Kirchhoff's current law).

The normal process to solve the circuit is to generate the following equation (called the *current-error vector*) by bringing both the currents and the charges of the non-linear sub-circuit into the frequency domain and then use a method to find all zeros of the equation simultaneously for all the selected harmonics,

$$\mathbf{F}(\mathbf{V}) = \mathbf{I}_{\mathrm{S}} + \mathbf{Y}_{N\times N}\mathbf{V} + j\mathbf{\Omega}\mathbf{Q} + \mathbf{I}_{\mathrm{G}} = \mathbf{0} \tag{1-2}$$

where $\boldsymbol{V}$ is the port-voltage vector, $\boldsymbol{I}_{\mathrm{S}}$ represents a set of current sources in parallel with the first N ports, $\boldsymbol{Y}_{N\times N}$ is a $N\times N$ sub-matrix of $\boldsymbol{Y}$, where $\boldsymbol{Y}$ is the transadmittance that represents the linear sub-circuit, $\boldsymbol{\Omega}$ is a diagonal matrix with N cycles of $(0, \ldots, K)\omega_{\mathrm{p}}$ along the main diagonal, $\boldsymbol{Q}$ is the charge vector, $\boldsymbol{I}_{\mathrm{G}}$ is the current vector for nonlinear conductances and controlled current sources.

To summarize, the sequence of steps is the following [El-Rabaie-88]:

a) Apply Kirchhoff's laws to the circuit.
b) Expand the equations generated in the previous step into the frequency domain according to the number of harmonics selected via Fourier analysis.
c) Subdivide the circuit into linear and nonlinear sub-circuits, i.e. rearrange the equations into linear terms on one side and nonlinear components on the other side. This indicates
d) Solve the resulting system of equations (2). The time waveform can be obtained by using an inverse Fourier transform.

Most implementations of the HB method utilize a Jacobian formulation and then apply a Newton or quasi-Newton iterative process to find all zeroes in the HB equation (1-2).

1.4. Previous Efforts to Parallelize the Harmonic Balance Algorithm

Two main efforts [Meng-11] [Li-12] to parallelize the HB algorithm have been identified and both use a very similar approach to achieve parallelization. Both take the Jacobian and use a

pre-conditioner (a block diagonal matrix with $2K+1$ blocks in the diagonal, all of which are independent) and then applying a Generalized Minimal Residual Method (GMRES) to sub-problems created by the partitioning. Additionally, the Fourier transforms are also multi-threaded since they are applied to multiple waveforms.

The results achieved by these efforts are quite good, with a maximum speed-up of 6.17X using 8 threads [Meng-11] when analyzing a mixer and a 3.97X improvement with 9 threads [Li-12] when applied to an oscillator. It is important to consider that the number of threads or processing elements may not necessarily be the limiting factor. A problem that J. Meng et al. mention is the memory usage that a straightforward parallel implementation would incur.

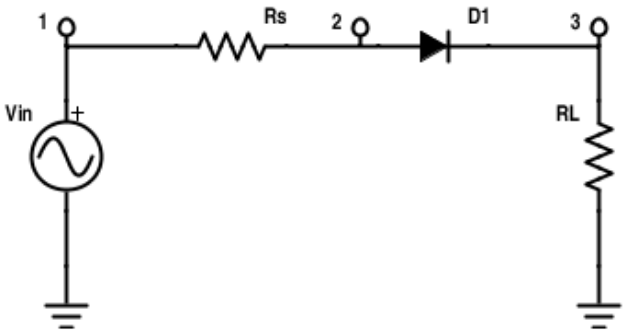
It is important to stress that this is not a purely academic endeavor, since both of the papers referenced above have involvement from established companies in the field (Cadence and TI, respectively). In this work, a slightly different parallelization method will be explored, and the results compared to those achieved in prior work to determine effectiveness.

1.5. Conclusions

The importance of adapting existing EDA algorithms and methods to fully take advantage of new software paradigms is evident in the amount of work that is being done by both industry and academia. Regarding HB parallelization, it has been shown that it is quite viable (two separate examples are referenced in this work) and that further work remains to be done on this subject. In future chapters, the focus will be on developing a Python implementation of the Harmonic Balance algorithm and exploring opportunities for parallelization.

2. Netlist Parsing and Related Infrastructure for Circuit Simulation

Circuit simulators, even those with Graphical User Interfaces (GUI), rely on a specific input format that describes both the circuit being simulated, as well as the operations to perform as part of the simulation process. In the particular case of SPICE [Vladimirescu-81], the circuit is described by providing a text file (called a netlist) that uses element "cards" (i.e., each line in the netlist file) to define circuit topology, component values, and control statements. For instance, Fig. 2-1 shows a SPICE netlist for a half-wave rectifier circuit, along with the equivalent schematic.




```
Half-wave rectifier
*
* This is a comment
*
Rs  1   2   50
D1  2   3   1N4001
RL  3   0   1000
Vin 1   0   SIN (0 1 1kHz)

.END
```

Fig. 2-1 Sample half-wave rectifier circuit and its corresponding SPICE netlist.

Having a standardized format in plain text has multiple advantages, such as allowing users to share circuit definitions, companies to develop competing software implementations capable of simulating the same circuit without having to define a proprietary way of describing it, enabling the output of one program to be used as input for another one (e.g., when driving a SPICE simulator from MATLAB or Python), etc.

It is because of these reasons (among others), that a robust and scalable parsing

infrastructure, along with the corresponding graphing capabilities, is required if a serious attempt is to be made at implementing a circuit solver. A Python-based implementation that meets these requirements is thus proposed, with the necessary capabilities to allow Harmonic Balance analysis.

2.1. Program Structure

The implementation described in this work is split into two main parts, one in charge of parsing the input netlist and providing the solver with all the needed information, and another that receives a result from the solver and creates graphs based on what the user specified in the netlist file. As one of the key objectives of this work is to provide a scalable and portable way to use arbitrary circuit solvers, it is necessary to ensure that the interfaces used for communication between the parser and solver be defined and standardized. Two objects are specified for this purpose:

- A Circuit object, which defines the input interface between the circuit parser (producer) and the solver (consumer). This object needs to contain all the information needed by the simulation process to provide a result, as well as specify the type of analysis to be performed. The following parameters are obtained from processing the netlist:
    - Type of analysis: a dictionary that specifies the types of analysis to be performed, along with the domain-specific parameters (e.g., the number of harmonics to evaluate in the case of Harmonic Balance).
    - Components: a dictionary that contains every component in the circuit, as well as all the relevant parameters. This is the "raw" data from the netlist.
    - Frequency: the main frequency to use for analysis.
    - Nodes: a list that contains all the unique nodes in the system.
    - Ungrounded nodes: a list that contains all the unique ungrounded nodes in the circuit. This is required for AC analysis methods, such as MNA[2].
    - Plots to create: a list that contains the names of the nodes and branches for which the voltage and current will be plotted, respectively.

[2] Modified Nodal Analysis (MNA) – an extension of nodal analysis that also determines certain branch currents, in addition to node voltages.

  - Sources: a list that includes an entry for each source in the circuit, including DC bias sources, along with additional information on these sources (e.g., frequency for a sinusoidal source).
  - Ports: a list that specifies where each non-linear component in the circuit is connected (for Harmonic Balance analysis), along with parameters that pertain to that component.
  - Integer values to specify the number of nodes, ports, sources and ungrounded nodes. Useful when iterating during matrix creation or expansion.

  In addition to this information, certain functionality that is deemed to be useful beyond the requirements of the solver is included in this object, such as error checking methods to ensure that the user's commands are valid (e.g., ensure that the system does not try to plot the voltage for a node that does not exist).

- A Result object, which defines the output interface between the solver (producer) and the plotter (consumer). This object has the values for the frequency domain voltages for all nodes and currents for all independent sources in the circuit.
  - Source currents: vector with the current for each of the independent sources at each of the evaluated harmonics.
  - Node voltage: vector with the voltage of each node in the circuit at each of the evaluated harmonics.

Fig. 2-2 shows the UML diagram of the attributes and methods of the Python implementation being described. At a very high level, the Parser grabs the netlist and produces a Circuit object. This object is then given to the solver and a Result object is expected in return. Having both the Circuit and Result objects available, the Plotter is able to generate the requested graphs. Fig. 2-3 shows the flow diagram of the parsing/plotting process in more detail.

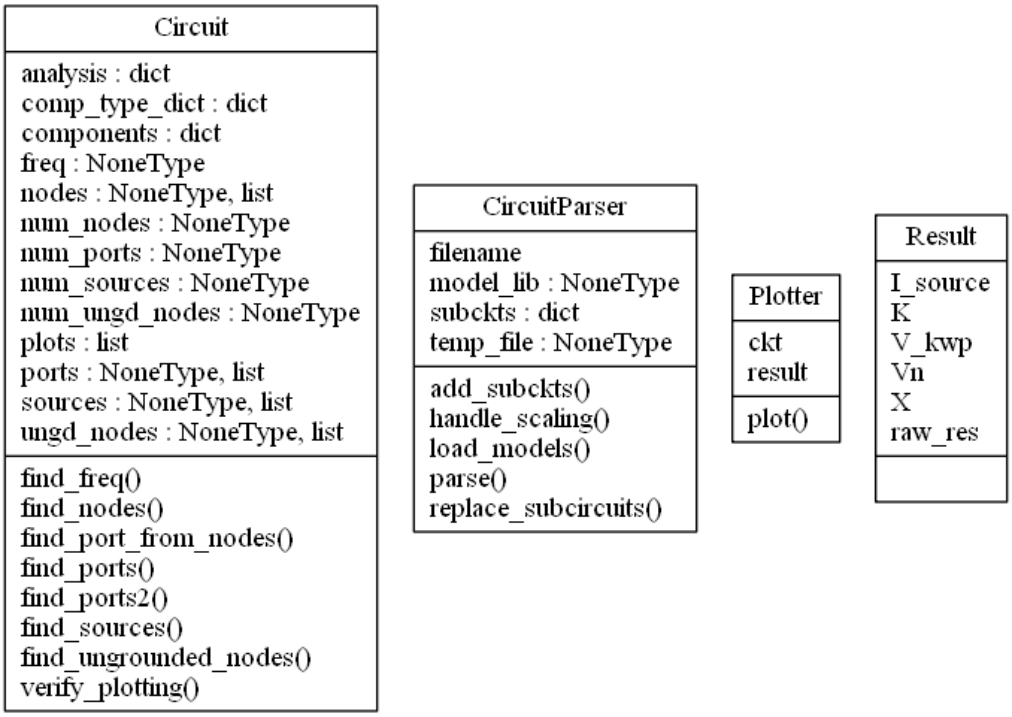


Fig. 2-2 UML class diagram of the parser and plotter application, describing the structure of the system's classes, attributes, and operations.

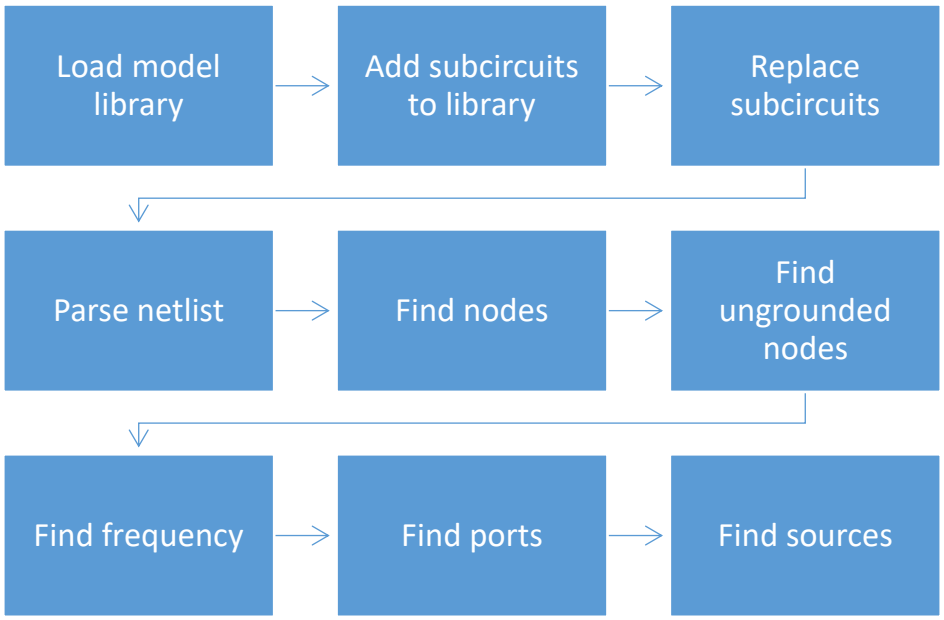


Fig. 2-3 Flow diagram for the netlist parsing process, as implemented in the parser described in this chapter.

2.2. Subcircuits and Models

Sub-circuits are the SPICE analogue to a function on a programming language. They are groupings of elements that are automatically replaced with their definition whenever they are referenced [eCircuitCenter-2013]. Component models tend to operate in a similar fashion, as internally they are replaced by their equivalent circuit (e.g., a bipolar transistor would be replaced by a model that includes parasitic capacitances and resistances) before computation is performed.

Handling sub-circuits and more complex device models presents a particular challenge from a parsing perspective, since it involves a significant amount of node renaming that needs to be handled by the program. To simplify this support, subcircuits and device models are handled in practically the same manner, the only difference being that subcircuits are expected to be defined in the netlist provided by the user, whereas device models are loaded from a predefined folder in the system.

The following four actions need to be taken when performing this process:

i. Identify those nodes that are only used internally by the subcircuit (i.e., those that are not connected to anything specified in the original netlist) and rename them to ensure they are uniquely named.

ii. Identify those nodes that are connected to devices external to the subcircuit (i.e., inputs or outputs, from the subcircuit's perspective) and replace the names used in the definition with those used by the user-provided netlist. For example, a subcircuit may define its inputs as A and B and its output as C, but when added to the rest of the circuit, those connections actually go to X, Y, and Z. Those original names need to be replaced to ensure circuit topology consistency.

iii. Since a particular device model or subcircuit can be used multiple times, it's necessary to ensure that all components defined as part of it have unique names.

iv. Update the reference nodes or branches used by any dependent-sources, since they would otherwise be referring to nodes/components that have changed names during the renaming process.

Once the replacing and renaming process has finished, a temporary netlist is created in the system that only contains component primitives (i.e., every line or component card in the netlist corresponds to one component from a simulation standpoint).

2.3. Parsing and Regular Expressions

The actual parsing process (i.e., scanning the strings contained in the input file) is performed by the Python regular expression (regex) engine. Regular expressions are specific kinds of text patterns that are used to verify whether or not an input string matches the pattern, to find and/or to replace text that matches the pattern, among other things, all without having to write extensive procedural code that would be very complex to maintain or modify. [Goyvaerts-12] Many programming languages (e.g., Java, Perl, Python, etc.) have built-in support for regular expressions.

Regular expressions work by having the engine walk through the regex [Goyvaerts-06], and attempting to match the next token (i.e., the next "character" in the pattern). If it matches, it continues with the next token. Otherwise, it backtracks and attempts to try a different path through the regex. Some examples of regex tokens are the following:

- \w – matches any character a-z, A-Z, 0-9, including _ (underscore)
- [ABCD] – matches a single character that is either A, B, C, or D.
- [0-9] – matches any number from 0 to 9
- + (plus sign) – operates on the previous token, and extends the match from a single character to 1 to N characters.
- . (dot) – wildcard character

Explaining all the regular expressions used in the parser, or the actual Python implementation of the regex engine is beyond the scope of this document. As a simple example, we go through the matching process for resistors, which uses the following regex:

```
"^([Rr]\w+) (\w+) (\w+) ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*[ohms]*"
```

The first token (^) forces the regex engine to match at the beginning of the line. The next one [Rr] matches against a single uppercase or lowercase R (since all resistor definitions in SPICE begin with this letter). Then, a \w+ token will match any alphanumeric string that comes directly after the R. This entire block, from the letter r to the first instance of a whitespace, is contained within parentheses, which causes the regex engine to not only match against what was specified, but also captures the value. From here, the name of the resistor is obtained.

