FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Radek Zikmund

NextGen SPICE – Electrical Circuit Simulation Library for .NET

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Pavel Ježek, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2018

Title: NextGen SPICE – Electrical Circuit Simulation Library for .NET

Author: Radek Zikmund

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The goal of this thesis was to create an extensible library for simulating electrical circuits for the .NET platform, which could be used in broad contexts like development of educational programs or applications that use evolutionary algorithms to evolve electrical circuits. Our library is inspired by the family of SPICE programs developed at University of California, Berkeley.

Initial version of our library implements the transient analysis of electrical circuits and supports basic devices like voltage and current sources, resistors, capacitors, inductors, but also semiconductor diode and BJT transistor devices. Our library is designed in such a way that both new circuit devices and circuit analyses can be added in future versions.

Other features of our library include importing circuits or their parts from the industry standard SPICE netlists and ability to modify circuit parameters during the simulation. In this thesis, we also investigate using double-double precision type to improve convergence during the simulation.

We also implement a simple SPICE-like console application to allow our simulation library to be used from command line.

Keywords: Electrical circuit simulation SPICE .NET Library

Název práce: NextGen SPICE – knihovna pro simulaci elektrických obvodů pro .NET

Autor: Radek Zikmund

Katedra: Department of Distributed and Dependable Systems

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce bylo vytvořit rozšiřitelnou knihovnu pro simulaci elektrických obvodů pro platformu .NET, která by byla uplatnitelná v širokém kontextu, jako je vývoj výukových programů nebo aplikací využívajících evolučních algoritmů pro evoluci elektrických obvodů. Naše knihovna je inspirována rodinou SPICE programů vyvíjených na Kalifornské univerzitě v Berkeley.

Počáteční verze naší knihovny implementuje transientní analýzu elektrických obvodů a podporuje základní součástky jako zdroje napětí a proudu, rezistory, kondenzátory a cívky, ale také polovodičové diody a BJT transistory. Naše knihovna je navržena takovým způsobem, že je možné v budoucích verzích knihovny přidat jak nové součástky, tak nové typy analýz.

Další vlastnosti naší knihovny zahrnují importování obvodů nebo jejich částí v průmyslově standardním SPICE netlist formátu a možnost upravovat parametry součástek během simulace. V této práci také prověřujeme použití typů s přesností double-double pro zlepšení konvergence během simulace.

Také jsme implementovali jednoduchou SPICE-like konzolovou aplikaci abychom umožnili používání naší simulační knihovny z příkazové řádky.

Klíčová slova: Simulace elektrických obvodů SPICE .NET Knihovna

I would like to thank my supervisor Mgr. Pavel Ježek, Ph.D. for his time, patience and the advice he gave me during our consultations. Also, special thanks go to my family, friends and everybody who supported me and helped me during my studies.

Contents

1. Introduction

The process of designing electrical circuits consists of several stages, starting with detailed specification from the customer that provides all necessary requirements, and ending with a working prototype of the final product. Intermediate stages include prototyping circuits using a construction base commonly referred to as breadboard, which is is often very slow and for complex integrated circuits even impossible. The task has been made easier with the invention of electrical circuit simulation programs, which allow quick circuit prototyping without the need of soldering iron.

1.1 The Berkeley SPICE

One of the most successful circuit simulators is the SPICE program[1] developed at EECS Department of the University of California, Berkeley. The original SPICE1 program is implemented in FØRTRAN language and was released in 1971. Its popularity quickly rose and few years later, Berkeley released SPICE2 with many performance improvements and model enhancements. SPICE programs developed at Berkeley heavily influenced development of future circuit simulation software, and we will describe them in more depth in following sections.

Netlists

Early versions of SPICE did not operate in interactive mode, therefore the input files contained both data and instructions for processing. These input files conventionally have `.cir` extension, and the contained circuit description is called *netlist*. A simple example of netlist is shown on the figure 1.1 on the left, with a schematic of corresponding circuit on the right. The meaning of individual lines of the netlist is explained in the next paragraph.

```
 1:  BRIDGE-T CIRCUIT
 2:  *
 3:  VBIAS 1 0 12
 4:  R1 1 2 10
 5:  R2 2 0 10
 6:  R3 2 3 5
 7:  R4 1 3 5
 8:  *
 9:  .OP
10:  .END
```

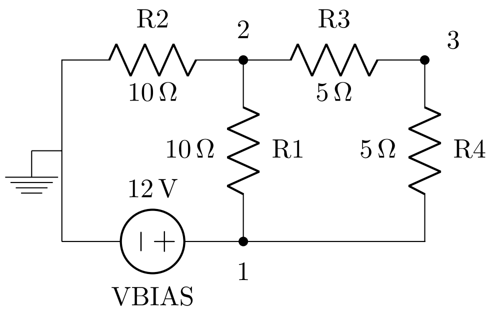


Figure 1.1: Example `.cir` netlist file, reproduced from The SPICE Book [1] and corresponding circuit schematic

The netlist in figure 1.1 is divided into three sections, separated by empty comment on lines 2 and 8. The first section of the netlist contains the name of circuit, then follows a second section with definitions of five devices: one voltage

[1]The name SPICE stands for Simulation Program with Integrated Circuit Emphasis.

source and four resistors. The type of each device is inferred from the first letter of its name. Each definition contains list of connected nodes and value of source voltage or resistor's ohmic resistance, respectively. Node 0 is special, because it corresponds to the ground and must be present in every circuit (detailed description of SPICE netlists syntax and rules for validating the circuit will be presented later in chapter 2). In the last section of the file, there is an `.OP` statement, which instructs the simulator to perform an operating point analysis to find stable values of node voltages of the circuit, and `.END` statement denoting the end of netlist. This syntax for describing electrical circuits became an industry standard during the 1970s, and most modern circuit simulators still recognize it.

When SPICE2 is run with the previously shown netlist file, it first reads all components, checks syntax and topology rules for the circuit, and then runs the operating point analysis – computes node voltages and current through the `VBIAS` voltage source. After that, it prints a rather verbose report, as shown in figure 1.2. First, description of the input circuit is repeated back to the user (only part of the description is shown in the figure for brevity, in top half in grey), then follows a list of node voltages and then currents flowing through voltage sources.

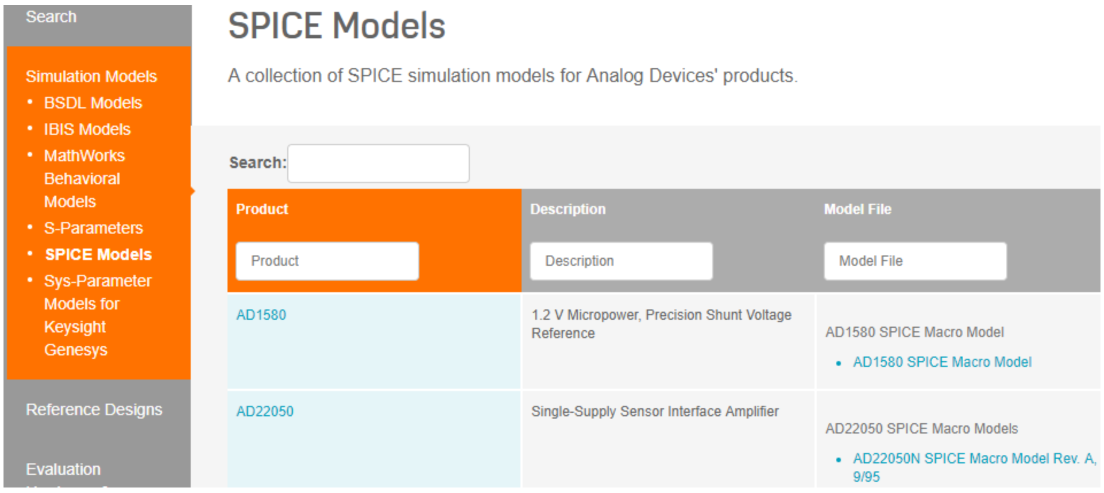
```
******* 03/19/91 ********* SPICE 2G.6  9/21/84 ********* 06:47:36 *********

BRIDGE-T CIRCUIT

 ****     CIRCUIT DESCRIPTION

****************************************************************************
*
VBIAS 1 0 12
...
.END

******* 03/19/91 ********* SPICE 2G.6  9/21/84 ********* 06:47:36 *********

BRIDGE-T CIRCUIT

 ****  SMALL SIGNAL BIAS SOLUTION   TEMPERATURE = 27.000 DEG C

****************************************************************************

 NODE    VOLTAGE     NODE    VOLTAGE     NODE    VOLTAGE    NODE   VOLTAGE

(    1)    12.0000  (    2)     8.0000  (    3)    10.0000

    VOLTAGE SOURCE CURRENTS
    NAME         CURRENT

    VBIAS       -8.000E-01

    TOTAL POWER DISSIPATION 9.60E+00 WATTS
```

Figure 1.2: Output of SPICE2G.6 for the example netlist file, from The SPICE Book [1]

Macromodels

One of the most useful features of SPICE is the ability to define custom subcircuits, called *macromodels*, composed from devices already integrated in the simulator, or other macromodels. Circuits can be then decomposed to individual subcircuits, similarly to how a computer program's source code can be decomposed into individual functions. One macromodel can then represent a complex real-life device, e.g. an amplifier, and can be simply reused throughout the whole circuit.

This allows device manufacturers to provide a SPICE netlists with macromodels that accurately model their devices. An example of such manufacturer is Analog Devices, macromodels for their products can be downloaded from their webpage, screenshot of which is shown in figure 1.3. The manufacturer's customers can then use these macromodels in their simulators to model behavior of circuit that uses the manufacturer's products.



SPICE Models

A collection of SPICE simulation models for Analog Devices' products.

Product	Description	Model File
AD1580	1.2 V Micropower, Precision Shunt Voltage Reference	AD1580 SPICE Macro Model • AD1580 SPICE Macro Model
AD22050	Single-Supply Sensor Interface Amplifier	AD22050 SPICE Macro Models • AD22050N SPICE Macro Model Rev. A, 9/95

Figure 1.3: Website of Analog Devices, where SPICE macromodels can be downloaded.

Today, vast libraries containing hundreds of SPICE netlists with macromodels exist, and therefore it is a very important feature of any modern circuit simulator to be able to import macromodels from these netlists.

Manufacturer-supplied macromodels are often very complex and the netlists are too long to be shown here. Instead, the use of macromodels is demonstrated on a simple macromodel for AC coupled amplifier. The netlist description is shown in figure 1.4. Very important part of the definition is the opening `.SUBCKT` statement, which denotes start of the subcircuit description, and specifies its name and terminal nodes which will serve as an interface to the outer circuit. After that follows description of the devices that constitute the macromodel, and finally, the subcircuit definition is ended by `.ENDS` statement. The subcircuit in the figure is named ACamplifier and terminal nodes are 1, 2 and 3. Other nodes (excluding `0`, which is global ground node) and all devices used between the `.SUBCKT` and `.ENDS` statement are strictly local to the subcircuit and are not visible to the outside circuit.

```
* external nodes:  in power out
.SUBCKT  ACamplifier 2 1 3
R1 1 4 2K
R2 4 0 500
C1 2 4 10n
Q1 3 4 5 2N2222
Rc 1 3 2K
Re 5 0 1e3
.MODEL 2N2222 NPN
+ (BF=50 IS=1E-13 VBF=50)
.ENDS
```

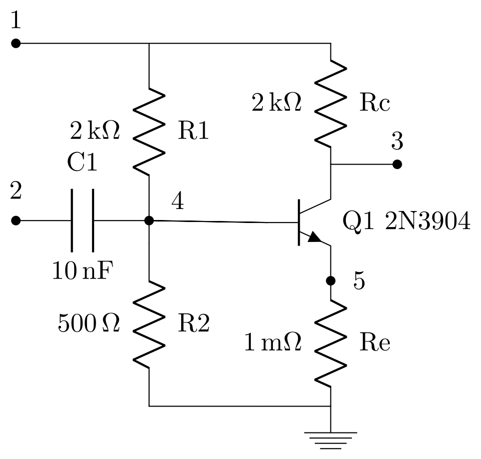


Figure 1.4: Simple macromodel example for AC coupled amplifier [2], and corresponding circuit schematic, adapted from a 5Spice tutorial by Richard P. Andresen

Such a macromodel can then be used by providing nodes for terminal connections. In SPICE netlist, a macromodel is represented by a device with name staring with an X. The actual macromodel to be used is specified as the last argument in the device statement. An example of a circuit that uses the ACamplifier macromodel is shown in figure 1.5. The macromodel definition is inlcuded from a separate file, similarly to the `#include` preprocessor directive in C or C++. In the actual circuit, the macromodel is represented by the `XAMP` device. Nodes 3, 2 and 1 are mapped onto the nodes 2, 1 and 3 from the macromodel description.

```
SUBCIRCUIT CALL EXAMPLE
*
.INCLUDE acamplifier.cir
*
V1 2 0 5
R1 2 3 10
XAMP 3 2 1 ACamplifier
R2 3 0 20
*
.END
```

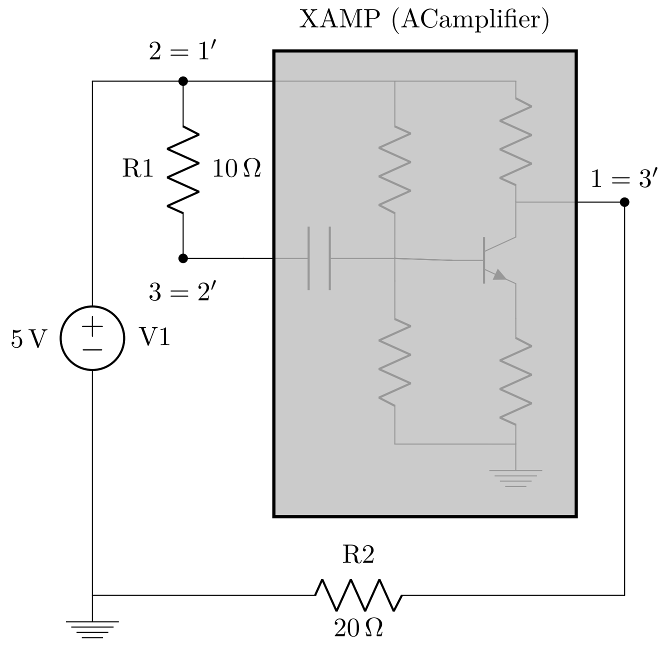


Figure 1.5: Illustrative example of circuit that uses macromodel from figure 1.4

From SPICE2 to SPICE3

SPICE2 was not the last SPICE program developed by Berkeley. With the increasingly more popular UNIX-based operating systems, it was possible for programs to be more interactive. Andrei Vladimirescu states in The SPICE Book [1] that SPICE2 was '*a FORTRAN batch program and was difficult to modify and limited in its potential use of C-shell utilities*'. These limitations led Berkeley to start development of SPICE3 in the C programming language during the 1980s. In addition to more detailed models and improved numerical accuracy, SPICE3 was to support interactive mode, which allowed separating circuit description and commands for requesting circuit analysis.

Unfortunately, the development was in the end left to a handful of students due to limited financial resources. The first release of SPICE3 was very buggy and was not backward compatible with SPICE2. This was a big problem, because hundreds of commonly used macro-models would have to be rewritten before they could be used in SPICE3. Even though most incompatibility issues were fixed in later releases, SPICE3 did not completely replace SPICE2 and both coexist as two standards for circuit simulations, with the SPICE2 one being subset of SPICE3 and therefore more portable.

1.2 Present-day Situation

SPICE programs developed at Berkeley strongly influenced the development of circuit simulators used in industry. Because of the vast existing libraries of SPICE netlist files for various circuits, most circuit simulators either use the syntax which is a superset of the one used in SPICE, or provide some other way of importing circuits from the standard SPICE format.[2] There is even a categorization of circuit simulators based on the backward compatibility with Berkeley SPICE programs. Ron Kielkowski summarizes this in Inside SPICE [3] as follows:

> *Of all the analog circuit simulation tools available, the overwhelming majority of them are SPICE-like or SPICE-compatible. SPICE-like means a simulator is capable of producing an analysis result similar to the SPICE result for a given circuit, although they many not be able to read a standard SPICE circuit. SPICE-compatible means a simulator can read a SPICE circuit file and produce the result in standard SPICE2G.6 form.*[3]

This only reinforces the idea that the backward compatibility with the original SPICE programs is an important feature of circuit simulators.

Today's circuit simulating programs commonly include a graphical tool for editing circuits and plotting the simulation results. Instead of writing netlist files by hand, circuits are edited using drag-and-drop operations. One such program is LTspice [4], whose graphical user interface is shown in figure 1.6.

[2]Example of a simulator which does not support SPICE syntax directly is QUCS simulator, which provides a tool for transforming the netlist in SPICE format into the QUCS format.

[3]Result shown in figure 1.2 is an example of such standardized output.

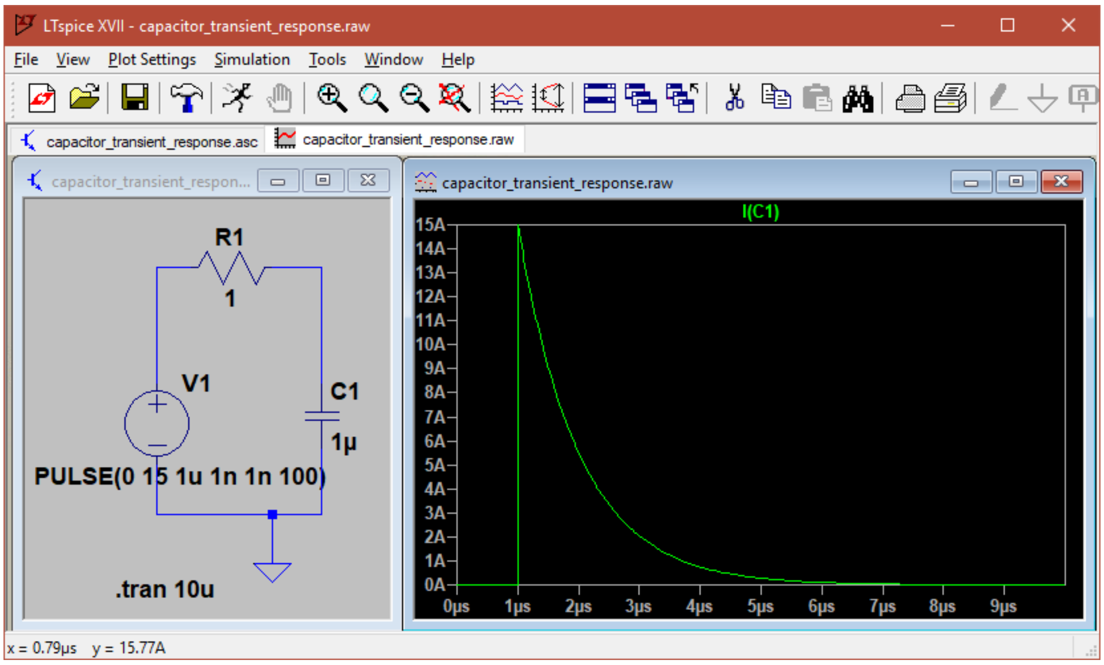

LTspice XVII - capacitor_transient_response.raw
File View Plot Settings Simulation Tools Window Help
capacitor_transient_response.asc
capacitor_transient_response.raw
R1
1
V1
C1
1µ
PULSE(0 15 1u 1n 1n 100)
.tran 10u
I(C1)
15A 14A 13A 12A 11A 10A 9A 8A 7A 6A 5A 4A 3A 2A 1A 0A
0µs 1µs 2µs 3µs 4µs 5µs 6µs 7µs 8µs 9µs
x = 0.79µs y = 15.77A

Figure 1.6: Graphical user interface of LTspice

1.3 Use of Circuit Simulators Outside Industry

Berkeley SPICE programs were designed from the start to be used as an aid for integrated circuit designers. The same can be said about the SPICE's successors that are used today. However, thanks to significant increase in computational power, it is now possible to use circuit simulators in other contexts too. There is an ongoing research on use of evolutionary algorithms to evolve circuit parameters and even circuit topology. For example, in 2016, Rojec et al. successfully evolved a passive low-pass filter [5]. Also, computers and other interactive equipment are often used for educational purposes in schools, so a circuit simulator can be used as part of an educational program intended for high school or university students.

We have in mind creating such applications, and their development would be greatly simplified if there was a suitable circuit simulation library. We would essentially like the simulator to allow what we will call a *live simulation*. For example, in a potential educational program, we would like to allow user (student) to e.g. flip switches or manipulate parameters of individual devices (e.g. resistances on the resistors), and hence provide an interactive experience. Also, circuit evolution applications could also make use of the possibility to simply manipulate circuit parameters. The first step of development of these applications would be finding and preparing such simulation library.

Since such applications would be used primarily for academic or educational purposes, it is desired that these programs be easy to develop, maintain, and – in case of educational applications – support multiple platforms. For these reasons it is probable that these applications would not be developed in some low level language like C++, but rather a more higher level language. One such language, C#, is part of the .NET developer platform, which is widely used in desktop application development. In recent years, .NET expanded to other platforms as well and would be therefore our choice in development of said applications.

As we have written earlier, SPICE programs and their descendants were designed mainly to be an aid for electrical engineers. As a consequence, they can only perform off-line simulation, where the simulated period needs to be set explicitly before the simulation starts. During our research we did not find any simulator which would allow the simulation to be continued where it left off.[4] Furthermore, most simulators do not offer binary API for other programs to use, which means that the communication with the simulator needs to be done through the SPICE netlist files. One simulator which stands out is ngspice [6], which exposes a set of functions that allow manipulation of the simulator from an external program. The ngspice API does not require the input file to be written on disk, but we are still required to convert a simulated circuit into the netlist description, which is then passed to ngspice using a `char**` pointer.[5]

Existing simulator programs therefore are not suitable for our purpose, we should search among the existing simulator libraries. At the time of assignment of this thesis, there was no implementation of a circuit simulation library for .NET Framework.[6] There are libraries for Python, for example PySpice [9] and PyOPUS [10] – but these two libraries rely on standalone simulators (namely ngspice [6] and HSPICE [11]) to do the simulation and therefore share the same limitations. There is also JSpice [12] library for Java, which implements it's own simulation engine, but just like other simulators requires the user to specify simulation duration beforehand.

Possible solution to the problems of using present-date simulator programs or libraries is rewriting an existing circuit simulator in .NET Framework and make necessary changes to the interface for our purpose. Berkley SPICE3 is still considered a reference simulator program, and since it is open source and freely available on the website of University of California, Berkeley [13], we will examine the possibility of rewriting it in .NET in the next subsection.

Rewriting SPICE3 for .NET, a Viable Option?

The SPICE3 was written in now non-standard C K&R norm, because the development started years before the first ANSI norm was released in 1990. Due to the nature of the C language, many parts of the simulator are very fragile and hard to maintain from today's perspective. Consider snippet in figure 1.7 taken from the last SPICE3 release (version 3f.5 from 1993). It contains a function which loads instances of voltage controlled current source devices to the circuit equation matrix. This function is just one example from the set of functions that must exist for each device in SPICE3 implementation. Other such functions include methods for model updating and data printing.

There are many points worth mentioning. In the declaration of the `VCCSload` function, the function parameters are specified by name only (line 3). The type of each parameter is then specified separately (lines 4–5). This is the main

[4]For simple circuits, this can be achieved by setting the initial conditions of the circuit devices to be equal to the state of the circuit at the end of previous simulation, and recomputing the parameters for input sources (like phase offset and pulse delay). However, it is impossible to get or let alone set state of a device inside a subcircuit (the X device).

[5]For more details, see section 19.3 (Shared ngspice API) in ngspice user manual [7].

[6]At the time of writing, there is the SpiceSharp [8] library, whose development started shortly after that of NextGen SPICE.

```
 1: /*ARGSUSED*/
 2: int
 3: VCCSload(inModel,ckt)
 4:     GENmodel *inModel;
 5:     CKTcircuit *ckt;
 6:         /* actually load the current values into the
 7:          * sparse matrix previously provided
 8:          */
 9: {
10:     register VCCSmodel *model = (VCCSmodel *)inModel;
11:     register VCCSinstance *here;
12:
13:     /*  loop through all the source models */
14:     for( ; model != NULL; model = model->VCCSnextModel ) {
15:
16:         /* loop through all the instances of the model */
17:         for (here = model->VCCSinstances; here != NULL ;
18:         here=here->VCCSnextInstance) {
19:
20:             *(here->VCCSposContPosptr) += here->VCCScoeff ;
21:             *(here->VCCSposContNegptr) -= here->VCCScoeff ;
22:             *(here->VCCSnegContPosptr) -= here->VCCScoeff ;
23:             *(here->VCCSnegContNegptr) += here->VCCScoeff ;
24:         }
25:     }
26:     return(OK);
27: }
```

Figure 1.7: Code snippet from `spice3f5/src/lib/dev/vccs/vccsload.c`

characteristic of the K&R C. In the ANSI C, equivalent declaration would be `VCCSload(GENmodel *inModel, CKTcircuit *ckt)`.

Next thing to notice is the pointer casting on line 10 – the `inModel` parameter is cast to pointer to the concrete type of the device that is being loaded. Practices like this are very common in C code due to the lack of higher-level language features like inheritance and polymorphism. Also, because no type checking occurs in C during pointer casting, it can be a source of hard-to-debug errors if the target object is of different type.

Lines 10 and 11 also include the `register` keyword, which used to be a hint for the compiler to store the variable in a CPU register for faster access. This keyword is now deprecated, because modern compilers can do a better job at assigning variables to registers than a human programmer.

The last point worth noticing is the usage of pointers when accessing the equation matrix (lines 20 to 23). C# does allow usage of pointers in so called *unsafe code block*, but they affect the performance of garbage collector, and should be used carefully.

Although it is possible to extend SPICE3 with new circuit devices by following instructions from Thomas Quarles, author of SPICE3 [14], adding new analysis types requires modifying method tables for all existing devices and other crucial parts of the simulator, and thus is time consuming and error-prone.

SPICE3 code also makes heavy use of `#define` `#ifdef` and other preprocessor directives which make the source code less readable. In combination with long functions and scarce source code documentation available (for an example, see file `spice3f5/src/lib/ckt/dctran.c`), rewriting the simulator is a very hard task requiring in-depth analysis and understanding of the simulator internals.

Overall, the programming style used in SPICE3 implementation is very different from the style used in modern object oriented programming languages like C#, and its code is not suitable for simple one-to-one translation from C to C#.

