Common LISP as Simulation Program (CLASP) of Electronic Circuits

David ČERNÝ, Josef DOBEŠ

Dept. of Radio Engineering, Czech Technical University in Prague, Technická 2, 166 27 Praha 6, Czech Republic

cernyd1@fel.cvut.cz, dobes@fel.cvut.cz

Abstract. *In this paper, an unusual and efficient usage of functional programming language Common LISP as simulation program (CLASP) for electronic circuits is proposed. The principle of automatic self-modifying program has enabled complete freedom in definition of methods for 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 and diode is given all together with 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. A comparison of performances of two different linear solvers (an original and the standard GNU GSL) for circuit equations is demonstrated by an algorithm for automatic generation of huge circuits.*

Keywords

Common LISP, computer simulation, electronic circuits, device models

1. Introduction

In the middle of the last century, increasing demand for more complex electronic devices and race with competitors forced companies to think more about their development processes. Tools for computer simulation and computer-aided design (CAD) were created. Between the years 1967 and 1971 the first version was formed of the program for simulation of electronic circuits in the University of California at Berkeley. Originally called CANCER [1], the program was later renamed to less misleading name SPICE, Simulation Program with Integrated-Circuit Emphasis [2].

The next generation of the program, SPICE2, was completed in 1975. It came with a new circuit representation, known as modified nodal analysis (MNA) [3], better memory management, time-step control mechanism and new reliably stable multiple-order integration method. Advanced capabilities of the program SPICE and its free redistribution under Berkeley open-source license ensured that the core of the SPICE program was derived into many other CAD programs. For example PSpice (Cadence), Affirma Spectre (Cadence), HSPICE (Synopsys), ELDO (Mentor Graphics), SmartSpice (SILVACO), NgSpice and many others. More on the history of the SPICE can be found in [4].

2. Computer Simulation Today

With the beginning of the 21st century, the approach has changed to computer simulation. It has become a common part of the manufacturing process used by teams of engineers and scientists. The collaborative way of use resulted in new requirements for versatility and re-usability of previous work Development of the Internet and communication media enabled a new self-enhancing production models, as "open-source" distribution of software and its development by collective work of interactive communities.

The modern programs for computer simulation are not single purpose simulators anymore. They developed into multi-functional systems that can offer wide range of functionality through defined high-level language or interactive scripting interface. They also support extension of functionality by addition of professional or user-defined packages.

3. Drawbacks of Current Concept

The majority of the simulation systems are shipped to a user as binary files. The source code is treated as trade secret and the users are not able to check the accuracy of simulation algorithms inside. This brings a new unwanted element to simulation, unreliability of results. Reading manuals of simulators one can often find a note challenging results and recommending testing before final implementation.

The simulators usually come with their own internal scripting language which originates in the good intention to make them attractive and more effective. Unfortunately there exist hundreds of different simulators today. Each with different programming semantics intended by different companies and with only partially guaranteed backward compatibility. Changes in the user defined simulation are generally necessary whenever a new version of simulator is released.

Despite all modularity of today's simulators, in many situations they are used on the border of their limits and sometimes the only way to perform optimized simulation of desired problem is total source code reorganization. It could be seen especially on the device models, which must be incorporated in program core to make different simulation types possible.

The first electronic simulators were written in programing languages as FORTRAN 4 and FORTRAN 77 or later in far more modern C or C++. There are very powerful programming languages, as to their concept in memory allocation and management, which used to be considered as the main evaluative criteria of programming language, but less practical in terms of their own dynamical readjustment. By the dynamical readjustment is meant the process when internal functions and variables of program can be redefined in the runtime to solve the problem. This ability is inherent in programming languages designed for genetic simulation and low level machine code languages.

4. Common LISP

The LISt Processing language (LISP) was designed by John McCarthy from Massachusetts Institute of Technology (MIT) in 1958 as a new symbolic data processing language [5]. But 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 the Cold War helped as the Pentagon supported projects for large-scale battlefield simulations, automated planning, and natural language interfaces. In 1981, the standardizing process of a new language called Common LISP (CL) began 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 [6].

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 [**?**].

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 the addition function.

The syntax of CL is case-insensitive. It means that it does not matter whether it is written as (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.

