Technical Report #9405

# SYMBOLIC ANALYSIS IN PARAMETER EXTRACTION ITS IMPLEMENTATION AND PERFORMANCE

by

W.M. Zuberek[†] and A. Konczykowska[‡]

[†] Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1C-5S7

[‡] Centre National d'Etudes des Télécommunications
Laboratoire de Bagneux
92220 Bagneux, France

November 1994

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5
tel: (709) 737–8627
fax: (709) 737–2009

Copyright © 1994 by W.M. Zuberek and A. Konczykowska.
All rights reserved.

The Natural Sciences and Engineering Research Council of Canada
partially supported this research through Research Grant A8222.

# SYMBOLIC ANALYSIS IN PARAMETER EXTRACTION ITS IMPLEMENTATION AND PERFORMANCE

**Abstract**

An interface between a symbolic analysis tool and a SPICE–like circuit simulation package has been developed in order to integrate numerical and symbolic circuit analyses. In effect, both numerical and symbolic analyses use the same internal representation of circuits which makes the two approaches truly complementary. This integrated simulation capability is used in simulation–based parameter extraction where all ac small-signal parameters are fitted through the symbolic analysis rather than numerical one, significantly reducing the execution time of the extraction process.

**Résumé**

Une interface entre un programme d'analyse symbolique et une bibliothèque de modules pour un simulateur de type SPICE a été développée afin d'intégrer les analyses numérique et symbolique. En effet, ces deux analyses utilisent une même représentation interne des circuits, ce qui rend les deux approches pleinement complémentaires. Cette intégration est utilisée dans un extracteur de paramètres basé sur la simulation, dans lequel les paramètres petit–signal (ac) sont ajustés par une analyse symbolique, plutôt que numérique, ce qui conduit à une réduction importante du temps d'exécution de l'extraction.

**Acknowledgement**

Collaboration with Michel Bon and Jean Godin, both of Centre National d'Etudes des Télécommunications, Laboratoire de Bagneux, Bagneux, France, is gratefully acknowledged.

## 1. INTRODUCTION

Because of rapid developments in semiconductor technologies, increasing performance and complexity of circuits, verifying designs through simulation has become an indispensable part of IC design process. This continuously creates demand for new and more efficient analog methods for circuit analysis. Integration of numerical and symbolic circuit analyses is one of possible improvements that can be used to increase the efficiency of circuit analysis tools.

Reliable computer–aided circuit analysis or circuit simulation cannot be obtained without accurate specification of circuit elements and device models. Existing device models use large sets of parameters, values of which must be properly determined to represent device characteristics accurately. Because of highly nonlinear device models, these parameters usually cannot be determined by direct measurements; popular extraction methods use iterative techniques to minimize differences between measurement data and model behavior in the full range of operating conditions.

Iterative extraction of model parameters can be regarded as an optimization process [BCYZ,BST,CCLL,DoSc,Garw] which minimizes the (total) differences between a set of measurement data and the corresponding circuit responses by adjusting the values of model parameters (which are optimization variables). The result of this optimization determines such values of model parameters for which the circuit responses are "as close as possible" to the measurement data (in the sense of the error function used).

One of flexible approaches to parameter extraction is to use a circuit simulator rather than a set of model equations (such an approach is called simulation–based parameter extraction). An important advantage of the simulation–based method is that the extractor can use all the capabilities of the circuit simulator, so all packaging and mounting parasitics can easily be taken into account during extraction, and the extraction can use many types of measurement data, including noise, distortion, etc. On the other hand, repeated simulations can easily become rather time–consuming, especially when numerous parameters are extracted from large sets of measurement data. In some cases (e.g., parameter extraction for microwave applications), groups of parameters correspond to linear analyses of the circuit. For linear analyses, the dependence of circuit responses on some variables can be derived in a symbolic form, and this symbolic form can be used very efficiently in repeated analyses of the same circuit for different combinations of values of the variables. Therefore, a circuit simulation package used for simulation–based parameter extraction (the FIT extractor) has recently been enhanced by an interface to symbolic circuit analysis. The interface is composed of a number of functions which provide a convenient access from the symbolic analyzer to the internal representation of the analyzed circuit. The input module of an existing symbolic analyzer (SYBILIN) has been modified to conform to the interface from a SPICE-like numerical simulator. This integrated numerical–symbolic simulation is used in a simulation–based data–driven parameter extraction program called FIT–S [ZK].

The concept of data–driven parameter extraction is related to flexibility of the extractor, a property quite important because of efficiency as well as the convergence properties of the extraction process. The popular optimization methods provide local optimization only, so in a case of numerous local minima, the starting point should be disturbed externally to

cover as large part of the feasible space as seems reasonable. However, local optimization algorithms are seldom satisfactory even when restarted from several randomly chosen initial points. Measurement error coupled with the large number of variables of a physically based circuit leads to an error function with many nonphysical local minima in addition to the global minimum [BST]. Therefore, more general (and efficient) global optimization methods are needed but they are rather difficult to find. However, quite often the extraction process can be decomposed into a sequence of "partial extractions", performed for small subsets of carefully selected parameters and relevant subsets of measurement data. Such an approach can eliminate numerous local minima and also minimize the number of iterations required to reach the solution.

In the data–driven organization, the extraction process is controlled by the (selected) variables and data. The FIT extractor [ZKAW] has built–in facilities to select variables and data interactively (so the next selection of variables and data may depend upon the results of the previous step) and the extractions can be performed on arbitrary subsets of variables and measurement data.

The measurement data normally include several types of data, for example, DC measurements, frequency–domain (AC) and/or time–domain (TR) measurements (used for large–signal analysis of the periodic steady–state); because of the simulation–based organization, FIT can also deal with harmonic and noise measurements. Measurements of the same type (e.g., AC for a given bias point, steady–state time–domain for a given frequency, etc.) form a data "group", so the collection of measurement data is simply a sequence of data groups. There is no limit imposed on the number or composition of data groups. In fact, a section of one data group can be repeated (with more data points) as another data group to provide a better fit in regions which are believed to be more important or more difficult for fitting (e.g., initial parts of characteristics or highly nonlinear regions).

A general scheme of data–driven extraction can be illustrated by the following outline of a typical "partial extraction":

```
select(variables);
select(data_groups);
continue_optimization := true;
while continue_optimization do
   update_the_values_of_variables;
   error := 0;
   for each selected_data_group do
      adjust_circuit_parameters;
      find_simulation_results(results);
      error := error + differences(data,results)
   endfor;
   if error < accuracy then continue_optimization := false
   else get_new_values_of_variables(error) endif
endwhile;
display_the_results;
```

where the `select` operations and the optimization are performed by interactive commands, the results of optimizations are presented to the user in some convenient form (e.g., as a

graphical plot), and the optimization loop can also be terminated if no progress is detected, or one of the optimization limits is reached.

## 2. NUMERICAL AND SYMBOLIC SIMULATION

The popular "third–generation" (numerical) circuit simulators [Ped] use a modified form of nodal analysis (modified to take care of voltage sources, floating sources, and inductive elements) and Newton–Raphson iteration to solve the system of simultaneous nonlinear algebraic equations [Coh,MC]

$$F(X) = 0$$

which describes the balance of currents at the nodes of the network in terms of node voltages (and some branch currents) $X$.

