New optimising feature in *SPICE*

Janez Puhan, Tadej Tuma, Iztok Fajfar
Fakulteta za elektrotehniko
Univerza v Ljubljani
Tržaška 25, 1001 Ljubljana, Slovenija
janez.puhan@fe.uni-lj.si, tadej.tuma@fe.uni-lj.si, iztok.fajfar@fe.uni-lj.si

Abstract

The field of optimisation is still not appropriately covered in modern computer programs for circuit analysis as SPICE, which was originally developed at the University of California at Berkeley. The last official release of SPICE was published in 1992 and includes interactive interpreter language called Nutmeg, which is at first glance very convenient for coding optimisation loops. But it turns out that Nutmeg has numerous of bugs and is therefore unsuitable for optimising larger circuits with more parameters. Because of that reason the original code was modified. The majority of bugs were found and fixed. The code was also modified in order to run under Windows 95/98/NT. New compilation of SPICE, which offers a native SPICE environment on PC and allows longer coding (for example optimisation method) in Nutmeg, is introduced in this paper. Further, Nutmeg shows as to slow because of its interpreter nature. To overcome this problem in optimisation loops, a new command for circuit optimisation was added. In the paper, coding of optimisation loop in Nutmeg and the new optimize command are presented through two cases. All simulation files as well as the new compilation of SPICE may be downloaded from http://fides.fe.uni-lj.si/spice/.

1 Introduction

SPICE (Simulation Program with Integrated Circuit Emphasis) is the most commonly used analog circuit simulator today. Through years it has become a nonofficial industrial standard for computer aided design of electronic circuits. *SPICE* is a general-purpose analog simulator. It contains models for most circuit elements and can handle large non-linear circuits. The simulator can perform several different types of circuit analyses. The most important are the *dc* analysis, *ac* small-signal analysis and transient analysis, which is numerically the most complex analysis in *SPICE*.

The simulator was developed at the University of California, Berkeley, and was first released in 1972. Many scientists at Berkeley and other institutions have contributed to the development and improvement in subsequent versions of *SPICE*. The next major release of *SPICE*, called *SPICE2*, was published in 1975 [1]. The core of the program has remained intact, even after many improvements and additions. The last major release, *SPICE3* [2], came in 1985 with a conversion of the source code from Fortran to the C programming language. The source code of Berkeley's *SPICE* is public domain and is still available as freeware at *ftp://ic.berkeley.edu/pub/ Spice3/.*

In the last officially published version 3f4, the program is divided into two parts: the simulator and the front-end. The front-end includes an interactive interpreter programming language called *Nutmeg*, which allows interactive *SPICE* sessions. It acts as a simple pre- and post-processor. On the other hand it has all the properties of a real programming language and should be therefore appropriate for coding of optimisation methods. Our previous research showed that this is the fastest possible approach to the circuit optimisation [3]. Unfortunately *Nutmeg* yielded a numerous of bugs. It turned out that it has serious problems with memory allocation and memory leaks respectively. That was the main reason, why *Nutmeg* was not able to perform optimising loops on larger circuits with more than three or four parameters.

So we decided to recompile whole *SPICE* version 3f4 again on an IBM compatible personal computer and at the same time find bugs and fix them. The original source code can be compiled on MS-DOS operating system to a limited extent (simulator only) without any modification. We modified original code in a such way, that a full version of *SPICE* 3f4 (simulator and front-end) can be compiled and run on a PC with Windows 95/98/NT.

2 An optimisation example in *Nutmeg*

We are looking for values of the two resistors R_2 and R_3 in a simple schmitt trigger circuit in Figure 1. We want to have switching voltages at 3V (up) and at 2V (down). The high and low level output voltages must differ for at least 10V.

First we have to define some explicit as well as one implicit constraint. The explicit constraints are obvious since we know that the resistor values can't possibly lie outside the intervals (1). And because we want a difference between high and low voltage at least 10V, the implicit constraint is given by (2).