The next two alphanumeric strings are captured as the names of the nodes between which

the resistor is connected by the two instances of (\w+). The next part of the pattern to be matched against and captured corresponds to the value of the resistor. Here, any number between 0 and 9 is accepted, and so are the decimal point, the letter e (for scientific notation) and the minus sign (likewise). The scaling prefix follows. Note how there is an ***** (asterisk) at the end of the matching pattern. This denotes that 0 to N instances of the pattern will be matched, since it's possible for a user to not specify any prefix, and the parser needs to understand that.

To provide a more specific example, a line describing a 1 Kohm resistor connected between nodes A and B would be: `R1 A B 1kohm`. When parsing this line, five different values are captured from it: the name (R1), nodes one and two (A and B), the numeric value (1), and the scaling prefix (k).

From the information obtained by the parsing process, it is possible to build the internal component dictionary describing the circuit topology and its components.

2.4. Plotting Results

Parsing a circuit netlist and performing a simulation is not useful unless the results are simple to obtain and analyze. For this purpose, a set of graphing tools is also developed based on the matplotlib [Hunter-15] library, a Python module meant for data analysis and visualization scientific designed to generate 2D plots from data contained in arrays. It was originally meant to duplicate functionality that exists in MATLAB.

The capabilities implemented for this work are limited to those required to observe the results from a Harmonic Balance simulation. This means that it is possible to plot the voltage from any node in the circuit as well as the current of any independent source. In order to create those plots, all the user has to do is use the same syntax as in SPICE and the plots will be generated automatically. Fig. 2-4 shows an example of a plot obtained from the results of the simulation of a half-wave rectifier circuit. The physical interpretation of the two plots shown in Fig. 2-4 is irrelevant at this moment.

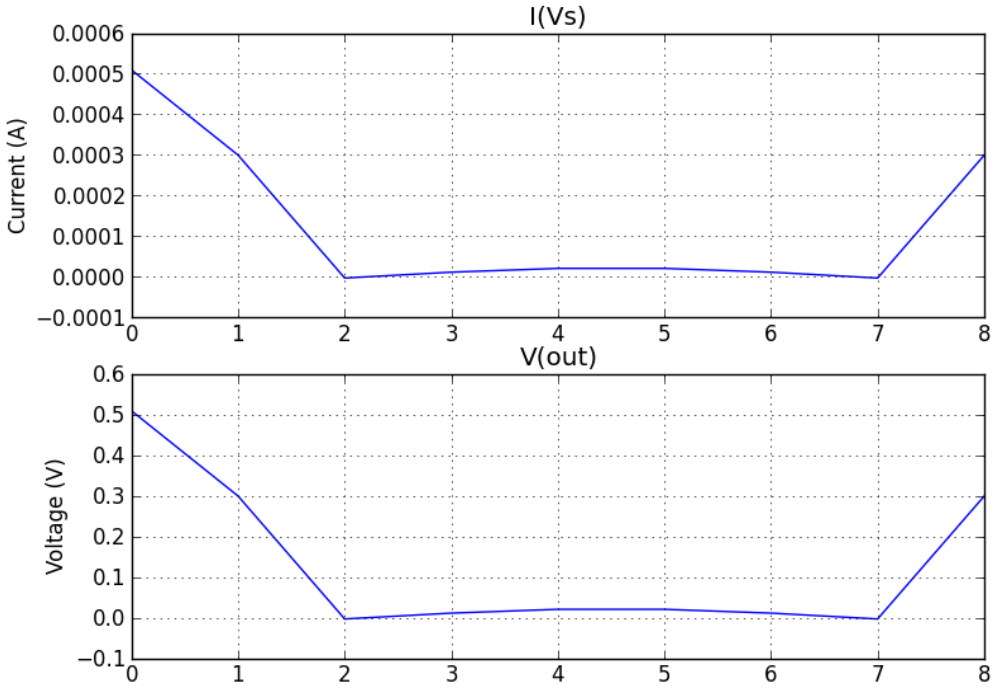


Fig. 2-4 Sample plot for the source current and the output voltage from a half-wave rectifier circuit, as generated by the plotter described in this chapter.

2.5. Defining a SPICE Command for Harmonic Balance

Since SPICE does not support the Harmonic Balance algorithm, new semantics need to be developed to specify that this simulation method is desired, along with the number of harmonics to evaluate. This command takes a very similar form to an AC analysis. It is requested by specifying ".HB K" in the netlist, where K is replaced by the desired number of harmonics to evaluate.

2.6. Results and Analysis

The circuit parser and plotter described in this work is currently being used to test an initial implementation of the Harmonic Balance algorithm. Although Python is not particularly known

for its speed, the parsing process was tested on several benchmark circuits from CircuitSim90[3] and the parsing and processing time never exceeded 72 milliseconds. This time is negligible from a circuit simulation standpoint given that the solving will take significantly more time, particularly for larger circuits.

2.7. Limitations and Potential Improvements

The capabilities of SPICE and the different programs which support it are too extensive and not completely required in the context of this work (i.e., providing infrastructure to simplify the usage of the Harmonic Balance algorithm). There are, therefore, limitations as to what is actually supported, which include, but are not limited, to:

- The current implementation of the circuit parser does not support nested sub-circuit definitions, nor does it support models that include sub-circuits (or vice-versa).
- The plotter assumes that results are in the frequency domain, and includes an inverse Fast Fourier Transform in order to plot the result in the time domain. This is done under the assumption that the Harmonic Balance algorithm will provide the steady-state behavior of the circuit.
- There are some limitations in the handling of both dependent and independent sources (e.g., there is no support for the VPULSE way of defining a voltage source).
- There is currently support only for Diodes and BJTs among semiconductor devices. JFETs and MOSFETs are not supported in the current implementation.

2.8. Conclusions

The functionality described in this chapter has been very useful in developing an HB simulator, since it greatly simplifies making changes to the circuit topology, component values and voltages/currents to plot. Although it meant dedicating significant effort upfront, it has enabled development to focus on the key tasks (i.e., the implementation of the Harmonic Balance solver)

[3] CircuitSim90 – Files from the November 1990 Circuit Simulation and Modeling Worksop at MCNC (1990), Mar. 30, 2015, available at: http://www.cbl.ncsu.edu:16080/benchmarks/CircuitSim90/CircuitSim90.readme

as opposed to spending considerable amounts of time writing temporary methods to review data or modify inputs.

3. The Harmonic Balance Algorithm and a Python Implementation

Obtaining the steady-state response of non-linear circuits through the use of circuit simulation generally requires the solution of differential equations that describe the system. This numerical integration has to be performed until the transient responses disappear and the corresponding steady-state response shows up. This is computationally expensive when it comes to circuits with long time constants relative to their frequencies of operation.

This problem was identified several decades ago, and many solutions and alternatives have been proposed to avoid the computationally intensive process of numerical integration. One such alternative became available as early as 1968, when E.M. Baily [Bailey-68] proposed the first application of the Harmonic Balance algorithm to the solution of non-linear circuits. In spite of having a quite mature theoretical formulation, most of the commercially available SPICE-like circuit simulators do not have implemented the HB algorithm.

In this chapter, the Harmonic Balance method is reviewed, emphasizing the steps involved in obtaining the Harmonic Balance equation from a general circuit description. A Python implementation of the Harmonic Balance algorithm is also presented and the corresponding results are discussed. The main challenges found in this implementation are briefly commented.

3.1. The Harmonic Balance Algorithm

Although it was an improvement in certain scenarios when compared to the “brute-force” approach of solving differential equations, the original Harmonic Balance technique still required too many optimized variables. Subsequent approaches [Nakhla-76] [Maas-03] take advantage of the fact that a considerable part of the network is usually linear (which holds true for microwave and RF circuits), and therefore reduces the number of variables to solve for by partitioning the circuit into linear and non-linear subnetworks, as shown on Fig. 1-2.

The linear subcircuit can be analyzed by relying on a representation such as its Y parameters (i.e., in the frequency domain). The non-linear subcircuit must be analyzed in the time

domain by modeling each component's *I/V* or *Q/V* characteristics. This allows the circuit to be reduced to a (*N*+*M*)-port network, where *N* is the number of non-linear devices in the circuit and *M* is the number of independent sources (including biasing DC sources).

Given the presence of non-linear components, the port voltages and currents have frequency components at harmonics of the original excitation. Since simulating an infinite number of harmonics is impractical, only the DC component and the first *K* frequencies are evaluated, where *K* is a large enough number of harmonics to provide the desired precision, but small enough to be computationally viable (since each additional harmonic requires us to optimize *N* more variables).

The circuit can be considered "solved" when a set of port voltages has been found, such that the port currents are equal for both the linear and non-linear subcircuits. In other words, it needs to comply with Kirchhoff's current law,

$$\begin{bmatrix} I_{1,0} \\ I_{1,1} \\ I_{1,2} \\ \cdots \\ I_{1,K} \\ I_{2,0} \\ I_{2,1} \\ \cdots \\ I_{2,K} \\ \cdots \\ I_{N,K} \end{bmatrix} + \begin{bmatrix} \hat{I}_{1,0} \\ \hat{I}_{1,1} \\ \hat{I}_{1,2} \\ \cdots \\ \hat{I}_{1,K} \\ \hat{I}_{2,0} \\ \hat{I}_{2,1} \\ \cdots \\ \hat{I}_{2,K} \\ \cdots \\ \hat{I}_{N,K} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \cdots \\ 0 \\ 0 \\ 0 \\ \cdots \\ 0 \\ \cdots \\ 0 \end{bmatrix} \tag{3-1}$$

3.1.1 Linear Subcircuit

The above allows us to perform the calculations for each of the current vectors independently. For the linear subcircuit, the current vector can be obtained from

$$\begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 \\ \cdots \\ \boldsymbol{I}_N \\ \boldsymbol{I}_{N+1} \\ \boldsymbol{I}_{N+2} \\ \cdots \\ \boldsymbol{I}_{N+M} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Y}_{1,1} & \boldsymbol{Y}_{1,2} & \cdots & \boldsymbol{Y}_{1,N} & \boldsymbol{Y}_{1,N+1} & \boldsymbol{Y}_{1,N+2} & \cdots & \boldsymbol{Y}_{1,N+M} \\ \boldsymbol{Y}_{2,1} & \boldsymbol{Y}_{2,2} & \cdots & \boldsymbol{Y}_{2,N} & \boldsymbol{Y}_{2,N+1} & \boldsymbol{Y}_{2,N+2} & \cdots & \boldsymbol{Y}_{2,N+M} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{Y}_{N,1} & \boldsymbol{Y}_{N,2} & \cdots & \boldsymbol{Y}_{N,N} & \boldsymbol{Y}_{N,N+1} & \boldsymbol{Y}_{N,N+2} & \cdots & \boldsymbol{Y}_{N,N+M} \\ \boldsymbol{Y}_{N+1,1} & \boldsymbol{Y}_{N+1,2} & \cdots & \boldsymbol{Y}_{N+1,N} & \boldsymbol{Y}_{N+1,N+1} & \boldsymbol{Y}_{N+1,N+2} & \cdots & \boldsymbol{Y}_{N+1,N+M} \\ \boldsymbol{Y}_{N+2,1} & \boldsymbol{Y}_{N+2,2} & \cdots & \boldsymbol{Y}_{N+2,N} & \boldsymbol{Y}_{N+2,N+1} & \boldsymbol{Y}_{N+2,N+2} & \cdots & \boldsymbol{Y}_{N+2,N+M} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{Y}_{N+M,1} & \boldsymbol{Y}_{N+M,2} & \cdots & \boldsymbol{Y}_{N+M,N} & \boldsymbol{Y}_{N+M,N+1} & \boldsymbol{Y}_{N+M,N+2} & \cdots & \boldsymbol{Y}_{N+M,N+M} \end{bmatrix} \begin{bmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \\ \cdots \\ \boldsymbol{V}_N \\ \boldsymbol{V}_{N+1} \\ \boldsymbol{V}_{N+2} \\ \cdots \\ \boldsymbol{V}_{N+M} \end{bmatrix} \tag{3-2}$$

$$\boldsymbol{I}_n = \begin{bmatrix} I_{n,0} \\ I_{n,1} \\ \cdots \\ I_{n,K} \end{bmatrix} \tag{3-3}$$

$$\boldsymbol{V}_n = \begin{bmatrix} V_{n,0} \\ V_{n,1} \\ \cdots \\ V_{n,K} \end{bmatrix} \tag{3-4}$$

where $\boldsymbol{I}_n$ and $\boldsymbol{V}_n$ are a set of sub-vectors that describe the harmonic currents (3-3) and voltages (3-4) at port n, respectively, and the N+M-port trans-admittance matrix is built from a set of diagonal sub-matrices that represent the impedance for that port at each of the harmonics being evaluated,

$$\boldsymbol{Y}_{m,n} = \begin{bmatrix} Y_{m,n}(0) & 0 & 0 & \cdots & 0 \\ 0 & Y_{m,n}(\omega_p) & 0 & \cdots & 0 \\ 0 & 0 & Y_{m,n}(2\omega_p) & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & 0 & \cdots & Y_{m,n}(K\omega_p) \end{bmatrix} \tag{3-5}$$

where ω_p is the angular frequency of the main excitation of the circuit.

For monotone systems (i.e. systems where there is a single excitation frequency, aside from any DC components), the vector containing the voltage for the independent sources is straightforward. It is built by a sub-vector representing each source and the excitation it induces at each of the relevant harmonics, as follows

$$\begin{bmatrix} \boldsymbol{V}_{N+1} \\ \boldsymbol{V}_{N+2} \\ \cdots \\ \boldsymbol{V}_{N+M} \end{bmatrix} = \begin{bmatrix} V_{b1} \\ V_s \\ 0 \\ 0 \\ \cdots \\ V_{b2} \\ 0 \\ \cdots \\ \cdots \\ V_{bm} \\ 0 \\ \cdots \\ 0 \end{bmatrix} \tag{3-6}$$

where V_{b1}, V_{b2}, …, V_{bm} are the DC components of all the independent sources, and V_s is the excitation voltage at the fundamental frequency.

Having established the equations needed to analyze the linear sub-circuit, the next step is to build the N-port admittance matrix from the circuit description. There are several ways to do this, but one that is computationally straightforward is to build an indeterminate impedance matrix. To do this, a zero matrix of size *J* x *J*, where *J* is the number of nodes in the circuit, is built. The admittance representation of the components is then "stamped" on top of it, the same way it would be done for a traditional nodal analysis [Queiroz-95]. Each element has a different stamp. For instance, the stamp for an admittance is

$$\begin{array}{c} \\ v^+ \\ v^- \end{array} \begin{array}{c} \begin{array}{cc} v^+ & v^- \end{array} \\ \begin{bmatrix} Y & -Y \\ -Y & Y \end{bmatrix} \end{array} \tag{3-7}$$

Since the values of the admittances can be a function of the frequency (e.g., for capacitors or inductors), the indeterminate matrix needs to be built for each harmonic being evaluated. Additionally, to avoid unconnected nodes which can in turn result in a singular matrix[4], all ports and independent sources are shunted with 100-ohm resistors. This added component can be subtracted from the port admittance matrix once it is generated.

To obtain the desired matrix, we first generate a port impedance matrix by applying a unity

[4] Harmonic Balance – Going through each Step (2007), May 4, 2015, available at: http://qucs.sourceforge.net/tech/node32.html

current to each of the ports. Having done that, the port impedance can be calculated by looking at the voltage drop seen at the port. In other words, the following linear system needs to be solved for each port, changing the nodes to which the unity current is applied,

$$\begin{bmatrix} Y_{1,1} & Y_{1,2} & \cdots & Y_{1,N} \\ Y_{2,1} & Y_{2,2} & \cdots & Y_{2,N} \\ \cdots & \cdots & \cdots & \cdots \\ Y_{N,1} & Y_{N,2} & \cdots & Y_{N,N} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \cdots \\ V_N \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ \cdots \\ 0 \end{bmatrix} \tag{3-8}$$

where the matrix on the left is the indeterminate admittance matrix that was obtained previously.