Main thesis goal

Because of reasons listed so far in this chapter, computer programs which would as part of their functionality perform live simulation of electrical circuits would have to implement their own simulator engine. As the primary goal of this thesis, we would like to simplify development of such applications by implementing such circuit simulator from the ground up in the form of portable .NET library. We will specify our requirements on the library in the next section.

1.4 Reimplementation of SPICE for .NET

Modern circuit simulators can do many types of circuit analyses and support many types of circuit devices. Because implementing range of functionality comparable to these programs would be out of scope of this thesis, we have selected a reasonable subset to be implemented, which is described in following subsection in greater detail. However we would like to be able to implement other features in the future without affecting user's code. We would therefore have to design the library to be appropriately extensible.

Requirements on the Library

We would like the library to be usable in broad contexts. Since .NET is now supported on many platforms, including Windows, Linux, and even mobile devices (Android, iOS and Windows Mobile), the library should be targeted to .NET Standard to make it maximally portable and usable on any .NET platform.

Our primary goal in the library will be supporting live simulation, as we described earlier. This essentially requires implementing equivalent of *transient analysis* of the SPICE simulator. However, the simulator should perform the individual timesteps on demand and allow making reasonable changes to the circuit devices like flipping switches, changing resistances, changing values of voltage and current sources.

The transient analysis requires so called *large-signal* models of the simulated devices, which are also used by other types of SPICE-like analyses, like DC Sweep Analysis. DC Sweep analysis calculates the circuit states for range of values for a certain circuit parameter (like resistance of a resistor or value of voltage/current source). Because we already wish to support changing parameters between individual timesteps, our library will also support DC Sweep analysis.

We would like to design our library so that it could potentially support the same set of devices as the SPICE simulators. However, the implementation of all SPICE devices is quite complex and would be out of scope of this thesis. We have therefore decided to implement the basic devices like ideal resistor, voltage or current sources, capacitor and inductor. Also, to demonstrate that even the complex semiconductor devices can be implemented, we will implement diode and BJT transistor devices as well. These two devices are reasonably simple to implement and at the same time complex enough to demonstrate the capabilities of the simulator. Because SPICE is considered the reference circuit simulator implementation, we would implement these devices using the same mathematical models that are used in the SPICE simulators.

In the future, we would also like to add other types of circuit analyses – such as AC frequency sweep analysis – and new devices, and even allow users of our library to implement their own. The library should be therefore extensible without modifying the core library's source code.

To provide academic researchers a way to test newly developed models and computational techniques, the library should be widely configurable, and preferably open-source, to allow its users to contribute to the library's development.

Also, since there already exist vast libraries of spice circuits and subcircuits with macromodels described in the industry-standard SPICE netlist syntax, we would like to our library to support importing circuits from SPICE netlists. We will therefore provide SPICE netlist parser as part of our library.

Since we have to recognize great portion of SPICE netlist syntax in order to parse macromodel descriptions, with little additional work we could implement the parser to recognize control statements for requesting individual circuit analyses (like the `.OP` statement we saw in figure 1.1). This would allow us to create a console application similar to the original SPICE and therefore allow capabilities of the library to be used in a standalone fashion using the SPICE netlist syntax. Since .NET Standard cannot be used to develop console applications, we require that the console application requires the next smallest set of API: .NET Core. Figure 1.8 shows the expected relationship between the simulator, console application and other programs that would use the simulator – parts in blue will be implemented as part of this thesis.

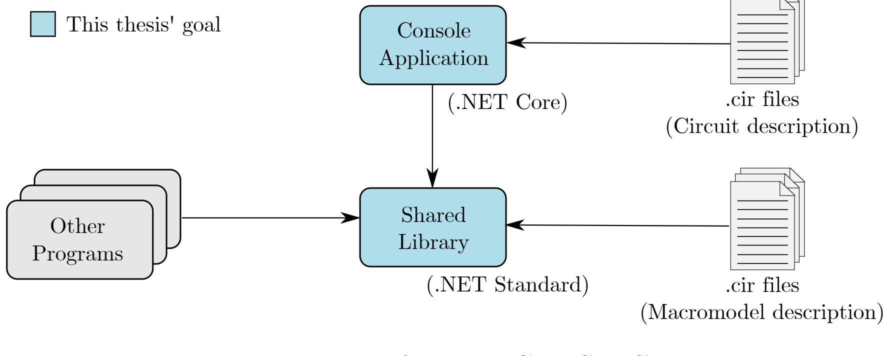


Figure 1.8: Dependency diagram for NextGen SPICE and other programs

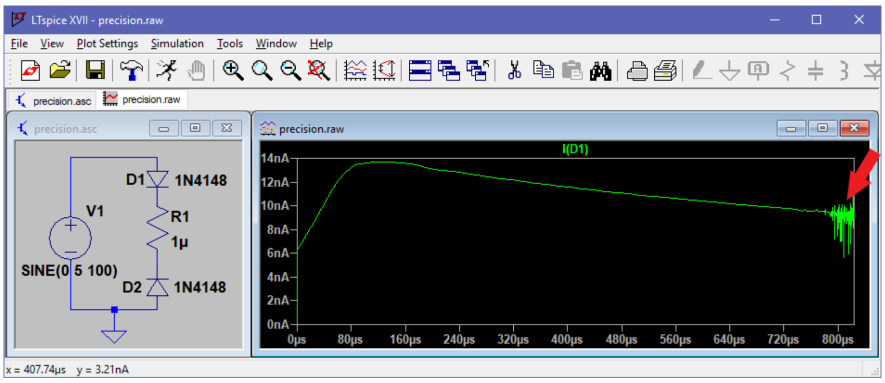
Nonconvergence Due to Low Precision

Implementing any kind of simulation software means choosing an appropriate mathematical model for given problem domain and, subsequently, suitable representation of the problem in computer memory. Representing real numbers is integral part of every physics simulation engine.

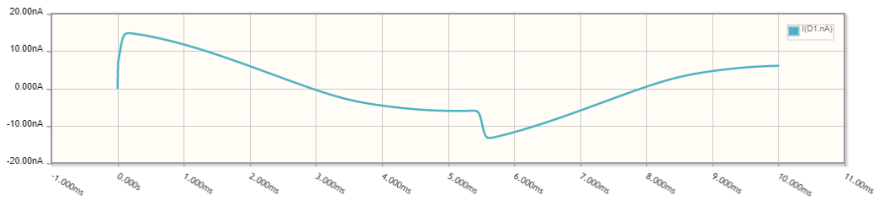
To achieve fast simulations, developers have to choose between representations having hardware support on the target platforms, which leaves them with IEEE 32-bit (`single`) and 64-bit (`double`) floating-point types.

Most of the common circuit simulators use type `double` with approximately 15 decimal digits precision. Due to the large dynamic range of the circuit variables, this leads to significant truncation errors when equation system coefficients differ in more than 15 orders of magnitude.

Coefficient differences of this magnitude may commonly occur when resistors with very small resistance values are used. Mike Robbins, one of the authors of online circuit simulator CircuitLab, provides a simple example of such ill-conditioned circuit in an article on their website [15]. This circuit along with the (partial) result plot from LTspice can be found in the upper part of figure 1.9. Notice the noise at the end of the simulated period emphasized by the red arrow. The lower part contains result of the same circuit in CircuitLab simulator.[7]


LTspice XVII - precision.raw
File View Plot Settings Simulation Tools Window Help
precision.asc
precision.raw
D1 1N4148
V1
R1
1µ
SINE(0 5 100)
D2 1N4148
I(D1)
14nA
12nA
10nA
8nA
6nA
4nA
2nA
0nA
0µs 80µs 160µs 240µs 320µs 400µs 480µs 560µs 640µs 720µs 800µs
x = 407.74µs y = 3.21nA

LTspice



CircuitLab

Figure 1.9: Example ill-conditioned circuit, and example results form LTspice (top, incomplete), and CircuitLab (bottom), adapted from [15]

The noise in the plot on the figure is caused by the $1\mu\Omega$ resistor in the circuit. In the comment section under the article, Robbins writes that such small resistors

[7]Keen reader might notice differences between the plot of CircuitLabs simulator results shown in this thesis and the one in the referenced article. These are due to the fact that CircuitLab uses slightly different model parameters for 1N4148 diode than LTspice. Plot shown in this thesis was obtained using the model parameters extracted from LTspice model library.

are not physical, meaning that they do not correspond to physical resistor devices, but can be produced during automated macromodel construction. Use of this small resistor led to too-big differences between the equation coefficients, which in turn led to significant truncation errors and produced noise seen in the plot. This noise later leads to nonconvergence of the equation solution. Nonconvergence of the solution is a rather technical issue, and essentially means that the simulator cannot determine the state of the circuit after the next timestep.[8] In this case, the nonconvergence is caused by low precision of the representation of real numbers.

Such noise can be eliminated by using more precise real number representation. One such option is using another type defined by IEEE 754 standard, namely 80-bit or even 128-bit floating point number formats. However, neither of these is commonly available on today's hardware, let alone in .NET runtime. If we would decide to use one of these two formats, every operation on such numbers would have to be emulated in software, which would greatly slow down the simulator.

Another option would be using .NET type `decimal` which is a 128-bit representation of floating point numbers different from the IEEE 128-bit format. Its format is not directly supported on currently used processors, and therefore all operations are implemented in software by bit manipulations. Also, this type is intended mainly for handling currency and has approximate range only $-7.9 \cdot 10^{28}$ to $7.9 \cdot 10^{28}$, which is too-narrow range for circuit simulation.

Instead, in the article mentioned above, Mike Robbins proposes using double-double technique. This technique represents a single value as a unevaluated sum of two `double` floating point values, each of which having its own significand and exponent. This principle is illustrated in figure 1.10 where number π is represented as a sum of two doubles.[9]

$$\underbrace{3.141592653589793}\underbrace{238462643383279}$$

$$3.141592653589793 \cdot 10^{0} + 2.38462643383279 \cdot 10^{16}$$

Figure 1.10: Decomposition of π using double-double technique

Contrary to the options mentioned above, operations on double-double format are implemented using standard operations on `double` format, which are supported in today's hardware. This means that much higher speeds can be achieved. Hida et al. created a C++ library, that implements both double-double arithmetic, and it's slightly more complicated version – quad-double arithmetic. More details about the algorithms used can be found in their published paper [17].

Using these enhanced precision types is attractive, because they can solve some convergence issues during simulation, and we would therefore like to use

[8]In general, there are many reasons why solution might not converge, several possible techniques and simulator parameters used to overcome nonconvergence are explained in Ron Kielkowski's Inside SPICE [3].

[9]One thing worth noting is that an implementation of the double-double format needs to decompose values based on the the binary representation of the real numbers. The QD library by Hida et al. [16] uses values 3.141592653589793116e+00 and 1.224646799147353207e-16 in the source code, possibly to compensate errors from truncating periodic binary representation of the mantissa.

them in our library. However, use of these precision types could lead to significant slowdown of the simulation, because multiple primitive operations on `double`s are done for each operation on the double-double and quad-double types. This could unnecessarily slow down simulations of circuits which do not require the precision provided by these types. We would therefore like to make these enhanced precision types optional, and otherwise use the standard `double` precision type. Because our implementation will allow using any of the `double`, double-double and quad-double types, we would like to compare the simulator performance – i.e. speed and accuracy – when using each of these types to get the basic idea when use of these types is appropriate.

1.5 Goals

1. Implement SPICE-like simulation library
   (a) Target .NET Standard for maximum portability
   (b) Support performing time-domain simulation of the circuit, and allow changing parameters of circuit devices between individual timesteps.
   (c) Support following set of devices
      i. Ideal resistor
      ii. Ideal voltage source
      iii. Ideal current source
      iv. Ideal inductor
      v. Ideal capacitor
      vi. SPICE diode
      vii. SPICE BJT transistor
   (d) Allow new types of circuit analyses and circuit devices to be added to the simulator without modifying the library's source code.
   (e) Implement SPICE netlist parser to allow importing circuits and macromodels from standard SPICE netlist files.
   (f) Allow users of the library to choose between double, double-double, and quad-double precision types and compare the library's performance with respect to speed and accuracy for each listed precision type.
2. Use the simulation library to implement SPICE-like console application for .NET Core, which would accept implemented subset of SPICE netlist syntax.

2. The SPICE Netlist Syntax

This chapter presents the netlist syntax, which we plan to support in the NextGen SPICE library for importing circuits and in the standalone console application. The syntax presented here is a subset of that supported by SPICE3, and can be further restricted to that of SPICE2.

2.1 General Syntax

The SPICE netlist syntax is case insensitive. The netlist file consists of individual statements, which are separated by line breaks. If a statement is to span multiple lines, every subsequent line must be prefixed with a + symbol. Following code fragments are therefore equivalent.

```
V1 0 1
+ SIN 0 5V 10KHZ
```

```
V1 0 1 SIN 0 5V 10KHZ
```

Statements themselves are made up of individual data fields, delimited by blank characters. The meaning of data fields depends on the actual statement and on the position inside the statement. Generally, a data field specifies either a name, or a numeric value, and is then called name field or number field, respectively.

2.1.1 Number Fields

If the statement expects a data field to represent a numeric value, then the (number) field should start with a digit. However, if a decimal number is specified, the leading zeros may be omitted (therefore, `.05` is a valid number field having same value as `0.05`). It is also possible to specify the scale by either using suffixes like `E-9` or one of the scaling factors listed in the following table.

Factor	Scale
T	10^{12}
G	10^{9}
MEG	10^{6}
K	10^{3}
M	10^{-3}
U	10^{-6}
N	10^{-9}
P	10^{-12}
F	10^{-15}

Any additional characters that follow the number and scale factor are ignored, so fields `10000`, `10E+3V`, `10K`, `10KV`, `10KVOLTS` and `10KHz` all represent the same value. This is convenient, because the ignored part may be used to specify units and thus improve readability of the netlist file.

2.1.2 Name Fields

On the other hand, if a data field is to represent a name, an arbitrary string of alphanumeric characters can be used.

2.1.3 Comments

The input file can also contain comments, which are completely ignored by the parser. Comments begin with an asterisk * symbol, and end by a line break. Comments can be used to further improve readability of the netlist file.

```
V1 0 in 5        * a 5V voltage source between the ground and 'in' node
```

2.2 Data Statements

Data statements are used to describe the actual circuit and can be further divided into *device statements* and *model statements*. Device statements specify individual circuit devices and their connections to circuit nodes and generally have the following form:

```
<device name> <terminal connections> <device arguments>
```

The concrete SPICE device type is determined by the first letter of the device name, and from the device type, the number of terminal connections and arguments is derived. Like SPICE3, we will not pose any restrictions on the length of the device name.[1]

After the name follows a list of nodes to which the device connects. Nodes are identified by arbitrary alphanumeric strings.[2] The ground node is identified as `0`. After the terminal connections follows a device dependent argument list.

Model statements are used to specify parameters for more complex semiconductor devices, so that multiple devices can have the same parameters without their extensive repetition throughout the netlist file. Model statement has the following structure:

```
.MODEL <model name> <model type> (<model parameter list>)
```

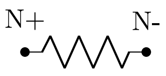
Each device can accept only model types corresponding to that particular device type. Each model type has a set of parameters, which are set in the model parameter list, each having its default value. When defining new model, only non-default values need to be specified by a list of key value pairs in the form `<name>=<value>`. The model name is then supplied as an argument to semiconductor devices such as a diode.

Following sections describe formats of several SPICE device statements. Values beginning with N are for node connections, values enclosed in square brackets are optional. Also, when applicable, the available model types and names of their parameters are listed.

[1]in SPICE2, the length was limited to seven characters only

[2]In SPICE2, nodes are identified by integers, one consequence of this is that `00` and `0` are equivalent in SPICE2, but not in SPICE3.

2.2.1 Resistor



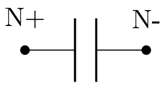
```
R<name> N+ N- <resistance>
```

A simple ideal resistor device. The order of N+ and N- nodes has no effect on the circuit behavior.

Examples:

```
R1 1 2 5OHM
R2 2 3 1K
```

2.2.2 Capacitor



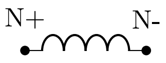
```
C<name> N+ N- <capacitance> [IC=<initial voltage>]
```

An ideal capacitor device, initial voltage can be specified to set specific condition on the beginning of the simulation. If initial condition is not present, capacitor is modeled as an open circuit in the first DC operating point calculation.

Examples:

```
C1 1 2 1F
C2 2 3 1N IC=1M
```

2.2.3 Inductor



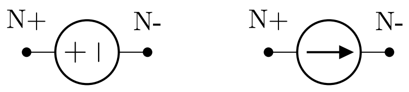
```
L<name> N+ N- <capacitance> [IC=<initial current>]
```

An ideal inductor device, initial current can be specified to set specific condition on the beginning of the simulation. If initial condition is not present, capacitor is modeled as an ideal short circuit in the first DC operating point calculation.

Examples:

```
L1 1 2 1F
L2 2 3 1N IC=1M
```

2.2.4 Input sources



```
V<name> N+ N- <source function>
I<name> N+ N- <source function>
```

NextGen SPICE supports complex specification of input source behaviors. Possible source functions are listed below.

DC Source

```
[DC] <voltage>
```

A source that has constant value, the `DC` identifier can be omitted.

Examples:

```
V1 1 0 5V
I2 1 2 DC 10KV
```

Sinusoidal Source

```
SIN <vo> <va> <freq> [<td> [<th> [<ph>]]]
```

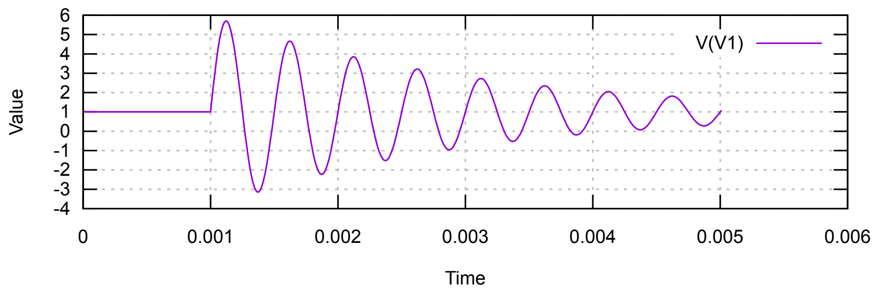
Parameter	Meaning
`<vo>`	Value offset
`<va>`	Value amplitude
`<fr>`	Waveform frequency
`<td>`	Delay time
`<th>`	Damping factor
`<phase>`	Phase offset

A sinusoidal source with amplitude damping. Value of the source function is given by

$$f(t) = \begin{cases} \texttt{<vo>} & \text{if } t < \texttt{<td>} \\ \texttt{<vo>} + \texttt{<va>} \begin{array}{l} \cdot e^{-(t-\texttt{<td>})\cdot\texttt{<th>}} \\ \cdot \sin\left(2\pi \cdot (\texttt{<fr>} \cdot (t - \texttt{<td>}) + \texttt{<ph>})\right) \end{array} & \text{if } t \geq \texttt{<td>} \end{cases}$$

Example:

```
V1 1 0 SIN 1 5 2KHZ 1MS 0.5K
```



Exponential Source

```
EXP <v1> <v2> [<td1> <tau1> [<td2> <tau2>]]
```

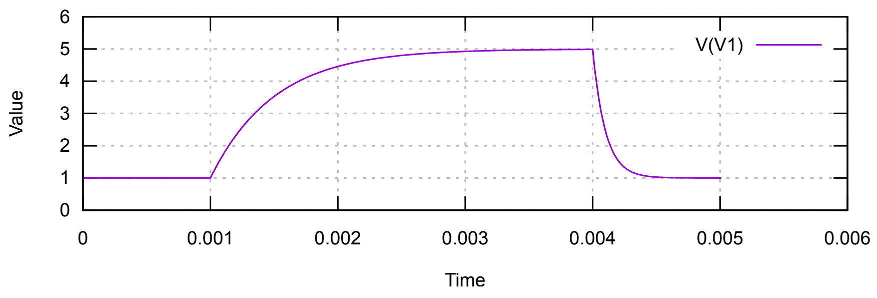
Parameter	Meaning
`<v1>`	Initial value
`<v2>`	Pulse value
`<td1>`	Delay before first edge
`<tau1>`	First edge time constant
`<td2>`	Delay before second edge
`<tau2>`	Second edge time constant

A pulsing source with exponential rising and falling edges. Values of the source are given by:

$$f(t) = \begin{cases} \texttt{<v1>} & \text{if } t < \texttt{<td1>} \\ \texttt{<v1>} + \left(\texttt{<v1>} - \texttt{<v2>}\left[1 - e^{\frac{-(t-\texttt{<td1>})}{\texttt{<tau1>}}}\right]\right) & \text{if } \texttt{<td1>} \geq t > \texttt{<td2>} \\ \texttt{<v1>} \begin{array}{l} + \left(\texttt{<v1>} - \texttt{<v2>}\left[1 - e^{\frac{-(t-\texttt{<td1>})}{\texttt{<tau1>}}}\right]\right) \\ + \left(\texttt{<v2>} - \texttt{<v1>}\left[1 - e^{\frac{-(t-\texttt{<td2>})}{\texttt{<tau2>}}}\right]\right) \end{array} & \text{if } t \geq \texttt{<td2>} \end{cases}$$

Example:

```
V1 1 0 EXP 1 5 1MS 0.5M 4MS 0.1M
```



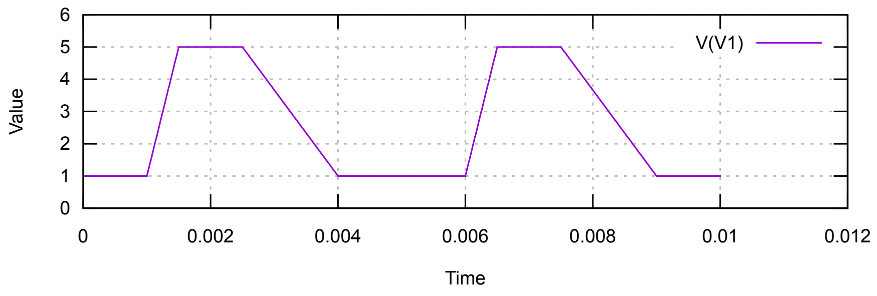
Pulse Source

```
PULSE <v1> <v2> <td> <tr> <tf> <ton> <period>
```

Parameter	Meaning
`<v1>`	Initial value
`<v2>`	Pulse value
`<td>`	Delay before rising edge of the pulse
`<tr>`	Time of the rising edge of the pulse
`<tf>`	Time of the falling edge of the pulse
`<ton>`	Duration of the pulse
`<period>`	Period of the source

A source that sends individual pulses.
Example:

```
V1 1 0 PULSE 1 5 1MS 0.5MS 1.5MS 1MS 5MS
```

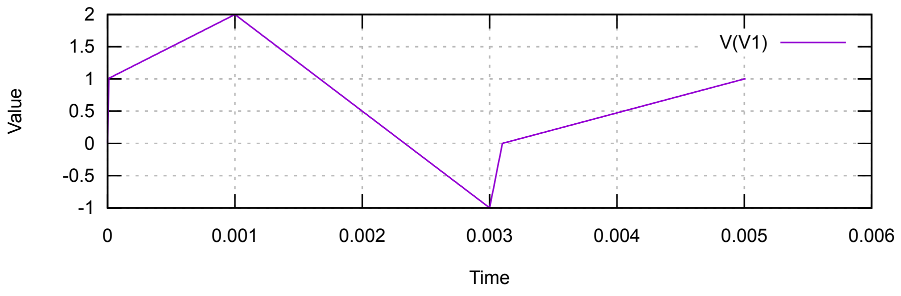


Piecewise Linear Source

```
PWL <t1> <v1> [<t2> <v2> [<t3> <v3> [...]]]
```

An arbitrary piece-wise linear source. The argument list consists of pairs of timepoints and source values. The intermediate values are determined using linear interpolation.
Example:

```
V1 1 0 PWL 0MS 1 1MS 2 3MS -1 3.1MS 0 5MS 1
```



AM Source

```
AM <amp> <dc> <fm> <fc> [<td> [<ph>]]
```

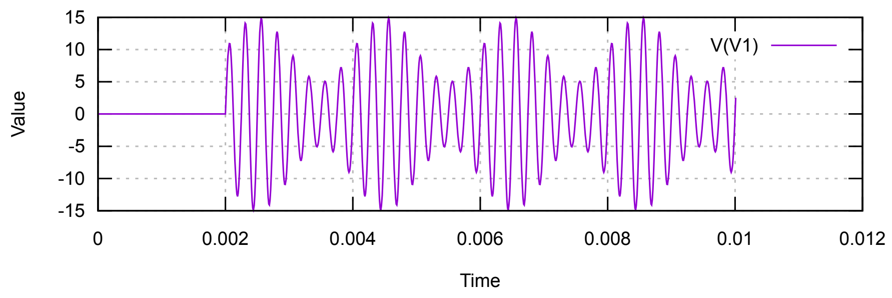
Parameter	Meaning
`<amp>`	Peak amplitude of the unmodulated signal.
`<dc>`	DC offset
`<fm>`	Modulation frequency
`<fc>`	Carrier frequency
`<td>`	Delay before the signal
`<ph>`	Phase offset

A source with amplitude modulated signal. Value at any given timepoint is given by

$$f(t) = \texttt{<amp>} \begin{array}{l} \cdot\left(\texttt{<dc>} + \sin\left(2\pi \cdot \texttt{<fm>} \cdot (t - \texttt{<td>})\right) + \texttt{<ph>}\right) \\ \cdot \sin\left(2\pi \cdot \texttt{<fc>} \cdot (t - \texttt{<td>}) + \texttt{<ph>}\right). \end{array}$$

Example:

```
V1 1 0 AM 5 2 0.5KHZ 4KHZ 2MS
```



SFFM Source

```
SFFM <dc> <amp> <fc> <m> <fm>
```

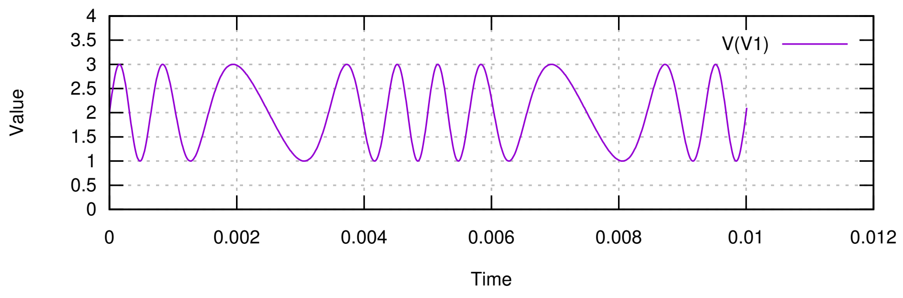
Parameter	Meaning
`<dc>`	DC offset of the signal
`<amp>`	Amplitude of the carrier.
`<fc>`	Carrier frequency
`<m>`	Modulation index
`<fm>`	Modulation frequency

A source with frequency modulated signal. Value at any given timepoint is given by

$$f(t) = \texttt{<dc>} + \texttt{<amp>} \cdot \sin\left(2\pi \cdot \texttt{<fc>} \cdot (t - \texttt{<td>}) + m \cdot \sin\left(2\pi \cdot \texttt{<fm>} \cdot (t - \texttt{<td>})\right)\right).$$

Example:

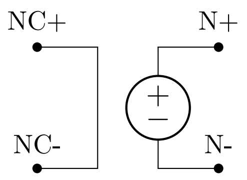
```
V1 1 0 SFFM 2 1 1KHZ 3 0.2KHZ
```



2.2.5 Controlled Sources

SPICE supports linear dependent sources, both current and voltage controlled. In case of voltage controlled source, an additional pair of terminals is specified, and value of the source is linearly dependent on the voltage between those control terminals. In case of current controlled sources, a name of a voltage source is supplied and the value of the source depends linearly on the current flowing through said device.[3] The coefficient of linear dependence, called *gain* is supplied as the last parameter.

