Implementation of a piecewise linear simulator

Dielen, M.

Award date:
1988

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
AUTOMATIC SYSTEM DESIGN GROUP

IMPLEMENTATION OF A PIECEWISE LINEAR SIMULATOR

M. Dielen

Master thesis
reporting on graduation work
performed from 01-12-86 to 30-11-87
by order of prof. dr. ing. J.A.G. Jess
supervised by ir. M.T. van Stiphout
and dr. ir. J.T.J. van Eijndhoven

CONTENTS

Abstract

The design of a piecewise linear simulator for mixed level simulation of electronic circuits is a huge task. A lot of different problems have to be solved. Not only the linearisation of non linearities of circuit elements but also simulation time and data storage determine the usefulness of the simulator.

All the circuit elements are described by pwl models. The linear complementarity problem are solved by an algorithm proposed by van de Panne and the solution over time of the differential equations is calculated by an implicit integration rule chosen by the user. The algorithm of van de Panne shows fast global convergence properties and the implemented integration rules have good accuracy and stability properties.

The basic problem of the simulator is finding for each time point a set of linear equations which represent the behaviour of the simulated circuit. The system matrix changes continuously during integration, the solution of the linearised system matrix has to be fast and efficient. On the system matrix an *LU* decomposition is performed which is solved by forward-backward substitution. The system matrix is stored sparse, to cope with circuits which grow larger and larger. Beside of that the circuit is stored in a hierarchical way and this storage is used to form a bordered block sparse matrix structure. Both the *LU* decomposition and the forward-backward substitution make use of the sparse matrix structure of the system matrix, the hierarchical storage of the circuit and the bordered block matrix structure.

The simulator is event driven, meaning that during simulation every cell is assigned its own time step. Every step of the simulation the leafcell with the nearest event is handled. With this approach only the circuit elements that change will be visited during simulation and computation time is automatically reduced.

The input nodes are handled in a different way. These nodes are not represented by pwl models but the user can assign mathematical functions to an input node and the simulator will compute the signals according to these functions. Because of the general framework of the simulator, only minor changes in sub-functions had to be made and the generality of the framework was not affected. With this tool it is possible for the user to assign values to a node in the task file without changing the circuit structure.

The interfaces to the user are fast and plain to understand. With the aid of a schematic capture program the user is able to build circuits in a hierarchical way, a structure which is maintained by the simulator and used to speed up simulation. The signal post processor visualises clearly the signals of the requested circuit elements and simple operations like e.g. addition, subtraction and comparison are possible.

At the moment the simulator is in a preliminary state, a lot of tests are performed during simulation, in spite of this the results until now are promising. The test circuits performed by the simulator show good results. Still some improvements have to be made. The data storage of the system matrix can be reduced by leaving out the system matrix and only storing the *LU* decomposition. The scaling of the input-output vector and implementation of a pivot strategy. The step size control of the integration rule can be elaborated. Research has to be carried out after integration rules which show better stability results for application of variable step sizes.

1. Introduction

At the automatic system design group the design of a piecewise linear (pwl) simulator is realised. This program is able to simulate large to very large electronic circuits. The circuits can be simulated on mixed levels, this means that simulation can be carried out for example at transistor level, digital level or even at a higher level and also a mixture of different levels is possible.

Traditionally, circuit simulation programs are based on the Newton Raphson scheme. Non linear equations are created and during simulation transformed to linear equations. A drawback of this method is that a solution not always is obtained because of the stringent convergence properties. A new promising technique makes use of pwl models for the representation of circuit elements. The pwl simulator gets a pwl description of a non linear element and solves a set of linear equations. The solution of the pwl description, the so called linear complementarity problem, is based on an algorithm developed by van de Panne, which has very good global convergency properties.

For the solution over time, several numerical integration rules have been implemented, to cope with the differential equations of the pwl model. The trapezium rule and the backward euler method can be used. But also more sophisticated methods like the two step backward differentiation method or a two step A-contractive method may be applied, which both give good stability and accuracy results.

For several years different people have been working on the theoretical and practical applications of the pwl simulator. A first implementation was finished for 3 years by van Eijndhoven [9]. Mainly due to the use of this program a better insight was gained in the advantages and disadvantages of the use of pwl models. Through this experience novel ideas put up to the building of a new simulator.

The simulator is implemented in C on the HP9000-500 system and is ported to the Apollo DN3000. The frame of the simulator had already been build when I entered the project. The main parts I worked at, together with M. van Stiphout, are the DC solution, the transient analysis and the implementation of the input nodes. The block structure of the simulator is as follows :

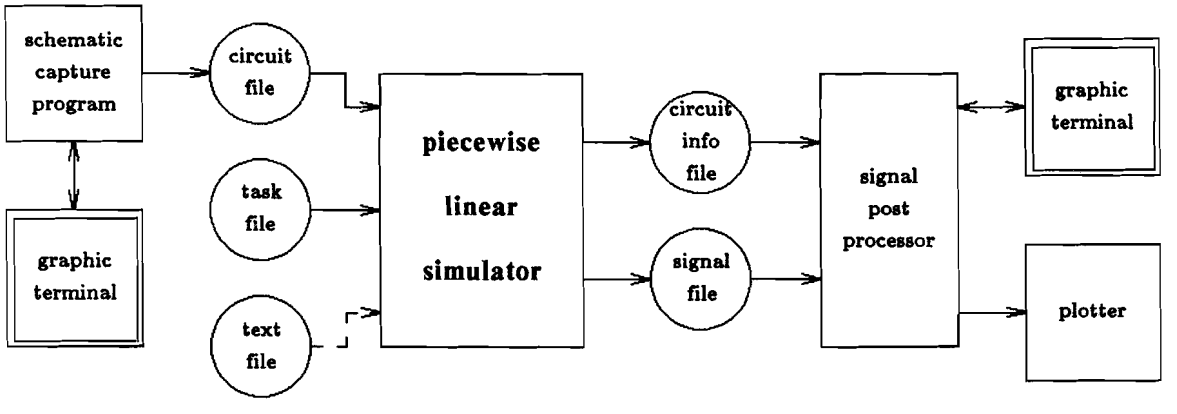


Figure 1.1. The block structure of the simulator

With the aid of the schematic capture program "ESCHER" a circuit is made, the circuit information is stored in the circuit file. The task file denotes which simulation has to be carried out, e.g. which circuit, which input signals etc... Another possibility is to enter

the circuit structure in a text file, this feature is still under development. The task file and the circuit- or text file are input to the simulator. The simulator outputs a file with general circuit information and a file with signal values. With the aid of a signal post processor these signals can be visualised on a graphics device or a plotter.

2. Piecewise linear models

2.1 The implicit notation

The simulator uses the implicit notation for the pwl models, as described by van Bokhoven [3] and van Eijndhoven [9]. This means that the inputs and outputs are joined together in one vector, see equation 2.1.

$$\begin{bmatrix} 0 \\ \dot{u} \\ v \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ u \\ i \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad (2.1)$$

x : *vector of inputs and outputs.*

$\frac{\partial u}{\partial t} = \dot{u}$

$v, i \geq 0$ and $v.i = 0$

The matrix A_{11} denotes the equation that represents the main behaviour of the pwl model. The square matrix A_{22} denotes the relation between u and $\dot{u}$. Via matrices A_{12} and A_{21} this matrix influences A_{11} during integration. The square matrix A_{33} defines several polytopes and divides the multidimensional space in subspaces. Generaly with every subspace has been associated a different A_{11}. The matrix A_{33} influences A_{11} via matrices A_{13} and A_{31}. If the influences on A_{11} are accounted for, the resulting linear representation found for the input-output vector is called the jacobian. From the matrix description it is clear that four different models are possible, with or without dynamic field and with or without polytope field. For convenience four examples are given :

1) resistor

$$0 = -V + R \cdot I$$

2) capacitor

$$\begin{aligned} 0 &= -V + u + V_{\text{init}} \\ \dot{u} &= \frac{1}{C} \cdot I \end{aligned}$$

3) inverter

$$\begin{aligned} 0 &= -out - i_1 + i_2 + 1 \\ v_1 &= -in + i_1 \\ v_2 &= -in + i_2 + 1 \end{aligned}$$

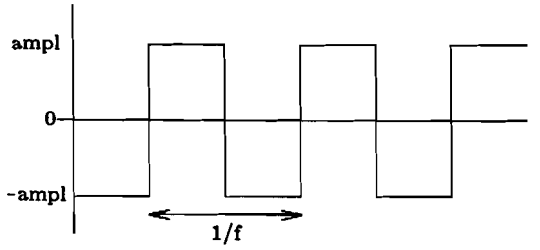
4) delay

$$\begin{aligned} 0 &= -out - i_2 + 1 \\ \dot{u} &= \frac{1}{delay} \cdot in + \frac{1}{delay} \cdot i_2 - \frac{1}{delay} \\ v_1 &= - i_2 + 1 \\ v_2 &= + u + i_1 - i_2 + 1 \end{aligned}$$

To avoid ambiguous results of the simulator, the description of pwl models has to satisfy rules. With the aid of a lexical analyser and parser generator grammar rules are composed. With the lexical analyser tokens are generated which the parser uses for syntax analysis. The parser provides error messages in case the user does not use the proper syntax. The syntax diagrams of the pwl model are depicted in appendix 1.