Let the solution be denoted by $X^*$. The Newton–Raphson iteration solves the original system of nonlinear equations through a sequence of linear approximations to the nonlinear function $F(X)$ at points $X^{(j)}$, $j = 1, 2, ...$

$$F(X^{(j)}) + G(X^{(j)})(X^* - X^{(j)}) \approx 0$$

where $G$ is the Jacobian of $F$ with respect to $X$ (evaluated at $X^{(j)}$). The $(j + 1)$ approximation to the solution $X^*$ is obtained by solving a system of simultaneous linear equations with respect to the correction $\Delta^{(j)}$

$$G(X^{(j)})\Delta^{(j)} = -F(X^{(j)})$$

and $X^{(j+1)} = X^{(j)} + \Delta^{(j)}$. The iteration terminates when $\Delta^{(j)}$ is sufficiently small.

The "main computational effort" is thus devoted to: (i) evaluating the Jacobian $G$ and the function $F$, and then (ii) solving the system of linear equations.

This basic scheme is used in the DC operating point, DC transfer curve, and even time–domain analysis; in the last case, the dependence upon time is eliminated by approximating the differential equations by difference equations [MC,VS]. Only frequency–domain (small–signal) analyses are significantly different because they require (for each frequency) a solution of a system of simultaneous linear equations in the complex domain; this is often done by separating the real and imaginary parts of coefficients and variables, and solving a twice as large system of linear equations in the real domain.

The principle of symbolic simulation [GS,Lin] is to derive analytic (or symbolic) network functions from a representation of a network rather than solve (numerically) the systems of circuit equations. This means that (some of) circuit parameters are represented by symbols in the derived functions, and then the circuit responses can be obtained very efficiently by evaluation of the derived analytic formulas.

For linear, lumped and stationary circuits, the transfer functions $\mathcal{H}(s)$ of two–port networks are in the form of rational functions of the complex frequency $s$:

$$H(x) \;=\; \frac{\mathcal{F}_j(s)}{\mathcal{F}_k(s)}$$

in which the numerator $\mathcal{F}_j(x)$ and the denominator $\mathcal{F}_k(x)$ are characteristic polynomials of the two–port:

$$\mathcal{F}_i(s) \;=\; \sum_{\ell=0}^{n_i} s^{\ell} \mathcal{P}_{i\ell}(x_1, ..., x_m)$$

and the coefficients $\mathcal{P}(x_1, ..., x_m)$ are (nested or expanded) polynomial functions in symbolic elements $x_1, ..., x_m$. In the fully expanded form, the polynomial coefficients are in the "sum–of–product" form:

$$\mathcal{P}(x_1, ..., x_m) \;=\; \sum_{i=1}^{p} C_i \prod_{j=1}^{r} x_{ij}$$

where $C_i$ are real numbers, $x_{ij}$ are circuit symbols, and $p$ and $r$ depend upon the topology of the circuit.

By extracting common factors and rearranging the terms, the coefficients $\mathcal{P}(x_1, ..., x_m)$ can be represented equivalently as

$$\mathcal{F}_i \;=\; s^{k_i} \mathcal{T}_i \sum_{j=0}^{n_i} s^j \mathcal{R}_{ij}$$

where each $\mathcal{T}_i$ is a product of a constant $C_i$ and (some) symbols $x_{ik}$, $k = 1, ..., m_i$

$$\mathcal{T}_i \;=\; C_i \prod_{k=1}^{m_i} x_{ik}$$

and each $\mathcal{R}_{ij}$, $j = 0, 1, ..., n_i$, is a sum of products

$$\mathcal{R}_{ij} \;=\; \sum_{k=1}^{\ell_{ij}} C_{ijk} \prod_{\ell=1}^{m_{ijk}} x_{ijk\ell}$$

An important aspect of integrated symbolic–numerical analysis is the representation of symbolic functions. FIT uses a collection of arrays (actually, vectors) which store (real) coefficients and (integer) indices to other arrays as well as symbol identifiers (which are integer indices in a "symbol table") for representation of (symbolic) characteristic polynomials. Several other representations are discussed and compared in section 6.

The representation of symbolic functions uses the following arrays:

- *Ntab* – integer, the degrees of characteristic polynomials,
- *Ktab* – integer, the exponents of $s$ associated with the products $\mathcal{T}_i$,

- *Ltab* – integer, the numbers of terms in $\mathcal{R}_{ij}$ sums,
- *Ctab* – real, the values of coefficients $C_i$ and $C_{ijk}$,
- *Mtab* – integer, the lengths of (i.e., the number of symbols in) products $\mathcal{T}_i$ and all terms of $\mathcal{R}_{ij}$,
- *Itab* – integer, the identifiers of symbols used in all products $\mathcal{T}_i$ and all terms of $\mathcal{R}_{ij}$.

The organization of these arrays for 5 symbolic functions, $i = 1, ..., 5$, is sketched in Fig.1.

| *Ntab* | *Ktab* | *Ltab* | *Ctab* | *Mtab* | *Itab* |
|---|---|---|---|---|---|
| $n_1$ | $k_1$ | $\ell_{10}$ | $C_1$ | $m_1$ | $x_{11}$ |
| $n_2$ | $k_2$ | $\ell_{11}$ | $C_{101}$ | $m_{101}$ | ... |
| $n_3$ | $k_3$ | ... | ... | ... | $x_{1m_1}$ |
| $n_4$ | $k_4$ | $\ell_{1n_1}$ | $C_{10\ell_{10}}$ | $m_{10\ell_{10}}$ | $x_{1011}$ |
| $n_5$ | $k_5$ | $\ell_{20}$ | $C_{111}$ | $m_{111}$ | ... |
| | | $\ell_{21}$ | ... | ... | $x_{101m_{101}}$ |
| | | ..... | $C_{11\ell_{11}}$ | $m_{11\ell_{11}}$ | $x_{1021}$ |
| | | | $C_{121}$ | $m_{121}$ | ..... |
| | | | ..... | ..... | $x_{10\ell_{10}m_{10\ell}}$ |
| | | | $C_{1n_1\ell_{1n_1}}$ | $m_{1n_1\ell_{1n_1}}$ | $x_{1111}$ |
| | | | $C_2$ | $m_2$ | .... |
| | | | $C_{201}$ | $m_{201}$ | $x_{11\ell_{11}m_{11\ell}}$ |
| | | | ..... | ..... | $x_{1211}$ |
| | | | | | ..... |
| | | | | | $x_{1m_1\ell_{1m_1}m_{11\ell}}$ |
| | | | | | $x_{21}$ |
| | | | | | ..... |

Fig.1. Representation of symbolic functions.

Symbolic simulators use different circuit representations and different algorithms to derive network functions. The algorithm used in SYBILIN [KMGB], the symbolic simulator

integrated with FIT, uses the Coates flowgraph representation, in which variables corresponding to graph nodes are the same as those used in the modified nodal analysis.