Voltage Controlled Voltage Source

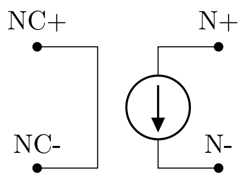


```
E<name> N+ N- NC+ NC- <gain>
```

Example:

```
E1 1 2 3 4 100
```

Voltage Controlled Current Source

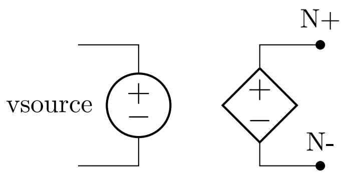


```
G<name> N+ N- NC+ NC- <gain>
```

Example:

```
G1 1 2 3 4 100
```

Current Controlled Voltage Source



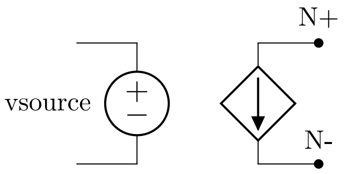
[3]The reason that only voltage source can be used is that the current flowing through the voltage source is directly accessible through a circuit variable in the equation system. Also, because earlier versions of SPICE could output only currents flowing through voltage sources, it became a standard practice to use 0V voltage sources as amperemeters.

```
H<name> N+ N- <vsource> <gain>
```

Example:

```
VMETER 1 2 0
H1 2 3 VMETER
```

Current Controlled Current Source

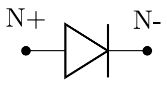


```
F<name> N+ N- <vsource> <gain>
```

Example:

```
VMETER 1 2 0
F1 2 3 VMETER
```

2.2.6 Diode



```
D<name> N+ N- <model name>
```

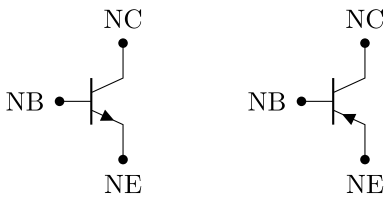
A semiconductor diode device. Physical parameters of diode are set using a `.MODEL` statement with `D` model type. Following table lists supported model parameters for the diode model. The parameters in gray are parsed, but do not affect the simulation in the current implementation.

Parameter name	Description	Default value	
IS	Saturation current	$1 \cdot 10^{-14}$	A
RS	Ohmic resistance	0	
N	Emission coefficient	1	
TT	Transit-time current	0	s
CJO	Zero-bias junction capacitance	0	F
VJ	Junction potential	1	V
M	Grading coefficient	0.5	
EG	Activation energy	1.11	eV
XTI	Saturation-current temperature exponent	3	
KF	Flicker noise coefficient	0	
AF	Flicker noise exponent	1	
FC	Coefficient for forward-bias depletion capacitance formula	0.5	
BV	Reverse breakdown voltage	∞	V
IBV	Current at breakdown voltage	$1 \cdot 10^{-3}$	A
TNOM	Parameter measurement temperature	27	°C

Example:

```
D1 1 2 1N4148
.MODEL 1N4148 D(IS=2.52N RS=.568 N=1.752 CJO=4P M=.4 TT=20N VJ=20 BV=75)
```

2.2.7 BJT Transistor



```
Q<name> NC NB NE <model name>
```

A semiconductor BJT transistor device. Physical parameters of BJT are set using a `.MODEL` statement, as well as the polarity of the transistor. There are two model types for BJT transistor: `NPN` and `PNP`. Both model types accept the same parameters, which are listed in the following table. The parameters in gray are parsed, but do not affect the simulation in the current implementation.

Parameter name	Description	Default value	
IS	Transport saturation current	1.0e16	A
BF	Ideal maximum forward beta	100	
NF	Forward current emission coefficient	1.0	
VAF	Forward Early voltage	∞	V
IKF	Corner for forward beta high current roll-off	∞	A
ISE	B-E leakage saturation current	0	A
NE	B-E leakage emission coefficient	1.5	
BR	Ideal maximum reverse beta	1	
NR	Reverse current emission coefficient	1	
VAR	Reverse Early voltage	∞	V
IKR	Corner for reverse beta high current roll-off	∞	A
ISC	Leakage saturation current	0	A
NC	Leakage emission coefficient	2	
RB	Zero bias base resistance	0	
IRB	Current where base resistance falls halfway to its min value	∞	A
RBM	Minimum base resistance at high currents	RB	
RE	Emitter resistance	0	
RC	Collector resistance	0	
CJE	B-E zero-bias depletion capacitance	0	F
VJE	B-E built-in potential	0.75	V
MJE	B-E junction exponential factor	0.33	
TF	Ideal forward transit time	0	s
XTF	Coefficient for bias dependence of TF	0	

VTF	Voltage describing VBC dependence of TF	∞	V
ITF	High-current parameter for effect on TF	0	A
PTF	Excess phase at freq=1.0/(TF*2PI) Hz	0	deg
CJC	B-C zero-bias depletion capacitance	0	F
VJC	B-C built-in potential	0.75	V
MJC	B-C junction exponential factor	0.33	
XCJC	Fraction of B-C depletion capacitance connected to internal base node	1	
TR	Ideal reverse transit time	0	s
CJS	Zero-bias collector-substrate capacitance	0	F
VJS	Substrate junction built-in potential	0.75	V
MJS	Substrate junction exponential factor	0	
XTB	Forward and reverse beta temperature exponent	0	
EG	Energy gap for temperature effect on IS	1.11	eV
XTI	Temperature exponent for effect on IS	3	
KF	Flicker-noise coefficient	0	
AF	Flicker-noise exponent	1	
FC	Coefficient for forward-bias depletion capacitance formula	0.5	
TNOM	Parameter measurement temperature	27	°C

Examples:

```
Q1 1 2 3 QMOD1
.MODEL QMOD1 PNP(IS=1P)

Q2 4 5 6 QNL
.MODEL QNL NPN(BF=80 RB=100 TF=0.3NS TR=6NS CJE=3PF CJC=2PF VAF=50V)
```

2.2.8 Subcircuits

Subcircuits are SPICE netlist term for device macromodels mentioned back in the introduction chapter. Following syntax is used.

```
.SUBCKT <subcircuit name> <terminal nodes>
<subcircuit description>
.ENDS
```

Description of the subcircuit has to be enclosed between `.SUBCKT` and `.ENDS` statement. The `.SUBCKT` statement states the name of the subcircuit and lists names of terminal nodes, which will be then used to connect the subcircuit to the outer circuit. There must be at lesat one terminal node and none of them can be 0 (ground node).

The actual description of the subcircuit can contain only data statements – device statements, `.MODEL` statements, and other `.SUBCKT` statements. Also, it is customary to place a comment line describing the meaning of the terminal nodes (like in figure 1.4).

Any names defined inside a subcircuit are strictly local to the subcircuit. Therefore, models and subcircuits defined inside the subcircuit cannot be used after the .ENDS statement.

Subcircuit can then be used as an individual device by following syntax:

```
X<name> <terminal nodes> <subcircuit name>
```

where `<subcircuit name>` is the name supplied in the corresponding .SUBCKT statement, and `<terminal nodes>` names appropriate number of nodes to which the subcircuit should connect.

Example:

```
.SUBCKT ACAMPLIFIER 2 1 3
R1 1 4 2K
R2 4 0 500
C1 2 4 10n
Q1 3 4 5 2N2222
Rc 1 3 2K
Re 5 0 1e3
.MODEL 2N2222 PNP(BF=50 IS=1E-13 VBF=50)
.ENDS

XOPAMP 1 2 3 ACAMPLIFIER
```

2.2.9 Summary of the Device Statements

Device	Syntax
Resistor	`R<name> N+ N- <resistance>`
Capacitor	`C<name> N+ N- <capacitance> [IC=<voltage>]`
Inductor	`L<name> N+ N- <inductance> [IC=<current>]`
Voltage source	`V<name> N+ N- <source function>`
Current source	`I<name> N+ N- <source function>`
Voltage controlled voltage source	`E<name> N+ N- NC+ NC- <gain>`
Voltage controlled current source	`G<name> N+ N- NC+ NC- <gain>`
Current controlled voltage source	`H<name> N+ N- <voltage source> <gain>`
Current controlled current source	`F<name> N+ N- <voltage source> <gain>`
Diode	`D<name> N+ N- <model name>`
BJT transistor	`Q<name> NC NB NE <model name>`

2.3 Control Statements

Control statements are used for performing circuit simulations.

2.3.1 .OP Statement

```
.OP
```

The `.OP` statement is used for requesting DC operating point analysis, which means calculating the values of node voltages and branch currents corresponding to a stable state of the circuit. This statements does not have any arguments.

2.3.2 .TRAN Statement

```
.TRAN <timestep> <stop time> [<start time>]
```

This statement is used for requesting the transient analysis — time-domain simulation of the circuit — for specified duration with given timestep. For each timepoint, an operating point is established, and time-dependent behavior of devices such as capacitor and inductor is modelled using numerical integration methods. If we are not interested only in data after certain timepoint, we can use the third optional parameter to instruct the simulator to not print simulation results until a certain timepoint.

2.4 Output Statements

Output statements can be used to select which data should be printed in the simulator's output. If no output statement is provided, NextGen SPICE will print all available data. Currently, the only supported statement is the `.PRINT` statement.

```
.PRINT <analysis type> <list of requested data>
```

Analysis type is either `OP` or `TRAN`, indicating from which analysis type the data are requested. Following table summarizes possible data specifiers.

Specifier	Description
`V(<node>)`	Voltage of the node `<node>`
`V(<node1>,<node2>)`	Voltage between nodes `<node1>` and `<node2>`
`V(<device>)`	Voltage across the `<device>` device[4]
`I(<device>)`	Current flowing through the `<device>` device[4].

[4]Only devices having exactly two terminals are allowed.

2.5 Netlist File Structure

The structure of the netlist can be almost arbitrary, the only restrictions are that the first line, called *title line*, should contain a brief description of the netlist contents (and is not interpreted as a statement), and that the last statement is a `.END` statement. There are no restrictions on the relative order of statements in the netlist. For example, semiconductor device models can be used in device statements even before they are defined by their respective `.MODEL` statement. However, even though any order is possible, netlist files are commonly structured in the following manner:

- Title
- Device statements
- `.MODEL` statements
- Control statements
- Output statements
- `.END` statement

2.6 Circuit Topology Constraints

There are some restrictions on how the circuit devices can be connected to nodes. These restrictions help ensuring that the equation system rising from the simulated circuit always has unique solution. The rules are the following:

- The circuit may not contain cycles consisting of voltage defined devices (e.g. voltage sources and inductors)
- The circuit may not contain cutsets consisting of current defined devices (e.g. current sources and capacitors)
- In a circuit, there must exist a path from each node to the ground node
- In a subcircuit, there must exist a path between each pair of terminals that does not contain the ground node

3. Implementation Analysis

Just like other modern day circuit simulators, our library is also heavily influenced by the original Berkeley SPICE. We have decided to reflect this fact on the name our library and call it NextGen SPICE, and we will use this name in the remainder of this thesis.

This chapter analyses various possibilities of NextGen SPICE simulator implementation. The reader should be acquainted with the necessary theory of circuit simulation. Ron Kielkowski's Inside SPICE [3] is an excellent source of details about the workings of SPICE-like simulators. Necessary mathematical theory is nicely summarized in the documentation of QUCS simulator [18], which we will cite frequently in the next chapters. Another great source is the Ph.D. theses of Laurence W. Nagel (Author of SPICE2) [19]. The last two sources are freely available and we include them in the attached CD for convenience.

3.1 Initial Organisation of the library

One of the goals of this thesis is implementation of configurable and extensible circuit simulation library. One of the requirements is that new types of circuit analyses as well as new circuit devices can be added without modification of the core library's source code (goal 1d). Before we start with the actual analysis, we will briefly describe how the original SPICE program operates.

3.1.1 Overview of SPICE Simulator Workflow

The top level view on the SPICE simulator is summarized in the figure 3.1. The user specifies the circuit to be simulated inside a SPICE netlist file (1). This file is parsed by SPICE, which constructs an internal representation of the circuit (2) and validates the circuit topology according to the rules described in section 2.6. If the circuit is correctly formed, SPICE performs the simulations specified in the netlist file. This consists of mapping the devices into coefficients in an equation system characterizing the circuit (3). This equation system is then solved to obtain the result of the analysis, which is then stored in the circuit representation. The user-specified characteristics of the circuit devices are then printed to the output (4).

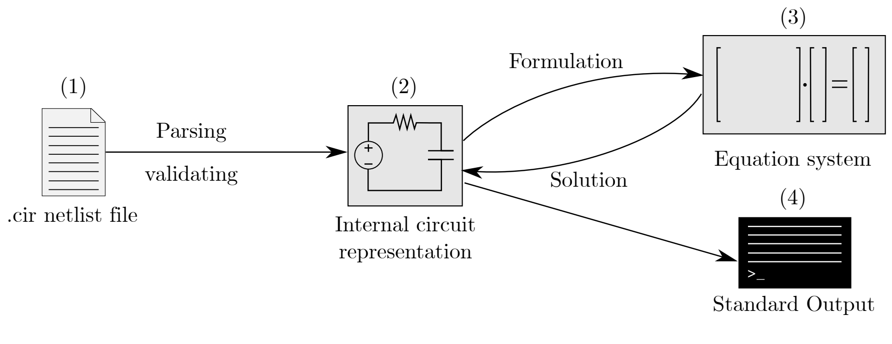


Figure 3.1: SPICE operation workflow.

Because different analyses simulate different characteristics of circuit devices, most devices contribute to the equation system coefficients differently in each circuit analysis. To provide a concrete example, the following two paragraphs briefly describe two distinct kinds of circuit analyses: transient analysis (implementation of which is part of this thesis), and AC frequency sweep analysis (which we would like to implement in future versions), and their specific way of simulating capacitors and inductors.

In transient analysis, the circuit's behavior is simulated over time. The first step of this analysis is calculating the initial DC bias of the circuit, which is the technical term for calculating the node voltages and currents flowing through the circuit branches. During this step, capacitors are modeled as ideal open circuits and inductors as ideal short circuit.[1] In subsequent timepoint calculations, capacitor and inductor devices are modeled by equivalent subcircuits consisting of a voltage or current source and a resistor. Values of voltage, current and resistance in these equivalent subcircuits are recomputed each timepoint to reflect energy storing behavior of these devices.

AC frequency sweep analysis simulates how the circuit behaves when a signal of certain frequency is applied to the it. As opposed to transient analysis, which iterates over time, AC frequency sweep iterates over frequency of the applied signal. It also starts by calculating the DC bias of the circuit, but after that, nonlinear characteristics of circuit devices are not modeled. Instead the behavior of each device around the established DC operating point is considered to be linear, which simplifies the analysis. Contrary to the transient analysis which models energy storing behavior of capacitors and inductors, AC frequency sweep models their reactance, which depends on the signal frequency and device's capacitance and inductance, respectively. Additional difference between transient and AC frequency sweep analyses is that in AC frequency sweep analysis, the equation system that characterizes the circuit contains complex numbers as coefficients, whereas in the transient analysis, only real numbers are needed.[2]

3.1.2 Separating Circuit Analyses

Suppose we used same workflow as in the figure 3.1 and implemented the transient analysis via instance methods on the circuit representation. When we would want to implement AC frequency sweep analysis to the simulator, we would have to modify the circuit representation and add new methods and data fields, which contradicts our goal of extensibility (goal 1d). In order to leave the transient analysis implementation intact, we would have to create a brand new circuit representation that would implement the operations needed by the AC frequency sweep analysis.

[1]Consequence of modeling capacitor as open circuit and inductor as short circuit is that the simulation starts from a stable (quiescent) state of the circuit, SPICE simulators allow user to specify custom initial conditions: voltage across the capacitor and current flowing through the inductor, and thereby starting the simulation from an unstable state.

[2]Additional details about how capacitors and inductors are modeled can be found in QUCS technical papers. For trasient analysis, see sections 6.3.1 and 6.3.2 for capacitor and inductor, respectively; and for AC analysis see sections 9.3 for capacitor and 9.4 for inductor. More detailed description of individual SPICE circuit analyses can be found in chapter 2 of Inside SPICE p. 37–41

Having multiple implementations of the circuit representation may seem to be excessive code duplication. However, as we have described in the previous paragraphs, implementation of AC frequency sweep analysis would be quite different from that of transient analysis: different contributions to circuit equation system, usage of complex numbers in the equation system, and no need for updating the inner state of the device. Because the two types of analyses are conceptually different, implementing them separately could actually improve readability and maintainability of the code base.

To allow performing different circuit analyses without having to create the specific circuit representation manually for each analysis, we decided to modify the workflow shown into the one shown in figure 3.2. We introduced an analysis-independent circuit description (which we will simply refer to as *circuit description*), which would be used to automatically create the analysis-specific circuit representation (which, we will refer to as *circut model* for brevity) on-demand. The library functionality can be thus partitioned to analysis-independent and analysis-dependent parts as illustrated on the figure.

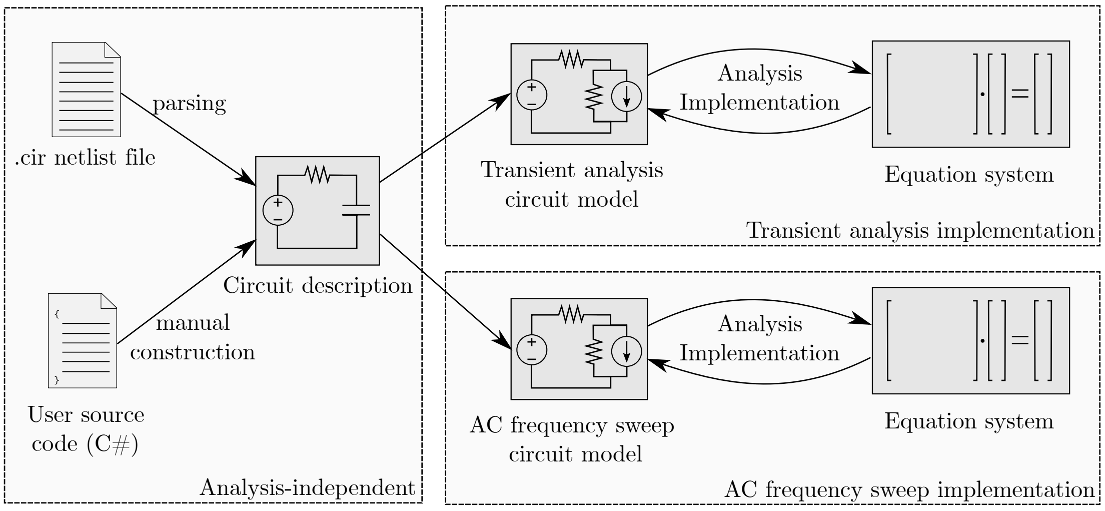


Figure 3.2: Workflow used in NextGen SPICE library

3.1.3 Devices and Device Models

We have also stated that we would like the possibility of adding new devices in the future. To achieve that, transient analysis (and each other analysis to be implemented) should specify operations required from each device via an interface. Adding new device would consist of implementing this interface for each analysis type. Figure 3.3 illustrates possible device implementation hierarchy. Suppose that transient analysis requires `ITransientDevice` interface, and AC frequecy sweep requires `IAcFreqSweepDevice` interface. We have already described that capacitor and inductor devices are handled differently in both of these analyses, and it would make sense to implement the behavior separately for each analysis. At the same time, some devices – such as resistor – are handled the same way. The implementation of resistor device could possibly implement both interfaces in the same class.

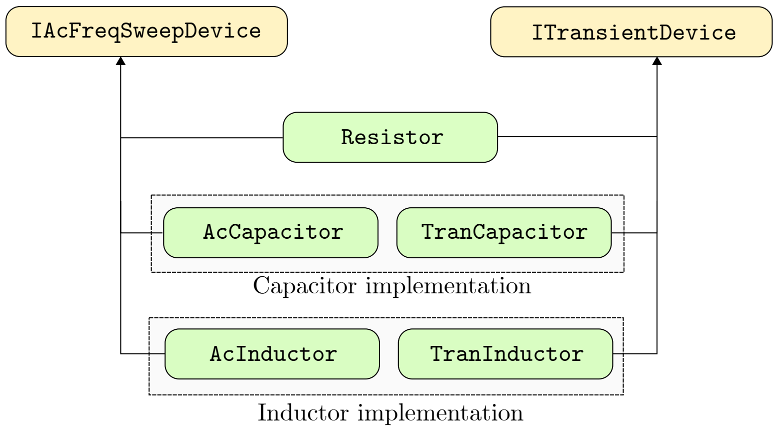


Figure 3.3: Implementation of devices for different circuit analyses

Furthermore, we also stated in requirements that it should be possible to easily change how each device is simulated. As an example, consider BJT transistor device. During transient analysis, BJT transistor can be modeled by using either Ebers-Moll model or more detailed Gummel-Poon model. Other analysis types can potentially use different transistor models. We would like to allow users to select which device model[3] should be used for the BJT device (and other devices as well). This will allow the library to be used for comparing existing device models and developing new ones. Concrete example hierarchy of the classes, which can be used to model BJT transistor in transient and AC frequency sweep analyses, is illustrated in figure 3.4. Classes `TranEbersMollBjt` and `TranGummelPoonBjt` implement the operations required by transient analysis for the respective transistor device models. During AC freqency sweep analysis, a completely different model (Hybrid-pi) is used, which is implemented by the `AcHybridPiBjt` class.

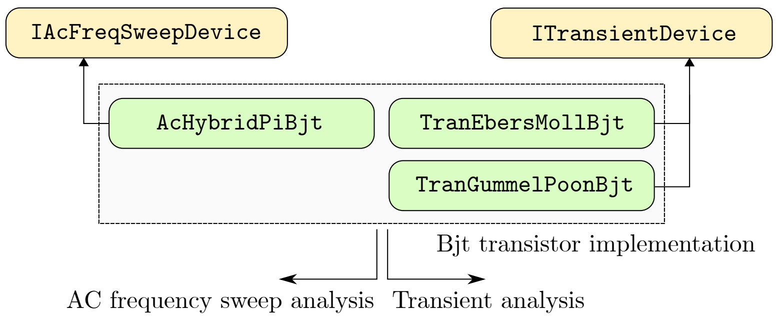


Figure 3.4: Separation of device implementation for each analysis type.

3.1.4 Splitting to Multiple Assemblies

A straightforward way of implementing the library would be putting everything into one assembly. However, that means that after each modification, the whole

[3]To avoid possible confusion with `.MODEL` statements used in netlist files, this thesis uses term *device model* exclusively to refer to the set of equations describing the device's behavior (such as the mentioned Ebers-Moll model). The entities described by `.MODEL` statement are referred to as *device model parameters*.

library has to be replaced by the new version. In the preceding section we decided to make the implementation of individual analysis types independent of each other, with separate set of classes for representing the circuit under analysis. As developers, we would like to be able to develop, update and deploy each type of circuit analysis independently of the other types. In order to achieve that, we decided to organize the library into assemblies as illustrated in figure 3.5. The `NextGenSpice.Core` is a shared assembly containing analysis-independent parts of the library, and is referenced by assemblies implementing individual analysis types. The implementation of the circuit model for transient analysis will be contained in `NextGenSpice.LargeSignal` assembly. We chose the name LargeSignal because the resulting model allows transient, DC sweep and DC operating point analyses, which all rely on the large-signal model of circuit devices.

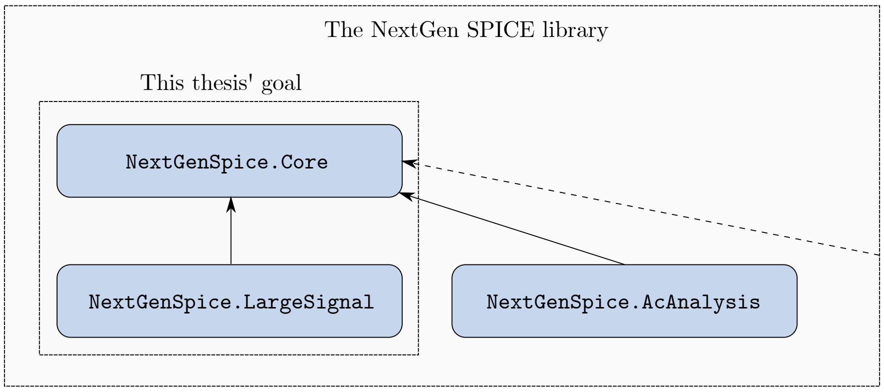


Figure 3.5: General organisation of the analysis types in the library.

The `NextGenSpice.Core` assembly should have no knowledge of the assemblies containing the analysis types. To achieve that, we will make use of a principle called *inversion of control* (IoC). In simple words, each assembly containing implementation of a circuit analysis type should inform the `NextGenSpice.Core` assembly of its existence and instruct it how it's circuit model can be created from the circuit description.

To make this procedure automatic without any action needed from the library's user, we decided to use an IoC framework. The functionality we require from the framework is quite simple and available in most IoC frameworks (importing implementations of certain interface from assemblies in the same directory as the core library). We decided to use MEF framework [20] by Microsoft, which is distributed as a NuGet package and is standardly used. Another attractive feature of MEF is that it allows exporting classes by simply adding an `[Export]` attribute.

In this section we described the reasons for the overall design of the NextGen SPICE library, in the sections that follow, the individual components of the library are analyzed in greater depth.

3.2 NextGenSpice.Core

The `NextGenSpice.Core` assembly will be the integral part of NextGen SPICE. It should contain the analysis-independent parts of the library: the circuit description classes, faculties for composing and validating the circuit, and the central mechanism that allows creation of analysis-specific circuit models. These parts will be discussed in the following subsections.