As an example the pwl model of a block generator is shown :

```
leafcell block_gen{erator}({output} out : signal);
   <<    frequency   : 0..* default 1 {Hz};
         amplitude   : default 1;
         dc_level    : default 0;
   >>;
(* symmetric block generator, 50 % duty cycle *)
{$L-}
(* starts (t = 0) at dc_level-amplitude
```




```
*)
begin
        var   (out,   u.1,          pl.1,   pl.2,          1   );
   zero.1 =    -1,     ,   2*amplitude,     ,   dc_level-amplitude  ;
     du.1 =     ,      ,  -4*frequency,     ,        2*frequency    ;
     pl.1 =     ,    -1,           -1,     1,                  1    ;
     pl.2 =     ,     ,            -1,     ,                  1    ;
end; (*block_gen*)
{$L+}
```

2.2 Static models

A model is called static, if the pwl variables, u and $\dot{u}$, are not present. These models don't have to be solved by the numerical integration rule. During transient analysis it must be checked whether by a change of the input variables the pwl model is still valid. if the pwl model is not any more valid the van de Panne algorithm has to be initiated.

A static model is denoted by the following matrix :

$$\begin{bmatrix} 0 \\ v \end{bmatrix} = \begin{bmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ i \end{bmatrix} + \begin{bmatrix} b_1 \\ b_3 \end{bmatrix} \qquad (2.2)$$

The jacobian is said to be valid, in case the pwl model remains in the subspace of which the following constraints are satisfied :

$$v \geq 0 \qquad (2.3)$$
$$i \geq 0$$
$$v.i = 0$$

Do we look at the example of the inverter we see that in case in=0 and $i_1 = i_2 = 0$, the model is valid :

$$0 = -out + 1$$
$$v_1 = 0$$
$$v_2 = 1$$

Does the input rise from zero to one, then v_1 turns negative :

$$0 = -out + 1$$
$$v_1 = -1$$
$$v_2 = 0$$

For reasons that will be explained in chapter 4, a pivot has to be performed in row 1 and column 1 of A_{33}. After the pivot the model gets the following form :

$$0 = -out - in + i_1 + 1$$
$$v_1 = \quad in + i_1$$
$$v_2 = \quad -in \quad + i_2 + 1$$

Now the model is valid, because v_1 is positive again :

$$0 = out$$
$$v_1 = 1$$
$$v_2 = 0$$

The curve for the relationship between the input and output is depicted in figure 2.1.

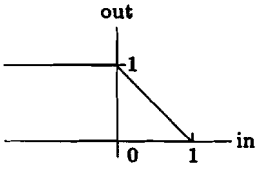


Figure 2.1. Input-output curve of the inverter

The above example of the inverter is very small, only four sub-spaces are possible. For a general $n \cdot n$ A_{33} matrix, 2^n different sub-spaces are possible. Although some sub-spaces can be empty, it is clear that finding a solution, by searching many of these sub-spaces, is not efficient. A lot of different algorithms have been designed to handle this problem, which are shown in van Eijndhoven [9]. The most promising and fitted algorithm to the problem of circuit simulation was found to be an algorithm designed by van de Panne and will be presented in chapter 4.

2.3 Dynamic models

A linear dynamic model is denoted by a pwl model without v and i variables. These models are never treated by the van de Panne algorithm. During transient analysis, a

proper jacobian is computed with the aid of the integration rule and the pwl variables are subsequently solved.

A linear dynamic model is represented by the following matrix :

$$\begin{bmatrix} 0 \\ \dot{u} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} x \\ u \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{2.4}$$

The integration over time takes care for the solution of the relation between u and $\dot{u}$. For the solution of the differential equations an integration rule is applied. The integration rule can be stated as follows :

$$f(u, \dot{u}, t) = 0 \tag{2.5}$$

With the aid of the integration rule the relation between $\dot{u}$ and u of the pwl model can be solved. Eliminating $\dot{u}$ from :

$$\dot{u} = A_{21} \cdot x + A_{22} \cdot u \tag{2.6}$$

yields a set of equations which have x and u as unknown, by transforming these equation in such a way that u remains on the left hand side, we can substitute them in

$$0 = A_{11} \cdot x + A_{12} \cdot u + b_1 \tag{2.7}$$

and a tableau remains which has only x as unknown, by substituting the newly derived jacobian in the system matrix, x can be solved.

3. Global overview of the simulator

3.1 The datastructure

With increasing complexity of IC technology and mixed analog/digital systems designers ask for the analysis of larger and larger circuits. An important factor is the storage of the circuit, on the first place to gain speed, but also to reduce algorithms complexity. The datastructure of the simulator can roughly be divided in two parts. On one hand the datastructure of the circuit and on the other the datastructure of the system matrix. First the circuit structure will be dealt with and after that the matrix will be treated.

3.1.1 The circuit

The storage of the circuit is hierarchically arranged in a tree. The circuit is divided in modules, which form the nodes of the tree. A module can be defined as follows : a module has exactly one father and one or more children. There are only two exceptions to this rule, the first one is the root of the tree, which has no father, the second one are the leafs of the tree, which have no children. The circuit has two different leafs, called leafcells and inputcells. A leafcell is defined by a pwl description and an inputcell is defined by a mathematical function. An example of a tree is stated in figure 3.1.

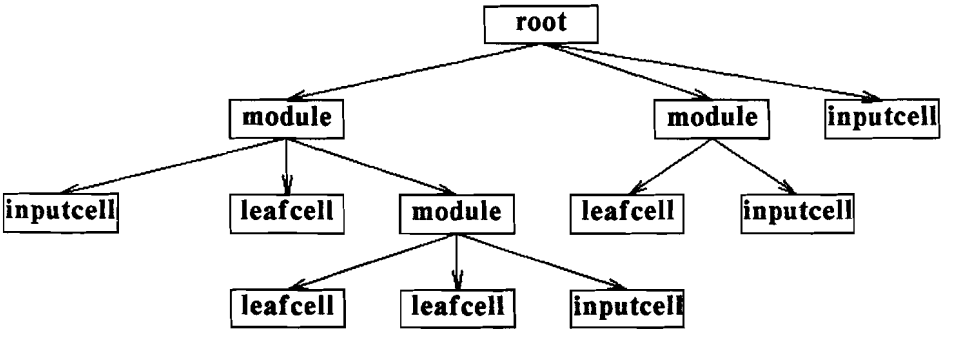


Figure 3.1. The hierarchically storage of the circuit

It is very easy to obtain this circuit structure because the schematic capture program supports this hierarchically building of the circuit, as well as the network description language. The datastructure of a module is depicted in figure 3.2. The module structure contains fields which give information about its identification, relation with the system matrix and timing information. Several pointers : a pointer to a list which contains the module definition, a pointer to a list which contains the terminals of the module and three pointers which point to the father module, to a list of brothers and to a list of children. Concerning exceptions, the root module structure is exactly the same, the leafcell- and inputcell structures are basically the same however they have an extension which contain fields concerning the pwl matrix or the input function.

The datastructure of the inputcell will be treated in chapter 6. The leafcell structure is denoted in figure 3.3. The first part is equal to the module structure. For convenience the variables will be explained here. Some of them may not be clear at this moment, they will be dealt with in the subsequent sections.