The port impedance is simply the port voltage (e.g., $Z_{11} = V_1 - V_2$, assuming the port has its positive terminal in node 1 and its negative terminal in node 2). All that is needed now is to invert the impedance matrix to obtain the desired port admittance matrix. One detail to remember is to remove the 100 Ω resistors that were added when generating the indeterminate admittance matrix. This can be achieved by subtracting 10 mS from the main diagonal of the port admittance matrix.

An additional optimization can be applied if the resulting matrix is analyzed more closely. The matrix that was built in the previous step is of size $(N+M) \times (N+M)$, where N is the number of non-linear devices and M is the number of independent sources. However, the only ports where the currents need to be balanced are those where non-linear devices are connected to the linear subcircuit. This means that we only need to obtain the values for the current sub-vectors from $\boldsymbol{I}_1$ to $\boldsymbol{I}_N$, eliminating the need for calculating $\boldsymbol{I}_{N+1}$ to $\boldsymbol{I}_{N+M}$. We can therefore restate (3-2) as

$$\begin{bmatrix} \boldsymbol{I}_1 \\ \boldsymbol{I}_2 \\ \cdots \\ \boldsymbol{I}_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{Y}_{1,N+1} & \boldsymbol{Y}_{1,N+2} & \cdots & \boldsymbol{Y}_{1,N+M} \\ \boldsymbol{Y}_{2,N+1} & \boldsymbol{Y}_{2,N+2} & \cdots & \boldsymbol{Y}_{2,N+M} \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{Y}_{N,N+1} & \boldsymbol{Y}_{N,N+2} & \cdots & \boldsymbol{Y}_{N,N+M} \end{bmatrix} \begin{bmatrix} \boldsymbol{V}_{N+1} \\ \boldsymbol{V}_{N+2} \\ \cdots \\ \boldsymbol{V}_{N+M} \end{bmatrix} + \begin{bmatrix} \boldsymbol{Y}_{1,1} & \boldsymbol{Y}_{1,2} & \cdots & \boldsymbol{Y}_{1,N} \\ \boldsymbol{Y}_{2,1} & \boldsymbol{Y}_{2,2} & \cdots & \boldsymbol{Y}_{2,N} \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{Y}_{N,1} & \boldsymbol{Y}_{N,2} & \cdots & \boldsymbol{Y}_{N,N} \end{bmatrix} \begin{bmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \\ \cdots \\ \boldsymbol{V}_N \end{bmatrix} \tag{3-9}$$

This is not to say that the rest of the matrix is useless. The lower right and lower left sides of the port admittance matrix can be used to obtain the current delivered by the independent sources, and thus allow for calculations such as input power, once the circuit has been solved, for which we can use

$$\begin{bmatrix} \boldsymbol{I}_{N+1} \\ \boldsymbol{I}_{N+2} \\ \cdots \\ \boldsymbol{I}_{N+M} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Y}_{N+1,1} & \boldsymbol{Y}_{N+1,2} & \cdots & \boldsymbol{Y}_{N+1,N} \\ \boldsymbol{Y}_{N+2,1} & \boldsymbol{Y}_{N+2,2} & \cdots & \boldsymbol{Y}_{N+2,N} \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{Y}_{N+M,1} & \boldsymbol{Y}_{N+M,2} & \cdots & \boldsymbol{Y}_{N+M,N} \end{bmatrix} \begin{bmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \\ \cdots \\ \boldsymbol{V}_N \end{bmatrix} + \begin{bmatrix} \boldsymbol{Y}_{N+1,N+1} & \boldsymbol{Y}_{N+1,N+2} & \cdots & \boldsymbol{Y}_{N+1,N+M} \\ \boldsymbol{Y}_{N+2,N+1} & \boldsymbol{Y}_{N+2,N+2} & \cdots & \boldsymbol{Y}_{N+2,N+M} \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{Y}_{N+M,N+1} & \boldsymbol{Y}_{N+M,N+2} & \cdots & \boldsymbol{Y}_{N+M,N+M} \end{bmatrix} \begin{bmatrix} \boldsymbol{V}_{N+1} \\ \boldsymbol{V}_{N+2} \\ \cdots \\ \boldsymbol{V}_{N+M} \end{bmatrix} \tag{3-10}$$

3.1.2 Non-Linear Subcircuit

Establishing the equations for the non-linear subcircuit is slightly more complex, since it requires the analysis to be performed in the time domain, as opposed to the frequency domain. In order to do this, the voltage sub-vector for each of the ports will be converted through an inverse Fourier transform,

$$\mathcal{F}^{-1}\{\boldsymbol{V}_n\} = v(t) \tag{3-11}$$

Depending on the component considered, the method used to obtain the current vector differs, since non-linear capacitors are handled differently than non-linear conductances. The capacitor is modeled by its Q/V relationship, so the charge is a function of the port voltages, and therefore can be expressed as such

$$q_n(t) = f_{qn}(v_1(t), v_2(t), \dots, v_N(t)) \tag{3-12}$$

which can be sent back to the frequency domain by applying a Fourier transform,

$$\mathcal{F}\{q_n(t)\} = \boldsymbol{Q}_n \tag{3-13}$$

where the charge vector $\boldsymbol{Q}$ includes the charge for each non-linear capacitor at each of the harmonics being evaluated.

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_1 \\ \boldsymbol{Q}_2 \\ \dots \\ \boldsymbol{Q}_N \end{bmatrix} = \begin{bmatrix} Q_{1,0} \\ Q_{1,1} \\ \dots \\ Q_{1,K} \\ Q_{2,0} \\ \dots \\ Q_{2,K} \\ \dots \\ Q_{N,K} \end{bmatrix} \tag{3-14}$$

However, this is still a charge vector, and the desired value is the current, so the time derivative of this vector needs to be calculated. Fortunately, this is straightforward in the frequency domain, since the time derivative simply requires multiplication by $j\omega$ [Korn-00]. So the non-linear capacitor current becomes

$$\boldsymbol{I}_c = j\boldsymbol{\Omega}\boldsymbol{Q} \tag{3-15}$$

Since $\boldsymbol{Q}$ is actually a vector and not a scalar, $\boldsymbol{\Omega}$ needs to be a diagonal matrix that includes each of the harmonics being evaluated

$$\boldsymbol{\Omega} = \begin{bmatrix} 0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \omega_p & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 2\omega_p & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & \cdots & K\omega_p & 0 & 0 & \cdots & \cdots \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 0 & \cdots & \cdots \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & \omega_p & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & K\omega_p \end{bmatrix} \tag{3-16}$$

Non-linear conductances are slightly simpler to handle since they do not require the additional matrix multiplication to perform time derivative, since they are modeled directly by their *I/V* relationship. All that is needed here is to perform the evaluation of the model in the time domain and the application of a Fourier transform to obtain the resulting current vector in the frequency domain.

$$i_{g,n}(t) = f_n(v_1(t), v_2(t), \ldots, v_N(t)) \tag{3-17}$$

$$\mathcal{F}\{i_{g,n}(t)\} = \boldsymbol{I}_{G,n} \tag{3-18}$$

where the $\boldsymbol{I}_{G,n}$ vector is

$$\boldsymbol{I}_G = \begin{bmatrix} \boldsymbol{I}_{G,1} \\ \boldsymbol{I}_{G,2} \\ \cdots \\ \boldsymbol{I}_{G,N} \end{bmatrix} \tag{3-19}$$

Substituting (3-9), (3-14), and (3-19) into (3-1), the following system of nonlinear equations is obtained

$$\boldsymbol{F}(\boldsymbol{V}) = \boldsymbol{I}_S + \boldsymbol{Y}_{N\times N}\boldsymbol{V} + j\boldsymbol{\Omega}\boldsymbol{Q} + \boldsymbol{I}_G = 0 \tag{3-20}$$

which is known as the Harmonic Balance equation. If $\boldsymbol{F(V)} = 0$, it means that $\boldsymbol{V}$ is a valid solution for the circuit and the analysis has been successful. This is the equation that needs to be solved by the simulator. Having reached this point, all that remains is to solve a system of non-linear equations. There are multiple ways to achieve this, but discussing them is beyond the scope of this

document. The Python implementation discussed in later sections uses one such method.

3.2. AC Analysis

As was stated before, the circuit is considered successfully analyzed once a port voltage vector has been found such that $\boldsymbol{F}(\boldsymbol{V}) = 0$. However, knowing this does not directly provide all the node voltages or branch currents, which may be desired as part of the circuit simulation results. To obtain those values, Modified Nodal Analysis (MNA) [Najm-10] is performed on the circuit at each of the harmonics, replacing the non-linear devices by independent voltage sources with a value equivalent to the port voltage for that port at that harmonic. The MNA equation is

$$\boldsymbol{HX} = \boldsymbol{W} \tag{3-21}$$

where $\boldsymbol{H}$ is built by initially stamping the elements that have an admittance representation, and $\boldsymbol{W}$ is created according to the independent sources. Although building the $\boldsymbol{H}$ matrix is similar to building the indeterminate admittance matrix, there are two key differences:

a) Only ungrounded nodes are taken into account.
b) For each voltage source that gets "stamped" into the matrix, a new row and column representing the branch current flowing through that source are added to the admittance matrix, and another row is added to the right hand side of the MNA equation. For instance, the element stamp for an independent voltage source is

$$\begin{array}{c} \\ v^+ \\ v^- \\ i \end{array} \begin{array}{c} \begin{array}{cccc} v^+ & v^- & i & RHS \end{array} \\ \begin{bmatrix} & & +1 & \\ & & -1 & \\ +1 & -1 & & V \end{bmatrix} \end{array} \tag{3-22}$$

Solving (21) for $\boldsymbol{X}$ will provide the voltage for all the nodes and the branch current flowing through all the independent sources (which in this case, includes the sources used to replace the non-linear components). Since this process was done separately for each of the harmonics, the $\boldsymbol{X}$ vector needs to be re-arranged to build voltage vectors that include all the harmonic values for each particular node, as

$$\left\{\begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_N \end{bmatrix}_1, \begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_N \end{bmatrix}_2, \dots, \begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_N \end{bmatrix}_K\right\} \rightarrow \left\{\begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_K \end{bmatrix}_1, \begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_K \end{bmatrix}_2, \dots, \begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_K \end{bmatrix}_N\right\} \tag{3-23}$$

3.3. Scientific Computing with Python

Scientific computing is a collection of tools, techniques and theories used to solve science and engineering problems through the use of computer mathematical models [Golub-12]. There are several different programming languages or computer applications tailored towards solving these problems, such as R[5], MATLAB[6], SciLab[7], C, Fortran and Python (aided by certain extensions), each with different strengths and weaknesses. R is primarily designed for statistical computing, C and Fortran are good choices for problems where performance is key, but are significantly more complex to use (since they are purely programming languages). MATLAB is considerably simpler to use and a great tool for a myriad of modeling and computation challenges, but is expensive to purchase. Python offers a good middle ground in terms of usability, performance and cost (free).

NumPy[8] is an open-source Python package that provides a multidimensional array object, along with routines required to interact with those objects, such as linear algebra, discrete Fourier transforms, etc. Although NumPy was not created with the intent of being a MATLAB clone (unlike Octave[9]), it does provide much of the functionality that typical users are accustomed to. The one key advantage that MATLAB retains is the extensive set of domain-specific add-ons (toolboxes) that are available for purchase.

SciPy[10] builds on top of the capabilities provided by NumPy and provides a collection of mathematical algorithms and other functions that are useful for scientific computing. It includes

[5] R: The R Project for Statistical Computing (2015), May 4, 2015, available at: http://www.r-project.org/
[6] MATLAB – The Language of Technical Computing (2015), May 4, 2015, available at: http://www.mathworks.com/products/matlab/index-b.html
[7] Home – Scilab (2015), May 4, 2015, available at: http://www.scilab.org/
[8] What is NumPy? – NumPy 1.9 Manual (2014), May 4, 2015, available at: http://docs.scipy.org/doc/numpy/user/whatisnumpy.html
[9] GNU Octave (2013), May 4, 2014, available at: http://www.gnu.org/software/octave/
[10] Introduction – SciPy v0.15.1 Reference Guide (2015), May 4, 2015, available at: http://docs.scipy.org/doc/scipy/reference/tutorial/general.html

algorithms for tasks such as integration, optimization, interpolation, Fourier transforms, signal processing, linear algebra, statistics, etc.

3.4. Implementation in Python

A Python implementation of the Harmonic Balance algorithm is now presented. It closely follows the algorithm described in section 3.1 of this chapter, with the exception that it currently lacks support for non-linear capacitors, so in all instances, $\boldsymbol{I}_c$ is considered to be 0. It leverages the netlist parsing and plotting work described in Chapter 2. The indeterminate admittance matrix is built from the component dictionary, and the desired number of harmonics is obtained from the circuit object returned by the netlist parser.

Another simplification is that the initial guess generated by the solver is a zero voltage vector. This may prove problematic since it can prevent the optimization algorithm from converging or can cause it to be stuck at a local minimum. However, for the purposes of demonstrating a Python implementation of the Harmonic Balance algorithm that can be built upon, it is considered acceptable.

Since SciPy supports a range of different optimization and root finding algorithms, the software was written in a way that allows the user to easily switch between them. For the results presented in later sections of this document, the Broyden-Fletcher-Goldfarb-Shanno (BFGS)[11] optimization algorithm was selected since it showed acceptable consistency and speed when solving the circuits tested.

Other operations rely on SciPy's existing capabilities, such as Fast Fourier Transforms (FFT) and Inverse Fast Fourier Transform (iFFT), along with matrix inversion and other linear algebra operations. Leveraging the built-in functions allows for quicker implementation and ensures correctness.

3.5. Implementation Challenges

SciPy's built-in optimization, minimization, and root-finding algorithms for non-linear

[11] BFGS is a quasi-Newton iterative method for solving unconstrained nonlinear optimization problems.

systems are meant to handle real numbers. When trying to solve the Harmonic Balance equation, however, the solution space is $\mathbb{C}^N \rightarrow \mathbb{C}^N$, where N is the number of non-linear devices present in the circuit. This leads the solution algorithm to drop the imaginary part of the solution and thus fail to converge.

The approach used in this work to sidestep this problem is to convert the $\mathbb{C}^N \rightarrow \mathbb{C}^N$ system into a $\mathbb{R}^{2N} \rightarrow \mathbb{R}^{2N}$ system, where the real and imaginary components of each variable are split in order to allow the solution algorithm to move in both planes. This allows existing algorithms to work correctly, but has the downside of doubling the number of variables that need to be optimized.

The optimization of functions that result in complex values also presents challenges, since complex numbers are not considered an ordered field, and thus it is hard to determine if one is greater than the other generically. To get around this, the error function chosen for this implementation takes the sum of the absolute value of the current at each harmonic at each of the ports (3-24).

$$E = \sum \text{abs}(\boldsymbol{F}(\boldsymbol{V}_n)) \tag{3-24}$$

3.6. Test Circuit

In order to verify the implementation discussed in this chapter, a simple half-wave rectifier was chosen, shown on Fig. 3-1. This circuit contains all the needed parts to verify proper operation of the algorithm: several linear components, a source and a non-linear device (an ideal diode). While the circuit itself can be considered trivial, the aim is to ensure that the underlying data structures are properly built and the functions are correctly written.