3.2.1 Representation of the Circuit

In the previous section we made an important decision to introduce separate sets of classes for circuit representation for each circuit analysis type. We have also decided to create another separate circuit description from which these analysis-specific representations should be constructed on-demand. The circuit description should be encapsulated in a single object of class `CircuitDefinition` for easy manipulation. Following subsubsections consider individual aspects of representing the circuit.

Representing Circuit Nodes

In the SPICE netlist syntax, as described in chapter 2, we allow circuit nodes to be identified by arbitrary alphanumeric strings, which would make C# type `string` a straightforward choice.

However, the main reason for using strings in the netlist syntax was probably to let users choose convenient names and make the netlists more readable. During the actual simulation and equation formulation, the circuit nodes need to be numbered in order to map the devices to corresponding equation matrix entries, and it would be more natural to use numbers to identify the circuit nodes.

Because we do not expect users of NextGen SPICE library to compose large circuits in the source-code manually, we decided to use C# type `int` for identifying individual nodes. This also implies that we need to translate the circuit node names while parsing SPICE netlists into node indices, and provide this mapping to the user.

Representing Individual Devices

We need a circuit device representation which allows simple adding of new devices. A natural way of representing cicruit devices in OOP language like C# is introducing a class per circuit device, which all implement the same interface, e.g. `ICircuitDefinitionDevice`. This would lead to a hierarchy parallel to that of the analysis-specific device implementation classes from section 3.1.3. Having these multiple parallel hierarchies would simplify implementation of the analysis-specific circuit model creation, because all it would need is a mapping between these two hierarchies, as illustrated in figure 3.6.

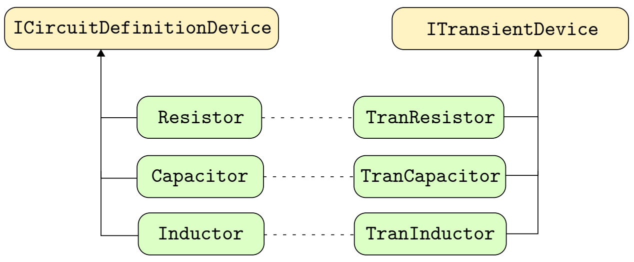


Figure 3.6: Mapping between circuit definition and analysis implementation classes

All data specific for any particular device would be stored in the respective properties of the device's class, and new devices can be added by creating another implementation of the `ICircuitDescriptionDevice` interface. Since this circuit device representation meets our requirements, we decided to use it.

Representing subcircuits

An important feature of SPICE netlists which we need to take into consideration when designing the circuit representation is the subcircuit feature (see section 2.2.8).

Since a subcircuit can be used multiple times throughout the netlist file (i.e. multiple `X` statements with the same subcircuit name), we have decided to create separate classes `SubcircuitDefinition` for the description of the subcircuit and `SubcircuitDevice` for the usage of the subcircuit. The subcircuit definition would need to store data about the inner devices and nodes, and which nodes are the connections to the outside circuit. The `SubcircuitDefinition` instances would be shared among potentially many `SubcircuitDevices`, which store data about the outside nodes to which the subcircuit is connected.

Enforcing Circuit Topology

In section 2.6, we described circuit topology restrictions that we will impose on the circuit to ensure that the circuit equations during the simulation have a unique solution. Also, we want to prohibit user from making changes to circuit topology during the circuit simulation, because it could cause the circuit to be no longer valid. We would also like to diagnose any circuit topology violations as soon as possible, preferably immediately after the whole circuit is constructed.

The changes to the circuit topology could be achieved by two ways: adding or removing a whole device, and changing the nodes to which the device was connected. Both can be forbidden by making the circuit description read-only. However, that would mean that all circuits devices would have to be supplied at once to the `CircuitDefinition` constructor, and the validation would have to be performed inside the constructor.

To move the validation code outside the `CircuitDefinition` class, we will use a separate class `CircuitBuilder` to incrementally add new devices, and perform the validation before creating the actual `CircuitDefinition` class instance.

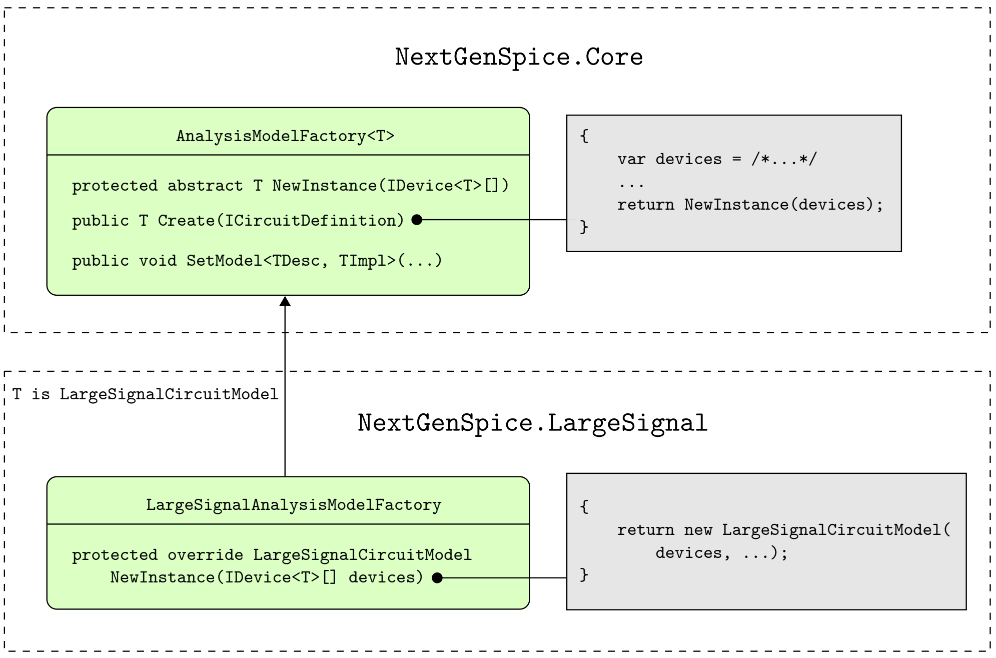
3.2.2 Creating Analysis-specific Circuit Representations

Back in section 3.1.2, we decided to introduce multiple parallel hierarchies for representing the electrical circuit, each one for a particular circuit analysis. In order for the library to be easy to use, the user should not have to create these representations separately for each circuit analysis type manually. Instead, the library should provide a way to create the analysis-specific circuit representations automatically.

Ideal user interface of the library would allow user to specify the mapping between the circuit description classes as discussed in section 3.2.1. If the user then requested the transient analysis circuit representation, the library would then use this mapping to instantiate classes that implement the transient analysis logic for each device from the circuit description, and return the result back to the user.

This mapping will also be used to specify which device model should be used during the analysis. To allow simple comparing of different device models (such as Ebers-Moll or Gummel-Poon BJT transistor models, which would be each implemented as a separate class, as discussed back in section 3.1.3), we would like to allow users to potentially specify multiple sets of mappings for a certain analysis type.

We have therefore decided to encapsulate these mappings in a *factory class*. However, the concrete class type of the analysis-specific circuit representation is not known, because the `NextGenSpice.Core` assembly does not contain any analysis-specific functionality. We will therefore use a class hierarchy similar to the one in figure 3.7. We will use the generic class feature of C# and implement abstract `AnalysisModelFactory<T>` class with one generic parameter for the circuit representation class type. This abstract class will implement methods for managing the mappings between device description and implementation classes (symbolized by the `SetModel` method). The actual instantiation of the analysis-specific circuit model (`LargeSignalCircuitModel` on the figure) is delegated to derived classes, which provide a way to create new methods by implementing the abstract `NewInstance` method.



NextGenSpice.Core
AnalysisModelFactory<T>
protected abstract T NewInstance(IDevice<T>[])
public T Create(ICircuitDefinition)
public void SetModel<TDesc, TImpl>(...)
{ var devices = /*...*/ ... return NewInstance(devices); }
T is LargeSignalCircuitModel
NextGenSpice.LargeSignal
LargeSignalAnalysisModelFactory
protected override LargeSignalCircuitModel NewInstance(IDevice<T>[] devices)
{ return new LargeSignalCircuitModel(devices, ...); }

Figure 3.7: Relationship between analysis model factories

In section 3.1.4, we decided to use MEF framework to automatically discover the available analysis types. This means that each assembly with a circuit analysis implementation needs to export it's own class derived from `AnalysisModel Factory<T>`. To provide a convenient place to aggregate the exported factories, we introduced yet another class: `AnalysisModelCreator`, which serves as a container for the analysis circuit model factories. The actual analysis model creation can be then implemented as a generic method on the `AnalysisModelCreator`, which will then find the appropriate factory to be used for the construction of the desired circuit model class specified as the type argument.

3.3 SPICE Netlist Parser

An important feature of our library is to allow users import subcircuits from SPICE netlist files (Goal 1e). Therefore, a SPICE netlist parser needs to be implmeneted.

Both lexer and parser can be either written manually or be generated by a tool. In .NET ecosystem, possible choices include tools generators such as ANTLR [21]. When using these generator tools, the language is described by a set of regular expressions and formal grammar rules, and the tool then generates the implementation of the lexer and parser.

Since the SPICE netlist grammar has a simple structure and hand-written parser would be easy to write, we have come to conclusion that the generator tools would only add unnecessary complexity to the library's implementation, and we will therefore implement the parser manually.

3.4 Double-double and Quad-double Arithmetic

According to goal 1f of this thesis, the library should allow representing real numbers with greater precision using the double-double and quad-double technique. Implementation of these techniques is very complex and requires deep knowledge and understanding of the algorithms to be done correctly. Therefore, we will not implement double-double and quad-double arithmetic ourselves, but we will use a third party library to provide as that functionality.

During our search for a suitable library, we did not find any implementation of double-double or quad-double in .NET. However, we discovered a C++ implementation by Hida et al. called QD [16] which is available under BSD license. Even though our library will be targeted at .NET Standard, we still can use this library because .NET Standard version 2.0 requires implementations of .NET to provide PInvoke, which is a feature that allows making calls to native code.

Because C++ is a compiled language, the implementation of QD needs to be recompiled for each platform, which would complicate the deployment of the library. However, the enhanced precision is needed only while solving the circuit equation, where the truncation errors could cause significant errors in the solution. Standard `double` type is sufficient for storing the parameters of the individual circuit devices. This means that enhanced types are not needed in the simulator's interface, but can be a detail of the equation system implementation, which can be changed without affecting the user's code.

We have therefore decided to use different internal real number representation based on compile time compilation symbols. When no symbols are specified, the library would be compiled without any dependencies on the native code (and hence use the `double` type). Library thus compiled can be distributed just like any .NET Library. On the other hand, users who wish to use enhanced precision types can still do so by compiling the library with appropriate symbols.

Using C++ implementation of individual arithmetic operations on the double-double and quad-double types also means that a call to native code has to be made for each numeric operation during equation solution. The transition between managed and unmanaged code for such trivial operations may incur nontrivial overhead. To gain better insight on how significant this overhead would be in our implementation, we implemented Gaussian elimination on `double` type in four variants.

- `Managed` – normal implementations in C# code.
- `Managed_Wrapped` – the `double` type was wrapped in a struct that implements necessary arithmetic operators by built-in operators on `double` type. This implementation reflects more closely how the double-double and quad-double types would perform when implemented in pure .NET.
- `Managed_Pinvoke` – similar to `Managed_Wrapped`, but the arithmetic operations are performed in C++ via a PInvoke call. This is how the enhanced precision types will perform when PInvoke is used for each arithmetic operation.
- `Native` – Whole algorithm is implemented in C++ and performed on one PInvoke call.

We used BenchmarkDotNet [22] for measuring the run times for equation systems. The table in figure 3.8 summarizes the results on equation systems with N variables for $N = 20$ and 200 variables. All times are in microseconds.[4]

Method	N	Mean (μs)	Error (μs)	StdDev (μs)	Scaled
`Managed`	20	7.900	0.0147	0.0114	1.00
`Managed_Wrapped`	20	27.233	0.1208	0.1070	3.45
`Managed_Pinvoke`	20	91.116	0.3146	0.2789	11.53
`Native`	20	3.446	0.0167	0.0156	0.44
`Managed`	200	6,242.546	25.2918	22.4205	1.00
`Managed_Wrapped`	200	22,026.703	97.9251	91.5992	3.53
`Managed_Pinvoke`	200	83,690.755	550.2520	487.7840	13.41
`Native`	200	2,413.073	5.9563	5.5715	0.39

Figure 3.8: Benchmark results for Gaussian elimination implementation.

These results show that the overhead of PInvoke would be certainly noticeable if it were used for each arithmetic operations and could slow down the simulation as much as an order of magnitude. To provide a way to avoid this overhead, we will implement the numeric routine for solving the equation system in both .NET and C++. This way, the transition between runtimes would occur only once per equation system solution. The choice of whether managed or native version would be used will depend on another conditional compilation symbol.

[4]The benchmarks were run on system with i5-6300HQ 2.30 GHz CPU using .NET Core 2.0.6 (CoreCLR 4.6.26212.01, CoreFX 4.6.26212.01), 64bit RyuJIT, Release mode.

3.5 NextGenSpice.LargeSignal

The `NextGenSpice.LargeSignal` assembly contains the implementation of transient analysis, implementation of which we set as goal 1b. To understand the motivation for choices made in our analysis, we first describe the process of transient analysis in greater detail. Then we proceed with the actual implementation analysis of transient analysis in the NextGen SPICE library.

3.5.1 Transient Analysis Overview

Transient analysis models the circuit's behavior over time. It is used to calculate values of node voltages and currents flowing through the circuit network (which is shortly referred to as the *DC bias* of the circuit) at specified timepoints in the simulated period. The top-level illustration of the simulation algorithm is shown in figure 3.9.

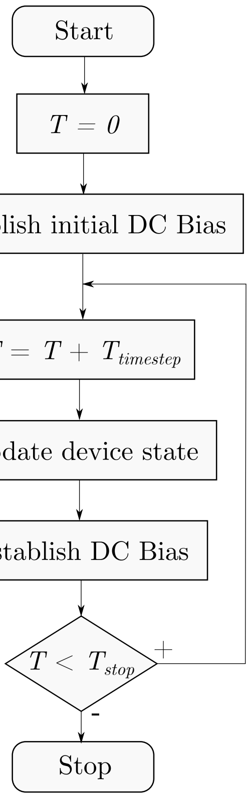


Figure 3.9: Top level description of the transient analysis algorithm

The simulation starts by calculating the initial DC bias of the circuit, during which the capacitor and inductor devices are modeled as ideal open circuits and closed circuits, respectively. After the initial state of the circuit is established, the state of of active devices like capacitors and inductors, or nonconstant voltage and current sources is updated to reflect the time passed between the two successive timepoints. To reflect the energy stored in the capacitor and inductor devices, these devices are replaced by so called *companion models*, which consist of either voltage or current source and a resistor. Parameters of the devices in the companion models are recomputed in each succesive timepoint (technical details will be explained later). Then a DC bias calculation is performed for the next timepoint and the process is repeated for the whole simulated period.

The details of DC bias calculation are described in the following subsections.

DC Bias Calculation

We will demonstrate the DC bias calculation on the circuit shown in figure 3.10. Different colors of the devices will serve a purpose later.

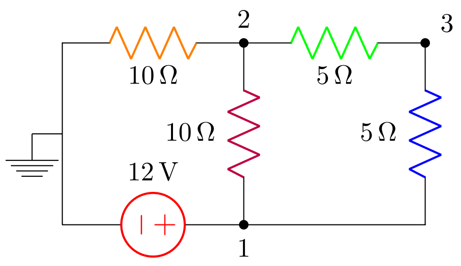
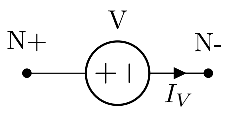
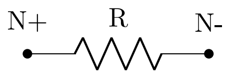


Figure 3.10: Example circuit for DC bias calculation.

To calculate the node voltages and branch currents, the equations corresponding to the Kirchhoff's circuit laws must be formulated. There are several methods for algorithmic formulation of the circuit equations. As an example, we will show the *Modified Nodal Analysis* (MNA) method. In MNA, there is a template for each device type's contribution to the equation system, which is called a device's *stamp*. Device stamps for constant voltage source and resistor are shown in figure 3.11. An important thing to notice is that voltage sources (and some other devices) require an additional variable in the equation system for calculating the current flowing through the device.





$$\begin{bmatrix} & & 1 \\ & & -1 \\ 1 & -1 & \end{bmatrix} \cdot \begin{bmatrix} V_{N+} \\ V_{N-} \\ I_V \end{bmatrix} = \begin{bmatrix} \\ \\ V \end{bmatrix} \qquad \begin{bmatrix} \frac{1}{R} & -\frac{1}{R} \\ -\frac{1}{R} & \frac{1}{R} \end{bmatrix} \cdot \begin{bmatrix} V_{N+} \\ V_{N-} \end{bmatrix} = \begin{bmatrix} \end{bmatrix}$$

Figure 3.11: Device stamps for the voltage source and resistor

Formulation of the circuit using MNA is done by iterating through the list of circuit devices and "stamping" them into the equation system. When applied to the circuit in figure 3.10 the MNA creates following the equation system shown in figure 3.12. Contributions from each device is shown in the same color as the device.

Because node 0 corresponds to the ground node, it always holds that $V_0 = 0$. Therefore, the corresponding row and column can be eliminated. By solving the resulting equation system, we obtain values $V_1 = 12$, $V_2 = 10$, $V_3 = 8$ and $I_V = -0.8$. Branch currents flowing through the resistors can be then trivially calculated using the formula $I_R = U_R/R$.

$$\begin{bmatrix} \frac{1}{10} & 0 & -\frac{1}{10} & 0 & 0 & 0-1 \\ 0 & \frac{1}{10}+\frac{1}{5} & -\frac{1}{10} & -\frac{1}{5} & 0 & 1 \\ -\frac{1}{10} & -\frac{1}{10} & \frac{1}{10}+\frac{1}{10}+\frac{1}{5} & -\frac{1}{5} & 0 & 0 \\ 0 & -\frac{1}{5} & -\frac{1}{5} & \frac{1}{5}+\frac{1}{5} & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \\ I_V \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 12 \end{bmatrix}$$

Figure 3.12: Equation system after stamping devices from our circuit

DC Bias Calculation - Nonlinear Devices

In the preceding example, only *linear* devices were used in the circuit. This means that the resulting equation system was also linear and an exact solution could be obtained. Another situation arises if a nonlinear device such as diode is used in the circuit. A semiconductor diode has nonlinear I-V characteristic which is approximated by the Shockley equation shown in figure 3.13.

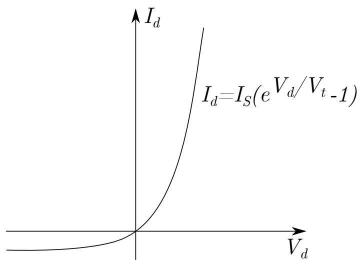


Figure 3.13: I-V characteristic of the diode

Circuits containing such devices can no longer be characterized by a system of linear equations. Instead, a system of nonlinear equations needs to be solved. Such systems are solved iteratively by Newton-Raphson method. In DC bias Calculation, Newton-Raphson algorithm is realized by repeatedly linearizing the I-V characteristics of nonlinear devices at the current candidate DC bias and solving the linearized equation system, to obtain next candidate DC bias. Figure 3.14 shows the linearization process on the example of diode from figure 3.13 around V_B, which is current guess of the voltage across the diode. The linearized I-V characteristic then corresponds to replacing the diode by an *equivalent circuit* consisting of current source with I_{eq} current and resistor with G_{eq} conductance, as shown in the figure.

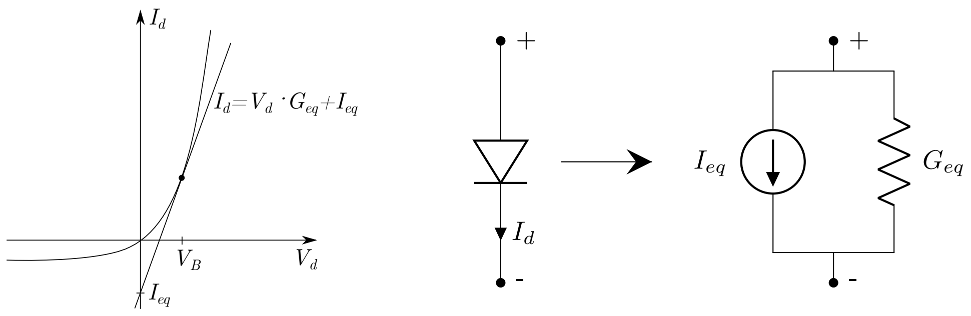
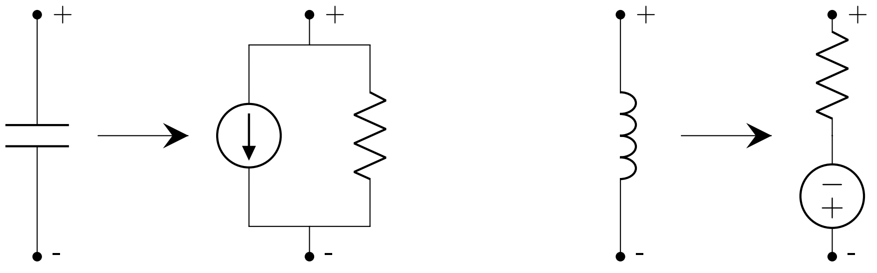


Figure 3.14: Linear equivalent circuit for the diode

This iterative process stops when the difference between the diode currents in two consecutive solutions fits in the relative and absolute tolerances, which are parameters of the simulation. Another simulation parameter is the upper limit on the Newton-Raphson iterations. If the solution does not converge until the specified limit, then the simulation is aborted.

DC Bias Calculation - Energy Storage Devices

We have already briefly mentioned at the beginning of the subsection that capacitor and inductor devices are modeled by their companion models, which characterize the I-V characteristic of the device for the current timepoint. Figure 3.15 shows the companion models for the capacitor and inductor devices.



(a) Capacitor (b) Inductor

Figure 3.15: Companion models of capacitor and inductor

Parameters of the companion models are updated each timepoint via *numerical integration* to reflect the energy stored in the device. However, if the timestep used in the simulation is too big, the local errors made during the integration may accumulate and therefore produce unrealistic results of the simulation. On the other hand, smaller timestep means that the simulation will take longer. Modern simulators internally choose the timestep dynamically: small timestep values when the values of companion models change rapidly, and greater timestep values otherwise. This way the simulator preserves both speed and accuracy. This process is illustrated in figure 3.16.

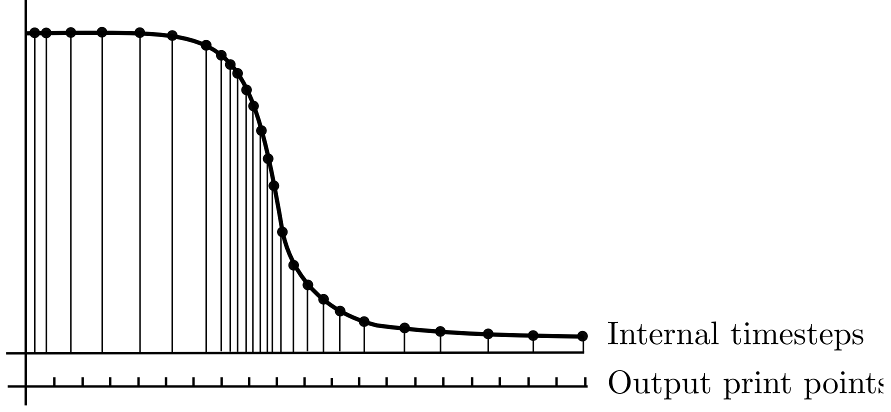


Figure 3.16: Depiction of dynamic timestep mechanism, reproduced from Inside SPICE [3]

This process is realized by iterating individual devices and estimating the maximum timestep for which the truncation error does not exceed the simulator tolerances. Minimum of these values is then chosen as the next timestep.

To implement the transient analysis as described so far, we need to make important decisions regarding individual parts of the algorithm. The main parts that need to be analyzed are the method used for equation system formulation, representation of the equation system, choice of numerical integration method, choice of timestep control mechanism and interface that will be required from device logic implementations.

3.5.2 Choice of Equation System Formulation

As we mentioned in the previous subsection, there are several methods for automated creation of the circuit equation system. Many of those are described in great depth in Nagel's PhD thesis [19], chapter III.

Nagel includes the comparison of the methods with respect to the size of the equation and the programmatic effort of implementing these methods. His research shows that the MNA method which was used in the example in section 3.5.1 produces comparatively smaller equation systems than other methods.

Another advantage of using MNA method is that it does not require finding loops and trees in the circuit graph, which are required by methods like Modified Tableau Analysis. Instead, as we have shown earlier, all devices contribute to the equation system via a device type stamp. Also, adding a new device type does not require changes to the formulation method, because the device stamp can be made part of the device's implementation.

For the reasons above, we decided to use the MNA method during the transient analysis.

3.5.3 Equation System Abstraction

There are multiple ways of representing the equation system. The most straightforward way is representing the equation matrix as full two-dimensional array. Such representation is intuitive and easy to manipulate in the program. However, if the equation matrix is sparse (many coefficients are equal to 0), this representation is inefficient. When using MNA, the number of nonzero elements in a row is roughly proportional to the number of devices that connects to that node. This means that in large circuits, where only small number of devices connects to the same node at once, it would be more appropriate to use sparse matrix representation.

We decided to use the full 2D array representation for simplicity, but in case that this representation would prove to be an issue in the future, we would like to change the implementation without affecting the user code. Therefore, we have to expose the equation system under an interface that can be efficiently implemented by sparse matrix representations too.

There is also an additional requirement on the equation system implementation. As we have shown in the figure 3.11, devices may need an additional circuit variable to be correctly simulated. The equation system implementation therefore needs to support adding additional variables at least during the initialization phase of the simulation.

Because we decided to use MNA method for equation system formulation, each device needs to modify only a small fixed set of equation system coefficients corresponding to the device's stamp. We have therefore decided to use an interface illustrated in figure 3.17. During the initialization phase of the simulation, each device would request any number of additional variables that it requires, and specify which equation system coefficients it contributes to. Access to these coefficients will be then provided using a proxy object for each accessed coefficient.