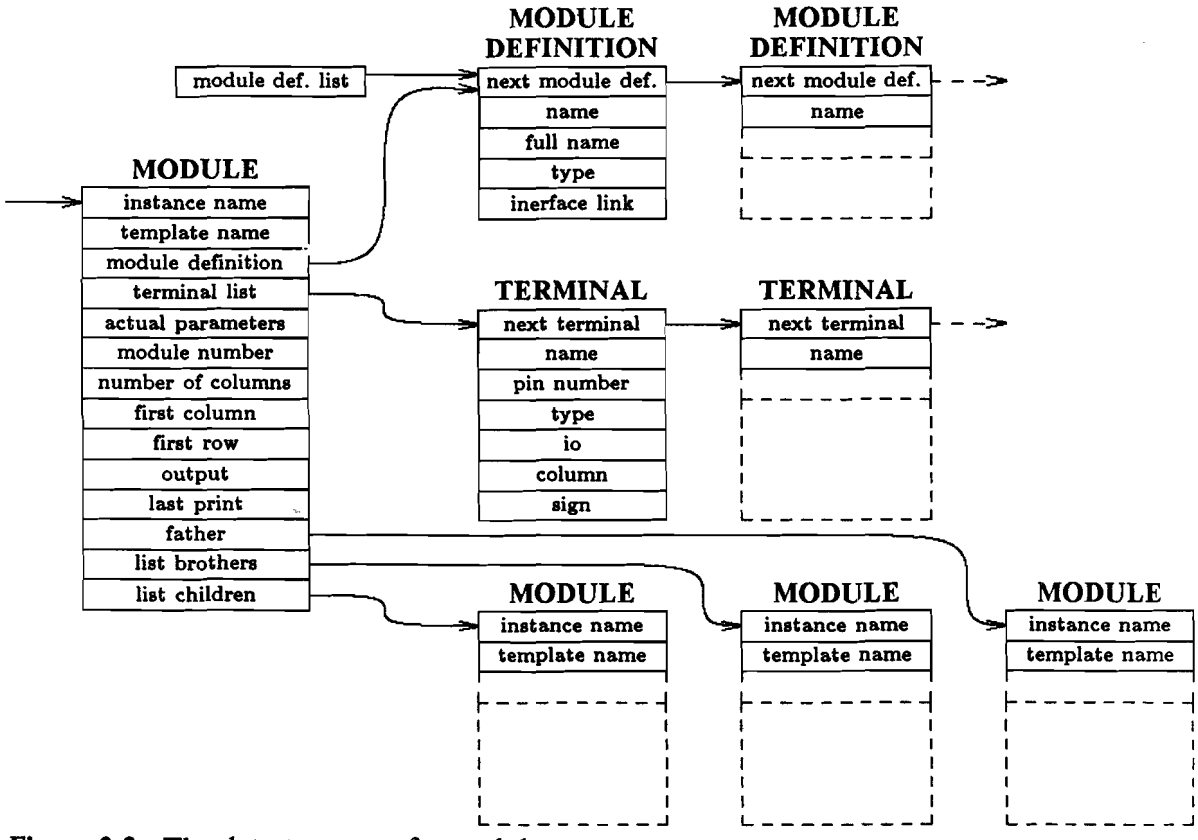


Figure 3.2. The datastructure of a module

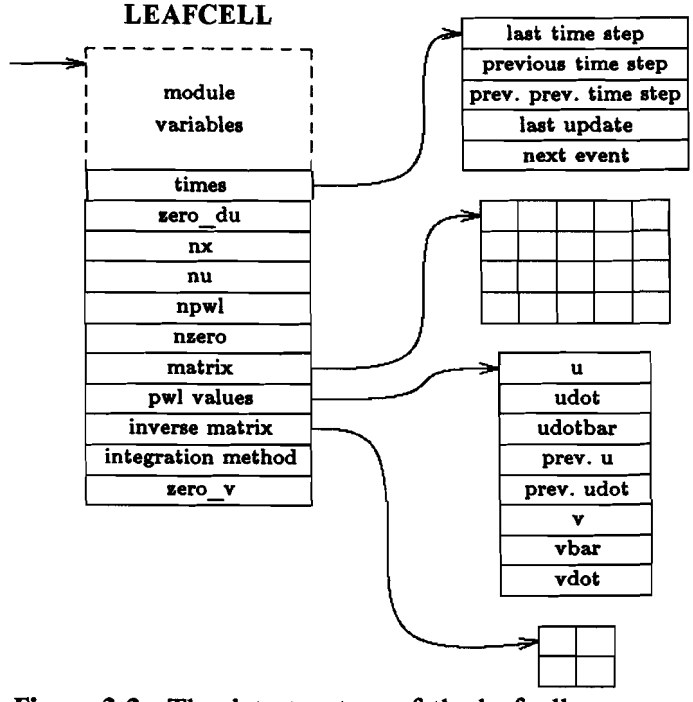


Figure 3.3. The datastructure of the leafcell

time :	array containing timing information, last time step previous time step previous previous time step last update next event
zero_du :	binary variable, denoting which elements the user wants to set to zero during the dc solution.
nx :	pwl matrix dimension, number of x elements
nu :	pwl matrix dimension, number of u elements
npwl :	pwl matrix dimension, number of v elements
nzero :	pwl matrix dimension, number of rows of A_{11}
matrix :	pwl matrix
pwl values :	matrix containing the pwl values, u_n, $\dot{u}_n$, $\bar{u}_n$, $\bar{\dot{u}}_n$, u_{n-1}, $\dot{u}_{n-1}$, v_n, $\bar{v}_n$, $\dot{v}_n$
inverse matrix :	$(a_0 \cdot I - b_0 \cdot h_n \cdot A_{22})^{-1}$
integration method :	contains the current integration method, forward euler backward euler trapezium rule two steps backward differentiation method two steps A-contractive method
zero_v :	denoting whether a panne event or a dynamic event is detected and if a panne event occurs it contains the row of the v element that turned zero.

3.1.2 The system matrix

Basically simulation is the successive solution of the equation $A \cdot x = b$ in which A represents the system matrix, x the concatenated input-output vector and b the source vector. The system matrix can become quite large for reasonably sized circuits. Fortunately, in practice, a lot of entries in the matrix appear to be zero, therefor the system matrix is stored sparse. This means that zero entries are not stored. The system matrix is in essence a double matrix, because as well the matrix A as the LU decomposition of A are stored in the same data structure, see figure 3.4. The storage of A itself is only in use now for debugging purpose, in the future only the LU decomposition will be stored and the elements of the system matrix will be indirectly derived via the pwl models. The two vectors situated along the matrix, which point to the first element of a row or column, are stored in a list during the build-up. After completion these lists are converted to arrays, so direct access is obtained to the first elements in a row or column. Also an array is constructed that contains the diagonal elements of the matrix, these elements are involved in a lot of operations during simulation, so direct access is of primary importance. An example of a sparse system matrix is shown in figure 3.5.

3.2 The program

The program can roughly be divided in the following tasks :