For the Coates flowgraph representation, the calcution of all two–terminal immitances (or two–port transfer functions) is based on the evaluation of 0–connections. 0–connection of a graph $\mathcal{G}$ is a subgraph composed of node–disjoint directed loops incident with all nodes of the graph [SK].

If $\mathcal{D}$ is an $n \times n$ matrix of a Coates graph with each element $d_{ij}$ denoting the set of directed edges from $i$ to $j$, then the set $Z$ of 0–connections of this graph is defined by:

$$Z \;=\; \bigcup_{(i_1,...,i_n)\in I_n} d_{1i_1} \times d_{2i_2} \times ... \times d_{ni_n}$$

where $I_n$ is the set of all permutations of the sequence $(1, 2, ..., n)$.

Characteristic functions are determined from the corresponding sets of 0–connections (i.e., sets of 0–connections of subgraphs of the Coates graph corresponding to cofactors of the indefinite admittance matrix):

$$\mathcal{F}_i(s) \;=\; \sum_{z\in Z_i} (-1)^{n+\ell(z)} \prod_{e\in z} y(e)$$

where:

$Z_i$ – is the set of 0–connections of the Coates graph corresponding to $\mathcal{F}_i$,
$n$ – is the number of the nodes of the graph,
$\ell(z)$ – is the number of loops in the 0–connection $z$,
$y(e)$ – is the (symbolic) admittance of element $e$ (or its equivalent description).

In order to eliminate repetition of identical computations, the symbolic analysis is performed in two stages. The first stage, performed only once, creates the Coates graph of the analyzed circuit and then generates symbolic products for each of the characteristic functions:

```
generate_Coates_graph;
for each characteristic_function do
   update_Coates_graph;
   generate_and_store_symbolic_products
endfor;
```

The second stage converts the stored symbolic products into the set of arrays representing the symbolic functions (Fig.1).

## 3. SYMBOLIC ANALYSIS IN PARAMETER EXTRACTION

For parameter extraction in general, but especially in the case of microwave applications, a significant part of the extraction process analyzes the small–signal, linear behavior of the circuit (which is typically rather small, with less than 10 nodes and less than 15 symbols). These linear analyses can conveniently be performed using symbolic simulation rather than

numerical one, providing circuit responses very efficiently from symbolic functions. Moreover, it is often the case that the extraction of a set of parameters is decomposed into a sequence of "partial extractions", performed on subsets of parameters and relevant subsets of measurement data [KZD]. For such partial extractions, the sets of parameters usually contains only a few symbols, which means that the corresponding symbolic functions are also quite simple.

For simulation–based parameter extraction (as implemented in the FIT program) the (iterative) frequency–domain analyses are performed within a general optimization scheme [ZK]:

```
continue_optimization := true;
while continue_optimization do
   update_the_values_of_variables;
   error := 0;
   for each frequency_domain_data_group do
      update_op_point_voltages_and_currents;
      find_the_op_point_solution;
      for each frequency do
         find_the_solution_of_linear_equations(results);
         error := error + differences(data,results)
      endfor
   endfor;
   for each non_frequency_domain_data_group do
      find_circuit_responses(results);
      error := error + differences(data,results)
   endfor;
   if error < accuracy then continue_optimization := false
   else get_new_values_of_variables(error) endif
endwhile;
```

Since all frequency–domain analyses are performed for the circuit with the same topology, the generation of symbolic functions can be done only once. Furthermore, the number of symbols which can change their values during one optimization cycle (or one "partial extraction") is rather small, and includes a subset of parameters which are updated in the optimization loop (i.e., which are optimization variables) and all those symbols which depend upon the operating point solution. All such symbols are called *variable* symbols while the remaining symbols are called *fixed* symbols. *Variable* symbols include *direct* symbols, i.e., symbols updated in the optimizations loop, and *dependent* symbols, i.e., symbols which depend upon the operating point solution. It should be observed that all *fixed* symbols can be replaced by their numerical values during the generation of the symbolic functions, reducing the functions and simplifying the subsequent evaluations.

The values of *variable* symbols can be retrieved in two steps: (i) at the beginning of the optimization loop (all *direct* symbols), and (ii) after each operating point solution (all *dependent* symbols). The values of *variable* symbols are used for a transformation of the symbolic functions to their reduced form:

$$\mathcal{F}_i^{(r)} \;=\; s^{k_i} A_i \sum_{j=0}^{n_i} s^j A_{ij}$$

where all $A_i$ and $A_{ij}$, $j = 0, 1, ..., n_i$, are constants provided that no frequency–dependent elements are used. Only this very simple polynomial form needs to be evaluated in the innermost (i.e., frequency) loop.

Symbolic simulation can be included in the previous optimization scheme in the following way:

```
retrieve_the_values_of_all_fixed_symbols;
generate_symbolic_products;
continue_optimization := true;
while continue_optimization do
   update_the_values_of_variables;
   retrieve_the_values_of_direct_symbols;
   delta := 0;
   for each frequency_domain_data_group do
      update_op_point_voltages_and_currents;
      find_the_op_point_solution;
      retrieve_the_values_of_dependent_symbols;
      evaluate_products;
      for each frequency do
         evaluate_reduced_functions(values);
         convert_to_circuit_responses(values,results);
         delta := delta + differences(data,results)
      endfor
   endfor;
   for each non_frequency_domain_data_group do
      find_circuit_responses(results);
      delta := delta + differences(data,results)
   endfor;
   if delta < accuracy then continue_optimization := false
   else get_new_values_of_variables(delta) endif
endwhile;
```

where the step `generate_symbolic_products` generates the products $\mathcal{T}_i$ and $\mathcal{R}_{ij}$ (Section 2), while the step `evaluate_products` calculates the values of $A_i$ and $A_{ij}$ using the retrieved values of variable symbols.

For the representation of symbolic products shown in Section 2, the outline of the step `evaluate_products` is as follows (`Nf` is the number of characteristic polynomials);

```
il := 0;
im := 0;
is := 0;
for i := 1 to Nf do
   A[i] := Product(im,is);
   for j := 0 to Ntab[i] do
      sum := 0.0;
      il := il + 1;
      for l := 1 to Ltab[il] do
         sum := sum + Product(im,is)
      endfor;
```

```
      A[i,j] := sum
   endfor
endfor;
```

where the real function `Product` increments the indicators `im` and `is`, so "passing by reference" is assumed (`ST` denotes the "Symbol Table", i.e., an array containing the values of all symbols (as well as their attributes)):

```
real function Product (int im, int is);
begin
   real val;
   int last;
   im := im + 1;
   val := Ctab[im];
   last := is + Mtab[im];
   while is < last do
      is := is + 1;
      val := val * ST[Itab[is]]
   endwhile;
   return val
end;
```

## 4. INTEGRATION OF NUMERICAL AND SYMBOLIC SIMULATION

Any integration of numerical and symbolic simulations must provide some sort of interaction between these two types of analyses. In the FIT–S program, the interaction is performed through an interface which supports the following operations (implemented as interface procedures):

- RESET(NAME),
- NEXTEL(DESC,TYPE,NODES,LEN),
- GETVAL(DESC,TYPE,VALUES,LEN).
- GETVAR(NAMES,LEN).

