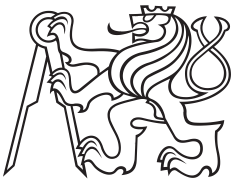
Ph.D. Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
13137 Department of Radioelectronics

Algorithms for Analysis of Nonlinear High-Frequency Circuits

Ing. David Černý

Supervisor: Doc. Ing. Josef Dobeš. CSc.
Field of study: P2612 Electrical Engineering and Information Technology
Subfield: Radioelectronics
August 2016

Acknowledgements

I would like to thank my supervisor doc. Ing. Josef Dobeš, CSc. for his active, methodical and technical support in my study. I very appreciate his help and his very useful advice and recommendation.

I would like to thank my parents Jan and Galina Černá for their infinite love and support.

Many thanks belong to Lucia Bugajová for her patience, motivation, and understanding during my study.

Declaration

I declare that my doctoral dissertation thesis was prepared personally and bibliography used duly cited. This thesis and the results presented were created without any violation of copyright of third parties.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 30. 8. 2016

.....

Abstract

The most efficient simulation solvers use composite procedures that adaptively rearrange computation algorithms to maximize simulation performance. Fast and stable processing optimized for given simulation problem is essential for any modern simulator. It is characteristic for electronic circuit analysis that complexity of simulation is affected by circuit size and used device models. Implementation of electronic device models in program SPICE uses traditional implementation allowing fast computation but further modification of model can be questionable.

The first fundamental thesis aim is scalability of the simulation based on the adaptive internal solver composing different algorithms according to properties of simulation problem to maximize simulation performance. In a case of the small circuit as faster solution prove simple, straightforward methods that utilize arithmetic operations without unnecessary condition jumping and memory rearrangements that can not be effectively optimized by a compiler. The limit of small size simulation problems is related to computation machine capabilities. The present day PC sets this limit to fifty independent voltage nodes where inefficiency of calculation procedure does not play any role in overall processor performance. The scalable solver must also be able to handle correctly simulation of large-scale circuits that requires entirely different approach apart to standard size circuits. The unique properties of simulation of the electronic circuits that played until this time only the minor role suddenly gain on significance for circuits with several thousand voltage nodes. In those particular cases, iterative algorithms based on Krylov subspace methods provide better results from the aspect of performance than standard direct methods. This thesis also proposes unique techniques of indexation of the large-scale sparse matrix system. The primary purpose is to reduce memory requirements for storing sparse matrices during simulation computation.

The second fundamental thesis aim is automatic adaptivity of device models definition respecting current simulation state and settings. This principle is denoted as Functional Chaining mechanism that is based on the principle of the automatic self-modifying procedure utilizing state-of-the-art functional computation layer during the simulation process. It can significantly improve mapping performance of circuit variables to device models; it also allows autonomous redefinition of simulation algorithms during analysis with an intention to reduce computation time. The core idea is based on utilization of programming principles related to functional programming languages. It is also presents possibilites of reimplementation to the modern object-oriented languages.

The third fundamental thesis aim focuses on simulation accuracy and reliability. Arbitrary precision variable types can directly lead to increased simulation accuracy but on the other hand; they can significantly decrease simulation performance. In last chapters, there are several algorithms provided with the claim to provide better simulation accuracy and suppress computation errors of floating point data types.

Keywords: Common LISP, SPICE, computer simulation, CLASP, electronic circuits, device models, CAD, DC analysis, transient analysis, nonlinear equations, numerical integration, LU factorization, sparce matrices, Newton-Raphson algorithm, modified nodal analysis Biconjugate Gradient Method, Krylov subspace methods, functional computation

Abstrakt

Nejefektivněji pracující výpočetní jednotky používají kompozitní procedury, které dokáží přizpůsobovat své vnitřní algoritmy takovým způsobem, aby maximalizovaly výpočetní výkon simulace. Rychlé a stabilní zpracování optimalizované pro daný simulační problém, je důležitou součástí moderních simulátorů. Pro analýzu elektronických obvodů je charakteristické, že složitost simulace je ovlivněna velikostí obvodu a použitých modelů součástek. Implementace modelů elektronických součástek v programu SPICE používá tradiční implementaci zajišťující rychlé výpočty, ale další modifikace těchto modelů ze strany uživatele je obtížná.

Prvním základním cílem této práce je škálovatelnost simulace založená na adaptivní proceduře kombinující algoritmy dle vlastností simulačního problému za účelem maximalizace výpočetního výkonu. V případě malého obvodů se jako nejlepší řešení ukazuje použití jednoduchých přímočarých metod, které používají aritmetické operace bez zbytečného kopírování paměti a podmíněních skoků, které nedokáží být efektivně zpracovány překladačem. Limit, do kterého lze daný obvod považovat za malý je samozřejmě závislý na vlastnostech výpočetního stroje. Dnešní osobní počítače nastavují tento limit na zhruba padesát nezávislých napěťových uzlů, u kterých neefektivita výpočetní procedury nehraje žádnou roli na celkový výkon procesoru. Škálování musí správně zvládat výpočty extrémně velkých obvodů. Speciální vlastnosti simulace elektronických obvodů, které do této doby hrály pouze minoritní roli, získají na významu v případě obvodů nad několik tisíc napěťových uzlů. V těchto speciálních případech, iterační algoritmy založené na metodách Krylova podprostoru poskytují daleko lepší výsledky z pohledu výpočetního výkonu než standardní přímé metody. Tato práce také předkládá speciální techniku indexace velkých řídkých maticových systémů. Jejím hlavním cílem je redukovat paměťové nároky potřebné na uložení velkých řídkých matic během simulačního výpočtu.

Druhým základním cílem této práce je automatická adaptabilita definice modelů součástek s ohledem na aktuální stav simulace a jejího nastavení. Teto princip je v práci označován jako Funkcionální řetězící mechanizmus, jenž je založen na principu samomodifikovatelné procedury vyžívající během simulačního procesu funkcionální výpočetní vrstvu. Umožňuje výrazně zlepšit výkon mapování obvodových veličin k modelům součástek, také dokáže během analýzy samostatně redefinovat simulační proces s cílem redukovat výpočetní čas . Základní ideou je použití principů, které jsou vlastní funkcionálním programovacím jazykům. V práci je též ukázáno, jak tyto procesy implementovat v moderních objektově orientovaných programovacích jazycích.

Třetí základní cíl této práce je zaměřen na stabilitu a věrohodnost simulace. Proměnné typy s libovolnou řádovou přesností mohou sice vést ke zlepšení přesnosti výsledků, ale na druhou stranu mohou též výrazně snížit výkon simulace. V poslední kapitole je představeno několik algoritmů s cílem poskytnout lepší přesnost výpočtu a potlačit výpočetní chyby způsobené datovými typy s plovoucí řádovou čárkou.

Klíčová slova: Common LISP, SPICE, počítačová simulace, CLASP, elektornické obvody, modelování součástek, CAD, nelineární obvody, numerická integrace, LU faktorizace, řídké matice, Newton-Raphson iterační algoritmus, modifikovaná metoda uzlových napětí, Biconjugate gradientní metoda, metody Krylova podprostoru, funkcionální výpočty

Překlad titulu: Algoritmy pro analýzu nelineárních vysokofrekvenčních obvodů

Contents

Figures

Tables

Chapter 1

Introduction

Nowadays, the simulation process is an essential part of a computer-aided design (CAD). Without reliable modeling, visualization, and computing, it's hard for an engineer to deliver a professional product within given time and budget. New possibilities in communication and media enabled self-enhancing software development processes based on collective work of communities. The modern simulation tools can be considered as multi-functional systems offering a broad range of functionality through defined interface. They play a major role in a manufacturing process with emphasis on collaboration and reusability of the previous work. Despite intensive development, the simulation of electronic circuits behavior still belongs to the most challenging discipline in CAD process.

There is no doubt that program SPICE (Simulation Program with Integrated Circuit Emphasis) has become one world standard in computer simulation of electrical circuits. The first version of the program was formed between the years 1967 and 1971 in the EECS Department of University of California at Berkeley. It should be noted that early years of SPICE development were dedicated to the investigation of the most accurate and efficient numerical methods for circuit representation, input language, nonlinear equation solution, integration algorithms, sparse-matrix solutions, and nonlinear semiconductor device modeling.

The next generation of the program, SPICE2, was completed in 1975. It came with a new circuit representation, known as modified nodal analysis (MNA) [HRB75a], better memory management, time-step control mechanism and new reliably stable multiple-order integration method. Originally, the core simulation algorithms were written in Fortran programming language and lately converted to C language [Ped84]. Not only advanced capabilities of the program SPICE and but also its free redistribution under Berkeley open-source license ensured that the computation core could be found in many professional CAD programs. For instance

- PSpice (Cadence),
- Affirma Spectre (Cadence),
- HSPICE (Synopsys),
- ELDO (Mentor Graphics),
- SmartSpice (SILVACO),
- NgSpice (ngspice.sourceforge.net)

An enormous impact of a significance of SPICE simulator can be easily seen on a diversity of simulation programs for electronic circuit into SPICE-like simulators (that use for simulation same principles as SPICE) and others (based on a different simulation approach). Despite its age, SPICE algorithms still provide extremely precise memory management and very efficient machine code algorithms.

1.1 Current Situation of the Studied Problem (State-of-the-Art)

Recent development in computer simulation of electronic circuits can be divided into several categories:

- **Device modeling** that primarily focusses on the improvement of device models or implementation of new ones. It also includes disciplines as the definition of physical behavior by approximating mathematical equations and their practical software application in simulation programs.
- **Simulation algorithms** that is dedicated to concrete simulation type or simulation problem where original simulation algorithms are ineffective or could not be used due to particular circuit setup.

Progress in electronics must necessarily be consistent with a development of device models. Concerning a role of device models as basic building blocks of any simulation presumably, there will be constant development on this field. The two main specializations are directed on:

- **Model enhacement** that is focused on improving capabilities of implemented device models in simulator core. The leading parameters are accuracy, reliability, simulation limits and model simulation options as for instance attempt [HC14] to implementation of delay-time-based non-quasi-static bipolar transistor models in circuit simulators,
- **Experimental devices** that introduces to the simulator core entirely new state of the art devices such as memristors [BBB], voltage excitation fluxgate sensor [YLF+13], solar cells [SSB14] or experimental devices emulating the electrochemical behavior of molecular based devices [KSG15].

It should be noted that research in the model development can be also branched out from implementation perspective on:

- **Macro models** that implements new model or model behavior by usage of the standard already implemented device models interconnected to a bigger system or subcircuit.
- **Native models** that are natively programmed into program software and therefore better optimized for professional usage.

The second category focuses on simulation capabilities, computation performance, and precision of entire simulation process.

- **Simulation capabilities** dedicate to concrete simulation type or simulation problem where standard simulation core could not be used. It specifically can occur in situations when given device models or simulation problem is not applicable to SPICE computation procedures. The most often it is a simulation of various DC-DC circuits, but also many other articles have proposed methods for enhancing core to simulate e.g. magnetic features and additional circuit side-effects. From recent works focused on strengthening simulation core it could be mentioned [FOP14], [WWTK14] or interesting work [ABB10] that proposes usage of nonsmooth dynamical systems (NSDS) approach with Moreau's time-stepping scheme to improve simulation stability.

- **Simulation accuracy** that is an essential factor of the reliability of any simulation program. The primary importance can be summarized in several aspects. The first one points to the band of the unknown variables that are simulated. They usually represent the real physical behavior of circuit devices, and their values can differ in magnitude by several orders. The difference can be in the extreme cases higher than a range of used floating point data type. To be able to simulate those extreme cases requires increasing precision of given data types. The second aspect of the accuracy problems is related to the solution of linear systems. Although standard circuit state (i.e. OP) can be well established, during transient analysis circuit matrix can change its properties. It makes a computation of specific time-points very problematic, especially in the case when matrix setup rapidly increases condition number. In that situation, some time-points may be affected by precision inaccuracies more than others. The real issue arises when these particular results are passed digitally for further computations. Then lower precision of the results become more important and can affect all computation chain.

- **Simulation performance** and optimization of computation efficiency allows simulating more complex problems in shorter time span. Every year various works are published suggesting modifications of SPICE simulation algorithms with the objective to improve simulation performance. They propose hardware specific improvements but also changes in simulation procedures such as matrix reordering techniques, LU factorization, and numerical integration.

From the recent publications in this area, can be recommended article [NZ15], that proposes the parallel sparse matrix solver runnning on field-programmable gate array (FPGS). The published method clearly outperforms standard direct solvers. In the article is also presented a comparison of the most powerful matrix solvers, such as UMFPACK, KLU[DPN10], and Kundert sparse matrix packages[KSV86]. Another interesting attempt to reimplement SPICE simulator core on a parallelized single field-programmable gate array (FPGA) is presented in the article [KD12]. Their technique allows not only to accelerate the evaluation of analysis but to reduce the energy needed for computation in comparison to conventional processors. With simulation performance enhacement deals also article [CRWY15] that proposes using of single GPU-accelerated units for LU factorization. The proposed LU factorization approach shows very impressive speed up on NVIDIA graphic card in comparison to high-performance packages for solving large sparse linear systems such as PARADISO [KLS13].

Previously mentioned articles propose improvements devoted to the usage of specialized HW. There could be found articles that suggest simulation improvements from the point of view of simulation algorithms. In the paper [KA01], a replacement of the standard direct method with an iterative Krylov-subspace solver with preconditioner was proposed. As it will be revealed later, a solver of sparse matrix systems based on the iterative method can compete with a direct solver only in exceptional situations (large, positive-definite or banded systems). Hence, it is questionable whether full replacement of the standard direct method is realistic. From the result presented in the article, it is not clear which of the proposed Krylov methods is better for simulation of electronic circuits. Instead, the article provides an unfair comparison between GMRES, QMR, BICGStab methods (that are for nonsymmetric problems) and CGS and BiCG methods that are primarily applicable to symmetric, positive-definite problems [Saa03]. The article also does not describe what kind of direct method is meant by "traditional Gaussian elimination" implemented in the older version of SPICE and why it was slower than iterative methods.

A better view is given in the article [KA00] where methods such as GMRES, CGS, and BICGStab are compared to a direct method and tested on several different circuits.

Unfortunately, the paper does not present any closer information about tested circuits (only names), and the reader can only guess their sizes and properties. Also, the article does not state whether the algorithms use preconditioning or not. It may be noted that GMRES works well for nonsymmetric problems, and it is also the most popular method in general. From the performance perspective, with enough memory and good preconditioner it is a very fast and stable method. However, when it comes to real implementation, it shows that the high memory requirements and complexity of computation together with the impossibility to directly access the solution and residual vector at any iterate disadvantages the practical use of this method as a successor to direct sparse solvers implemented in SPICE simulators.

GMRES method accompanied with preconditioner that comes directly from the previously factorized L and U matrices is proposed in the article [LS05]. There are three methods of computation of preconditioner suggested for GMRES iterative solver. Although it is achieved at the cost of reduced precision (10^{-8}), we must admit that the given algorithm (regardless of used preconditioner) accelerates the computation of transient analysis compared to direct method. It can be seen problematic that the algorithm achieves best results only in the case of small changes in variables between transient steps (and low condition number of circuit matrix) ensuring a very low number of iteration cycles of NR algorithm.

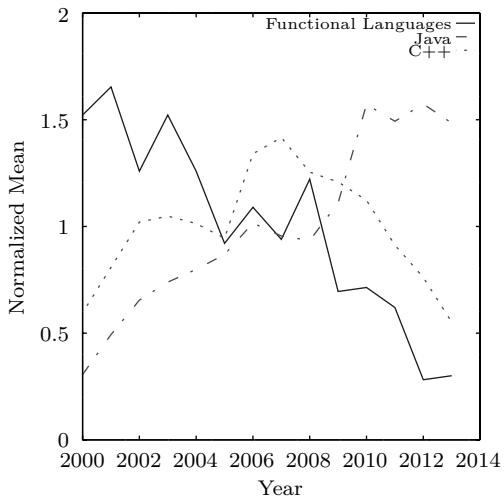


Figure 1.1: Comparison of sourceforge.org project repositories.

Simulation core algorithms can be divided by the computation approach on numerical and symbolic. Where SPICE simulator is a typical example of a numerical simulator, symbolic simulators use a different computation approach. It is based on a symbolic solution of given equations or simulation state. The certain advantage of the symbolic solution is its ability to reduce the complexity of the problem before it is enumerated. It can significantly reduce computation time and improve accuracy. This ability is referred in the thesis as the dynamical readjustment and is inherent in programming languages allowing functional definition.

For many developers, functional languages may seem to be still too exotic for a professional software development not to mention their implementation into the simulator. To verify it a small review on a use of functional languages in last decade is presented at Fig. 1.1. As a source, it was used a well known online software repository sourceforge.org. It had been queried ≈ 250 thousand repositories uploaded to Sourceforge between years 1999 and 2013.

The comparison was made by picking standard object-oriented language C++, Java, and several more or less pure functional languages as (Scala, ML, Lisp, Scheme, Haskel, Erlang).

Search query resulted in total ≈ 29 thousand programs written partly or whole in C++, to ≈ 220 thousand in Java and only ≈ 600 in functional languages. It is evident that the number of new projects written purely in functional languages is decreasing. It may be caused by many factors from a small support, slow performance to old fashion semantics [Wad98].

Nevertheless, the most popular languages as Java (since version 8, 2014), C# (since .NET 1.0, 2002) and C++ (since C++11, 2011) included functional programming to its core and semantics. Moreover many modern programming syntaxes directly encourage developers to use unnamed function and functional definition. Particularly in a situation when problem touches parallelism, multi-task/thread management, and asynchronous operations, the functional definition provides better options and rapidly simplify code syntax.

Once you deal with the functional paradigm, you start to realize its scale invariance. Regardless how deep you have already got, there is another possibility even deeper. It is an essential characteristic for λ-calculus [Roj15], and together with strict tend to recursive definitions it is a base building stone of functional languages [LS09]. In a book [Sei06], you can find a very cryptic definition for functional language Lisp, which describes it as “the programmable programming language." This unique ability of language to “reprogram" itself makes it far more powerful for a core of electronic circuits simulator than any other software architecture.

Then internal functions and variables of a program can be redefined in the runtime to solve the problem more effectively, or entire simulation process can modify itself to improve simulation parameters. For example by suppressing floating point errors. In the thesis is presented a state of the art implementation of simulation in programming language LISP and its adaptation in structured programming languages as C/C++, Java.

1.2 Aims and Contributions of the Thesis

The thesis primarily focuses on an improvement simulation algorithms, performance, and accuracy in general but from the practical reason of having a reliable reference it compares results with capabilities of SPICE simulator, more precisely, with its open source continuation NgSpice. Three primary research goals were set to provide contributions of this thesis:

- **Scalable simulation procedure** presenting simulation process able to simulate circuits from variety size from small to large scale circuit with preserved simulation performance. Additionally, because of the unique properties of the circuit matrix that is constructed before each simulation, new sparse matrix indexing techniques developed especially for large-scale circuit simulation were developed, and their definition and functionality are also part of this thesis.

- **Definable simulation solver** that proposes unique simulation procedure with dynamical readjustment of internal simulation processes. The core idea is based on a functional definition of simulation process based on anonymous function definition. Capabilities of the solver are also presented together with developed functional chaining method allowing to simulator definition of analysis through the chained functions that evaluate themselves directly upon a call.

- **Simulation accuracy** and reliability of the results precision of the be allowing to improve simulation accuracy by usage of arbitrary precision numbers. The certain attention is also devoted to stability and convergence properties of numerical integration methods applicated in SPICE simulator. The primary aim of presented algorithms is to improve simulation accuracy without computation performance reduction.

1.2.1 Aims of the Thesis

This thesis is organized as follows:

- **Chapter 1: Introduction** gives a brief overview of the current situation of the studied problem and aims of the thesis.
- **Chapter 2: SPICE** In this chapter, are presented basic concepts of simulation of electrical circuits. It follows simulation procedure implemented in SPICE simulator and briefly explains each step.
- **Chapter 3: Scalable Solver** In this chapter the algorithms for SPICE simulation core allowing scalability of the simulation of electrical circuits are described. By the term of scalability is meant an implementation of such computation procedures that optimize analysis performance regardless of problem size. The proposed method is based on adaptive usage of direct and iterative BiConjugate Gradient Stabilized method with a support of incomplete LU factorization.
- **Chapter 4: Sparse Matrix Storage** The high sparsity of the simulation matrices is a typical property of simulation of electronic circuits. In this chapter are presented new sparse matrix indexing techniques with dynamically sizeable containers for sparse matrix simulation indexation indices optimized for use in circuit simulators. The comparison of the performance of standard matrix storages and novel sparse matrix ordering methods is summarized.
- **Chapter 5: CLASP** In Chapter 5 fundamentally new approach to computer-aided design and simulation of electrical circuits based on a use of λ-calculus for circuit device model representation is presented. This approach differs from a traditional procedure where all devices were closed structures that enter simulation as object references. The new principle of automatic self-modifying device models has enabled complete freedom in a definition of methods for the optimized solution of any problem and speeding up the entire process of simulation.
- **Chapter 6: Definable Solver** This chapter puts proposed approach from Chapter 5 to context of a recent development in programming languages. The functional definition is more generalized, and it is introduced in a form applicable to any present day programming language that defines functional programming functions and unnamed functions. The main intention is to build computation layer that separates strong relations between device models and simulation definition. When the simulation core is stripped from these stiff bounds, it is more applicable to a process of parallelization and the most importantly self-optimization.
- **Chapter 6: Methods Improving Accuracy** In this chapter methods for improvement simulation accuracy with preserved performance are presented.
- **Summarized**, Chapter 2 serves as an introduction of classical simulation algorithm used in standard SPICE family programs. Chapters 3 - 7 present new results gathered during my Ph.D. studies.

Chapter 2
SPICE

In this chapter are summarized fundamental concepts of analysis electronic circuits. It also presents implementation difficulties together with actual implementation in simulation program SPICE. The real electronic devices such as resistors, capacitors, diodes, or MOS transistors are represented in simulation programs by their software models. Those models are interpreted in the simulation core as a set of mathematical equations with the purpose of accurately approximating device behavior in a particular region of operation. It is a fact that device models become not very reliable when simulation gets out of assumed operation bounds. It is mainly caused by the developers intent to keep models equations simple and efficient with fast convergence behavior while they are still precise within their operation point reagon [MIT01, AMM93]. If simulation takes place within the device's model bounds, and there is no error in circuit design the next stage of device modeling is a rate of affection by simulation changes.

2.1 Simulation Overview

In the SPICE program, there are four fundamental analyses for simulation of electronic circuits:

- Transient Analysis,
- Operation Point (DC),
- Small Signal Analysis
- Noise Analysis.

All of them shares a common family of computation algorithms. As it has been mentioned in many papers, [Sha93, GZ91, VC93, Dob06, Dob02] transient analysis is the most demanding type of analysis regarding simulation performance, accuracy and computation convergence of a solution. A computation of the transient analysis is divided into a sequence of quasi-static evaluations. During this sequence, various algorithms and techniques must be applied to obtain correct solution such as Newton-Raphson (NR) iterative algorithm, implicit numerical integration method (NI), Gaussian elimination and sparse matrix techniques. A simple overview of simulation process in SPICE can be made from the following algorithm 1 that characterizes simulation of the circuit with nonlinear devices performed by transient analysis. More rigorous description of SPICE algorithms can be found in the books of [Nag75, CL75, TB09].

In particular, when time dependent devices enter a simulation, it also needs to enumerate numerical integration using Trapezoidal, Gear or similar NI method. It should be pointed

Algorithm 1 Transient Simulation of Nonlinear Circuit

```
Initial computation of operating point
repeat {Time interval loop: T_min → T_max}
  repeat {Newton-Raphson iterative loop}
    Linearize semiconductor device around operating point
    Load linear conductances in circuit matrix
    Solve Linear Equation
    if Convergence then
      Increment time
      Break
    else
      Define new trial OP
    end if
  until Maximum number of iteration loops
  if not (Convergence) then
    Error: Maximum number of iteration loops exceeded
  end if
  Discretize differential equations in time
until End of time interval
```

out that computation of each time-step requires a solution of the linear system produced by Jacobian matrix. It is done by direct LU factorization method based on modified Crout algorithm.

2.2 Characteristics of Electronic Circuit Simulation

In program SPICE, modified nodal formulation (MNF) [HRB75b] and the sparse tableau formulation (from ASTAP) [HBG71] are used for the composition of sparse matrix system representing a set of simulation problems. The formal description of the MNF considers, among other things, that for each independent voltage node the size of the circuit matrix must be increased. With an assumption that each new voltage node introduces a constant number of new nonzero values to the matrix, it is evident, that sparsity will rapidly grow with matrix dimension. MNF typically produces a system of nonlinear first-order differential algebraic equations of the form:

$$F(\mathbf{x}, \dot{\mathbf{x}}, t) = 0, \tag{2.1}$$

where $\mathbf{x}$ is the vector of unknown circuit variables, $\dot{\mathbf{x}}$ is the derivative of vector $\mathbf{x}$, with respect to time t. In real implementation values of $\mathbf{x}$ and $\dot{\mathbf{x}}$ depend on current state of the simulation. In linear algebra, electronic circuit matrices represent one special group of linear problems. They can be characterized as extremely sparse matrices whose density of nonzero values quadratically decreases with dimension grow. Visualization of reordered circuit matrix of resistor mesh circuit after MNF and matrix pivoting that was constructed from 400 simple linear device models can be seen in Figure 2.1. Even though nodes of used resistors were mostly connected to at least four devices, the matrix sparsity is considerable.

Time dependency of the circuit is resolved over a specified time interval $(0, T)$ by numerical Trapezoidal method or by Gear integration method. Both methods transform continuous time into a sequence of discrete time points with variable step size. Before computation both (or each step in a case of automatic timestep control) methods must determine the size of

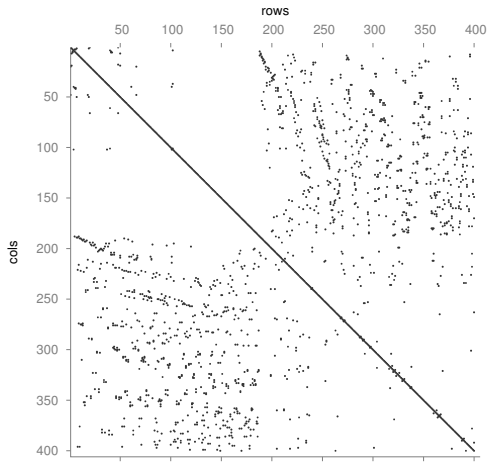


Figure 2.1: Visualisation of resistor array after reordering

the following integration step. Determination of size of consecutive step h_{n+1} is performed with Local Truncation Error algorithm that is derived from Taylor series approximation [WJM+73]:

$$h_{n+1} = \sqrt{\frac{7 \cdot E}{\max\left(\frac{DD_3}{12}, \epsilon_a\right)}}, \tag{2.2}$$

where E is the upper bound error (a maximal value of heuristically scaled solution of x and $\dot{x}$) and $\frac{DD_3}{12}$ is divided difference of order three. The algorithm checks whether it was computed with the sufficiently small step. If a size of the step was less than 0.9 of previous one t_n (actual time is t_{n+1}) the step is rejected, and the solution must recompute it with a smaller step [Coh75] [NV11]. Then the solution of x_{n+1} at discrete time point t_{n+1} is given the previously computed one.

$$t_{n+1} = t_n + h_n \tag{2.3}$$

SPICE simulation procedure with incorporated numerical integration procedure (NI) is summarized in Algorithm 2.

Algorithm 2 Numerical Integration

```
Select step size h
Calculate t_0 = 0 state (OP)
repeat
  t_{n+1} = t_n + h
  Numerical integration (Trapezodial / Gear method )
  Nonlinear solver
  Calculate the error
  if The error is too large then
    Reject step t_{n+1}
    Reduce h
  else
    Store timepoint solution
    t_n = t_{n+1}
  end if
until t < T
```

After the numerical integration, (2.1) set of simulation equations is reduced to a system of nonlinear equations whose coefficients differ over discrete integration steps. Program SPICE uses Newton-Raphson (NR) iteration algorithm to linearize the problem with various self enhacements such as the possibility of damping to prevent oscillations in iteration steps. The simplified procedure that is implemented in NgSpice simulator is presented in Algorithm 3.

Algorithm 3 Nonlinear Solver

Initial value estimate $\mathbf{x}_0$
repeat {Newton-Raphson}
  $\mathbf{Jx} = \mathbf{b}$ (Linear solver)
  if Damping required **then**
    $\kappa_{df} = \max(0.1, 10/\max(\mathbf{x}_{n+1} - \mathbf{x}_n))$
    $\mathbf{x}_{n+1} = \mathbf{x}_n + (\kappa_{df} * (\mathbf{x}_{n+1} - \mathbf{x}_n))$
  end if
  Compute residual $\|\mathbf{x}_n - \mathbf{x}_{n+1}\|$
until Stopping criteria

It should be noted that SPICE does not compute Jacobian matrix in a standard manner and certainly does not compute costly matrix inversion. Instead, each iteration is evaluated from:

$$J(\mathbf{x}_{n+1})\mathbf{x}_{n+1} = J(\mathbf{x}_n)\mathbf{x}_n - F(\mathbf{x}_n) \tag{2.4}$$

where $J(\mathbf{x}_{n+1})$ is a coefficient matrix with the constant vector as its right-hand side. (It should be noted that it is only efficiently redefined Newton's method). The linear equation is repeatedly solved, updating the coefficient matrix and constant vector each time, until x_{n+1} has converged sufficiently. Regardless of the computation procedure, one iteration represents a simple linear system of the form:

$$\mathbf{Jx} = \mathbf{b} \tag{2.5}$$

with the unknown vector $\mathbf{x}$, constant right-hand side $\mathbf{b}$ and sparse matrix $\mathbf{J}$. In program SPICE, it is solved in three steps. Initially, the algorithm performs pivotization and row reordering based on a computation of Markowitz criterion. Although matrix reordering is the most demanding operation (as can be viewed from Table 3.2), it is essential for further computation and therefore must be performed regardless of the method used. After pivoting, the result of the linear system is obtained by LUF method optimized for sparse matrices followed by very efficient Forward Elimination and Back Substitution as shows Algorithm 4.