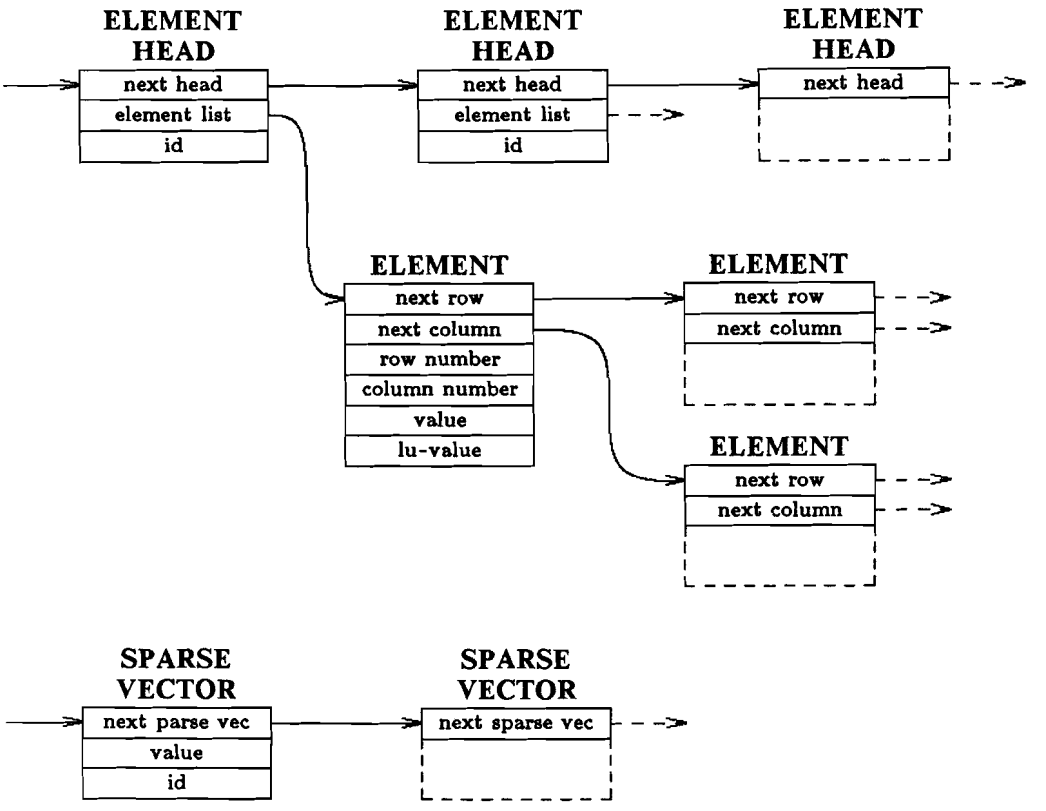


Figure 3.4. The datastructure of the sparse system matrix

read simulation task description;
build circuit equations;
match circuit equations;
LU decomposition;
dc solution;
transient analysis;

The most consuming task is the transient analysis. Nevertheless the other tasks are important for a good initialisation of the simulation. The first four functions will be treated in the next sections, the last two functions i.e. the dc solution and the transient analysis are dealt with in chapter 4 and 5 respectively.

3.2.1 The task file

The task file describes the simulation task that has to be performed, giving the circuit name, integration interval, accuracies, etc. With the aid of a lexical analyser and a parser generator, grammar rules have been constructed, to provide a good interface with the user. The syntax diagrams of the task file are depicted in appendix 2.

Below an example of a task file is given :

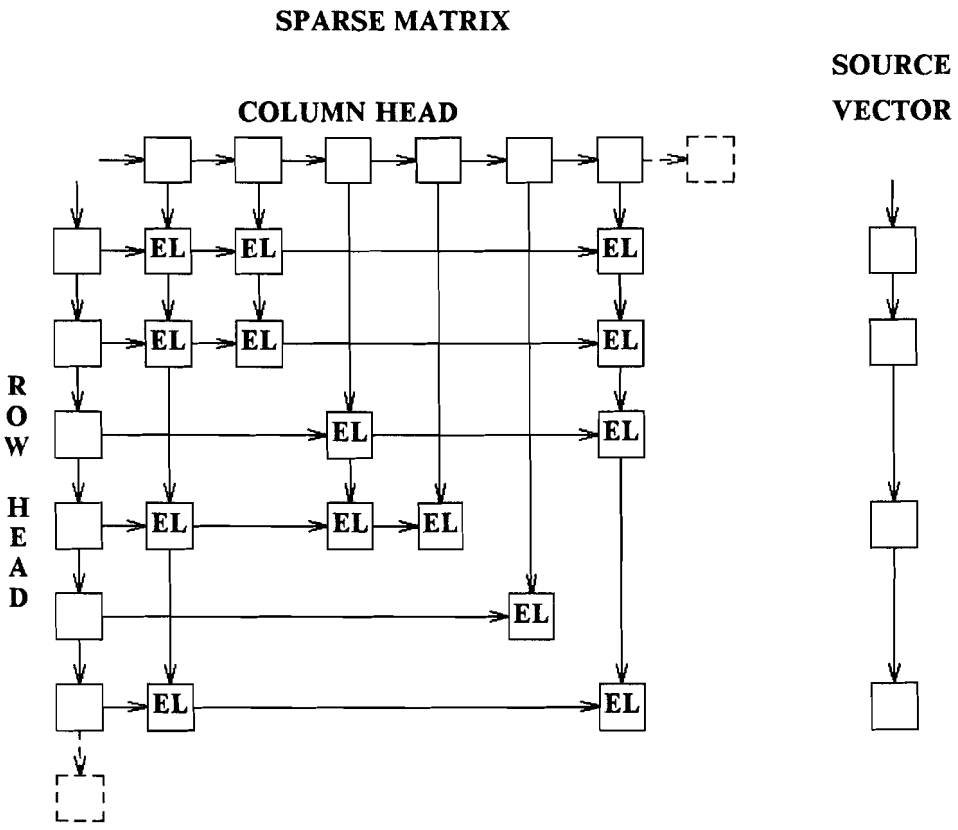


Figure 3.5. The datastructure of the sparse system matrix

```
MODULE example;
INPUT /in = PULSE( 1.0, 0.0, 0.0, 1.0, 0.0, 10.0, 0.0)[V];
REFERENCE /ref;
OUTPUT /submodule1;
TRANSIENT 0.0[S], 1.5[S], 0.001[S];
ACCURACY INTEGRATION ( 0.1e-3, 0.5e-1 );
INTEGRATION_METHOD ACT;
```

3.2.2 The construction of the circuit equations

The hierarchical structure of the circuit is also maintained in the system matrix, as described by Vlach [22]. Each subcircuit contributes to the system matrix a bordered block diagonal matrix. In the sparse system matrix this occurs as a bordered block structure, see figure 3.6.

The internal nodes of the subcircuit correspondent to the blocks on the diagonal and the external nodes correspondent to the bordered oblong matrices. The structure of the matrix results directly from the hierarchical definition of the system.

The storage of the system matrix as a bordered block diagonal matrix yields nice features for the *LU* decomposition and forward-backward substitution which will be treated in section 3.2.4.

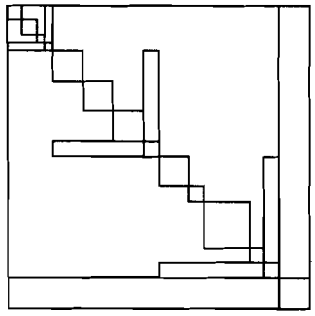


Figure 3.6. The bordered block structure of the system matrix

3.2.3 Matching of the circuit equations

At the moment a suitable pivot order is determined by applying a straight-forward bipartite matching algorithm. In the future this method is to be replaced by a weighted matching, taking in account sparsity and numerical considerations.

3.2.4 LU decomposition

In the simulator an LU decomposition is implemented, because often only the source vector b changes while the system matrix A, and thus the LU decomposition remains the same. On the contrary to the Gaussian elimination method which has to be repleated on the entire system matrix even if only the source vector changes. For the full LU decomposition the modified Gaussian elimination method is used. This method transforms the system matrix in two triangular matrices, a lower (L) and an upper (U) matrix, to yield the following equation :

$$A \cdot x = b \quad \rightarrow \quad L \cdot U \cdot x = b \tag{3.1}$$

This system is solved by the so called backward forward substitution :

$$\begin{aligned} L \cdot y &= b \\ U \cdot x &= y \end{aligned} \tag{3.2}$$

It is noticed that the transformation of the system matrix into the triangular form is underdetermined, n elements can be chosen free. We chose to set the diagonal elements of L to 1.

the system matrix is taken to be :

$$\begin{aligned} &a_{11} \cdot x_1 + a_{12} \cdot x_2 \cdots a_{1n} \cdot x_n = b_1 \\ &a_{21} \cdot x_1 + a_{22} \cdot x_2 \cdots a_{2n} \cdot x_n = b_2 \\ &\cdot \qquad \cdot \qquad \cdot \\ &\cdot \qquad \cdot \qquad \cdot \\ &a_{n1} \cdot x_1 + a_{n2} \cdot x_2 \cdots a_{nn} \cdot x_n = b_n \end{aligned}$$