RESET must always be used as the first operation, before any other operation of the interface; it initializes extraction of circuit elements for symbolic analysis; its parameter NAME is either "∗" which indicates all elements of the simulated circuits, or it must be a name of a subcircuit expansion (i.e., an `X`–name in the SPICE convention) which indicates the subcircuit for symbolic analysis.

NEXTEL returns the descriptor DESC, the type TYPE, and the list of nodes NODES of length LEN of the next circuit element (or indicates that the "next" element does not exist); it is implemented in such a way that consecutive invocations of this operation return

descriptions of consecutive circuit elements (according to the internal representation of the circuit); zero returned as the value of DESC indicates that there are no more elements.

GETVAL uses the vector VALUES to return the numerical value(s) of parameters associated with an element identified by DESC and TYPE; LEN is set to the number of values returned in VALUES.

GETVAR uses the character vector NAMES to return the identifiers (or names) of all declared variables (i.e., symbols) used in symbolic functions (in the internal representation of symbolic functions all symbols are replaced by numbers indicating their positions in the vector NAMES); NUM returns the number of symbols.

Typical sequence of interface operations (during generation of symbolic functions) is as follows:

```
getvar(vnames,len);
reset(cktname);
graph := 0;
nextel(desc,type,nodes,num);
while desc > 0 do
   add_to_flowgraph(id,nodes,graph);
   add_to_symbol_table(desc,type,id);
   if fixed_symbol(desc) then
      getval(desc,type,value,len);
      store_in_symbol_table(id,value,"fixed")
   endif;
   nextel(desc,type,nodes,num)
endwhile;
generate_intermediate_representation(graph);
```

The function `fixed_symbol(desc)` first checks if the symbol identified by `desc` is *dependent* and returns FALSE if it is; then it checks if the symbol `desc` belongs to variables returned by `getvar` (in `vnames`); if it does, the symbols is *direct* and `fixed_symbol` returns FALSE, otherwise it is a *fixed* symbol, so `fixed_symbol` returns TRUE.

The symbol table combines all attributes of all symbols used in the simulation and extraction. These attributes include the class of symbols (fixed, direct, dependent), which is used for selective retrieval of values. For example, after each solution of the operating point, the values of all operating-point dependent symbols are retrieved from the circuit description and stored in the symbol table:

```
for each symbol in symbol_table do
   if dependent(symbol) then
      getval(desc,type,value,len);
      store_in_symbol_table(id,value)
   endif
endfor;
```

The table of symbols is used during evaluation of all coefficients $A_i$ and $A_{ij}$ of reduced functions.

**Note:** Symbolic simulation can be used only if the circuit description file contains a directive `.ACSYMB`, a new directive introduced for integrated symbolic analysis [ZK], which specifies the subcircuit for the small–signal analysis, its input port and its output port. The subcircuit must be indicated by its expansion (an "X"–class name) or "*" if the whole circuit is to be analyzed. The ports are indicated by pairs of nodes, and if the second node is the "reference" node (i.e., "0"), it can be omitted; for example:

```
.ACSYMB ckt(Xsub1) input(11) output(22)
```

(the order of input, output and circuit sections is immaterial).

## 5. EXAMPLE

A comparison of numerical and symbolic simulation is given for parameter extraction of a submicron (0.25 $\mu$) GaAs FET on InP substrate. A small-signal model (with its parameters) is shown in Fig.2.

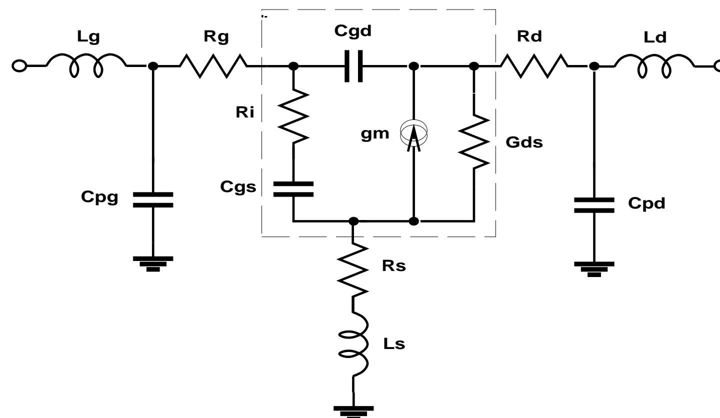



Fig.2. GaAs FET small-signal model.

For the model shown in Fig.2, the (five) reduced symbolic functions are polynomials of degrees 3, 6, 7, 5 and 6, while the exponents of the common factors are equal to -2, -3, -3, -3 and -3.

Polynomial evaluations can be performed in the complex domain, or in the real domain; in the second case, the evaluation can be performed for combined real and imaginary parts or independently for the two parts. The evaluation times for 1,000,000 evaluations of polynomials of several different degrees (the text of the testing program is given in Appendix 1) are compared in the following table (the execution times are in seconds, on a SPARCstation 2):

| *polynomial's degree* | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| real domain – independent Re/Im | 6.97 | 8.02 | 8.95 | 9.85 | 10.8 | 11.7 | 11.8 |
| real domain – combined Re/Im | 7.44 | 9.47 | 11.5 | 13.5 | 15.5 | 17.6 | 19.5 |
| complex domain | 12.4 | 16.8 | 21.3 | 26.0 | 30.3 | 34.8 | 39.3 |

For the real domain, the "combined" evaluation uses twice as many multiplications as the "independent" scheme (it updates the real and imaginary parts for consecutive values of the exponent); the "independent" scheme evaluates the real part for even values of the exponent and the imaginary part for odd values of the exponent only. The evaluation in the complex domain (which uses a subroutine for complex multiplication in double precision) is clearly unattractive. Therefore FIT–S performs all evaluations of the symbolic functions in the real domain, independently for the real and imaginary parts.

The evaluation of the symbolic functions is only a rather small part of all computations involved in parameter extraction; the values of symbolic functions must be converted into S–parameters (or some other form which is used by the measurement data), they must be stored in a database of results, compared with the corresponding measurement values to update the value of the error function, etc. Therefore, a more realistic comparison of symbolic and numerical analysis is obtained by measuring the total execution times for a typical extraction process. Such comparison results, with corresponding values of the speedup, are shown in the following table in which the columns correspond to data groups with 10, 20, 50 and 100 frequency values (the execution times are in seconds, on a SPARCstation 2, for 100 iteration steps):

| *number of frequencies (per data group)* | 10 | 20 | 50 | 100 |
|---|---|---|---|---|
| execution time – symbolic analysis | 3.30 | 3.62 | 4.32 | 5.61 |
| execution time – numerical analysis | 8.95 | 12.9 | 24.8 | 44.6 |
| speedup | 2.7 | 3.6 | 5.7 | 8.0 |

The number of variable symbols influences the execution time rather insignificantly, especially for data groups with large numbers of frequency values; the number of variable symbols affects the evaluation of the coefficients of reduced functions, but this evaluation is performed only once for each data group.