Algorithm 4 Linear Solver

if Reordering required **then**
  Pivoting (Markowitz pivot strategy)
end if
Crout LUF (Row at the time)
Forward elimination and back substitution (FEBS)

It must be clear now that the most critical algorithm of all the processing is the solution of the linear system representing circuit state in a given time step. As it has been said, in linear algebra, electronic circuit matrices represent one special group of linear problems. They can be characterized as extremely sparse matrices whose density of nonzero values quadratically

decreases with dimension. They are certainly nonsymmetric but permutable to a block of triangular form. In particular cases, they can be ill-conditioned or temporarily approach the singularity. The performance of all operation such as factorization, row ordering and even convergence depends on the matrix properties. Matrix dimension can be considered as the main one. Regardless of used device models, it is dominantly given by the topology of simulated circuit. Based on the dimension of simulation matrix n, circuit simulation can be divided into three groups:

$$\begin{cases} \text{Trivial} & n < 50 \\ \text{Standard} & n \in (50; 5000) \\ \text{Large} & n > 5000 \end{cases}$$

1. In trivial cases, simulation problem is so small that circuit matrix can be considered as almost dense. In that situation, a complicated implementation of the solver of sparse linear systems become meaningless, and can be outperformed by "naive" method. Although relative performance difference may reach 50 percent, in absolute numbers it is clearly below of meaningful resolution.

2. It is denoted as a standard problem where the algorithm implemented in SPICE for the direct solution of a linear sparse system has no competition. It proves to be a fast solver without extensive routines and able to work with real or complex numbers. It is tough to outperform it even with decent sparse matrix systems such as UMFPACK[Dav06] and SuperLU [Li05].

3. As large can be considered a linear system compiled from several sub-circuit parts resulting in a matrix of dimension higher than 5000. In the past, this discipline belonged to supercomputers and was taken as a hypothetical possibility. With increasing performance of computers, it is not difficult to perform a simulation of this size in reasonable time. For example, to find a DC solution of the matrix with 41000 non-zero values and dimension 5000 takes less than 0.5s. [DČY11] [CD15]

The fact that it is tough to outperform the SPICE internal direct solver in a region of standard size problems stops to be valid for large scale systems. In that case, iterative numerical methods based on successive approximations gain in importance. Especially methods from family of nonstationary methods, applicable to a nonsymmetric linear problems such as GMRES [SS86], BICGStab [vdV92], QMR [Not93].

2.3 Integration Methods in SPICE

The Transient Analysis simulation implemented in SPICE first solves DC operating point for all voltage nodes; this is not always necessary, sometimes DC solution can be automatically skipped. Then the result is passed as an initial value for next computing time. Seldom, all devices are linear in the circuit. Therefore simulator must linearize all nonlinear differential equations by iteration of NR iterative algorithm and implicit NI method. Following equation defines general category of polynomial integration method

$$x_{n+1} = \sum_{i=0}^{n} a_i x_{n-1} + \sum_{i=-1}^{n} b_i \dot{x}_{n-i} \tag{2.6}$$

If b_{-1} is zero, method is explicit, and if b_{-1} is nonzero, the method is implicit. The algorithm is a multi-step algorithm if $n > 1$, that is if more than one point from the past is needed to compute next one x_{n+1}.

In SPICE family programs it user can choose one from two different NI methods. The default one used to be trapezoidal (TRAPEZ) method, but it does not to be valid in a new simulators where it is usualy overcomed by the second possibility that is Gear method (GEAR). In any cases, the TRAPEZ method is used as a fail-safe method when integration by GEAR method diverges. More details about these methods can be found in the works by [Nag75, McC88, HNW08]. It could be noted that both methods work as implicit NI, but for i.e. in NgSpice explicit NI method can be added to computation core to improve computation. The form of TRAPEZ method, which is used in NgSpice, can be simply derived from the following equation

$$\dot{x} = f(x, t) \tag{2.7}$$

the general solution at time t_{n+1} is

$$x_{n+1} = x_n + \frac{1}{2}h(f(x_n, t_n) + f(x_{n+1}, t_{n+1})) \tag{2.8}$$

from which can be finally obtained

$$\dot{x}_{n+1} = f(x_{n+1}, t_{n+1}) = -f(x_n, t_n) + \frac{2}{h}(x_{n+1} - x_n) \tag{2.9}$$

Therefore TRAPEZ method is recognized as a second-order method. When convergence test failed, SPICE changes the order of method to one that is simple implicit Euler method, More complex description of implemented algorithms can be found in article [GZ91]. An implementation of the GEAR algorithm in SPICE is done by a standard implicit formulation of this method, that can be found in various mathematical literature [HNW08].

2.3.1 Automatic Time Step Control

Automatic time step control (ATSC) is NI algorithm, that sets size of the iteration steps during time domain analysis. The size of these steps may vary over many decades to improve accuracy or speed of NI algorithm. Let's assume interval given by t_{start}, t_{stop} and t_{step}, which denotes a beginning, an end and the step size of NI method respectively. When the transient analysis is performed, SPICE takes this interval and evaluates a number of steps together with maximal initial step size. It is done by rough estimation by following algorithm

if $(t_{stop} - t_{start})/50 > t_{step})$ **then**
  $t_{max} := t_{step}$
else
  $t_{max} := t_{step} := (t_{stop} - t_{start})/50$
end if

It should be pointed out, that it computes transient analysis at least in 50 points insensibility of step size and interval bounds defined by the user. Because the solution is evaluated by iteration algorithm, it can happen that the program could not be able to find any solution, and a maximal number of iterations will be exceeded. In that case, ATSC decrees the time step and change the order of NI to one.

It is necessary to check whether the solution is sufficiently close to right solution. Every solution, that was computed by numerical integration method, must by checked for its accuracy. It is done by local truncation error algorithm (LTE), that is based on the upper bound E, a

maximal value of heuristically scaled solutions of x and $\dot{x}$). Solution of LTE is equal to upper bound of size of the next step and is computed by following inequality:

The default value for CHGTOL is 10^{-14}

$$E = max\left(\epsilon_{\dot{x}}, \epsilon_x\right) \tag{2.10}$$

$$h_{n+1} \leq \sqrt{\frac{6E}{\left|\frac{d^3x_n}{dt^3}\right|}} \tag{2.11}$$

The equation above is of course unsuitable for numerical evaluation. Therefore SPICE implementation introduce LTE equation in better form for numerical computing:

$$h_{n+1} = \sqrt{\frac{7 \cdot E}{max\left(\frac{DD_3}{12}, \epsilon_a\right)}} \tag{2.12}$$

where $\frac{DD_3}{12}$ is divided difference of order three.

2.3.2 Step Rejection Algorithm

Let's assume a solution obtained in time t_{n+1}. The algorithm checks whether it was computed with a sufficiently small step. If the desired size of the step was less than 0.9 of previous one t_n, notice, the present time is t_{n+1}, the step is rejected, and the solution must recompute it with the smaller step.

Compute $h_{n+1} = f(LTE)$
if $h_{n+1} < 0.9 \cdot h_n$) **then**
  reject t_{n+1}
  recompute new t_{n+1}
  $h_n = h_{n+1}$
else
  accept t_{n+1}
  increase order of method
end if

The order of the method is increased only if and if the new value of t_{n+1} is accepted.

2.4 Chapter Summary

In this chapter, were presented basic concepts of simulation of electrical circuits that are implemented in SPICE and its open-source continuation NgSpice. The main attention was devoted to Transient analysis that belongs to the most complex simulation types. There are also other complicated analyses as Pole-Zero Analysis, Noise Analysis, Sensitivity Analysis, however, even that the internal processes in that particular analysis differ all of them uses the algorithms from the same family of implemented algorithms in SPICE simulation and computation core.

Chapter 3

Scalable Solver

By the term of scalable linear solver is meant an implementation of such computation procedures that optimize analysis performance regardless of problem size. Standard direct solver implemented in SPICE ceases to be efficient for large linear systems. In literature GMRES method is usually presented as the best choice for an iterative solution of large linear systems [SS86]. It is applicable to nonsymmetric matrices and leads to the smallest residual with significantly reduced number of iterations. However, there are some aspects that makes GMRES problematic for practical implementation. The first one, related to nature of the method, is that it is difficult to limit the number of iteration steps in a standard manner. The computation time depends on the limit of available storage and number of restart points. The correct settings of those variables depend on matrix and the right-hand side vector properties. The fact that the algorithm does not stop iteration at certain point makes this method unsuitable as a competitor to direct solvers. The second aspect relates to relatively high memory requirements of the method. As the most promising proved to be Biconjugate Gradient Stabilized method (BICGStab) [vdV92]. It is often described in literature as a fast and stable method avoiding irregular convergence behaviors [BBC+94]. The second promising algorithm method turned out to be Quasi-Minimal Residual (QMR) [FN91] that in some particular cases can be even faster than BICGStab method. As problematic can be taken its slightly worse stability than the one obtained with BICGStab.

3.1 Iterative Methods for Linear Systems

Iterative methods refer to techniques that use successive approximations to obtain a more accurate solution to the linear system at each step. There are so-called Stationary methods and the Nonstationary methods based on the idea of sequences of orthogonal vectors [BBC+94]. Stationary methods perform in each step a same operation on the current values. In the non-stationary method, iterations depend on coefficients. From stationary methods it can be point out following methods:

- The Jacobi Method (JACOBI)
- The Gauss-Seidel Method (GS)
- The Successive Overrelaxation Method (SOR)

These method are presented mostly from historical reasons. Despite of their simplicity they are mostly applicable to strictly diagonally dominant, or symmetric positive definite matrices. For other cases they suffer from slow convergency (or they diverge). In general they are not competitive with the nonstationary methods and certainly not with direct methods. From the

Nonstationary method (sometimes reffered as Krylov subspace methods) could be pointed out following methods:

- Conjugate Gradient Method (CG)
- Generalized Minimal Residual (GMRES)
- Conjugate Gradient Squared Method (CGS)
- BiConjugate Gradient Stabilized (BICGStab)
- BiConjugate Gradient (BICG)
- Quasi-Minimal Residual (QMR)

The properties of those methods differ a lot. CG method is best applicable to positive definite systems that is not a case of the matrices constructed by MNF. From literature [LS05] GMRES results as the best applicable method for nonsymetric matrices that leads to the smallest residual. As it will be shown GMRES proved to be problematic from implementation point of view, because it does not limit number of iteration steps in standard manner. The computation time depends on the limit of available storage and number of restart points. The correct settings of those variables depends on matrix and righ-hand side vector properties.

The fact that simulator is unable to stop iteration at the certain point makes this method unsuitable as competitor method to direct solver. It is a special property of MNF that regardless of matrix condition number, number or nonzero entries and matrix dimension will not change during a simulation. Therefore time needed for solution by the direct method will remain constant during computation and of course will be known from initial t = 0 solution. It is a limiting time for an iterative method and should never be exceeded otherwise usage of iterative becomes meaningless. In the Table 3.1 there is a comparison of different stationary and nonstationary methods applicable into NgSpice simulator and the computation results after simulating the particular circuit. The abbreviations of simulated circuits in the table stands for:

- Linear - Simple linear circuit mostly constructed from resistors interconnected to mesh.
- Nonlin. and Nonlin. L - Simple nonlinear circuit constructed of resistors and basic nonlinear device models as diodes and capacitors. Abbreviation L stands for Large circuit.
- Shift - CMOS stepping regulator circuit
- Dif. Amp. - Differential Amplifier
- Adder - CMOS Adder constructed from MOS transistors

The table also presents results from NgSpice internal direct solver (row SPICE). Each cell of the table holds final precision of the computed simulation. The cells of the table with “x” symbols denotes states where methods diverge. It is also important to note that particular methods in some cases resulted in convergence but performing backward check proved that resulting numbers are totally wrong. It is a case of JACOBI, PCR, and SOR. This states can be denoted as “false convergence” and are in practical implementation more dangerous than standard error. As the most promising method for implementation into NGSpice simulator proved to be BICGStab, GMRES and QMR methods.

Name	Linear.	Nonlin	Nonlin. L	Shift	Dif. Amp.	Adder
Spice	2.55e-12	2.78e-17	1.15e-12	3.24e-06	9.103-16	7.11e-05
Jacobi	8.41e-01	1.06e-04	1.36e-12	3.58e-08	x	5.44e-18
Chebi	x	x	x	x	x	x
SOR	7.67e-01	8.69e-05	8.19e-05	2.20e-15	x	6.61e-17
GS	6.94e-01	8.07e-05	1.71e-12	3.59e-14	x	5.98e-17
CG	2.28e-11	5.70e-17	x	x	x	6.13e-11
PCG	1.97e-11	x	x	x	x	1.60e-11
PCR	9.43e-01	1.35e-12	x	4.97e-01	2.58e-02	1.53e-03
GMRES	1.83e-12	3.36e-16	2.78e-10	2.15e-15	1.77e-11	5.09e-16
CGS	x	1.97e-16	1.68e-11	7.26e-14	1.62e-12	5.32e-15
BICGS	3.10e-10	3.96e-18	4.99e-12	3.15e-14	3.31e-11	3.19e-15
QMR	1.53e-11	2.92e-17	2.29e-11	3.01e-14	5.86e-12	9.52e-15

Table 3.1: Applicability of stationary and nonstationary methods on simulation of electrical circuits

Dim.	802		7138		20098	
	Op.	Δt (sec)	Op.	Δt (sec)	Op.	Δt (sec)
Reorder	4	2.14e-2	4	5.49e-1	4	8.37
LUF	45	1.96e-3	174	0.419	175	2.18
FEBS	45	9.31e-4	174	0.226	175	1.22

Table 3.2: Computation times for different parts of NgSpice direct solver

3.2 BiConjugate Gradient Stabilized (BICGStab)

The definition of BiConjugate Gradient Stabilized method (BICGStab) algorithm [vdV92] as it can be found in the literature is usually written in a form of mathematical pseudocode. It is not very explanatory especially if presented without preceding mathematical derivation, although that algorithm is actually very simple. Therefore its implementation is showed in GNU Octave [EBH+15] source code 3.1. It should be noted that for real implementation that algorithm must be redefined to be able to work with large sparse matrices and also computation with preconditioner needs to be revised. In the presented code (3.1), this is suppressed with a use of backslash operator "\" which performs direct solution of given linear problem. For a more real-world implementation of the algorithm, it is recommend to look at the particular method in source codes of Portable, Extensible Toolkit for Scientific computation PETSC [BGMS97, BAA+16]. There is also additional problem connected with a usage of iterative methods. With a respect to direct ones it is evident that number of iterations is key factor that affects method performance. It is evident from the example that every additional iteration requires a certain number of operations that will prolong computation time. It must be reduced so that the method could compete with the direct solver. The optimization can be done in several ways:

- reducing precision or iteration number,
- more precise initial estimate,
- preconditioner (3.5).

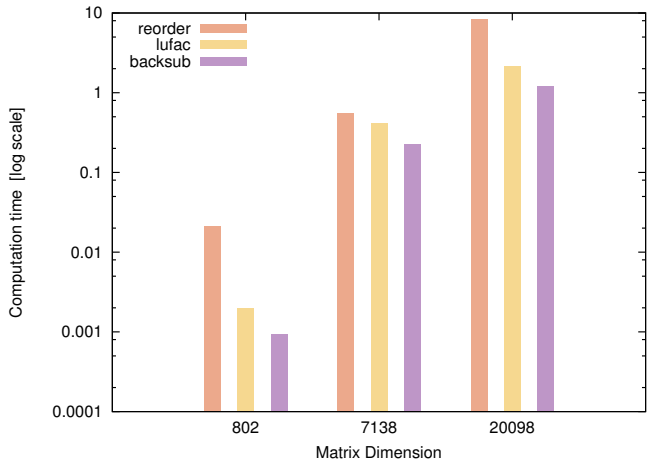


Figure 3.1: Computation times for different parts of NgSpice direct solver

Listing 3.1: Simplified Implementation of BICGStab in GNU Octave

```
function [x, flag, relres, iter] =
  bicgstab(A, x, b, M, tol, maxit)
# input   A        matrix
#         x        initial guess vector
#         b        right hand side vector
#         M        preconditioner matrix
#         tol      error tolerance
#         max_it   maximum number of iterations
# output  x        solution vector
#         flag     output (0:OK, 1:error)
#         relres   error norm
#         iter     number of iterations performed
  norm_b = norm(b);
  res = b - Ax(x);
  rr = res;
  flag = 1;
  for iter = 1:maxit
    rho_1 = rr' * res;
    if (iter == 1)
      p = res;
    else
      beta = (rho_1 / rho_2) * (alpha / omega);
      p = res + beta * (p - omega * v);
    endif
     phat = M \ p;
     v = Ax * phat;
    alpha = rho_1 / (rr' * v);
    s = res - alpha * v;
     shat = M \ s;
     t = Ax * shat;
    omega = (s' * t) / (t' * t);
    x = x + alpha * phat + omega * shat;
    res = s - omega * t;
    rho_2 = rho_1;
     relres = norm(res) / norm_b;
     if (relres <= tol)
      flag = 0;
      break;
     endif
  endfor
```

3.3 Reduced Precision or Iteration Number

In the article [Tis99] it is proved that for a stable linear solver with a close enough solution, the norm of residual will be reduced after each step of Newton-Raphson until limiting accuracy is reached. It also states that reduced accuracy of a linear solution may affect the rate of convergence but not limit accuracy. Since the proposed method contains two nested iterative algorithms, it opens a possibility to accelerate computation with a limit of the number of iterations or residual precision of inner method. Mainly for the reason of relatively slow convergence rate of unpreconditioned BICGStab (and all linear solvers in general) in comparison to almost quadratic convergence rate of nonlinear NR, the iteration limit does not provide recognizable performance improvement unless preconditioning is applied. Without it the number of successful iterations of BICGStab method may vary a lot for different linear systems and therefore maximum iteration number must be chosen based on matrix properties. As an indicator may seem to be the matrix condition number efficiently computed by one of the methods published in [Dea15, HT00]. They allow estimation of condition number even in the case of highly sparse matrices. Although, it can give us some information about invertibility of a matrix and also absolute achievable precision, it does not prove to be a reliable indicator of iteration numbers for the BICGStab method. A more efficient approach proved to be the use of preconditioner with fixed maximal iteration number.

3.4 Initial Estimation

Iterative algorithms allow to specify an initial iteration guess. It is a well-known fact that an initial estimate close to the solution rapidly reduces the number of required iterations. It works extremely well for NR method implemented in Spice simulator and at the first sight it may seem to apply to inner iterative linear solvers. The main issue is given by varying properties of a linear system between iteration steps. Iteration steps computed from Jacobian matrix can transiently become singular or have extremely large condition number. In that situation, it can not be expected that the solution from the previous iteration be any close to the subsequent one (it is a big issue for algorithm presented in [LS05]). Also, linear solvers are more sensitive to wrong initial guess than nonlinear methods when the wrong guess can cause not only increase of required iterations but also a divergence of the method. It is illustrated in Figure 3.2. Example shown was performed on tridiagonal matrix with dimension 200 defined in 3.1.

$$a_{i,j} = \begin{cases} 2 & \text{for} \quad i = j \\ -1 & \text{for} \quad i = j - 1 \\ 1 & \text{for} \quad i = j + 1 \\ 0 & \text{otherwise,} \end{cases} \tag{3.1}$$

3.5 Precoditioner

The term preconditioning refers to a technique of transformation of a linear system to another with more favorable properties. In general, preconditioning attempts to improve the spectral properties of the coefficient matrix. If $\mathbf{M}$ is a nonsingular matrix that approximates $\mathbf{J}$, then the linear system

$$\mathbf{M}^{-1}\mathbf{J}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \tag{3.2}$$

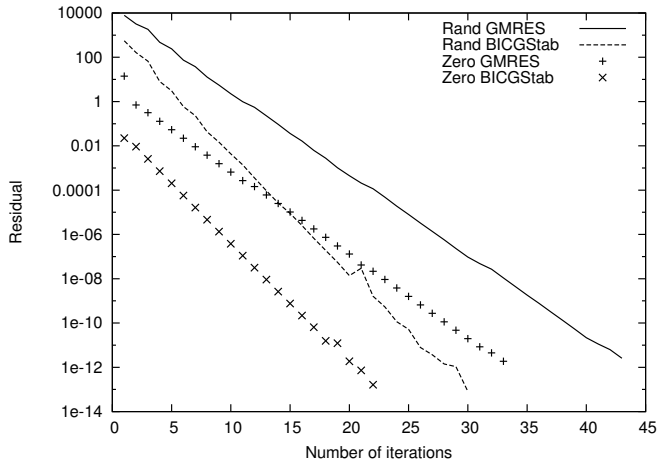


Figure 3.2: Effect of starting values on method convergence rate

is better applicable to computer implementation. In practice, $\mathbf{M}^{-1}$ is not directly computed. It would lead to a loss of sparsity. Instead, matrix-vector products with $\mathbf{J}$ and solutions of a linear system are performed. Additionally, left, right or symmetric preconditioning is also possible. Residual minimizing methods as GMRES and BICGStab most often use right preconditioning. The choice of a good preconditioner is a trade-off between spending more time computing it or having a slower convergence of the method.

3.6 ILU preconditioner

With growing dimension a linear system gains on the significance of probability of fill-ins. Although reordering and pivotization try to minimize them, it is often not possible to fully avoid them. It implies that to compete with the direct solver it should use a preconditioner that entirely avoids fill-ins but has similar properties. As the best for computation of highly sparse nonsymmetric matrices sorted out Incomplete LU Preconditioner (ILU). The primary intention of the ILU is to suppress possible fill-ins maximally. Regardless of modification of the method, it is done simply by dropping them. For subset of nonzero positions in the matrix S, one factorization step can be formally described as

$$A_{i,j} \leftarrow \begin{cases} A_{i,j} - A_{i,k}/A_{k,k}^{-1} a_{k,j} & \text{for } (i,j) \in S, \\ a_{ij} & \text{otherwise,} \end{cases} \tag{3.3}$$

for each k and for $i, j > k$ [Saa99]. It is important to note that the incomplete factorization may fail due to division by zero. It indicates that matrix pivoting can not be avoided even with usage of iterative methods. From an implementation viewpoint, it is not a problem because it is already there and used by the direct method. The implementation of the algorithm in GNU Octave code is shown in Algorithm 3.2. It should be noted that real implementation should be able to work with sparse matrices. Iteration over all indexes (even zero ones) as it is in the algorithm is not acceptable. Convergence rate of computation of real CMOS Voltage shifter circuit with ILU preconditioned BICGStab method and without it are in Figure 3.7. The improvement in convergence rate in both methods GMRES and BICGStab is indisputable.

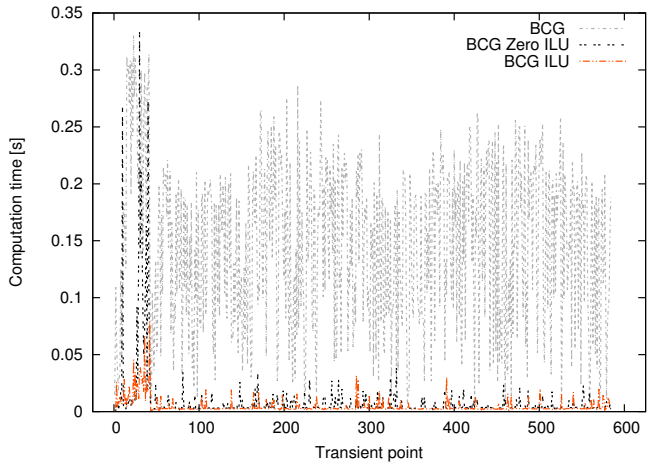


Figure 3.3: Comparison of BICGStab method performance for computation of each iteration step

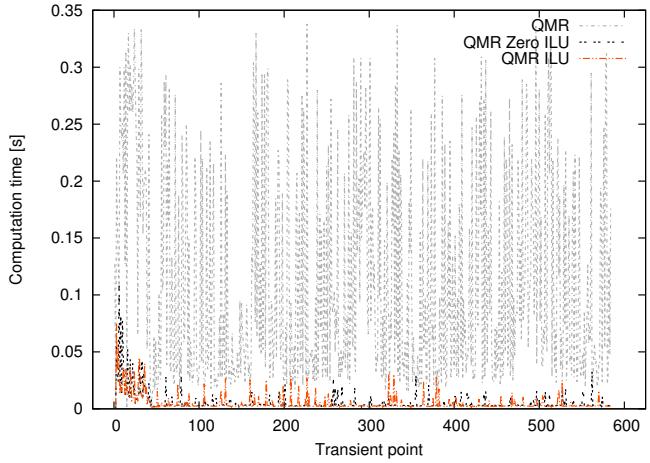


Figure 3.4: Comparison of QMR method performance for computation of each iteration step

Listing 3.2: Incomplete LU Factorization Algorithm Octave

```
function [M]= ILU(A)
# input     A       input matrix
# output    M       incomplete LU factorization
  n = length(A);
  M = A;

  for k = 1:n-1
       for i = k+1:n
      if A(i,k)==0
        continue
       end
       M(i,k) = M(i,k) / M(k,k);
       for j = k+1:n
         if A(i,j)==0
           continue
         end
         M(i,j) = M(i,j) - M(i,k) * M(k,j);
       end
    end
  end
```

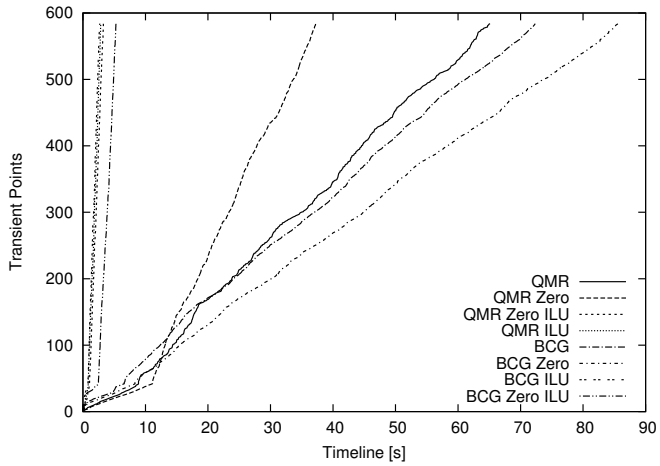


Figure 3.5: Computation performance of Nonstationary methods

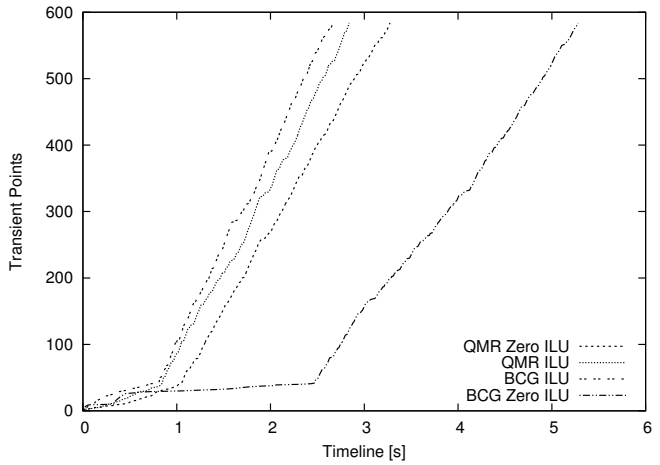


Figure 3.6: Detail in performance of QMR and BICGStab

The effect of usage of preconditioner in real simulation performed by NgSpice simulator can be seen in Figures 3.8 and 3.9. It shows average time (on a log scale) needed for the linear solver to compute solution for given iteration step of NR method. The graph was constructed as the average value of hundred repeated analyses of CMOS shifter circuit, but similar results were obtained with other circuits. A very high computation time should be noted in the beginning of simulation with not preconditioned solver 3.8. At this point, OP of the circuit is computed. It is computed with an initial random guess for NR that can even cause singularity of the system. Consequently, it causes longer computation time for the iterative linear solver. This effect is suppressed with a use of preconditioner as shown in Figure 3.9.