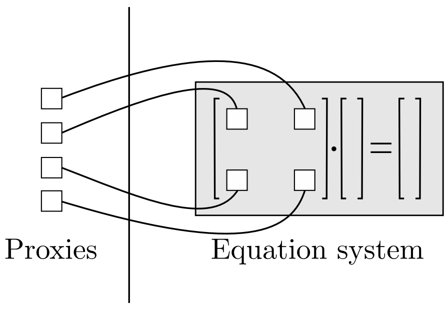


Figure 3.17: Equation system abstraction

This interface is general enough to allow any internal equation system representation while providing efficient access to individual coefficients.

3.5.4 Choice of Numerical Integration Method

In the description of transient analysis of energy storage devices, we mentioned that the inner state of the device is updated based on the circuit state of the previous timepoint using numerical integration. Commonly used integration methods include Backward Euler, Trapezoidal Rule, and Gear method. Summary of these and other numerical integration methods can be found in QUCS Technical Papers, section 6.1. Nagel's PhD thesis also provides detailed analysis of common numerical integration methods in chapter VI. None of these methods can be considered best for the circuit simulation. For example, although the trapezoidal rule is very good in terms of accuracy and speed, it may sometimes lead to a phenomenon called *trapezoidal ringing*, which means that the value oscilates between the exact value. The trapezoidal ringing is shown in figure 3.18.

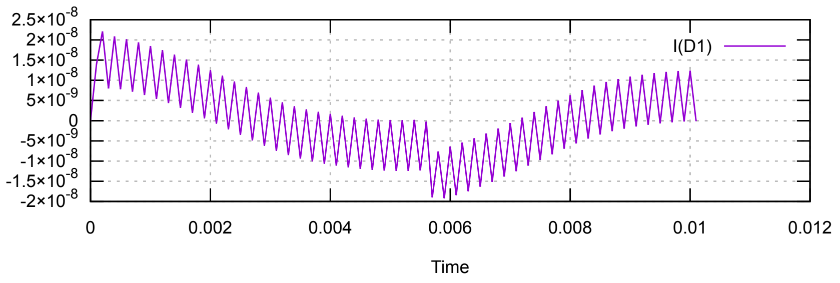


Figure 3.18: Trapezoidal ringing.

For this reason, modern circuit simulators implement multiple integration methods. If the default method proves inappropriate, the user can instruct the simulator to use different method. We would therefore like to allow using different integration methods in our simulator, and even allow user to add new integration methods.

3.5.5 Choice of Timestep

While describing the simulation of energy storage devices back in section 3.5.1, we briefly mentioned that an additional precision can be achieved by choosing the timestep dynamically. The dynamic methods choose the timestep such that error introduced by the integration method is lesser than a certain threshold. The timestep choice therefore depends on the integration method and device's state in previous timepoints.

We decided to simplify the library's implementation and not implement the dynamic timestep in the initial version of the library. Therefore, the library will rely on the user to specify an appropriate timestep value that will be used throughout the simulation. However, the dynamic timestep is an attractive feature and we would like to add it in the next versions of the library.

4. Developer Documentation

The implementation of the NextGen SPICE library is contained in one Visual Studio 2017 solution which consists of 10 projects, the overall structure of the solution is illustrated in figure 4.1.

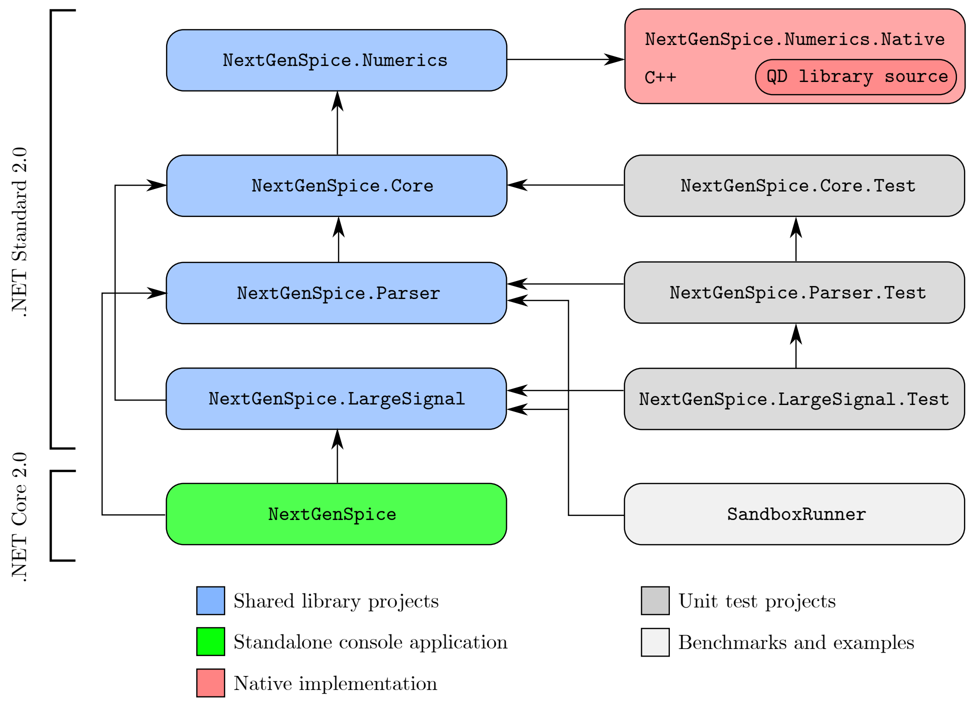


Figure 4.1: Project structure of the solution

`NextGenSpice.Core` project is the core project of the library, it contains classes for creating and validating electrical circuits. It also contains code that automatically discovers available analysis types through the MEF framework, and constructs analysis-specific circuit models from the circuit description.

`NextGenSpice.LargeSignal` project contains implementation of large-signal circuit model which allows performing DC operating point and transient analyses.

`NextGenSpice.Parser` project contains implementation of the SPICE netlist parser and allows users of the library to import circuits, subcircuits and device model parameters into the simulator.

`NextGenSpice.Numerics` project defines classes for creating and representing equation systems for the simulator, as well as other mathematical functions that are needed in the simulator. It also serves as a managed wrapper around the `NextGenSpice.Numerics.Native` C++ project, which is used to build the QD library for double-double and quad-double arithmetic, and expose it's methods to managed (C#) code.

`NextGenSpice` represents the standalone console application, implementation of which was the goal 2 of this thesis. It is also the only library project which targets the .NET Core platform.

The other projects are not part of the distributed library or console application. The `SandboxRunner` console application project is used to run benchmarks for the circuit simulator, and the projects with `.Test` suffix contain unit tests for individual parts of the library.

License

The whole NextGen SPICE solution is provided under the MIT license. More details can be found in `LICENSE.TXT` file in the solution folder (located in `/sources` folder in the attached CD).

4.1 Compilation

To allow using enhanced precision types in the simulator library, the solution contains one C++ (`NextGenSpice.Numerics.Native`), which potentially inhibits the portability of the library. As discussed in analysis section 3.4, we decided that the use of C++ code should be conditional on presence of compile time symbols. The use of native code depends on the conditional compilation symbols used when compiling the `NextGenSpice.Numerics` project. In Visual Studio 2017, these symbols can be set in project properties → Build → Conditional compilation symbols. Following tables lists the symbols and their meaning for the compilation.

Symbol	Precision type used
`dd_precision`	Use the `dd_real` type and double-double arithmetic
`qd_precision`	Use the `qd_real` type and quad-double arithmetic
no symbol	Use the `double` type

Symbol	Choice of Gauss elimination implementation
`native_gauss`	Use native implementation
no symbol	Use managed implementation

The `NextGenSpice.Numerics.Native.dll` dll is defaultly compiled for 64-bit runtime. To use NextGen SPICE library in 32-bit process, 32-bit version of the native dll must be compiled.

4.2 NextGenSpice.Core

The `NextGenSpice.Core` assembly contains the analysis independent parts of the NextGen SPICE library, namely the classes for circuit description, logic for validating the circuit, and generic factory to be used for creating analysis-specific circuit models.

4.2.1 Circuit Description

In the analysis section 3.2.1, we decided to represent the circuit description using a separate class for each device. These classes implement the `ICircuitDefinitionDevice` which defines the members `ConnectedNodes` and `GetBranchMetadata` which are used to validate the circuit topology. It also defines members `Tag` which can be used to identify individual devices and `Clone` method for duplicating the device.

The `ConnectedNodes` property returns an instance of `NodeConnectionSet` class which encapsulates the node connections. This class has `internal` setters, so that the connections are set only by the library and cannot be modified by the user.

The `GetBranchMetadata` is used to retrieve the characteristics of the connections which are important for the circuit topology validation. The topology rules require that there is no cycle of voltage sources and inductors, and no cutset of current sources and capacitors. The `GetBranchMetadata` therefore returns collection of `CircuitBranchMetadata` which contain information whether the device is *current defined* or *voltage defined*. If device is neither (e.g. resistor), then the `GetBranchMetadata` returns empty collection.

The circuit description is contained in the `CircuitDefinition` class, which simply wraps the collection of `ICircuitDefinitionDevice` and provides other convenient methods (such as `FindDevice` to find a device by its tag). This class also has `internal` constructors, so that it can be created only by the library's code. The circuit definition is created by the `CircuitBuilder` class, which is also responsible for validating the circuit's topology.

The SPICE subcircuit representation consists of two parts: a `Subcircuit Definition` class which contains the description of the subcircuit in a similar manner as the `CircuitDefinition` class, and the `Subcircuit` class which implements the `ICircuitDefinitionDevice` interface and represents the subcircuit usage in the circuit. The `SubcircuitDefinition` instances are shared among all coresponding `Subcircuit` classes.

4.2.2 General Analysis Implementation Design

Before we cover the mechanism which creates the analysis-specific circuit models from the circuit description, we will describe the interface hierarchies between the circuit description and analysis implementation. The `ICircuitDefinition` and `ICircuitDefinitionDevice` interfaces were described in previous subsection. The interfaces on the analysis implementation counterparts are `IAnalysis CircuitModel<TDevice>` and `IAnalysisDeviceModel<TAnalysis>`.

Because the relationship between these two interfaces is not trivial, we will explain them in the context of large-signal analysis model implementation. Figure 4.2 shows the relationship between the classes and interfaces for circuit description and large-signal circuit model.

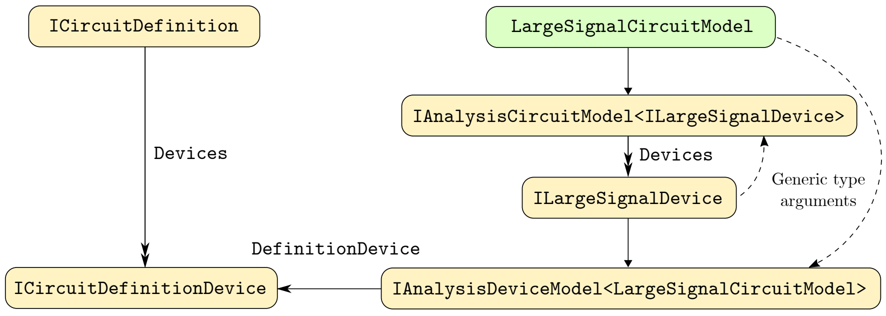


Figure 4.2: Connection between the circuit description and analysis implementation

The implementations of concrete analysis types are expected to extend the IAnalysisDeviceModel interface by adding methods required from the device's implementation. The additional methods for large-signal analysis are defined in ILargeSignalDevice which extends the IAnalysisDeviceModel<LargeSignal CircuitModel> interface. The generic argument in the IAnalysisDeviceModel interface is not used in the defined members, but provides metadata useful for enforcing that only the classes implementing ILargeSignalDevice interface can be registered as device models for the LargeSignalCircuitModel (the model registration and analysis model factories will be covered in later subsection). The IAnalysisDeviceModel interface defines the DefinitionDevice property which serves as the link to the device from the original circuit description. The classes implementing the device-specific simulation logic will read the device's parameters from it's definition device, which allows reacting to the changes of the device's parameters, as requested by goal 1b.

The IAnalysisCircuitModel interface has generic parameter for the base type of the device's implementation, which is also the type of the items in it's Devices property. The LargeSignalCircuitModel class requires each device to implement the ILargeSignalDevice interface.

4.2.3 Analysis Model Creation

The creation of analysis-specific circuit models is done via a set of factories. Each analysis type assembly (currently, only NextGenSpice.LargeSignal is implemented) exports an implementation of IAnalysisModelFactory<T> specialized on the class which will be used to represent the circuit for the particular analysis. In case of NextGenSpice.LargeSignal, the class LargeSignalAnalysisModel Factory implements the IAnalysisModelFactory<LargeSignalCircuitModel> interface and is used to build LargeSignalCircuitModel instances.

The core functionality for registering models for individual circuit devices is implemented in the abstract AnalysisModelFactory<T> class. The registering is done by calling the generic method SetModel<TDesc, TImpl> specialized on definition device type and implementation device type. This function takes a factory function as a parameter, which will be later used to construct the implementation class. For example, the registering of LargeSignalResistor as the implementation for Resistor device is done by calling

```
IAnalysisModelFactory<LargeSignalCircuitModel> factory = /*...*/;
factory.SetModel<Resistor, LargeSignalResistor>(
    e => new LargeSignalResistor(e));
```

Thanks to the interface definitions as described in the previous section, it is possible to constrain the type arguments of the SetModel method to statically check that the LargeSignalResistor class implements IAnalysisDeviceModel <LargeSignalCircuitModel> interface (which it does by implementing ILarge SignalDevice).

When a new circuit analysis model is to be created, all circuit devices from the ICircuitDefiniton.Devices collection are transformed by applying the supplied factory methods. Actual instantiating of the circuit model classes is delegated to the classes deriving from the AnalysisModelFactory<T> via protected abstract method CreateInstance. In the implementation of the LargeSignal AnalysisModelFactory, this method return a new instance of LargeSignal CircuitModel class.

4.2.4 Discovering Analysis Model Implementations

To simplify management of existing IAnalysisModelFactory<T> interface implementations, we introduced AnalysisModelCreator class. This class defines the Create<TAnalysis> method, which finds appropriate factory which to be used for creating the TAnalysis circuit model. The factory can be either registered manually using the SetFactory<TAnalysis> method, or automatically by adding [Export(typeof(IAnalysisModelFactory<T>))] attribute to the factory class. The implementation is then discovered by MEF framework.

To change the mapping between the circuit devices and their implementations, the respective factory can be obtained by GetFactory<TAnalysis> method. The AnalysisModelCreator also defines static property Instance which holds the global singleton, however, it is possible to create multiple instances of this class and have different mappings in each instance.

To simplify the library's interface for scenarios where only the global Analysis ModelCreator instance is needed, implementations of the circuit analyses are encouraged to define an extension function on ICircuitDefinition which provides simple way of creating analysis models using the global instance. For example, NextGenSpice.LargeSignal assembly defines following extension method.

```
public static LargeSignalCircuitModel
GetLargeSignalModel(this ICircuitDefinition definition)
{
    if (definition == null)
        throw new ArgumentNullException(nameof(definition));
    return AnalysisModelCreator.Instance
        .Create<LargeSignalCircuitModel>(definition);
}
```

4.3 NextGenSpice.Parser

As we stated in the analysis chapter in section 3.3, we have decided to implement both the parser and lexer manually. The implementation is contained in the NextGenSpice.Parser project. The whole functionality is exposed through the instance methods of SpiceNetlistParser class.

The class itself does not contain code for handling specific netlist statements, processing these statements is delegated to *statement processors*, which will be described later in greater detail. A new instance of the parser can be obtained by SpiceNetlistParser.Empty() method. The returned instance does not contain any statement processors, these would have to be registered manually. For convenience, the parser class contains a static method SpiceNetlistParser .WithDefaults(), which creates a new instance of the parser and automatically registers statement processors implemented as part of this thesis.

The parsing itself is done simply by calling Parse() method, which accepts an instance of TextWriter class from which the netlist should be read. this methods returns an instance of `SpiceNetlistParserResult` class, which encapsulates all the information from the parsed file: `CircuitDescription` for the contained circuit, collections of defined models and subcircuit, used node names, encountered errors etc.

All information regarding the netlist that is currently being parsed is aggregated in the `ParsingContext` class. This class contains information such as list of already parsed statements, instance of `CircuitBuilder` for constructing a circuit, and reference to an instance `SymbolTable` holding all so far encountered devices, models and subcircuits.

The general algorithm for parsing SPICE netlists consists of following steps:

1. Reading the Title statement from the input file
2. Reading and processing each SPICE statement by following steps
   (a) Tokenizing the statement
   (b) Determinig the statement type and finding suitable statement processor
   (c) Processing the statement by making changes to `ParsingContext`
3. Constructing the circuit description and returning the results in an instance of `SpiceCodeParserResult` class

4.3.1 Tokenizing

The tokenizing of the statement is done using the `TokenStream`, which is a wrapper around the input `TextWriter` instance. The main purpose of this class is to read and return tokens that form individual statements, which is done in the `TokenStream.ReadStatement()` method.

One SPICE statement may span multiple lines, where each subsequent line starts with a `+` symbol (see section 2.1). `TokenStream` class joins these lines together, annotates individual `string` tokens with line numbers and line offsets, and returns them as an `IEnumerable<Token>` instance.

4.3.2 Processing Statements

The SPICE netlist statements can be divided into two distinct sets of statements. The first are *device statements*, which always begin with a letter (which then determines the device's type), and *dot statements*, which begin with a `.` character followed by name of the statement, which can be arbitrary alphanumeric string, for example `.MODEL` statement. Device statements can be distinguished by inspecting only the first letter of the statement, but for the other statements, the whole first token must be considered. Therefore, these two types of statements are handled separately by classes implementing the `IDeviceStatementProcessor` and `IDotStatementProcessor` interfaces, which can be added to the parser instance by `RegisterDevice` and `RegisterStatement` methods. The appropriate statement processor is then looked up by comparing with the `Discriminator` property on the statement processor classes from appropriate collection.

Because the statements in the netlist can occur in almost arbitrary order, their processing is not entirely trivial. Consider following legal netlist fragment:

```
R1 2 3 5
D1 2 3 DMOD
.MODEL DMOD D(IS=1p)
```

The first statement states that there is 5Ω resistor between the nodes 2 and 3. This statement can be directly processed and corresponding call to the circuit builder can be made. However, the second statement says that there is a diode between the nodes 2 and 3, and that it's parameters are specified by the `DMOD` model. However, at the time of parsing that statement, the `DMOD` model not known yet, because it is specified after the diode statement. It could be said that the processing of the diode statement is dependent on processing of the corresponding .MODEL statement.

Therefore, before each statement is applied to the circuit, all its dependencies are checked, and if there are unresolved dependencies, the the processing of the statement is deferred until the whole file has been parsed. This is achieved by adding a corresponding class derived from `DeferredStatement` instance to the `ParsingContext.DeferredStatements` collection.

After all the statements have been parsed, each deferred statement is repeatedly checked, and applied if their dependencies have been resolved by applying some other statement. If any statement remains unprocessed, then a corresponding error is recorded to the `ParsingContext.Errors` collection.

There is also a static `ParserHelper` class, which contains extension method `GetNumericValue(this Token, ICollection<SpiceParserError>)` that simplifies parsing numbers and scale factors (see section 2.1.1), and method `ToError (this Token, SpiceParserErrorCode, param object[])` which simplifies creating error messages.

4.4 NextGenSpice.LargeSignal

The `NextGenSpice.LargeSignal` project contains the implementation of the large-signal circuit model which is used to perform transient and DC analysis of circuits. The simulation functionality is exposed through the `LargeSignal CircuitModel` class which implements the simulation algorithm described in section 3.5.1. The simulation is managed by `EstablishDcBias` method, which calculates the circuit state at time 0, and `AdvanceInTime` which advances the state by given timestep. The device-specific simulation logic is delegated to classes implementing the `ILargeSignalDevice` interface. This interface defines the following methods.

- `RegisterAdditionalVariables` – Allows device implementation to add additional variables to the equation system (like branch variable for the voltage source as shown in section 3.5.1).
- `Initialize` – Lets devices request equation system coefficient proxies, and perform other necessary initialization.

- `ApplyModelValues` – Applies devices stamp into the equation system. This is where the nonlinear devices are linearized.

- `OnEquationSolution` – Lets devices update the inner state based on the last solution of the equation system. Also, nonlinear devices should check for solution convergence (compare current solution with the previous one).

- `OnDcBiasEstablished` – Called after the Newton-Raphson iterations have reached a fixed point and the calculation of the current timestep has completed.

The figure 4.3 shows the implemented simulation algorithm and when the respective methods are called. The methods from the `ILargeSignalDevice` interface are in shown in green and are always called for every `ILargeSignalDevice` in the `LargeSignalCircuitModel`.

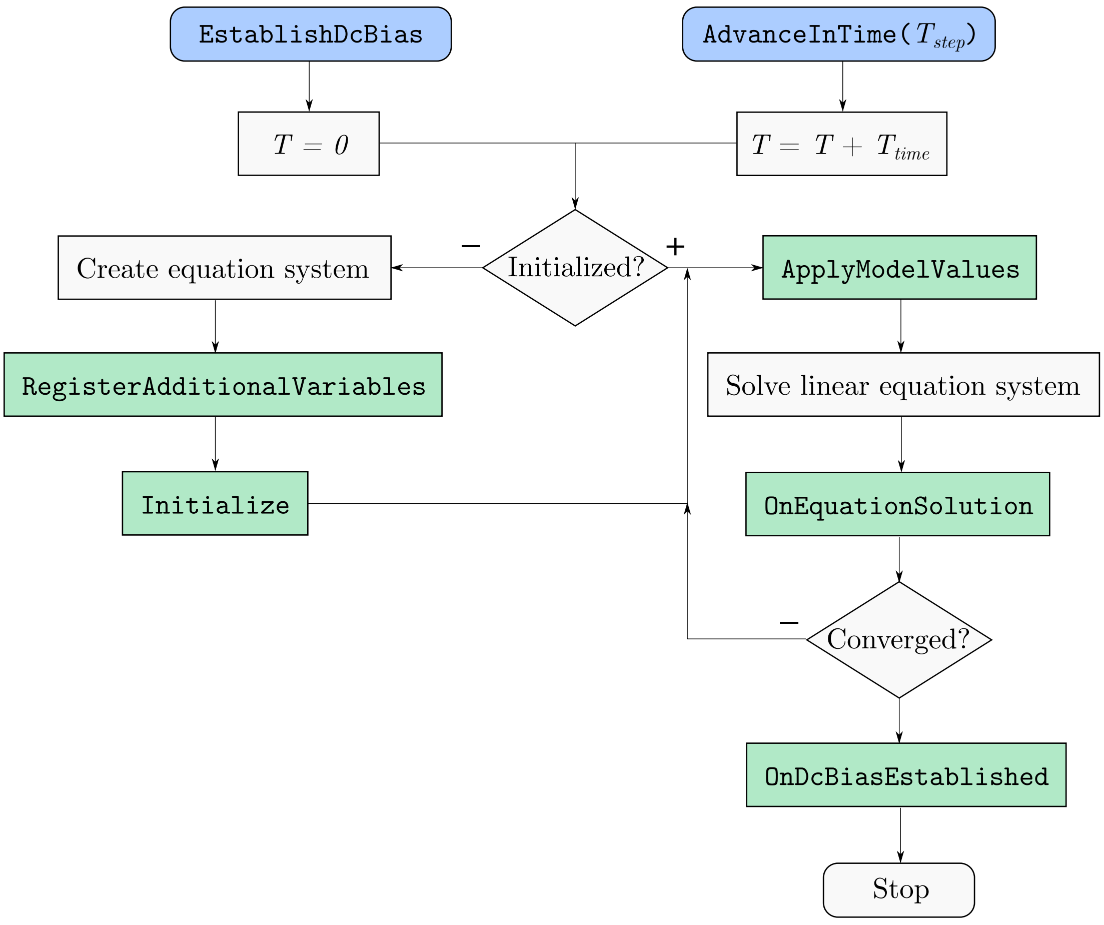


Figure 4.3: Implementation of the simulation algorithm

The following table lists the classes which implement the large-signal logic for individual circuit devices, and corresponding section in QUCS Technical papers which describe the stamps and mathematical models used in the implementation. Also, the models for semiconductor devices (diode and BJT) are described in depth in Semiconductor Device Modeling with SPICE by G. Massobrio [23], ch 1 and 2.

Device class	QUCS section
`LargeSignalResistor`	9.2
`LargeSignalCapacitor`	6.3.1
`LargeSignalInductor`	6.3.2
`LargeSignalVoltageSource`	9.18
`LargeSignalCurrentSource`	9.18
`LargeSignalVccs`	9.20.1
`LargeSignalCccs`	9.20.2
`LargeSignalVcvs`	9.20.3
`LargeSignalCcvs`	9.20.4
`LargeSignalDiode`	10.2
`LargeSignalBjt`	10.4

The actual stamping is delegated to `<device>Stamper` classes to make the implementation cleaner.

4.5 NextGenSpice.Numerics and NextGenSpice.Numerics.Native

`NextGenSpice.Numerics.Native` is the only C++ project in the solution. It is used to build a dynamically linked library, which contains the implementation of QD library used by NextGen SPICE for double-double and quad-double precision types. It also contains native implementation of Gaussian elimination algorithm for solving the equation system to speed up the the simulation (see analysis section 3.4).

`NextGenSpice.Numerics` is the managed counterpart of the `NextGenSpice.Numerics.Native` project. It contains C# wrappers around the `dd_real` and `qd_real` types from QD library, with PInvoke calls to its exported methods. It also contains the managed implementation of gauss-elimination that is used when the library is compiled without the conditional compilation symbols mentioned at the beginning of this chapter.

4.5.1 Equation System Implementation

The `NextGenSpice.Numerics` project contains implementations of the equation system. There is a separate class for equation system for each precision type: `EquationSystem`, `DdEquationSystem` and `QdEquationSystem`. These classes represent the equation system as the full matrix and vectors. To make the interface independent of the implementation and the actual precision type used, the equation system is exposed to through the implementations of `IEquationSystemAdapter` and `IEquationSystemAdapterWide` interfaces. The first interface defines methods for getting proxies proxies for individual equation system coefficients and the equation solution as described in the analysis chapter in section 3.5.3. This interface will be exposed to the device's implementation. The other interface defines another methods that the simulator needs, like `Solve` and `Clear`. The actual implementation of these interfaces is requested from static class `EquationSystemAdapterFactory`. This arrangement was chosen mainly to allow runtime changes of precision type for benchmarking purposes.

If the implementation were to change to a sparse matrix representation, then all needs to be done is implementing the `IEquatioSystemAdapterWide` interface and replacing the class instantiated in the `EquationSystemAdapterFactory`.