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 call returns number 1.

One major aspect of CL is its ability of general abstraction by "functionals". The term "functional" denotes a special function that has one or more functions as arguments, or

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 return 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 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 the 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 little modified to show capabilities of CL syntactic abbreviations.

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 table does not contain a symbol with that name, the reader makes a new symbol and adds it to 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 the STRING argument. If there is no such symbol in the OBARRAY table defined by argument PACKAGE, INTERN creates a new one, adds it to the new OBARRAY, and returns it. If the PACKAGE argument is omitted, global OBARRAY is used.

Major advantages of CL have been lightly touched to clarify the 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 [5], [6].

5. CLASP

The acronym CLASP is 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.

6. Device Modeling

To be able to model any electronic device in CLASP, it is mandatory to formulate its characteristic equations according to Kirchhoff's circuit laws. These equations are construction blocks for design of modified nodal formulation (MNF) [3] by inspection. To meet capabilities of CLASP, the original inspection method was extended introducing new matrices and vectors to circuit system equations. In CLASP, standard MNF representation 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 will be stored in the matrix $\mathbf{G}$, values that are associated with frequency variable will be 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 device with respect to its position in the matrix. This is possible due to the CLASP capability of dynamical variables assignment. The remaining equations describing model of device 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. More detailed description of the system equations will be given in Section 7.

6.1 Resistor

The simple model for ideal resistor in Fig. 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 device. It is better, whenever it is possible, to use admittance description for

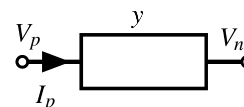


Fig. 1. Resistor graphical symbol.

device model, because it at least partially helps to reduce sparsity of resulting system.

The node voltage at positive terminal of 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 resistor from 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

$$\begin{aligned} I_p &= y(V_p - V_n), \\ I_n &= -y(V_p - V_n) \end{aligned} \tag{1}$$

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

$$G = \begin{array}{c} \\ I_p \\ I_n \end{array} \begin{array}{c} \begin{matrix} V_p & V_n \end{matrix} \\ \begin{pmatrix} y & -y \\ -y & y \end{pmatrix} \end{array}.$$

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

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

6.2 Voltage Source

The graphical symbol of voltage source model is shown in Fig. 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.

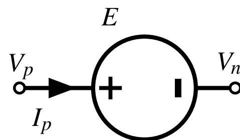


Fig. 2. Voltage source symbol.

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

$$\begin{aligned} V_p - V_n &= E, \\ I_p &= I, \\ I_n &= -I, \end{aligned} \tag{2}$$

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

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

Increased size of matrix is indicated by index "+".

6.3 Simple Diode

A simple diode model will be defined in this section. Generally, the diode is a two-terminal electronic device with nonlinear current-voltage characteristics. The diode symbol is shown in Fig. 3.

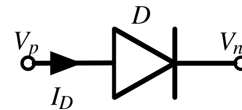


Fig. 3. Diode symbol.

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

$$I_\mathrm{D} = I_\mathrm{S}\left(e^{V_\mathrm{D}/(nV_\mathrm{T})} - 1\right) \tag{3}$$

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. The thermal voltage V_T is well known constant defined by the equation $V_\mathrm{T} = kT/q$, where k is the Boltzmann constant (1.38×10^{-23} J/K), T is absolute temperature of the P-N junction, and q is elementary charge (1.602×10^{-19} 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:

$$\begin{aligned} I_\mathrm{D} &= I_\mathrm{S}\left(e^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} - 1\right), \\ I_p &= I_\mathrm{D}, \\ I_n &= -I_\mathrm{D}, \end{aligned} \tag{4}$$

from which the Jacobian matrix can be written directly as:

$$\mathbf{M}|_\mathbf{x} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ \lambda e^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & -\lambda e^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & -1 \end{pmatrix} \tag{5}$$

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

Putting 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

$$\mathbf{G} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & -1 \end{pmatrix},$$
$$\mathbf{D} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \lambda e^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & -\lambda e^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})} & 0 \end{pmatrix}, \tag{6}$$
$$\mathbf{e}_\mathrm{q} = \begin{pmatrix} 0 & 0 & I_\mathrm{S}\left(e^{(V_\mathrm{p}-V_\mathrm{n})/(nV_\mathrm{T})}-1\right) \end{pmatrix}^t$$

where t denotes transposition of 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 with respect to node voltage.