3.7 Implementation Note of Scalable Solver in SPICE

The standard simulation procedure needs to be modified to allow scalability of the solver in SPICE simulator. All implementations and tests were performed with simulation program NgSpice [NV16], which is an open source continuation of no longer developed Berkeley's version of SPICE. The first attempt to implement iterative linear solvers was made with a source code presented in the book TEMPLATES Iterative Solvers for Linear Systems [BBC+94]. Even though, the algorithms presented in the book can be implemented from

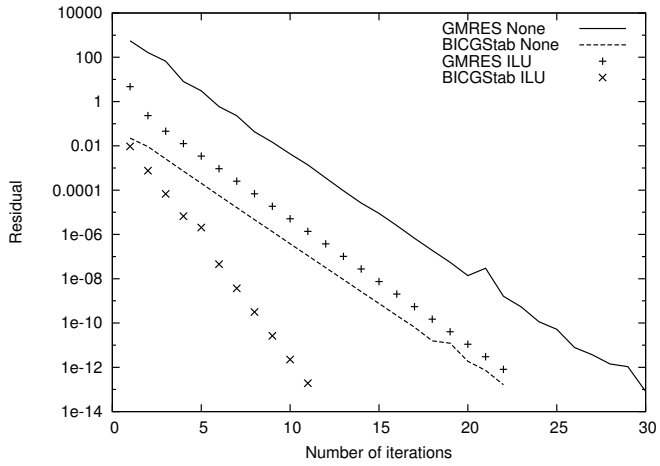


Figure 3.7: Effect of used ILU preconditioner on convergence rate

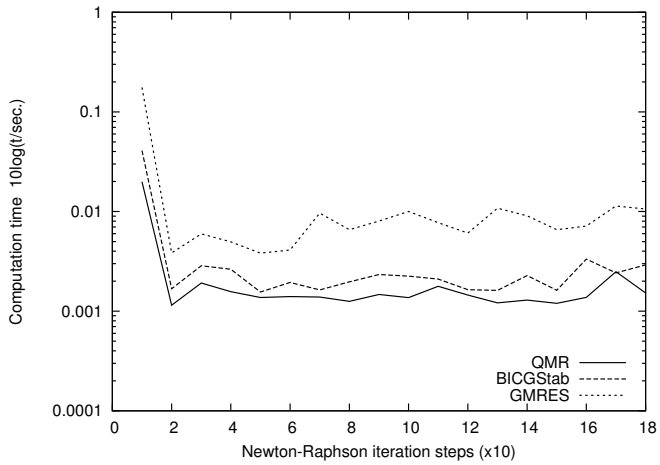


Figure 3.8: Required time for each step of transient analysis (Not-Preconditioned)

their source code examples in a very short time, they showed to be not applicable to NgSpice simulation core due to the lack of missing sparse matrix storage support. While they can provide some idea of the difference in performance of algorithms and their convergence properties, it is not possible to compare them with direct methods optimized for sparse matrices implemented in NgSpice. Therefore, it is strongly recommended to use PETSc library [BGMS97, BAA+16]. It provides a full variety of iterative linear solvers and even some experimental nonlinear ones. It also provides an interface to external direct solver libraries, such as UMFPACK[Dav06] or SuperLU [Li05]. There is one particular problem with the implementation and it is incompatible sparse matrix storage between NgSpice and PETSC. Additional translating routines can resolved it but for future development, it is suggested to rewrite entire proprietary NgSpice sparse matrix storage system to one implemented in PETSC. The effect of the modification of original algorithm is showed in Procedure 5. In the case of small and normal size circuits, it will primarily use the direct linear solver but for large circuits (of a dimension > 5000) it will perform the iterative algorithm. BICGStab method combined with ILU preconditioner proved to be the best iterative method for electronic circuit simulations. The modified algorithm is derived from PETSC functions and their input parameters. Based on matrix size (parameter MATSIZE) it decides whether to use the iterative or direct algorithm. It also allows predefining of the method for given simulation

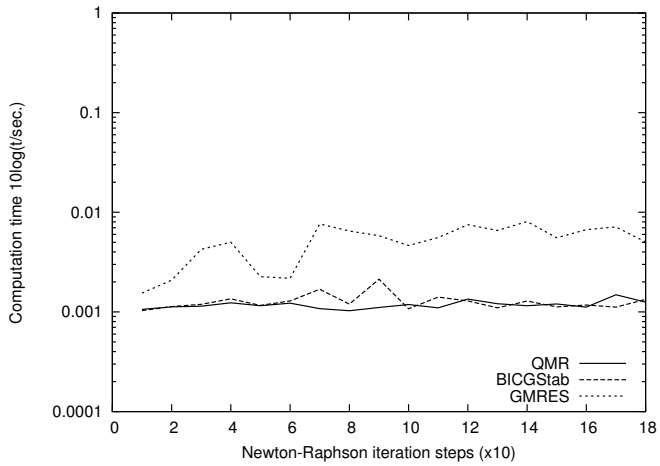


Figure 3.9: Required time for each step of transient analysis (Preconditioned)

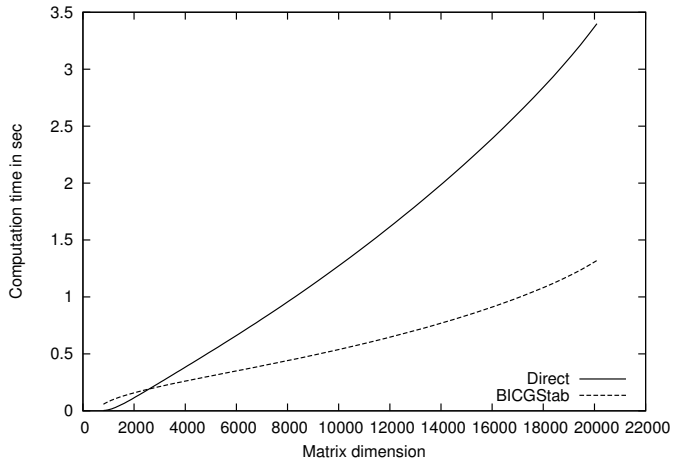


Figure 3.10: Performance comparison of iterative BICGStab and direct method implemented in NgSpice

with variable LINSLV (type of linear solver). Iteration method by its nature computes at each step residual of the result. In algorithm it can be omitted or defined to compute it after a specific number of steps to improve performance. Hovewer, tests proved that avoiding residual computation has a very small effect on overall performance when ILU preconditioner is used. On the other hand, the direct method does not compute residual of the solution at all and therefore it needs to be explicitly added for use of consecutive procedures such as NR. Additionally, proposed implementation of the algorithm can switch to fails safe operation and try to obtain a solution with the direct method based on a return parameter LINSTOP of the stopping event. LINSTOP corresponds to PETSC return codes which for positive values indicates convergence and for negative values until -5 method divergency. Lower values than -5 (RETURN CODE) indicates that the process was interrupted by error mostly caused by the occurrence of NaN or INF values in a sparse array. A manual specification of the method should be available to the user through a set of redefinable properties. In a similar manner as used in NgSpice it specify the additional set of simulation parameters:

```
LINSLV # type of linear solver (DIRECT, ITER)
KSPTYPE # linear method BICGStab (Default)
KSPPREC # preconditioner ILU ( Default)
KSPRELTOL # relative stopping tolerance
```

```
KSPABSTOL  # absolute stopping tolerance
KSPMAXIT   # iteration limit
```

Parameter LINSLV predefines linear solver for the analysis and suppresses adaptive selection based on the size of circuit matrix. PETSC library offers a full range of different linear solvers and preconditioners. They are mostly compatible with each other. Once the library is integrated to NgSpice, it is possible to use all its methods freely (in this case they can be chosen by definition of parameters KSPTYPE and KSPPREC). It also defines parametres restraining maximum number of iterations and the relative and absolute stopping tolerances (KSPRELTOL, KSPABSTOL, KSPMAXIT). In Figure 3.10 there is the comparison of proposed procedure 5 performed by NgSpice with direct method for the solution of linear system. Iterative algorithm BICGStab was implemented with ILU preconditioner. To avoid slow convergence of the method in the case of wrong initial guess of NR method it setups the limited the number of iteration of BICGStab method to 10. The relative precision of the algorithm was set to 10^{-16} and the absolute one to 10^{-50}. Simulation time corresponds to a required time to obtain operation point of given circuit.

Algorithm 5 Scalable Linear Solver

```
if Reordering required then
  Pivoting (Markowitz pivot strategy)
end if
if LINSLV=ITER or n > MATSIZE then
  BICGStab Method
  Obtain LINSTOP
end if
if LINSTOP=BREAK or LINSLV=DIRECT then
  Direct method
  Compute residual
end if
if Residual=NaN or Residual=Inf then
  Stop NR
end if
```

3.8 Chapter Summary

It is clear that performance of computer simulation of electrical circuits depends on the implementation of fast and reliable computation procedures. In this chapter adaptive internal solver was proposed, based on BICGStab for the iterative solution of nonsymmetric linear systems supplemented with Incomplete LU factorization presented in Procedure 5. The algorithm (based on return values from linear iteration cycle) can automatically switch to direct method or directly reject computation step in a situation when iteration results to non-numeral types. It is an very important operation that can during simulation of large scale circuit significantly reduce the number of unnecessary iteration cycles of NR algorithm. The proposed combination of the linear solver and preconditioner greatly improves the convergence rate of the linear method and make it compatible with standard LU solver from circuit matrix dimension up to 5000. For a smaller circuit, the iterative solver can obtain a result with improved precision avoiding errors caused by operation with finite floating point precision. In Figure 3.10 there is a comparison of time required for computation of OP with the use of the direct method and iterative algorithm using developed composition method based on

BICGStab method. In this case, linear solver was automaticaly derived with NgSpice internal methods and therefore no additional "fail-safe" computation was performed. It should be pointed out that very important part of the proposed methid is ILU preconditioner that avoids redundant computations and ensures a constant number of the matrix non-zero entries. It allows to the BICGStab iteration, especially during successful iteration cycles of NR algorithm, to obtain solutions much faster than standard direct method. This difference in performance rapidly grows with the size of the circuit matrix as it is visible from Figure 3.10.

3.8.1 Research Contributions Summary

- Classification of simulation by the size of circuit matrix dimension.
- Self-Adapting simulation procedure mechanism for small, normal, large circuits.
- More efficient simulation procedure for large and extra large circuits based on BICGStab iteration method accompanied with Incomplete LU factorization.

Chapter 4

Sparse Matrix Storage

4.1 Introduction

The high sparsity is the particular property of every circuit matrix constructed by modified nodal analysis (MNA) [CL75] for simulation of the electronic circuits. It could be pointed out that the sparsity of even very simple nonlinear circuit with MOS transistors with circuit matrix dimension ≈ 100 [DČY11] is around 99% . A memory requirement for holding a full matrix of this trivial circuit is nothing extraordinary and is around 80kB in double precision. It is interesting that non-zero data takes only 0.8kB. The difference between zero and non-zero data grows rapidly with increasing matrix dimension as, for example, a matrix with dimension 10^6 holds 7TB unnecessary zero values [TW67]. The number of operations necessary to solve a system of equations by Gaussian elimination or by LU factorization is approximately $n^3/3$, where n is circuit matrix dimension. It is valid if all operations are performed, even those where numbers were multiplied by zero. The computation time in the circuit simulation is reduced by not performing these multiplications and additions (subtractions). In fact, those zeros do not need to be stored. Every efficient electronic circuit simulator must be able to operate with sparse matrices. In this chapter sparse matrix indexation techniquest developed for usage in circuit simulators are presented.

4.2 Sparse Matrix Storages

Sparse matrix algorithms neither store nor perform operations on zero. Primary sparse matrix techniques decompose a two-dimensional matrix to several vectors. The various techniques have been published in the literature, for instance, Yale matrix, Skyline storage, Profile storage matrix, Diagonal storage matrix, Compressed sparse column, Compressed sparse row and many others [BB05, Tew73]. There have been published many approaches about storing sparse matrix entries, but most commonly used is based on decomposition into three value vector [Tew73] where $\boldsymbol{z} = [x, y, v]$ represents a position of matrix entries by column x, row y and cell value v. These known techniques for the sparse matrix manipulation may be impractical when they deal with extremely sparse matrice systems produced by MNA during electronic circuit analysis. To solve this problem, following new sparse matrix ordering techniques optimized for ECS have been developed [CD15] [CD14a]:

- Modified Profile-In Skyline Storage (MPSLS) that is derived from Profile-In Skyline Storage indexation.
- Full Profile Skyline Storage (FPSS) that is new and is based on an order of LU factorization procedures

- Enhanced Full Profile Skyline Storage (EFPSS) that is enhanced modification of FPSS.

4.3 Modified Profile-In Skyline Storage

This method decomposes a matrix into several vectors. Three of them hold matrix values, and two index vectors. Vectors **u** and **l** are both value vectors, that are accompanied by two indexes vector $\mathbf{i}_u$, $\mathbf{i}_l$. Index vectors $\mathbf{i}_u$ and $\mathbf{i}_l$ combine ordering procedures of the row and column sparse matrix techniques. The index vector in $\mathbf{i}_u$ uses *up to down, left to right* organization scheme, vector $\mathbf{i}_l$ works with a *left to right, up to down* scheme. Vector **d** holds diagonal matrix values and therefore, non-zero values. After successful pivoting, there should be no zero values in vector **d**. Therefore, it will have full length in any case, and does not need to be accompanied by any index vector. Visual example of the transformation of the matrix $\boldsymbol{M}$ to MPSLS sparse matrix ordering technique follows:

$$\boldsymbol{M} = \begin{bmatrix} d & 0 & 1 & 3 \\ 0 & d & 2 & 4 \\ 1 & 2 & d & 5 \\ 3 & 4 & 5 & d \end{bmatrix}, \tag{4.1}$$

where vector decomposition of matrix (4.1) will result in set of value vectors **u**, **l** and **d**:

$$\begin{array}{lllllll} \mathbf{u} = & [0 & 1 & 2 & 3 & 4 & 5], \\ \mathbf{l} = & [0 & 1 & 2 & 3 & 4 & 5], \\ \mathbf{d} = & [d & d & d & d], & & \end{array} \tag{4.2}$$

and set of index vectors $\mathbf{i}_u$ and $\mathbf{i}_u$:

$$\begin{array}{lllllll} \mathbf{i}_u = & [0 & 1 & 2 & 3 & 4 & 5], \\ \mathbf{i}_l = & [0 & 1 & 2 & 3 & 4 & 5]. \end{array} \tag{4.3}$$

To be able actively and fast convert column and row indexing to MPLS, and back it has been developed conversion algorithms.

4.3.1 Forward Conversion of MPLS

The forward conversion algorithm first checks whether a value is on a diagonal of the matrix. If so, the conversion is easy, and value is inserted into the value vector **d**. Otherwise, it must be converted by the following algorithms:

$$\begin{array}{lll} d = & x & \text{for } x = y, \\ i_u = & y(y+1)/2 + x & \text{for } x < y, \\ i_l = & x(x+1)/2 + y & \text{for } x > y, \end{array} \tag{4.4}$$

where x and y are zero based index pairs of original matrix.

The value vector values pair with index vector values and they together can be in any position of vector. It is also very convenient in the cases when new values are added to the matrix. This principle allows resizing matrix without affecting indexation values. It is important to point out that presented algorithms (FPSS and MPLS) have zero-based indexation (indexation vectors start with 0). The complete definition of the forward and backward conversion algorithms for MPLS follows.

4.3.2 Backward Conversion of MPLS

The backward conversion of MPLS requires a little more operations than the forward one. To obtain original matrix indexation values x and y the values from vectors $\mathbf{i}_l$ and $\mathbf{i}_u$ must be converted to intermediate indexes m and n.

$$\begin{aligned} n &= \text{isqrt}(z), \\ m &= z - ((n+1)n/2), \end{aligned} \tag{4.5}$$

where $z \in \{\mathbf{i}_u; \mathbf{i}_i\}$. It should be noted a usage of the Integer Square Root function (ISQRT). In number theory, the ISQRT of a positive integer n is the positive integer m, which is the greatest integer less than or equal to the square root. This function is a standard part of every mathematical library of common programming languages:

The need of this intermediate step arises from the fact that this conversion is a minimization of quadratic form on integer-valued domain. This is usually very problematic and ISQRT together with special constrains of resulting domain helps effectively solve this situation. After obtaining n and m variables, it can decode correct x and y values simply from index vector $\mathbf{i}_u$ as

$$[x, y] = [n - 1, n + m], \tag{4.6}$$

and from index vector $\mathbf{i}_l$ as

$$[x, y] = [n + m, n - 1]. \tag{4.7}$$

The vector $\mathbf{d}$ is sorted in same order as values on diagonal of original matrix thus position of values equal to their indexes. Conversion from $\mathbf{d}$ is simple assignment:

$$[x, y] = [i_d, i_d]. \tag{4.8}$$

4.4 Full Profile Skyline Storage

The idea of FPSS based on the convenient ordering of matrix entries according to operations performed during LU factorization. Therefore, values in vector decomposed from the original sparse matrix are ordered in the same way as LU factorization is performed. It also reduces the number of index vectors to only one. It is the most important advantage of this storage and must be pointed out that only one key index can obtain any value in the matrix. Another benefit of presented storage is the possibility of matrix division to parts with equal memory sizes. Those parts can be stored in different places in memory. It is very convenient when simulation needs to deal with more than 1×10^6 entries.

FPSS index vector is composed directly from position of values in sparse matrix. For illustration lets present in on a small example. The values in matrix $\boldsymbol{M}$ represent in this situation special indexing order of FPSS:

$$\boldsymbol{M} = \begin{bmatrix} 0 & 2 & 6 & 12 \\ 1 & 3 & 7 & 13 \\ 4 & 5 & 8 & 14 \\ 9 & 10 & 11 & 15 \end{bmatrix}. \tag{4.9}$$

Then its decomposition to indexation and value vectors will be trivial:

$$\begin{aligned} \mathbf{v} &= [0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad \dots \quad 15], \\ \mathbf{i} &= [0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad \dots \quad 15]. \end{aligned} \tag{4.10}$$

FPSS is based on row, column indexation. Values in the matrix are indexed from left top corner along diagonal in row-column order to right bottom. The special way of ordering allowed to develop the algorithm for very efficient conversion of values from standard $[x, y]$ (two-dimensional) way to one dimensional $[z]$. Definition of developed FPSS conversion algorithms follows.

4.4.1 Forward conversion of FPSS

The forward conversion of FPSS is trivial and therefore can be performed very fast. Only difficult part is an evaluation of "if" condition. However, it can also be easily solved by the use of templates, functions or even better functionals:

$$f(x,y) = \begin{cases} x + y^2 & \text{for } x > y, \\ y(y+1) + x & \text{for } x < y, \\ (x+1)^2 - 1 & \text{for } x = y. \end{cases} \tag{4.11}$$

4.4.2 Backward conversion of FPSS

The backward conversion as well as the one in MPLS still requires usage of ISQRT function. Nevertheless, the algorithm is still extremely simple and can be implemented in the very efficient way:

$$\begin{aligned} n &= \text{isqrt}(z), \\ d &= n^2, \\ m &= z - d, \end{aligned} \tag{4.12}$$

and

$$[x,y] = \begin{cases} [n, n] & \text{for } n/2 = m, \\ [n, m] & \text{for } n > m, \\ [m - n, n] & \text{for } n < m. \end{cases} \tag{4.13}$$

As it has been told, FPSS allows storing values in different but equally significant memory places. It could be performed for example by a very simple algorithm. Let's say that there is a matrix with 16 values. Then slice every four values to different memory places (files, databases ...). The FPSS algorithm first computes index value z and then by simple modulo division, it will obtain right memory location index.

Number of Matrix Entries	Capability of storages		
	Array	MPLS	FPSS
100 000	yes	yes	yes
1 000 000	yes	yes	0.08*
10 000 000	no	no	3.62*
100 000 000	no	no	11.4*

Table 4.1: Capabilities of different storage systems

yes Storage is capable of holding this number of entries.

no Storage is not capable of holding this number of entries.

***** Storage is capable of holding this number of entries. It contains the total time in seconds to iterate through all values stored in the storage.

Matrix Size	Total Time (sec.)			
	Matlab	GSL LU	Sparse CLU	LISP
50×50	-	0.33	0.004	0.43
100×100	0.01	0.43	0.027	12.73
200×200	0.02	0.96	0.21	-
500×500	0.20	3.20	2.42	-
$1,000 \times 1,000$	1.7	20.2	19.25	-

Table 4.2: Computation performance of sparse matrix LU factorization

4.5 Enhanced Full Profile Skyline Storage

An optimization of the original algorithm can be done by removing one unnecessary operation allowing it to be implemented by only one if/else condition. It leads to my newly developed Enhanced Full Profile Skyline Storage (EFPSS) system definition, where storage index value can be obtained for matrix value entry at row position y and column x:

$$\begin{array}{lccccccccccc} \boldsymbol{x} = & [0 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & \ldots & 4], \\ \boldsymbol{y} = & [0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \ldots & 4], \\ \boldsymbol{i} = & [0 & 2 & 6 & 12 & 20 & 1 & 3 & 7 & \ldots & 24]. \end{array} \tag{4.14}$$

4.5.1 Forward conversion of EFPSS

Mathematical definition of enhanced forward conversion results into definition:

$$f(x,y) = \begin{cases} x + y^2 & \text{for } x < y, \\ x(x+1) + y & \text{for } x >= y, \end{cases} \tag{4.15}$$

where only one condition is needed in the algorithm implementation.

4.5.2 Backward conversion of EFPSS

Backward conversion of index i can be also reduced removing one condition that was redundant in original definition:

$$\begin{aligned} n &= \text{isqrt}(i), \\ d &= n^2, \\ m &= i - d, \end{aligned} \tag{4.16}$$

where, "isqrt" again stands for Integer Square Root Function, and

$$[x,y] = \begin{cases} [n,m] & \text{for } n >= m, \\ [m-n,n] & \text{for } n < m. \end{cases} \tag{4.17}$$

The main use of EFPSS algorithm is indexation algorithm of huge sparse matrices generated by functional λ-calculus device models or special n-order tensors [CD15]. However, regarding implementation, speed and memory consumption, it can become part of any other sparse matrix algorithm. Base on mathematical definition, its functional implementation in language LISP can be the following:

```
(DEFUN GENERATE-FPSS (DIMENSION)
  (LOOP FOR ROW IN
    (INDEX-MATRIX DIMENSION) COLLECT
```

el.	SMIT		EFPSS	
10^4	integer	16 x 2	integer	16 x 1
10^6	integer	16 x 2	long	32 x 1
10^8	long	32 x 2	long	32 x 1
10^{10}	long	32 x 2	long long	64 x 1
10^{12}	long	32 x 2	long long	64 x 1
10^{14}	long	32 x 2	long long	64 x 1

Table 4.3: Comparison of standard indexing and EFPSS

```
    (INDEX-MATRIX-ROW ROW)))

(DEFUN INDEX-MATRIX (DIMENSION)
  (LOOP FOR Y FROM 0 TO DIMENSION COLLECT
  (LOOP FOR X FROM 0 TO DIMENSION COLLECT
    (LIST X Y ))))

(DEFUN INDEX-MATRIX-ROW (ROW)
  (LOOP FOR INDEX IN ROW COLLECT
    (INDEX-MATRIX-CELL
      (FIRST INDEX) (SECOND INDEX))))

(DEFUN INDEX-MATRIX-CELL (X Y)
  (COND
    ((< X Y) (+ X (* Y Y)))
    (T (+ X Y (* X X)))))
```

4.6 Optimization Techniques

4.6.1 Adaptive Indexation Numeric Type

The standard method for a sparse matrix manipulation uses two indexing values for assignment. These values are usually defined by an integer or long variable type. It is usually enough even for huge arrays with the number of elements more than 10^{14}. The problem occurs when EFPSS or similar index transformation method is used. An idea of EFPSS is based on a composition of two index variables into one. To ensure that this index key is unique, it must be the same or higher than the power of highest index value in the set:

$$z \in (0, \max([x, y])^2). \tag{4.18}$$

In Table 4.3, it is presented increase of required memory for index types. Left column is the number of elements in the array; center column represents standard sparse matrix indexing technique (SMIT) and right column EFPSS. It is clearly visible that the use of highest possible container type for indexing (long-long) would be universal in all situations but it leads to a waste of memory space during a simulation of smaller arrays. It brought us to an idea of adaptive indexing technique. The idea is based on adaptively growing character chains used as sparse matrix indexation keys. In a same manner, as a matrix grows the size of containers holding its indexation keys can grow too.

The several sparse matrices were generated to support the idea using a random distribution of non-zero values similar to the distribution that produces standard MNA. For each non-zero

el.	2^{16}	2^{24}	2^{32}	2^{40}	2^{48}
10^4	500	0	0	0	0
10^6	355	4645	0	0	0
10^8	45	8419	41536	0	0
10^{10}	9	1985	497754	252	0
10^{12}	66	19242	4978335	2357	0
10^{14}	755	193896	49795933	3027	6389

Table 4.4: Distribution of non-zero value types

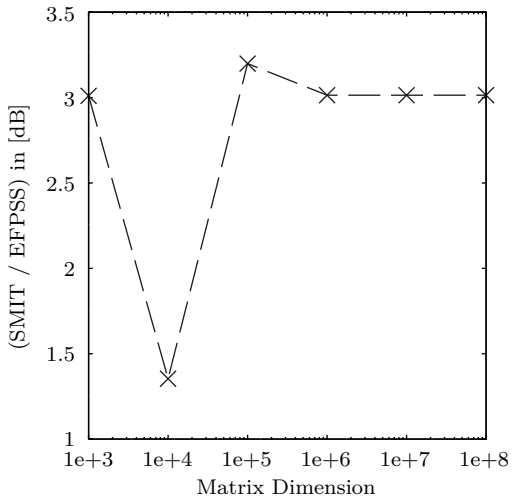


Figure 4.1: Difference between SMIT and EFPPS.

value in those matrices, it was computed EFPSS indexation key and counted its sums grouped by size of variable container type (16b, 24b, 32b, 40b, 48b). This comparison can be found in Table 4.4 where left column stands for number of matrix elements, top row stands for variable container type sizes and in each column, it is a sum of indexing keys generated by EFPSS that can be represented by this particular container type. The results prove that small arrays with dimension up to 100 does not need to use different indexation type than a standard 16b integer. However, for huge matrices with dimension more than 10^5 there is a full variety of different types. It is obvious that when adaptive memory container is embedded into a process of value indexing, it can save much memory during a program run. It is most visible in a matrix with dimension 10^7 where the container with size 2^{40} is needed only for 6 thousand values, but the container for 2^{32} holds almost 50 million values. Comparison in MB between standard memory management with one static container SMIT, FPSS, and EFPSS with sizeable container type is in Table 4.5.

4.6.2 Compressed Indexation

In this section as a huge matrix system is denoted a system that needs to be indexed with numeric types of long. It happens when standard 32b unsigned integer numeric type for index values is not big enough to index all values in a matrix. It should be pointed out that even

el.	SMIT	FPSS	EFPSS
10^4	0.001907348633	0.001907348633	0.0009536743164
10^6	0.01907348633	0.01907348633	0.01396656036
10^8	0.3814697266	0.1907348633	0.1826200485
10^{10}	3.814697266	1.908309937	1.905678749
10^{12}	38.14697266	19.08247757	19.05725765
10^{14}	381.4697266	190.7707825	190.5635824

Table 4.5: Memory consumption in MB of matrix indexes

many present day popular compilers use as a standard for long-long only 64b (C99 and G95 standards). Given huge matrix system tends to produce very long indexation numbers that are difficult to identify for the user. It does not seem to be important as for instance computation performance but from readability and value assignment. It proved to be necessary to develop some human-readable automatic notation matrice index values. Of course, one can always manually label every value, but an automatic system is needed especially in the situation when generated numbers of indexes go over several orders as in (4.19):

$$\boldsymbol{z} = \begin{bmatrix} 429967297 \\ 45645876521 \\ 249767106568 \end{bmatrix}. \tag{4.19}$$

To get rid of this long numeric strings, it was decided to adapt logic that is successfully used in software versioning repository systems and in hashing in general. Instead of numeric values each index from EFPSS using a specially defined algorithm adapted from base64 dictionary (A-Za-z0-9_-) for generation of alpha-numeric guid (hash). The main difference from standard base64 is that it is a just simple conversion from numeric to alphanumeric character strings and not cryptographic conversion. An advantage of using this small array of symbols (only 64) and not full ASCII is that all of those characters can be displayed onscreen and can be typed on the keyboard. Mentioned sparse matrix indexation values then becomes:

$$\boldsymbol{z} = \begin{bmatrix} 429967297 & => & \text{YnLeB} \\ 45645876521 & => & \text{pfsLzp} \\ 249767106568 & => & \text{CnmQ5fI} \end{bmatrix}. \tag{4.20}$$

Implementation of this modified Base64 conversion in functional programming language follows:

```
(DEFUN RETURN-CHARACTER (POS)
      (CHAR *BASE64URL-ALPHABET* POS))

(DEFUN BASE64-ENCODE-RAW (NUM)
  (MULTIPLE-VALUE-BIND (X Y)
    (FLOOR NUM 64)
    (APPEND
      (WHEN (> X 0)
        (BASE64-ENCODE-RAW (- X 1)))
      (LIST (RETURN-CHARACTER Y)))))
```

4.7 Chapter Summary

A capability of different storage systems to hold a large amount of data is shown in Table 4.1. The first column represents the number of matrix entries that go from 1×10^5 to 1×10^8 entries. Three following columns show three different matrix storage systems. First one is an array that is a simple array allocated in memory. It represents biggest allocable continual space on disk. MPLS and FPSS are a state of the art matrix storage systems. The special algorithm for an automatic generation of circuits with random devices and their parameters was created for the purpose of this simulation. The main intention of the algorithm was a comparison of computation speed between sparse matrix storage and full matrix. The algorithm assumes mesh of random four pole devices connected to each other. Every node in a circuit connects maximally two devices. This definition allows us to create an electronics network of the arbitrary size. RHS vector and device parameters are randomly generated. In Table 5.2 is compared the performance of different LU solvers. The first column in the table represents a size of tested matrix. The second one holds values from Matlab solver. The third one represents sparse matrix storage accompanied with GNU GSL library. The fourth one is LU solver with FPSS, and the last one shows results from the LISP sparse matrix implementation. The last column was included only to illustrate the difference between LU implementation in C (Fortran) code and LISP. The high sparsity of simulation matrices of an electronics circuit is a serious problem. Not only that zero entries of a matrix but they produce additional operations that do not need to be performed. MPLS is comparable with a normal array, but in contrast to a normal array, it can handle sparse matrix indexation. FPSS reduces the number of indexation vectors and is capable of holding a huge amount of data. Implementation of proposed sparse matrix storages to LU solver regarding performance proved to be comparable with GNU GSL LU solver. Together with increasing hardware performance, grows higher requirements on simulation and computation capabilities. Presented modified sparse matrix indexing technique called EFPSS that can together with adaptive indexation types save more than 50% of memory required for sparse matrix indexation. The comparisons of memory consumptions between standard indexation using two index vectors: FPSS algorithm using one vector, and enhanced algorithm EFPSS using automatically sizable indexation types can be found in Table 4.5. Improvement in memory consumption is also clearly visible from the graphical representation of a difference between normal indexing and EFPSS that can be found in Fig. 4.1.

4.7.1 Research Contributions Summary

- Two new sparse matrix indexation procedures Modified Profile-In Skyline Storage (MPLS) and Full Profile Skyline Storage (FPSS) developed specially for usage in circuit simulators.
- New enhanced matrix indexation technique Enhanced Full Profile Skyline Storage (EFPSS) developed for general usage.
- Developed techniques can save up to 50% of required memory in comparison to standard methods.
- Adaptive indexation numeric type and Compressed Indexation for better sparse matrix management and handling.

Chapter 5
CLASP

The increasing significance of functional languages could be observed in programming and software development. Many new tools and development frameworks starting to provide in standard structural and object oriented environment also the functional style of programming or even turning all programming paradigm to functional style. In this section, an unusual and efficient usage of functional programming language Common LISP as simulation program (CLASP) for electronic circuits is proposed. The principle of the automatic self-modifying program has enabled complete freedom in a definition of methods for the optimized solution of any problem and speeding up the entire process of simulation. A new approach to program structure in electronic circuit simulator CLASP is described. The definition of simple electronic devices as resistor, voltage source, diode, and MOS transistor are given all together with the description of their memory management in program CLASP. Other circuit elements can be easily defined in the same way. Simulation methods for electronic circuits as linear and nonlinear direct current analysis (DC) are suggested. The performance of simulator is demonstrated with comparison with two standard solvers (an original and the standard GNU GSL).

5.1 Common LISP

The LISt Processing language (LISP) was designed by John McCarthy from Massachusetts Institute of Technology (MIT) already in 1958 as a new symbolic data processing language [McC65]. However, most attention was given to LISP during the artificial intelligence (AI) boom of the 1980s, when LISP became a tool for solving hard problems such as automated theorem proving, computer vision or artificial intelligence simulation. Also, Cold War helped as the Pentagon supported projects for large-scale battlefield simulations, automated planning, and natural language interfaces. In 1981, the standardizing process began of a new language called Common LISP (CL) that combined the best features from the existing LISP dialects. It resulted in granting of standard by the American National Standards Institute (ANSI) in 1996 and It could be said, that since then, there was almost no change [JLW15]. LISP language is still taken as an experimental programming language that has no practical use in professional development. From the recent publications can be pointed out [Sei06] that tries to popularize LISP by implementation examples of real problems.

5.1.1 Basic principle

CL is an expression-oriented language. It reads expressions, evaluates them in accordance with the rules of CL, and prints the result. That endless cycle of reading, evaluating and printing is called read-eval-print loop (REPL). It must be pointed out that unlike most other

languages, no distinction is made between “expressions” and “statements”; program code and data are written in a form of symbolic expressions (SEXPS), which makes the syntax of language extremely regular.