4.6 NextGenSpice

This project contains the implementation of the standalone SPICE-like console application. This application directly uses the other parts of the NextGen SPICE library described in the preceding sections. This project implements necessary statement processors to handle `.PRINT`, `.OP` and `.TRAN` statements in the input netlist files. These statements are inserted into `OtherStatements` collection on the `ParsingContext` and later retrieved from the property of same name on `SpiceNetlistParserResult` object. If no error occurs while parsing the netlist, the application executes the requested simulations and prints the results on the standard output.

4.7 SandboxRunner

This project serves only development purposes and is not deployed as part of the library. This project uses BenchmarkDotNet NuGet package to run benchmarks for comparing the individual precision methods. It also contains examples of code which uses the library.

4.8 Unit Test Projects

There are three unit test projects in the solution, `NextGenSpice.Core.Test` for the core simulation library, `NextGenSpice.Parser.Test` for the parser and `NextGenSpice.LargeSignal.Test` for the large-signal simulations. These project use the XUnit NuGet package to run unit tests for the main parts of the tested projects. The unit tests do not aim for strict 100% code coverage (The coverage is around 80% for each project), instead, they test the most important parts of the library to avoid breaking the code when implementing new features to the library.

5. User Documentation - Library

This chapter describes how to use the NextGen SPICE simulator, and provides the guidelines for adding new circuit analyses and device types. The necessary binaries are available in the `/binaries` folder on the attached CD for copying.

5.1 Tutorials

This sections shows the usage of the library on simple examples to present the basic idea of how to use the NextGen SPICE library. All these examples will use the .NET Core Console Application project template. The project should reference the `NextGenSpice.Core` assembly for circuit description, `NextGenSpice.Parser` assembly for parser implementation, `NextGenSpice.LargeSignal` assembly for the actual simulator, and all assemblies with the `System.Composition` preffix from the `/binaries` folder. Also, `NextGenSpice.Numerics.Native.dll` needs to be copied to the same folder as the compiled executable. We will use gnuplot [24] to create the plots of the simulation results, so reader should have it installed as well.

5.1.1 Calculating DC Bias of the Ciruit

Suppose we wanted to calculate node voltages in the circuit shown in figure 5.1.

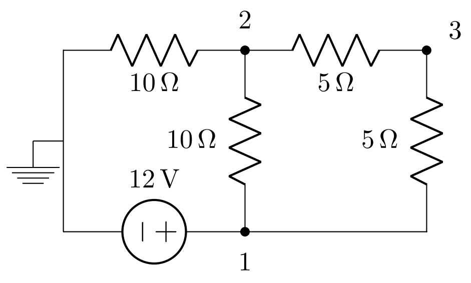


Figure 5.1: Example circuit

Before we start with the actual analysis, we first need to construct the circuit representation. This is done using the `CircuitBuilder` class. Following code fragment constructs the circuit description of our circuit.

```
// requires NextGenSpice.Core.Circuit and
// NextGenSpice.Core.Devices namespace
var builder = new CircuitBuilder();
builder
    .AddDevice(new[] {1, 0}, new VoltageSource(12))
    .AddDevice(new[] {0, 2}, new Resistor(10))
    .AddDevice(new[] {1, 2}, new Resistor(10))
    .AddDevice(new[] {2, 3}, new Resistor(5))
    .AddDevice(new[] {1, 3}, new Resistor(5));
var circuit = builder.BuildCircuit();
```

For convenience, class `CircuitBuilderExtensions` defines extensions methods that can be used to make the code more readable. The circuit can be equivalently created by the following code.

```
// extensinos are contained in
// NextGenSpice.Core.Extensions namespace
var builder = new CircuitBuilder();
builder
    .AddVoltageSource(1, 0, 12)
    .AddResistor(0, 2, 10)
    .AddResistor(1, 2, 10)
    .AddResistor(2, 3, 5)
    .AddResistor(1, 3, 5);
var circuit = builder.BuildCircuit();
```

The `circuit` object created in the preceding code fragments is only a description of the circuit. In the next step, we use it to create it's large-signal model, which we will use to perform the actual analysis.

```
// requires NextGenSpice.LargeSignal namespace
// equivalent to
// var m = AnalysisModelCreator
//       .Instance.Create<LargeSignalCircuitModel>(circuit);
var model = circuit.GetLargeSignalModel();
```

The call to the method `EstablishDcBias` performs the actual node voltage calculation. Calculated node voltages can be found in an array in `LargeSignalCircuitModel.NodeVoltages` property.

```
model.EstablishDcBias();
Console.WriteLine(model.NodeVoltages[1]); // 12
Console.WriteLine(model.NodeVoltages[2]); //  8
Console.WriteLine(model.NodeVoltages[3]); // 10
```

The simulator also automatically calculates data for individual devices, and stores them in their large-signal model instances contained in the `LargeSignalCircuitModel.Devices` collection. In our case, all currents flowing through the circuit devices are calculated. As an example, we show how to get value of current flowing through the voltage source in our circuit.

First, we need to identify the device in the collection. This can be done by providing a *tag* parameter during the circuit representation construction. Arbitrary object can be used as a tag, we will use a string tag.

```
builder
    .AddVoltageSource(1, 0, 12, "VS")
//    equivalent to
//  .AddDevice(new[] { 1, 0 }, new VoltageSourceDevice(12, "VS"))
    ...
```

This tag is then used to query the `LargeSignalCircuitModel.Devices` collection, which stores the implementations of `ILargeSignalDevice` interface for each circuit from the description. For convenience, each analysis model defines method `FindDevice()` to simplify the syntax.

```
// requires NextGenSpice.LargeSignal.Devices namespace
// equivalent to
// var vsource = (ITwoTerminalLargeSignalDevice) model.Devices
//    .SingleOrDefault(dev => Equals(dev.DefinitionDevice.Tag, "VS"));
var vsouce = (ITwoTerminalLargeSignalDevice) model.FindDevice("VS");
Console.WriteLine(vsouce.Current); // -0.8
```

The casting to `ITwoTerminalLargeSignalDevice` interface is necessary, because some circuit devices may have more than two terminals (e.g. transistors), and the Current property would note make sense for them. The whole example example then reads as follows:

```
var builder = new CircuitBuilder();
builder
.AddVoltageSource(1, 0, 12, "VS")
.AddResistor(0, 2, 10)
.AddResistor(1, 2, 10)
.AddResistor(2, 3, 5)
.AddResistor(1, 3, 5);
var circuit = builder.BuildCircuit();

var model = circuit.GetLargeSignalModel(); ;
model.EstablishDcBias();

Console.WriteLine(model.NodeVoltages[1]); // 12
Console.WriteLine(model.NodeVoltages[2]); //  8
Console.WriteLine(model.NodeVoltages[3]); // 10

var vsouce = (ITwoTerminalLargeSignalDevice) model.FindDevice("VS");
Console.WriteLine(vsouce.Current); // -0.8
```

5.1.2 Performing Transient Analysis

In previous section, we showed how to use the library to compute DC Bias of the circuit. Now we will show how to perform transient analysis. Consider the circuit shown in figure 5.2.

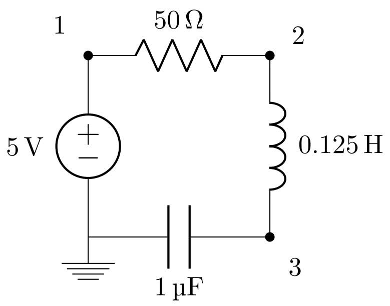


Figure 5.2: Simple series RLC circuit.

We will calculate how the circuit behaves if the 5 V from the voltage source come in a sudden pulse. The NextGen SPICE library supports many input source behaviors, the full list can be found in section 5.2.2. We will specify the desired behavior by passing an instance of `PulseBehavior` to voltage source when building the circuit.

```
var circuit = new CircuitBuilder()
    // behaviors are located in NextGenSpice.Core.BehaviorParams namespace
    .AddVoltageSource(1, 0, new PulseBehavior()
    {
        InitialLevel = 0,
        PulseLevel = 5,
        Delay = 5e-3, // 5 ms
        PulseWidth = 25e-3 // 25 ms
    })
    .AddResistor(1, 2, 50)
    .AddInductor(2, 3, 0.125)
    .AddCapacitor(3, 0, 1e-6)
    .BuildCircuit();
```

Then, as in the previous example, we get the `LargeSignalCircuitModel` and calculate the initial state of the circuit using the `EstablishDcBias()` method. After that, we can call the `AdvanceInTime()` method to perform the timesteps. Following code fragment can be then used to print voltage values of nodes 1 and 3 over time.

```
var model = circuit.GetLargeSignalModel();
model.EstablishDcBias();

Console.WriteLine("Time V(1) V(3)");

var timestep = 0.2e-3; // use 0.2 ms timestep
while (model.CurrentTimePoint <= 55e-3) // simulate for 55 ms
{
    var time = model.CurrentTimePoint;
    var v1 = model.NodeVoltages[1];
    var v3 = model.NodeVoltages[3];

    Console.WriteLine($"{time} {v1} {v3}");

    model.AdvanceInTime(timestep);
}
```

If we redirect the program output to a file named `output.txt`, we can run gnuplot and use following commands to create a `plot.svg` file with plot of the voltage values over time as shown in figure 5.3.

```
set terminal svg size 600, 250
set output 'plot.svg'
set key autotitle columnhead
set xlabel 'Time'
set ylabel 'Value'
set grid ytics lt 0 lw 1 lc rgb '#bbbbbb'
set grid xtics lt 0 lw 1 lc rgb '#bbbbbb'
plot for [i=2:3] 'output.txt' using 1:i with lines
```

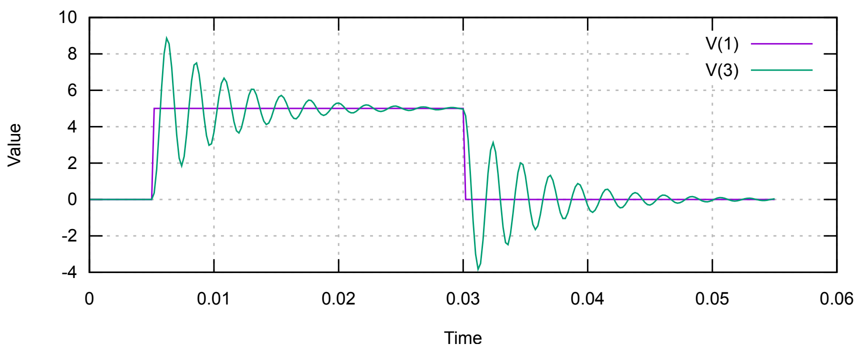


Figure 5.3: Results on the RLC circuit.

5.1.3 Loading Circuits from Netlists

The NextGen SPICE library supports loading circuit description from SPICE netlist files. The supported syntax is shown in chapter 2. We will demonstrate this on the circuit shown in figure 5.4. This is the same circuit we have shown earlier in the introduction chapter because of it's $1\mu\Omega$ resistor.

```
Back to Back diodes
*
V1 IN 0 SIN(0 5 100 0 0 0)
D1 IN A D1N4148
R1 A B 1e-6
D2 0 B D1N4148
*
.MODEL D1N4148 D (
+ IS=2.52e-9 N=1.752 TT=2e-8
+ CJO=9e-13 M=0.25 VJ=20 BV=75
+ RS=0.568)
```

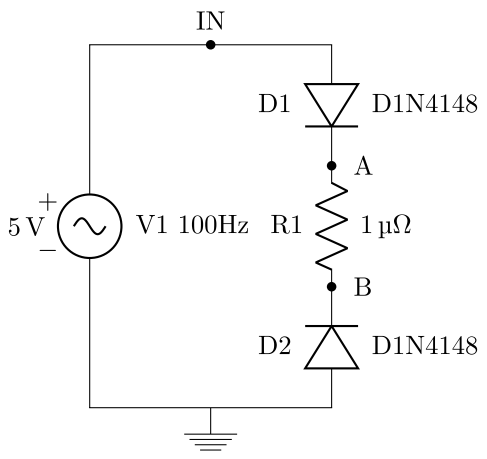


Figure 5.4: Back to back diode circuit and corresponding netlist

When imported from a netlist file, each device is automatically tagged by its name in uppercase letters so that it can be found among the other devices. Following code fragments prints the current flowing through the `D1` diode, and figure 5.5 shows the plot of the data.

```
// parser is located in NextGenSpice.Parser namespace

// import circuit definiton from the file
var parser = SpiceNetlistParser.WithDefaults();
var result = parser.Parse(new StreamReader("circuit.cir"));
var circuit = result.CircuitDefinition;

// get simulation model
var model = circuit.GetLargeSignalModel();
var d1 = (ITwoTerminalLargeSignalDevice) model.FindDevice("D1");
var inNode = result.NodeIndices["IN"];

```

```
Console.WriteLine("Time V(IN) I(D1)");

var timestep = 10e-6; // use 10 us timestep
while (model.CurrentTimePoint <= 10e-3) // simulate for 10 ms
{
    var time = model.CurrentTimePoint;
    var vin = model.NodeVoltages[inNode];
    var id1 = d1.Current;

    Console.WriteLine($"{time} {vin} {id1}");

    model.AdvanceInTime(timestep);
}
```

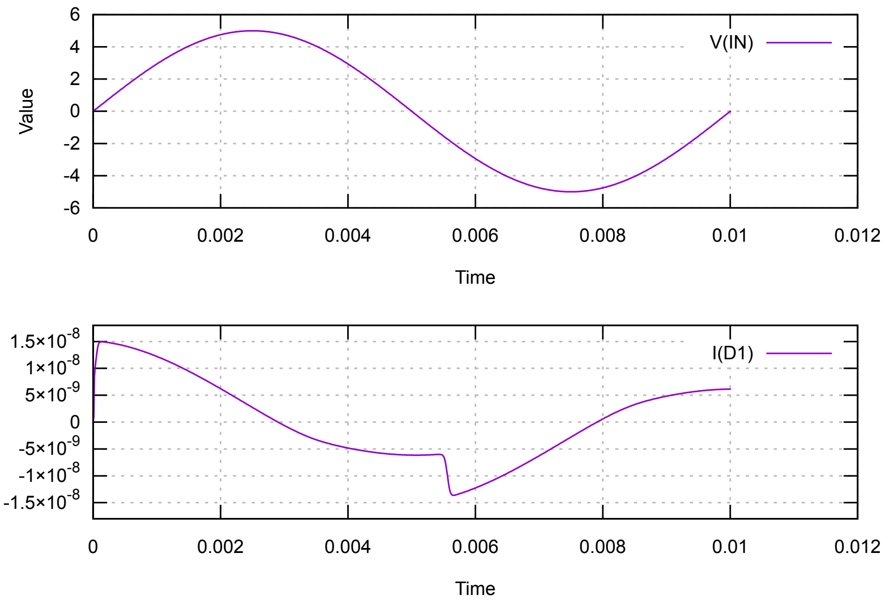


Figure 5.5: Plot of simulation output of back-to-back diode circuit.

5.1.4 Defining a Subcircuit in Source Code

NextGen SPICE support defining a custom subcircuit, which then can be used multiple times throughout the circuit and even in different circuits. In this tutorial, we will define a subcircuit representing a $9V$ battery with 1.5Ω internal resistance, which is modelled as $9V$ voltage source and 1.5Ω resistor in series, as shown in figure 5.6.

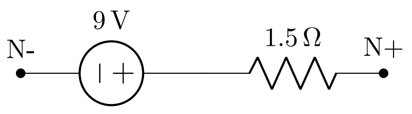


Figure 5.6: Subcircuit for $9V$, 1.5Ω battery

The subcircuit is created using the `CircuitBuilder` class, just as if it were a normal circuit, only instead of `BuildCircuit()` method, the `BuildSubcircuit()`

method needs to be called with an array specifying which subcircuit nodes should be treated as terminals. The battery definition then can be used as an argument to `AddSubcircuit()` extension method, or passed to the `SubcircuitDevice` class constructor. Following code fragment constructs part of a circuit with two $9V$ batteries in series.

```
var builder = new CircuitBuilder();
var batteryDefinition = builder
    .AddVoltageSource(1, 2, 9)
    .AddResistor(2, 3, 1.5)
    .BuildSubcircuit(new [] {1, 3});

builder.Clear();
builder
    .AddDevice(new[] {0, 1}, new Subcircuit(batteryDefinition))
    .AddSubcircuit(new[] {1, 2}, batteryDefinition);
    ...
```

It is also possible to inspect the state of the devices inside the subcircuit during simulation. The `SubcircuitDevice`'s counterpart in `LargeSignalDeviceModel.Devices` implements `ILargeSignalSubcircuit` interface, which exposes the devices inside via the `Devices` property.

In this section, we described the usage of the simulator on simple examples, in next section, we revisit individual parts of the simulator and provide detailed description of it's interface.

5.2 NextGenSpice.Core

The `NextGenSpice.Core` assembly contains the analysis independent parts of the NextGen SPICE library, namely the classes for circuit description, and logic for validating the circuit.

5.2.1 Creating the Circuit Description

As we briefly described in the tutorials in previous section. The NextGen SPICE library works by first creating an analysis-independent `CircuitDefinition` class, which is then transformed into the analysis-dependent circuit model. The description of the circuit (the `CircuitDefinition` class) is created using the `CircuitBuilder` class using the `Add(int[] terminals, ICircuitDefinitionDevice device)` method. The circuit builder then saves the reference to the device and sets node connections appropriately. If multiple copies of the same device should be added to the circuit, it is necessary to clone the device via the `ICircuitDefinitionDevice.Clone` method.

For convenience, there is a static class `CircuitBuilderExtensions` which contains extension methods on `CircuitBuilder` like `AddResistor`, `AddDiode` etc., which can be used to shorten the code that adds the individual devices.

5.2.2 Supported Circuit Devices

Following table lists devices available in the NextGen SPICE simulation library.

Device	Nodes	Parameters
`Resistor`	N_+, N_-	Resistance
`Capacitor`	N_+, N_-	Capacitance, initial voltage
`Inductor`	N_+, N_-	Inductance, initial current
`VoltageSource`	N_+, N_-	Voltage or `InputSourceBehavior`
`CurrentSource`	N_+, N_-	Current or `InputSourceBehavior`
`Vccs`	N_+, N_-, N_{Ref+}, N_{Ref-}	Gain
`Vcvs`	N_+, N_-, N_{Ref+}, N_{Ref-}	Gain
`Cccs`	N_+, N_-	`VoltageSource`, gain
`Ccvs`	N_+, N_-	`VoltageSource`, gain
`Diode`	N_+, N_-	`DiodeParam`
`Bjt`	N_C, N_B, N_E, N_S	`BjtParam`

The parameters listed in the table are set in the class constructor and directly map to the SPICE netlist statement parameters which were described in depth in section 2.2. The `DiodeParam` and `BjtParam` classes encapsulate the model parameters for the diode and BJT devices, and again map to the same parameters as in the netlist `.MODEL` parameters. Additionally, each device class accepts an optional `object` parameter as its tag.

The `InputSourceBehavior` class is the base class for several source behavior specification classes. Again, these classes directly map to the transient input sources from the SPICE netlist syntax, as described in 2.2.4. The supported behaviors are listed in the following table.

Behavior class	Netlist	Description
`ConstantBehavior`	DC	Constant input source value
`SinusoidalBehavior`	SIN	Sinusoidal source
`PieceWiseLinearBehavior`	PWL	Arbitrary piece-wise linear source
`PulseBehavior`	PULSE	Source with regular pulses
`ExponentialBehavior`	EXP	Source with exponential edges
`AmBehavior`	AM	Amplitude modulated source
`SffmBehavior`	SFFM	Single frequency modulated source

5.2.3 Creating Analysis-Specific Circuit Models

Before any circuit analysis can be performed, the circuit definition must be transformed into the analysis-specific circuit model. NextGen SPICE currently supports only the `LargeSignalCircuitModel` which will be described in section 5.4 in more detail.

The circuit models are created using the `AnalysisModelCreator` class and its instance method `GetModel<TCircuitModel>(ICircuitDescription)`. This class encapsulates `IAnalysisModelFactory<TCircuitModel>` implementations for individual analysis model types. Factory implementations for analysis models provided by NextGen SPICE library are automatically registered using the MEF framework. These factories can be configured to use specific implementations for individual circuit devices. The configuration is done on the factory itself, which can be obtained using the `AnalysisModelCreator.GetFactory<TCircuit Model>()`.

There is a global instance of the `AnalysisModelCreator` class available at `AnalysisModelCreator.Instance` property. There is also an extension method on the `ICircuitDefinition` interface which uses the global model creator to create the `LargeSignalCircuitModel`, which can be used to simplify the code.

5.3 NextGenSpice.Parser

The NextGenSpice.Parser assembly contains the implementation of the SPICE netlist parser. The parser itself is represented by the `SpiceNetlistParser` class. The parser is designed to be extensible by registering statement processors for individual netlist statements, which will be covered later. An instance of the parser can be obtained via static methods `SpiceNetlistParser.Empty()` which return an instance of the parser without any statement processors registered, and `SpiceNetlistParser.WithDefaults()` which returns instance with handlers for all devices implemented in the NextGen SPICE library.

The parsing itself is done using the `Parse` method which returns an instance of `SpiceNetlistParserResult` which contains following properties:

- `CircuitDefinition` – Definition of the circuit contained in the netlist.
- `Subcircuits` – Collection of `ISubcircuitDefinition` classes for the subcircuits defined in the top-level of the netlist (Subcircuits defined inside another subcircuit are not returned).
- `Models` – collection of device model (parameter sets) defined in the netlist (again, models defined inside a subcircuit are not returned).
- `Title` – The content of the title statement from the netlist.
- `Errors` – Collection of `SpiceParserError` classes which wrap the errors encountered during the parsing.
- `NodeNames` – NextGen SPICE uses `int` type to store the id of a node, this array can be indexed by the node id to obtain the string name that was used in the netlist.
- `NodeIndices` – Dictionary providing the inverse mapping to `NodeNames` property.
- `OtherStatements` – Collection of `SpiceStatement`s that can be used to return user-defined statements from the parser.

To simplify obtaining references to individual circuit device instances, each instance has been tagged with the uppercase name of the device used in the netlist, so that a particular device can be easily obtained by `FindDevice` methods on circuit definition and circuit models.

5.4 NextGenSpice.LargeSignal

The `NextGenSpice.LargeSignal` assembly contains the implementation of the large-signal circuit model which is used to perform transient analysis of circuits. The simulation functionality is exposed through the `LargeSignalCircuitModel` class, and its `EstalblishDcBias` and `AdvanceInTime` instance methods. The `EstablishDcBias` method is used to calculates the DC bias of the circuit at initial timepoint, and `AdvanceInTime` method is used to get the DC bias after given timestep.

5.4.1 Accessing the Computed State

The calculated node voltages are stored in an array in the `NodeVoltages` property on the `LargeSignalCircuitModel` class. The `Devices` property stores large-signal representation classes as `ILargeSignalDevice` instances for the devices used in the circuit. These instances can be used to inspect the state computed for each circuit device. The `ILargeSignalDevice` instance for a particular device can be obtained by using the `FindDevice` method, which has overloads accepting `ICircuitDefinitionDevice` instance, or an `object` which was used as a tag during circuit construction. The state is exposed in two ways: `GetDeviceStateProviders` method returning a collection of `IDeviceStateProvider` classes, and through the individual properties on the device implementation instance.

The `IDeviceStateProvider` instances can be used to print all the available state variables. The `Name` property gives the name of the variable, like "I" for the current flowing through the device, and `GetValue` method can be used to obtain the respective value.

The same state can be accessed through individual properties on the device implementation class instance. For example, in case of two terminal devices like resistor or voltage source, the respective class implements the `ITwoTerminalLargeSignalDevice` interface, which exposes `Voltage` and `Current` property containing the voltage across the device and current flowing through the device, respectively. The `ITwoTerminalLargeSignalDevice` interface is implemented by all implemented devices except the BJT transistor.

The BJT transistor implementation class `LargeSignalBjt` exposes properties `CurrentBase`, `CurrentCollector` and `CurrentEmitter` which expose the currents flowing through the respective terminals, and `VoltageBaseCollector`, `VoltageBaseEmitter` and `VoltageCollectorEmitter` for voltages between individual pairs of terminals.

5.4.2 Modifying the Device Parameters

The device's implementation used in the `LargeSigalCircuitModel` retain references to the corresponding circuit definition classes. This means that changes made to the device parameters inside `CircuitDefinition` are reflected in the next DC bias point calculation. To demonstrate, recall the circuit we used in the DC Bias calculation tutorial (figure 5.1). The following code snippet prints the DC bias values for increasing values of the rightmost resistor.

```
// build circuit
var builder = new CircuitBuilder();
builder
    .AddVoltageSource(1, 0, 12, "VS")
    .AddResistor(0, 2, 10)
    .AddResistor(1, 2, 10)
    .AddResistor(2, 3, 5)
    .AddResistor(1, 3, 5, "R1");
var circuit = builder.BuildCircuit();
var model = circuit.GetLargeSignalModel();
var vsouce = (ITwoTerminalLargeSignalDevice)model.FindDevice("VS");
var res = (ResistorDevice)circuit.FindDevice("R1");

// sweep for values from 1 Ohm to 15 Ohm
for (int i = 1; i <= 15; i++)
{
    res.Resistance = i+1; // set resistance

    // calculate
    model.EstablishDcBias();
    var v1 = model.NodeVoltages[1];
    var v2 = model.NodeVoltages[2];
    var v3 = model.NodeVoltages[3];
    var iV = vsouce.Current;

    // print values
    Console.WriteLine($"{i+1}Ohm: {v1}V {v2}V {v3}V {iV}A");
}
```

5.4.3 Changing the Integration Method

Some circuits are sensitive to the choice of numerical integration method used during the simulator. The default integration method is GEAR-2 method, which is a reasonable compromise between accuracy and stability. However, use of GEAR-2 method dampens the oscillations of RLC circuits. Consider the trivial circuit in figure 5.7. Because this circuit lacks any resistance, the voltage at node 1 should oscillate indefinitely with amplitude of 1V.