Starting the transformation of the system matrix into the LU decomposition, the first row of U is equal to the first row of A and the first column of the lower triangular matrix is formed by $L_{i1} = A_{i1} \; / \; U_{11}$. Furthermore the first row of A multiplied with L_{i1} has to be

subtracted from all the other rows. After this step the matrix will have changed to :

$$
\begin{aligned}
&u_{11}\cdot x_1 + u_{12}\cdot x_2 \cdots u_{1n}\cdot x_n = c_1 \\
&l_{21}\cdot x_1 + a'_{22}\cdot x_2 \cdots a'_{2n}\cdot x_n = b'_2 \\
&\quad\cdot \qquad\quad \cdot \qquad\quad\ \cdot \\
&\quad\cdot \qquad\quad \cdot \qquad\quad\ \cdot \\
&l_{n1}\cdot x_1 + a'_{n2}\cdot x_2 \cdots a'_{nn}\cdot x_n = b'_n
\end{aligned}
$$

By repetition of this transformation, the *LU* decomposition will be finished after n - 1 steps. With this last addition the algorithm to perform an *LU* decomposition on an $n \cdot n$ matrix becomes:

```
for ( j = 1; j <= n; j++ ) {
      if ( |u_jj| ==small )
            find new u_jj;
      for ( i = j+1; i <= n; i++ ) {
            l_ij = l_ij / u_jj;
            for ( k = j+1; k <= n; k++ )
                  a_ij -= l_ij · u_jk;
      }
}
```

This full *LU* decomposition according to Gauss is only performed once, to get an initial *LU* decomposition for the simulator. Although the Gaussian elimination method is fast and straightforward the order of the algorithm is $o(n^3)$, for full matrices of size n.

For maintaining the *LU* decomposition other techniques are used. During simulation the system matrix is changed to follow non linearities in time or circuit elements, these changes are small and it is not necessary to recompute the whole *LU* decomposition. The *LU* decomposition based on the method of Crout, differs only slightly from the Gauss method. The algorithm is based on the following equation :

$$
\begin{aligned}
&k = 1,\ 2, \ldots,\ n \qquad\qquad (3.3)\\
&u_{kj} = a_{kj} - \sum_{p=1}^{k-1} l_{kp}\cdot u_{pj} \qquad j = k,\ k+1, \ldots,\ n \\
&l_{ik} = \left(a_{ik} - \sum_{p=1}^{k-1} l_{ip}\cdot u_{pk} \right) / u_{kk} \qquad i = k+1, \ldots,\ n
\end{aligned}
$$

Although the order is the same, Gauss uses the right lower corner of the system matrix to store information about the *LU* decomposition, this information has to be build up from the start of the algorithm. The method of Crout makes use of the information stored in the *L* and *U* matrices, in the right upper and the left lower corner of the matrix, that's why it is not necessary to perform a full *LU* decomposition. In case an *LU* decomposition is present, the *LU* decomposition can start at the diagonal with index equal to the minimum column or row of the new inserted elements. An illustrative picture that well illustrates the differences of Gauss's and Crout's method can be found in Ruehli [18] and is depicted in figure 3.7. The property of Crout's method can be used for updating the *LU*-decomposition from a leafcell along a path to the root in the hierarchy of the circuit. This updating can terminate before reaching the root, e.g. if a change of a leafcell only affects a small part of the total circuit. In this way it is possible to decrease the computation time for finding a new *LU* decomposition.

The difficulty behind the algorithm is the avoidance of zero pivots. Encountering a zero

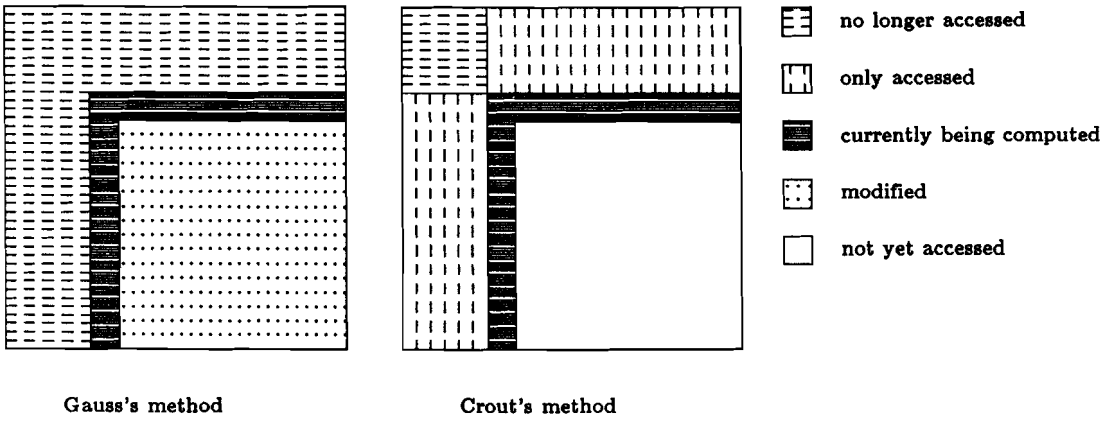


Figure 3.7. The *LU* decomposition according to Gauss and Crout

pivot implies the exchange of the currently computed row with another row, which has a nonzero entry in the column of the pivot. This exchanging of rows in a sparse matrix structure is time consuming.

An important factor for numerical stability during the *LU* decomposition is the so called pivot strategy. It is shown by Wilkinson [23], [24] that growth in element sizes has to be avoided. This implies that pivots have to be chosen as big as possible. This pivot strategy has not yet been implemented.

A second important factor is maintaining the sparsity, as the computation time is directly related with the number of nonzero elements in the system matrix. In Sangiovanni [20] it is pointed out that minimization of the number of nonzero elements created in A during the *LU* decomposition, is a criterion for the selection of a permutation of a row or column. However the problem of finding a permutation which minimizes the number of fill-ins, the so called minimum fill-in problem belongs to the class of *NP*-hard problems. An heuristic algorithm is proposed by Markowitz which is used by *SPICE*. The criteria for pivoting, maintaining accuracy and sparsity, may conflict. A convenient way avoiding this is using diagonal pivots only.

Another possibility to improve accuracy is the scaling of the x vector (the vector of inputs and outputs). The elements of the x vector can be scaled differently , e.g. scaling factors of $10^{-3} or 10^{-6}$, this is done to obtain a system matrix with equal sized elements. Using this technique it is avoided that during the pivoting process some elements grow excessively large.

An other method for maintaining the *LU* decomposition is the rank 1 update, which is described in Bennet [1] and in Eijndhoven [11]. Suppose the system matrix A has been decomposed in the matrices $L \cdot D \cdot U$, where L is a lower triangular matrix, U an upper triangular matrix and D a diagonal matrix. If A has to be modified by the addition of $X \cdot Y^t$, where X and Y are vectors of length n, then a method is proposed that uses the old *LU* decompostion to obtain the modified *LU* decompostion of $A + X \cdot Y^t$, the order of the method is equal to $o(2 \cdot n^2)$.

The rank 1 update is based on the partitioning of the system matrix into :

$$A = \begin{bmatrix} a_{11} & F \\ G & H \end{bmatrix} = L^1 \cdot D \cdot U^1 \tag{3.4}$$

$$where : L^1 = \begin{bmatrix} 1 & 0 \\ G/a_{11} & I_{n-1} \end{bmatrix} \qquad U^1 = \begin{bmatrix} 1 & F/a_{11} \\ 0 & I_{n-1} \end{bmatrix}$$

$$then : D = L^{1I} \cdot A \cdot U^{1I} = \begin{bmatrix} a_{11} & 0 \\ 0 & H - 1/a_{11} \cdot G \cdot F \end{bmatrix} \tag{3.5}$$

Do we refer to A as A^1 and take for A^2 :

$$A^2 = H - 1/a_{11} \cdot G \cdot F \tag{3.6}$$

Then the above process repeated on A^2, yields the next column of L, the next row of U and a_{22} of D. Further iteration complete the process.

Suppose that we have to deal with the modified matrix $A + X \cdot Y^t$. The first column, row and element of U, L and D respectively, are directly derived. Do we write :

$$B^1 = A^1 + X^1 . Y^{1t} \tag{3.7}$$

then the abovementioned process can be applied on B^1 to obtain B^2 and B^2 can be expressed as $A^2 + X^2 \cdot Y^{2t}$. Further repetition completes the process.