5.1.2 Symbolic expressions

CL SEXPS are composed of forms. The most common CL form is function application. For instance the application of mathematical notation $f_1(x_1, x_2, \ldots, x_n)$ is defined by CL syntax as

```
(F X1 X2 ... XN).
```

The use of parentheses is very typical for CL. Anything inside parentheses is treated as a list. CL evaluates lists by picking the first element as the name of a function and the rest of the elements as arguments to the function. As in other programming languages CL evaluates functions in applicative order, which means that all the argument forms are evaluated before the function is invoked. When CL evaluates expression:

```
(+ 1 (* 2 3))
```

CL reader evaluates list of three elements: symbol "+" which refers to function for addition, symbol 1 which is self-evaluating form (evaluates itself) and nested list:

```
(* 2 3)
```

which is evaluated before it is passed as an argument to addition function.

5.1.3 Syntax

The syntax of CL is case-insensitive. It means that it does not matter whether we write (factorial n) or (FACTORIAL N). But to better recognize the mathematical formulas, the capital letter notation will be used for CL.

Variable assignments (and creating new variables) can be done with the SETQ function. The code below creates a variable called VAR and stores a list in it:

```
(SETQ VAR (QUOTE (HELLO WORLD !!)))
```

The QUOTE function in this example is used to designate that something in brackets is not a SEXPS but a list, which can itself be an argument of the function.

5.1.4 Functions

Functions in CL are represented internally as distinct function objects. For their definition DEFUN macro is used.

```
(DEFUN FUNCTION-NAME (ARGUMENTS...)
  "Function␣documentation␣string␣..."
  BODY...)
```

DEFUN macro defines a new function named FUNCTION-NAME in the global environment. It can be used to define a new function, to install a corrected version of an incorrect definition, to redefine an already-defined function, or to redefine a macro as a function. The example of the use of DEFUN macro is shown by implementation of factorial $n! = \prod_{k=1}^{n} k$ definition:

```
(DEFUN FACTORIAL(N)
  (COND ((EQUAL N 0) 1)
   (T (* N (FACTORIAL (- N 1))))))
```

The principle of previous CL code incorporates the use of recursion for evaluation of product over k. The condition defined by macro COND stands for recursion stopping criteria. The symbol "T" in code stands for condition branch called whenever none of the conditions is met. The FACTORIAL function is recursively called until the condition $(N = 0)$ is met, then recursion calls finish and last function returns number 1.

5.1.5 Functionals

One major aspect of CL is its ability of general abstraction by "functionals". The term "functional" denotes special function that has one or more functions as arguments, or it returns a function as a result. For instance in CL it is possible to pass a function as an argument to another function. This function, denotable as a "general functional", can returns another function with redefined behavior. The definition of such general functional follows:

```
(DEFUN FUNGEN (F X)
  (FUNCTION
    (LAMBDA (Y)
      (FUNCALL F X Y))))
```

This example defines a new contemporary function (so called LAMBDA function) with one argument Y. The body of this function is composed by FUNCALL function:

```
(FUNCALL FUNCTION &REST ARGUMENTS...)
```

FUNCALL is special function which evaluates its first argument as a function call. The following arguments are evaluated in the standard way. It must be pointed out that in CL the names of functions are represented as a symbols. The symbol "+" shall be quoted by function QUOTE or by its syntactic abbreviation to indicate that it is symbol and not a variable. When FUNGEN is evaluated with following arguments:

```
(FUNGEN (QUOTE +) 1)
```

another function is returned as a result. To see what new function actually does, it must be used as an argument of another FUNCALL function. Then the definition of

```
(FUNCALL (FUNGEN '+ 1) 2)
```

results in number 3. The definition of original function was a little modified to show capabilities of CL syntactic abbreviations.

5.1.6 Dynamic variable creation

Due to CL unique SEXPS mechanism it is also possible to dynamically add new variables in runtime of a program. When the CL reader encounters a symbol, it reads all characters of the name. Then it "hashes" those characters to find an index in a table called OBARRAY. If a symbol with the desired name is found, the reader uses that symbol. If the OBARRAY does not contain a symbol with that name, the reader makes a new symbol and adds it to the OBARRAY. Finding or adding a symbol with certain name is called "interning". Dynamical adding of a new variables in CL could be implemented in several ways. The most straightforward is the use of the function INTERN:

```
(INTERN STRING &OPTIONAL PACKAGE)
```

It accepts a string as a name of new variable and returns the interned symbol whose name is given by STRING argument. If there is no such symbol in the OBARRAY defined by argument PACKAGE. INTERN creates a new one, adds it to the OBARRAY, and returns it. If the PACKAGE argument is omitted, global OBARRAY is used.

5.1.7 Further reading

Major advantages of CL have been lightly touched to make clearer idea of key concept of using CL as simulation program. It was by no means all that CL can offer. For more complete description about CL capabilities see the references [McC65, Sei06].

5.2 CLASP

The acronym CLASP is an abbreviation of initial letters of phrase Common LISP as Simulation Program. It should be noted that CLASP is not conceptually full program. It is far better to understand it as a sort of CL source code with a capability of run-time self-redefinition. This source code definition will be referred to as the program CLASP in this paper.

5.3 Modified Nodal Formulation

To be able to model any electronic device for computer simulation, it needs to formulate its characteristic equations according to Kirchhoff's circuit laws (KCL). The method is based on the observation that sum of currents at any node is zero (KCL). At any node, there are two sets of currents, those flowing through the passive components and those due to the independent current sources. KCL at each node is defined as

$$\sum \begin{pmatrix} \text{currents leaving the node} \\ \text{through components} \end{pmatrix} = \sum \begin{pmatrix} \text{currents entering the node} \\ \text{from independent sources} \end{pmatrix}.$$

For computer formulation, an element-by-element approach is preferable, as the equations are set up in a single scan of the element list. These equations are construction blocks for a design of modified nodal formulation (MNF)[HRB75a] by inspection.

In CLASP the original inspection method was extended introducing new matrices and vectors to circuit system equations. In CLASP, standard representation of MNF is divided into four matrices, denoted as matrix $\mathbf{G}$, $\mathbf{E}$, $\mathbf{Z}$, and $\mathbf{D}$, two right-hand-side (RHS) vectors $\mathbf{r}_n$, $\mathbf{r}_q$ and one equation vector $\mathbf{e}_q$.

All linear and frequency independent elements of models are stored in the matrix $\mathbf{G}$. The values associated with frequency are stored in both matrix $\mathbf{E}$ and $\mathbf{Z}$ where matrix $\mathbf{E}$ serves as unit matrix for introducing new frequency dependent values, and $\mathbf{Z}$ matrix holds their coefficients.

In the case of nonlinear model (independently whether it is frequency dependent or not) its nonlinear characteristic equations are stored in matrix $\mathbf{D}$ and vector $\mathbf{e}_q$. $\mathbf{D}$ matrix serves, together with values from linear matrix $\mathbf{G}$, for generation of Jacobian matrix. It holds derivatives of nonlinear equations describing a device on its position in the matrix. It is possible due to CLASP capability of dynamical variables assignment. The remaining equations describing the device model are stored in equation vector $\mathbf{e}_q$. Right-hand side vectors $\mathbf{r}_n$, $\mathbf{r}_q$ hold source values and source describing equations respectively.

5.4 Circuit Description

The circuit to be analyzed is described as a set of CLASP functions. The circuit topology, element values, and type of simulation are defined by these functions. Notation of circuit description is partly inspired by original SPICE notation but follows the rules of CLASP.

There are two mandatory things in every circuit definition. Each circuit has to have a ground node, and the ground node shall be named “0” (zero).

```
([device] "[unique name]" [positive node] [negative node] [value])
```

It defines resistor device with name "R1" and connected to nodes 1 and 0 (ground). The resistance value of the resistor will be 100kΩ. Implementation of scale factors (μ, m, k, G,M...), on capabilities of CLASP, proved to be redundant. Example of definition of complete resistor circuit follows

```
(EF "E1" 1 0
  #'(LAMBDA ()
    (* (SIN *TIME*))))
(R "R1" 1 0 500)
(R "R2" 1 2 500)
(D "D1" 2 0)
(DC)
```

The previous CLASP code defines the voltage source device with name E1 at nodes 1 and 0 (ground). The last parameter is the definition of voltage source function. Whenever global variable *TIME* changes the value of the voltage source changes as well. It also defines several devices as resistors and diode. The last line of circuit notation invokes DC analysis of the circuit.

5.5 Circuit Matrix and Sparsity

To demonstrate sparse matrix problematic and increasing the size of a circuit matrix, here is a small example of the voltage source and its definition of circuit equations. Constitutive equations of voltage source have the following form

$$\begin{aligned} V_p - V_n &= E, \\ I_p &= I, \\ I_n &= -I, \end{aligned} \tag{5.1}$$

The voltage source currents I_p and I_n have to be incorporated in constitutive equations together with a new variable I. This operation will increase a size of matrices by adding one row and one column. The resulting modified nodal formulation [HRB75a] of voltage source will be:

$$\begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} \end{array} = \begin{array}{c} RHS \\ \begin{pmatrix} 0 \\ 0 \\ E \end{pmatrix} \end{array}.$$

As it could be seen, matrix dimension was increased by adding new row and column, but only four nonzero values were added to computation. Therefore, if the matrix has a size of N rows and columns, this operation produces $2N - 5$ additional zero values, which need to be stored in the computer memory.

Increased size of the matrix is indicated by index “+”. The voltage source is very often grounded in a circuit. In that case row and column of particular "grounded node" will not be present in the matrix of the voltage source model.

5.6 Device Modeling

Turning simulation program to functional style gives an opportunity to create an environment, which could change itself during its run [McC65]. Functions could be created and mapped to variables at any time of the program run. It defines a very powerful tool to create special input syntax, which turns, for example, mathematical equations to functions. It is not parsing. The specific property of functional languages is a possibility of language syntax redefinition to special one [LS09]. It allows, for example, in electronics simulation to specify arbitrary complex device model just by definition by its mathematical circuit equations. The implementation of a diode model in the functional language Common LISP (CL) is illustrated to clear this statement. Let's start with well known Shockley's equation of a diode to LISP code:

$$I_\mathrm{D} = I_\mathrm{S}\left(\mathrm{e}^{V_\mathrm{D}/(nV_\mathrm{T})} - 1\right), \tag{5.2}$$

where I_D is diode current, I_S is saturation current, $V_\mathrm{D} = V_\mathrm{p} - V_\mathrm{n}$ is voltage across the diode, V_T is thermal voltage, and n is emission coefficient. Definition of this mathematical equation in functional language Common LISP will look as

```
(DEFUN DIODE-CURRENT (VP VN)
  #'(LAMBDA ()
    (* 10e-12
      (- (EXP
        (/ (- (EVAL VP) (EVAL VN))
          26e-3))
          1))))
```

where $I_\mathrm{S} = 10^{-12}$ A, $V_\mathrm{T} = 26 \times 10^{-3}$ V and $n = 1$, e.g.

This function (DIODE-CURRENT) with parameters (VP VN) returns another function (it is functional), defined by LAMBDA operator, which computes current I_D through the diode. It could be mentioned that it is a trivial redefinition of a mathematical equation in LISP code. It is an introduction for real magic which comes with a mapping function. Following (very self-explanatory) LISP code is complete mapping function for previously defined Zener diode.

```
(defmethod map-device
 ((d class-diode)
  (m class-matrix-system))

 (let ((v+ (make-var-node 'v (node+ d)))
       (v- (make-var-node 'v (node- d)))
       (i  (make-var-name 'i (name  d))))

         (set-g-value m v+ i #'+  1)
         (set-g-value m v- i #'+ -1)
         (set-g-value m i  i #'+ -1)

         (set-equations-value m i
           (diode-current d v+ v-))

         (set-d-value m i  v+
           (diode-current-dva d v+ v-))
         (set-d-value m i  v-
           (diode-current-dvc d v+ v-))))
```

One of the main advantages of functionals is that they can be handled in the same way as values. They can be dynamically created with different parameters, stored in variable or array. It allows to them to be used as circuit device templates and so significantly

enhance device redefinition speed. There is no more need to incorporate device definition into simulation algorithms. Simulation program should be able to assign defined device functionals automatically to correct positions in simulation chain and invoke them in the right order and time when they are needed.

5.6.1 Resistor

The simple model for an ideal resistor in Figure 5.1 can be defined in CLASP in two equivalent ways. The constitutive equations of the resistor device can be arranged by using impedance or admittance description of the device. It is better, whenever it is possible, to use admittance description for device model because it at least partially helps to reduce sparsity of resulting system.

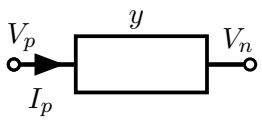


Figure 5.1: Resistor graphical symbol

The node voltage at a positive terminal of the model is denoted as V_p and the negative as V_n. To unify the notation, the currents at terminals V_p or V_n are considered to be positive; thus the current passing through the resistor from the positive terminal to negative terminal is $I_p = I$ and for opposite direction $I_n = -I$. The constitutive equation for positive and negative current direction is then

$$\begin{aligned} I_p &= y(V_p - V_n), \\ I_n &= -y(V_p - V_n), \end{aligned} \tag{5.3}$$

where y denotes admittance of the resistor. The ideal resistor is linear and frequency independent device, all its characteristic values shall be added to values in matrix $\mathbf{G}$. The sub-matrix block denoting admittance model has the following form

$$\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \end{array} \begin{array}{c} \begin{array}{cc} V_p & V_n \end{array} \\ \begin{pmatrix} y & -y \\ -y & y \end{pmatrix} \end{array}. \tag{5.4}$$

Definition of the resistor by impedance model requires adding a new variable (current I_+) to sub-matrix block. Symbol "+" indicates increased the size of matrix G and z is the impedance of resistor. The sub-matrix block of impedance description has the following formulation

$$\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & -z \end{pmatrix} \end{array}. \tag{5.5}$$

Definition in functional language LISP as is implemented in CLASP simulator follows. The preselecting of dynamic voltage node variables is performed to by external function "make-var-node".

```
(defmethod  map-device ((d class-resistor) (m matrice-system))
  (let ((v+       (make-var-node 'v (node+ d)))
        (v-       (make-var-node 'v (node- d)))
        (g      (value d)))
```

```
(setf (g-number-array v+ v+ m) g)
(setf (g-number-array v+ v- m) (- g))
(setf (g-number-array v- v+ m) (- g))
(setf (g-number-array v- v- m) g)))
```

It should be pointed out that this definition is mapping conductivity of resistor device. It is far better approach because then simulator does not need to handle possible Zero Division exception in this part of the program.

5.6.2 Voltage Source

The graphical symbol of voltage source model is shown in Figure 5.2. The voltage nodes are denoted as V_n and V_p, the voltage value as symbol E and the currents at terminal nodes as I_p and I_n.

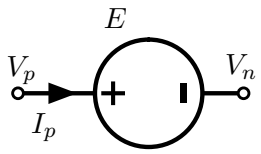


Figure 5.2: Voltage source symbol

In this case, the currents I_p and I_n have to be incorporated in constitutive equations together with a new variable I. It increases a size of matrices by adding one row and one column. Constitutive equations of voltage source have the following form

$$\begin{aligned} V_p - V_n &= E, \\ I_p &= I, \\ I_n &= -I, \end{aligned} \tag{5.6}$$

from which matrix $\mathbf{G}$ with the source vector $\mathbf{r}_n$ can be formulated as:

$$\begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} \end{array} = \begin{array}{c} RHS \\ \begin{pmatrix} 0 \\ 0 \\ E \end{pmatrix} \end{array}. \tag{5.7}$$

Increased size of the matrix is indicated by index “+”. The voltage source is very often grounded in a circuit. In that case row and column of particular "grounded node" will not be present in the matrix of the voltage source model. Simplicity of definition of voltage source model can be seen from following CLASP device implementation.

```
(defmethod map-device ((d class-source-voltage) (m matrice-system))
  (let ((v+       (make-var-node 'v (node+ d)))
        (v-       (make-var-node 'v (node- d)))
        (i        (make-var-name 'i (name  d)))
        (vol    (voltage d)))
    (setf (g-number-array i v+ m)   1)
    (setf (g-number-array i v- m)  -1)
    (setf (g-number-array v+ i m)   1)
    (setf (g-number-array v- i m)  -1)
    (setf (rhs-number-vector i m)  vol)))
```

This source code “stub” shows also how vaules in RHS vecros are set and their relation to circuit matrix.

5.6.3 Simple Diode

A simple diode model will be defined in this section. The diode is a two-terminal electronic device with nonlinear current-voltage characteristics. The diode symbol is shown in Fig.5.3.

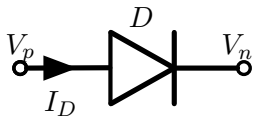


Figure 5.3: Diode symbol

The definition of the simple diode will be given for its simplicity and clearness by Shockley ideal diode equation. The fundamental equation of ideal diode current will become

$$I_\mathrm{D} = I_\mathrm{S}\left(\mathrm{e}^{V_\mathrm{D}/(nV_\mathrm{T})} - 1\right), \tag{5.8}$$

where I_D is diode current, I_S is saturation current, $V_\mathrm{D} = V_\mathrm{p} - V_\mathrm{n}$ is voltage across the diode. V_T is thermal voltage, and n is emission coefficient.

The thermal voltage V_T is well known constant defined by the equation $V_\mathrm{T} = kT/q$, where k is the Boltzmann constant ($1.38 \times 10^{-23}\mathrm{J/K}$), T is absolute temperature of the P-N junction, and q is elementary charge ($1.602 \times 10^{-19}\mathrm{C}$). The value of V_T will be set to 25.85 mV, saturation current $I_\mathrm{S} = 10^{-12}$ A and emission coefficient $n = 1$. In the case, when the diode current is requested as an output variable, the diode constitutive equations will become:

$$\begin{aligned} I_\mathrm{D} &= I_\mathrm{S}\left(\mathrm{e}^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} - 1\right), \\ I_p &= I_\mathrm{D}, \\ I_n &= -I_\mathrm{D}, \end{aligned} \tag{5.9}$$

from which the Jacobian matrix can be written directly as:

$$\mathbf{M}|_\mathbf{x} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ \lambda\mathrm{e}^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & -\lambda\mathrm{e}^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & -1 \end{pmatrix}, \tag{5.10}$$

where $\lambda = I_\mathrm{S}/(nV_\mathrm{T})$, and $\mathbf{x} = [V_\mathrm{p}\, V_\mathrm{n}\, I_\mathrm{D}]^t$.

Putting it all together, diode CLASP matrices can be formulated. As it has been mentioned, to model nonlinear and frequency independent device the use of matrices $\mathbf{G}$, $\mathbf{D}$ and equations vector $\mathbf{e}_\mathrm{q}$ is needed. The resulting formulation is

$$\mathbf{G} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -1 \end{pmatrix}, \quad \mathbf{D} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \lambda\mathrm{e}^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & -\lambda\mathrm{e}^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & 0 \end{pmatrix}, \tag{5.11}$$

$$\mathbf{e}_\mathrm{q} = \begin{pmatrix} 0 & 0 & I_\mathrm{S}\left(\mathrm{e}^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} - 1\right) \end{pmatrix}^\mathrm{T}, \tag{5.12}$$

where T denotes transposition of the vector.

The implementation of the diode model in CLASP will be better illustrated for the reader. The function DIODE-CURRENT expect arguments VP and VN. They are the names of particular node variables to which anode and cathode of the diode are connected. In the mathematical notation, VP and VN correspond to V_p and V_n, respectively.

```
(DEFUN DIODE-CURRENT (VP VN)
  #'(LAMBDA ()
    (* 10e-12
      (- (EXP
        (/ (- (EVAL VP) (EVAL VN))
          26e-3))
          1))))
```

This function returns another function, defined by LAMBDA operator, which computes current I_D through the diode. The second CLASP source code implements derivative of the diode current on node voltage.

```
(DEFUN DIODE-CURRENT-DVP (VP VN)
  #'(LAMBDA ()
    (* (EXP
        (/ (- (EVAL VP) (EVAL VN))
          26e-3))
      (/ 10e-12 26e-3))))
```

It generates functionals that are stored in the matrix $\mathbf{D}$ and in the vector $\mathbf{e}_\mathrm{q}$. When they are invoked, they evaluate themselves according to an actual state of variables VP and VN. These values should be known at the time of simulation because finding the solution of the nonlinear system requires the use of an iteration method such as the Newton-Raphson method. The values VP and VN will usually be given by random prediction or as a result of the previous iteration of an algorithm.

```
(defmethod map-device ((d class-diode) (m matrice-system))
  (let ((v+       (make-var-node 'v (node+ d)))
        (v-       (make-var-node 'v (node- d)))
        (i        (make-var-name 'i (name  d))))
    (setf (g-number-array v+ i m) 1)
    (setf (g-number-array v- i m) -1)
    (setf (g-number-array i i m) -1)
    (setf (nonlinear-equation-vector i m)  (diode-current d v+ v-))
    (setf (differetial-equation-array i v+ m) (diode-current-dva d v+ v-))
    (setf (differetial-equation-array i v- m)   (diode-current-dvc d v+ v-))))
```

5.6.4 Open and Short Circuit

The model of open or short circuit can be easily derived from equations for a simple conductance as it has been done in 5.3. The difference is that current will be taken into account by additional column at this time. Impedance model 5.5 remains unchanged.

$$\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ y & -y & -1 \end{pmatrix} \end{array},$$

The open circuit requires conductance $y = 0$ while for a short circuit impedance shall be $z = 0$.

$$\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & -z \end{pmatrix} \end{array},$$

Coresponding formulation for the open circuit and short circuit in CLASP will then be:

$$\underbrace{\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -1 \end{pmatrix} \end{array},}_{\text{open circuit}} \qquad \underbrace{\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ I_+ \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_+ \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & 0 \end{pmatrix} \end{array},}_{\text{short circuit}}$$

5.6.5 Capacitor

A capacitor is an electronic device capable of accumulating electrical charge Q. Mathematical relation between voltage and electrical charge is

$$Q = C \cdot U, \tag{5.13}$$

where constant C is capacitance. Relation between electrical charge Q on defined capacity C and current $i(t)$ is given by differential equation as

$$i(t) = \frac{dQ}{dt}, \tag{5.14}$$

substituing equation 5.13 to 5.14 from is received

$$i(t) = C \cdot \frac{du(t)}{dt}, \tag{5.15}$$

which indicates that current passing through capacitor equal to time variation of voltage. Although it is generally known statement, it has been presented here to highlight time dependency of capacitor. When capacitor model is incorporated to system equations, it will change the system from linear/non-linear to differential system of the first order. For evaluation of such system numerical integration method is required. When an integration method is applier to get an expression for v_{n+1}, then equation can be reformulated as

$$v_{n+1} = v_n + \frac{\Delta t}{C} I, \tag{5.16}$$

where I is determined by integration method. The ideal model of the capacitor as it is

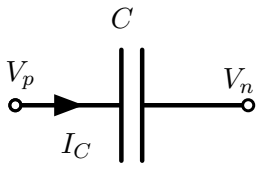


Figure 5.4: Capacitor symbol

implemented in CLASP program originate from three matrices: **G**, **E** and **Z**. The two matrices **G** and **E** help analysis to faster initialize device model during DC analysis. In DC analysis ideal capacitor acts as open circuit, this beaviour is then simulated by combination of matrices **G** and **E**, which produces in-fact open circuit. It is usually a very fast procedure that can help to the simulator estimate for example initial numerical iteration guess. The graphical symbol of the capacitor device is shown in Figure 5.4. In matrix **Z** are values that are tune dependent and will be multiplied with time stepping algorithm. The formulation of linear capacitor model follows:

$$\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ V_q \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_q \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix} \end{array}, \quad \mathbf{E} = \begin{array}{c} \\ I_p \\ I_n \\ V_q \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_q \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \mathbf{Z} = \begin{array}{c} \\ I_p \\ I_n \\ I_q \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_q \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & -1 & -1/C \end{pmatrix} \end{array}.$$

Size of the matrices was increased introducing a new variable I_q to system equation of the circuit (denoted as "q" in CLASP definition).

```
(defmethod map-device ((d class-capacitor) (m matrice-system))
  (let ((n+       (make-var-node  'v (node+ d)))
        (n-       (make-var-node  'v (node- d)))
        (q        (make-var-name  'q (name  d)))
        (c    (value d)))
    (setf (g-number-array q q  m)  -1)
    (setf (e-number-array n+ q  m)  1)
    (setf (e-number-array n- q  m)  -1)
    (setf (z-number-array q n+  m)  c)
    (setf (z-number-array q n-  m)  (- c))))
```

5.6.6 Inductor

An ideal inductor offers no resistance to a constant direct current. The differential equation describes the relationship between the time-varying voltage $v(t)$ across an inductor with inductance L and the time-varying current $i(t)$ passing through it

$$v(t) = L\frac{di(t)}{dt} \tag{5.17}$$

to make this applicable for integration method the equation need to be modified as

$$i_{n+1} = i_{n+1} + \frac{\delta t}{L}V, \tag{5.18}$$

where V is determined by integration method.

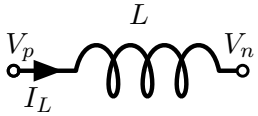


Figure 5.5: Inductor symbol

Again, it need to split values of inductor between three matrices **G**, **E** and **Z** in the same manner as it was presented in the case of capacitor. Again, in this case for DC simulation by combination of matrices **G** and **E** model become short circuits. In matrix **Z** are then time dependent values that are passed computed with time step produced by numerical integration method. Model formulation is defined:

$$\mathbf{G} = \begin{array}{c} \\ I_p \\ I_n \\ I_L \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_L \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}. \quad \mathbf{E} = \begin{array}{c} \\ I_p \\ I_n \\ I_L \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_L \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & 0 \end{pmatrix} \end{array}. \quad \mathbf{Z} = \begin{array}{c} \\ I_p \\ I_n \\ I_L \end{array} \begin{array}{c} \begin{array}{ccc} V_p & V_n & I_L \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -L \end{pmatrix} \end{array}.$$

```
(defmethod map-device ((d class-inductor) (m matrice-system))
  (let ((n+       (make-var-node 'v (node+ d)))
        (n-       (make-var-node 'v (node- d)))
        (i        (make-var-name 'i (name  d)))
        (L    (value d)))
        (set-g-value m  i n+ #'+  1)
        (set-g-value m  i n- #'+ -1)
        (set-e-value m n+ i  #'+  1)
        (set-e-value m n- i  #'+ -1)
        (set-z-value m i i   #'-  L)))
```

5.6.7 Zener diode

Zener diode is a special kind of diode which allows current to flow in the forward direction same as an ideal diode, but will also permit it to flow in the reverse direction when the voltage is above a certain value known as the breakdown voltage, "Zener voltage." The symbol of the Zener diode is in Figure 5.6.

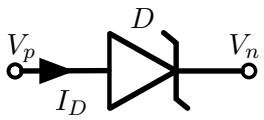


Figure 5.6: Zener diode symbol

The model of Zener diode implemented in CLASP is based on the simple diode model defined by forward and backward current.

$$I_\mathrm{D} = I_\mathrm{S}\left(\mathrm{e}^{V_\mathrm{D}/(nV_\mathrm{T})} - 1\right),$$
$$I_\mathrm{D} = -I_\mathrm{S}\left(\mathrm{e}^{V_\mathrm{D}/(nV_\mathrm{T})} - \mathrm{e}^{-(V_\mathrm{D}+V_B)/(nV_\mathrm{T})}\right).$$

Following example defines basic iplementation without nonlinear defined functions with first order derivations. Complete CLASP code implementation of Zener diode can be found in Appendix 8.

```
(defmethod map-device ((d class-diode-simple-zener) (m matrice-system))
  (let ((v+       (make-var-node 'v (node+ d)))
        (v-       (make-var-node 'v (node- d)))
        (i        (make-var-name 'i (name  d))))
    (set-model-value m 'g-number-array v+ i 1)
    (set-model-value m 'g-number-array v- i -1)
    (set-model-value m 'g-number-array i i -1)
    (set-model-value m 'nonlinear-equation-vector i
       (diode-simple-zener-current d v+ v-))
    (set-model-value m 'differetial-equation-array i v+
       (diode-simple-zener-current-dv+ d v+ v-))
    (set-model-value m 'differetial-equation-array i v-
       (diode-simple-zener-current-dv- d v+ v-))
    (set-model-value m 'initial-number-vector i 0.05d0)))
```

In the example, it is also presented initial simulation device presets. During simulation of an electronic circuit with ideal device models can numerical iteration method iterate diverge, start to oscillate or finish iteration finding the only local minimum. There are many techniques to prevent this from happening. For example in literature are presented various dumping methods. In line with code "(set-model-value m 'initial-number-vector)" is a

presented simplest method to avoid divergence by predefinition of initial numerical estimation in device model.

5.6.8 Thermistor

A thermistor is a type of resistor whose resistance varies significantly with temperature, more so than in standard resistors. The model of CLASP thermistor uses Steinhart-Hart equation.

$$\frac{1}{T} = a + b\,\ln(R) + c\,\ln^3(R) \tag{5.19}$$

where a, b and c are called the Steinhart-Hart parameters, and must be specified for each device. T is the temperature in Kelvin and R is the resistance in ohms. Complete CLASP code implementation of thermistor device can be found in Appendix C.