## 6. OTHER REPRESENTATIONS

In addition to the "table" representation described in Section 2, several other methods were considered for representation of symbolic functions but were found overall less attractive than the implemented one:

- high–level code, compiled and linked with FIT–S; in this approach, the symbolic functions are generated in the source form (as a Fortran, C, or any other popular language routine) which is then compiled and linked with the program; any invocation of this routine (with the frequency as a parameter) returns the corresponding values of characteristic polynomials which are used for computation of S–parameters (or some other results); the solution is rather inconvenient and definitely inefficient because of the compilation and linking involved;

- binary code; basically this approach is similar to the previous one, but both compilation and linking are eliminated by generating directly binary code which, when

invoked, determines the values of characteristic functions; this solution is the most efficient one, but it also is machine–dependent and generally not portable (because of machine dependence);

- postfix notation (reverse Polish); the (reduced) symbolic functions are represented in postfix notation (as a sequence of arguments and operators) and evaluated by a simple stack–based interpreter; for example, a polynomial of degree 3, which in infix parenthesized Horner's form is:

$$\mathcal{P}^{(3)}(s) \;=\; ((a_3 * s + a_2) * s + a_1) * s + a_0$$

in the postfix notation is:

$$postfix(\mathcal{P}^{(3)}(s)) \;=\; a_3\, s * a_2 + s * a_1 + s * a_0 +$$

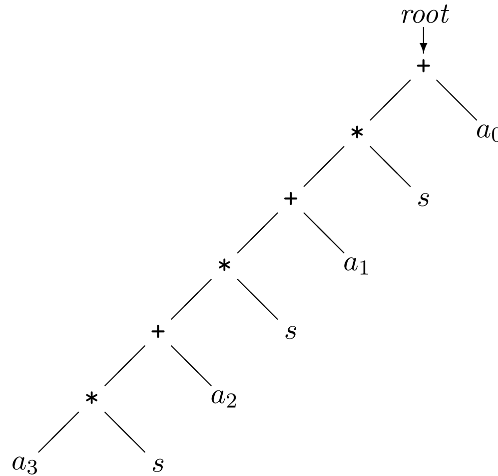
The evaluation of postfix expressions uses an argument stack with operations `push(x)` storing its argument on the "top" of the stack and a function `pop` which returns the top element of the stack and deletes this top element from the stack ("uncovering" the second top element). Assuming that the reverse representation is stored in an array $R$ in which it is terminated by a "special" element, the evaluation scheme is:

```
i := 0;
cont := true;
while cont do
   i := i + 1;
   a := R[i];
   if argument(a) then push(a)
   else if operator(a) then
      y := pop;
      x := pop;
      push(apply(a,x,y))
   else
      cont := false
   endif
endwhile;
value := pop;
```

where the logical functions `argument(a)` and `operator(a)` return TRUE is the argument `a` is an argument or an operator, respectively, and `apply(a,x,y)` simply applies the operator "$a$" to its arguments $x$ and $y$ (assuming that $a$ is a binary operator), so `apply(+,x,y)` returns the values of $x + y$. The result of evaluation is left on the stack, so the last `pop` operation retrieves it from there.

For evaluations in the real domain, two postfix expressions are generated and evaluated, one for the real and the second for the imaginary parts.

- parse trees; the (reduced) symbolic functions are represented by trees (which for binary operators are binary trees); the polynomial $\mathcal{P}^{(3)}(s)$ is represented by the following parse tree:




Evaluation is performed by a recursive scheme which implements a depth–first traversal of the tree (`root` indicates the root of the tree; the two outgoing branches (if any) are denoted `root.left` and `root.right`, the operator associated with it `root.oper`, and the argument `root.val`; it should be observed that all argument nodes have `nil` (or "empty") outgoing branches, while for binary operators, both branches are non–empty):

```
real function eval(tree root);
begin
  if root.left = nil and root.right = nil then return root.val
  else apply(root.oper,eval(root.left),eval(root.right)) endif
end;
```

For evaluation in the real domain, each polynomial is represented by a pair of trees, one for the real part and the second for the imaginary part of the polynomial.

For comparison, the following table shows the execution times of 1,000,000 evaluations of polynomials of several degrees using the binary code, postfix notation and parse tree representations; all evaluations are performed in the real domain, and the execution times are in seconds, for a SPARCstation 2 (the corresponding code is shown in Appendix 2, partially in Fortran and partially in C – because of pointers used in parse tree representation):

| *polynomiaal's degree* | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| binary code | 6.70 | 7.02 | 7.30 | 7.70 | 8.07 | 8.42 | 8.68 |
| postfix notation | 17.9 | 24.7 | 31.5 | 38.1 | 44.8 | 51.7 | 58.0 |
| parse tree | 14.1 | 19.6 | 25.3 | 39.7 | 54.3 | 76.2 | 101.0 |

The evaluation times for the binary code are only slightly better than those for the table representation discussed in the previous section.

It should be noted that the evaluation times of the polynomials shown in the table above account for a rather small part of the total time of symbolic analysis. The total evaluation time of (reduced) characteristic functions (i.e., the evaluation of the polynomials as well as the exponential factors) and its contribution to the execution time of symbolic analysis is as follows (the execution times are in seconds, for a SPARCstation 2, and correspond to 100 iteration steps):

| *number of frequencies (per group)* | 10 | 20 | 50 | 100 |
|---|---|---|---|---|
| execution time (symbolic analysis) | 3.30 | 3.62 | 4.32 | 5.61 |
| evaluation time (reduced functions) | 0.13 | 0.26 | 0.64 | 1.29 |
| percent of total execution time | 3.9 | 7.2 | 14.8 | 23.0 |

If the conversion to S–parameters is taken into account, the comparison is as follows:

| *number of frequencies (per group)* | 10 | 20 | 50 | 100 |
|---|---|---|---|---|
| execution time (symbolic analysis) | 3.30 | 3.62 | 4.32 | 5.61 |
| evaluation time (results) | 0.23 | 0.45 | 1.12 | 2.27 |
| percent of total execution time | 7.0 | 12.4 | 25.9 | 40.5 |

The difference between the total execution times and the evaluation times shown in the table is almost independent of the number of frequencies; this difference corresponds to: (i) updating group parameters, (ii) finding operating point solutions, (iii) retrieving the values of all variable symbols, (iv) evaluation of coefficients of reduced functions, and (v) storing the results and evaluation of the error function (only this part depends upon the number of frequencies, but its contribution is rather insignificant).

Since the differences between different representations influence the overall performance rather insignificantly, other considerations (like memory requirements and simplicity of implementation) may have more influence on the choice of the representation than just the execution time of the evaluation scheme.

## 7. CONCLUDING REMARKS

An integration of symbolic approach with traditional numerical computer–aided circuit analysis can significantly reduce the simulation time. This reduction can be used for more sophisticated simulation strategies, which – in general – are more computationally demanding.