Below the algorithm of the rank 1 update is shown, where r_k, c_k, q_k, D_{old}, D_n, q_n and r_n are variables which help to compute the new LU decomposition economically.

```
d_k = 1
for ( i = 1; i < n+1; i++ ) {
      r_k = Y_i; c_k = X_i;
      p_k - c_k*d_k; q_k = r_k*d_k;
      D_old = A_ii;
      A_ii += p_k*r_k;
      q_n = q_k/A_ii; D_n = A_ii/D_old; r_n = r_k/D_old;
      d_k -=q_n*p_k;
      for ( j = i; j < n+1; j++ ) {
            Y_j -= r_n*U_ij;
            U_ij = U_ij*D_n + p_k*Y_j;
            if ( j = i ) continue;
            X_j -= c_k*L_ji;
            L_ji += q_n*X_j;
      }
}
```

4. DC-solution

4.1 The linear complementarity problem

The DC-solution is the calculation of the outputs, given the input, for the time point zero. So a valid solution is searched for :

$$A \cdot x = b \tag{4.1}$$

At the beginning all the pwl models are in their initial state, e.g. the output of an inverter is initially high. Suppose however that the input of the inverter at time zero is high, then the pwl model has to be transformed and therefor the system matrix changes. So finding a DC-solution narrows down to moving the leafcells in the right spaces, which are defined by A_{31}, A_{32}, A_{33} and b_3. This implies the manipulation of the pwl models, such that given the input values, the matrices A_{33} have to be changed in such a way that all the v vectors become positive. This problem is called the linear complementarity problem (*LCP*) and is well known in mathematics. It is stated as follows :

$$v = M \cdot i + b \tag{4.2}$$

$$v \cdot i = 0$$
$$v \, , \, i \geq 0$$

v and i are vectors of length n, b is a given source vector and M is an $n \cdot n$ matrix. Although this basically is our problem, the implementation in the simulator is different :

A. The matrix M is not present in the simulator, this is done to save storage. A very huge matrix would be derived, because the matrices A_{31} and A_{13} in the pwl models influence the values of x and v, they should also be stored:

$$\begin{bmatrix} 0 \\ v \end{bmatrix} = \begin{bmatrix} A & A^*_{13} \\ A^*_{31} & A^*_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ i \end{bmatrix} + \begin{bmatrix} b_1 \\ b_3 \end{bmatrix} \tag{4.3}$$

Now the matrix M is derived by :

$$M = A^*_{33} - A^*_{31} \cdot A^{I} \cdot A^*_{13} \tag{4.4}$$

In the simulator only the system matrix A is stored and the elements of the matrices M are indirectly derived via the pwl models.

B. During the solution of the problem, the total pwl model has to be updated. Beside of that the jacobian of the model can change, this implies a change in the system matrix and an update for the LU decomposition has to be performed.

For the solution of the *LCP*, an algorithm is chosen to implement which uses pivots only. This algorithm is based on a method developed by van de Panne [17], who has based his method on an algorithm developed by Lemke [13]. Van de Panne's algorithm is more powerful because only diagonal block pivots are performed and an infeasibility test is possible during the run of the algorithm, however at the cost of a more complex code.

4.2 The van de Panne algorithm

4.2.1 Determining the basic solution

For the case that b is positive one possible solution of the *LCP* is given as :

$$\begin{aligned} v &= b \\ i &= 0 \end{aligned} \tag{4.5}$$

The elements of v are called basic variables, elements of i non-basic. In case of one or more negative elements in b, the problem is extended with a vector a multiplied by an extra non-basic variable lambda. These two disturb the originally stated problem in such a way that v becomes positive. The problem becomes now :

$$v = M \cdot i + b + a \cdot \lambda \tag{4.6}$$

$$\begin{aligned} v \cdot i &= 0 \qquad \lambda \geq 0 \\ v,\ i &\geq 0 \qquad a \geq 0 \end{aligned}$$

By assigning elements of a the value 1 at places that correspond with negative elements of v, the other elements remain zero, and taking lambda high enough, a solution of the disturbed problem is :

$$\begin{aligned} v &= b + a \cdot \lambda \\ i &= 0 \end{aligned} \tag{4.7}$$

The goal of the solution process is moving lambda down to zero, to obtain the originally stated problem.

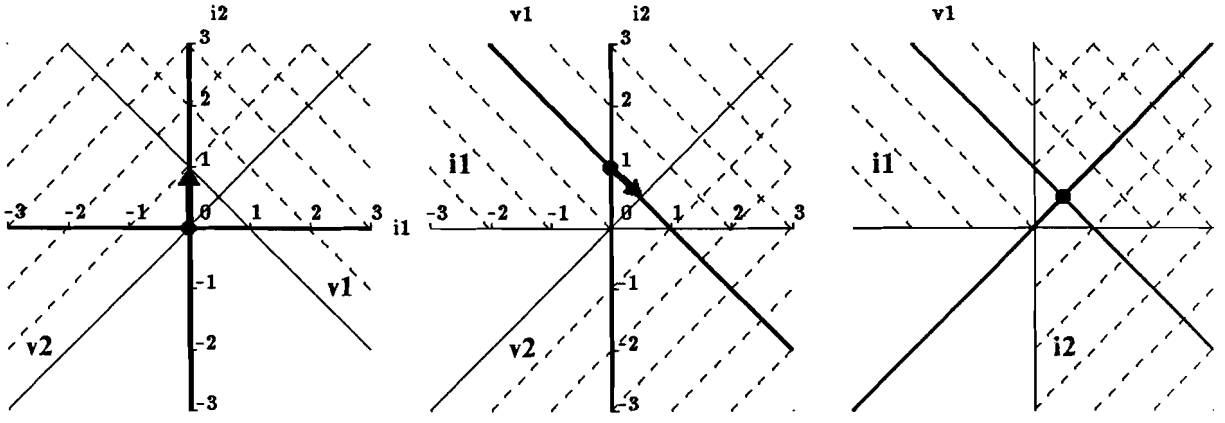
4.2.2 The pivot process

Before drawing attention to the solution process of the *LCP*, first the pivoting process, as the van de Panne algorithm is based on a continues pivoting, will be dealt with. Note that the pivots mentioned here are basically different from a pivot in an *LU* decomposition. The purpose of performing a pivot is exchanging an element of the v vector with an element of the i vector. In other words pivoting is a basis transformation in which a basis vector (i_j) is replaced by a vector (v_k) pointing in the direction of a polytope. This exchanging of vector elements is in fact a motion with the origin through the n-dimensional space. This space is divided in subspaces by the n polytopes defined by the matrix M.

The transformation of the matrix through a pivot is straightforward. Suppose we want to exchange v_k and i_l, pivoting on m_{kl} yields the following equations :

$$\begin{aligned}
m'_{kl} &= 1/m_{kl} && \\
b'_k &= -b_k/m_{kl} && \\
m'_{kj} &= -m_{kj}/m_{kl} && j=1,2,...,n \quad j\neq l \\
m'_{il} &= m_{il}/m_{kl} && i=1,2,...,n \quad i\neq k \\
b'_i &= b_i - b_k \cdot m_{il}/m_{kl} && i=1,2,...,n \quad i\neq k \\
m'_{ij} &= m_{ij} - m_{kj} \cdot m_{il}/m_{kl} && i=1,2,...,n \quad i\neq k \\
& && j=1,2,...,n \quad j\neq l
\end{aligned} \tag{4.8}$$

At this moment it is not yet explained why a particular pivot is chosen to lead to a solution of the *LCP*, this will be explained in the next section. However the impact of a pivot on the *LCP* can be visualised clearly, see figure 4.1 and 4.2 .



a) first pivot b) second pivot c) after pivoting

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} \begin{bmatrix} i_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} 1/2 & -1/2 \\ 1/2 & 1/2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix}$$