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

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

7. Simulation and Analysis in CLASP

Programs for the electronic circuits simulation usually come with variety of simulation processes and analyses, i.e. operating point (OP), sensitivity analysis, transient analysis and many others. In this paper main focus will be pointed to the linear and nonlinear direct current (DC) analysis, and DC sweep analysis.

7.1 Linear DC Analysis

CLASP divides DC analysis to linear DC and nonlinear DC analysis. When a circuit includes only linear devices, the simulation enters matrices **G**, and **E** together with right-hand-side (RHS) vectors $\mathbf{r}_\mathrm{n}$ and $\mathbf{r}_\mathrm{q}$. The remaining matrices and vectors are during linear DC analysis equal to zero.

Matrix **E** has two functions. Normally it holds unity values of frequency dependent devices as capacitors and inductors. But during DC analysis matrix **E** is added together with matrix **G**. This operation removes frequency dependent devices. It replaces inductors with shortcuts and disconnects all capacitors. The advantage is that the voltage nodes and their variables, even if they are in disconnected part of circuit, stay in the equations. The resulting linear system is defined as

$$(\mathbf{G}+\mathbf{E})\,\mathbf{x} = \mathbf{r}_\mathrm{q}(\mathbf{s}) + \mathbf{r}_\mathrm{n} \tag{7}$$

where vector **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 **s** is vector of circuit system parameters which shall be known at any time of simulation. The solution of linear system (7) is best computed with respect to accuracy of results by some factorization method. In network application LU factorization algorithm [7] is often used.

Implementation of LU factorization in CLASP is a not straight-forward 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 bignums, rational and complex numbers. For instance, a common property of mathematical numbers x/10 + y/10 = (x + y) /10, which does not hold for floating point numbers [8]:

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

and results in mathematically incorrect value 0.70000005 (single float). It can be solved in CLASP converting floating point numbers to rational numbers:

```
(+ 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 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.

However, a series of tests has proven that the rich and accurate hierarchy of numbers in CLASP slows down efficiency of evaluation in high performance commutating tasks.

Even with very optimal code, CLASP can not compete 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.

7.2 Nonlinear DC Analysis

When circuit includes some nonlinear models of devices as for example diodes, the solution of network equations cannot be obtained directly and equations shall be linearized before. One of the basic methods for solving of nonlinear equations implemented in CLASP is Newton-Raphson (NR) method [9]. Brief description of its implementation in CLASP program follows. Lets start with definition of a system $\mathbf{f}$ of n nonlinear equations

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \tag{8}$$

substituting CLASP matrix system, it changes to form:

$$(\mathbf{G}+\mathbf{E})\mathbf{x}+\mathbf{e}_\mathrm{q}(\mathbf{x})-\mathbf{r}_\mathrm{q}(\mathbf{s})-\mathbf{r}_\mathrm{n}=\mathbf{0}. \tag{9}$$

To derive NR algorithm let's denote the solution of the system as vector $\mathbf{x}^*$ and vector $\mathbf{x}$ as any different solution. Applying Taylor expansion and neglecting higher terms in expansion, the system can be formulated in a linearized form, which will result, by the use of CLASP matrices, in the following notation:

$$\mathbf{f}(\mathbf{x}^*) \approx (\mathbf{G}+\mathbf{E})\mathbf{x}+\mathbf{e}_\mathrm{q}(\mathbf{x})-\mathbf{r}_\mathrm{q}(\mathbf{s})-\mathbf{r}_\mathrm{n}+\mathbf{M}(\triangle\mathbf{x}) \tag{10}$$

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

$$\mathbf{M}|_\mathbf{x} = \begin{pmatrix} \frac{\partial D_{1,1}}{\partial x_1} & \frac{\partial D_{1,2}}{\partial x_2} & \cdots & \frac{\partial D_{1,n}}{\partial x_n} \\ \frac{\partial D_{2,1}}{\partial x_1} & \frac{\partial D_{2,2}}{\partial x_2} & \cdots & \frac{\partial D_{2,n}}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial D_{n,1}}{\partial x_1} & \frac{\partial D_{n,2}}{\partial x_2} & \cdots & \frac{\partial D_{n,n}}{\partial x_n} \end{pmatrix} + \mathbf{G} + \mathbf{E} \tag{11}$$

is the Jacobian matrix compiled of matrix $\mathbf{G}$ and $\mathbf{E}$, and evaluated expressions from matrix $\mathbf{D}$. As follows from (8) both sides of (10) shall be zero. Evidently, when (10) is set equal to zero and solved, the result will not be the vector $\mathbf{x}^*$ (because the higher-order terms in Taylor expansion have been neglected) but some new value for $\mathbf{x}$. Using superscripts to indicate iteration sequence, the equation gets this form

$$(\mathbf{G}+\mathbf{E})\mathbf{x}^k+\mathbf{e}_\mathrm{q}(\mathbf{x}^k)-\mathbf{r}_\mathrm{q}(\mathbf{s})-\mathbf{r}_\mathrm{n}+\mathbf{M}(\mathbf{x}^{k+1}-\mathbf{x}^k)=\mathbf{0}. \tag{12}$$

The solution for next iteration $\mathbf{x}^{k+1}$ is usually obtained by LU factorization, it is convenient to rewrite (12) to form:

$$\mathbf{M}(\triangle\mathbf{x}^k) = -\mathbf{e}_\mathrm{q}(\mathbf{x}^k)-(\mathbf{G}+\mathbf{E})\mathbf{x}^k+\mathbf{r}_\mathrm{q}(\mathbf{s})+\mathbf{r}_\mathrm{n} \tag{13}$$

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

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \triangle\mathbf{x}^k. \tag{14}$$

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:

$$\left\|\mathbf{f}(\mathbf{x}^k)\right\| - \left\|\mathbf{f}(\mathbf{x}^{k+1})\right\| \le \varepsilon \tag{15}$$

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

7.3 DC Sweep

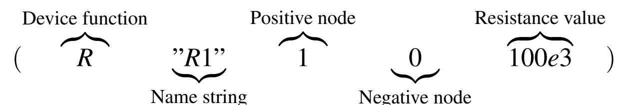
The DC Sweep analysis performs a series of operating point analyses, modifying the selected parameter in predefined steps, given by a function. Special voltage source was defined for the purposes of the DC Sweep simulation. Its value depends on a predefined function. The example of CLASP implementation of time dependent voltage source with sinusoidal voltage output follows:

```
(EF ”E1” 1 0
  #'(LAMBDA ()
    (* (SIN *TIME*))))
```

In CLASP, a starred symbol denotes global variable. In this case it is the time variable. The value of the variable *TIME* is increased after every evaluation during the DC Sweep simulation. DC Sweep simulation can sweep over any other global variable as for example temperature.

8. 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.



This makes a resistor device with name “R1” and connected to nodes 1 and 0 (ground). The resistance value of the resistor will be 100 kΩ. Implementation of scale factors (μ, m, k, G, M...), with respect to capabilities of CLASP, proved to be redundant. Example of definition of complete resistor circuit from Fig. 4 follows

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

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

9. Simulation Results

9.1 Benchmark of LU Solver

Performance of CLASP capability of solving linear equations was demonstrated by implementation of two solvers. The first implemented algorithm into CLASP for LU factorization was a modified Crout algorithm. The original algorithm was modified for optimized solution of sparse matrices. The main part of CLASP code implementation can be found in the Appendix. This solver will be referred to as CLASP LU. 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 [?]. 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.

For automatized testing of huge sparse matrices a function was created for automatic generation of circuit with random device values. All tests were run on a computer with two Intel Core 2 Duo, with CPU frequency 2.26 GHz and 3.9 GB RAM. Operating system was Ubuntu with Linux Kernel 2.6.38-10-generic-pae.

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
CLASP LU	1,024	0.293466	1,295,888
	9,216	12.373999	29,603,072
	102,400	1,083.0775	1,040,184,336

Tab. 1. Comparison of different LU solvers.

The results are shown in Tab. 1. The column MATRIX SIZE includes 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 limited to 102,400 in the first CLASP LU solver due to estimated time of the solution being out of reasonable range.

9.2 Simulation of Nonlinear Device

A simple nonlinear circuit is shown in Fig. 4. This circuit was used as an example of simulation with nonlinear devices in CLASP. It is consists of the voltage source V1, two resistors R1 and R2, and diode D1. The simple diode model is used in this simulation defined by Shockley equation. Diode model definition by Shockley equation had been defined in previous section. The resulting I-V characteristic of simulation of the diode circuit is shown in Fig. 5.

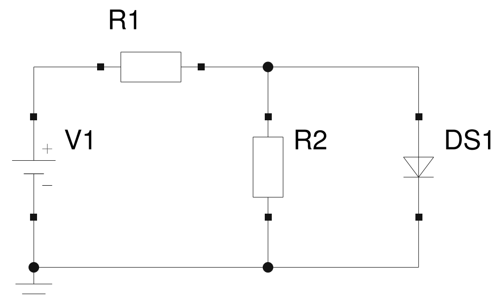


Fig. 4. Simple diode circuit.

The circuit was evaluated in 50 different points corresponding with the value of voltage source from 0 to 5 V. The resistance values of both resistors were set to 500 Ω. NR did not encounter any nonconvergence during evaluation. The starting values of NR algorithm were randomly generated in every point of the voltage source sweep. NR iterative algorithm took on average 69.7 iteration loops.

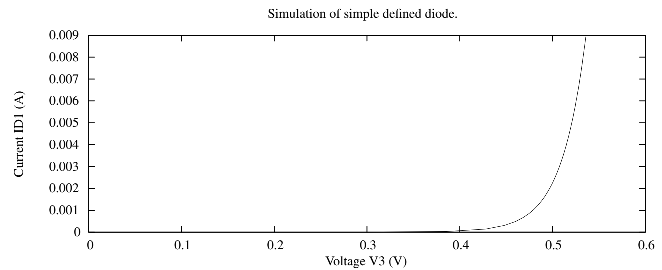


Fig. 5. I-V diode characteristics.

9.3 Full-way Rectifier

A little more complex situation is shown in Fig. 6. It is a simple full-way rectifier consisting of four diodes D1-D4, sinusoidal voltage source V1, and resistor R1. The absence of reverse break-down voltage definition slows down the convergence of NR algorithm in some cases. Therefore, the definition of Shockley diode equation was expanded by setting the reverse current to zero. The definition of the simple diode in CLASP will now have the form:

```
(COND
((< (- (EVAL VP) (EVAL VN)) 0) 0)
```

```
(T
  (* 10e-12
    (- (EXP
         (/ (- (EVAL VP) (EVAL VN))
           26e-3)) 1))))
```

When the voltage over diode from anode to cathode is positive then normal Shockley equation is used. Whenever the voltage is negative, then the current through the diode is set to zero. The result from the simulation of full-way rectifier is shown in Fig. 7.

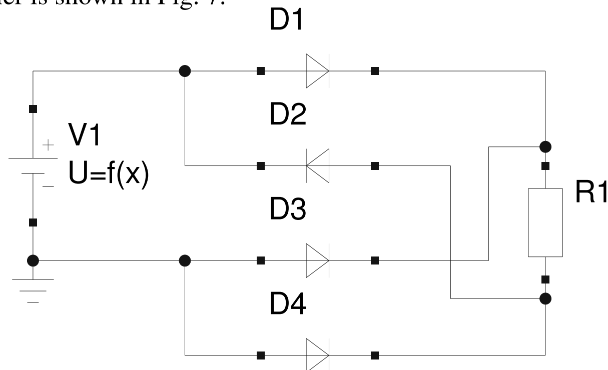


Fig. 6. Full-way rectifier circuit.

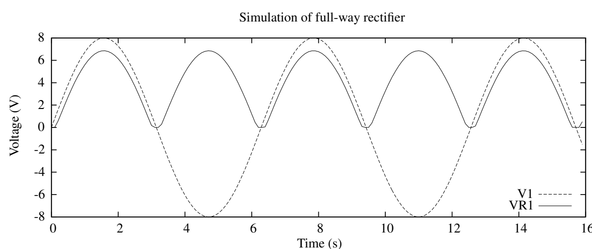


Fig. 7. I/O Voltage of full-way rectifier. The time axis represents DC sweep independent variable.

10. Conclusion

CLASP proved to be a very effective tool in circuit simulation and electronic device modeling. Particular device model is made by rewriting its mathematical description of circuit equations to CLASP code only. Once the device is incorporated into CLASP, it can be used in any defined simulation. The core idea which allows this approach lies in CLASP capability of dynamical variable creation and assignment. It is important to pick out the fact that not only variables can be dynamically defined, but also functions and even entire program source code can redefine itself in runtime. This is important advantage of CLASP.

The capabilities of CLASP were demonstrated by two simple circuits Fig. 4 and Fig. 6, the resulting graphs Fig. 5 and Fig. 7 from their simulations serve as an ilustration of the simplicity and variability of CLASP. The new approach to principle of the program organization allowed to eliminate the redundancy in device modeling and helped to reduce requirements on programming skills of the user. The simplicity of doing electronic simulation in CLASP is also proved by the fact that several pages of this paper were enough to present practically entire implementation of the simulator.

In this paper, comparison between CLASP implementation of LU solver and GNU GSL library functions were also presented. The benchmark simulation (Tab. 1) clearly shows that far better results were achieved by the use of the GNU GSL library. This implies that it is a far better approach to assign the difficult tasks, requiring high performance computation, to another computation system. The use of GNU GSL library was only one option at this moment. The main disadvantage of GNU GSL library was absence of any LU factorization solver for sparse matrices. Further development of CLASP should solve this problem.

Acknowledgements

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/286/OHK3/3T/13, and by the Grant Agency of the Czech Republic, grant No P102/10/1665.

References

[1] NAGEL, L. W., ROHRER, R. A. Computer analysis of nonlinear circuits, excluding radiation (CANCER). *IEEE Journal of Solid-State Circuits*, 1971, vol. 6, no. 4, p. 166 - 182.

[2] NAGEL, L. W. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis. Berkeley (CA, USA): University of California, 1975.

[3] HO, C.-W., RUEHLI, A. E., BRENNAN, P. A. The modified nodal approach to network analysis. *IEEE Transactions on Circuits and Systems*, 1975, vol. 22, no. 6, p. 504 - 509.

[4] PEDERSON, D. A historical review of circuit simulation. *IEEE Transactions and Circuits and Systems*, 1984, vol. 31, no. 1, p. 103 - 111.

[5] MCCARTHY, J. *LISP 1.5 Programmer's Manual*. Cambridge (MA, USA): MIT Press, 1965.

[6] SEIBEL, P. *Practical Common Lisp*. New York: Apress, 2005.

[7] HIGHAM, N. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial Mathematics, 2002.

[8] GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 1991, vol. 23, no. 1, p. 5 - 48.

[9] SÜLI, E., MAYERS, D. *An Introduction to Numerical Analysis*. Cambridge (UK): Cambridge University Press, 2003.

About Authors...

David ČERNÝ was born in Prague, Czech Republic, in 1985. He received his M.Sc. degree in 2009 from the Faculty of Electrical Engineering of the Czech Technical University in Prague. Since September 2009 he has been a PhD student at the Department of Radioelectronics at Czech Technical

University in Prague. His research interests include simulation of high frequency circuits and physical modeling of electrical devices.

Josef DOBEŠ received the Ph.D. degree in microelectronics from the Czech Technical University in Prague in 1986. From 1986 to 1992, he was a researcher of the TESLA Research Institute, where he performed analyses on algorithms for CMOS Technology Simulators. Currently, he works at the Department of Radio Electronics of the Czech Technical University in Prague. His research interests include the physical modeling of radio electronic circuit elements, especially RF and microwave transistors and transmission lines, creating or improving special algorithms for the circuit analysis and optimization, such as time- and frequency-domain sensitivity, poles-zeros or steady-state analyses, and creating a comprehensive CAD tool for the analysis and optimization of RF and microwave circuits.

Appendix

Here is CLASP implementation of the sparse LU solver (CLASP LU). It is a novel modification of the Crout algorithm for solving sparse LU factorization, which has not been published and is capable of solving the problem with arbitrary precision.

```
;; 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 substraction 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 substraction 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)))
```