In the case of parameter extraction, the analyzed circuits are rather small (typically they contain less than 10 nodes and less than 15 elements), so the symbolic functions are relatively simple and no function approximations are really needed. Moreover, many symbols can usually be eliminated during the generation of symbolic function because their values cannot change during the extraction (*fixed* symbols). More general applications of symbolic and

integrated numerical/symbolic simulation must take into account that for larger circuits the symbolic functions become very complex, so additional function simplification is required [GS].

The presented approach was developed with the assumption that no frequency–dependent elements are used in the small-signal analysis, because the proposed "reduced" deos not support frequency–dependency. If this assumption is not true and frequency–dependent elements are to be taken into account, the approach must be modified by introducing slightly different reduction step, which creates "reduced symbolic products" by elimination all frequency–independent symbols. These (usually small) reduced symbolic products must be evaluated for each frequency (instead of simple polynomials). An outline of the modified reduction step, which creates (new, reduced) tables *LtabR*, *CtabR*, *MtabR* and *ItabR*) can be as follows:

```
il := 0;
im := 0;
is := 0;
imr := 0;
isr := 0;
for i := 1 to Nf do
   sbase := isr;
   imr := imr + 1;
   CtabR[imr] := Reduce(im,is,isr);
   MtarR[imr] := isr - sbase;
   for j := 0 to Ntab[i] do
      sum := 0.0;
      imbase := imr;
      il := il + 1;
      for k := 1 to Ltab[il] do
         isbase := isr;
         val := Reduce(im,is,isr);
         if isr = isbase then
            sum := sum + val
         else
            imr := imr + 1;
            CtabR[imr] := val;
            MtabR[imr] := isr - isbase
         endif
      endfor;
      if sum <> 0.0 then
         imr := imr + 1;
         CtabR[imr] := sum;
         MtabR[imr] := 0
      endif;
      LtabR[il] := imr - imbase;
   endfor
endfor;
```

where the procedure `Reduce` processes one symbolic product, checking for frequency–dependent symbols, and returning the product of all frequency–independent symbols (while copying

all frequency–dependent symbols to *ItabR*):

```
real function Reduce (int im, int is, int isr);
begin
   real val;
   int last;
   im := im + 1;
   val := Ctab[im];
   last := is + Mtab[im];
   while is < last do
      is := is + 1;
      if frequency_dependent(symbol(Itab[is])) then
         isr := isr + 1;
         ItabR[isr] := Itab[is]
      else val := val * ST[Itab[is]] endif
   endwhile;
   return val
end;
```

It should be observed that if there are no frequency–dependent symbols, the (completely) reduced form (all elements of *MtabR* are zeros and all elements of *LtabR* are equal to 1) is equivalent to a "reduced polynomial" but is less efficient in both representation and evaluation time than a simple polynomial discussed earlier.

In FIT–S, symbolic analysis is used for (small–signal) frequency–domain analyses only. If ongoing research succeeds in developing symbolic methods that can be applied to non–LLS (linear, lumped and stationary) circuits, further reduction of the "computational effort" required for simulation–based parameter extraction will be possible.

## References


[BCYZ] J.W. Bandler, S.H. Chen, S. Ye, Q-J. Zhang, "Integrated model parameter extraction using large–scale optimization concepts"; IEEE Trans. on Microwave Theory and Techniques, vol.36, no.12, pp.1629-1638, 1988.

[BST] G.L. Bilbro, M.B. Steer, R.J. Trew, C-R Chang, S.G. Skaggs, "Extraction of the parameters of equivalent circuits of microwave transistors using tree annealing"; IEEE Trans. on Microwave Theory and Techniques, vol.38, no.11, pp.1711-1718, 1990.

[CCLL] P. Conway, C. Cahill, W.A. Lane, S.U. Lidholm, "Extraction of MOSFET parameters using the simplex direct search optimization method"; IEEE Trans. on Computer-Aided Design, vol.4, no.4, pp.694-698, 1985.

[Coh] E. Cohen, "Program reference for SPICE 2"; Memorandum UCB/ERL M592, University of California, Berkeley, CA 94720, 1976.

[DoSc] K. Doganis, D.L. Scharfetter, "General optimization and extraction of IC device model parameters"; IEEE Trans. on Electron Devices, vol.30, no.9, pp.1219-1228, 1983.

[Garw] K. Garwacki, "Extraction of BJT model parameters using optimization method"; IEEE Trans. on Computer-Aided Design, vol.7, no.8, pp.850-854, 1988.

[GS] G. Gielen, W. Sansen, "Symbolic analysis for automated design of analog intergrated circuits"; Kluwer Academic Publ. 1991.

[KMGB] A. Konczykowska, V. Morin, J. Godin, M. Bon, "Symbolic analysis for CAD of microvawe circuits"; Proc. Symp. on Computer Aided Design of Microwave Circuits, London, England, 1985.

[KZD] A. Konczykowska, W.M. Zuberek, J, Dangla, "Parameter extraction with the FIT–2 program"; Proc. European Conf. on Circuit Theory and Design, Copenhagen, Denmark, pp.762–771, 1991.

[Lin] P-M. Lin, "Symbolic network analysis"; Elsevier 1991.

[MC] W.J. McCalla, "Fundamentals of computer-aided circuit simulation"; Kluwer Academic Publ. 1988.

[Ped] D.O. Pederson, "A historical review of circuit simulation"; IEEE Trans. on Circuits and Systems, vol.31, no.1, pp.103–111, 1984.

[SK] J. Starzyk, A. Konczykowska, "Flowgraph analysis of large electronic networks"; IEEE Trans. on Circuits and Systems, vol.33, no.3, pp.302-315, 1986.

[VS] J. Vlach, K. Singhal, "Computer methods for circuit analysis and design"; Van Nostrand Reinhold 1983.

[ZK] W.M. Zuberek, A. Konczykowska, "FIT–S, a simulation–based data–driven parameter extraction program"; Technical Report #9402, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada A1C 5S7, 1994.

[ZKAW] W.M. Zuberek, A. Konczykowska, C. Algani, H. Wang, J. Dangla, "Simulation–based parameter extraction, its implementation and some applications"; IEE Proc. on Circuits, Devices and Systems, vol.141, no.2, pp.129-134, 1994.