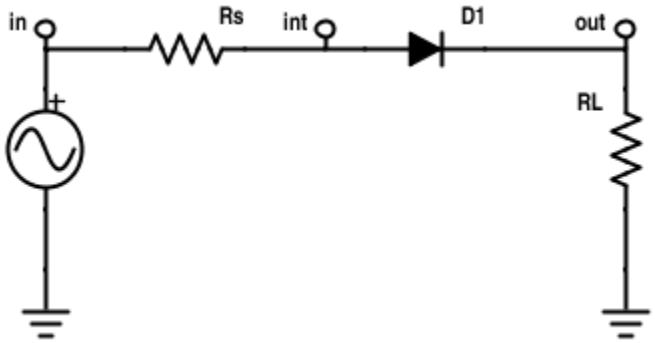
3.7. Results and Analysis

As mentioned previously, the BFGS algorithm was chosen to optimize the variables. The circuit was simulated with two separate levels of precision, 8 and 32 harmonics. Simulation results for the 32 harmonics case are shown on Figs. 3 and 4, and performance metrics can be found on Table I. There are a few crucial data points in the results that are worth analyzing:

a) Although there is only one non-linear device, roughly 77-82% of the execution time is

spent evaluating the non-linear subcircuit. This makes sense, since it requires the calculation of both an FFT and an iFFT in addition to the actual *I/V* relationship evaluation (in this case, Shockley's equation). This presents a good opportunity for speed-up in this area of the problem. Further work will show the potential benefits of parallelizing this analysis when dealing with multiple non-linear devices.

b) For the circuit in question, the algorithm selected is relatively good at finding a solution. Even if not 100% consistent (i.e., certain runs end up diverging due to precision loss), the total amount of error is at least 4 orders of magnitude smaller than the current of the non-linear sub-circuit.




```
Half-wave rectifier

Rs  in      int   50
D1  int     out   Ideal_Diode
RL  out     0     1000
Vin in      0     SIN (0 1 1kHz)

.HB 32 BFGS

.PLOT HB I(Vs) V(out)
.PLOT HB V(in) V(out)

.END
```

Fig. 3-1 Sample half-wave rectifier circuit with a Harmonic Balance netlist.

TABLE I
SIMULATION RESULTS FOR 32 HARMONICS

Time	8 harmonics	32 harmonics
Parsing time (ms)	7.9	7.9
Linear evaluation time (ms)	3286	6439
Non-linear evaluation time (ms)	11292	31320

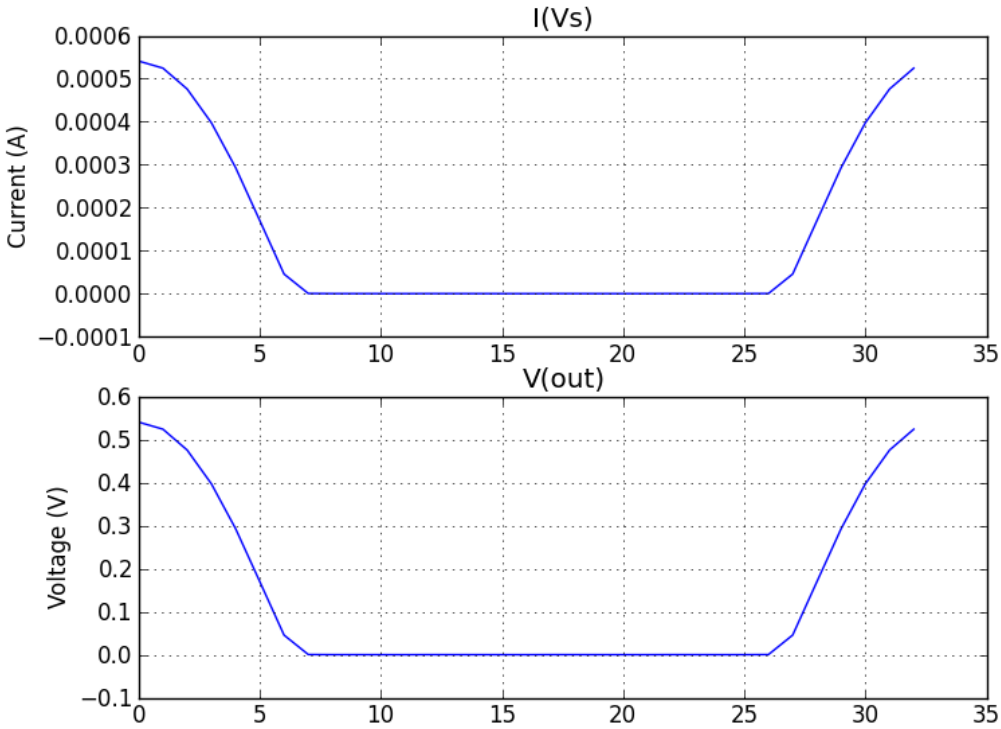


Fig. 3-2 Periodic steady-state responses obtained from the HB simulation of the half-wave rectifier shown in Fig. 3-1, evaluated with 32 harmonics.

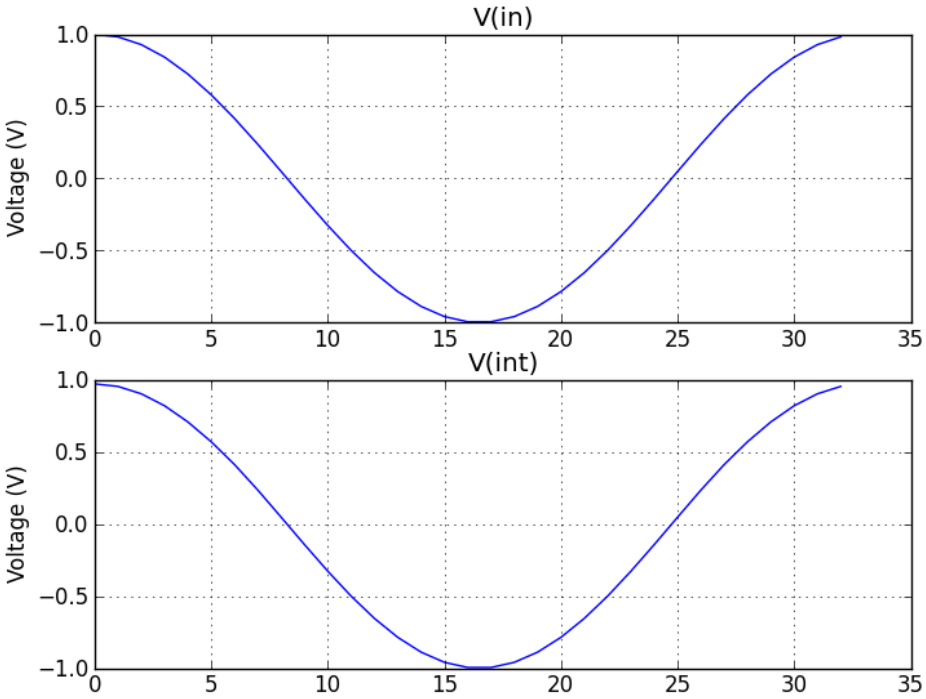


Fig. 3-3 Periodic steady-state responses obtained from the HB simulation of the half-wave rectifier shown in Fig. 3-1, evaluated with 32 harmonics, continued.

3.8. Conclusions

Implementing a Harmonic Balance algorithm that is able to solve generic circuits, instead of developing a dedicated implementation applicable to a specific circuit, was a challenging task. The tools and experience developed in previous chapters to simplify parsing and plotting aided significantly in this process. One of the main roadblocks found in this implementation is associated to the numerical solution of a system of nonlinear equations whose variables are complex, which imposed the use of a non-conventional objective function formulation.

Even without optimizing the performance or improving initial seed calculation or the optimization algorithm being used, the implementation discussed in this work was able to solve the test circuit with a good degree of accuracy and was robust enough to handle 32 harmonics (i.e., 64 variables, given the details of the implementation). It provides a good stepping-stone for the next chapters, which will focus on extracting more performance from parallelization opportunities in the analysis of non-linear devices.

4. Parallelizing the Harmonic Balance Algorithm in Python

As with many other engineering or data science tasks, circuit simulation (and EDA more generally) is limited in what it can achieve, both in terms of the complexity of the circuits being evaluated, as well as in terms of the precision of the simulation, by the computing power available. Over the past several decades, both algorithms (and their implementations) and hardware have improved significantly, leading to even more capable and accurate simulators.

Historically, on the hardware front, additional performance had been provided by newer generations of processors and computing systems in a relatively straightforward fashion, with few to no modifications required to existing software. In recent years, however, gains have not kept pace with historical norms, as frequency increases have stalled [Ross-08] and IPC[12] gains have not compensated for it[13]. This has lead scientists and engineers to look for additional ways to improve the simulation performance[14], and the Harmonic Balance algorithm is no exception [El-Rabaie-88] [Meng-11].

4.1. The Argument for Parallelism

Numerous methods are available to improve the performance of a circuit simulation algorithm (or any other computation task for that matter). All have a set of tradeoffs between additional cost, complexity, and applicability, since not all of them may lend themselves to every problem. These are some of the main avenues to achieve a computational speed-up:

a) Run the simulation on a faster system. While this is very simple in terms of implementation complexity (i.e., no modifications are required to the existing code),

[12] IPC – Instructions per Cycle – a measure of how many instructions are executed by the processor on a given clock cycle. It is a fundamental factor in overall computing performance.
[13] The Free Lunch is Over – a Fundamental Turn Toward Concurrency in Software, Herb Sutter (2005), Jan 10, 2016, available at: http://www.gotw.ca/publications/concurrency-ddj.htm
[14] A quick look at the IEEE IEEExplore database shows a total of 9229 publications related to simulation and parallel processing between 1995 and 2005. For the 2005-2015 period, the number is 31232, suggesting a significant increase in interest. Information obtained on Jan 12, 2016, available at: http://ieeexplore.ieee.org/

cost can be prohibitive, or such a system may simple not be available.

b) Utilize more efficient algorithms that are able to achieve similar/identical results, while requiring fewer computations. For the specific case of circuit simulation, algorithms such as versions of LU factorization (a method by which a matrix is factored into the product of a lower and an upper triangular matrix [Golub-13]), targeted at sparse matrices, can be very useful to reduce the amount of calculations, since sparse matrices tend to be common in such problems [Najm-10].

c) Optimize the current implementation, looking to eliminate unnecessary operations or redundancies, or by using a different programming language, that provides lower overhead. As a broad generalization, interpreted languages (such as MATLAB, Python, Perl, etc.) tend to be slower than compiled languages (such as C, C++, FORTRAN, etc.)[15]. However, development tends to be easier and faster on interpreted languages.

d) Make better use of the capabilities provided by the hardware. Modern microprocessors provide special instructions that can significantly accelerate the execution of certain tasks. One such example is the use of SIMD[16] instructions (such as those included in x86 extensions like SSE [Patterson-12]) that perform the same operation on multiple data at the same time. Another possibility in this area, which is the one explored in this work, is to extract additional performance by explicitly splitting a subset of the required calculations between multiple execution units (i.e., to parallelize the simulation). The main drawback of these methods is that they typically require significant implementation changes to take advantage of the additional hardware.

Considering that in the past few years, companies that design microprocessors (such as Intel, AMD, or ARM) have placed significant emphasis on increasing performance by adding multiple CPU instances [Intel-04], the potential performance gains for those applications that can efficiently take advantage of the additional hardware resources can be staggering.

The result is that, depending on the type of simulation in question, the tradeoff between

[15] Differences between compiled and Interpreted Languages – Codeproject (2013), Jan 10, 2016, available at: http://www.codeproject.com/Articles/696764/Differences-between-compiled-and-Interpreted-Langu

[16] SIMD – Single Instruction Multiple Data. These instructions allow the execution of the same operation on multiple pieces of data at the same time. This is very useful when dealing with arrays in for loops. Initially, entire computers were designed around this paradigm, but its current application is through a set of extensions made to the x86 architecture by Intel and AMD.

additional work/complexity of having to re-write (at least partially) the implementation, can be worth it when considering that the performance increase can be considerable. [Intel-12a]

4.2. Requirements for Parallel Programming

All modern processors employ multiple techniques (e.g., pipelining, out-of-order execution, etc.) to extract Instruction-level parallelism (ILP) [Hennessy-12] from sequential programs without requiring any changes to software. Unfortunately, taking advantage of additional CPU cores requires the user to not only explicitly partition the original application in a way such that work can happen concurrently on independent tasks, but also to account for data dependencies[17] between these tasks to avoid problems. There are two common approaches to this partitioning process: task-parallelism and data-parallelism.

In task-parallelism, each agent is assigned a different task that was part of the original program. A simple example would be having a program that has to calculate the minimum, maximum, and average of a large dataset. Separate CPU's could perform each of those tasks concurrently.[18]

In data-parallelism, each processor performs the same task on different data. Applying data-parallelism to the previous example, the dataset would be split into N parts, and each CPU would be assigned one of these data sub-sets to operate on, with one of them being responsible for putting the results together.

As was hinted earlier, not all problems are good candidates for parallelization, since no program can run faster than the longest chain of dependent computations. It is also important to consider that even for tasks that can be split to be executed in multiple threads; the overhead introduced by the need for threads to communicate and synchronize with each other may outweigh the increase in performance obtained from parallel execution.

[17] If task B requires the output of task A, then it is impossible for both to be executed concurrently.

[18] Understanding task and data parallelism – ZDNet (2007), Jan 14, 2016, available at: http://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129/

4.3. Opportunities for Parallelization in the Harmonic Balance algorithm

The previous chapter described the steps required to apply the Harmonic Balance (HB) algorithm to a circuit. In order to identify the sections best suited for parallelization, the process can be organized in three main groups of tasks:

1) Building the required admittance matrices: performed once and therefore unlikely to be a significant contributor to total execution time. Strong data-dependencies between the different steps involved in matrix creation renders this unsuitable to task-parallelization as well. This can be seen in Fig. 4-1.
2) Evaluating the linear sub-circuit: performed on every iteration of the algorithm, could benefit from data-parallelism.
3) Evaluating the non-linear sub-circuit: performed on every iteration of the algorithm, could also benefit from data-parallelism.

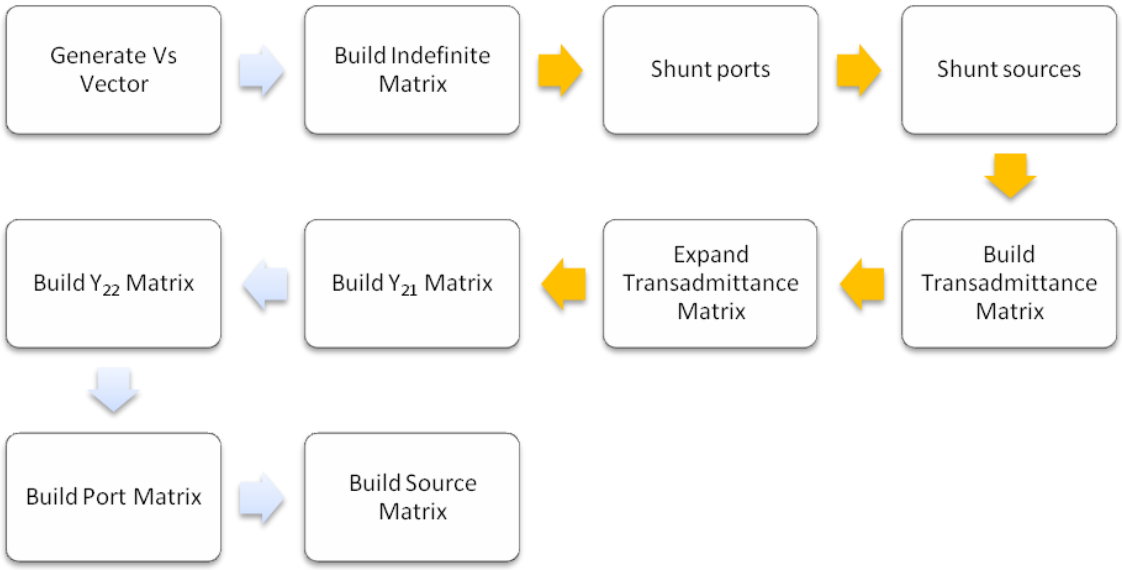


Fig. 4-1 High-level diagram of the initial setup required to generate the matrices required for the Harmonic Balance algorithm. Arrows in orange indicate data dependencies between steps.

