MATEC Web of Conferences **76**, 01007 (2016) DOI: 10.1051/matecconf/20167601007
CSCC 2016

An Efficient Procedure for Transient Analysis of Electronic Circuits with Increased Precision

David Cerny[1,a] and *Josef* Dobes[1,b]

[1]*Czech Technical University in Prague*
Department of Radio Engineering
Technicka 2, 166 27 Prague 6, Czech Republic

Abstract. This article presents an efficient procedure allowing to increase the final accuracy of the electronic circuits simulation. To obtain Transient Analysis of a nonlinear circuit the simulator needs to perform various algorithms, from an evaluation of initial DC state through a nonlinear iterative algorithm, to a numerical integration. These complex computations have to be implemented effectively with regards to memory and processor usage. Direct implementation of arbitrary precision types can directly lead to increased simulation accuracy but also significantly decrease simulation performance. As a compromise, we suggest using an additional algorithm based on Newton Iteration method for the solution of matrix inversion. The method was modified to be comparable to direct LUF solver.

1 Introduction

Every year significant number of articles focus on simulation core of Spice and its device models is published . As interesting works on similar problematic to this article we can point out [1–4]. It is without a doubt that the accuracy of a simulation of the electronic circuits is essential. The primary importance can be seen 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 datatype. 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 may occur situations when a circuit matrix is changed a lot. 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.

2 Simulation Overview

In the Spice program, there are four basic analysis for simulation of electronic circuits: Transient Analysis, Operation Point (DC), Small Signal Analysis and Noise Analysis. All of them shares a common family of computation algorithms. As the most critical Transient analysis can be considered which uses the most time demanding operations. Spice simulation core to compute Transient analysis needs to evaluate a system of nonlinear differential equations defined by device models. It is done by Newton-Raphson iteration algorithm together with LU factorization (LUF) method capable of handling sparse matrices and pivoting.

Algorithm 1 Transient Simulation Procedure
```
Simulation definition
Modified Nodal Analysis
for Time do
  Initial value estimation
  DC Analysis (optional)
  if Non-linear System then
    repeat
      Generate Jacobian matrix
      Reordering and LU factorisation
      Evaluation of next value estimation
      Compute residual
    until Stopping criteria or convergence
  end if
end for
```

In particular cases, when time dependent devices enter a simulation, it also needs to enumerate numerical integration using Trapezoidal, Gear or another method. Algorithm 1 characterize simulation of the circuit with nonlinear devices performed by Transient analysis. It should be pointed out that computation of each time-step requires solution of the linear system produced by Jacobian ma-

[a]e-mail: cernyd1@fel.cvut.cz
[b]e-mail: dobes@fel.cvut.cz

Table 1. Comparison of Performance of NIM and LUF to Matrix Dimension

Dim.	LUF [s]	NIM [s]	LUF -O1 [s]	NIM -O1 [s]	LUF -Ofast [s]	NIM -Ofast [s]	ratio [%]
50	0.001643	0.00121	0.000539	0.000271	0.000405	0.000215	53.08641975
100	0.00785	0.008951	0.001903	0.001944	0.00192	0.000819	42.65625
150	0.023481	0.029905	0.00893	0.006858	0.005238	0.003025	57.75105002
200	0.030418	0.070165	0.018022	0.016332	0.009884	0.007513	76.01173614
250	0.053492	0.136668	0.016572	0.031944	0.010891	0.01431	131.3928932
300	0.07998	0.234934	0.035751	0.055234	0.021445	0.023843	111.1820937
350	0.174865	0.371465	0.046123	0.086999	0.031859	0.036893	115.8008726
400	0.206388	0.566372	0.075759	0.130182	0.042249	0.052848	125.0869843
450	0.291311	0.787467	0.109412	0.184372	0.0471	0.077378	164.2845011
500	0.482965	1.077401	0.137663	0.25123	0.064424	0.101362	157.3357755

trix. It is done by direct LU factorization method based on modified Crout algorithm.

3 Precision Problems

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

$$1.0999999999999999923753143406777998958290 \times 10^{-8}$$

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). 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, if we have a system:

$$\begin{cases} 4.012x_1 + 4.013x_2 = 4011 \\ 4.013x_1 + 4.013x_2 = 4012 \end{cases} \tag{1}$$

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 $r < 10^{-15}$.