Figure 4.1. The pivoting process

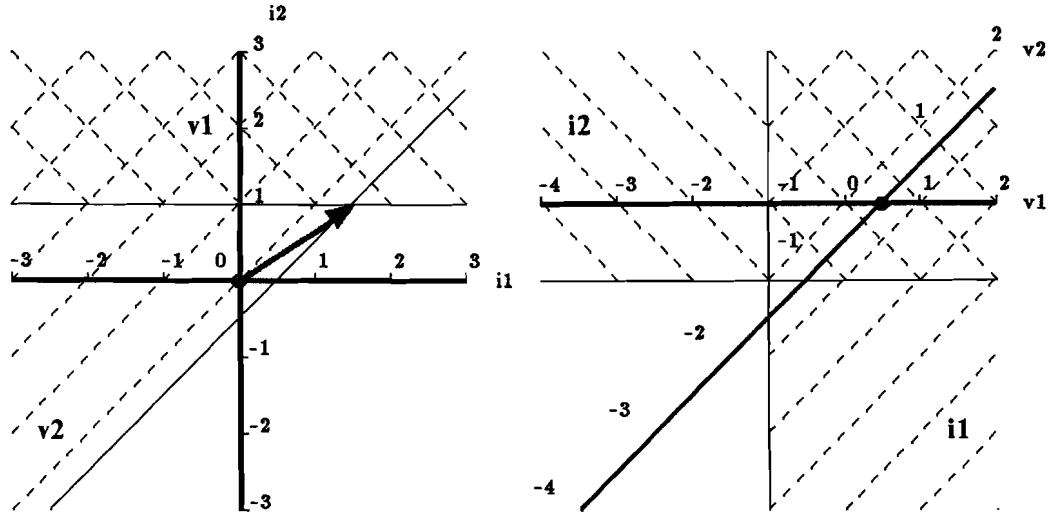
The spaces that are allowed for v_1 and v_2 ($v \geq 0$) are shaded. It is seen that the double shaded area is the solution space, v_1 and v_2 both positive, so the origin ($i = 0$) has to be moved to the double shaded area. In figure 4.1 v_1 is negative and v_2 zero. By exchanging v_1 and i_1 through a pivot on m_{11} we find that v_1 is positive but now v_2 has turned negative. Through a pivot on m_{22}, both v_1 and v_2 become positive and a solution is found.

Van de Panne allows only diagonal pivots or diagonal block pivots, this is why this method is called the complementary variant of Lemke, because it is never possible that corresponding elements of i and v are together in one vector. A diagonal pivot is not possible if the corresponding m_{ii} is zero. In this case a block pivot is performed. A block pivot is a set of off-diagonal pivots such that after this sequence of pivots v_i and i_i have been exchanged, and the condition i and v contain no complementary elements is still valid. In figure 4.2. we see a block pivot performed.

4.2.3 The solution process

Every iteration step during the solution process a valid range is determined for lambda, this range is bounded by a lower- ($\underline{\lambda}$) and an upper bound of lambda ($\bar{\lambda}$). Through pivoting the lower and upper bounds can be changed. Beside of that every step of the solution process has its own direction, meaning the bound that is treated, in case of a lower bound the direction is down and with an upper bound it is up. The aim of the method is to unblock the increase or decrease of lambda at the critical value.

Initially the direction is set to down, the lower bound is equal to the minimal value



a) block pivot

b) after pivoting

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 1/2 \end{bmatrix} \begin{bmatrix} i_2 \\ v_2 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ v_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3/2 \end{bmatrix} \begin{bmatrix} i_2 \\ i_1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_2 \\ v_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3/2 \end{bmatrix}$$

Figure 4.2. The pivoting process

possible, such that v remains non negative :

$$\underline{\lambda} = max(-b_i/a_i : a_i>0) \tag{4.9}$$

The upper bound is initially equal to infinity. Next the diagonal element in the row which causes the lower bound is determined. Two cases arise :

A) The diagonal element is not zero. This element will be used as a pivot, four possibilities in connection with lambda can appear :

Suppose the direction is down and the pivot has to be performed on row k, see equation 4.10.

$$\begin{aligned} v_k &= b_k + \underline{\lambda} \cdot a_k = 0 \qquad a_k > 0 \quad b_k < 0 \\ v_k &= \sum_{i=1}^{n} m_{ki} \cdot i_i + b_k + \underline{\lambda} \cdot a_k \\ i_k &= \frac{1}{m_{kk}} \left\{ v_k - \sum_{i=1 i!=m}^{n} m_{ki} \cdot i_i - b_k - \underline{\lambda} \cdot a_k \right\} \\ &\qquad = 0 \qquad\quad = 0 \end{aligned} \tag{4.10}$$

With respect to lambda two different cases are recognised :

1 : $m_{kk} > 0$ A drop of lambda will cause i_k to be more positive. So lambda may further drop, implying a new lower bound. However lambda may not

rise above the old lower bound, so the upper bound is equal to the old lower bound. Lambda is unblocked.

2 : $m_{kk} < 0$ A rise of lambda will cause i_k to be more positive. So lambda may further rise, implying a new upper bound. However lambda may not drop below the old lower bound, so the lower bound is equal to the old lower bound. Lambda is deflected.

Suppose the direction is up and the pivot has to be performed on row k, see equation 4.11.

$$v_k = b_k + \lambda \cdot a_k = 0 \qquad a_k < 0 \quad b_k > 0$$

$$v_k = \sum_{i=1}^{n} m_{ki} \cdot i_i + b_k + \lambda \cdot a_k \tag{4.11}$$

$$i_k = \frac{1}{m_{kk}} \left\{ v_k - \sum_{i=1 i!=m}^{n} m_{ki} \cdot i_i - b_k - \lambda \cdot a_k \right\}$$

$$= 0 \qquad = 0$$

Again with respect to lambda two different cases arise :

3 : $m_{kk} > 0$ A rise of lambda will cause i_k to be more positive, so a new upper bound is determined. The lower bound is equal to the old upper bound. Lambda is deflected.

4 : $m_{kk} < 0$ A drop of lambda will cause i_k to be more positive, implying a new lower bound. The upper bound is equal to the old upper bound. Lambda is unblocked.

B) The diagonal element is zero. In this case the corresponding i variable (i_k) is handled exactly the same as lambda, the i variable is said to be blocked, lambda is kept at the current blocked value. Initially the direction of i_k is set to up, the upper bound is set to the minimal value possible :

$$i_k = (\min(\frac{b_i + a_i \cdot \lambda^*}{a_{ik}}) : a_{ik} < 0) \tag{4.12}$$

Suppose this minimum is found in row j then :

$$i_k^* = \frac{b_j + a_j \cdot \lambda^*}{a_{kj}} \tag{4.13}$$

The values of the basic variables are equal to :

$$v_i = a_{ik} \cdot i_k + b_i + a_i \cdot \lambda^* \tag{4.14}$$

If there is no negative element in the critical column a solution is not feasible and the algorithm terminates unsuccessfully.

Lambda and/or i have to be unblocked, by pivoting. Three possibilities arise :

$a_{kj} \neq 0$: a_{kj} and a_{jk} are used as a block pivot, lambda and i_k are unblocked.

$a_{kj} = 0 \wedge a_{jj} \neq 0$: See A, a_{jj} is used as a pivot and i_k is unblocked. The impact of the pivot is the same as depicted for lambda.

$a_{kj} = 0 \wedge a_{jj} = 0$: Now i_j is increased and handled exactly the same as i_k in case B, there will be three blocked variables now, lambda, i_k and i_j. Depending on the encountered pivots this number may increase or decrease.

Suppose the non-basic variable can be varied down to zero, then the direction of the preceding fixed variable should be set to the opposite direction of the non-basic variable that has become zero.

This pivoting is maintained until lambda and possible blocked i variables have been moved down to zero. This implies that all b_i are greater than zero and the solution for the originally stated *LCP* is found.

The algorithm can terminate in three ways :

1) A solution is found for lambda is zero
2) No upper bound for lambda can be found.
3) No upper bound for a non-basic variable can be found.

In the latter two cases it was found by Lemke and Eaves that a solution is infeasible if the matrix M is copositive plus or belongs to class L. The complementary variant of van de Panne detects whether a solution is still feasible by examining the critical column for negative elements. So it is not necessary to establish beforehand whether M is copositive plus or belongs to L.

4.3 The program

In this section the program is presented, in a C-like language, as it is implemented for the simulator. The van de Panne algorithm tries to move lambda down to zero, however the real problem of the simulator is finding a dc solution. So during the iteration steps of van de Panne we keep track of the changes in the system matrix A and the source vector $\bar{b}$ and solve the equation :

$$L \cdot U \cdot \bar{x} = \bar{b} \tag{4.15}$$

If van de Panne finds that there are no columns on the stack which can be unblocked the x vector is updated with $\bar{x}$. At this place it is appropriate to note that the calculation of x and $\bar{x}$ make use of the sparsity of the source vector and the hierarchically stored circuit. Only leafcells and variables of x are treated which have a change in their corresponding source vector elements. So latency effects of the simulated circuit are fully exploited and used to speed up simulation. The updating of the leafcell variables $\bar{v}$ and $\bar{u}$ makes use of these latency effects.