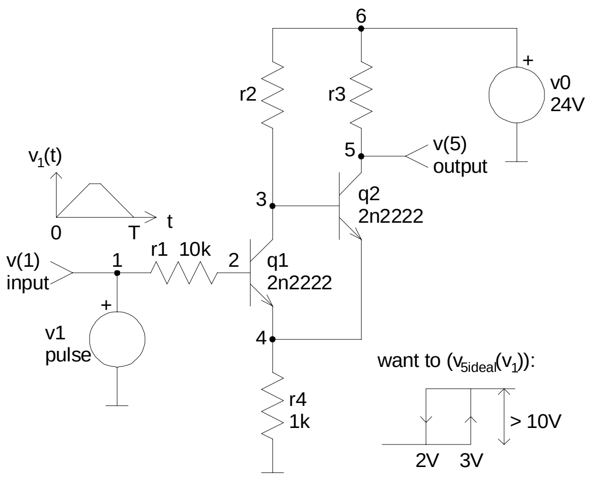


Figure 1: A simple schmitt trigger circuit

$$\begin{aligned} 10\text{k}\Omega \le R_2 \le 30\text{k}\Omega \\ 1\text{k}\Omega \le R_3 \le 50\text{k}\Omega \end{aligned} \quad (1)$$

$$v_{5\max}(R_2, R_3) - v_{5\min}(R_2, R_3) > 10\text{V} \quad (2)$$

The most important part of any optimisation is certainly the cost function. In our case we want the cost function to reflect the accuracy of switching voltages. Since we intend to do transient analyses, we define the cost function as a normalised square area between the real and ideal response:

$$E(R_2, R_3) = \frac{\int_0^T (v_5(R_2, R_3, t) - v_{5\text{ideal}}(R_2, R_3, t))^2\, dt}{(v_{5\max}(R_2, R_3) - v_{5\min}(R_2, R_3))^2} \quad (3)$$

The cost function must be a positive scalar and will be minimised during the optimisation procedure. The ideal value is zero, in which case the switching voltages are exactly at 3V and 2V. Transient response v_5 has to be transformed because of the numerical noise in it. So instead of v_5 we will use the expression (4), where boolean operator > is used. Using normalised ideal response $v_{5\text{ideal-norm}}$ (5) the cost function can be further simplified (6).

$$\left(v_5 > \frac{v_{5\max} + v_{5\min}}{2} \right) (v_{5\max} - v_{5\min}) + v_{5\min} \quad (4)$$

$$v_{5\text{ideal}} = v_{5\text{ideal-norm}} (v_{5\max} - v_{5\min}) + v_{5\min} \quad (5)$$

$$E = \int_0^T \left| \left(v_5 > \frac{v_{5\max} + v_{5\min}}{2} \right) - v_{5\text{ideal-norm}} \right| dt \quad (6)$$

The implementation of the implicit constraint and the cost function into *Nutmeg* source code is fairly simple:

```
* transient analysis (T = 11s)
tran 1ms 11s
linearize
* the cost function (v(5)[5500] = v(5)max)
let ideal = (vector(11001) gt 2999) and (vector(11001) lt 9001)
alias cost let cf = mean(abs((v(5) gt ((v(5)[5500] + v(5)[0]) / 2))
   + - ideal))
```

As for the optimisation algorithm, we decided to use well known constrained simplex optimisation method. It is robust and simple. Since the implementation of the optimisation algorithm takes over 200 *Nutmeg* programming lines, we will not discus it any further at this point. The entire simulation file *schmitt_trigger.cir* can be downloaded from *http://fides.fe.uni-lj.si/spice/*.

In our case the parameter space is two dimensional, so the cost function can be plotted over the explicitly constrained parameter plain. It has several channels and local minima as we can see in Figure 2. The channels are consequence of numerical errors, which take place because the time step in transient analysis is too long. It is determined automatically. We could define its maximum length, but one analysis would take more time than.