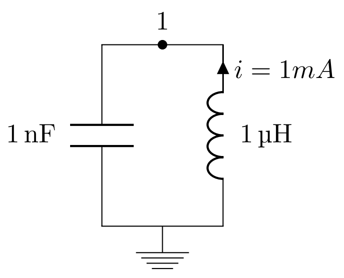


Figure 5.7: Indefinitely oscillating circuit

However, the GEAR-2 method dampens the oscillation. The simulator results with GEAR-2 method are shown in figure 5.8

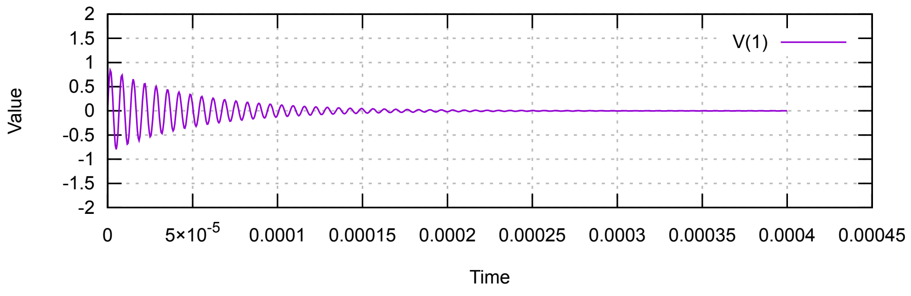


Figure 5.8: Simulation results on oscillating circuit using GEAR-2 method

For comparison, the figure 5.9 shows the same simulation with trapezoidal integration method.

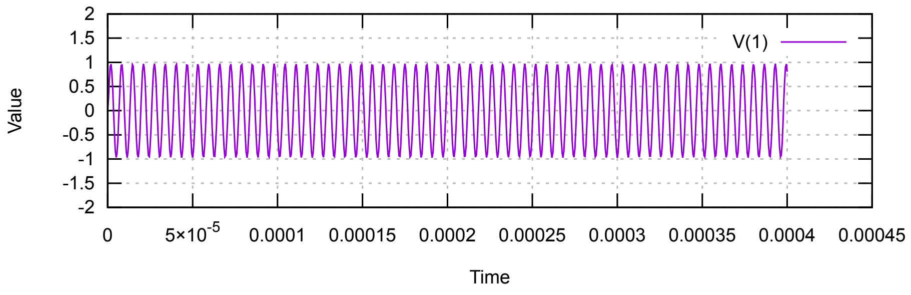


Figure 5.9: Simulation results on oscillating circuit using trapezoidal rule

Even though the trapezoidal method is more precise in terms of smaller local truncation error and does not dampen the oscillations, it is less stable than other methods. For comparison figure 5.10 shows plots of data obtained using the trapezoidal rule on the back-to-back diode circuit we have shown back in figure 5.4. Notice the numerical noise, also known as *trapezoidal ringing*, present in the plot.[1]

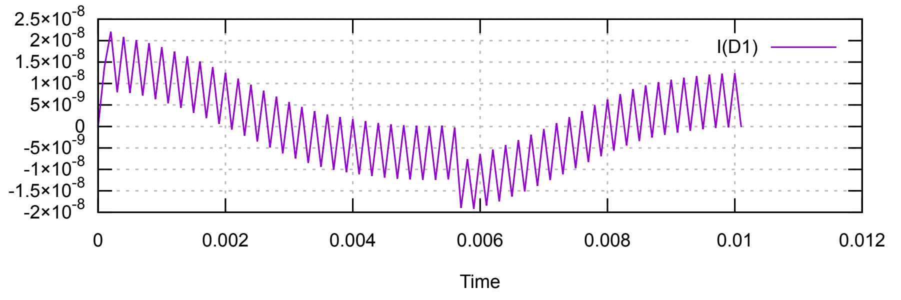


Figure 5.10: Simulation results of back-to-back diode using the trapezoidal rule

The numerical method that is used during the simulation can be changed by replacing the `IntegrationMethodFactory` property on `LargeSignalCircuitModel.CircuitParameters` object. Following code fragment shows how to configure the simulator to use the Trapezoidal Rule integration method.

[1]We have increased the timestep to $100\mu s$, the original $10\mu s$ timestep also produces numerical oscilation, but due to the density of the oscilation the plot would be merged into a single thick line.

```
LargeSignalCircuitModel model = /* ... */;

// requires NextGenSpice.LargeSignal.NumIntegration namespace
model.CircuitParameters.IntegrationMethodFactory =
    new SimpleIntegrationMethodFactory(
        () => new TrapezoidalIntegrationMethod());
```

The supported integration methods are listed in the following table

Integration method	Class name
GEAR method	`GearIntegrationMethod`
Trapezoidal rule	`TrapezoidalIntegrationMethod`
Backward (implicit) Euler	`BackwardEulerIntegrationMethod`
Adams Moulton	`AdamsMoultonIntegrationMethod`

6. Extending the Library

This chapter describes the functionality of the library can be extended by the user. Also, in section 6.3, we provide an example code that implements a simple diode device that will demonstrate the process on a concrete example.

6.1 Adding New Circuit Devices

Adding new circuit devices requires both creating a class that is used in circuit description and class that implements the actual large-signal simulation logic. We will describe both parts in separate subsections.

6.1.1 Adding Device Description

Each device description class must implement the `ICircuitDefinitionDevice` interface. To simplify the implementation of new devices, the library provides basic implementation of this interface in `CircuitDefinitionDevice` abstract class. Furthermore, there exists `TwoTerminalCircuitDevice` abstract class which defines additional member which can be useful for devices with two terminals.

These classes can be then immediately used as arguments to the `CircuitBuilder.AddDevice` method and thereby in the circuit definition. If the device should participate in the circuit topology validation, then the derived classes should override the `GetBranchMetadata` method and return the `CircuitBranchMetadata` instances that describe which terminals are connected by voltage-defined and current-defined branches.

6.1.2 Adding Large-Signal Device Implementation

To use these devices during a circuit analysis, their analysis-specific logic must be implemented and then registered in the respective `IAnalysisModelFactory` instance, so that the `AnalysisModelCreator` knows which implementation to use when creating the `LargeSignalCircuitModel`.

The Device's large-signal implementation needs to implement the `ILargeSignalDevice` interface. This interface defines following members:

- `RegisterAdditionalVariables` – Allows implementation to add additional variables to the equation system via supplied `IEquationSystemAdapter` instance.
- `Initialize` – Lets devices request equation system proxies from `IEquationSystemAdapter`, and perform other necessary initialization, like getting numerical integrators from `ISimulationContex.SimulationParameters.IntegrationMethodFactory`.
- `ApplyModelValues` – Applies devices MNA stamp into the equation system. If the device is nonlinear, the linearized equivalent circuit should be stamped.

- `OnEquationSolution` – Lets devices check for solution convergence (compare current solution with the previous one). The tolerancies can be accessed in `ISimulationContext.SimulationParameters` object, and if the solution did not converge, the `ISimulationContext.ReportNotConverged` method should be called.

- `OnDcBiasEstablished` – Called after the Newton-Raphson iterations have reached a fixed point and the calculation of the current timestep has completed.

The mapping between circuit definition classes and their analysis implementations is done for each analysis model separately by calling the `SetModel<TRep, TMod>` method on the respective `IAnalysisModelFactory` instance. The method accepts a function which creates the implementation class from the circuit definition class. The following code snippet shows how to register `LargeSignalResistor` as the implementation of `Resistor` device for `LargeSignalCircuitModel`.

```
var factory = AnalysisModelCreator.Instance
    .GetFactory<LargeSignalCircuitModel>();

factory.SetModel<Resistor, LargeSignalResistor>(
    resistor => new LargeSignalResistor(resistor));
```

This mapping can be later changed by another call to `SetModel` method. It is also possible to create separate `AnalysisModelCreator` instances and have different mappings in each instance.

6.2 Extending the Parser

An optional step when adding a new device to the simulator is extending the parser to allow importing the new device from spice netlists.

6.2.1 Adding New Device Processors

The parser implemented in NextGen SPICE library delegates parsing of individual device statements to classes implementing the `IDeviceStatementProcessor` interface. Implementations of this interface can be added to the parser by calling the `RegisterDevice` method. We recommend creating new device statement processors by deriving from `DeviceStatementProcessor` abstract class. The derived class then needs to implement only the `Discriminator` char property which specifies the letter identifying the device (The letter should be uppercase), and `DoProcess` method that does the actual processing.

For simple devices with no dependencies, the `DoProcess` method may directly add the device to the `CircuitBuilder` accessible on the `protected` property of the same name in `DeviceStatementProcessor`. In case of devices like diode which depend on other statements (diode statement depends on `.MODEL` statement which defines its model parameters), the statement processing needs to be deferred until later. This is done by adding a `DeferredStatement` on the active

ParsingContext accessible through the Context property of DeviceStatement Processor base class. The DeferredStatement defines the following methods.

- CanApply – Should return true if all dependencies of the statement have been processed and the statement can be processed next.
- Apply – Should add the device into the circuit using the CircuitBuilder on the ParsingContext
- GetErrors – Should return a collection of SpiceParserErrors that describe errors prevent the statement processing. This method is called once it is certain that no statement can be processed.

The DeviceStatementProcessor class also provides property DeviceName that exposes the name of the currently parsed device, and following methods.

- GetNodeIds(int start, int count) – Returns node ids of count nodes specified in the statement starting by token on with index start.
- GetValue(int index) – parses the numeric value out of the token on indexth token.

Both these method do the necessary error handling. The number of errors generated can be checked in the Errors property, and additional errors can be added to the Context.Errors collection. If no error is encountered, the processor should either add the device to the circuit using the CircuitBuilder property, or add a DeferredStatement to Context.DeferredStatements collection. The following code fragment shows how the resistor statements are parsed.

```
protected override void DoProcess()
{
    var nodes = GetNodeIds(1, 2);
    var rvalue = GetValue(3);

    if (Errors == 0)
        CircuitBuilder.AddDevice(nodes, new Resistor(rvalue, DeviceName));
}
```

Because the resistor device does not depend on any other statement, the device can be added to the circuit straight away. In case of diode statements, diode devices cannot be added to circuit until the corresponding .MODEL statement is parsed. Because .MODEL statements are used to set parameters for many device types, there is a ModeledDeviceDeferredStatement<T> class deriving from DeferredStatement. which implements the checking for models. This class is used in the DiodeStatementProcessor as shown in the following code fragment.

```
protected override void DoProcess()
{
    var nodes = GetNodeIds(1, 2);
    // cannot check for model existence yet, defer checking for model later

    if (Errors > 0)    return;

    var name = DeviceName; // captured in lambda
    var modelToken = RawStatement[3];
    Context.DeferredStatements.Add(
        new ModeledDeviceDeferedStatement<DiodeParams>(
            scope: Context.CurrentScope,
            addFunc: (par, cb) => cb.AddDevice(nodes, new Diode(par, name)),
            modelNameToken: modelToken));
}
```

6.2.2 Adding New Model Types

Some devices (like diode and BJT transistor) require a corresponding `.MODEL` statement. The SpiceNetlistParser therefore can be extended to parse new models for new devices. The handling of models for the device is done by returning `IDeviceModelHandler` instances that do the parsing. The library again supplies a abstract class `DeviceModelHandlerBase<TParam>` that implements the basic logic. Derived classed need only specify the mapping to the parameter names. We show an example implementation of the `IDeviceModelHandler` later in section 6.3.2.

6.3 Example: Adding a Diode Device

In previous sections we described all the steps needed to add a new device to the simulator. To provide a concrete example, we will show in this section how to implement a simple diode device. The NextGen SPICE library already contains diode device implementation, represented by classes `Diode` and `LargeSignalDiode`, which is more complex version of the one we will show in this section. The library is designed so that circuit description classes can be reused, and the new diode implementation would require implementing only the device's large-signal logic. However, to demonstrate how completely new devices can be added to the library, we will not reuse the existing diode class. Our diode will be modeled solely by the Shockley diode equation and parameters shown in figure 6.1. The names in capital letters will be later used in diode statements in SPICE netlists.

$$I = I_S \left(e^{\frac{V}{n \cdot V_t}} - 1 \right)$$

Parameter	Name	Description
I_S	IS	Saturation current
n	N	Ideality coefficient
V_t	VT	Thermal voltage

Figure 6.1: Shockley diode equation and its parameters for modeling the diode

To differentiate from diode already implemented in the library, we will use the prefix `Shockley` for the classes implemented in this tutorial.

6.3.1 Creating a Diode Device Definition

First, we have to create a class implementing `ICircuitDescriptionDevice`, which will be used to represent the diode in the circuit description. Since diode has two terminals, we will derive our class from the `TwoTerminalDevice` class, which already implements members needed by the interface. The only thing we have to add are the diode parameters. To keep with the practice of encapsulating the device parameters in separate class for semiconductor devices, we will define classes `ShockleyDiode` and `ShockleyDiodeParams` as follows.

```
public class ShockleyDiodeParams
{
    // specify default parameters

    public double SaturationCurrent { get; set; } = 1e-14;
    public double ThermalVoltage { get; set; } = 25.8563e-3;
    public double IdealityCoefficient { get; set; } = 1;
}

// requires .Core.Devices namespace
public class ShockleyDiode : TwoTerminalCircuitDevice
{
    public ShockleyDiodeParams Param { get; set; }

    public ShockleyDiode(ShockleyDiodeParams param, object tag = null)
        : base(tag)
    {
        Param = param;
    }
}
```

6.3.2 Parsing diode statements

Next we will show how to extend the parser to handle the new device. Suppose our Shockley diode statement should be specified by following syntax.

```
S<name> <anode> <cathode> <model name>
.MODEL <name> SHOCKLEY([IS=<val>] [N=<val>] [VT=<val>])
```

We will start with the `.MODEL` statement. Parsing `ShockleyDiodeParams` from the `.MODEL` statement is done by a class deriving from `DeviceModelHandlerBase<T>` specialized on the `ShockleyDiodeParams` type. The mapping of individual properties is set in the constructor by calling the `Map` method as shown in the following code fragment.

```
// requires NextGenSpice.Parser.Statements.Devices; namespace
private class ShockleyDiodeModelHandler
    : DeviceModelHandlerBase<ShockleyDiodeParams>
{
    public ShockleyDiodeModelHandler()
    {
        Map(p => p.SaturationCurrent, "IS");
        Map(p => p.ThermalVoltage, "VT");
        Map(p => p.IdealityCoefficient, "N");
    }

    public override string Discriminator => "SHOCKLEY";

    protected override ShockleyDiodeParams CreateDefaultModel()
    {
        return new ShockleyDiodeParams();
    }
}
```

Now we will create the actual Shockle diode statement processor by deriving from `DeviceStatementProcessor`. This class will also override the `GetModelStatementHandlers` method and return an instance of the `ShockleyDiodeModelHandler` class which we implemented above.

```
public class ShockleyDiodeStatementProcessor : DeviceStatementProcessor
{
    public override char Discriminator => 'S';

    public ShockleyDiodeStatementProcessor()
    {
        MinArgs = MaxArgs = 3;
    }

    protected override void DoProcess()
    {
        var nodes = GetNodeIndices(1, 2);
        // cannot check for model existence yet,
        // defer checking for model later

        if (Errors == 0) // no errors in node names or device name
        {
            // use local variable to be captured in lambda
            var name = DeviceName;
            var modelToken = RawStatement[3];
            Context.DeferredStatements.Add(
                new ModeledDeviceDeferedStatement<ShockleyDiodeParams>(
                    Context.CurrentScope,
                    (par, cb) =>
                         cb.AddDevice(nodes, new ShockleyDiode(par, name)),
                    modelToken));
        }
    }

    public override IEnumerable<IDeviceModelHandler>
        GetModelStatementHandlers()
    {
        return new[] { new ShockleyDiodeModelHandler() };
    }
}
```

We now can register this class to `SpiceNetlistParser`, which will use them when parsing the netlist files.

```
var parser =  SpiceNetlistParser.WithDefaults();
parser.RegisterDevice(new ShockleyDiodeStatementProcessor());
```

6.3.3 Implementing Large-Signal Diode Logic

Lastly, we will implement the large-signal logic for the shockley diode as the `LargeSignalShockleyDiode`. We will derive our class from `TwoTerminalLargeSignalDevice<ShockleyDiode>` class which provides the basic members needed by the `ILargeSignalDevice` interface.

Our `ShockleyDiode` device has nonlinear I-V characteristic, which needs to be iteratively linearized, as described in section 3.5.1. The large-signal logic should therefore liearize the I-V characteristic at the candidate DC bias point and enter the corresponding coefficients into the equation system. To simplify manipulation with `IEquationSystemCoefficientProxy` objects, we will use the `CurrentStamper` and `ConductanceStamper` classes, which encapsulate writing stamps for current source and resistor devices, which form the linearized diode equivalent model. Also, we will use the `VoltageProxy` wrapper which will read the anode and cathode node voltages and return their difference, which is the voltage across the diode. These classes will be initialized by calling the `Register` method with the `IEquationSystemAdapter` during the initialization phase.

```
// classes encapsulating the work with equation system coefficient proxies
// requires NextGenSpice.LargeSignal.Stamping namespace
private VoltageProxy voltage; // used to get voltage across the diode

// stamping equivalent circuit model
private CurrentStamper currentStamper;
private ConductanceStamper conductanceStamper;

public LargeSignalShockleyDiode(ShockleyDiode definitionDevice)
    : base(definitionDevice)
{
    voltage = new VoltageProxy();
    currentStamper = new CurrentStamper();
    conductanceStamper = new ConductanceStamper();
}

public override void Initialize(IEquationSystemAdapter adapter,
ISimulationContext context)
{
    // get proxies
    voltage.Register(adapter, Anode, Cathode);
    currentStamper.Register(adapter, Anode, Cathode);
    conductanceStamper.Register(adapter, Anode, Cathode);
}
```

The actual logic for writing the equation system coefficients is done in the `ApplyModelValues` method. We will use the `DeviceHelpers` class to calculate the linear equivalents of the diode, and then use the `Stamp` method on the stamper classes to write the coefficients into the equation system.

```
public override void ApplyModelValues(ISimulationContext context)
{
    var Is = DefinitionDevice.Param.SaturationCurrent;
    var Vt = DefinitionDevice.Param.ThermalVoltage;
    var n = DefinitionDevice.Param.IdealityCoefficient;

```

```
        var Vd = voltage.GetValue();
        // calculates current through the diode and it's derivative
        DeviceHelpers.PnJunction(Is, Vd, Vt * n, out var Id, out var Geq);
        Current = Id;

        // stamp the equivalent circuit
        var Ieq = Id - Geq * Vd;
        conductanceStamper.Stamp(Geq);
        currentStamper.Stamp(Ieq);
    }
```

Because the diode is a nonlinear device, the DC bias calculation needs to be iterated until the values are in the specified tolerances. The convergence check for nonlinear devices is done in the `OnEquationSolution` method. Also we will update the `Voltage` property so that the voltage across the diode can be accessed by the outside code once the calculation is completed.

```
    public override void OnEquationSolution(ISimulationContext context)
    {
        var newVoltage = voltage.GetValue();
        var abstol = context.SimulationParameters.AbsoluteTolerance;
        var reltol = context.SimulationParameters.RelativeTolerance;

        // Check if converged
        if (!MathHelper.InTollerance(newVoltage, Voltage, abstol, reltol))
        {
            // request additional DC bias iteration
            context.ReportNotConverged(this);
        }

        // update voltage for reading
        Voltage = newVoltage;
    }
```

To use the `LargeSignalShockleyDiode` as the diode implementation in `LargeSignalCircuitModel`, we need to register it in the `AnalysisModelCreator` instance used to create the circuit model.

```
    // requires NextGenSpice.Core.Representation
    var factory = AnalysisModelCreator.Instance
        .GetFactory<LargeSignalCircuitModel>();
    factory.SetModel<ShockleyDiode, LargeSignalShockleyDiode>(
        e => new LargeSignalShockleyDiode(e));
```

This concludes the diode implementation, the whole example can be found in the `/sources/SandboxRunner/DiodeImplExample.cs` file on the attached CD.

7. User Documentation - Standalone Application

This chapter describes how to use the NextGenSpice standalone application. To run the application, the user needs to have .NET Core Runtime version 2.0 or newer installed on their computer. The latest version of the .NET Core Runtime can be downloaded from the official website[1]. Because the program is command line based, an external tool is needed for the visualization of the transient analysis output. For this purpose, we recommend downloading and installing gnuplot from the official website.[2]

The program's binaries can be found in the `/binaries` folder on the attached CD. Before the library can be used, the contents of the `/binaries` folder need to be copied to an appropriate location (in this text, we will assume that the files are copied to `C:\NextGenSpice\` folder). If everything is set correctly, following command should produced the output shown.

```
C:\NextGenSpice> dotnet NextGenSpice.dll
Usage: dotnet NextGenSpice.dll <input file>
C:\NextGenSpice>
```

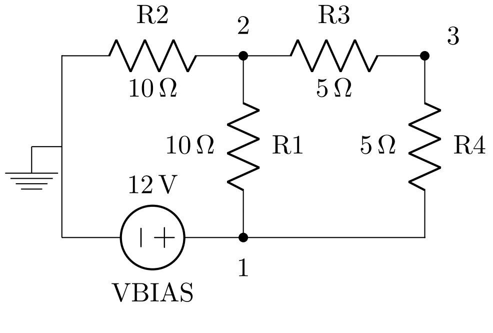
The program does accept exactly one argument: path to a file containing the circuit in SPICE netlist format, as described in chapter 2. The application reads all the statements in the file and reports back any errors encountered. If there are no errors, then individual circuit analyses are executed. First, the respective simulation statement is printed back to standard output, and after that the simulation results are printed. The output format is described in the following sections.

.OP Statement Output Format

In the operating point analysis, the output consists of series of `<variable> = <value>` pairs, each on separate line. If no `.PRINT` statement is specified, then all available data is printed.

To illustrate, consider following netlist file.

```
BRIDGE-T CIRCUIT
*
VBIAS 1 0 12
R1 1 2 10
R2 2 0 10
R3 2 3 5
R4 1 3 5
*
.OP
.END
```



[1]`https://www.microsoft.com/net/download/Windows/run`
[2]`http://www.gnuplot.info/download.html`

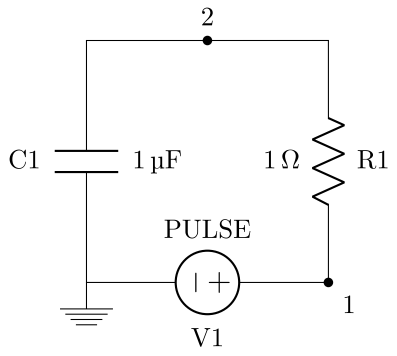
This file is available on the attached CD in `/examples/bridget.txt`. Copy this file to the `C:\NextGenSpice\` folder. Following code snippet shows how the NextGenSpice output for this netlist file.

```
C:\NextGenSpice> dotnet NextGenSpice.dll bridget.txt
.OP
V(1) = 12
V(2) = 8
V(3) = 10

I(VBIAS) = -0.8
V(VBIAS) = 12

I(R1) = 0.4
V(R1) = 4

I(R2) = 0.8
V(R2) = 8

I(R3) = -0.4
V(R3) = -2

I(R4) = 0.4
V(R4) = 2
C:\NextGenSpice>
```

.TRAN Statement Output Format

In the transient analysis, the program prints the output data in individual rows for each simulated timepoint. First, the program prints a header which identifies the data in each column. The first column always specifies the timepoint value, the other columns hold the data specified by the `.PRINT` statement. The individual columns are separated by one space character. As an example, we show a simple circuit which demonstrates the time-domain capacitor behavior.

```
SIMPLE CAPACITOR CIRCUIT

V1 1 0 PULSE(0 15 0 1N 1N 100)
C1 0 2 1U
R1 2 1 1

.TRAN .5U 6U
.PRINT TRAN V(2) I(C1)
.END
```



This netlist can be found in `/examples/capacitor.txt` in the attached CD. Copy this file to the `C:\NextGenSpice\` folder and run the following command. You should see the following output.

```
C:\NextGenSpice> dotnet NextGenSpice.dll capacitor.txt
.TRAN 5E-07 6E-06 0
Time V(2) I(C1)
0 0 0
5E-07 3 -12
1E-06 7.8 -7.2
1.5E-06 10.68 -4.32
2E-06 12.408 -2.592
2.5E-06 13.4448 -1.5552
3E-06 14.06688 -0.933120000000001
3.5E-06 14.440128 -0.559872
4E-06 14.6640768 -0.335923200000001
4.5E-06 14.79844608 -0.201553919999999
5E-06 14.879067648 -0.120932351999998
5.5E-06 14.9274405888 -0.0725594111999996
6E-06 14.95646435328 -0.0435356467200009
C:\NextGenSpice>
```

To simplify visualizing the data output from the `.TRAN` statement using gnuplot, we have included the `plot.ps1` PowerShell script in the `/binaries` folder. This script performs following steps:

1. Run the NextGenSpice program with specified input file.
2. Save the output to a file with same name and `.out` extension.
3. Runs gnuplot and creates an `.svg` file containing the plotted simulation data.
4. Opens the `.svg` file for viewing.

It is important that the input netlist file contains exactly one `.TRAN` statement, and that the path to `gnuplot` executable is set in the PATH environmental variable. To use the script, use the PowerShell command prompt and run the following command. The output plot file should be immediately opened by the web browser.

```
PS C:\NextGenSpice> .\plot.ps1 capacitor.txt
```

The attached CD contains more examples in the `/examples` folder.

8. Results

In this chapter ve evaluate the simulator the NextGen SPICE library's performance based on the used precision type. We will also compare the our simulator with the ngspice and SpiceSharp simulator. We will test the performance on the following circuits:

- `adder` – a four-bit adder circuit.
- `astable` – a simple stable multivibrator.
- `backtoback` – the back-to-back diode circuit from the Mike Robbins' paper about using double-double in circuit simulation [15].
- `cfflop` – a saturating complementary flip flop.
- `choke` – circuit containing two diodes used to choke the voltage source.
- `diffpair` – simple differential pair.
- `ecl` – emmiter coupled logic inverter.
- `rca3040` – circuit simulating a RCA3040 wideband amplifier
- `rtlinv` – cascade RTL inverters.
- `sbdgate` – Shottky-barrier TTL inverter.
- `ua709` – circuit simulating the UA 709 opamp.

The `backtoback` circuit is the circuit with two back-to-back diodes and a $1\mu\Omega$ resistor which was shown back in the introduction chapter in section 1.4. The `adder` circuit was taken from Andrei Vladimirescu's The SPICE Book [1], p. 199. Detailed description of the other circuits, including their schematics, can be found in appendix I of the Nagel's Ph.D. thesis [19]. For convenience, we included a copy of his thesis in the attached CD at `/references/ERL-520.pdf`.

The circuits taken from the Nagel's Ph.D. thesis needed to be slightly modified, because they were intended for SPICE2, which uses different names for some model parameters than SPICE3 (e.g. SPICE2 uses `CCS` for collector-substrate junction capacitance, SPICE3 and NextGen SPICE library uses `CJS`). Moreover, the `.OPTION` statements had to be removed because they are not implemented in our parser. However, no other modifications to the netlists were needed in order to parse them in our library. The actual versions of the netlist files which were used for benchmarks can be found on the attached CD in the `/examples` folder.