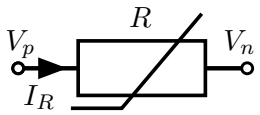


Figure 5.7: Thermistor symbol

```
(defmethod map-device ((d class-thermistor) (m matrice-system))
  (let ((v+        (make-var-node 'v (node+ d)))
        (v-        (make-var-node 'v (node- d)))
        (i         (make-var-name 'i (name  d))))
    (set-g-value m v+ i #'+  1)
    (set-g-value m v- i #'+ -1)
    (set-g-value m i  i #'+ -1)
    (set-equations-value m i (thermistor-current d v+ v-))
    (set-d-value m i  v+  (thermistor-current-dv+ d v+ v-))
    (set-d-value m i  v-  (thermistor-current-dv- d v+ v-))))
```

5.6.9 MOS Transistor

CLASP implementation of very basic (Shichman-Hodges) model of MOS transistor used in SPICE as described in [Vla94]. The drain-source current I_{DS} of an n-channel device is then

$$I_{DS=}\begin{cases} 0 & V_{GS} \le V_{th} \\ \beta\,(V_{GST} - V_{DS}/2)\,V_{DS} & 0 < V_{DS} < V_{GST} \\ \beta/2\,(V_{GST})^2\,(1 + \lambda V_{DS}) & 0 < V_{GST} < V_{DS}, \end{cases} \tag{5.20}$$

$$I_{DS} = \begin{cases} 0 & V_{GS} \le V_{TH} \\ \beta_f V_\omega^2 X_\lambda & 0 < V_\omega \le V_{DS} \\ \beta_f V_{DS}(2V_\omega - V_{DS})X_\lambda & 0 < V_{DS} < V_\omega, \end{cases} \tag{5.21}$$

where each case defines different region of an operation cutoff, saturation, and linear. Variables V_ω and X_λ are subtitutions defined as $V_\omega = V_{GS} - V_{TH}$ and $X_\lambda = 1 + \lambda V_{DS}$. Parameter β_f is substitution for

$$\beta_f = \frac{K}{2}\frac{W}{L_{ef}} \tag{5.22}$$

where $L_{ef} = L - 2L_D$ is effective channel length corrected for the lateral diffusion, L_D, of the drain and source, and

$$V_{TH} = V_{TO} + \gamma(\sqrt{\phi - V_{BS}} - \sqrt{\phi}) \tag{5.23}$$

is the threshold voltage in the presence of back-gate bias, $V_{BS} < 0$. Parametres V_{TO}, K, γ, ϕ and λ are the electric parametres of a MOSFET model, representing the threshold voltage, transconductance factor, bulk threshold parameter, surface potential, and output conductance factor in saturation, respectively. where β stands for "Trans Conductance" defined as $\beta = \frac{\mu\epsilon W}{L_D}$. Parametr W is channel width, L_D is lateral diffusion and V_{GST} replacing $V_{GST} = V_{GS} - V_{TH}$, where V_{TH} is threshold voltage and λ is the channel length modulation factor. In the simulation, it is going to use n-channel devices only with bulk connected to the source. Therefore it can be simplified by putting V_{TH} to a constant value. The variables V_{DS}, V_{GS} depend on circuit definition and current state. At the time of model definition, those variables remain unknown and will be evaluable at the moment of circuit simulation. Complete definition of previously defined equations for a simple model of MOS transistor in functional language LISP follows

```
(defmethod mos_current_drain
  ((mos class-mos) vg vs vd)
  #'(lambda ()
    (let*
      ((v_gs (- (eval vg) (eval vs) ))
       (v_ds (- (eval vd) (eval vs)))
       (v_omega (- v_gs (VTH mos)))
       (v_lambda
         (+ 1 (* (LAMBDA mos) v_ds))))
      (cond
        ((< v_gs (VTH mos)) 0)
        ((and (< 0 v_ds) (< v_ds v_omega))
          (*
            (BETA mos) v_ds v_lambda
            (- (* 2 v_omega) v_ds)))
        ((and
          (< 0 v_omega )
          (< v_omega v_ds))
          (* (BETA mos) v_lambda
            (expt v_omega 2)))))))
```

The method denoted as "mos_current_drain" is a wrapper for inner functional definition.

```
(defmethod mos_current_drain
  ((mos class-mos) vg vs vd)
  #'(lambda ()
```

It is defined without any parameters, but the wrapper function, on its call, will set up all internal variables of the device model to references passed through input argument list. The functional arguments vg, vs, vd are simulation variables and will be set to circuit voltages concerning a location where the device is placed in a circuit. It is done by calling "eval" function during enumeration:

```
(vgs (- (eval vg) (eval vs)))
```

For completeness, it should be pointed out that for each nonlinear model there have to be defined all derivatives for all nonlinear functions and time dependent parameters. They will

be used for generation of Jacobian matrix that is required by Newton-Raphson algorithm. In this case, it has to be the derivative of $\partial i_{DS}/\partial v_{GS}$, $\partial i_{DS}/\partial v_{DS}$ and $\partial i_{DS}/\partial v_{BS}$ that will result in three additional functions wrapping their functionals.

```
(defmethod mos-current-drain-dvgs)
(defmethod mos-current-drain-dvds)
(defmethod mos-current-drain-dvbs)
```

The full definition requires derivation for each nodal voltage direction. It will result in growing of Jacobian matrix during evaluation. More information about the concepts of functional programming can be found in [JLW15]. Comprehensive definition of nonlinear device modeling in Lisp with explanation of further steps can be found in [ČD11] and [CD14a].

5.7 CLASP Simulation Algorithms

One important fact must be pointed out about functional languages. And actually about any other programming language higher than C. No matter, how many optimization algorithms are given to compiler, resulting binary will be in any case slower than C or Fortran compilation. The result can differ only in several instructions, but this will get on significance when it is performed thousand times over. To compete in performance with other simulators, all its computation demanding tasks must delegate to C language, Fortran or even lower language. In CL, it is possible though so-called Foreign Array Interface mechanism. It allows to CL to use, create and call functions and variables from other programming languages, for example in C. All demanding tasks can be passed and evaluated by code written in C language. It proved to be a good approach to use GNU Scientific Library (GSL) [MG09]. It offers many mathematical functions and is very fast. The disadvantage of this solution is that GSL does not support sparse matrix storage. This fact could be very problematic in electronic circuit simulation because high sparsity is a characteristic property of circuit equations.

5.7.1 Linear DC Analysis

CLASP divides DC analysis to linear DC and nonlinear DC analysis. When circuit includes only linear devices, evaluation of analysis is performed with matrices $\mathbf{G}$, and $\mathbf{E}$ together with right-hand-side (RHS) vectors $\mathbf{r}_\mathrm{n}$ and $\mathbf{r}_\mathrm{q}$. The remaining matrices and vectors are not required during linear DC analysis. Matrix $\mathbf{E}$ has two functions. Normally it holds unity values of frequency dependent devices as capacitors and inductors. But during DC analysis matrix $\mathbf{E}$ is added together with matrix $\mathbf{G}$. Due to the smart structure of matrix $\mathbf{E}$ this operation removes time dependent devices. Inductors are replaced with shortcuts and capacitors are disconnected from the circuit. The advantage is that the voltage nodes and their variables, even if they are in a disconnected part of the circuit, stay in the equations. The resulting linear system is defined as

$$(\mathbf{G} + \mathbf{E})\,\mathbf{x} = \mathbf{r}_\mathrm{q}(\mathbf{s}) + \mathbf{r}_\mathrm{n}, \tag{5.24}$$

where vector $\mathbf{x}$ denotes unknown variables as node voltages, currents through devices, electric charges, etc. RHS vectors $\mathbf{r}_\mathrm{n}$ and $\mathbf{r}_\mathrm{q}(\mathbf{s})$ are source vectors, they hold voltage source values and voltage source equations respectively. For instance in a use of time stepping voltage source, the characteristic time dependent equation of voltage source will be added to vector $\mathbf{r}_\mathrm{q}(\mathbf{s})$. Vector $\mathbf{s}$ is a vector of circuit system parameters, which shall be known at any time of simulation. Then the solution of linear system (Equation 5.24) computed by factorization method [Hig02].

LU Factorization Algorithm

By the linear algebra definition, LU factorization is an algorithm that writes a matrix as the product of a lower and upper triangular matrix. Algorithms for triangular factorization are closely related to Gaussian elimination, though the computations might be performed to optimize a number of operations in a different sequence. Let's have a linear system for given unknown vector $\mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \mathbf{y}, \tag{5.25}$$

where $\mathbf{A}$ is a linear matrix and $\mathbf{y}$ is RHS vector. Assume that the matrix $\mathbf{A}$ can be factored as follow

$$\mathbf{L}\mathbf{U} = \mathbf{A}, \tag{5.26}$$

$$\mathbf{L} = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & u_{12} & \dots & u_{1n} \\ & 1 & \dots & u_{2n} \\ & & \ddots & \vdots \\ & & & 1 \end{bmatrix}. \tag{5.27}$$

$\mathbf{L}$ stands for lower triangular and $\mathbf{U}$ for upper triangular. It should be noted that ones are on the main diagonal of $\mathbf{U}$ matrix. This means that the determinant of $\mathbf{A}$ is found as the product of the l_{ii} elements of the matrix $\mathbf{L}$. The linear system from 5.25 can be reformulated to:

$$\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{y}. \tag{5.28}$$

Defining of auxiliary vector $\mathbf{z}$, results into the basic principle of LU factorization:

$$\begin{aligned} \mathbf{L}\mathbf{z} &= \mathbf{y}, \\ \mathbf{U}\mathbf{x} &= \mathbf{z}. \end{aligned} \tag{5.29}$$

Pivoting

LU factorization shall be supplemented by pivoting technique to improve the accuracy of the solution of a system of linear equations. Selection of the pivot may be governed by various rules if accuracy is of prime importance, the pivot should be the element with the largest absolute value, because this approach is not fully compatible with fast modes of LU factorization, especially when sparse factorization algorithm is performed.

LU Factorization in CLASP

The performance of CLASP capability of solving linear equations was tested by the implementation of two solvers. The first implemented algorithm into CLASP for LU factorization was modified Crout algorithm. The original algorithm was modified for the optimized solution of sparse matrices. This solver will be referred to as CLASP LU. The main part of CLASP LU implementation can be found in the Appendix 8. The second solver incorporated into the CLASP was taken from the GNU GSL library. It is based on Gaussian Elimination with partial pivoting described in [GV96]. Due the fact that GNU GSL library is written in C, it works as an external library. Therefore, the GNU GSL functions for LU factorization shall be called by CLASP foreign interface mechanism.

However, a series of tests has proven that the rich and accurate hierarchy of numbers in CLASP slows down the efficiency of evaluation in high-performance commutating tasks. Even with very optimal code, CLASP can not compete on efficiency and memory management of languages such as Fortran or C. Therefore, the decision was adopted to distribute this computationally intensive tasks through CLASP foreign array interface to GNU Scientific Library (GSL). The GSL is a numerical library written in C and C++ language. It is free software under the GNU General Public License and provides a wide range of mathematical routines such as random number generators, special functions, and least-squares fitting. Also, it supports Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK) routines.

Accuracy of the solution in CLASP

Efficient numerical computation in CLASP is a not straightforward thing. The environment provides a rich hierarchy of numbers that is integrated with the rest of the language. Together with the standard number representation (integers, floats, double-floats, etc.) it also offers exact arithmetic such as implicitly created value types (bignums, rational and complex numbers). It gains on importance in the case when precision is more important than performance. It is known that precision of floating point numbers in computer memory is limited [Gol91]. For instance, a common property of floating point numbers caused by round-off error is demonstrated in the following equation:

$$x/10 + y/10 \neq (x + y)/10, \tag{5.30}$$

that could be demonstrated in CLASP invoking following function:

```
(+ 0.1 0.1 0.1 0.1 0.1 0.1 0.1)
```

that results in mathematically incorrect value 0.70000005 (single float). Even it is expected it is entirely an inappropriate form point of view of user experience. Simulation program should be arbitrary precise on user input. Simulation procedure and results are an additional thing that will be discussed further. In Common LISP and also CLASP the effect of round-off error can be resolved to convert floating point numbers to rational numbers that are very simple operation in CLASP:

```
(+ 1/10 1/10 1/10 1/10 1/10 1/10 1/10)
```

At this time the example results in mathematically correct value 7/10. In CLASP all numbers are always automatically converted to the appropriate format. Therefore it does not matter whether mathematical operation includes integers together with double floats and rational numbers. The result will be correctly evaluated with an output precision determined by the lowest precision in computation. It gives to simulation additional self-optimization possibilities. In a case that all device constants are defined as arbitrary functionals or function that evaluates upon a call to given precision it is possible for the simulator to scale dynamically precision based on user settings.

5.7.2 Nonlinear DC Analysis

When circuit includes some nonlinear models of devices such as diodes, a solution of network equations cannot be obtained directly, and equations shall be linearized before. One of the basic methods for solving nonlinear equations implemented in CLASP is Newton-Raphson (NR) method [SM03]. NR method is a method for finding the roots (or zeros) of a real-valued function. It has fast convergence (quadratic close to the solution), but the disadvantage

of NR method is that the Jacobian matrix calculation is needed. A brief description of its implementation in CLASP program follows. Let's start with definition of a system $\mathbf{f}$ with n nonlinear equations

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \tag{5.31}$$

substituting CLASP matrices to this system the formulation changes to this form

$$(\mathbf{G} + \mathbf{E})\mathbf{x} + \mathbf{e}_{\mathrm{q}}(\mathbf{x}) - \mathbf{r}_{\mathrm{q}}(\mathbf{s}) - \mathbf{r}_{\mathrm{n}} = \mathbf{0}. \tag{5.32}$$

To derive NR algorithm lets denote the solution of the system as vector $\mathbf{x}^*$ and vector $\mathbf{x}$ as any different solution. Applying Tailor expansion

$$\begin{aligned}
f_1(x^*) &= f_1(x) + \frac{\partial f_1}{\partial x_1}(x_1^* - x_1) + \frac{\partial f_1}{\partial x_2}(x_2^* - x_2) + \ldots + \frac{\partial f_1}{\partial x_n}(x_n^* - x_n) \\
f_2(x^*) &= f_2(x) + \frac{\partial f_2}{\partial x_1}(x_1^* - x_1) + \frac{\partial f_2}{\partial x_2}(x_2^* - x_2) + \ldots + \frac{\partial f_2}{\partial x_n}(x_n^* - x_n) \\
&\vdots \\
f_n(x^*) &= f_n(x) + \frac{\partial f_n}{\partial x_1}(x_1^* - x_1) + \frac{\partial f_n}{\partial x_2}(x_2^* - x_2) + \ldots + \frac{\partial f_n}{\partial x_n}(x_n^* - x_n)
\end{aligned}$$

and neglecting higher terms in expansion, the system can be formulated in linearized form, which will result, by the use of CLASP matrices, to the following notation:

$$\mathbf{f}(\mathbf{x}^*) \approx (\mathbf{G} + \mathbf{E})\mathbf{x} + \mathbf{e}_{\mathrm{q}}(\mathbf{x}) - \mathbf{r}_{\mathrm{q}}(\mathbf{s}) - \mathbf{r}_{\mathrm{n}} + \mathbf{M}(\triangle\mathbf{x}), \tag{5.33}$$

where $\triangle\mathbf{x} = (\mathbf{x}^* - \mathbf{x})$ and

$$\mathbf{M}|_{\mathbf{x}} = \begin{pmatrix} \frac{\partial D_{1,1}}{\partial x_1} & \frac{\partial D_{1,2}}{\partial x_2} & \cdots & \frac{\partial D_{1,n}}{\partial x_n} \\ \frac{\partial D_{2,1}}{\partial x_1} & \frac{\partial D_{2,2}}{\partial x_2} & \cdots & \frac{\partial D_{2,n}}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial D_{n,1}}{\partial x_1} & \frac{\partial D_{n,2}}{\partial x_2} & \cdots & \frac{\partial D_{n,n}}{\partial x_n} \end{pmatrix} + \mathbf{G} + \mathbf{E}, \tag{5.34}$$

is the Jacobian matrix compiled of matrix $\mathbf{G}$ and $\mathbf{E}$, and evaluated expressions from matrix $\mathbf{D}$. As follows from (5.31) both sides of (5.33) shall be zero. Evidently, when 5.33 is set equal to zero and solved, the result will not be the vector $\mathbf{x}^*$ (because the higher-order terms in Tailor expansion have been neglected) but some new value for $\mathbf{x}$. Using superscripts to indicate iteration sequence, the equation gets this form

$$(\mathbf{G} + \mathbf{E})\mathbf{x}^k + \mathbf{e}_{\mathrm{q}}(\mathbf{x}^k) - \mathbf{r}_{\mathrm{q}}(\mathbf{s}) - \mathbf{r}_{\mathrm{n}} + \mathbf{M}(\mathbf{x}^{k+1} - \mathbf{x}^k) = \mathbf{0}. \tag{5.35}$$

The solution for next iteration $\mathbf{x}^{k+1}$ is usually obtained by LU factorization, it is convenient to rewrites (5.35) to form:

$$\mathbf{M}(\triangle\mathbf{x}^k) = -\mathbf{e}_{\mathrm{q}}(\mathbf{x}^k) - (\mathbf{G} + \mathbf{E})\mathbf{x}^k + \mathbf{r}_{\mathrm{q}}(\mathbf{s}) + \mathbf{r}_{\mathrm{n}}, \tag{5.36}$$

where $\triangle\mathbf{x}^k = \mathbf{x}^{k+1} - \mathbf{x}^k$ and the value for next iteration can be easily obtained from

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \triangle\mathbf{x}^k. \tag{5.37}$$

The NR iterative algorithm always starts with initial estimation of variable vector. The algorithm continues with generating new values until the solution meets the stopping criterion:

$$\left\|\mathbf{f}(\mathbf{x}^k)\right\| - \left\|\mathbf{f}(\mathbf{x}^{k+1})\right\| \leq \epsilon, \tag{5.38}$$

when the iteration ended and solution of the nonlinear system is returned. Otherwise, it continues until the maximum number of iteration loops is exceeded. The value of the parameter ϵ from 5.38 is absolute error and depends on machine precision. It is usually set to the value of the order less than 10^{-7} in single floating point precision and to 5×10^{-15} in double precision. The relative error stopping criteria can be used as well.

Since 5.37 may not result automatically in each step, a modification is sometimes used

$$\mathbf{x}^{k+1} = \mathbf{x}^k + t^k \triangle \mathbf{x}^k. \tag{5.39}$$

The parameter t^k is selected such that previous is satisfied. Usually $0 < t^k \leq 1$.

5.7.3 Transient Analysis

One of the most complex algorithms implemented in CLASP is the Transient Analysis. CLASP uses an approximation method (it implements single-step methods, namely Trapezoidal Rule and Gear method) to compute the defined integral. Those methods are called through FFI from GNU GSL library. The structure of the transient analysis algorithm is as follow:

while Transient Loop **do**
  Formulate Companion Models for Energy Components
  while Newton-Raphson Loop **do**
    Formulate Companion Models for Non-linear Components
    Solve Time Point
  end while
end while

The full definition of the algorithm with the support of GNU GSL can be found in Appendix E.

5.8 CLASP Experimental Algorithms

5.8.1 Evolutionary Newton-Raphson

To enhance the accuracy of the solution of nonlinear equations, new algorithms have been implemented to CLASP. Evolutionary Newton-Raphson (NR) performs a numerical evaluation of the system with nonlinear equations for several different variable vectors (called individuals). Several random vectors are generated at the beginning of the NR instead of making one random prediction for unknown variables. They represent the first population in zero generation. Each vector represents one individual. The standard NR method is performed than for each species. The advantage of this method is that the system matrix remains unchanged during evaluation of all population. Only the vector of unknowns is varying. After first generation fitness function is computed for each and individual with lowest fitness function is discarded from the population. This process continues until the only one individual remains in population, and its values fulfill stopping criteria of NR algorithm.

5.8.2 Damped Newton-Raphson

Since 5.37 may not result automatically in each step, a modification of NR algorithm called Damped Newton-Raphson had been implemented to CLASP. It differ from standard NR in computation of the values for next iteration:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + t^k \triangle \mathbf{x}^k. \tag{5.40}$$

where parameter t^k is selected such that 5.38 is satisfied. Usually

$$0 < t^k \leq 1. \tag{5.41}$$

In CLASP it defines t equal to 0.6 and 0.8.

5.8.3 Particle Swarm

Particle Swarm Optimization (PSO) [EK+95] is a form of a genetic algorithm for numerical optimization which uses advantages of swarm intelligence. Initialization of PSO is done by the generation of first random population referred as a swarm. The velocity vector is assigned to each particle in the swarm. The objective function is evaluated by the position of each particle. The position of a particle with the highest value is saved to global memory. Every individual knows the position of the individual with the best objective function. This best position is called $\mathbf{g}_b$. Every particle knows its best position in all generation and checks whether the current position is not better than known one. This post is denoted as $\mathbf{p}_b$. When every particle knows its $\mathbf{g}_b$ and $\mathbf{p}_b$ position, next move of swarm can be performed, and particles change their velocity and position according to

$$\mathbf{v}_i^{k+1} = \mathbf{v}_i^k + c_1 \cdot \xi(\mathbf{p}_i - \mathbf{x}_i^k) + c_2 \cdot \xi(\mathbf{g} - \mathbf{x}_i^k), \tag{5.42}$$

where $\mathbf{v}_i$ is velocity of the particle, c_1 and c_2 are learning factors, ξ is random variable in the interval $(0, 1)$ and $\mathbf{x}_i$ is position of particle i.

$$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + v_i^{k+1}. \tag{5.43}$$

The advantage of the algorithm is its simplicity. Computation of Jacobian matrix or inversion of the matrix system is not required. Simple vector operations are performed. On the other hand, the probability of a finding of the optimal solution depends on bounds of the search space.

5.9 CLASP Performance

5.9.1 LU Solver

The special algorithm has examined a performance of Native LU solver for automatized testing and creation of huge sparse matrices. The algorithm created a random linear circuit with defined number of devices. The results from benchmark simulation are shown in Table 5.1. There are presented to give some idea between tradeoff of performance and arbitrary accuracy that is part of Native LU solver. The column MATRIX SIZE includes a number of the elements in circuit matrix. The column TOT. TIME stands for total computing time of the LU factorization in seconds. The last column BYTES shows memory usage during the simulation. It should be noted that maximum size of the matrix was in the first Native LU solver limited to $102,400$ due to estimated time of the solution, that was out of reasonable range.

SOLVER	MATRIX SIZE	TOT. TIME	BYTES
GSL LU	1,024	0.413	33,232,832
	9,216	0.446	36,950,800
	102,400	0.918	66,508,816
	921,600	4.144260	299,094,240
Native LU	1,024	0.293466	1,295,888
	9,216	12.373999	29,603,072
	102,400	1,083.0775	1,040,184,336

Table 5.1: Comparison of different CLASP LU solvers

5.9.2 Comparison with MATLAB

It could be noted that professional computer simulation environment MATLAB is more suitable for any simulation. For example the problem of LU factorization of the complex matrix **A** requires in Matlab only one command [L,U]=lu(y) and computation of a complex matrix

$$\mathbf{A} = \xi_{ij} + j \cdot \xi_{ij} \tag{5.44}$$

with random values required less than one second of CPU time. For comparison of CLASP with MATLAB series of experimental tests have been made. The comparison of the results is placed in Table 5.2. In column MATRIX SIZE is number of rows and columns, remaining two columns include appropriate times of the factorization in seconds.

MATRIX SIZE	TOT. TIME	
	MATLAB	GSL LU
100×100	0.01	0.43
500×500	0.2	3.2
$1,000 \times 1,000$	1.7	20.2
$2,000 \times 2,000$	20.3	116.4

Table 5.2: Complex matrix factorization

5.9.3 Simulation of Nonlinear Device

Several simple nonlinear circuits were used for the testing capability of solving nonlinear equations in CLASP. Testing criterion was convergence criterion. As it has been said, several different nonlinear solvers had been implemented. In Table 5.3 are the results. In the first column are placed abbreviations of the solvers: NR denotes standard Newton-Raphson algorithm, DNR is damped Newton-Raphson and GNR is Generic Newton-Raphson algorithm. In remaining columns are averages numbers of iteration. Every simulation has been repeated at least in 100 times.

5.10 Chapter Summary

In this chapter, several new aspects of circuit simulation have been presented. The most important part dealt with an unusual usage of the functional programming language Common LISP as a simulation program for electronic circuits denoted as CLASP. Special attention has

Method	Simple diode	Zener diode	ZD stabilizator	Thermistor	Full-way rectifier
NR	40.72	27.16	2.00	2.00	16.84
DNR - 0.8	54.70	26.98	13.00	12.00	15.78
DNR - 0.6	75.48	28.78	21.00	20.00	20.94
DNR - 0.4	116.32	44.84	36.70	33.00	32.58
DNR - 0.2	237.72	80.58	80.00	67.00	68.18
GNR - 2	80.16	300.00	174.42	1.00	25.58
GNR - 3	52.40	83.94	2.00	2.00	14.54
GNR - 4	65.98	71.42	96.98	1.64	16.24
GNR - 5	49.04	127.66	98.32	2.00	13.08
GNR - 6	43.66	151.16	157.48	2.38	11.9
GNR - 7	31.00	133.16	3.68	2.80	14.44
GNR - 8	21.48	94.98	3.52	3.00	15.42
GNR - 9	11.14	119.00	119.74	3.00	14.14
GNR - 10	21.42	163.74	163.74	3.00	10.98

Table 5.3: Average number of iterations

been paid to its principle of the automatic self-modifying engine that created very powerful and efficient possibilities applicable to the circuit simulator. The unique organization of the program allowed to eliminate the redundancy in device modeling and helped to reduce requirements on programming skills of the user. The particular device model is simply defined by rewriting its mathematical description to CLASP code. Once the device is incorporated into CLASP, it can be used in any defined simulation. It is important to emphasize the fact that not only variables can be dynamically defined, but also functions and even entire program source code can redefine itself in run-time. To demonstrate the simplicity of the device modeling in program CLASP several device models were presented in the section about device modeling. The representation in system matrix has been given for trivial devices as a voltage source, resistor, capacitor, diode and more. Full source code of the slightly more complicated ones has been placed into the Appendix of the thesis.

The usage of foreign library interface to incorporate LU solver written in C (GNU GSL library) to CLASP program proved to be suitable solution 5.3. Although, that created LU solver was designed to work with sparse matrices (additionally it can work with the arbitrary precision) it showed to be slower in performance and efficiency than the solver implemented in the extern library. Taking into account that CLASP is not going to be a new competitor of MATLAB (it represents missing supplement in circuit simulation) linear solver, the differences in times are negligible. Table 5.3 shows results of several solvers of nonlinear equations.

The most significant difference between CLASP and other simulation programs from SPICE family lies in the usage of functional and unnamed functions. The capabilities of CLASP based on the ideology of functional language allows adapting the self-modifying process of simulation that can arbitrary optimize itself to fulfill particular needs of simulation problem. Nowadays, programming languages experiencing a rebirth of functional programming and many of popular languages such as C++ or Java were extended to support functional programming and unnamed function definition. It opens new ways to implement presented functional

ideology natively in structural language. The following Chapter 6 presents the state of the art programming possibilities applicable not only for SPICE simulator but a simulation of electronic circuits in general.

5.10.1 Research Contributions Summary

- New simulation program for electronic circuits denoted as CLASP based on unusual usage of the functional programming language Common LISP.
- Device models defined by functional syntax allowing simpler definition and modification of device types.
- The functional chaining mechanism that transforms the standard definition of simulation to a chain of linked functions that evaluates upon call.
- Improvement of the CLASP simulator performance through Foreign Array Interface.

Chapter 6

Definable Solver

6.1 Introduction

In this chapter, an enhancement of SPICE simulation core is proposed with modified nodal formulation implemented in functional programming style allowing functional chaining and adaptive simulation definitions. The core idea is based on an additional computation layer between model definitions and their enumeration during analysis procedure. This can be achieved with using a functional language together with Foreign Function Interface FFI procedure or directly in C++-11, C++-14 programming languages with functional programming style. More information on the advantages of usage of functions/aspect oriented programming style can be obtained from [EFB01, CAT14, ZCF+10, ABH06]. Basing on the assumption that electronic device models definitions are introduced to the simulation as chained functionals that will autonomously evaluate themselves to values upon call. It is evident that trying to convert those entries to matrix system to compute matrix inversion or solution of the linear system is redundant operation. It seems to be a solution to chain all entries to another function that on call evaluate entire simulation. The key factor of implementation lies in the use of iterative recursion. From the point of view of the simulation core, it is not clear how many unknown variables each model introduces to a circuit. One possible solution is to make a generic get/set function returning values by their reference. A better implementation can be done utilizing a two-or-more-stage self-redefinition mechanism denoted as "functional chaining". In the first stage, model functions returns, as a result, another function with internal variables adapted to simulation unknowns as voltages, currents or temperature.

```
(defun create-function
    #'(lambda ()))
```

On the second call to the chaining procedure, all unknown variables will evaluate themselves according to simulation state. This mechanism allows to handle device models as variables and keep them together with other numeric values in circuit array. In practice, it means that during the computation of particular analysis, computation core handles values and functionals as numeric types that evaluate themselves to numbers when they are used. Evaluation of functional objects can be done in LISP by calling "FUNCALL" function:

```
(defun eval-function-list (function-list)
  (loop for x in function-list
    collect (funcall x )))
```

It should be noted that it will result into a list of evaluated numbers i.e.

```
(list 1.0 2.0 3.0 4.0)
```

or it could be direclty added to a result applying function

```
(defun sum-eval-function-list (flst)
  (apply #'+
    (loop for x in flst
      collect (funcall x )))
```

Functions defined through this mechanism can be treated as values [ČD11] despite that they contain unknown values. It allows computation core to handle those functions, referred to as functionals, as any normal number and store them in vectors, matrices or sparse matrix systems. It is extremely useful for Jacobi matrix definition where the list of nonlinear functions is evaluated into a sparse matrice entries.

$$\mathbf{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \tag{6.1}$$

where each entry is given by derivative of all devices connected to the particular node. Then in a case, that to node 1 are connected three devices then each entry is defined by

$$\frac{\partial f_1}{\partial x_1} = \frac{\partial f_{1d1}}{\partial x_1} + \frac{\partial f_{1d2}}{\partial x_1} + \frac{\partial f_{1d3}}{\partial x_1} \tag{6.2}$$

where index d stands for particular device model. Those entries can be numerical values, zero values for linear states or functions for more complicated derivation. Enumeration of the previous statement with functionals can be done in symbolic-like style when all equations are parameters of parental or so-called root functions that on call evaluates all equations to a single total. It could be easily performed in functional language LISP as follows

```
(defun eval-j-func-list (j-func-list)
  (loop for row in j-func-list
    collect
      (apply #'+
        (eval-function-list row ))))
```

In the case of the numerical solver, it is often difficult to implement mechanism allowing to keep values and nonlinear function objects in one place. In symbolic computation this is suppressed by nature of the language but in structured programming paradigm, it is solved by external calls. They are initiated whenever model evaluation is needed and simply perform the following operation: it first checks all models in the circuit and their settings and then iterate over all of them and resulting values adds to relevant places in circuit matrix. It could be seen for example in implementation of CMOS model defined by open-source continuation of SPICE program NgSpice:

```
/*  loop through all the MOS3 device models */
for( ; model != NULL;
        model = model->MOS3nextModel ) {
/* loop through all the instances of the model */
  for (here = model->MOS3instances; here != NULL ;
     here=here->MOS3nextInstance) {...}}
```

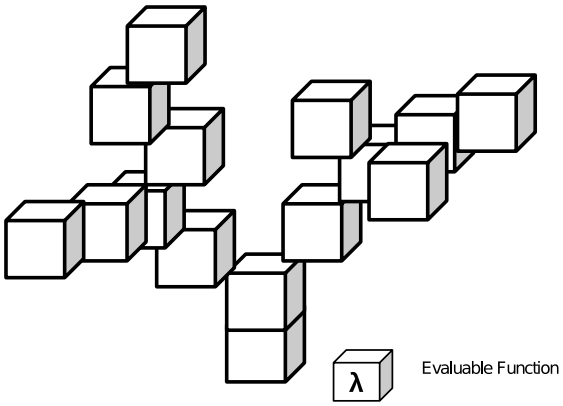


Figure 6.1: Visualisation of functional chaining mechanism

1M operations	add.	mult.	div.
C Double (ms)	3.081	2.802	24.268
C Arbitrary (ms)	31.9	69.8	137
Lisp Double (ms)	1126.504	1143.344	1127.40
Lisp rationals (ms)	1778.264	1431.30	1513.983

Table 6.1: Comparison of one million operations in C language and LISP

In this thesis described mechanism of self-evaluating functions depending on each other that result on a call to a numeric value is referred as "functional chaining". Visualization of the chained functional tree can be seen in Figure 6.1. Each block represents one functional which on call results in value. The chain is given by dependence of internal function values on another function that needs to be evaluated before its parent function. Then entire list on one call, based on current circuit settings, can evaluate itself to one numeric value. This mechanism, unfortunately, brings several drawbacks to the analysis process. As the most problematic can be marked computation routines implemented in the functional programming language. Functional languages suffer a lot from computation efficiency viewpoint. It is mainly caused by real memory size of functional numeric types. In Table 6.1, there is a comparison between the performance of basic operations in C and LISP. The difference in performance between the two mentioned languages is without a doubt. The performance of computation is affected in functional languages not only by memory requirements but also by programming syntax that forces syntactic correctness at the expense of efficiency. As an example, it can be again mention previous LISP definition where conversion of evaluated functions into a list is obviously redundant operation.