4. PARALLELIZING THE HARMONIC BALANCE ALGORITHM IN PYTHON

Considering the HB equation,

$$\boldsymbol{F}(\boldsymbol{V}) = \boldsymbol{I}_S + \boldsymbol{Y}_{N\times N}\boldsymbol{V} + j\boldsymbol{\Omega}\boldsymbol{Q} + \boldsymbol{I}_G = 0 \tag{4-1}$$

it is seen that the current evaluation for the linear sub-circuit is the sum of two components. Although it may be tempting to look for parallelization opportunities for these calculations, closer observation shows that the $\boldsymbol{I}_S$ vector needs to be calculated once, which leaves only $\boldsymbol{Y}_{N\times N}\boldsymbol{V}$ as the operation that has to be performed on each iteration on the algorithm. From a computational standpoint, a single matrix multiplication is unlikely to benefit from multiple execution threads once the additional overhead required is considered.

Obtaining the current values for the non-linear subcircuit is significantly more expensive in computational terms. The main reason for this is the fact that the calculations need to be performed in the time domain instead of the frequency domain (as with the linear sub-circuit). In order to do this, the voltage sub-vector for each of the ports has to be converted through an inverse Fourier transform,

$$\mathcal{F}^{-1}\{\boldsymbol{V}_n\} = v(t) \tag{4-2}$$

Once the *I/V* and *Q/V* for non-linear conductances and capacitances have been evaluated in the time domain, additional applications of the Fourier transform are required to go back to the frequency domain in order to calculate the Harmonic Balance equation,

$$\mathcal{F}\{i_{g,n}(t)\} = \boldsymbol{I}_{G,n} \tag{4-3}$$

$$\mathcal{F}\{q_n(t)\} = \boldsymbol{Q}_n \tag{4-4}$$

$$\boldsymbol{I}_c = j\boldsymbol{\Omega}\boldsymbol{Q} \tag{4-5}$$

Both of these computations are great candidates for data parallelization, since the transformation into time domain (and back into frequency domain) for each of the ports where non-linear components are connected are not data-dependent between ports, and thus can be evaluated concurrently on separate CPU's. Doing this converts (4-1) into a set of *N* independent inverse Fourier transforms

$$\{\mathcal{F}^{-1}\{\boldsymbol{V}_1\}, \mathcal{F}^{-1}\{\boldsymbol{V}_2\}, \dots, \mathcal{F}^{-1}\{\boldsymbol{V}_N\}\} = \{v_1(t), v_2(t), \dots, v_N(t)\} \tag{4-6}$$

where $\boldsymbol{V}_N$ is the voltage at port *N* in the frequency domain. The computations to convert the non-linear charge and current vectors back to the frequency domain can also be split in a similar fashion, so (4-3) and (4-4) respectively become

$$\left\{\mathcal{F}\{i_{g,1}(t)\}, \mathcal{F}\{i_{g,2}(t)\}, \dots, \mathcal{F}\{i_{g,N}(t)\}\right\} = \{\boldsymbol{I}_{G,1}, \boldsymbol{I}_{G,2}, \dots, \boldsymbol{I}_{G,N}\} \tag{4-7}$$

$$\{\mathcal{F}\{q_1(t)\}, \mathcal{F}\{q_2(t)\}, \dots, \mathcal{F}\{q_N(t)\}\} = \{\boldsymbol{Q}_1, \boldsymbol{Q}_2, \dots, \boldsymbol{Q}_N\} \tag{4-8}$$

However, these steps are not the only parts of the non-linear subcircuit evaluation that can lend themselves well to data-parallelism. If one considers that some non-linear devices, such as diodes, have an *I/V* relationship that depends entirely on the voltage applied across their two terminals, it becomes evident that there would be no data-dependencies between the *I/V* relationships of two such devices when solving

$$i_{g,n}(t) = f_n(v_1(t), v_2(t), \dots, v_N(t)) \tag{4-9}$$

Thus, it becomes possible to restate the previous equation into something that allows for concurrent execution across multiple threads

$$\{i_{g,1}(t), i_{g,2}(t), \dots, i_{g,N}(t)\} = \{\boldsymbol{f}_1(v_1(t)), \boldsymbol{f}_2(v_2(t)), \dots, \boldsymbol{f}_N(v_N(t))\} \tag{4-10}$$

Considering the previous parallelization opportunities, it becomes possible (at least in some case) to analyze each non-linear component independently, or to be more precise, the *I/V* relationship in the non-linear side of each port may be evaluated independently. Non-linear capacitances can be subjected to a similar approach

$$q_n(t) = f_{qn}(v_1(t), v_2(t), \dots, v_N(t)) \tag{4-11}$$

$$\{q_1(t), q_2(t), \dots, q_N(t)\} = \{\boldsymbol{f}_{q1}(v_1(t)), \boldsymbol{f}_{q2}(v_2(t)), \dots, \boldsymbol{f}_{qN}(v_N(t))\} \tag{4-12}$$

One restriction that has so far been applied to this parallelization effort is the assumption that the current across a port is dependent solely on the voltage across its two terminals. This is not entirely necessary. A hybrid of data and task-parallelism could be employed such that each thread can operate on the entirety of the dataset (i.e., the voltage vector) and perform a different task to obtain the current at each of the ports. This could be expressed in the following manner

$$\{i_{g,1}(t), i_{g,1}(t), \dots, i_{g,1}(t)\} = \begin{matrix} \{\boldsymbol{f}_1(v_1(t), v_2(t), \dots, v_N(t)), \\ \boldsymbol{f}_2(v_1(t), v_2(t), \dots, v_N(t)), \\ \dots, \boldsymbol{f}_N(v_1(t), v_2(t), \dots, v_N(t))\} \end{matrix} \tag{4-13}$$

where each function can be evaluated independently on a separate thread. Again, an equivalent can be stated for non-linear capacitances.

Finally, the evaluation of the non-linear elements is not the only area that lends itself reasonably well to parallelization. The AC analysis that is performed once the HB equation is considered solved can also be parallelized, since the results at each frequency are independent

from each other, and can therefore be calculated concurrently. However, based on experimental data, the time spent in this process is not long significant enough to merit the effort or the overhead required to handle this process in multiple threads (considering that there are communication and serialization tasks that are required for parallel computation).

4.4. Implementation in Python

Having identified several opportunities for parallelization, the Python implementation described in the previous chapter needs to be modified to allow for concurrent execution across multiple threads. Fortunately, the changes required to parallelize the non-linear sub-circuit evaluation as described in the previous section are constrained to a specific function in the existing implementation.

The change required lies in the function that builds the $\boldsymbol{I}_G$ vector that corresponds to the current for non-linear conductances. Fig. 4-2 shows the steps involved in generating this vector, along with those that can be safely executed concurrently for different ports. Before starting this part of the evaluation, the main thread will build the voltage vector for all the ports, and provide each available execution thread with the data that corresponds to one particular port. This process is continued until all ports have had their currents evaluated.

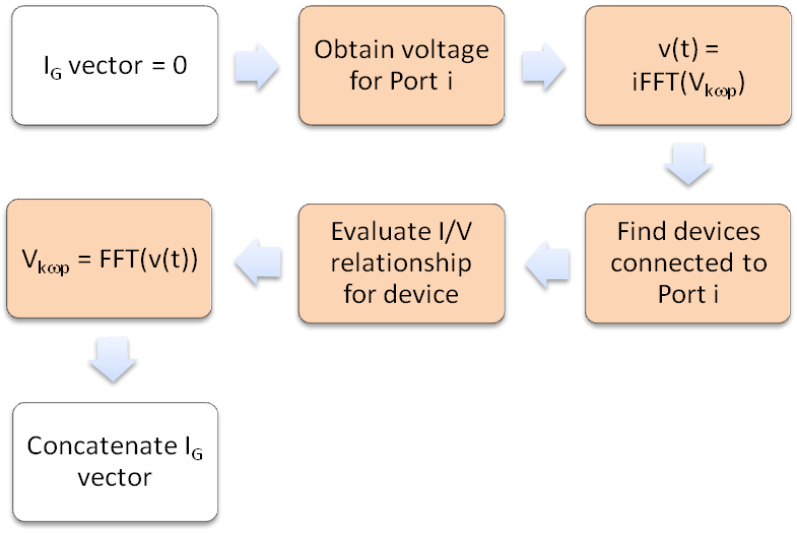


Fig. 4-2 High-level diagram of the steps required to calculate $\boldsymbol{I}_G$. Blocks in orange are executed in parallel for each port.

The main execution thread will wait until all CPU's have finished with their calculations before continuing with the next step of the algorithm. This is required to ensure data integrity.

4.5. Implementation Challenges

The default Python implementation (called CPython) has a key limitation when it comes to parallelizing a particular workload. To make sure that code is thread-safe[19], a mechanism called the Global Interpreter Lock was created, which prevents true parallel execution from occurring, even when utilizing the typical facilities provided for multi-threading. Thankfully, a Python package called multiprocessing[20] allows this limitation to be bypassed by generating multiple instances of the Python interpreter.

Even after employing this mechanism, initial work resulted in the parallelization overhead leading to much longer execution times when compared to the original, serial implementation (several orders of magnitude slower). After significant debug, the cause for this slowdown was determined to be suboptimal use of the Process Pool capabilities.

The problem lies in the fact that creating new processes is significantly more expensive than creating new threads, particularly when running on Windows. This is because creating a new process requires a new "environment" to be generated. To accomplish this, multiple data structures first need to be copied from the original process, and then modified to reflect the fact it is a separate process from its parent (i.e., the process that created it) [Love-10].

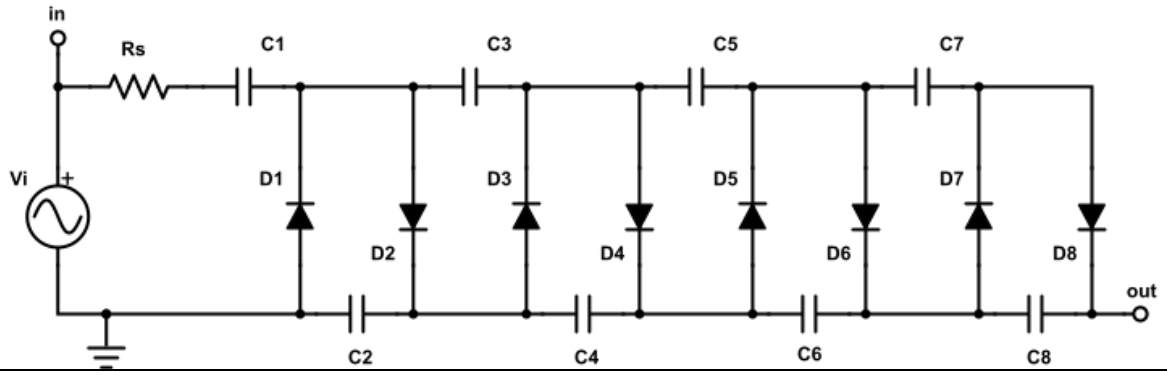
The specific issue encountered during this effort was caused by the implementation re-creating the processes required for parallel execution on each iteration of the equation-solving algorithm. It was solved by creating an initial "pool" of processes that are always waiting for new data to work on, and having the main execution thread assign them work when required.

All the synchronization and passing of data is abstracted by the multiprocessing package, making the implementation itself quite straightforward.

[19] A piece of code is said to be thread-safe if it manipulates all shared resources in a way that guarantees safe execution by multiple threads concurrently.
[20] 16.6 multiprocessing – Process-based "threading" interface – Python foundation (2015), Jan 16, 2016, available at: https://docs.python.org/2/library/multiprocessing.html

4.6. Test Circuit

In order to assess whether the parallelization strategy discussed here yields any performance improvements, it was necessary to select a circuit that contained more than one non-linear component, but was still relatively straightforward. Although it is not used in RF or high frequency applications[21], the Cockcroft-Walton voltage multiplier, shown in Fig. 4-3, is a particularly good choice for this task. It allows to select how many non-linear components are desired, and the non-linear components are all diodes, which makes the implementation simpler, but still allows evaluating the effectiveness and correctness of the proposal.




```
8X Cockcroft-Walton Voltage Multiplier
Vi  in  0  SIN(0V 5V 1000)
Rs  in  99 50ohms
C1  99 1   1000uF
C2  2  0   1000uF
C3  1  3   1000uF
C4  2  4   1000uF
C5  3  5   1000uF
C6  4  6   1000uF
C7  5  7   1000uF
C8  6  out 1000uF
D1  0  1   Ideal_Diode
D2  1  2   Ideal_Diode
D3  2  3   Ideal_Diode
D4  3  4   Ideal_Diode
D5  4  5   Ideal_Diode
D6  5  6   Ideal_Diode
D7  6  7   Ideal_Diode
D8  7  out Ideal_Diode
* Setup for 16 harmonics, solved via BFGS and running on 1 thread
.HB 16 BFGS 1
```

Fig. 4-3 Cockcroft-Walton voltage multiplier circuit used for benchmarking and its corresponding netlist.

[21] While the Cockcroft-Walton voltage multiplier is not really used in RF or high frequency circuits, it has had very interesting applications related to particle accelerators, as well as in nuclear engineering.

The circuit used for benchmarking the implementation includes eight ideal diodes and eight linear capacitors, as illustrated in Fig. 4-3. The corresponding responses of the circuit, after Harmonic Balance analysis, are shown in Fig. 4-4 and Fig. 4-5.

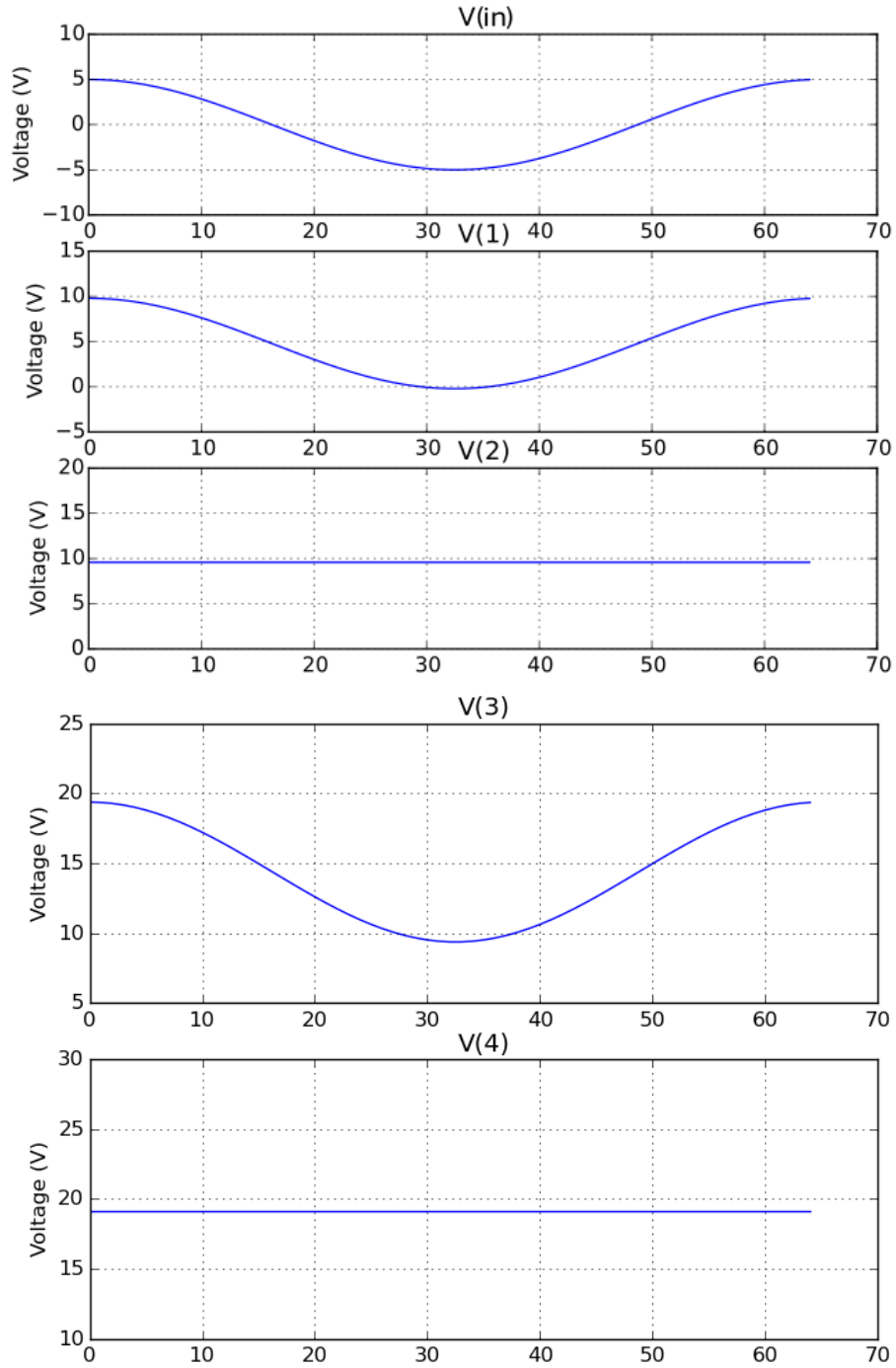


Fig. 4-4 Steady state circuit responses for the Cockcroft-Walton voltage multiplier used for benchmarking, part 1.