## APPENDIX 1

`TESTPOL` compares execution times for evaluation of polynomials in the real and complex domains for polynomials of several degrees.

```
      PROGRAM TESTPOL
      PARAMETER (NDEG=8)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(10)
      DATA A / 1.D1,9.D0,8.D0,7.D0,6.D0,5.D0,4.D0,3.D0,2.D0,1.D0 /
C ... set the number of repetitions
      NREP=1000000
      WRITE(*,100) NREP
  100 FORMAT(' test-pol : number of repetitions : ',I7)
C ... set the frequency
      OMEGA1=1.5D0
C ... print the header
      WRITE(*,200)
  200 FORMAT(/'  deg   real/i    real/c    complex')
C ... for consecutive degrees
      DO 80 N=1,NDEG
        T1=exectime()
        OMEGA=OMEGA1
        DO 91 II=1,NREP
          OMEGA=OMEGA+1.D0
          CALL EVPOLR(N,OMEGA,A,V1RE,V1IM)
   91   CONTINUE
        T2=exectime()
        OMEGA=OMEGA1
        DO 92 II=1,NREP
          OMEGA=OMEGA+1.D0
          CALL EVPOLS(N,OMEGA,A,V2RE,V2IM)
   92   CONTINUE
        T3=exectime()
        OMEGA=OMEGA1
        DO 93 II=1,NREP
          OMEGA=OMEGA+1.D0
          CALL EVPOLC(N,OMEGA,A,V3RE,V3IM)
   93   CONTINUE
        T4=exectime()
        WRITE(*,300) N,T2-T1,T3-T2,T4-T3
  300   FORMAT(I5,1P3D10.2)
   80 CONTINUE
      STOP
      END
C --- evpolc ----------------------------------------------------------*
C  This routine evaluates the polynomial in the complex domain; CMXMUL
C  performs complex multiplication in double precision.
C
      SUBROUTINE EVPOLC (NDEG,OMEGA,A,VALRE,VALIM)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(10)
      VALRE=A(NDEG+1)
      VALIM=0.D0
      DO 10 I=NDEG,1,-1
        CALL CMXMUL(VALRE,VALIM,0.D0,OMEGA)
        VALRE=VALRE+A(I)
```

```
   10 CONTINUE
      RETURN
      END
C --- evpolr -----------------------------------------------------------*
C  This routine evaluates the polynomial in the real domain.
C
      SUBROUTINE EVPOLR (NDEG,OMEGA,A,VALRE,VALIM)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(10)
      OMEGA2=-OMEGA*OMEGA
C ... first evaluate the real part
      NE=NDEG
      IF ((NDEG.AND.1).EQ.0) NE=NE+1
      VALRE=A(NE)
      DO 10 I=NE-2,1,-2
        VALRE=VALRE*OMEGA2+A(I)
   10 CONTINUE
C ... then the imaginary part
      NN=NDEG
      IF (NE.EQ.NDEG) NN=NN+1
      VALIM=A(NN)
      DO 20 I=NN-2,2,-2
        VALIM=VALIM*OMEGA2+A(I)
   20 CONTINUE
      VALIM=VALIM*OMEGA
      RETURN
      END
C --- evpols -----------------------------------------------------------*
C  This routine evaluates the polynomial in the real domain.
C
      SUBROUTINE EVPOLS (NDEG,OMEGA,A,VALRE,VALIM)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(10)
      VALRE=A(NDEG+1)
      VALIM=0.D0
      DO 10 I=NDEG,1,-1
        VAL=VALRE*OMEGA
        VALRE=-VALIM*OMEGA+A(I)
        VALIM=VAL
   10 CONTINUE
      RETURN
      END
C --- cmxmul -----------------------------------------------------------*
C  Complex multiply (A,Bj) = (A,Bj)*(C,Dj)
C
      SUBROUTINE CMXMUL (A,B,C,D)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      X=A*C-B*D
      B=A*D+B*C
      A=X
      RETURN
      END
```

```
/* C-language routine returns the execution (or processor) time ------- */
#include <sys/time.h>
#include <sys/times.h>
#include <sys/types.h>
#include <unistd.h>

double exectime_ ()
{
     double e;
     struct tms     buffer;
     static int     mark = 0;
     if (mark == 0) mark = sysconf(_SC_CLK_TCK);
     times(&buffer);
     e = ((float) buffer.tms_utime) / mark;
     return(e);
}
```