6.2 Foreign Array Interface

Efficiency problem caused by a switch to functional language can be resolved in two ways. The first is based on a property of functional language LISP which allows calling directly external functions written in C/C++ language through a mechanism called Foreign Function Interface (FFI). In other languages offering similar functionality, it can be denoted as language bindings (ADA) or Java Native Interface (Java). This mechanism comes together with the possibility to pass values or large array between languages called Foreign Array Interface (FAI). With FFI, it can extend each block of chained tree with an external C/C++ layer

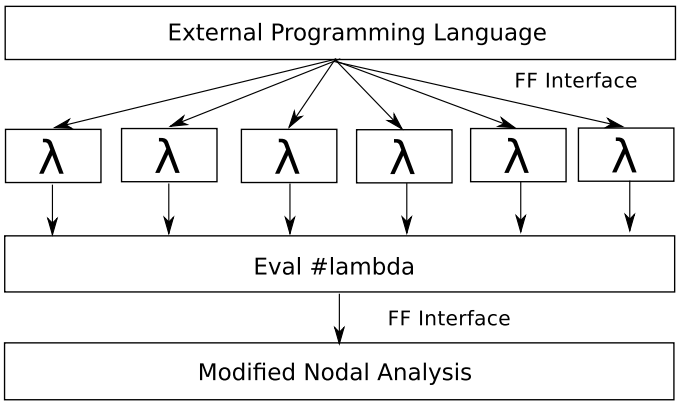


Figure 6.2: Visualisation of cooperation of FFI with functionals

responsible for demanding computations. It can be seen in 6.2. Functional defined λ symbol denoted self-evaluated functions. That function uses through FFI functions from external programming language (most probably C/C++). Results after evaluation are passed to eval lambda function which produces entry numerical result for MNA at given time point.

```
(multiple-value-bind
  (jacobi-matrix perm)
  (gsl:lu-decomposition jacobi-matrix)
  (ignore-errors
    (gsl:lu-solve jacobi-matrix rhs-vector perm)))
```

This mechanism works only for calling functions for passing values from functional language LISP to C/C++ language. It is useful in the situation when all simulation program is implemented in the functional language, and it uses external language only for demanding computations. For current implementation of NgSpice program, which is entirely written in C/C++ language, it is more convenient to use the opposite way when C/C++ language calls through some interface functions implemented in the functional language. For this purpose Embeddable Common Lisp (ECL) can be used. The simulation core and algorithms of NgSpice program can then remain the same, and only device models must be rewritten into a functional language. The external call for evaluation of the whole circuit network can be done by calling:

```
#include <ecl/ecl.h>
...
extern void init_lib_NET_LISP(cl_object);
ecl_init_module(NULL, init_lib_NET_LISP);
cl_eval(c_string_to_object("(generate-net)"));
...
```

6.3 Native Fuctional Interface

As the second possibility, the native functional interface offered by a new version of C++ language can be used. Since an update of the language standard informally denoted as C++11, it provides a new functional library with polymorphic function wrapper. Instances of "std" functions can store, copy and invoke any callable target - functions, lambda expressions (anonymous functions). This standard was approved by International Organization for

Standardization (ISO) in 2011 and extended in 2014 by the actual standard C++14 formally known as "International Standard ISO/IEC 14882:2014(E) Programming Language C++". The functional library can be added to code by including:

```
#include <functional>
```

The new standard allows bringing advantages of functional programming languages into originally structured language. From the performance point of view, it is also better to perform all computations in one place. The previously defined MOS device model can be directly rewritten with C++-14 language standard as follows:

```
auto mos_current_drain =
  [](class_mos mos, auto vg, auto vs, auto vd){
  return [mos, vg, vs , vd]() {
    auto v_gs  = vg() - vs();
    auto v_ds  = vd() - vs();
    auto v_omega  = v_gs - mos.VTH;
    auto v_lambda = 1.0 + ( mos.LAMBDA* v_ds);
    if(v_gs < mos.VTH)
      return 0.0;
    else if(0.0 < v_ds && v_ds < v_omega)
      return (mos.BETA * v_ds *
        v_lambda * (2*v_omega - v_ds));
    else if ( 0.0 < v_omega && v_omega < v_ds)
      return mos.BETA * v_lambda *
        v_omega * v_omega;
    };
};
```

It could be seen that the standard comes with new unassigned variable types. These values are set to variable type "auto" and it is a decision of the compiler to set those values (important is the fact that "auto" variable type with can hold not only primitive values such as int, double etc. but also functions). In the example, they mostly evaluate to double precision value type but for example values vg, vs, vd which return functionals need to evaluate by chaining mechanism. It means that variables representing circuit voltages are also evaluable functions whose values are unknown in the time of definition and will be evaluated in the time of analysis by "mos_current_drain" function. It is obvious from the structure of the definition that outer function returns another unnamed function. It allows to assign it to the variable and save it together with standard values in vector or array. For example, if it is required to pair voltage values with randomly generated RHS value vector defined for example with random starting values as

```
  double RHS [5] = { 1., 2., 3., 4., 5. };
```

then it can define a function with generating self-redefinable function for each voltage node

```
  auto capture = [](auto * b) {
    return [=]()
    {
      return  *b ;
    };
  };
```

and invoke it automatically on each voltage node of particular device. In a case of Junction diode it will be

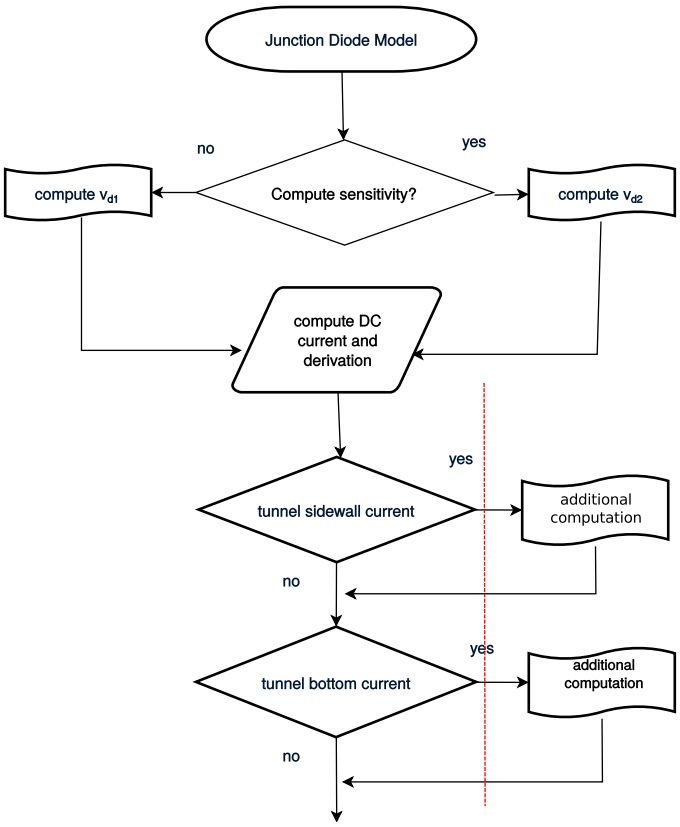


Figure 6.3: Part of Junction Diode Model Evaluation (red line indicates optional part)

```
auto vs = capture(&RHS[i++]);
auto vd = capture(&RHS[i++]);
auto vg = capture(&RHS[i++]);
```

Then, it is possible to implement a functional algorithm that for each device creates a set of functional voltage nodes (or reuse already created nodes) and pass those values to generating function for device models. Resulting functional model can be saved in a vector or array and be handled as standard number

```
class_mos mos_model;
std::function<double()> net[2] = {
  mos_current_drain(mos_model, vg, vs, vd),
  mos_current_drain(mos_model, vg, vs, vd)
};
```

In the example variable “net” represents sparse matrix array holding only nonlinear function definition. In the same manner, it can be implemented for entire sparse matrix system of the circuit.

6.4 Performance of Functional Definition

Regardless of the implementation, functional definition of electronic device models can improve the performance of their evaluation. It is mainly done by the ability of defined functionals to change themselves based on current settings of the circuit. It can be imagined as the C++ object that can redefine its internal algorithms based on constructor parameters. During

	Number of Comptation Loops			
	100000		1000000	
op.	cnd	direct	cnd	direct
1	0.001243	0.001227	0.014535	0.011456
5	0.010895	0.007782	0.059708	0.04871
10	0.017508	0.010138	0.119693	0.09735
20	0.03048	0.019403	0.215232	0.193446
30	0.038576	0.027959	0.334274	0.304412

Table 6.2: The influence of redundand conditions on time of calculation

analysis, NgSpice program for each time-point iterates over all present device models and recomputes circuit variables. During it, same code, no matter how the model was initially defined, is called. It results in needles condition checks of states that can never happen. There is the diagram showing part of Diode model evaluation in Figure 6.3. It should be noted that execution path of given part is known to the simulator at the beginning of analysis and stays the same to the end of the simulation. Therefore, it is redundant to evaluate them for each computation.

In standard simulation process when Junction Diode Model is invoked, the computation core goes through all definition and conditions of the model and checks whether given condition branch should be computed or not. Those condition checks are repetitively checked each time when the model is enumerated and even in the situations that those conditions during this particular simulation could not have a chance to occur because of the definition of simulation or by device parameters definition. It is evident that those checks are redundant and can be removed. In Figure 6.3 can be seen three conditions asking whether sensitivity should be computed while diode voltage is computed. Also, there are two additional conditions for computation of tunnel sidewall current and tunnel bottom current. A user can preset all of those values at the beginning of the simulation and stay the same for entire simulation. Therefore some mechanism allowing to modify device model in a way that those unnecessary operations are stripped would be beneficial. Presented functional definition allows exactly this optimizing process when calculation are chained into the sequence individually by given model settings and thereby it allows to remove unnecessary operations.

The comparison of computation time of redundant conditional computation (denoted as "cnd") and direct computing (direct) is in Table 6.2. It is showed for 100 thousand and 1 million loops. In column "op." are numbers of unnecesarry condition calls. For simple devices as presented diodes those calls are around five conditions with more sophisticated device models such as BSIM3 it can grow over 20 calls.

From Figure 6.4 it is evident that native implementation of functional device models (NFI) is a better solution than the external one (FAI). The native definition gives additional optimization advantage compared to standard implementation. As it was introduced before with defined functionals use in their function definition real adresses of current circuit variables. In standard implementation it is realized by the model variable that represents pointer offset in vector array. To give some idea in NgSpice for diode model parametres it is realized as

```
vd= *(ckt->CKTstate0 + here->DIOvoltage);
cd= *(ckt->CKTstate0 + here->DIOcurrent);
gd= *(ckt->CKTstate0 + here->DIOconduct);
```

where "ckt->CKTstate0" is unknown variable array and "here->DIOvoltage" is offset for current device model. With direct addressing, this is not needed anymore, and all values are assigned in a time of model definition and directly obtained during model evaluation.

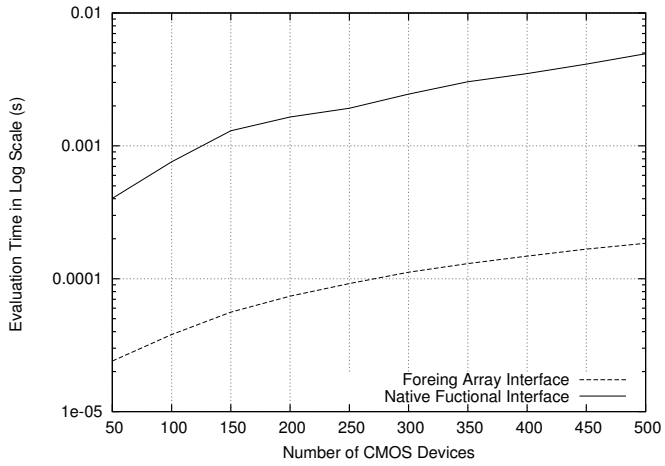


Figure 6.4: Comparison of computation performance for FAI and NFI

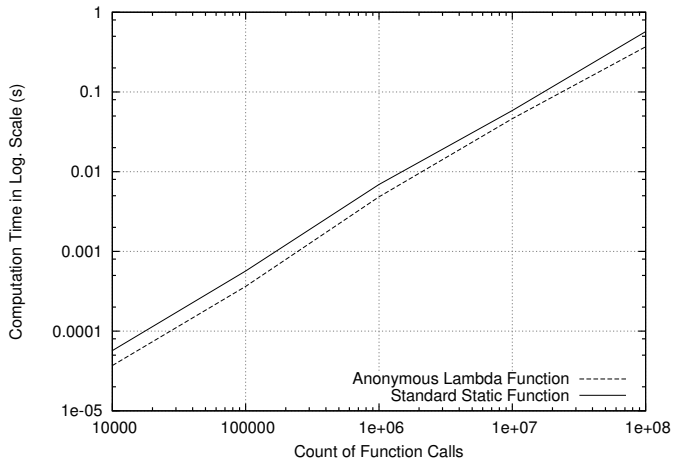


Figure 6.5: Performance comparison of anonymous and standard function calls

It provides additional improvement in model evaluation performance and correspondingly in the overall performance. In Figure 6.5, there is the performance comparison of standard static defined function calls and functionally defined calls of anonymous functions where the standard static function performs variable evaluation as it is defined in NgSpice and the anonymous function performs functional direct addressing.

6.5 Chapter Summary

It should be noted that functional languages can be very beneficial from the point of view of simulation variability and definition. They can bring new possibilities to the device models and also to analysis definition. On the other hand, it will be very inappropriate to use them for demanding numerical operations that are required by simulation analysis. Therefore it is recommanded to use FFI mechanism allowing to pass extensive computations to more appropriate language for numerical computation such as C/C++ (FORTRAN). As it has been presented in this chapter it can be implemented together with methods for enhancing circuit simulator core with functional layer to the core of open-source circuit simulator NgSpice in two ways.

The first method is based on an assumption that all device models were rewritten in

a functional language and only parts of model definitions that can affect computation performance where recomputed by external C/C++ through FFI code. It is evident not only from performance comparison (Figure 6.4) that this method will be slower than the native one. On the other hand, the usage of functional language brings advantages of computation with arbitrary precision numbers. It should be noted that evaluation of entries of circuit matrix is only small part of the entire analysis process. As the most demanding can be marked pivotization together with matrix factorization. Therefore, it can be assumed that overall performance will be affected only partially, and a user will benefit from evaluation with arbitrary precision without additional code modifications. In that case, if computation core allows it, the precision of the analysis results can be significantly increased.

The second proposed method is based on the use of new standard libraries allowing functional operation directly in C/C++ language. It is clearly faster solution as can be seen in Figure 6.4. It can also be natively implemented to NgSpice without linking additional libraries. On reimplementation of CMOS transistor, it has been shown that it is possible to implement device model with same manner and functionality as with a usage functional language LISP. Despite the fact of slower evaluation in the first case, both methods extend standard models with one important mechanism. It is a product of inner functional chaining and allows introducing behavioral modifications through functionally defined variables of models. The presented functional implementation offers a similar functionality as the one in NgSpice's extension XSpice. It provides the additional environment for simple implementation of new models (as in XSpice) but also it provides a mechanism for changing device behavior. With the functional definition, this is natively allowed for all devices without any limitation. As mentioned before, turning the definition of circuit models into self-evaluable functions allows to implement functional chaining mechanism and handle computation in a manner similar to symbolic solvers. This mechanism gives to simulator core a possibility of self-optimization of individual device models where they can be at the beginning of the analysis stripped of redundant conditional computations. The functional models can obtain circuit variables directly through their memory pointers unlike in standard implementation where this is performed through integer indexes. Both features can slightly improve computation performance as is shown in Table 6.2 and Figure 6.5. Even though that provided methods can be easily implemented into a NgSpice simulator, they still require more or less rewriting of standard models. This can be also resolved with functional changing mechanism with an implementation of functional wrapper between model representation and MNF. Each model can then remain without any change, and only specific functions are called through the functional layer.

6.5.1 Research Contributions Summary

- Functional layer allowing the introduction of user-defined dependencies and variables to simulation.
- Self-modifying process of simulation that can arbitrary optimize itself to fulfill particular needs of simulation problem.
- Principle, definition, and syntax of functional simulation core of electronic circuits simulator defined in functional programming language LISP but also in object-oriented language C++ with the possibility to reimplement the same principles in other languages such as Java or Python.

Chapter 7

Methods Improving Accuracy

7.1 Introduction

Floating point numbers are designed to be an efficient representation of the real numbers but with a limited accuracy that is directly related to their magnitude. It si perfect idea that works for most cases. For double precision numbers (64 bit machine) IEEE 1987 defines 52 significant bits plus 1 sign and 11 exponent bits. It gives us range from 10^{-323} to 10^{308}. It seems to be an enormous scale of numbers, but there is always a catch. Nevertheless, of scalability floating point numbers are precise only within some region of magnitude. It implies that the situation when precision goes wrong can occur, and, unfortunately, it have a higher probability to happen during simulation of electronic circuits. Floating point numbers can not be as precise as their real counterparts defined by mathematical apparatus. For example number, 0.1 will become

$$0.1000000000000000055511151231257827021181$$

and the even more different result can be obtained by definition of 1.1×10^{-8} that will become

$$1.099999999999999923753143406777998958290 \times 10^{-8}$$

Although those numbers have a finite decimal representation, in computer memory they are an infinitely repeating numeric sequence. It is caused by the fact that they lie exactly between two representable floating point numbers and therefore in computer memory they have to be rounded to one of them. Floating point numbers also include several non-numeric definition as infinity for larger (or smaller) numbers than highest (or lowest) representative number, positive and negative zero and "not a number" (NaN) definition for undefined states (division by zero, etc.). There are also other problems that are caused by operations performed with floating point numbers. One of them is called Catastrophic Cancelation. It hapens during substraction of two nearby floating point numbers. There are two kinds of it: Catastrophic and Benign. Benign Cancellation is in most cases treated by guard digits, and therefore will result in relative small error. On the other hand Catastrophic Cancellation most offen happen while two numbers, accuratly represented in memory, are subtracted one from each other. In that situation cancellation can wipe out all valid bits and leave only those contaminated by rounding error. To see Let's consider relation

$$y = f(x) \tag{7.1}$$

for x where $|x - \hat{x}| - > 0$ it have

$$\hat{y} = y + f'(x)(\hat{x} - x) \tag{7.2}$$

for which relative error in y to the relative error in x called is defined as:

$$\kappa(x) = \left| \frac{f'(x)x}{f(x)} \right|. \tag{7.3}$$

Bringing up four basic operations that can be performed on floating point numbers: addition, subtraction, multiplication, and division it can define relative error of individual operations that can affect the computation. Let's consider given operation between accurate number a and floating point number x affected by the relative ϵ_x:

7.1.1 Multiplication

$$a * x, \epsilon_y = \left| \frac{ax}{ax} \right| \epsilon_x = \epsilon_x, \tag{7.4}$$

7.1.2 Division

$$y = a/x, \epsilon_y = \left| \frac{ax^2}{ax^2} \right| \epsilon_x = \epsilon_x, \tag{7.5}$$

7.1.3 Substraction

$$y = a - x, \epsilon_y = \left| \frac{a}{a-x} \right| \epsilon_x. \tag{7.6}$$

It is visible from Eqations 7.4 and 7.5 that multiplication and division are both safe operation. The relative error ratio that is added to result is equal to the error of input. On the other hand subtraction and addition (Equation 7.6) specially when numbers have opposite sign are both unsafe uperation when the numbers are close to each other. It is clearly visible that denominator less than one can rapidly increase relative error. There is another implication of problems caused by a limitting precision of the floating point numbers. In a case ill-conditioned linear systems the accuracy of the solution can be rapidly decreased. Although that matrix condition number is not directly related to the accuracy, it can be taken as an indicator that the solution may be affected by an error. For example, having a system:

$$\begin{cases} 4.012x_1 + 4.013x_2 = 4011 \\ 4.013x_1 + 4.013x_2 = 4012 \end{cases} \tag{7.7}$$

its computation using double precision will result in residual error r computed by L2 norm $r = 4.547474 * 10^{-13}$ it is significantly higher contrary to expected value $r < 10^{-15}$.

7.2 Arbitrary Newton Iteration Method

The original mathematical definition of Newton Iteration Method (NIM) [BI65, BIC66] can be rewritten to the form applicable for implementation as an iterative algorithm for precision enhancement. Denoting Jacobian matrix as $\mathbf{J}$ and Identity matrix as $\mathbf{I}$, for inverted Jacobian matrix $\mathbf{J^{-1}}$ then it could be written

$$\mathbf{J}^{-1} = \mathbf{J}^{-1}(2\mathbf{I} - \mathbf{I}) = \mathbf{J}^{-1}(2\mathbf{I} - \mathbf{J}\mathbf{J}^{-1}). \tag{7.8}$$

For an arbitrary real matrix, $\mathbf{J}$, the above statement allows for each iteration step n to define a generalized inverse (pseudo inverse) denoted as $\mathbf{X_n}$

Dim.	Initial Estim.	Residual	30 Iterations
50	0.000072	0.000072	0.004254
100	0.000414	0.000414	0.023759
150	0.001334	0.001334	0.086467
200	0.003287	0.003287	0.216549
250	0.006375	0.006375	0.412877
300	0.011092	0.011092	0.693523
350	0.017103	0.017103	1.07149
400	0.02443	0.02443	1.536133
450	0.034815	0.034815	2.190221
500	0.045693	0.045693	2.821963

Table 7.1: Computation time in Sec. of individual steps of NIM

$$\mathbf{X_n} = \mathbf{J}^{-1} + \varepsilon_\mathbf{n} \Rightarrow \mathbf{X_{n+1}} = \mathbf{J}^{-1} + \varepsilon_{n+1}, \tag{7.9}$$

where ε_n is an error matrix. From that it is received

$$\mathbf{J}^{-1} + \varepsilon_{n+1} = (\mathbf{J}^{-1} + \varepsilon_n)(2\mathbf{I} - \mathbf{J}(\mathbf{J}^{-1} + \varepsilon_n)). \tag{7.10}$$

The above form can be rewritten using $\mathbf{X_n}$ and $\mathbf{X}_{n+1}$ to a more readable form

$$\mathbf{X}_{n+1} = \mathbf{X}_n \left(2\mathbf{I} - \mathbf{J}\mathbf{X}_n\right). \tag{7.11}$$

It is obvious that previous equation will iterate to the solution if

$$0 < \|2\mathbf{I} - \mathbf{J}\mathbf{X}_n\| < 1. \tag{7.12}$$

There are two important remarks for this solution. First is connected to nature of Spice simulator. It does not compute matrix inversion of Jacobian matrix. Instead, it computes next iteration steps of NR algorithm directly evaluating equation

$$J(x_n)(x_{n+1} - x_n) = -F(x_n). \tag{7.13}$$

The second remark is that the circuit matrix is going to be very sparse and more of that its density decrease with increasing matrix size. It concludes that computation of matrix inversion proves to be very inefficient for huge matrices from a point of view of memory handling. On the other hand stability of algorithm and simplicity of implementation together with possible optimization makes algorithm very efficient for computation with matrix dimension up to 200 (circuits with up to 200 independent voltage nodes). Putting it all together an then final algorithm received:

$$\begin{cases} \mathbf{X}_D = \mathbf{J}_D^{-1} \\ \mathbf{X}_{D+} = \mathbf{X}_D \left(2\mathbf{I} - \mathbf{J_D}\mathbf{X}_D\right), \end{cases}$$

where $\mathbf{J}^{-1}$ stands for inversion of the current computation of the Jacobian matrix in current precision that can be easily obtained from last LU factorization. This will be used as a starting point for linear solver. $\mathbf{X}_{D+}$ represents matrix inversion with increased precision. Both matrix multiplications in second line must be computed with increased arbitrary precision to achieve increased accuracy in $\mathbf{X}_{D+}$. Then more accurate solution of the time step is given by simple vector matrix multiplication

option	optimization level
-O0	opt. for compilation time (default)
-O1 or -O	opt. for code size and execution time
-O2/O3	opt. more for code size and exec. time
-Os	opt. for code size
-Ofast	O3 with fast none accurate math calc.

Table 7.2: Overview of compiler optimization settings

Dim.	LUF	NIM	LUF -O1	NIM -O1	LUF -Ofast	NIM -Ofast	ratio [%]
50	0.0016	0.0012	0.00053	0.00027	0.000405	0.000	53.08
100	0.0078	0.0089	0.0019	0.00194	0.0019	0.000	42.65
150	0.023	0.029	0.0089	0.006	0.0052	0.003	57.75
200	0.0304	0.070	0.018	0.016	0.0098	0.007	76.01
250	0.053	0.136	0.016	0.031	0.0108	0.014	131.39
300	0.079	0.234	0.035	0.055	0.0214	0.023	111.18
350	0.17	0.371	0.046	0.086	0.031	0.036	115.80
400	0.206	0.566	0.075	0.13	0.042	0.052	125.08
450	0.29	0.787	0.109	0.18	0.047	0.077	164.28
500	0.48	1.077	0.137	0.25	0.064	0.101	157.33

Table 7.3: Comparison of performance of NIM and LUF to matrix dimension

$$\mathbf{X}_{D+}\mathbf{y}_{RHS} = \mathbf{x}_{D+}$$

The simplicity of implementation of NIM allows to compiler optimize the code in such a way that performance can be rapidly increased, but with increasing of matrix dimension memory requirements of this method rapidly decrease performance. This can be visible from Table 7.1. The number of required iteration to obtain a result with particular precision, unfortunately, depends on initial estimation and matrix dimension. Therefore, this algorithm will offer best results only in a case of one iteration.

7.2.1 Performance

Adapt NIM algorithm as an efficient competitor to LUF is not an easy task. First of all, it's hard to optimize two matrix multiplications that must be performed during NIM computation. Secondly, with increasing size of sparse Jacobian matrix with also, increase unfortunately dense matrix J^{-1}. It is caused by the nature of matrix inversion, and there is no way to suppress it. Therefore, the best usage of the algorithm is for matrices with dimension up to 200 when it can be taken as dense. Although, optimization of efficiency can be done. Current compilers (e.g., GCC) offers various optimizations settings that can rapidly improve calculation performance and memory handling without a need of parallelisation or other techniques (Table 7.2). It should be noted that it does not directly concludes that higher optimization attribute equals more efficient result. It is nature and simplicity of NIM that it is so well optimizable. It is hard for iterative methods or even impossible to overcome direct one. Mainly it is because that direct methods evaluate directly to the solution contrary to the iterative algorithms that need several iteration loops before they obtain a result with sufficient precision. It is the reason why LUF is a better option than NIM for standard computation.

Another optimization can be done reducing the number of iterations. It is obvious that

Algorithm 6 Modified Transient Procedure

```
Simulation definition
Modified Nodal Analysis
for Time do
  Initial value estimation
  DC Analysis (optional)
  NIM - Arbitrary (optional)
  if Non-linear System then
    repeat
      Generate Jacobian matrix
      LU of Jacobian matrix
      Evaluation of next value estimation
      Compute residual
    until Stopping criteria or convergence
  end if
  NIM - Arbitrary
end for
```

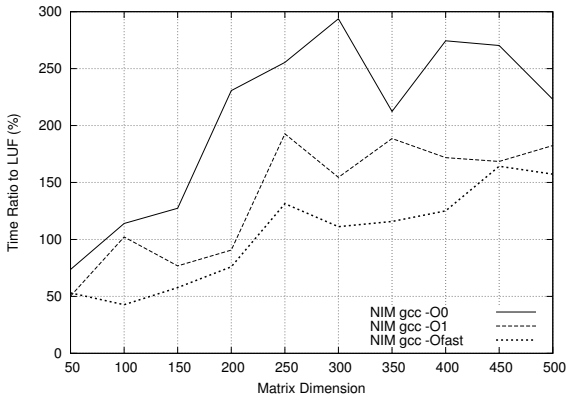


Figure 7.1: Time-ratio of NIM to LUF vs matrix dimension

fewer iterations mean less computation time. Therefore, the algorithm was limited to only one iteration. There is computation of initial inversion matrix from already precomputed LUF followed by two matrix-matrix products. This algorithm is pretty simple and straightforward (no factorisation or precision comparison is needed). It gives to NIM for small matrice an advantage among over any other algorithms. Together with very straight convergence a simplicity of the algorithm allows very fast implementation of arbitrary precision numeric types. It will be shown that usage of arbitrary precision numbers in computation even for one iteration of NIM can directly improve resulting accuracy by factor two. For instance, if the final precision of LUF was around 10^{-16} next iteration with NIM could increase it up to 10^{-32}.