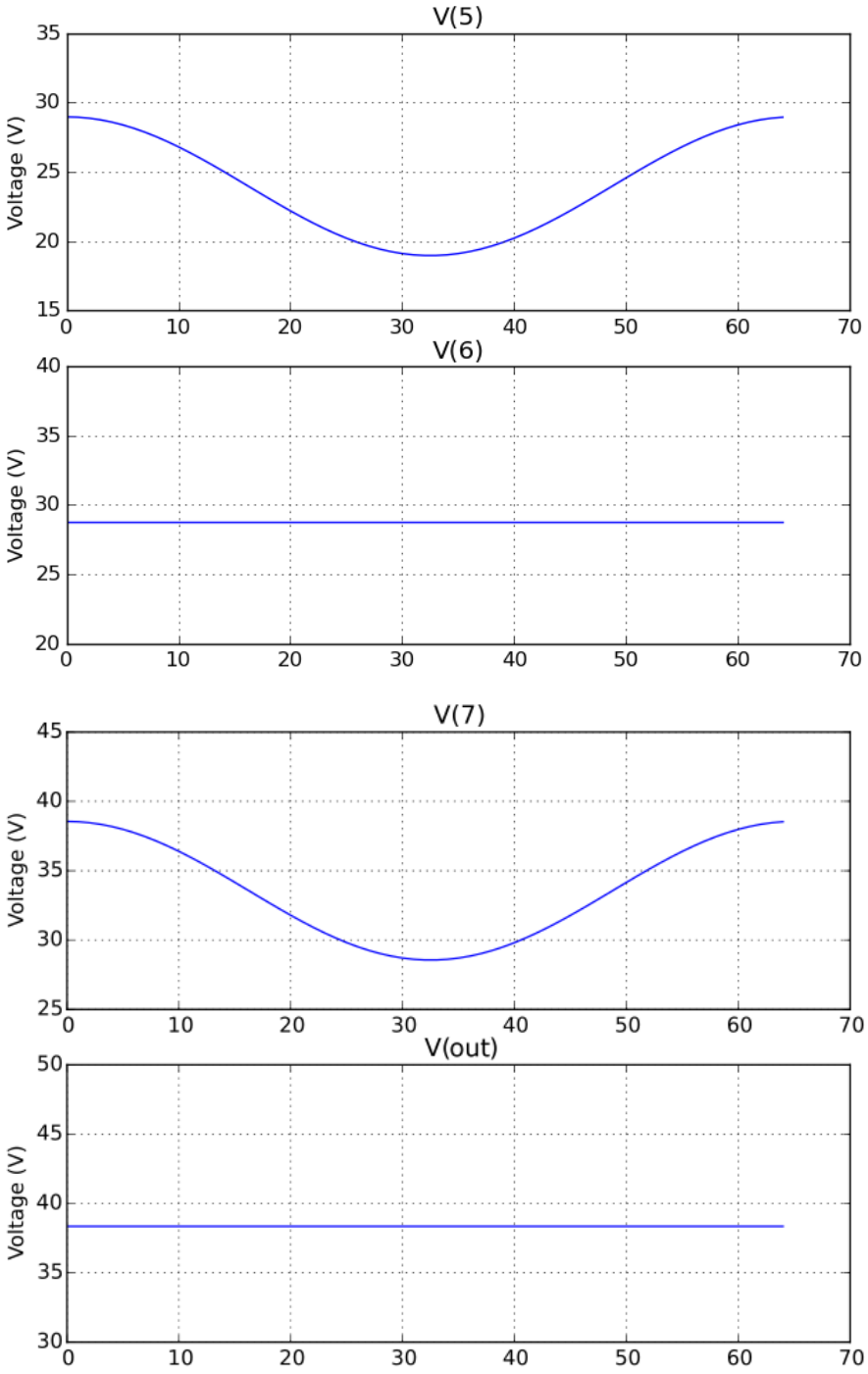


Fig. 4-5 Steady state circuit responses for the Cockcroft-Walton voltage multiplier used for benchmarking, part 2.

4.7. Results and Analysis

The circuit shown on Fig. 4-3 was simulated with three different levels of precision, 16, 32, and 64 harmonics. The program was executed on an Intel® Core i7-3770K quad core processor with eight logical execution threads running Ubuntu Linux 14.04.2 LTS. The results shown on Tables II, III, and IV are the averages of three separate runs for each instance. There are several interesting pieces of data that are worth analyzing:

a) At the lower precision levels (16 and 32 harmonics), performance always increases with additional thread count. This is not the case for 64 harmonics, where the performance with eight threads is significantly worse than with four. There are two possible reasons for this, and both are likely contributors.
    a. The CPU in question only contains four physical cores, so any number of threads beyond four will start sharing hardware resources from the same core.
    b. The dataset is significantly larger as additional harmonics are evaluated, which increases the amount of data that needs to be moved back and forth between processes, adding overhead.

b) The speed-up obtained by parallelization is higher at the lower harmonic counts, with the 16-harmonic run exhibiting a 1.43X speed-up, but the 64-harmonic only improving by 1.17X.

c) Although previous experimentation had shown that up to 82% of the time was spent evaluating the non-linear sub-circuit, not all of that time is actually spent evaluating the error function (which is the specific part of the code that was parallelized), but there's a significant time component that is spent inside the optimization algorithm, performing other operations. This has a direct impact in the performance improvement that can be obtained by the parallel implementation. Due to this, the largest improvement observed for the evaluation of the non-linear sub-circuit is ~1.6X.

d) In all the cases evaluated, the parallel implementation is faster than the single threaded one, even if only slightly. Additionally, using two threads tends to be enough to see most of the achievable speed-up.

TABLE II

PARALLELIZATION SIMULATION RESULTS FOR 16 HARMONICS

Time	1 Thread	2 Threads	4 Threads	8 Threads
Total time (s)	19.629	14.109	13.922	13.723
Non-linear evaluation time (s)	16.089	10.573	10.245	10.048

TABLE III
PARALLELIZATION SIMULATION RESULTS FOR 32 HARMONICS

Time	1 Thread	2 Threads	4 Threads	8 Threads
Total time (s)	25.808	20.586	19.481	19.232
Non-linear evaluation time (s)	17.701	12.152	11.307	11.024

TABLE IV
PARALLELIZATION SIMULATION RESULTS FOR 64 HARMONICS

Time	1 Thread	2 Threads	4 Threads	8 Threads
Total time (s)	51.869	44.104	42.896	44.333
Non-linear evaluation time (s)	24.923	17.363	15.406	16.534

4.8. Conclusions

Once the Harmonic Balance algorithm was implemented in Python in the previous chapter, it became relatively straightforward to apply parallelization to those areas where the return on investment was deemed highest. What is presented in this chapter is by no means a complete simulator, nor an optimal one, but it provides a good stepping-stone for additional research into parallelization in circuit simulation.

It is evident that a lot of additional optimizations and capabilities could be added to the existing simulator implementation. Having said that, what's currently available is enough to demonstrate that performance gains are to be had by taking advantage of existing hardware, since

most PC’s shipping today have at least 2 CPU cores. They key takeaway from this effort is that the Harmonic Balance algorithm can definitely benefit from even relatively simple parallelization efforts.

General Conclusions

This thesis focused on the benefits that parallelization can bring to EDA software and its applications. Specifically, it elaborated on the Harmonic Balance algorithm and how some of its characteristics, particularly those related to the evaluation of non-linear circuit components in time domain, via the use of the Fast Fourier Transform, make it a good candidate for relatively straightforward parallelization efforts. Although the performance of the final implementation described in this work does not scale linearly with number of threads or compute engines, it does show significant improvements, particularly if one considers that only a subset of the entire simulation process was parallelized.

Additional contributions from this work include generic circuit netlist parser and plotting tools that can be leveraged for other work in the area of circuit simulation with very little effort and no financial cost, considering that Python does not require a commercial license to be purchased, and is one of the easier programming languages to learn.

Future work could focus on a multitude of paths. Additional performance profiling and support for a more varied set of non-linear devices should result in a faster and more capable HB-based simulator. Alternatively, replacing Python with a faster programming language or more specifically, one meant for massively parallel implementations, such as OpenCL or CUDA, would be able to leverage existing kernels meant to speed-up operations such as Fourier Transforms or Conjugate Gradient calculations.

Appendix

A. GENERATED FILES

Filename	Author	Short Description
HB_circuit_partition.vsd	J. García Bedoy Torres	Diagram showing circuit partitioning used for Harmonic Balance analysis.
MP_power_comparison.vsd	J. García Bedoy Torres	Diagram comparing power consumption for a particular task between single and dual processors.
Half_wave_rectifier.png	J. García Bedoy Torres	Sample half-wave rectifier circuit with SPICE netlist
uml_diagram.png	J. García Bedoy Torres	UML diagram of the parser and plotter application
parser_flow.docx	J. García Bedoy Torres	Flow diagram for the netlist parsing process.docx
plot_example.png	J. García Bedoy Torres	Sample plot from a half-wave rectifier circuit
Half_wave_rectifier.png	J. García Bedoy Torres	Sample half-wave rectifier circuit with SPICE netlist
half_wave_fig1.png	J. García Bedoy Torres	Periodic response obtained from the simulation results of the half-wave rectifier shown on Figure 2, continued
half_wave_fig2.png	J. García Bedoy Torres	Periodic response obtained from the simulation results of the half-wave rectifier shown on Figure 2, continued
Setup_diagram.docx	J. Garcia-Bedoy-Torres	Diagram of the setup steps for HB algorithm.
Voltage_multiplier.png	J. Garcia-Bedoy-Torres	Schematic of the 8X Cockcroft-Walton voltage multiplier.
Ig_diagram.docx	J. Garcia-Bedoy-Torres	Diagram of the steps to calculate I_G
cw_8diode_64har_8thread_1.png	J. Garcia-Bedoy-Torres	Circuit response for the Cockcroft-Walton voltage multiplier used for benchmarking, part 1.
cw_8diode_64har_8thread_2.png	J. Garcia-Bedoy-Torres	Circuit response for the Cockcroft-Walton voltage multiplier used for benchmarking, part 2.
cw_8diode_64har_8thread_3.png	J. Garcia-Bedoy-Torres	Circuit response for the Cockcroft-Walton voltage multiplier used for benchmarking, part 3.
cw_8diode_64har_8thread_4.png	J. Garcia-Bedoy-Torres	Circuit response for the Cockcroft-Walton voltage multiplier used for benchmarking, part 4.

B. PYTHON CODE FOR THE HARMONIC BALANCE IMPLEMENTATION

This appendix contains all the Python code used in the previous chapters.

circuit.py

```
import sys


class Circuit(object):
    """
        This is the object returned by the circuit parser, which is then
        provided to the circuit simulator.

        Still need to provide a structure for the circuit object to contain
        the description of the analysis to be performed. Specifically,
        we need to create an "HB" syntax for the netlist, where the
        number of harmonics will be defined.

        The plotting infrastructure also needs to be written, but that will
        live somewhere else.
    """

    comp_type_dict = {"d": "diode", "g": "nl_vccs"}

    def __init__(self):
        self.nodes = None
        self.num_nodes = None
        self.ports = None
        self.num_ports = None
        self.sources = None
        self.num_sources = None
        self.freq = None
        self.ungd_nodes = None
        self.num_ungd_nodes = None
        self.components = {}
        self.analysis = {}
        self.plots = []

    def find_nodes(self):
        """
            goes through the components dictionary and looks for all nodes
            and builds them into a list.
        """
        self.nodes = []

        for k, v in self.components.items():
            for node in v["nodes"]:
                if node not in self.nodes:
                    self.nodes.append(node)
            # Not sure this code is right... it seems ok to add an extra node
            # since it will generate another column/row for the indefinite matrix
            # but the name of the node does not seem correct, since it is not
            # technically a node.

            # if k.startswith(("F", "f", "H", "h")):
            #    if v["vcontrol"] not in self.nodes:
            #        self.nodes.append(v["vcontrol"])

        self.num_nodes = len(self.nodes)

    def find_ungrounded_nodes(self):
        """
            goes through the components dictionary and looks for all nodes
```

```
        and builds them into a list.
    """
    self.ungd_nodes = []

    for k, v in self.components.items():
        for node in v["nodes"]:
            if node not in self.ungd_nodes and node != "0":
                self.ungd_nodes.append(node)
        # Not sure this code is right... it seems ok to add an extra node
        # since it will generate another column/row for the indefinite matrix
        # but the name of the node does not seem correct, since it is not
        # technically a node.

        # if k.startswith(("F", "f", "H", "h")):
        #    if v["vcontrol"] not in self.nodes:
        #        self.nodes.append(v["vcontrol"])

    self.num_ungd_nodes = len(self.ungd_nodes)
    # print "ungd nodes", self.ungd_nodes
    # print "num ungd nodes", self.num_ungd_nodes

def find_freq(self):
    """
        Goes through the component dictionary to find a sinusoidal source
        and identifies the frequency it has.

        This works because the simulator only supports monotone HB analysis
        so there should be a single sinusoidal source.

        Generating the actual source vector (Vn) will be the responsbility
        of the HB solver
    """
    for k, v in self.components.items():
        if v["source"]:
            if v["sin"]:
                self.freq = v["freq"]

def find_ports(self):
    """
        Since there can be multiple non-linear devices in each port, there
        needs to be a way to handle this.

        The diode list will be different from the sources list in the
        following ways:

        - Indexes 0 and 1 are the nodes where the port is
        - Index 2 will be a dictionary of the devices connected to the port.

        To do this, we need to go through every non-linear device and
        check if the
        current port list already includes a port defined across the 2 nodes described.
        If there is, we add the device to the dictionary.

        We need to be careful, since the device can be connected in the opposite direction
        across the same nodes in the port, so we need to account for direction in the
        dictionary so we can adjust the sign of the current when generating
        the vector.
    """
    self.ports = []
    for k, v in self.components.items():
        if k.startswith(("D", "d")):
            self.ports.append(
                (v["nodes"][0], v["nodes"][1], "diode", v["parameters"]))
        elif k.startswith(("F", "f")):
            # We will treat current-controlled current-sources as if they were non-linear
            # elements from a matrix building standpoint.
            self.ports.append((v["nodes"][0], v["nodes"][1], "cccs", k))

    self.num_ports = len(self.ports)
```

```
def find_ports2(self):
    """
        Since there can be multiple non-linear devices in each port, there
        needs to be a way to handle this.

        The diode list will be different from the sources list in the
        following ways:

        - Indexes 0 and 1 are the nodes where the port is
        - Index 2 will be a list of dictionaries representing the devices present in the
          port.

        To do this, we need to go through every non-linear device and
        check if the
        current port list already includes a port defined across the 2 nodes described.
        If there is, we add the device to the dictionary.

        We need to be careful, since the device can be connected in the opposite direction
        across the same nodes in the port, so we need to account for direction in the
        dictionary so we can adjust the sign of the current when generating
        the vector.

        For devices such as a non-linear VCCS (used to model BJT), we will add the
        control nodes as tuple in the new_device dictionary.

    """
    self.ports = []
    for k, v in self.components.items():
        # Need to add other component types. We can add "G" since this would enable
        # BJT simulation.
        if k.startswith(("D", "d")) or k.startswith(("G", "g")):
            n0 = v["nodes"][0]
            n1 = v["nodes"][1]
            new_device = {
                "type": Circuit.comp_type_dict[
                    k[0].lower()],
                "parameters": v["parameters"],
                "inverted": False,
                "ctrl_nodes": v["ctrl_nodes"]}

            # Check if there already is a port between the two nodes
            port_exists = False
            for p in range(0, len(self.ports)):
                if (n0 == self.ports[p][0] and n1 == self.ports[p][1]):
                    port_exists = True
                    break
                elif (n1 == self.ports[p][0] and n0 == self.ports[p][1]):
                    port_exists = True
                    new_device["inverted"] = True
                    break

            if port_exists:
                self.ports[p][2].append(new_device)
            else:
                self.ports.append(
                    (v["nodes"][0], v["nodes"][1], [new_device]))
        # elif k.startswith(("F", "f")):
        #     # We will treat current-controlled current-sources as if they were non-linear
        #     # elements from a matrix building standpoint.
        #     self.ports.append((v["nodes"][0], v["nodes"][1], "cccs", k))

    self.num_ports = len(self.ports)

def find_sources(self):
    """
        Go through the components directory and identify every voltage sources.
        Depending on type (DC vs SIN), build the sources list.
    """
```

```
        self.sources = []
        for k, v in self.components.items():
            if k.startswith(("V", "v")):
                # We append the name of the sources as well since this will be
                # useful eventually
                if v["sin"]:  # For sin sources, we save offset and amplitude
                    self.sources.append(
                        (v["nodes"][0], v["nodes"][1], v["offset"], v["value"], k))
                else:  # For DC sources
                    self.sources.append(
                        (v["nodes"][0], v["nodes"][1], v["value"], 0, k))

        self.num_sources = len(self.sources)

    def verify_plotting(self):
        """
            1. we can only plot current for independent voltage sources
            2. we can only plot voltage for nodes that exist in the circuit
        """

        for plot in self.plots:
            currents = plot[1]
            voltages = plot[0]

            for current in currents:
                if current not in self.components.keys():
                    print "Element %s not found in the circuit, review your netlist" % current
                    sys.exit(-1)
                if self.components[current]["source"] == False:
                    print "Element %s is not a source. Currents can only be plotted for
independent sources" % current
                    sys.exit(-1)

            for voltage in voltages:
                if voltage not in self.nodes:
                    print "Node %s not found in the circuit, review your netlist" % voltage
                    sys.exit(-1)

    def find_port_from_nodes(self, nodes):
        """
            This method will take a pair of nodes and return which index of the
            ports list it corresponds to, as well as whether or not the order
            of the nodes is the inverse of what the port definition is.
        """
        (v_p, v_n) = nodes

        for port in range(0, len(self.ports)):
            if v_p == self.ports[port][0] and v_n == self.ports[port][1]:
                return (port, 1)
            if v_p == self.ports[port][1] and v_n == self.ports[port][0]:
                return (port, -1)

        # If the port is not found, we return none. Theoretically we should raise
        # an exception here.
        return (None, 1)
```