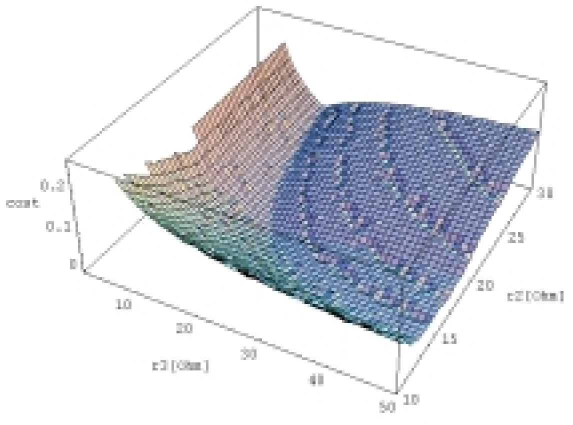


Figure 2: Cost function

The execution of our optimisation loop takes 70 transient analyses and stops at the global minimum with a cost function value of 0.004 and resistor values of R_2 = 18kΩ and R_3 = 20kΩ respectively. The optimisation takes about 27 seconds on a Pentium 350MHz personal computer running Windows 95.

3 New *optimize* command

Any optimisation problem can be handled with *Nutmeg*. To solve a particular problem we have to choose appropriate optimisation method and code it in *Nutmeg* code as it was shown in the previous section. But that kind of approach to optimisation takes a great deal of time for writing and debugging our own *Nutmeg* programs. We added a new command called *optimize* to *Nutmeg* to avoid programming. The syntax of the *optimize* command is as follows:

```
optimize [analysis [<n> delete | <expression>] |
    cost [<expression>] |
    implicit [<n> delete | <expression>] |
    method [<name> [<parameter1> <value1>
        [<parameter2> <value2> ...]]] |
    parameter [<n> delete | {[element <name>]
        [parameter <name>] [low <l>] [high <h>]
        [initial <i>]}] |
    options [<option1> <value1>] [<option2> <value2>] ...
```

We can define the cost function, one or more analyses and implicit and explicit constraints of the parameters in a very simple way with the *optimize* command. The optimising problem from the previous section can now be described in a few lines:

```
* parameters, and explicit constraints and initial point
optimize parameter 0 element r2 parameter resistance
    low 10k high 30k initial 12k
optimize parameter 1 element r3 parameter resistance
    low 1k high 50k initial 8k
* one implicit constraint
optimize implicit 0 v(5)[5500] - v(5)[0] gt 10
* analyses performed in every iteration
optimize analysis 0 tran 1ms 11s
optimize analysis 1 linearize
* cost function
optimize cost mean(abs((v(5) gt ((v(5)[5500] + v(5)[0]) / 2)) -
    ideal))
* optimising method is constrained simplex (all method
* parameters are left default)
optimize method complex
* run optimisation algorithm
optimize
```

The results of the optimisation with *optimize* command are approximately the same as described in the previous section. Small differences occurred because of the numerical noise in both cases. The *optimize* command needed slightly less time because all loops are now hard coded.

In *optimize method* line we can choose among several optimisation methods:

- *steepest descent*
- *Newton's method*
- *Davidon-Fletcher-Powell's method*
- *random search*
- *grid search method*
- *search along coordinate axes*
- *Powell's method*
- *Hooke-Jeeves's method*
- *constrained simplex method*
- *simple genetic algorithm*
- *evaluating cost function across whole parameter space*

When we run particular optimisation method on a given circuit with *optimize* command a general optimisation loop is performed and it is always the same. Its algorithm is as follows:

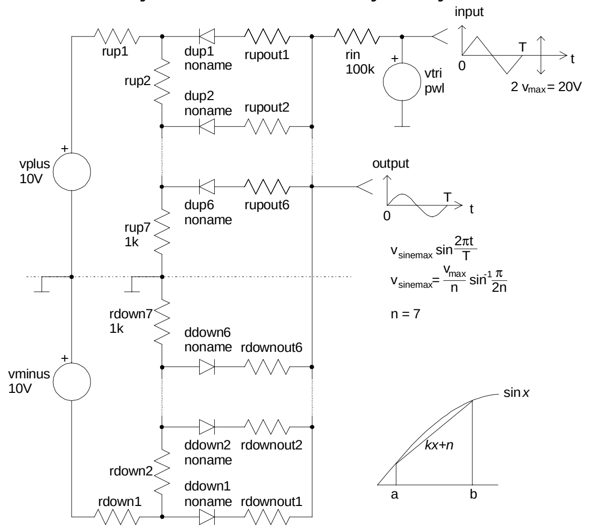
```
do
    if number of iterations > max break
    change parameter values
    verify explicit constraints and correct parameter values
        if necessary
    execute all analysis commands
        (analysis 0, analysis 1, ..., analysis N)
    number of iterations = number of iterations + 1
    if not implicit constraint 0 or not implicit constraint 1 or
        ... or not implicit constraint M continue
    calculate cost function
while termination criteria not satisfied
```

4 An optimisation example with *optimize* command

Let us now consider a circuit depicted in Figure 3, which converts triangular signal waveform to sine waveform. We would like to set the resistances in the circuit so that the purest possible sine signal can be obtained at the output.

Figure 3: Triangle to sine waveform converter

We will try to solve circuit analytically and calculate



the values of the unknown resistors. The first problem is where to choose break points. The relative error on one section equals (7) assuming $b - a < \pi/2$.

$$\varepsilon_r = \frac{\int_a^b \sin x\,dx - \int_a^b (kx+n)\,dx}{\int_a^b \sin x\,dx} = \tag{7}$$

$$= 1 - \frac{b-a}{2}\operatorname{ctg}\frac{b-a}{2} \approx \frac{(b-a)^2}{12}$$

This result tells us that the best choice are equidistant break points. If we assume ideal diodes with cutin voltage ≈ 0.5V and $R_{\mathrm{up}i} << R_{\mathrm{upout}j}$, than resistor values can be calculated (Table 1).

Now we will optimise our circuit with the *optimize* command. Because of symmetry of the circuit we will observe only one half of the circuit in one quarter of a period. In this case we have 12 parameters to optimise $R_{\mathrm{up}1}$ ... $R_{\mathrm{up}6}$, $R_{\mathrm{upout}1}$ ... $R_{\mathrm{upout}6}$. The explicit constraints (Table 1) are set around analytically calculated values. There are no implicit constraints. Let the cost function be defined as normalised square area between the real and ideal response:

$$E = \frac{\int_0^{T/4} (v_{\mathrm{out}}(t) - v_{\mathrm{sine}}(t))^2\,dt}{\int_0^{T/4} v_{\mathrm{sine}}^2(t)\,dt} \tag{8}$$

It is a positive scalar and the circuit with ideal sine response would have its value zero.

The initial point of optimisation method is approximately the center of the explicitly constrained parameter space.

Table 1: Resistor values (all values are rounded)

	analytic	exp. const.	initial	optim.
R_{up1}	4.6k	1k ÷ 10k	5k	3.8k
R_{up2}	510	100 ÷ 1k	500	480
R_{up3}	820	100 ÷ 1k	500	800
R_{up4}	1.1k	500 ÷ 5k	2.5k	670
R_{up5}	1.3k	500 ÷ 5k	2.5k	1k
R_{up6}	1.5k	500 ÷ 5k	2.5k	1.1k
R_{upout1}	17k	100 ÷ 50k	25k	9.1k
R_{upout2}	88k	10k ÷ 100k	50k	83k
R_{upout3}	220k	50k ÷ 500k	250k	270k
R_{upout4}	430k	100k ÷ 1meg	500k	560k
R_{upout5}	830k	100k ÷ 1meg	500k	740k
R_{upout6}	1.9meg	500k ÷ 5meg	2.5meg	2.2meg

We ran the optimisation with different optimisation methods from the same initial point. The parameters of all methods were set to their default values. So results (Table 2) with the particular method could be improved by fine tuning method parameters. Anyway we can see that gradient methods have local character. They are quickly trapped in a local minimum, which can be just a consequence of numerical noise. Random and genetic algorithm search across whole parameter space but they need a lot of iterations to get satisfying results. Direct search methods performed better. Hooke-Jeeves's method and constrained simplex method were the best among direct search methods. Hooke-Jeeves converged fast, but constrained simplex gave slightly better results. Time needed for optimisation was measured on 350Mhz Pentium personal computer running Windows 95. Optimal parameter values obtained by constrained simplex method are shown in Table 1.