In the Table 7.3 there is a comparison of computation times of LUF and one iteration of NIM for different optimization levels of GCC compiler. It is clearly visible how optimization attributes to improve performance. It can be seen that up to matrix dimension 200, implementation with NIM proves as a faster solution. It is also shown in the Figure 7.1. Table 7.1 gives a slight look on efficiency of different stages of NIM iterations for various matrix dimensions. It is visible that the most critical is a fact that several iteration loops are required. It is evident that the iteration loops can be taken as the most time demanding operation

Dim.	Double	→ Arbit.	Long	→ Arbit.
50	1.47E-14	2.72E-28	1.56E-17	3.92E-30
100	1.14E-15	5.21E-29	2.86E-17	6.21E-30
150	6.32E-15	1.89E-29	2.54E-17	2.18E-31
200	2.03E-13	2.69E-27	1.58E-16	3.85E-29
250	2.16E-11	3.27E-24	8.96E-16	5.47E-28

Table 7.4: Accuracy boost given by one iteration loop of NIM with arbitrary precision

during entire evaluation. Not only from this reason NIM is not used for computation or as replacement of LUF method but only as a so-called precision booster that improves the accuracy of the result during a single iteration. In Table 7.4 is the comparison of the accuracy of LUF (column: Double and Long Double) and LUF with additional NIM cycle computed with arbitrary numbers (columns with → Arbitrary) to matrix dimension. It is clearly visible that accuracy was significantly increased for both data types (Double and Long Double). The adapted Transient analysis completed by single loop NIM with arbitrary precision is in Algorithm 6. It got an additional precision boost computed by NIM after each successful iteration time-step.

7.3 Enhanced Accuracy of Numerical Integration Algorithm

There are two standard numerical integration methods in SPICE program, TRAPEZ and GEAR up to the order six. Each of the method comes up with own disadvantages, which can lead in a worst-case to non-convergence or wrong solution. To avoid this, an additional integration approach was developed and implemented into the SPICE family simulator. In this section for simulation is used open-source mixed-level circuit simulator NgSpice, which is based on three open source software packages: Spice3f5, Cider1b1 and Xspice. Of course for the simulation only first one of the listed is needed.

7.3.1 Numerical Integration with Predictor

NgSpice simulator includes various methods for evaluation. There are two methods as a base for checking accuracy of the simulation. The implicit TRAPEZ and GEAR method without predictor and TRAPEZ and GEAR method with appropriate explicit corrector. These two methods are already implemented in the NgSpice and therefore can be assumed as correct ones.

7.3.2 Numerical Integration with Modified Predictor-Corrector

The new approach is letting the simulator to obtain a solution by both methods together, TRAPEZ and GEAR and use solution as a leading one and second one as a supporting of the first one. The several heuristic approaches can be performed to obtain final solution.

$$\mathbf{x_n} = \sum_{i=0}^{k} \frac{'x_n^i + x_n^i}{2} \tag{7.14}$$

where $'x_n^i$ is a solution of GEAR method at node i and x_n^i is a solution of TRAPEZ method at node i.

It can be shown that this may lead to the less accurate solution, compared to one, obtained by using standard approach. For example, in the case when both, TRAPEZ and GEAR methods are close to the solution from a same side. Regardless, using this approach, it can suppress a solution in some situations, where particular method has some divergence problems.

New comparing approach was implemented for both, implicit and explicit algorithm. One important thing must be noticed. This algorithm is bypassed when an order of method increase to higher value.

7.3.3 Numerical Integration with Newton Predictor

New Method has been also implemented to the NgSpice new method utilizing divided differences.

From a theory is well known that Newton interpolation polynomial is modification of Lagrange interpolation polynomial with respect to optimize number of evaluation, when new value is added.

This is very convenient for numerical computation not only for time computation save but also because of better memory handling.

The implementation of explicit Newton predictor (NEWTON) method is composed by divided differences. Let's assume that there are known k steps of numerical iteration. $h_n \cdots h_{n-k}$. Then it also know k time position of NI and therefore it can be obtain a basis of (NEWTON) by formula:

$$\Psi_k(t_{n+1}) = \prod_{i=0}^{k-1} (t_{n+1} - t_{n-i}) \tag{7.15}$$

where k now denote an order of the method and could vary from $k = 1, \ldots, n$, where n is maximal order of predictor. In this case maximal order was 6, same like in GEAR method. $\Psi_0 = 1$ is done by the convention.

Zero order difference is done by simply by formula

$$\alpha_{n-k} = f(t_{n-k}) = f[t_{n-k}] = x_{n-k} \tag{7.16}$$

is is evaluation of the same value. First order equation is then defined by

$$\alpha_{n-k+1} = \frac{f(t_{n-k+1}) - f(t_{n-k})}{t_{n-k} - t_{n-k+1}} = f[t_{n-k}, t_{n-k+1}] \tag{7.17}$$

using induction it iterate to the solution k-th divide difference

$$\alpha_n = \frac{f[t_{n-1}, \ldots, t_{n-k}] - f[t_n, \ldots, t_{n-k-1}]}{t_{n-k} - t_n} \tag{7.18}$$

And finally it can be written down explicit Newton's Integration method of degree k, as

$$x_{n+1} = f[t_{n-k}] + \sum_{i=1}^{k} f[t_{n-k}, \ldots, t_{n-k+i}] \Psi_i(t_{n+1}) \tag{7.19}$$

It is important to mention that this implementation is using implicit GEAR corrector together with explicit NEWTON corrector.

7.3.4 Simulated Circuit

For simulation were picked three different circuits, which are well described in various literature and can be easily gathered for example, from the Web. In a following sections are listed only that parts of SPICE sources, which were changed for a purpose of simulation.

Method	MULTIPLIER	MOSOSCIL	MOSAMPL
Implicit GEAR	3869	16107	108571
GEAR with predictor	8281	18105	78823
GEAR with TRAPEZ	6206	20933	88042
Implicit TRAPEZ	3871	17846	91236
TRAPEZ predictor	3871	16436	91236
TRAPEZ with GEAR	6206	14480	99246
NEWTON predictor	8281	18105	78823

Table 7.5: Number of iterations

Method	MULTIPLIER	MOSOSCIL	MOSAMPL
Implicit TRAPEZ	100	100	100
TRAPEZ predictor	337,07	113,56	72,56
TRAPEZ with GEAR	225,57	132,32	81,04
Implicit GEAR	100,10	112,34	84,06
GEAR predictor	100,10	103,59	84,06
GEAR with TRAPEZ	225,57	90,793	84,97

Table 7.6: Percentage of of non-convergences

Multiplier

This circuit is a multiplier which originates from article [Dob09]. It is a low-voltage low-power CMOS four-quadrant microwave multiplier. It uses semi-empirical models of transistors of level 3. Because circuit is well designed, the lowest number of non-convergences occurred during transient simulation.

MOS Relaxation Oscillator

It is a very simple circuit of MOS relaxation oscillator, which originates from book [Vla94] (pages 102-103). It presented to show accuracy problems, which may be produced by GEAR method. It uses a simple model of NMOS transistor with threshold voltage VTO, transconductance KP, and gate-source overlap capacitor CGSO.

MOS Amplifier

This circuit is MOS transistor amplifier. It originates from NgSpice testing repository and can be downloaded with the actual version of NgSpice21. It is modeled by MOS transistors with substrate doping NSUB, gate-source overlap capacitor CGSO, gate-drain overlap cap CGDO, surface state density NSS and surface mobility UO, oxide thickness TOX, lateral diffusion LD and critical field exponent for mobility degradation UEXP.

7.4 Chapter Summary

In this chapter, several methods for enhancing the accuracy and reliability of TRAN analysis have been suggested. Achievable precision during computation with floating-point precision numbers is strictly limited by a size of the variable type. It has been demonstrated that because of precision problems caused by the nature of floating point numbers, the total accuracy can be achieved only in very trivial cases. Mostly computation finish regardless on used numeric type with precision around 10^{-16} or less. The only way that can provide results with greater precision is to switch to arbitrary precision numbers. Arbitrary-precision numbers allow to setup any precision, but they are also very demanding from the point of view of performance. The iterative method that increases the precision of analysis of electrical circuit by suppressing errors produced by floating point variables. It can be applied to the end DC simulation or as a final procedure after each time-step of TRAN analysis. Standard LUF followed with a single iteration of NIM with arbitrary precision can definitely improve the accuracy with preserved performance for dense matrices. Although, the method can provide a rapid increase in precision of a result in a case of sparse matrices efficiency of NIM is discussable and will be decreasing with growing sparsity of the matrix. Therefore, the method is applicable only for small circuits (up to 200 nodes) and in cases when the precision of the result is more important than overall performance.

In modern simulators of electronic circuits, it has been replaced by method GEAR but still we can find it implemented as the fail-safe method, which is invoked in situations when default method failed. A combination of two algorithms for numerical integration seems to be a promising solution. The results of simulation done by GEAR and TRAPEZ method together give a good trade-off between the number of non-convergences and precision of analysis.

7.4.1 Research Contributions Summary

- Special usage of Newton Iteration Method (NIM) modified for usage in the core of the electronic circuit simulator to provide a precision boost at the end of computation or at each time step of TRAN analysis. The proposed algorithm can increase precision by factor two. It is best applicable for circuit up to 200 voltage nodes.
- Proposal of usage of parallel computation of two integration methods to improve stability of numerical integration implemented in SPICE simulator

Chapter 8

Conclusion

The first fundamental thesis aim proposes modified computation procedure allowing optimization of simulation algorithms according to the size of the simulated problem. The practical demands on the simulation performance rapidly increased since first simulators for electronic circuits were constructed. It could be mentioned that at the beginning of the nineties as a big circuit was referred one with fifty devices. This number is nowadays more than hundred times higher.

In Chapter 3 is proposed a procedure based on suitable nonstationary BICGStab method supplemented with ILU preconditioner. ILU avoids redundant computations and ensures a constant size of the matrix. In this chapter was demonstrated that usage of this composite iteration method, especially during iteration cycles of NR algorithm, can provide better computation performance for large-scale circuits than can be achieved by standard direct method. It is clearly visible from comparison Figure 3.10 showing the time required for computation with direct method (LUF) and iterative method (BICGStab + ILU).

The following Chapter 4 is continuing with a focus on the simulation of large-scale circuits especially on memory consumption of simulation. It introduces state-of-the-art sparse matrix indexation techniques. The core idea is based on advantageous indexing of sparse matrix non-zero values in the way that only one index is required for each matrix entry. The proposed methods were published in the paper in the [CD15].

The second fundamental thesis aim targets device modeling, device representation during simulation process and their effect on simulation performance. In Chapter 5 is proposed a new, unusual way of implementation of simulation process based on chaining of all entries to the functionals that on call evaluate entire simulation. This mechanism is in the thesis referred as "Functional Chaining mechanism". The first part of this chapter deals with an unusual and efficient usage of the functional programming language Common LISP as a simulation program for electronic circuits denoted as CLASP. Basing on the process it is important to emphasize the fact that not only variables can be dynamically defined, but also functions and even entire program source code can redefine itself in run-time. Particular attention is paid to the introduction of linear device models such as resistors, voltage sources and nonlinear or time dependent such as capacitors, inductors, diodes, and transistors. This implementation was published in a paper in the impacted journal Radioengineering [ČD11] and in the WoS-indexed conference [CD14b]

The next Chapter 6 extends ideas of functional programming and its use in simulation of electronic circuits. It proposes implementation guidelines based on recent updates in object-oriented programming languages standards. As mentioned before, turning the definition of circuit models into self-evaluable functions allows to implement functional chaining mechanism and handle computation in a manner similar to symbolic solvers. Usage of natively functional languages as LISP can on the other hand provide functional definitions natively but because

of very slow (in comparison to C or FORTRAN) computation performance, they are not suitable for demanding numerical computations. The usage of the recent standard of C++ language with support of functional programming is clearly faster solution as can be seen in Figure 6.4. It can also be natively implemented to NgSpice without linking additional foreign interfaces. The chapter proposes C++ reimplementation of CMOS transistor with same manner and functionality as it could be done with the usage functional language LISP. It should be mentioned that usage of functional definition (Functional Chaining Mechanism) allows to implement self-optimization of individual device models with a respect to the current simulation settings, or analysis state by stripping of redundant conditional jumps. Additionally, functional models can obtain circuit variables directly through the memory pointers unlike in standard implementation where this is performed through integer indexes. Both features can improve computation performance as is shown in Table 6.2 and Figure 6.5.

Last but not least thesis aim is simulation accuracy and precision of the results. The first proposed technique is based on Newton Iteration method with arbitrary precision numbers. In Chapter 7, it is demonstrated that because of precision problems caused by the nature of floating point numbers, the total accuracy can be achieved only in very trivial cases. The only way that can provide results with greater precision is to switch to arbitrary precision numbers. Therefore, a new iterative method specially developed for simulation of electronic circuits is presented that increases the precision suppressing errors produced by floating point variables. It can be applied to the end of OP simulation or as a final procedure after each time-step of TRAN analysis. The second part the chapter is dedicated to numerical integration and improvement of its stability. The proposed method is based on a combination of two algorithms for numerical integration. Both can be run simultaneously in parallel and support itself in a case of non-convergences. The results of simulation done by GEAR and TRAPEZ method together give a good trade-off between a number of non-convergences and computation stability. Results were published in the conference [ČD10].

Bibliography

[ABB10] V. Acary, O. Bonnefon, and B. Brogliato, *Time-stepping numerical simulation of switched circuits within the nonsmooth dynamical systems approach*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **29** (2010), no. 7, 1042–1055.

[ABH06] Umut A Acar, Guy E Blelloch, and Robert Harper, *Adaptive functional programming*, ACM Transactions on Programming Languages and Systems (TOPLAS) **28** (2006), no. 6, 990–1034.

[AMM93] Paolo Antognetti, Giuseppe Massobrio, and Guiseppe Massobrio, *Semiconductor device modeling with SPICE*, McGraw-Hill, Inc., 1993.

[BAA+16] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[BB05] M. W. Berry and M. Browne, *Understanding search engines: Mathematical modeling and text retrieval*, Software, Environments and Tools, 2005.

[BBB] Zdeněk Biolek, Dalibor Biolek, and Viera Biolková, *Spice model of memristor with nonlinear dopant drift*, Radioengineering, 210–214.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods, 2nd edition*, SIAM, Philadelphia, PA, 1994.

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, Modern Software Tools in Scientific Computing (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), Birkhäuser Press, 1997, pp. 163–202.

[BI65] Adi Ben-Israel, *An iterative method for computing the generalized inverse of an arbitrary matrix*, Mathematics of Computation (1965), 452–455.

[BIC66] Adi Ben-Israel and Dan Cohen, *On iterative computation of generalized inverses and associated projections*, SIAM Journal on Numerical Analysis **3** (1966), no. 3, 410–419.

[CAT14] Yan Chen, Umut A Acar, and Kanat Tangwongsan, *Functional programming for dynamic and large data with self-adjusting computation*, ACM SIGPLAN Notices **49** (2014), no. 9, 227–240.

[ČD10] David Černỳ and Josef Dobeš, *New approach for enhancing efficiency of computer aided design in circuit simulation*, Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on, IEEE, 2010, pp. 635–638.

[ČD11] ———, *Common LISP as simulation program (CLASP) of electronic circuits.*, Radioengineering **20** (2011), no. 4, 880–889.

[CD14a] D. Cerny and J. Dobes, *Functional programming languages in computer simulation of electronics circuits*, Computational Science and Computational Intelligence (CSCI), 2014 International Conference on, vol. 1, March 2014, pp. 229–234.

[CD14b] David Cerny and Josef Dobes, *Functional programming languages in computer simulation of electronics circuits*, Computational Science and Computational Intelligence (CSCI), 2014 International Conference on, vol. 1, IEEE, 2014, pp. 229–234.

[CD15] ———, *Adaptive sparse matrix indexing technique for simulation of electronic circuits based on λ-calculus*, Circuit Theory and Design (ECCTD), 2015 European Conference on, IEEE, 2015, pp. 1–4.

[CL75] L.O. Chua and P.M. Lin, *Computer aided analysis of electronic circuits: Algorithms and computational techniques*, NJ: Prentice Hall, 1975.

[Coh75] E. Cohen, *Program reference for SPICE2*, Tech. Report UCBIERL M75/520, Univ. of California, Berkeley,, May 1975.

[CRWY15] Xiaoming Chen, Ling Ren, Yu Wang, and Huazhong Yang, *GPU-accelerated sparse LU factorization for circuit simulation with performance modeling*, Parallel and Distributed Systems, IEEE Transactions on **26** (2015), no. 3, 786–795.

[Dav06] T. Davis, *Direct methods for sparse linear systems*, Society for Industrial and Applied Mathematics, 2006.

[DČY11] Josef Dobeš, David Černỳ, and Abhimanyu Yadav, *A more efficient arrangement of the sparse lu factorization for the large-scale circuit analysis*, 2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, IEEE, 2011, pp. 416–421.

[Dea15] Edvin Deadman, *Estimating the condition number of f(a)b*, Numerical Algorithms **70** (2015), no. 2, 287–308.

[Dob02] J. Dobes, *Nonstandard sensitivity analyses in frequency and time domains*, 9th International Conference on Electronics, Circuits and Systems, (2002).

[Dob06] ———, *Efficient procedures for analyzing large-scale RF circuits*, IEEE Dallas/-CAS Workshop on Design, Applications, Integration and Software (2006).

[Dob09] ———, *Advanced types of the sensitivity analysis in frequency and time domains*, Int. J. Electron. Commun. (AEÜ) **63** (2009), 52 – 64.

[DPN10] Timothy A Davis and Ekanathan Palamadai Natarajan, *Algorithm 907: Klu, a direct sparse solver for circuit simulation problems*, ACM Transactions on Mathematical Software (TOMS) **37** (2010), no. 3, 36.

[EBH+15] John W. Eaton, David Bateman, Soren Hauberg, , and Rik Wehbring, *Gnu octave version 4.0.0 manual: a high-level interactive language for numerical computations*, 2015.

[EFB01] Tzilla Elrad, Robert E Filman, and Atef Bader, *Aspect-oriented programming: Introduction*, Communications of the ACM **44** (2001), no. 10, 29–32.

[EK+95] Russ C Eberhart, James Kennedy, et al., *A new optimizer using particle swarm theory*, Proceedings of the sixth international symposium on micro machine and human science, vol. 1, New York, NY, 1995, pp. 39–43.

[FN91] Roland W. Freund and Noël M. Nachtigal, *QMR: a quasi-minimal residual method for non-hermitian linear systems*, Numerische Mathematik **60** (1991), no. 1, 315–339.

[FOP14] D. Ferreira, J.F. Oliveira, and J.C. Pedro, *A novel time-domain CAD technique based on automatic time-slot division for the numerical simulation of highly nonlinear RF circuits*, Microwave Theory and Techniques, IEEE Transactions on **62** (2014), no. 1, 18–27.

[Gol91] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys **23** (1991), no. 1, 5–48.

[GV96] G.H. Golub and C.F. Van Loan, *Matrix computations*, vol. 3, Johns Hopkins Univ Pr, 1996.

[GZ91] P. Gubian and M. Zanella, *Stability properties of integration methods in SPICE transient analysis*, IEEE International Sympoisum on Circuits and Systems (1991).

[HBG71] G. Hachtel, R. Brayton, and F. Gustavson, *The sparse tableau approach to network analysis and design*, IEEE Transactions on Circuit Theory **18** (1971), no. 1, 101–113.

[HC14] Z. Huszka and A. Chakravorty, *Implementation of delay-time-based nonquasi-static bipolar transistor models in circuit simulators*, Electron Devices, IEEE Transactions on **61** (2014), no. 8, 3004–3006.

[Hig02] N.J. Higham, *Accuracy and stability of numerical algorithms*, Society for Industrial Mathematics, 2002.

[HNW08] E. Hairer, S. P. Norsett, and G. Wanner, *Solving ordinary differential equations I*, Springer, 2008.

[HRB75a] C. Ho, A. E. Ruehli, and P. A. Brennan, *The modified nodal approach to network analysis*, IEEE Transactions on Circuits and Systems **CAS-22** (1975), 504–509.

[HRB75b] Chung-Wen Ho, A. Ruehli, and P. Brennan, *The modified nodal approach to network analysis*, IEEE Transactions on Circuits and Systems **22** (1975), no. 6, 504–509.

[HT00] Nicholas J. Higham and Françoise Tisseur, *A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra*, SIAM J. Matrix Anal. Appl **21** (2000), 1185–1201.

[JLW15] Song-lei Jian, Kai Lu, and Xiao-ping Wang, *A survey on concepts and the state of the art of functional programming languages*, Systems and Computer Technology: Proceedings of the 2014 Internaional Symposium on Systmes and Computer technology,(ISSCT 2014), Shanghai, China, 15-17 November 2014, CRC Press, 2015, p. 71.

[KA00] M. A. Kleiner and M. N. Afsar, *Determining the steady-state responses in RF circuits using GMRES, CGS and BICGSTAB solution in sSPICE for linux*, Microwave Symposium Digest. 2000 IEEE MTT-S International, vol. 1, June 2000, pp. 87–90 vol.1.

[KA01] M. A. Kleiner and R. N. Afsar, *Steady-state determination for RF circuits using krylov-subspace methods in SPICE*, Microwave Symposium Digest, 2001 IEEE MTT-S International, vol. 3, May 2001, pp. 2091–2094 vol.3.

[KD12] N. Kapre and A. DeHon, *SPICE2 : Spatial processors interconnected for concurrent execution for accelerating the SPICE circuit simulator using an FPGA*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **31** (2012), no. 1, 9–22.

[KLS13] Andrey Kuzmin, Mathieu Luisier, and Olaf Schenk, *Fast methods for computing selected elements of the green's function in massively parallel nanoelectronic device simulations*, European Conference on Parallel Processing, Springer, 2013, pp. 533–544.

[KSG15] R. Kumawat, V. Sahula, and M.S. Gaur, *Probabilistic model for nanocell reliability evaluation in presence of transient errors*, Computers Digital Techniques, IET **9** (2015), no. 4, 213–220.

[KSV86] Kenneth S Kundert and Alberto Sangiovanni-Vincentelli, *Simulation of nonlinear circuits in the frequency domain*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **5** (1986), no. 4, 521–535.

[Li05] Xiaoye S. Li, *An overview of SuperLU: Algorithms, implementation, and user interface*, TOMS **31** (2005), no. 3, 302–325.

[LS05] Zhao Li and C. J. R. Shi, *An efficiently preconditioned GMRES method for fast parasitic-sensitive deep-submicron vlsi circuit simulation*, Design, Automation and Test in Europe, March 2005, pp. 752–757 Vol. 2.

[LS09] George F Luger and William A Stubblefield, *Ai algorithms, data structures, and idioms in prolog, lisp, and java*, Pearson Addison-Wesley, 2009.

[McC65] J. McCarthy, *Lisp 1.5 programmer's manual*, The MIT Press, 1965.

[McC88] W. J. McCalla, *Fundamentals of computer-aided circuit simulation*, Boston:Kluwer Academic, 1988.

[MG09] J. Theiler M. Galassi, J. Davies, *Gnu scientific library reference manual*, Network Theory Ltd, 2009.

[MIT01] PA Mawby, PM Igic, and MS Towers, *Physically based compact device models for circuit modelling applications*, Microelectronics journal **32** (2001), no. 5, 433–447.

[Nag75] L. W. Nagel, *SPICE2: A computer program to simulate semiconductor circuits*, Ph.D. thesis, Uni. of California, Berkeley, May 1975.

[Not93] Yvan Notay, *On the convergence rate of the conjugate gradients in presence of rounding errors*, Numerische Mathematik **65** (1993), no. 1, 301–317.

[NV11] Paolo Nenzi and Holger Vogt, *Ngspice users manual version 23*, 2011.

[NV16] P. Nenzi and H. Vogt, *Ngspice users manual 26*, 2016.

[NZ15] T. Nechma and M. Zwolinski, *Parallel sparse matrix solution for circuit simulation on FPGAs*, IEEE Transactions on Computers **64** (2015), no. 4, 1090–1103.

[Ped84] D.O Pederson, *A historical review of circuit simulation,*, IEEE TRansactions and CIrcuits and Systems **31** (1984), 103–111.

[Roj15] Raúl Rojas, *A tutorial introduction to the lambda calculus*, arXiv preprint arXiv:1503.09060 (2015).

[Saa99] Y. Saad, *Ilum: A multi-elimination ilu preconditioner for general sparse matrices*, SIAM J. Sci. Comput **17** (1999), 830–847.

[Saa03] Y. Saad, *Iterative methods for sparse linear systems*, second ed., Society for Industrial and Applied Mathematics, 2003.

[Sei06] Peter Seibel, *Practical common lisp*, Apress, 2006.

[Sha93] Z.Q. Shang, *The convergence problem in SPICE*, IEE Colloquium on SPICE: Surviving Problems in Circuit Evaluation (1993).

[SM03] E. Süli and D.F. Mayers, *An introduction to numerical analysis*, Cambridge Univ Pr, 2003.

[SS86] Youcef Saad and Martin H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing **7** (1986), no. 3, 856–869.

[SSB14] M. Steiner, G. Siefer, and A.W. Bett, *SPICE network simulation to calculate thermal runaway in III-V solar cells in CPV modules*, Photovoltaics, IEEE Journal of **4** (2014), no. 2, 749–754.

[TB09] T. Tuma and A. Burmen, *Circuit simulation with SPICE OPUS*, Birkhauser Boston, 2009.

[Tew73] Reginald P Tewarson, *Sparse matrices*, Academic Press, 1973.

[Tis99] Françoise Tisseur, *Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems*, SIAM J. Matrix Anal. Appl **22** (1999), 1038–1057.

[TW67] William F Tinney and John W Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proceedings of the IEEE **55** (1967), no. 11, 1801–1809.

[VC93] A. Vladimirescu and J.-J. Charlot, *Challenges of mos analog circuit simulation with SPICE*, IEE Colloquium on SPICE: Surviving Problems in Circuit Evaluation, (1993).

[vdV92] H. A. van der Vorst, *BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput. **13** (1992), no. 2, 631–644.

[Vla94] A. Vladimirescu, *The spice book*, J. Wiley and Sons, 1994.

[Wad98] Philip Wadler, *Why no one uses functional languages*, SIGPLAN Not. **33** (1998), no. 8, 23–27.

[WJM+73] W. Weeks, A. Jimenez, G. Mahoney, D. Mehta, H. Qassemzadeh, and T. Scott, *Algorithms for astap–a network-analysis program*, IEEE Transactions on Circuit Theory **20** (1973), no. 6, 628–634.

[WWTK14] Oi-Ying Wong, Hei Wong, Wing-Shan Tam, and Chi-Wah Kok, *Dynamic analysis of two-phase switched-capacitor DC-DC converters*, Power Electronics, IEEE Transactions on **29** (2014), no. 1, 302–317.

[YLF+13] Shanglin Yang, Shibin Liu, Wenguang Feng, Bo Guo, Xiaowei Hou, and Juping Li, *SPICE circuit model of voltage excitation fluxgate sensor*, Science, Measurement Technology, IET **7** (2013), no. 3, 145–150.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, *Spark: Cluster computing with working sets.*, HotCloud **10** (2010), 10–10.

List of Author's Works
David Černý

List of Candidate's Works Relating to the Doctoral Thesis

Authors' contributions are assumed to be equal.

Papers in Impacted Journals

[A1] D. Černý and J. Dobeš. Composing Scalable Solver for Simulation of Electronic Circuits in SPICE. *AEÜ - International Journal of Electronics and Communications*. 2016, 14 pages. ISSN 1434-8411. Under review.

[A2] D. Černý and J. Dobeš. Functional Chaining Mechanism Allowing Self-Optimized Device Models in Electronic Circuit Simulation. *IET Circuits, Devices & Systems*. 2016, 8 pages. ISSN 1751-858X. Under review.

[A3] J. Dobeš, J. Míchal, J. Popp, D. Černý, M. Grábner, F. Vejražka, J. Kakona and Š. Matějka. Precise Characterization and Multiobjective Optimization of Low Noise Amplifiers. *Radioengineering*. 24 no. 3, 2015, pp. 670-680. ISSN 1210-2512. Candidate's contribution is approximately equal to the one for a uniform distribution.

[A4] D. Černý, J. Dobeš. Common LISP as Simulation Program (CLASP) of Electronic Circuits. *Radioengineering*. 20, no. 4, 2011, pp. 880-889. ISSN 1210-2512.

Papers Excerpted in Web of Science

[B1] D. Černý and J. Dobeš. Functional Programming Languages in Computer Simulation of Electronics Circuits. In *Proceedings of The 2014 International Conference on Computational Science and Computational Intelligence*. Las Vegas, Los Alamitos: IEEE Computer Society. 2014, pp. 229-234. ISBN 978-1-4799-3009-8.

[B2] J. Dobeš, V. Paňko, S. Banáš, and David Černý. An improved model of high-voltage power LDMOSFET and its usage in multi-objective optimization of radio-frequency amplifiers. In *Proceedings of the 14th Workshop on Control and Modeling for Power Electronics (COMPEL)*. pp. 1-5. IEEE, 2013. ISSN 2151-0997. ISBN 978-1-4673-4916-1.

[B3] J. Dobeš, V. Paňko, D. Černý, and J. Divin. A flexible algorithm for solving systems of circuit differential-algebraic equations with integrated Fortran 95 compiler for modeling. In *IEEE AFRICON 2013*. Piscataway: IEEE. 2013, pp. 1-5. ISSN 2153-0025. ISBN 978-1-4673-5943-6.

[B4] J. Dobeš, A. Yadav, and D. Černý. Efficient algorithm for solving systems of circuit differential-algebraic equations with reliable divergence suppression in DC and time domains. In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*. Piscataway: IEEE, 2011, p. 1-4. ISSN 1548-3746. ISBN 978-1-61284-857-0.

[B5] J. Míchal, J. Dobeš, and D. Černý. Multiobjective optimization with an asymptotically uniform coverage of Pareto front. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pp. 2912-2915. IEEE, 2010. ISBN: 978-1-4244-5308-5.

Papers in Reviewed Journals

[C1] J. Dobeš, D. Biolek, J. Míchal, D. Černý, L. Sláma. Precise Algorithms for Pole-Zero Analysis in Electronic Circuit Design. *International Journal of Circuits, Systems and Signal Processing*. Volume 5, Issue 3, 2011, pp. 228-236. ISSN: 19984464. (In Scopus)

Other Publications

[D1] D. Černý, J. Dobeš. An Efficient Procedure for Transient Analysis of Electronic Circuits with Increased Precision. In *Proceedings of the 20th International Conference on Circuits, Systems, Communications and Computers* 2016, Corfu. In print.