models.py

```
import math


class Diode(object):

    def __init__(self, parameters):
        """
```

```
            We make it each devices responsibility to parse the parameters
            string and understand what is needed.

            For the diode, the first word (i.e. anything before the first whitespace)
            will be considered the name. Anything after that will be used for
            parameter overrides, but that won't be implemented right now.
        """
        self.model_name = parameters.split()[0]

        if self.model_name is None:
            self.Is = 2.2e-11  # Isat
            self.n = 1.08  # ideality factor
        elif self.model_name == "Ideal_Diode":
            self.Is = 1e-12
            self.n = 1
        else:   # Default to an ideal diode anyway
            self.Is = 1e-12
            self.n = 1

    def get_current(self, V):
        current = []
        for v_t in V:
            if v_t > 18:
                # Return a huge value and prevent an overflow
                current.append(1e15)
            else:
                current.append(
                    self.Is * (math.exp(v_t.real / (self.n * 26e-3)) - 1))
        return current


class BJT(object):

    def __init__(self, model_name=None):
        self.model_name = model_name
        if self.model_name is None:
            self.Is = 2.2e-11  # Isat
            self.n = 1.08  # ideality factor
        elif self.model_name == "Ideal_Diode":
            self.Is = 1e-12
            self.n = 1


class NonLinearVCCS(object):

    def __init__(self, parameters):
        """
            Parameters will need to include the following:

            - Which nodes are used for the voltage reference
            - The direction of those nodes (in case we need to adjust the voltage
              sign to account for how things work)
            - Equation used - this will allow us to make this generic by having
              us use eval for this equation. This shifts the burden to the model
              description as opposed to the Python implementation of it.
        """
        self.Is = 200 * 1e-12
        self.n = 1

    def get_current(self, V):
        current = []
        for v_t in V:
            if v_t > 18:
                # Return a huge value and prevent an overflow
                current.append(1e15)
            else:
                current.append(
                    self.Is * (math.exp(v_t.real / (self.n * 26e-3)) - 1))
        return current
```

parser.py

```
import re
import os
import sys
import random
import circuit


class CircuitParser(object):
    """
        TODO: need to support subcircuits
        TODO: this returns an indefinite matrix - ZBB, will return a circuit object
        TODO: define some form of syntax in the spice file for hb simulation
        TODO: add better parsing for sinewave sources
        TODO: design a way to support models being defined in the netlist
        TODO: create a scalable way to have a component library <- this should mostly work
        TODO: get a few more models for diodes <- bleh
        TODO: support BJT's and FET's <- wip
        TODO: need to support current controlled current sources <- wip
    """

    def __init__(self, filename, model_lib=None):
        self.filename = filename
        self.model_lib = model_lib
        self.temp_file = None
        self.subckts = {}

    def handle_scaling(self, value, scaling):
        """
            This is a very basic function that is used to scale values based
            on the prefix used for the unit.
        """
        if scaling:
            if scaling.lower() == "g":
                value *= 1e9
            elif scaling.lower() == "meg":
                value *= 1e6
            elif scaling.lower() == "k":
                value *= 1000
            elif scaling.lower() == "m":
                value *= 0.001
            elif scaling.lower() == "u":
                value *= 1e-6
            elif scaling.lower() == "n":
                value *= 1e-9
            elif scaling.lower() == "p":
                value *= 1e-12
            elif scaling.lower() == "f":
                value *= 1e-15
        return value

    def load_models(self):
        """
            This method will look for the subcircuit definitions for
            models and load them into a dictionary that will replace any
            known subcircuits or models.

            Models for stuff like BJT's or MOSFET's are defined as subcircuits
            in separate library files to be able to re-use the parsing.
        """
        if self.model_lib is None:
            #model_lib_path = r"C:\circuitparser\model_lib"
            model_lib_path = r"/home/wajo/Documents/ITESO/MDE/Thesis/hb/model_lib"
```

```
            # Look in the default directory
        else:
            model_lib_path = self.model_lib

        if not os.path.isdir(model_lib_path):
            print "Error loading the model library"
            sys.exit(-1)

        for f in os.listdir(model_lib_path):
            if f.endswith(".lib"):
                # load any files that end with .lib and store the info in
                # a dictionary that allows us to replace complex components
                # with their equivalent subcircuits. This same behavior
                # is reused for subckts defined by the user.
                f = open(os.path.join(model_lib_path, f))
                self._load_subckt(f)
                f.close()

    def _load_subckt(self, f):
        """
            Common function used both to load model libraries (stored as
            subcircuits) as well as runtime loading of subcircuits from
            users netlists.
        """
        status = 0
        sbc_start = re.compile(r"^\.SUBCKT")
        sbc_end = re.compile(r"^\.ENDS")
        current_subckt = None

        for line in f.readlines():
            if sbc_start.match(line):  # Found subcircuit definition
                current_subckt = line.split()[1]
                sbc_nodes = line.split()[2:]
                if current_subckt not in self.subckts.keys():
                    self.subckts[current_subckt] = {
                        "nodes": sbc_nodes, "ckt": []}
                    continue
                else:
                    print "Duplicate subckt definition, ignoring..."
                    current_subckt = None

            if current_subckt is not None:
                if sbc_end.match(line):
                    # We're done reading the equivalente circuit for this
                    # subcircuit or model.
                    current_subckt = None
                else:
                    # We're currently just reading the line as-is, maybe
                    # we should change this to actually store the nodes separate
                    # from other parameters
                    self.subckts[current_subckt]["ckt"].append(line.strip())

        if current_subckt is not None:
            # We started a sub-circuit but never found the end
            print "Error parsing sub-circuit, exiting..."
            status = 1

        return status

    def add_subckts(self):
        """
            This method goes and looks for any subcircuit definitions
            in the netlist provided to the parser.

            If any are found, they are added to the definition list.

            TODO: need to generate unique names for internal nodes
            in subcircuits when generating the final netlist. This is
            probably something that will be done at subcircuit replacement
```

```
        time.

        We do not support nested subcircuit definitions right now.
    """
    try:
        f = open(self.filename, 'r')
    except:
        print "Error opening file"
        sys.exit(-1)

    status = self._load_subckt(f)
    f.close()
    if status:
        sys.exit(-1)

    f.close()

def replace_subcircuits(self):
    """
        The first thing we do is to replace subcircuits in the chip.
        the parser will look for BJT's, MESFET's, etc. and replace
        them in a temporary netlist file, so we can have a unified
        component parser.

        When we call this, we already have the subcircuit dictionary built
        up, so we need to do dynamic renaming of anything that requires
        a sub circuit, this involves renaming both the internal and
        the external nets in the subcircuit.
    """
    try:
        f = open(self.filename, 'r')
    except:
        print "Error opening file"
        sys.exit(-1)

    f2 = open("%s.temp" % self.filename, "w")
    bf = []

    renamed_nodes = {}
    renamed_components = {}

    for line in f.readlines():
        if line.startswith(("X", "Q", "M", "x", "q", "m")):
            # If we find a subckt, bjt or mosfet, we need to replace it
            # with its equivalent subcircuit.

            subnam = line.split()[-1]
            if subnam in self.subckts.keys():
                # We need to find the name of the nets used for the subckt
                # as they are called in the actual circuit.
                num_real_nodes = 3
                if line.startswith("X"):
                    num_real_nodes = len(self.subckts[subnam]["nodes"])
                real_nodes = line.split()[1:num_real_nodes + 1]

                # We need to do 4 things here:
                # 1. find nodes that are internal to the subcircuit
                #    and make sure they are unique
                # 2. find nodes that are external to the subcircuit
                #    and replace the names inside the subcircuit with
                #    the external ones.
                # 3. rename the components that are internal to the subckt
                #    since otherwise there will be duplicate names.
                # 4. update the references of dependent-sources since they
                #    will be refering to stuff that potentially will change names
                #    This still isn't done. Need to change how parsing is done, since
                #    we'll have to have renamed everything before we can do this.
                #    To accomplish this, we'll save a dictionary of every renamed
                #    node and component to use later.
```

```
                for l in self.subckts[subnam]["ckt"]:
                    l = l.split()

                    if l[0].startswith(("Q", "q", "M", "m")):
                        nets = 3
                    else:
                        nets = 2

                    # Rename internal components by adding the name of the
                    # subckt/component name at the end of each one, it should
                    # ensure unique names.
                    old_comp_name = l[0]
                    l[0] = "%s_%s" % (l[0], line.split()[0])
                    renamed_components[old_comp_name] = l[0]

                    for i in range(1, nets + 1):
                        if l[i] in self.subckts[subnam]["nodes"]:
                            # Replace external nodes with the circuit names
                            j = self.subckts[subnam]["nodes"].index(l[i])
                            # print "l[i]", l[i], "real_nodes[j]", real_nodes[j]
                            # I think this fixes the renaming issue
                            renamed_nodes[l[i]] = real_nodes[j]
                            l[i] = real_nodes[j]
                        else:
                            # Randomize names of internal nodes
                            if l[i] in renamed_nodes.keys():
                                l[i] = renamed_nodes[l[i]]
                            else:
                                old_name = l[i]
                                l[i] = "%s_%d" % (
                                    l[i], random.randint(0, 1e6))
                                renamed_nodes[old_name] = l[i]
                    bf.append(" ".join(l))
                    #f2.write(" ".join(l))
                    # f2.write("\n")
            else:
                # TODO: here we could have stuff like BJT's or mosfets
                # default to a basic model definition.
                # print subnam
                print "Error... did not find model definition"
                sys.exit(-1)
        else:
            # otherwise, we just copy the line the way it is
            bf.append(line)
            # f2.write(line)

    # We do special handling for dependent sources since renaming needs
    # to be complete to ensure we do things right.
    for line in bf:
        if line.startswith(("F", "f", "H", "h")):
            line = line.split()
            if line[3] in renamed_components.keys():
                line[3] = renamed_components[line[3]]
            line = " ".join(line)
        # TODO - fix this.. this does not work :( I believe it is fixed now
        if line.startswith(("E", "e", "G", "g")):
            line = line.split()
            if line[3] in renamed_nodes.keys():
                line[3] = renamed_nodes[line[3]]
            if line[4] in renamed_nodes.keys():
                line[4] = renamed_nodes[line[4]]
            line = " ".join(line)
        f2.write(line.strip())
        f2.write("\n")

    f.close()
    f2.close()

def parse(self):
```

```
        """
            Once sub-circuits have been replaced, the circuit will be
            parsed and a dictionary with all the components, including type,
            value, etc. will be built. This is the object that will be
            provided to the circuit simulator, it will take the dictionary
            (maybe a custom object?) and generate the indefinite matrix from it.
        """
        ckt = circuit.Circuit()

        # We first go through all the linear components
        res = re.compile(
            r"^([Rr]\w+) (\w+) (\w+) ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*[ohms]*")
        cap = re.compile(
            r"^([Cc]\w+) (\w+) (\w+) ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P)*[(F|f)]*")
        ind = re.compile(
            r"^([Ll]\w+) (\w+) (\w+) ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*[(H|h)]*")

        vcvs = re.compile(
            r"^([Ee]\w+) (\w+) (\w+) (\w+) (\w+) ([0-9e.-
]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*")
        vccs = re.compile(r"^([Gg]\w+) (\w+) (\w+) (\w+) (\w+) (.*)")
        ccvs = re.compile(
            r"^([Hh]\w+) (\w+) (\w+) (\w+) ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*")
        cccs = re.compile(
            r"^([Ff]\w+) (\w+) (\w+) (\w+) ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*")

        # Add standalone DC voltage sources
        vs = re.compile(
            r"^([Vv]\w+) (\w+) (\w+) DC ([0-9e.-]+)(G|g|K|k|M|m|Meg|meg|u|U|n|N|p|P|f|F)*[Vv]*")
        #vs = re.compile("^([Vv]\w+) (\w+) DC ([0-9e.-]+)")

        # Add sinusoidal voltage sources as well
        vsin = re.compile(
            r"^([Vv][\w]+) (\w+) (\w+)\s(sin|SIN)\(([0-9e.-]+)(M|m|u|U)*[Vv]* ([0-9e.-
]+)(M|m|u|U)*[Vv]* ([0-9e.-]+)(G|g|K|k|Meg|meg)*[HhzZ]*\)")
        #vsin = re.compile("^([Vv][\w]+) (\w+) (\w+)\s(sin|SIN)\(([0-9e.-]+)[Vv]*")

        # Add diodes - we currently don't support any other non-linear device right now
        # given that BJT's are modeled with 2 diodes and 2 linear CCCS
        # Maybe MOSFET modeling will require another non-linear device
        diode = re.compile(r"^([Dd][\w]+) (\w+) (\w+) (.*)")

        # Additionally, the analysis requested needs to be parsed. The
        # simulator will only support Harmonic-Balance, so we only check
        # for it, but in theory the parser could be a generic front-end
        # for any simulator.
        hb = re.compile(r"^\.HB (\d+) (\w+)\s?(\d+)?")
        #hb = re.compile("^\.PLOT (\w+) ()")

        plot = re.compile(r"^\.PLOT HB")

        try:
            f = open("%s.temp" % self.filename, "r")
        except:
            print "Error opening file"
            sys.exit(-1)

        for line in f.readlines():
            found_res = res.match(line)
            found_cap = cap.match(line)
            found_ind = ind.match(line)
            found_vcvs = vcvs.match(line)
            found_vccs = vccs.match(line)
            found_ccvs = ccvs.match(line)
            found_cccs = cccs.match(line)
            found_vs = vs.match(line)
            found_vsin = vsin.match(line)
            found_d = diode.match(line)
            found_hb = hb.match(line)
```

```
found_plot = plot.match(line)

# TODO: add an error if two or more regex are valid (which
# shouldn't happen)
comp = None
for v in [
        found_res,
        found_cap,
        found_ind,
        found_vcvs,
        found_ccvs,
        found_cccs,
        found_vs]:
    if v is not None:
        comp = v
        if found_ccvs or found_cccs:
            value = self.handle_scaling(
                float(comp.group(5)), comp.group(6))
        else:
            value = self.handle_scaling(
                float(comp.group(4)), comp.group(5))

if comp is not None:
    if found_res or found_cap or found_ind:
        ckt.components[
            comp.group(1)] = {
            "nodes": [
                comp.group(2),
                comp.group(3)],
            "value": value,
            "ctrl_nodes": None,
            "vcontrol": None,
            "source": False}

    if found_vcvs:
        ckt.components[
            comp.group(1)] = {
            "nodes": [
                comp.group(2),
                comp.group(3)],
            "value": value,
            "ctrl_nodes": (
                comp.group(4),
                comp.group(5)),
            "vcontrol": None,
            "source": False}

    if found_ccvs or found_cccs:
        ckt.components[
            comp.group(1)] = {
            "nodes": [
                comp.group(2),
                comp.group(3)],
            "value": value,
            "ctrl_nodes": None,
            "vcontrol": comp.group(4),
            "source": False}
    if found_vs:
        ckt.components[
            comp.group(1)] = {
            "nodes": [
                comp.group(2),
                comp.group(3)],
            "value": value,
            "ctrl_nodes": None,
            "vcontrol": None,
            "source": True,
            "sin": False}
```

```
# We do some special handling for our sinusoidal source, since we
# only have 1
if found_vsin:
    comp = found_vsin
    offset = self.handle_scaling(
        float(comp.group(5)), comp.group(6))
    amplitude = self.handle_scaling(
        float(comp.group(7)), comp.group(8))
    freq = self.handle_scaling(
        float(comp.group(9)), comp.group(10))
    ckt.components[
        comp.group(1)] = {
        "nodes": [
            comp.group(2),
            comp.group(3)],
        "value": amplitude,
        "ctrl_nodes": None,
        "vcontrol": None,
        "source": True,
        "sin": True,
        "offset": offset,
        "freq": freq}

if found_d:
    comp = found_d
    ckt.components[
        comp.group(1)] = {
        "nodes": [
            comp.group(2),
            comp.group(3)],
        "value": None,
        "ctrl_nodes": None,
        "vcontrol": None,
        "source": False,
        "parameters": comp.group(4)}

# We do some separate handling of vccs for BJT modeling.
if found_vccs:
    comp = found_vccs
    # print "found the thing"
    ckt.components[
        comp.group(1)] = {
        "nodes": [
            comp.group(2),
            comp.group(3)],
        "value": None,
        "ctrl_nodes": (
            comp.group(4),
            comp.group(5)),
        "vcontrol": None,
        "source": False,
        "parameters": comp.group(6)}

if found_hb:
    ckt.analysis["HB"] = {
        "K": int(
            found_hb.group(1)),
        "method": found_hb.group(2)}
    if found_hb.group(3):
        ckt.analysis["HB"]["threads"] = int(found_hb.group(3))
    else:
        ckt.analysis["HB"]["threads"] = 1

if found_plot:
    v_to_plot = re.findall(r"V\((\w+)\)", line)
    i_to_plot = re.findall(r"I\((\w+)\)", line)
    # Each plot statement should generate a new figure, so we store
    # each command in a list of tupples.
    ckt.plots.append((v_to_plot, i_to_plot))
```

```
                # print v_to_plot

        f.close()

        ckt.find_nodes()
        ckt.find_ungrounded_nodes()
        ckt.find_freq()
        # ckt.find_ports()
        ckt.find_ports2()
        ckt.find_sources()

        # We do checking for plotting here.
        ckt.verify_plotting()

        return ckt
```