## APPENDIX 2

TESTREP compares execution times for evaluation of polynomials using several different represetations of polynomials.

```
      PROGRAM TESTREP
      PARAMETER(LR=50,NDEG=8)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION SYMB(15)
      DIMENSION KOD(LR)
      DATA SYMB / 0.D0,0.D0,13.D0,12.D0,11.D0,10.D0,9.D0,8.D0,7.D0,6.D0,
     +            5.D0,4.D0,3.D0,2.D0,1.D0 /
C ... set the number of repetitions
      NREP=1000000
      WRITE(*,100) NREP
  100 FORMAT(' test-rep : number of repetitions : ',I7)
C ... set the frequency
      OMEGA1=1.5D0
C ... print the header
      WRITE(*,150)
  150 FORMAT(/'  deg    tree     reverse   binary ')
C ... for consecutive degrees
      DO 80 N=2,NDEG
C ..... generate reverse representation
        CALL REVREP(N,KOD,LR)
C ..... create parse tree representation
        CALL mktree(N)
C ..... evaluate tree representation
        T0=exectime()
        OMEGA=OMEGA1
        DO 91 II=1,NREP
          OMEGA=OMEGA+1.D0
          CALL evtree(OMEGA,SYMB(1),POLRE(1),POLIM(1))
   91   CONTINUE
        T1=exectime()
        OMEGA=OMEGA1
        DO 92 II=1,NREP
          OMEGA=OMEGA+1.D0
          CALL EVALRR(OMEGA,KOD,SYMB,VALR1,VALI1)
   92   CONTINUE
        T2=exectime()
        OMEGA=OMEGA1
        DO 93 II=1,NREP
          OMEGA=OMEGA+1.D0
          CALL SPSEVB(N,OMEGA,SYMB,VALR2,VALI2)
   93   CONTINUE
        T3=exectime()
        WRITE(*,900) N,T1-T0,T2-T1,T3-T2
  900   FORMAT(I5,1P3D10.2)
   80 CONTINUE
      STOP
      END
C --- revrep -----------------------------------------------------------*
C  This routine generates the reverse representation in KOD; arguments
C  are represented by their subscripts+3, operators: * as -3, + as -1.
C  It is assumed that SYMB[1]=omega and SYMB[2]=-omega*omega.
C
```

```
      SUBROUTINE REVREP (N,KOD,LR)
      DIMENSION KOD(*)
C ... first generate code for the real part of the polynomial
      I=1
      NE=N
      IF ((NE.AND.1).NE.0) NE=NE-1
      KOD(I)=NE+3
      DO 20 J=NE-2,0,-2
        I=I+1
        KOD(I)=2
        I=I+1
        KOD(I)=-3
        I=I+1
        KOD(I)=J+3
        I=I+1
        KOD(I)=-1
   20 CONTINUE
C ... then generate code for the imaginary part
      NN=N
      IF (NE.EQ.N) NN=N-1
      I=I+1
      KOD(I)=NN+3
      DO 30 J=NN-2,1,-2
        I=I+1
        KOD(I)=2
        I=I+1
        KOD(I)=-3
        I=I+1
        KOD(I)=J+3
        I=I+1
        KOD(I)=-1
   30 CONTINUE
      I=I+1
      KOD(I)=1
      I=I+1
      KOD(I)=-3
C ... finaly append the terminator
      I=I+1
      KOD(I)=0
      IF (I.GT.LR) STOP 'insufficient length of KOD.-'
      LR=I
      RETURN
      END
C --- evalrr -----------------------------------------------------------*
C  This routine evaluates the reverse representation stored in KOD.
C  It sets SYMB[1]=omega and SYMB[2]=-omega*omega.
C
      SUBROUTINE EVALRR (OMEGA,KOD,SYMB,VALRE,VALIM)
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
      DIMENSION KOD(*),SYMB(*)
      DIMENSION ARGS(15)
      SYMB(1)=OMEGA
      SYMB(2)=-OMEGA*OMEGA
      ITOP=0
      IP=1
   10 K=KOD(IP)
      IP=IP+1
      IF (K.NE.0) THEN
        IF (K.GT.0) THEN
```

```
          ITOP=ITOP+1
          ARGS(ITOP)=SYMB(K)
        ELSE
          ITOP=ITOP-1
          GO TO (21,22,23,24),-K
          STOP 'SPSEVF : undefined operator.-'
   21     ARGS(ITOP)=ARGS(ITOP)+ARGS(ITOP+1)
          GO TO 30
   22     ARGS(ITOP)=ARGS(ITOP)-ARGS(ITOP+1)
          GO TO 30
   23     ARGS(ITOP)=ARGS(ITOP)*ARGS(ITOP+1)
          GO TO 30
   24     ARGS(ITOP)=ARGS(ITOP)/ARGS(ITOP+1)
   30     CONTINUE
        ENDIF
        GO TO 10
      ENDIF
      IF (ITOP.NE.2) WRITE(*,900) ITOP
  900 FORMAT(' ... SPSEVF :: incorrect argument stack :',I3)
      VALRE=ARGS(1)
      VALIM=ARGS(2)
      RETURN
      END
C --- spsevb --------------------------------------- standard --------*
C  This routine performs "binary code" evaluation.
C
      SUBROUTINE SPSEVB (N,OMEGA,SYMB,VALRE,VALIM)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION SYMB(*)
      OMEGA2=-OMEGA*OMEGA
      VALRE=0.D0
      VALIM=0.D0
      GO TO (11,12,13,14,15,16,17,18,19),N
   19 VALIM=((((SYMB(12)*OMEGA2+
     +      SYMB(10))*OMEGA2+
     +      SYMB(8))*OMEGA2+
     +      SYMB(6))*OMEGA2+
     +      SYMB(4))*OMEGA
   18 VALRE=(((SYMB(11)*OMEGA2+
     +      SYMB(9))*OMEGA2+
     +      SYMB(7))*OMEGA2+
     +      SYMB(5))*OMEGA2+
     +      SYMB(3)
      IF (N.EQ.9) RETURN
   17 VALIM=(((SYMB(10)*OMEGA2+
     +      SYMB(8))*OMEGA2+
     +      SYMB(6))*OMEGA2+
     +      SYMB(4))*OMEGA
      IF (N.EQ.8) RETURN
   16 VALRE=((SYMB(9)*OMEGA2+
     +      SYMB(7))*OMEGA2+
     +      SYMB(5))*OMEGA2+
     +      SYMB(3)
      IF (N.EQ.7) RETURN
   15 VALIM=((SYMB(8)*OMEGA2+
     +      SYMB(6))*OMEGA2+
     +      SYMB(4))*OMEGA
      IF (N.EQ.6) RETURN
   14 VALRE=(SYMB(7)*OMEGA2+
```

```
     +      SYMB(5))*OMEGA2+
     +      SYMB(3)
      IF (N.EQ.5) RETURN
   13 VALIM=(SYMB(6)*OMEGA2+
     +      SYMB(4))*OMEGA
      IF (N.EQ.4) RETURN
   12 VALRE=SYMB(5)*OMEGA2+
     +      SYMB(3)
      IF (N.EQ.3) RETURN
   11 VALIM=SYMB(4)*OMEGA
      IF (N.EQ.2) RETURN
      VALRE=SYMB(3)
      RETURN
      END
```

```
/* C-language routines for parse tree generation and evalaution --------- */
#include <stdio.h>
#include <ctype.h>

/* tree structure */

typedef struct tree {
     struct tree        * left;
     struct tree        * right;
     int                val;
} tree;

extern char * malloc (int);

tree      * TreeRe = NULL;
tree      * TreeIm = NULL;

tree      * FreeNode = NULL;
int       NumGetNode = 0;
int       NumMemNode = 0;
int       NumRelNode = 0;

tree * GetNode () /* ---------------------------------------------------- */
{
     tree * node = NULL;
     if (FreeNode != NULL)
     {
          node = FreeNode;
          FreeNode = node->left;
          NumGetNode++;
     }
     else
     {
          if ((node = (tree *) malloc(sizeof(tree))) == NULL)
          {
               fprintf(stdout," ... <malloc> : no available memory.-");
               exit(1);
          }
          NumMemNode++;
     }
     return (node);
}

void RelNode (tree * node) /* -------------------------------------------- */
{
     node->left = FreeNode;
     node->right = FreeNode;
     FreeNode = node;
     NumRelNode++;
}

void RelTree (tree * node) /*- ------------------------------------------- */
{
     if (node->left != NULL) RelTree(node->left);
     if (node->right != NULL) RelTree(node->right);
     RelNode(node);
}

tree * Node (int val) /* ------------------------------------------------ */
```

```
{
     tree * node = NULL;
     node = GetNode();
     node->left = NULL;
     node->right = NULL;
     node->val = val;
     return (node);
}

void mktree_ (int * deg) /* --------------------------------------------- */
{
     tree * p = NULL;
     int  i;
     for (i = 0; i <= *deg; i = i+2)
     {
          if (i < *deg-1)
          {
               if (i == 0)
               {
                    p = Node(-1);
                    TreeRe = p;
               }
               else
               {
                    p->left = Node(-1);
                    p = p->left;
               }
               p->right = Node(i+2);
               p->left = Node(-3);
               p = p->left;
               p->right = Node(1);
          }
          else p->left = Node(i+2);
     }
     TreeIm = Node(-3);
     p = TreeIm;
     p->right = Node(0);
     for (i = 1; i <= *deg; i = i+2)
     {
          if (i < *deg-1)
          {
               p->left = Node(-1);
               p = p->left;
               p->right = Node(i+2);
               p->left = Node(-3);
               p = p->left;
               p->right = Node(1);
          }
          else p->left = Node(i+2);
     }
}

double Eval (tree *root,double Val[]) /* -------------------------------- */
{
     int k;
     double a1;
     double a2;
     if ((k = root->val) >= 0) return (Val[k]);
     a1 = Eval(root->left,Val);
```

```
     a2 = Eval(root->right,Val);
     switch (k) {
     case -1 : return (a1 + a2);
     case -2 : return (a1 - a2);
     case -3 : return (a1 * a2);
     case -4 : return (a1 / a2);
     default : fprintf(stdout," ... undefined operator : %d\n",root->val);
     exit (-1);
     }
}

evtree_ (double * Freq,double Val[],double * Re,double * Im) /* --------- */
{
     double x;
     double y;
     Val[0] = *Freq;
     Val[1] = - Val[0] * Val[0];
     x = Eval(TreeRe,Val);
     y = Eval(TreeIm,Val);
     *Re = x;
     *Im = y;
}
```