[D2] D. Černý, J. Dobeš. Adaptive sparse matrix indexing technique for simulation of electronic circuits based on λ-calculus. In *Proceedings of the 22nd European Conference on Circuit Theory and Design (ECCTD)*. Trondheim, Piscataway: IEEE. 2015, pp. 1-4. ISBN 978-1-4799-9877-7. (In Scopus)

[D3] J. Dobeš, D. Černý, F. Vejražka and V. Navrátil. Comparing the Steady-State Procedures Based on Epsilon-Algorithm and Sensitivity Analysis. In *Proceedings of the 22nd IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. Cairo, Monterey: IEEE Circuits and Systems Society. 2015, pp. 601-604. ISBN 978-1-4799-2451-6. (In Scopus)

[D4] D. Černý and J. Dobeš. New Sparse Matrix Ordering Techniques for Computer Simulation Of Electronics Circuits. In *Proceedings of Recent Advances in Communications, Circuits and Technological Innovations*. 2012. ISBN: 978-1-61804-138-8

[D5] J. Dobeš, D. Černý, and D. Biolek. Efficient procedure for solving circuit algebraic-differential equations with modified sparse LU factorization improving fill-in suppression. In *Proceedings of the 20th European Conference on Circuit Theory and Design (ECCTD)*. pp. 689-692. IEEE, 2011. ISBN: 978-1-4577-0617-2 (In Scopus)

[D6] J. Dobeš, D. Černý, and A. Yadav. A more efficient arrangement of the sparse LU factorization for the large-scale circuit analysis. In *Proceedings of the 19th International Conference on VLSI and System-on-Chip (VLSI-SOC)*, pp. 416-421. IEEE, 2011. ISBN: 978-1-4577-0170-2. (In Scopus)

[D7] D. Černý, and J. Dobeš. New approach for enhancing efficiency of computer aided design in circuit simulation. In *Electronics, Circuits, and Systems (ICECS)*, 2010 17th IEEE International Conference on, pp. 635-638. IEEE, 2010. (In Scopus)

[D8] J. Dobeš, D. Černý, L. Sláma, D. Biolek, and J. Míchal. New methods for improving the pole-zero analysis accuracy. In *Proceedings of the 2010 International Conference on Mathematical Models for Engineering Science*, pp. 155-158. 2010. (In Scopus)

Response and Reviews

Citations

Paper [A3]

D. Černý, J. Dobeš. Common LISP as Simulation Program (CLASP) of Electronic Circuits. *Radioengineering*. 20, no. 4. 2011. ISSN 1210-2512.

was cited in:

[G1] C. Lei. SIM-DSP: A DSP-Enhanced CAD Platform for Signal Integrity Macromodeling and Simulation. In *Radioengineering*, 2014. ISSN 1210-2512.

[G2] D. Mirkovic, P. Petkovic, V. Litovski. Experience in Using Open Command Environment for Analysis in Education. In *Proceedings of 1st International Conference on Electrical, Electronic and Computing Engineering (IcETRAN 2014)*. pp. 1-5. Vrnjacka Banja, Serbia, 2014, ISBN 978-86-80509-70-9.

Paper [B1]

D. Černý, J. Dobeš. Functional Programming Languages in Computer Simulation of Electronics Circuits. In *Proceedings of 2014 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, March 2014, ISBN: 978-1-4799-3010-4.

was cited in:

[G3] J.G.H. Pineda, A. Puertas-Gonzalez. La programacion funcional y las arquitecturas multicore: estado del arte. In *Ingenio Magno 6* , 124-136, 2016.

Paper [B4]

J. Dobeš, A. Yadav, and D. Černý. Efficient algorithm for solving systems of circuit differential-algebraic equations with reliable divergence suppression in DC and time domains. In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*. Piscataway: IEEE, 2011, p. 1-4. ISSN 1548-3746. ISBN 978-1-61284-857-0.

was cited in:

[G4] J. Ogrodzki. Analog Approach to Mixed Analog-Digital Circuit Simulation. In *Photonics Applications In Astronomy, Communications, Industry, And High-energy Physics Experiments* 2013. ISSN 0277-786X.

[G5] Y. Lin, and E. Gad. Formulation of the Obreshkov-Based Transient Circuit Simulator in the Presence of Nonlinear Memory Elements. In *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*. 2015. ISSN 0278-0070.

[G6] Y. Lin. Improvements in Obreshkov-based High-Order Circuit Simulation Method. In *Doctoral dissertation*. Université d'Ottawa/University of Ottawa, 2015.

Paper [D5]

J. Dobeš, D. Černý, and D. Biolek. Efficient procedure for solving circuit algebraic-differential equations with modified sparse LU factorization improving fill-in suppression. In *Proceedings of the 20th European Conference on Circuit Theory and Design (ECCTD)*. pp. 689-692. IEEE, 2011. ISBN: 978-1-4577-0617-2

was cited in:

[G7] EBB Costa, FT Matsunaga, JD Brancher: Analise de escalabilidade e eficiencia da fatorac ao LU usando CPU x GPU. Biblioteca Digital Brasileira de Computacao.

[G8] E. B. B. Costa, F. T. Matsunaga, J. Duilio. Análise Comparativa da Fatoracao LU: Estudo de Caso OpenCL CPU x GPU. *11o Simposio Argentino de Investigacion Operativa*. ISSN: 1850-2865

Paper [D6]

J. Dobeš, D. Černý, A. Yadav. A more efficient arrangement of the sparse LU factorization for the large-scale circuit analysis. In *Proceedings of the 19th International Conference on VLSI and System-on-Chip*, pp. 416-421. IEEE, 2011. ISBN: 978-1-4577-0170-2. ISBN 978-1-4577-0170-2.

was cited in:

[G9] J. Ogrodzki. Analog Approach to Mixed Analog-Digital Circuit Simulation. In *Photonics Applications In Astronomy, Communications, Industry, And High-energy Physics Experiments* 2013. ISSN 0277-786X

Paper [B5]

J. Míchal, J. Dobeš, and D. Černý. Multiobjective optimization with an asymptotically uniform coverage of Pareto front. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. pp. 2912-2915. IEEE, 2010. ISBN: 978-1-4244-5308-5.

was cited in:

[G10] Y. Li. MOMCMC: An efficient Monte Carlo method for multi-objective sampling over real parameter space. In *Computers and Mathematics with Applications*. pp. 3542-3556. 2012. ISSN 0898-1221.

[G11] F. Karimi, S. Lotfi. Solving Multi-Objective Problems using SPEA2 and Tabu Search. In *2014 Iranian Conference On Intelligent Systems (ICIS)*, 2014. ISBN 978-1-4799-3351-8.

Short Curriculum Vitae
David Černý

PERSONAL INFORMATION

Name	David
Surname	Černý
Address	Ořechová 194, 53352 Staré Hradiště, Czech Republic
Email	david.cerny@senman.cz

EDUCATION

2009–Present **PhD Student in Radioelectronics**
Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Radio Engineering

2007–2009 **Master's degree in Radioelectronics**
Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Radio Engineering
Thesis: Measurement of electromagnetic field in living cells in high frequency band

2004–2007 **Bachelor's degree in Electronics and Communications**
Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Radio Engineering
Thesis: Measurement of electromagnetic field in living cells in high frequency band

EXPERIENCE

2012–Present **CEO, Project Leader**
Senman s.r.o. Argentinská 38, Praha 7, CZ

2011–2012 **R&D Contracter**
Retia a.s. Pražská 341, 530 02, Pardubice

2006–2010 **Software Developer**
Konektel a.s. Prazska 152 530 06 Pardubice

INTERSHIP

2010 **International Working Intership**
Smarterphone AS (previously Kvaleberg) Oscars gate 50, NO-0258 Oslo, Norway, www.smarterphone.com

2009 **International Working Intership**
CANON INC. Headquarters, 30-2, Shimomaruko 3-chome, Ohta-ku, Tokyo 146-8501, Japan, www.canon.com

LANGUAGES

English Fluent

SKILLS

Prog. Languages	C, C++, C#, Java, Common Lisp, Scheme, Haskel, SQL, Python, PHP, Java Script
Sim. Languages	Spice, Hspice, NgSpice, Spectre
Other Tools	Octave, Matlab, MathCAD, ICCAP

GROUPS AND ASSOCIATIONS

2011-2014	**IAESTE Czech Republic**
	National Secretary of IAESTE Czech Republic
2009-2011	**IAESTE CTU Prague**
	Head of local committee IAESTE CTU Prague
2006-2009	**IAESTE CTU Prague**
	Member of IAESTE CTU Prague

List of Notation

Symbol	Meaning
ATSC	Automatic time slep Control
BICG	BiConjugate Gradient
BICGStab	BiConjugate Gradient Stabilized
BLAS	Basic Linear Algebra Subprograms
CG	Conjugate Gradient Method
CGS	Conjugate Gradient Squared Method
CLASP	Common LISP as Simulation Program
EFPSS	Enhanced Full Profile Skyline Storage
FAI	Foreign Array Interface
FDM	Functional Device Modeling
FFI	Foreign Function Interface
FPSS	Full Profile Skyline Storage
GEAR	Gear's Numerical Integration Method
GMRES	Generalized Minimal Residual
GNU GSL	Numerical C and C++ library providing mathematical
GNU Octave	High-level computations library.
GS	The Gauss-Seidel Method
ILU	Incomplete LU factorization
JACOBI	The Jacobi Method
LAPACK	Linear Algebra Package
LISP	The LISt Processing language
LTE	Local Truncation Error Algorithm
MNA	Modified Nodal Analysis
MNF	The Modified Nodal Formulation
MPSLS	Modified Profile-In Skyline Storage
NFI	Native Implementation of functional device models
NgSpice	SPICE continuation based on Spice3f5, Xspice and Cider1b1
NI	Newton Itegration
NIM	Newton Iteration method
NR	Newton-Raphson iteration method

Symbol	Meaning
OP	Operating Point
PETSC	Portable, Extensible Toolkit for Scientific computation
PSO	Particle Swarm Optimization
QMR	Quasi-Minimal Residual
RHS	Right Hand Side
SMIT	Standard Sparse Matrix Indexing Technique
SOR	The Successive Overrelaxation Method
SPICE	Simulation Program with Integrated Circuit Emphasis
TRAN	Transient Analysis
TRAPEZ	Trapezoidal Numerical Integration Method

Appendix A

CLASP LU solver

Here is CLASP implementation of the sparse LU solver (CLASP LU), which had been used for comparison of different LU solvers in benchmark section.

```
;; Evaluation of equation Ax = b
(DEFUN SOLVE-SOLVE-SPARSE-TABLE ()
  (LET ((TABLE-SIZE (SIZE-OF-FEATURE-DATABASE)))
    (LU-FACTORIZE TABLE-SIZE)
    (FORWARD-LU-SUBSTITUTION TABLE-SIZE)
    (BACKWARD-LU-SUBSTITUION TABLE-SIZE)))

;; Forward substitution
(DEFUN FORWARD-LU-SUBSTITUION (TABLE-SIZE)
  (LOOP FOR LINE FROM 1 TO TABLE-SIZE DO
    (FORWARD-FACTOR-LIST-VALUES LINE
      (REMOVE LINE (GET-ALL-LINE-INDEXES LINE) :TEST #'<= ))))

;; Forward factorize division
(DEFUN DIVIDE-FORWARD-FACTOR-ONE-VALUE (LINE VALUE)
  (SET-RHS-VALUE LINE
    (/ VALUE (GET-ONE-VALUE LINE LINE))))

;; Forward factorization
(DEFUN FORWARD-FACTOR-LIST-VALUES (LINE INDEXES)
  (DIVIDE-FORWARD-FACTOR-ONE-VALUE LINE
    (+ (GET-RHS-VALUE LINE)
       (LOOP FOR INDEX IN INDEXES SUM
         (* (GET-ONE-VALUE LINE INDEX) (GET-RHS-VALUE INDEX) -1 )))))

;; Backward substitution
(DEFUN BACKWARD-LU-SUBSTITUION (TABLE-SIZE)
  (LOOP FOR LINE FROM TABLE-SIZE DOWNTO 1 DO
    (BACKWARD-FACTOR-LIST-VALUES LINE
      (REMOVE LINE (GET-ALL-LINE-INDEXES LINE) :TEST #'>= ))))

;; Backward factorization
(DEFUN BACKWARD-FACTOR-LIST-VALUES (LINE INDEXES)
  (SET-RHS-VALUE LINE
    (+ (GET-RHS-VALUE LINE)
       (LOOP FOR INDEX IN INDEXES SUM
         (* (GET-ONE-VALUE LINE INDEX) (GET-RHS-VALUE INDEX) -1 )))))

;; Main LU factorization function
(DEFUN LU-FACTORIZE (TABLE-SIZE)
(LOOP FOR MAIN-INDICIE-POS FROM 1 TO (- TABLE-SIZE 1) DO
  (LET ((MAIN-LINE-INDEXES
          (REMOVE MAIN-INDICIE-POS
            (GET-ALL-LINE-INDEXES MAIN-INDICIE-POS) :TEST #'>= ))
        (INDICIE-VALUE
          (GET-ONE-VALUE MAIN-INDICIE-POS MAIN-INDICIE-POS)))
    (DIVIDE-LU-FACTOR-LIST-VALUES
```

```
        MAIN-INDICIE-POS
        MAIN-LINE-INDEXES
        INDICIE-VALUE)
      (LOOP FOR AUX-INDICIE-POS FROM (+ MAIN-INDICIE-POS 1)
            TO TABLE-SIZE
            DO
        (IF (GET-ONE-VALUE AUX-INDICIE-POS MAIN-INDICIE-POS)
          (LET ((AUX-LINE-INDEXES
                  (REMOVE MAIN-INDICIE-POS
                    (GET-ALL-LINE-INDEXES AUX-INDICIE-POS) :TEST #'>= ))
                (INDICIE-VALUE
                  (GET-ONE-VALUE AUX-INDICIE-POS MAIN-INDICIE-POS)))
                (MINUS-LU-FACTOR-LIST-VALUES
                  MAIN-INDICIE-POS AUX-INDICIE-POS
                  (INTERSECTION MAIN-LINE-INDEXES AUX-LINE-INDEXES)
                    INDICIE-VALUE)
                (MINUS-ZERO-LU-FACTOR-LIST-VALUES
                  MAIN-INDICIE-POS AUX-INDICIE-POS
                  (SET-DIFFERENCE MAIN-LINE-INDEXES AUX-LINE-INDEXES)
                  INDICIE-VALUE))))))
                    T)

;; LU subtraction functions
(DEFUN MINUS-LU-FACTOR-LIST-VALUES (MAIN-LINE RES-LINE INDEXES VALUE)
  (MAPCAR #'(LAMBDA (X)
    (MINUS-LU-FACTOR-ONE-VALUE MAIN-LINE RES-LINE X VALUE)) INDEXES))

(DEFUN MINUS-LU-FACTOR-ONE-VALUE (MAIN-LINE RES-LINE INDEX VALUE)
  (SET-ONE-VALUE RES-LINE INDEX
    (- (GET-ONE-VALUE RES-LINE INDEX) (* VALUE (GET-ONE-VALUE MAIN-LINE INDEX)))))

;; LU subtraction from zero element functions
(DEFUN MINUS-ZERO-LU-FACTOR-LIST-VALUES (MAIN-LINE RES-LINE INDEXES VALUE)
  (MAPCAR #'(LAMBDA (X)
    (MINUS-ZERO-LU-FACTOR-ONE-VALUE MAIN-LINE RES-LINE X VALUE)) INDEXES))

(DEFUN MINUS-ZERO-LU-FACTOR-ONE-VALUE (MAIN-LINE RES-LINE INDEX VALUE)
  (SET-NEW-ONE-VALUE RES-LINE INDEX
    (* VALUE (GET-ONE-VALUE MAIN-LINE INDEX) -1 )))

;; LU division functions
(DEFUN DIVIDE-LU-FACTOR-LIST-VALUES (RES-LINE INDEXES VALUE)
  (MAPCAR #'(LAMBDA (X)
    (DIVIDE-LU-FACTOR-ONE-VALUE RES-LINE X VALUE)) INDEXES))

(DEFUN DIVIDE-LU-FACTOR-ONE-VALUE (RES-LINE INDEX VALUE)
  (SET-ONE-VALUE RES-LINE INDEX
    (/ (GET-ONE-VALUE RES-LINE INDEX) VALUE)))
```

Appendix B

Zener diode model

```
;; Forward and Backward current definition
(DEFMETHOD DIODE-ZENER-CURRENT ((DIODE CLASS-ZENER) V+ V-)
  #'(LAMBDA ()
      (COND
        ((< (- (EVAL V+) (EVAL V-)) (- (BV DIODE)))
          (* (- (IS DIODE))
             (- (EXP (/
                        (+ (- (EVAL V+) (EVAL V-))
                           (BV DIODE))
                        -26E-3))
                (EXP (/
                        (- (EVAL V+) (EVAL V-))
                        26E-3)))))
        (T
          (* (IS DIODE)
             (- (EXP
                   (/ (- (EVAL V+) (EVAL V-))
                      26E-3))
                1))))))

;; Derivative of forward and backward current with respect to V+
(DEFMETHOD DIODE-SIMPLE-ZENER-CURRENT-DV+ ((DIODE CLASS-DIODE-SIMPLE-ZENER) V+ V-)
  #'(LAMBDA ()
      (COND
        ((< (- (EVAL V+) (EVAL V-)) (- (BV DIODE)))
          (*
            (/ (IS DIODE) 26E-3)
            (+
              (EXP (/ (+ (- (EVAL V+) (EVAL V-)) (BV DIODE)) -26E-3))
              (EXP (/  (- (EVAL V+) (EVAL V-))  26E-3)))))
        (T (* (EXP (/ (- (EVAL V+) (EVAL V-)) 26E-3))  (/ (IS DIODE)  26E-3))))))

;; Derivative of forward and backward current with respect to V-
(DEFMETHOD DIODE-SIMPLE-ZENER-CURRENT-DV- ((DIODE CLASS-DIODE-SIMPLE-ZENER) V+ V-)
  #'(LAMBDA ()
      (COND
        ((< (- (EVAL V+) (EVAL V-)) (- (BV DIODE)))
          (*
            (/ (IS DIODE) -26E-3)
            (+
              (EXP (/ (+ (- (EVAL V+) (EVAL V-)) (BV DIODE)) -26E-3))
              (EXP (/  (- (EVAL V+) (EVAL V-))  26E-3)))))
        (T (* (EXP (/ (- (EVAL V+) (EVAL V-)) 26E-3))  (/ (IS DIODE)  26E-3) )))))
```

Appendix C

Thermistor model

```
;; Thermistor current definition
(DEFMETHOD THERMISTOR-CURRENT ((THERMISTOR CLASS-THERMISTOR) V+ V-)
  #'(LAMBDA ()
    (/ (-   (EVAL V+) (EVAL V-))
       (EXP (- (EXPT (- (EXPT   (+ (EXPT (/ (THERM-B THERMISTOR)
                                            (* 3 (THERM-C THERMISTOR)))
                                         3)
                                   (EXPT (/
                                            (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                                            (* 16 (THERM-C THERMISTOR))) 2))
                                1/2)
                             (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                              (* 2 (THERM-C THERMISTOR))))
                       1/3)
                    (EXPT
                      (+
                        (EXPT
                          (+
                            (EXPT (/ (THERM-B THERMISTOR)
                                     (* 3 (THERM-C THERMISTOR))) 3)
                            (EXPT (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                                     (* 16 (THERM-C THERMISTOR))) 2))
                          1/2)
                        (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                         (* 2 (THERM-C THERMISTOR))))
                      1/3))))))

;; Derivative of thermistor current with respect to V+
(DEFMETHOD THERMISTOR-CURRENT-DV+ ((THERMISTOR CLASS-THERMISTOR) V+ V-)
  #'(LAMBDA ()
    (/ 1
      (EXP
        (-
          (EXPT
            (-
              (EXPT
                (+
                  (EXPT (/ (THERM-B THERMISTOR)
                           (* 3 (THERM-C THERMISTOR))) 3)
                  (EXPT (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                           (* 16 (THERM-C THERMISTOR))) 2))
                1/2)
              (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
               (* 2 (THERM-C THERMISTOR))))
            1/3)
          (EXPT
            (+
              (EXPT
                (+
```

```
                    (EXPT (/ (THERM-B THERMISTOR)
                             (* 3 (THERM-C THERMISTOR))) 3)
                    (EXPT (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                             (* 16 (THERM-C THERMISTOR))) 2))
                  1/2)
                (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                   (* 2 (THERM-C THERMISTOR))))
              1/3))))))

;; Derivative of thermistor current with respect to V-
(DEFMETHOD THERMISTOR-CURRENT-DV- ((THERMISTOR CLASS-THERMISTOR) V+ V-)
  #'(LAMBDA ()
    (/ -1
      (EXP
        (-
          (EXPT
            (-
              (EXPT
                (+
                  (EXPT (/ (THERM-B THERMISTOR)
                           (* 3 (THERM-C THERMISTOR))) 3)
                  (EXPT (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                           (* 16 (THERM-C THERMISTOR))) 2))
                1/2)
              (/ (- (A THERMISTOR) (/ 1 *TEMPERATURE*))
                 (* 2 (THERM-C THERMISTOR))))
            1/3)
          (EXPT
            (+
              (EXPT
                (+
                  (EXPT (/ (THERM-B THERMISTOR)
                           (* 3 (THERM-C THERMISTOR))) 3)
                  (EXPT (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                           (* 16 (THERM-C THERMISTOR))) 2))
                1/2)
              (/ (- (THERM-A THERMISTOR) (/ 1 *TEMPERATURE*))
                 (* 2 (THERM-C THERMISTOR))))
            1/3))))))
```

Appendix D

Particle SWARM optimization algorithm

```
;; Particle swarm algorithm
(DEFMETHOD SOLVER-PARTICLE-SWARM ((M CLASS-MATRIX-SYSTEM)
                                  (VAR-ARR CLASS-VARIABLES)
                                  I EPSILON MAX-GENERATIONS POP-SIZE C1 C2 SPACE-SIZE)
;; Inicialization of algortihm
  (LET (
    (SUB-ITER-LOOP POP-SIZE)
    (LINEAR-MATRICES
      (GRID:MAP-N-GRIDS
            :SOURCES
              (LIST
                (LIST (GET-SUB-G-ARRAY M 1 1 I I) NIL)
                (LIST (GET-SUB-E-ARRAY M 1 1 I I) NIL))
            :COMBINATION-FUNCTION (LAMBDA (A B ) (+ A B))))
    (POP-POS-MATRIX  (GRID:MAKE-FOREIGN-ARRAY 'DOUBLE-FLOAT
                          :DIMENSIONS (LIST POP-SIZE I )
                          :INITIAL-ELEMENT 0.0D0))
    (BEST-POS-MATRIX  (GRID:MAKE-FOREIGN-ARRAY 'DOUBLE-FLOAT
                          :DIMENSIONS (LIST POP-SIZE I )
                          :INITIAL-ELEMENT 0.0D0))
    (RESIDUAL-BEST-POS (MAKE-ARRAY POP-SIZE))
    (POP-VELOCITY-MATRIX  (GRID:MAKE-FOREIGN-ARRAY 'DOUBLE-FLOAT
                              :DIMENSIONS (LIST POP-SIZE I )
                              :INITIAL-ELEMENT 0.0D0))
    (RESIDUAL-BEST-POP 0)
    (BEST-POP-VECTOR  (GRID:MAKE-FOREIGN-ARRAY 'DOUBLE-FLOAT
                          :DIMENSIONS (LIST I )
                          :INITIAL-ELEMENT 0.0D0))
;; Random generator
    (RNG (GSL:MAKE-RANDOM-NUMBER-GENERATOR GSL:+MT19937+
      (RANDOM POP-SIZE))))

;; Getting the first generation of particles
      (ITER:ITER (ITER:FOR COL FROM 0 BELOW I)
       (SETF (GRID:COLUMN POP-POS-MATRIX COL)
       (GRID:MAKE-FOREIGN-ARRAY 'DOUBLE-FLOAT
            :DIMENSIONS POP-SIZE
            :INITIAL-CONTENTS
          (LOOP FOR I FROM 0 BELOW POP-SIZE COLLECT
            (*  (- (GSL:SAMPLE RNG :UNIFORM) 0.5))))))

;; Main iteration algorithm
    (LOOP FOR ITER-LOOP FROM 0
          WHILE (< ITER-LOOP MAX-GENERATIONS)
          DO
       (LOOP FOR PERS FROM 0 BELOW POP-SIZE
             DO
         (ITER:ITER (ITER:FOR K FROM 1 BELOW (SIZE M))
```

```
                (SET (GRID:GREF (STACK M) K)
                  (GRID:GREF POP-POS-MATRIX PERS (- K 1)  )))
          (LET (
            (FITNESS-VECTOR
              (GRID:MAP-N-GRIDS
                    :SOURCES
                      (LIST
                        (LIST ;; RHS equations
                          (GRID:MAP-GRID
                                :SOURCE (GET-SUB-RHS-EQUATIONS-VECTOR M 1 I)
                                :ELEMENT-FUNCTION (LAMBDA (X)
                                  (COERCE (APPLY #'+ (MAPCAR #'FUNCALL X))
                                    'DOUBLE-FLOAT))) NIL)
                        (LIST ;; RHS values
                          (GRID:MAP-GRID
                                :SOURCE (GET-SUB-RHS-NUMBER-VECTOR M 1 I)
                                :ELEMENT-FUNCTION (LAMBDA (X)
                                  (COERCE X 'DOUBLE-FLOAT))) NIL)
                        (LIST ;; linear matrix product
                          (GSLL:MATRIX-PRODUCT
                            LINEAR-MATRICES
                            (GRID:MAP-GRID
                                  :SOURCE (GRID:ROW POP-POS-MATRIX PERS))) NIL)
                        (LIST ;equations evaluation
                          (GRID:MAP-GRID
                                :SOURCE (GET-SUB-EQUATIONS-VECTOR M 1 (SIZE M))
                                :ELEMENT-FUNCTION (LAMBDA (X)
                                  (COERCE (APPLY #'+ (MAPCAR #'FUNCALL X))
                                    'DOUBLE-FLOAT))) NIL))
                    :COMBINATION-FUNCTION (LAMBDA (A B C D)
                      (- (+ A B) (+ C D )))
                    :DESTINATION-SPECIFICATION
                      `((GRID:FOREIGN-ARRAY ,I) DOUBLE-FLOAT))))
           (LET ((RESIDUAL
              (ABS
                (GSL:EUCLIDEAN-NORM FITNESS-VECTOR))))
            (COND
              ((AND (< PERS 1) (<  ITER-LOOP 1))
                (SETF (AREF RESIDUAL-BEST-POS PERS) RESIDUAL)
                (SETF (GRID:ROW BEST-POS-MATRIX PERS)
                  (GRID:MAP-GRID :SOURCE (GRID:ROW POP-POS-MATRIX PERS))))
              ((< RESIDUAL (AREF RESIDUAL-BEST-POS PERS) )
                (SETF (AREF RESIDUAL-BEST-POS PERS) RESIDUAL)
                (SETF (GRID:ROW BEST-POS-MATRIX PERS)
                  (GRID:MAP-GRID :SOURCE (GRID:ROW POP-POS-MATRIX PERS)))))

            ; Setting best particle in swarm
            (COND
              ((AND (< PERS 1) (<  ITER-LOOP 1))
                 (SETF RESIDUAL-BEST-POP RESIDUAL)
                 (SETF BEST-POP-VECTOR
                   (GRID:MAP-GRID
                         :SOURCE (GRID:ROW POP-POS-MATRIX PERS))))
              ((< RESIDUAL RESIDUAL-BEST-POP )
                 (SETF RESIDUAL-BEST-POP RESIDUAL)
                 (SETF BEST-POP-VECTOR
                   (GRID:MAP-GRID
                         :SOURCE (GRID:ROW POP-POS-MATRIX PERS)))))))))
;; PSO algorithm
    (LET (
      (RNG
        (GSL:MAKE-RANDOM-NUMBER-GENERATOR GSL:+MT19937+ (RANDOM POP-SIZE))))
      (ITER:ITER (ITER:FOR PERS FROM 0 BELOW POP-SIZE)
        (ITER:ITER (ITER:FOR K FROM 0 BELOW I)
          (SETF
            (GRID:GREF POP-VELOCITY-MATRIX PERS K)
              (+
                (GRID:GREF POP-VELOCITY-MATRIX PERS K)
                (*
                  (GSL:SAMPLE RNG :UNIFORM)
                  C1
```

```
                (-
                  (GRID:GREF BEST-POS-MATRIX PERS K)
                  (GRID:GREF POP-POS-MATRIX PERS K)))
              (*
                (GSL:SAMPLE RNG :UNIFORM)
                C2
                (-
                  (GRID:GREF BEST-POP-VECTOR K)
                  (GRID:GREF POP-POS-MATRIX PERS K))))))))
    (SETF POP-POS-MATRIX
        (GRID:MAP-N-GRIDS
            :SOURCES
              (LIST
                (LIST POP-VELOCITY-MATRIX NIL)
                (LIST POP-POS-MATRIX NIL))
            :COMBINATION-FUNCTION (LAMBDA (A B) (+ A B))))
FINALLY
  (RETURN T))))
```

Appendix E

Transient Analysis in CLASP

```
(DEFUN TRANS (START STEP STOP &OPTIONAL SOLVER)
  (MAKE-INSTANCE 'MATRICE-SYSTEM)
  (MAKE-INSTANCE 'CLASS-VARIABLES)
  (LET ((M (MAKE-INSTANCE 'MATRICE-SYSTEM))
        (H 1E-8)
        (RELTOL 1E-3)
        (ABSTOL 1E-12)
        (CHGTOL 1E-14)
        (VTRANS (MAKE-INSTANCE 'CLASS-VARIABLES)))
    (MAP-ALL-DEVICES M)
    (PRINT (GET-VARIABLE-LABEL-VECTOR M ))
    (SET-NEW-SYMBOL-VAR-MATRIX VTRANS START STEP STOP (GET-VARIABLE-LABEL-VECTOR M))
    (SETF *TIME-POS* 0)
    (SETF *TIME* (GET-SYMBOL-VAR-TIME VTRANS *TIME-POS*))
    (SOLVER-NEWTON-RAPHSON M VTRANS (1- (SIZE M)) *RESIDUAL* 50)
    (SETF *TIME-POS* 1)
    (SET-SYMBOL-VAR-TIME VTRANS *TIME-POS* (+ *TIME* H))
    (SETF *TIME* (+ *TIME* H))
    (SOLVER-BDF-0 M VTRANS (1- (SIZE M)) H *RESIDUAL* 50)
    (SETF *TIME-POS* 2)
    (SETF *TIME* (GET-SYMBOL-VAR-TIME VTRANS *TIME-POS*))
    (SOLVER-BDF-1 M VTRANS (1- (SIZE M)) H *RESIDUAL* 50)
    (SETF *TIME-POS* 3)
    (SETF *TIME* (GET-SYMBOL-VAR-TIME VTRANS *TIME-POS*))
    (SOLVER-BDF-2 M VTRANS (1- (SIZE M)) H *RESIDUAL* 50)
    (LOOP FOR I FROM 4 BELOW (FIRST (SIZE VTRANS)) DO
      (SETF *TIME-POS* I)
      (SETF *TIME* (GET-SYMBOL-VAR-TIME VTRANS *TIME-POS*))
      (SOLVER-BDF-1 M VTRANS (1- (SIZE M)) H *RESIDUAL* 50))
    (GET-ALL-SYMBOLS VTRANS)))
```