plotter.py

```
import matplotlib.pyplot as plt
import numpy as np

hack_ylim = {
    'x': (-10, 10),
    '1': (-5, 15),
    '2': (0, 20),
    '3': (5, 25),
    '4': (10, 30),
    '5': (15, 35),
    '6': (20, 40),
    '7': (25, 45),
    '8': (30, 50)
}


class Plotter(object):

    def __init__(self, ckt, result):
        self.ckt = ckt
        self.result = result

    def plot(self):
        """
            We create one figure per plot requested.
        """

        for p in range(0, len(self.ckt.plots)):

            plt.figure(p + 1)
            c = np.zeros(self.ckt.num_ports *
                         (self.result.K + 1), dtype=complex)

            #num_subplots = len(self.ckt.plots[p][0]) + len(self.ckt.plots[p][1])
            num_subplots = 1
            i = 1

            for current in self.ckt.plots[p][1]:
                v_src = current
                # print "vsrc", v_src
                for src in range(0, len(self.ckt.sources)):
                    if self.ckt.sources[src][4] == v_src:
                        c = self.result.I_source[
                            (src * (self.result.K + 1)):(src * (self.result.K + 1)) +
self.result.K + 1]
                        # print c
                c = (self.result.K + 1) * np.fft.ifft(c)
```

```
            t = plt.subplot(num_subplots, 1, i)
            t.set_title("I(%s)" % v_src)
            t.plot(c)
            t.set_ylabel("Current (A)")
            t.grid(True)
            #i += 1

        for voltage in self.ckt.plots[p][0]:
            # print "voltage", voltage
            # Find the node
            v = self.result.X[self.ckt.ungd_nodes.index(voltage)]

            # print "length", len(v)

            v = (self.result.K + 1) * np.fft.ifft(v)
            t = plt.subplot(num_subplots, 1, i)
            t.set_title("V(%s)" % voltage)
            # t.set_ylim(hack_ylim[voltage])
            t.set_ylim(0, 40)
            t.plot(v)
            t.set_ylabel("Voltage (V)")
            t.grid(True)
            #i += 1

        plt.plot()
        plt.show()
```

result.py

```
class Result(object):

    def __init__(self, vn, vkwp, isrc, raw_res, K, X, F_v):
        self.Vn = vn
        self.V_kwp = vkwp
        self.I_source = isrc
        self.raw_res = raw_res
        self.K = K
        self.X = X
        self.F_v = F_v
```

test.py

```
import sys
import time
import numpy as np
from parser2 import CircuitParser
from hb import HBSolver
import matplotlib.pyplot as plt
from plotter import Plotter


if __name__ == "__main__":

    start_time = time.time()

    parser = CircuitParser(sys.argv[1])
    parser.load_models()
    parser.add_subckts()
    parser.replace_subcircuits()
    ckt = parser.parse()

    solver = HBSolver(ckt)
```

```
if len(sys.argv) > 2:
    solver.max_fev = int(sys.argv[2])

solver.create_pool()

print "setup time", time.time() - start_time

solver_result = solver.solve()
results = solver_result.raw_res

print "Harmonics:", ckt.analysis["HB"]["K"]
print "Function evaluations:", results.nfev
print "non-linear", solver.ig_time
print "linear", solver.other_time
print "total time:", time.time() - start_time

p = Plotter(ckt, solver_result)
p.plot()

error = sum([abs(f_k) for f_k in solver_result.F_v])
error2 = sum([abs(f_k) for f_k in solver.IG])

print "Relative error:", error / error2

vkwp_temp = []
# This is wrong now
for i in range(0, len(results.x), 2):
    vkwp_temp.append(complex(results.x[i], results.x[i + 1]))

print "vkwp", vkwp_temp
```

C. LIST OF INTERNAL RESEARCH REPORTS

[1] J. García-Bedoy-Torres and J.E. Rayas-Sánchez, "Massively parallel implementation of the harmonic balance algorithm – a review of the state of the art," Internal Report *CAECAS-12-15-R*, ITESO, Tlaquepaque, México, Jul. 2012.

[2] J. García-Bedoy-Torres and J.E. Rayas-Sánchez, "Yield optimization of high-frequency circuits exploiting a multi-threaded implementation," Internal Report *CAECAS-13-06-R*, ITESO, Tlaquepaque, México, Nov. 2013.

[3] J. García-Bedoy-Torres and J.E. Rayas-Sánchez, "Netlist parsing and related infrastructure for circuit simulation," Internal Report *CAECAS-15-03-R*, ITESO, Tlaquepaque, México, Apr. 2015.

[4] J. García-Bedoy-Torres and J.E. Rayas-Sánchez, "The harmonic balance algorithm and a Python implementation," Internal Report *CAECAS-15-05-R*, ITESO, Tlaquepaque, México, May. 2015.

[5] J. García-Bedoy-Torres and J.E. Rayas-Sánchez, "Paralellizing the harmonic balance algorithm in Python," Internal Report *CAECAS-16-03-R*, ITESO, Tlaquepaque, México, Mar. 2016.

References

[Bailey-68] E. M. Baily, *Steady state harmonic analysis of nonlinear networks*, Ph.D. Dissertation, Stanford University, Stanford, CA, 1968.

[Ditzel-10] D. R. Ditzel, "Dynamic translation for EPIC architectures" in *8th Annual IEEE/ACM international symposium on Code generation and optimization (CGO 2010)*, Toronto, Canada, Apr. 2010.

[eCircuitCenter-13] eCircuit Center, *SPICE Basics (2013),* Mar. 30, 2015, http://www.ecircuitcenter.com/Basics.htm

[Edahiro-09] M. Edahiro, "Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration", in *Design Automation Conference, 2009, ASP-DAC 2009. Asia and South Pacific*, pp. 230-233.

[El-Rabaie-88] S. El-Rabaie, V. F. Fusco, and C. Stewart, "Harmonic balance evaluation of nonlinear microwave circuits – a tutorial approach", *IEEE Transactions on Education*, vol. 31, no. 3, pp 181-192, Aug. 1988.

[Golub-12] G. H. Golub and J. M. Ortega, *Scientific Computing and Differential Equations*, Boston, MA: Academic Press, 2012.

[Golub-13] G. H. Golub, C. F. Van Loan, *Matrix Computations (Johns Hopkins Studies in the Mathematical Sciences)*, Baltimore, MD: The Johns Hopkins University Press, 2013.

[Goyvaerts-06] J. Goyvaerts, *Regular Expressions: The Complete Tutorial*, Lulu.com, 2006.

[Goyvaerts-12] J. Goyvaerts and S. Levithan, *Regular Expressions Cookbook*. Sebastopol, CA: O'Reilly Media, 2012.

[Hennessy-12] J. L. Hennessy, D. A. Patterson, *Computer Architecture: a Quantitative Approach*, San Francisco, CA: Morgan Kaufman, 2012.

[Hunter-15] J. Hunter, et al, (2015, Feb 16). *Matplotlib Release 1.4.3* [Online]. Available: http://matplotlib.org/Matplotlib.pdf

[Intel-04] Intel Research and Development, *Architecting the Era of Tera (2004),* Jan. 11, 2016, https://software.intel.com/sites/default/files/27/61/Tera_Era.pdf

[Intel-12a] Intel Corporation, *Frequently Asked Questions: Intel® Multi-Core Processor Architecture (2012),* Mar, 5, 2012, https://software.intel.com/en-us/articles/frequently-asked-questions-intel-multi-core-processor-architecture

[Intel-12b] Intel Corporation, *Moore's Law Inspures Intel Innovation (2012),* Jul. 4, 2012, http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html

[Kapre-09] N. Kapre, and A. DeHon, "Performance comparison of single-precision SPICE Model-Evaluation on FPGA, GPU, Cell, and multi-core processors", in *Field Programmable Logic and Applications, 2009. International Conference on*, pp. 65 - 72.

[Korn-00] G. A. Korn and T. M. Korn, *Mathematical Handbook for Scientists and Engineers*, Mineaola, NY: Dover Publications, 2000.

[Li-12] P. Li, and W. Dong, “Parallel preconditioned hierarchical harmonic balance for analog and RF circuit simulation” in *Advances in Analog Circuits*. Edited by Esteban Tlelo-Cuautle, 2012, pp. 111-130.

[Love-10] R. Love, *Linux Kernel Development*. Crawfordsville, IN: Addison-Wesley Professional, 2010.

[Maas-03] S. A. Maas, *Nonlinear Microwave and RF Circuits*, Norwood, MA: Artech House, 2003.

[Maringanti-09] A. Maringanti, V. Athavale, and S. Patkar, “Acceleration of conjugate gradient method for circuit simulation using CUDA”, in *High Performance Computing (HiPC), 2009 International Conference on*, pp. 438-444.

[Meng-11] J. Meng, “Multi-core accelerated harmonic balance method for multi-tone full chip RFIC simulation”, in *Microwave Symposium Digest (MTT), 2011 IEEE MTT-S International*, pp. 1-4

[Munshi-11] A. Munshi, B. R, Gaster, T. G. Mattson, J. Fung, D. Ginsburg, *OpenCL Programming Guide*, Ann Arbor, MI: Addison-Wesley, 2011.

[Najm-10] F. N. Najm, *Circuit Simulation*, Hoboken, NJ: Wiley, 2010.

[Nakhla-76] M. Nakhla and J. Vlach, “A piecewise harmonic balance technique for determination of periodic response of nonlinear systems,” *IEEE Trans. Circuits and Systems*, vol. 23, no. 2, pp. 85-91, Feb. 1976.

[Patterson-12] D. A. Patterson, J. L. Hennessy, *Computer Organization and Design, Fourth edition: The Hardware/Software Interface*. San Francisco, CA: Morgan Kaufman, 2012.

[Queiroz-95] A. C. M. de Queiroz, “Compact nodal analysis with controlled sources modeled by ideal operational amplifiers”, in *Proc. Midwest Symp. Circuits and Systems*, Rio de Janeiro, Brazil, Aug. 1995, pp. 1205-1208.

[Ramanathan-06] R. M. Ramanathan, “White Paper: Intel Multi-Core Processors – Making the Move to Quad-Core and Beyond”, Intel Corporation, 2006.

[Ross-08] P. E. Ross, "Why CPU Frequency Stalled," in *IEEE Spectrum*, vol. 45, no. 4, pp. 72-72, April 2008.

[Topaloglu-11] R. Topaloglu, “GPU programming for EDA with OpenCL”, in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, Milpitas, CA, USA, pp. 63-66

[Vladimirescu-81] A. Vladimirescu, et al, (Aug. 1981). *SPICE Version 2G User’s Guide*, University of California, Berkeley, CA, [Online] Available: http://www.ece.umn.edu/~harjani/courses/common/spice2G6.pdf

[Zhang-11] Y. Zhang, “Logic simulation acceleration based on GPU” in *Mixed Design of Integrated Circuits and Systems (MIXDES), 2011 Proceedings of the 18th International Conference*,

Index