4 Precision of the Algorithm

The original mathematical definition of Newton Iteration Method (NIM) [5, 6] can be rewritten to the form applicable for implementation as an iterative algorithm for precision enhancement. Denoting Jacobian matrix as **J** and Identity matrix as **I**, for inverted Jacobian matrix $\mathbf{J^{-1}}$ we can write

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

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

$$\mathbf{X_n} = \mathbf{J}^{-1} + \varepsilon_\mathbf{n} \Rightarrow \mathbf{X_{n+1}} = \mathbf{J}^{-1} + \varepsilon_{n+1}, \tag{3}$$

where ε_n is an error matrix. From that, we get to

$$\mathbf{J}^{-1} + \varepsilon_{n+1} = (\mathbf{J}^{-1} + \varepsilon_n)(2\mathbf{I} - \mathbf{J}(\mathbf{J}^{-1} + \varepsilon_n)). \tag{4}$$

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

$$\mathbf{X}_{n+1} = \mathbf{X}_n\left(2\mathbf{I} - \mathbf{J}\mathbf{X}_n\right). \tag{5}$$

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

$$0 < \|2\mathbf{I} - \mathbf{J}\mathbf{X}_n\| < 1. \tag{6}$$

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

$$J(x_n)(x_{n+1} - x_n) = -F(x_n). \tag{7}$$

Table 2. Overview of Compiler Optimization Settings

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.

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 we finally receive final algorithm:

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

where A^{-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

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

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 3. 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.

5 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 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.

Algorithm 2 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
```

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

6 Results

In the Table 1 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 showed in the Figure 1. Table 3 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

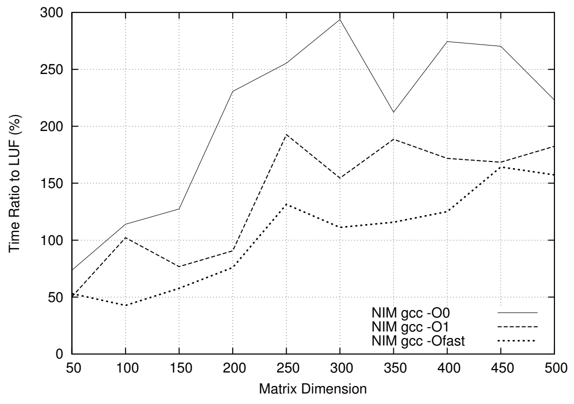

Figure 1. Time Ration of NIM to LUF vs Matrix Dimension

Table 3. Computation Times in Sec. of Individual Steps of NIM

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 4. Accuracy Boost Given by One Iteration Loop of NIM with Arbitrary Precision

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

can be taken as the most time demanding operation 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 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 2. It got an additional precision boost computed by NIM after each successful iteration time-step.

7 Conclusion

In this article, we presented an 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 of OP simulation or as a final procedure after each time-step of Transient analysis. It is clear that iterative NIM will always be outperformed by direct LUF method, especially in a case of sparse matrices. However, the 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, we can achieve total accuracy 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 a point of view of performance. Standard LUF followed with a single iteration of NIM with arbitrary precision can definetely improve the accuracy with preserved performance for dense matrices. In a case of sparse matrices efficiency of NIM is discussable and will be decreasing with growing sparsity of the matrix. Therefore method is aplicable for small circuits only (up to 200 nodes).

Acknowledgment

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS15/161/OHK3/2T/13

References

[1] A. Subbiah, O. Wasynczuk, IEEE Transactions on Power Electronics **31**, 6351 (2016)
[2] I. Mejia, G. Gutierrez-Heredia, A.L. Salas-Villasenor, C.G. Alvarado-Beltran, C. Avila-Avendano, M.A. Quevedo-Lopez, IEEE Transactions on Electron Devices **63**, 1437 (2016)
[3] D. Bulakh, *Fast SPICE based circuit simulation using source code generation approach*, in *Internet Technologies and Applications (ITA), 2015* (2015), pp. 53–55
[4] X. Chen, L. Ren, Y. Wang, H. Yang, IEEE Transactions on Parallel and Distributed Systems **26**, 786 (2015)
[5] A. Ben-Israel, Mathematics of Computation pp. 452–455 (1965)
[6] A. Ben-Israel, D. Cohen, SIAM Journal on Numerical Analysis **3**, 410 (1966)