Table 2: Optimisation statistics

	Iterations	cost value	time [s]
initial		$2.7\ 10^{-3}$	
analytic		$3.2\ 10^{-5}$	
steepest descent	114	$7.4\ 10^{-5}$	6
newton	610	$4.8\ 10^{-4}$	30
dfp	315	$3.8\ 10^{-5}$	16
monte carlo	4000	$3.8\ 10^{-5}$	180
grid search	4092	$1.1\ 10^{-5}$	197
axis search	878	$2.1\ 10^{-5}$	40
powell	1193	$2.1\ 10^{-5}$	55
hooke jeeves	1320	$8.7\ 10^{-6}$	68
complex	3931	$6\ 10^{-6}$	209
sga	4160	$3.5\ 10^{-5}$	194

To check if we improved analytically calculated circuit, we will take a closer look after the harmonic distortion of the output signal. We can see (Table 3) that optimisation method starting from initial circuit, which is worse than analytical circuit, gave better results than analytically calculated. The reason is in taking real diodes into account instead of ideal diodes in analytic circuit.

Table 3: Harmonic distortion (normalised magnitudes)

	initial	analytic	optimised
f_0	1	1	1
$2\ f_0$	$4.1\ 10^{-6}$	$3.4\ 10^{-8}$	$2.4\ 10^{-11}$
$3\ f_0$	$3.8\ 10^{-2}$	$4.6\ 10^{-3}$	$7.4\ 10^{-4}$
$4\ f_0$	$4\ 10^{-6}$	$3.4\ 10^{-8}$	$2.4\ 10^{-11}$
$5\ f_0$	$1.6\ 10^{-2}$	$2.9\ 10^{-3}$	$1.3\ 10^{-3}$
$6\ f_0$	$4\ 10^{-6}$	$3.3\ 10^{-8}$	$2.4\ 10^{-11}$
$7\ f_0$	$2.1\ 10^{-3}$	$1.2\ 10^{-3}$	$1.3\ 10^{-3}$
$8\ f_0$	$3.9\ 10^{-6}$	$3.3\ 10^{-8}$	$2.3\ 10^{-11}$
$9\ f_0$	$1.1\ 10^{-3}$	$1.3\ 10^{-3}$	$3.7\ 10^{-4}$

5 Conclusions

A new *optimize* command was added to *Nutmeg*. It represent a general optimising tool. Arbitrary circuits can be optimised and arbitrary cost function can be defined. The number of parameters is also not constrained, and explicit and implicit constraints can be defined.

Several optimising methods were built in the *optimize* command. The well known difficulties of gradient methods shown up. On the other hand direct methods worked better. The Hooke-Jeeves's and constrained simplex method gave the best results on real circuits, but we will still have to do some tests on more real cases.

References

[1] Laurence W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", Memorandum No. ERL-M520, University of California, Berkeley, 1975

[2] Thomas L. Quarles, "Analysis of Performance and Convergence Issues for Circuit Simulation", Memorandum No. ERL M89/42, University of California, Berkeley, 1989

[3] Janez Puhan, Tadej Tuma, "Optimisation of analog circuits with SPICE 3f4", Proceedings of ECCTD '97, Volume 1, pages 177 - 180, Budapest, 1997

[4] T. Quarles, A. R. Newton, D. O. Pederson, A. Sangiovanni - Vincentelli, "SPICE3 Version 3f3 User's Manual", Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1993