8.1 Comparison of Precision Type Performance

We have run a transient analysis on these circuits using the double, double-double and quad-double precision types with native implementation Gaussian elimination algorithm. We used the BenchmarkDotNet library to obtain the results shown in the following table. The simulations are grouped by the simulated circuits, also, relevant information about each circuit is included. Times in the table do not include the time spent parsing the circuit, rows with `NA` mean that the calculation of some timepoint did not converge after 10000 iterations.[1]

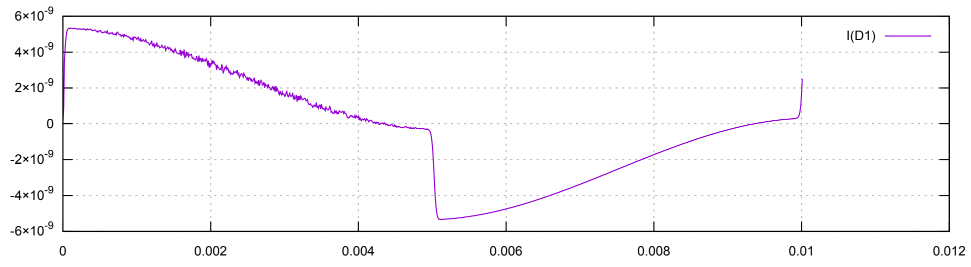
Method	Mean	Error	StdDev	Scaled
`adder`: 451 variables, 446 devices (180 BJT transistors), 50 timepoints				
double	NA	NA	NA	?
double-double	9.147 s	0.2595 s	0.1716 s	?
quad-double	86.230 s	1.3810 s	0.9134 s	?
`astable`: 17 variables, 10 devices, 100 timepoints				
double	14.78 ms	0.1047 ms	0.0979 ms	1.00
double-double	178.91 ms	0.6506 ms	0.5768 ms	12.11
quad-double	1,404.55 ms	17.1084 ms	14.2863 ms	95.04
`backtoback`: 7 variables, 4 devices, 1000 timepoints				
double	2.640 ms	0.0758 ms	0.0843 ms	1.00
double-double	6.696 ms	0.0684 ms	0.0640 ms	2.54
quad-double	46.549 ms	0.3565 ms	0.3335 ms	17.65
`cfflop`: 18 variables, 19 devices, 1000 timepoints				
double	16.493 ms	0.3227 ms	0.6061 ms	1.00
double-double	57.694 ms	0.5024 ms	0.4699 ms	3.50
quad-double	460.006 ms	7.4204 ms	6.9410 ms	27.93
`choke`: 11 variables, 8 devices, 100 timepoints				
double	752.0 μs	14.774 μs	13.820 μs	1.00
double-double	2,223.9 μs	44.123 μs	45.311 μs	2.96
quad-double	17,512.9 μs	248.354 μs	220.159 μs	23.30
`diffpair`: 21 variables, 12 devices, 100 timepoints				
double	NA	NA	NA	?
double-double	8.487 ms	0.0807 ms	0.0630 ms	?
quad-double	65.206 ms	0.1859 ms	0.1553 ms	?
`ecl`: 25 variables, 12 devices, 50 timepoints				
double	NA	NA	NA	?
double-double	2.282 ms	0.0191 ms	0.0179 ms	?
quad-double	18.135 ms	0.4415 ms	0.7254 ms	?
`rca3040`: 44 variables, 26 devices, 400 timepoints				
double	NA	NA	NA	?
double-double	150.7 ms	20.12 ms	13.31 ms	?
quad-double	1,286.7 ms	29.24 ms	19.34 ms	?
`rtlinv`: 15 variables, 8 devices, 100 timepoints				
double	549.5 μs	5.197 μs	4.607 μs	1.00
double-double	1,632.0 μs	31.832 μs	44.625 μs	2.97
quad-double	11,414.4 μs	152.975 μs	135.608 μs	20.77

[1]The simulations mentioned in this chapter were run on system with i5-6300HQ 2.30 GHz CPU using .NET Core 2.0.6 (CoreCLR 4.6.26212.01, CoreFX 4.6.26212.01), 64bit RyuJIT, Release mode.

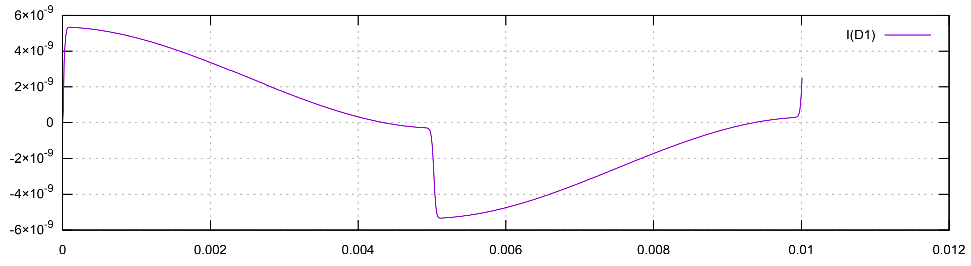
`sbdgate`: 65 variables, 35 devices, 200 timepoints				
double	NA	NA	NA	?
double-double	130.3 ms	25.47 ms	16.85 ms	?
quad-double	1,273.4 ms	27.82 ms	18.40 ms	?
`ua709`: 61 variables, 39 devices, 125 timepoints				
double	NA	NA	NA	?
double-double	123.5 ms	20.96 ms	13.86 ms	?
quad-double	1,244.7 ms	27.23 ms	18.01 ms	?

As seen from the table, using the built-in double type leads to the fastest simulation. However, the simulation of `adder`, `diffpair`, `ecl`, `rca3040`, `sbdgate` and `ua709` did not converge. Because the same circuits converge when enhanced precision types are used, the nonconvergence is probably due to truncation errors during the equation solution, which lead to oscillation around the correct solution, but outside the simulator tolerances. We also ran all these simulations in ngspice simulator successfully without any nonconvergence issues, even though the ngspice uses only standard double precision. We attribute this to the fact that ngspice has been in development for many years and contains many tweaks to ensure convergence.

In terms of simulator output, the output values differ mostly in the 10th significant digit or lower and the data plots for each precision type are visually indistinguishable from each other. The sole exception is the `backtoback` circuit. In the version where only double precision was used, the truncation errors lead to numerical noise discussed in section 1.4 of the introduction chapter. Figure 8.1 shows the plots for the double and double-double precision type.



double



double-double

Figure 8.1: Comparison of the simulation results for `backtoback` circuit for double and double-double precision type

Nevertheless, the noise seems to be much smaller than the one produced when simulating the circuit with LTspice (see figure 5.4 back in the introduction chapter).

Using double-double precision over standard double precision helped solve convergence issues in many circuits and helped eliminate noise at the cost of slower simulation speed. From our observations, using quad-double over double-double does not bring any additional benefits (number of iterations needed to simulate the circuit stays the same, double-double already eliminates possible noise from truncation errors in the `backtoback` circuit) and only slows down the simulation by an order of magnitude. Therefore, we do not recommend using the quad-double type when using the NextGen SPICE library. Instead, we recommend using the double-double precision even though the simulation may be slower than when using only double precision.

In the future versions of the library, we would certainly like to add more convergence aids to ensure convergence even with double precision type. Then the choice of precision type would be based on whether the circuit is expected to be ill-conditioned or not.

8.2 Comparison with Ngspice Simulator

Following table compares the simulation times of our library with the ngspice simulator. The times listed in the table were obtained using the `rusage trantime` command in ngspice interactive mode, which lists the time the simulator spent on the transient analysis in seconds with three decimal places. Because of the low resolution of the `trantime` information, we compared the runs only on the simulations which take several milliseconds. The ngspice simulator was run 5 times and the run times were averaged. The `adder` circuit was simulated for both 50 timesteps (as in previous section) and 6400 timesteps.

Circuit	ngspice	NextGen SPICE (double-double)
`cfflop`	11 ms	57 ms
`rca3040`	8 ms	150 ms
`sbdgate`	6 ms	130 ms
`ua709`	5 ms	124 ms
`adder` (50 ns)	67 ms	9,147 ms
`adder` (6400 ns)	8.3 s	898.1 s

The ngspice simulator is noticeably faster. The ratio of the runtimes of ngspice and that of our library seem roughly proportional to the number of variables in the equation system, in case of the `adder` circuit with 451 variables, the ngspice simulator was more than 100 times faster. We used the performance profiler integrated in Visual Studio 2017 to find out which parts of our simulator are the slowest. It turns out that the simulator spends more than 95% of the time solving the equation system. The equation systems for the larger circuits are very sparse, in case of `adder` circuit, only around 1% of the matrix entries are nonzero. Because our simulator uses full matrix representation, it spends too much time on multiplying and adding the zero entries. On the other hand, ngspice uses sparse matrix representation which performs only the necessary arithmetical operations.

Therefore, in the next versions of the library, we should implement the sparse matrix representation, and perhaps also consider using different method for solving the equation system to speed up the simulation. Possible choice would be using LU factorization which is also used by ngspice, or even iterative methods like Gauss-Seidel.

8.3 Comparison with SpiceSharp

NextGen SPICE is not the only circuit simulator library for .NET in development. There is SpiceSharp [8], whose development started shortly after that of our library. The SpiceSharp is made to resemble the original SPICE3F5, with some modifications to make the source code more appropriate for the .NET platform. Several parts of the source code contain comments with references to the original SPICE3's source and explanation how is it modified.

To compare the user interface of SpiceSharp with that of NextGen SPICE library, consider following code fragment for simulating the simple RLC circuit, which we have simulated in the NextGen SPICE tutorials back in section 5.1.2. We omitted the declarations of outer class and namespace usings for brevity.

```
private static void SpiceSharp()
{
    var circuit = new Circuit(
        new VoltageSource("V1", "1", "0",
            new Pulse(0, 5, 5e-3, 0, 0, 25e-3, double.MaxValue)),
        new Resistor("R1", "1", "2", 50),
        new Inductor("I1", "2", "3", 0.125),
        new Capacitor("C1", "3", "0", 1e-6)
    );

    Transient tran = new Transient("TRAN", 0.2e-3, 55e-3);
    Console.WriteLine("Time V(1) V(3) I(VC)");

    RealPropertyExport i_v1 = new RealPropertyExport(tran, "V1", "i");
    Console.WriteLine("Time V(1) V(3)");
    tran.OnExportSimulationData += (sender, args) =>
    {
        var time = args.Time;
        var v1 = args.GetVoltage("1");
        var v3 = args.GetVoltage("3");
        var i = i_v1.Value;

        Console.WriteLine($"{time} {v1} {v3} {i}");
    };

    tran.Run(circuit);
}
```

The SpiceSharp user interface is very similar to the SPICE netlist syntax. This means that all devices and nodes are identified by a string. In the NextGen SPICE library, the individual devices can be (optionally) tagged by an arbitrary C# object.

Another difference is how a circuit is constructed, in case of simple devices, the identifiers of connected nodes are passed to the constructor, and the constructed devices can be passed to the Circuit class constructor. However, the connections for transistors can be only set by calling the Connect instance method, which we found counterintuitive.

The individual simulations are done by calling a Run method on a dedicated simulation object with the simulated circuit as an argument (see lines 11 and 26 of the source above). Getting the simulation data is a bit tricky. The easiest way to get the node voltages in the individual timepoints is registering a handler on the OnExportSimulationData event (lines 14 to 20). To get e.g. current flowing through the voltage source, one has to use a RealPropertyExport class (line 14) and specify the name of the device and a string name of the property to be extracted. The value of this voltage source current can be then accessed via the Value property during the OnExportSimulationData callback (line 21).

The SpiceSharp's interface makes heavy use of string values, in the example above we show that getting a current though a device requires knowledge of how the appropriate string identifier. What's worse, the strings are also used to set individual parameters for semiconductor devices. following code fragment is taken from the SpiceSharpTest/Models/Semiconductors/DIO/DiodeTests.cs file from the SpiceSharp's Github repository (version from 4th May 2018).

```
 // Build circuit
Circuit ckt = new Circuit();
ckt.Objects.Add(
    CreateDiode("D1", "OUT", "0", "1N914",
        "Is=2.52e-9 Rs=0.568 N=1.752 Cjo=4e-12 M=0.4 tt=20e-9"),
    new VoltageSource("V1", "OUT", "0", 0.0)
);
```

The CreateDiode method is a helper method which parses the parameter string and sets individual parameters by calling SetParameter method on Diode Model class, which internally uses reflection to set appropriate property. The downside of this is that the parameter names cannot be hinted by automatic code completion feature of the IDE (like Intellisense in Visual Studio). These actual object on which the parameters are stored can be accessed, but it takes several casts and is by no means intuitive. Because of this, we consider the interface of our NextGen SPICE library superior to that of SpiceSharp.

Although it's interface is user unfriendly, SpiceSharp implements more circuit analyses and more circuit devices. Also, the implemented models for semiconductor devices are more detailed than those implemented in NextGen SPICE library. We would like to address this in the next version of our library and expand the set of implemented devices and add new analysis types.

We tried to compare the performance of SpiceSharp and the NextGen SPICE library. However, we have had difficulties parsing the netlist files containing the benchmark circuits in the SpiceSharp parser. Some netlist needed to be altered (e.g. SIN source changed to SINE, adding an argument with default value which is not strictly needed by SPICE3, or converting parts of the netlist to lowercase). The SpiceSharp simulator then reported a singular matrix when simulating the adder and sbdgate circuit, and in the circuits which were actually simulated, the SpiceSharp simulator used bigger timestep than we specified. This lead to very shorter simulation times and lower-resolution plots. This makes it difficult to compare the simulators performances. Following table lists measured times and number of timesteps computed.

Method	Timepoints	Mean	Error	StdDev	Scaled
`backtoback`					
NextGen SPICE	1000	846.4 μs	10.319 μs	9.148 μs	1.00
SpiceSharp	5550	69,987.3 μs	565.747 μs	472.424 μs	82.70
`cfflop`					
NextGen SPICE	1000	59,352.3 μs	1,151.949 μs	1,232.573 μs	1.00
SpiceSharp	81	2,051.9 μs	29.520 μs	27.613 μs	0.03
`choke`					
NextGen SPICE	100	2,236.1 μs	42.844 μs	40.076 μs	1.00
SpiceSharp	73	557.4 μs	7.231 μs	6.764 μs	0.25
`diffpair`					
NextGen SPICE	100	8,576.7 μs	165.504 μs	215.202 μs	1.00
SpiceSharp	59	1,032.8 μs	10.943 μs	10.236 μs	0.12
`rtlinv`					
NextGen SPICE	100	1,569.9 μs	13.843 μs	12.271 μs	1.00
SpiceSharp	80	988.7 μs	10.505 μs	9.826 μs	0.63

When comparing the runtimes per individual timepoint, the NextGen SPICE is around four times slower, which can be explained by the usage of double-double precision in NextGen SPICE and sparse matrix representation in SpiceSharp. However, when simulating te `backtoback` circuit, the SpiceSharp needed far more timepoint computations. As seen from in figure 8.2, the SpiceSharp does not correctly handle circuits with very small resistors.

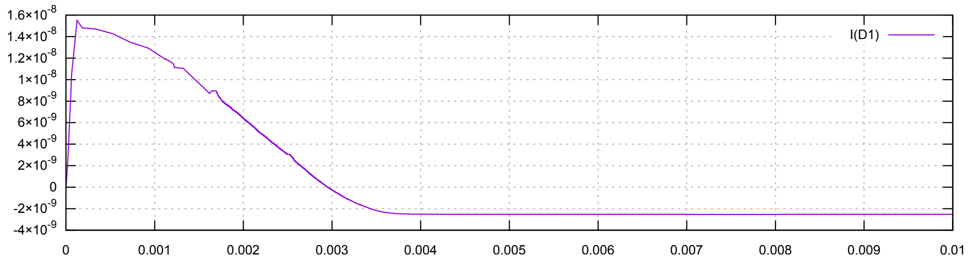
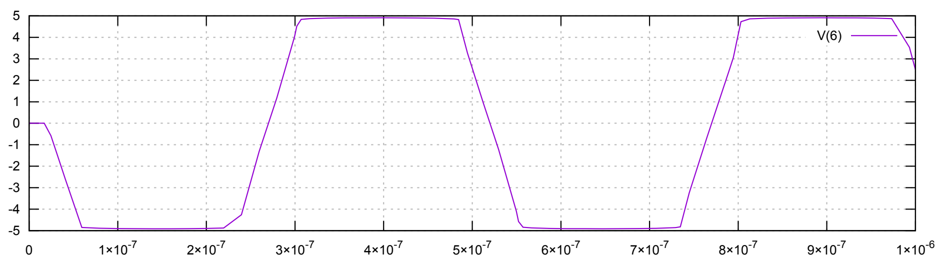
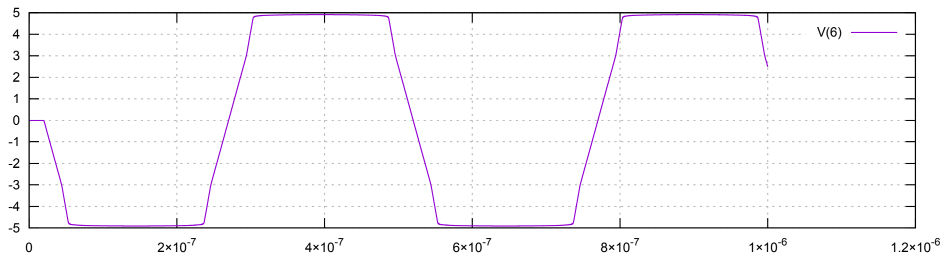


Figure 8.2: Plot of the `backtoback` circuit from SpiceSharp

The plots of other circuits were similar to the ones produced by the NextGen SPICE, although the greater timestep resulted in visible straight regions in the plot. For example, figure 8.3 shows plot of voltage in `cfflop` circuit on the `6` node. At $2{\cdot}10^{-7}$ mark, the SpiceSharp output has visible straight edge. Also, the output our NextGen SPICE shows clearly the slight S shape of the slopes when voltage changes. This shape is not clearly visible on the SpiceSharp output.



SpiceSharp



NextGen SPICE

Figure 8.3: Comparison of simulation results on the `cfflop` circuit

8.4 Summary

Considering the measurements listed in this chapter, the representation of equation system and method for solving it is crucial part of the circuit simulator. Our choice of the simplest implementation possible – full matrix and Gaussian elimination – caused our simulator to be orders of magnitude slower on large circuits than other circuit simulators. However, thanks to the abstraction we used during the implementation, more appropriate methods can be implemented in future versions of the library.

In terms of the simulator output, NextGen SPICE produces visually same plots as other circuit simulators, and because of the double-double precision type used, it does not suffer from the noise caused by truncation error during equation system solution.

Conclusion

To conclude our thesis, we will revisit the goals we set in the introduction chapter in section 1.5

1. *Implement SPICE-like simulation library*

   (a) *Target .NET Standard for maximum portability* – Our library requires .NET Standard 2.0, which makes it available on all major platforms running one of the newer version of .NET runtime.

   (b) *Support performing time-domain simulation of the circuit, and allow changing parameters of circuit devices between individual timesteps.* – We designed our simulator in such a way that users of our library decide when next circuit state is computed and how big the timestep should be. Between the individual timesteps, users can modify parameters of the devices in the simulated circuit, and have the changes affect the next calculated circuit state.

   (c) *Support following set of devices*

      i. *Ideal resistor*

      ii. *Ideal voltage source*

      iii. *Ideal current source*

      iv. *Ideal inductor*

      v. *Ideal capacitor*

      vi. *SPICE diode*

      vii. *SPICE BJT transistor*

      We successfully implemented all circuit devices listed above. In addition, we also implemented linear controlled sources: voltage controlled voltage source, voltage controlled current source, current controlled voltage source and current controlled current source.

   (d) *Allow new types of circuit analyses and circuit devices to be added to the simulator without modifying the library's source code.* – We have written a guide on how to add new devices in library's user documentation in chapter 6 and provided an example of adding new device in section 6.3.

   (e) *Implement SPICE netlist parser to allow importing circuits and macromodels from standard SPICE netlist files.* – Our parser supports sufficient subset of the SPICE3 netlist syntax to allow importing circuits and subcircuits (macromodels) containing devices implemented in our simulator. We have tested our parser on existing SPICE netlists with success. However, because the parser implementation present in the library implements only the data statements (devices, subcircuits and device models), it is necessary to remove any control statements from the netlist file before parsing them in NextGen SPICE.

   (f) *Allow users of the library to choose between double, double-double, and quad-double precision types and compare the library's performance with* *respect to speed and accuracy for each listed precision type.* – Users can compile our library themselves and choose the precision type to be used by defining a certain conditional compilation symbol. We compared the simulator performance for each precision type and found out that the double-double type currently provides the best combination of convergence and simulation speed for our library.

2. *Use the simulation library to implement SPICE-like console application for .NET Core, which would accept implemented subset of SPICE netlist syntax.* – Our `NextGenSpice` application targets .NET Core 2.0 and provides the desired functionality by extending the library's parser to handle `.TRAN` `.OP` and `.PRINT` statements. We then used the library's functionality to run the simulations and print the requested data to standard output.

Future Work

The NextGen SPICE library offers only a narrow subset of the SPICE-like simulators used today. Following list contains features which we consider most beneficial for the next version of the library.

- *Sparse matrix representation* – As discussed in the 8.1, the Gauss-Jordan elimination and full matrix representation proved to be a performance bottleneck when simulating larger circuits. Using sparse matrix methods which are used by the standard SPICE implementations would significantly speed up the simulation.

- *Dynamic timestep* – Current implementation of the transient analysis algorithm relies on the user to choose a fixed timestep. As discussed in the analysis 3.5.5, dynamic timestep mechanism can speedup simulation in regions where the capacitor charges and inductor fluxes do not change quickly.

- *Implementing `.INCLUDE` statement* – Currently all used models and subcircuits need to be defined in the netlist file. SPICE3's `.INCLUDE` statement works similarly to the `#include` directive in C or C++ languages: the contents of the included file are treated as if they were copied and pasted in place of the `.INCLUDE` statement. This allows better reuse of the subcircuits and defined models.

- *Interactive console application* – Current `NextGenSpice` console application offers limited interaction with the user. Also, when the user wants to run the same simulation with different parameters, the netlist file must be edited and the application run again. SPICE3 introduced an interactive mode, in which the program reads only the circuit description from the netlist file. The simulation statements and other control statements are then supplied on the standard input by the user.

- *More devices and analysis types* – Last but not least, the NextGen SPICE library as implemented in this thesis offers only a fraction of circuit analysis types and circuit devices. Examples of devices which are completely missing are switches (voltage and current controlled), other types of transistors

(JFET, MOSFET), transmission lines, coupled inductors and semiconductor versions of resistor and capacitor devices. From the analysis types, the NextGenSpice library is missing e.g. the AC frequency sweep analysis, which requires small-signal models of the simulated devices.

Bibliography

[1] Andrei Vladimirescu. *The Spice book*. J. Wiley, 1994.

[2] Richard P. Anderson. Creating a SPICE subcircuit. `http://www.5spice.com/html/Subckts.html`, 2003.

[3] Ron M. Kielkowski. *Inside SPICE*. McGraw-Hill, 1998.

[4] Inc. Analog Devices. LTspice - official website. `http://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html`.

[5] Žiga Rojec, Árpád Bűrmen, and Iztok Fajfar. An evolution-driven analog circuit topology synthesis. In *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pages 1–6. IEEE, 2016.

[6] Ngspice circuit simulator. `http://ngspice.sourceforge.net/`.

[7] Paolo Nenzi Holger Vogt, Marcel Hendrix. Ngspice users manual version 27plus. `http://ngspice.sourceforge.net/docs/ngspice-manual.pdf`, January 2018.

[8] Sven Boulanger. SpiceSharp. `https://github.com/SpiceSharp/SpiceSharp`, 2018.

[9] Fabrice Salvaire. PySpice. `https://github.com/FabriceSalvaire/PySpice`, 2018.

[10] Prof. Dr. Árpád Bűrmen. PyOPUS library - circuit simulation and optimization in python. `http://fides.fe.uni-lj.si/~arpadb/software-pyopus.html`, 2017.

[11] HSPICE. `https://www.synopsys.com/verification/ams-verification/circuit-simulation/hspice.html`.

[12] Tim Moltner. JSpice. `https://github.com/knowm/jspice`, 2018.

[13] Official SPICE3 webpage. `https://embedded.eecs.berkeley.edu/pubs/downloads/spice/spice3f5.tar.gz`, 1993.

[14] T. Quarles. Adding devices to SPICE3. Technical Report UCB/ERL M89/45, EECS Department, University of California, Berkeley, 1989.

[15] Mike Robbins. Extended-precision simulation cures SPICE convergence problems. `https://www.edn.com/design/analog/4418707/1/Extended-precision-simulation-cures-SPICE-convergence-problems`, July 2013.

[16] David H. Bailey Yozo Hida, Xiaoye S. Li. Library for double-double and quad-double arithmetic. `http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.20.tar.gz`, May 2008.

[17] Yozo Hida, Xiaoye S Li, and David H Bailey. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory*, 2008.

[18] Stefan Jahn, Michael Margraf, Vincent Habchi, and Raimund Jacob. Qucs technical papers. *Qucs-Project*, 2003.

[19] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[20] MEF framework. `https://docs.microsoft.com/cs-cz/dotnet/framework/mef/`.

[21] ANTLR. `http://www.antlr.org/`.

[22] BenchmarkDotNet. `https://github.com/dotnet/BenchmarkDotNet`.

[23] Giuseppe Massabrio. *Semiconductor Device Modeling With Spice*. Mc Graw Hill India, 2010.

[24] gnuplot. `http://www.gnuplot.info/`.

Attachments

Contents of the attached CD

- `/sources` – folder containing the `NextGenSpice` solution.
- `/examples` – folder containing sample input files for the `NextGenSpice` standalone program.
- `/binaries` – folder containing the binary files for the NextGen SPICE library and standalone console application, and `plot.ps1` Powershell script for running and automatically plotting the output data.
- `/references` – folder containing copy of the documents which were the source for circuit simulation theory for this thesis.

  `/qucs.pdf` – the QUCS Technical Papers [18]

  `/ERL-520.pdf` – Laurence W. Nagel's PhD thesis on SPICE2 [19]
- `/documentation` – folder containing the PDF version of the API reference for the NextGen SPICE simulator library.
- `/tex` – folder containing the LaTeX source for this thesis

  `/en` – folder with the `.tex` files

  `/img` – folder with images used in this thesis

  `/LICENSE.TXT` – file containing licensing information
- `/thesis.pdf` – file containing this thesis.
- `/README.txt` – file describing the contents of the CD.