Because the matrix M is not directly available in the simulator, the elements are derived via the pwl models. Performing a pivot, the value of the element in M is determined by $d\ v_k / d\ i_j$, four possibilities arise with respect to the value of a pivot in the pwl matrix and the matrix M :

1) The pivot is zero in the pwl model as well as in the matrix M. The van de Panne algorithm will search for a block pivot.
2) The pivot is non zero in the pwl model and zero in the matrix M. The van de Panne algorithm will search for a block pivot.

3) The pivot is zero in the pwl model and non zero in the matrix M. The van de Panne algorithm will take this element as a pivot only as a row of $0 + A_{11} \cdot x + A_{12} + \cdot u + A_{13} \cdot i + b_1$ can be added to the row of the pivot, resulting in a non zero pivot in the pwl model and the matrix M. If not, the program will perform an exit and generate an error message. The van

4) The pivot is non zero in both the pwl model and the matrix M. The van de Panne algorithm will take this element as a pivot.

First some global variables and functions are explained, the pwl model of the leafcells is extended with lambda and a :

$$\begin{bmatrix} 0 \\ \dot{u} \\ v \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ u \\ i \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \lambda \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \qquad (4.16)$$

Initially u is zero, so the pwl model reduces to :

$$\begin{aligned} 0 &= A_{11} \cdot x + A_{13} \cdot i + b_1 + \lambda \cdot a_1 \\ \dot{u} &= A_{21} \cdot x + A_{22} \cdot i + b_2 + \lambda \cdot a_2 \\ v &= A_{31} \cdot x + A_{32} \cdot i + b_3 + \lambda \cdot a_3 \end{aligned} \qquad (4.17)$$

However for users which are not content with this initialisation, the possibility exists to specify in the task file and/or the pwl description which elements of $\dot{u}$ have to be equal to zero. By pivoting on A_{22}, the elements of u and $\dot{u}$ are exchanged in such a way that all the elements on the right hand side, call this vector u', are equal to zero. Now the mathematical problem is exactly the same, the u' is set to zero and the solution process can be started, afterwards the elements of u' and $\dot{u}'$ have to be exchanged again. For simplicity we assume that the user is content with the initialisation $u = 0$, to avoid complex formulas, so equation 4.17 is correct.

Furthermore the program is divided in 7 steps, to distinguish the different actions performed to find a solution, which are also denoted by van Eijndhoven [9]. A variable is called active if it is the current variable handled by van de Panne. First the program is presented in a C-like language. the global variables and also the functions depicted with capitals, which influence leafcells only, will be explained.

Global variables :

lambda	:	extra non basic variable.
stack	:	the stack of van de Panne, containing the leafcells with their corresponding blocked column and direction.
top_leaf	:	top of stack leafcell.
top_col	:	blocked column of top of stack leafcell.
top_dir	:	direction of blocked variable of top of stack leafcell.
min_leaf	:	leafcell with minimal negative v.
min_row	:	row in which minimal negative v occurs.
proper	:	true if v of a leafcell is positive.
d	:	number of unblocked rows.
sign	:	direction of first blocked row on stack.
theta	:	value of minimal negative v element.
zero_i	:	true if current i variable became zero.

impr_leafs : list of improper leafs.
xbar_leafs : list of leafcells with nonzero $\bar{x}$.

operations :

push(leaf, column, dir)	:	push leafcell, blocking column and direction on the top of the stack
pop(leaf)	:	pop leafcell from the top of the stack

The steps of the program :

```
dc-solution()
{
step1 :  sparse solve( b̄ );
         for ( all leafcells ) {
              INIT_LEAF( leaf, proper, min_leaf, min_row, lambda);
              if ( ! proper ) put leaf in impr_leafs;
         }
step2 :  if ( impr_leafs not empty ) {
              for ( all leafs in impr_leafs ) INIT_PANNE( leaf );
              push( min_leaf, lambda, down );
              push( min_leaf, min_row, up );
         }
         while ( no solution ) {
step3 :       if ( top_col == lambda )
                   for ( all leafs in impr_leafs ) GEN_SOURCE( leaf, b̄ );
              else
                   GEN_SOURCE( top_leaf, b̄ );
              sparse solve( b̄);
              compose xbar_leafs;
              d = number of unblocked rows on stack;
              sign = direction of first blocked row on stack;
              if ( d == 0 ) {
step4 :            for ( all leafs in xbar_leafs )
                        CALC_THETA( leaf, min_leaf, max_theta, min_row);
                   θ = maximal theta;
                   min_leaf = leaf with maximal theta;
                   min_row = row with maximal theta;
                   if ( θ == infinite ) {
                        Lemke cannot solve; exit;
                   }
                   update_xvec;
                   for ( all leafs in xbar_leafs )
                        CALC_STEP( leaf, zero_i );
                   if ( top_col == lambda )
                        lambda += top_dir · θ;
                   if ( top_col == lambda && top_dir == down && lambda == 0 )
                        solution found;
                   else if ( top_col != lambda && top_dir == down && zero_i )
step5 :                 pop( top_leaf );
                   else
step6 :                 push( min_leaf, min_row , up );
              }
              else {
step7 :            for ( d leafcells on stack ) {
                        PANNE_PIVOT( leaf );
                        update system matrix;
                        LU decomposition update;
                   }
                   pop d leafs;
                   top_dir *= sign;
              }
         }
         for ( impr_leafs not empty ) END_PANNE( leaf );
}
```

The program

The leafcell functions :

INIT_LEAF : $\dot{u} = A_{21} \cdot x + b_2$
$v = A_{31} \cdot x + b_3$
If v variable is negative then mark this cell improper.
Keep track of minimal v value, corresponding leafcell and row.
Store them in lambda, min_leaf and min_row.

INIT_PANNE : Initialise the vector a, an element of a equals one if v is negative else zero. At the same time add lambda to the negative v variable.

GEN_SOURCE : if active variable is lambda then $\bar{b} = a \cdot$
if active variable is i_j then $\bar{b} = A_{13}[j]$

SIGN : return sign of $d\ v_k / d\ i_j$
if blocked variable is lambda then $d\ v_k / d\ i_j = A_{31}[k, .] \cdot \bar{x} + a_3[k]$
if blocked variable is i_j then $d\ v_k / d\ i_j = A_{31}[k, .] \cdot \bar{x} + A_{33}[k, j]$

CALC_THETA : Calculate $\bar{v}$ and maximal theta :
if leafcell is proper then $\bar{v} = A_{31} \cdot \bar{x}$
if blocked variable is lambda then $\bar{v} = A_{31} \cdot \bar{x} + a_3$
if blocked variable is i_j then $\bar{v} = A_{31} \cdot \bar{x} + A_{33}[., j]$
if ($v > 0 \wedge \bar{v} \cdot dir < 0$) then $\theta = \max(-v / \bar{v} \cdot dir)$

CALC_STEP : if leaf is proper : $\dot{u} = A_{21} \cdot \bar{x} \cdot \theta \cdot dir$
if blocked variable is lambda : $\dot{u} = (A_{21} \cdot \bar{x} + a_2) \cdot \theta \cdot dir$
if blocked variable is i_j : $\dot{u} = (A_{21} \cdot \bar{x} + A_{23}[., j]) \cdot \theta \cdot dir$

if blocked variable is i_j : $i_j + i_j + \theta \cdot dir$
else $v = v + \bar{v} \cdot \theta \cdot dir$

PANNE_PIVOT : determine pivot in pwl matrix,
if zero then exit
else perform pivot operation in pwl matrix
give update vectors for rank 1 update.

END_PANNE : $a_1 = a_2 = a_3 = 0$;

4.4 Examples

In this section two examples will be given to illustrate the van de Panne algorithm. The examples have already been used in section 4.2 to illustrate the behaviour of pivots. In the first example a matrix M is denoted with nonzero diagonal elements, the second one illustrates the actions of van de Panne if a zero pivot is encountered. The examples are self explanatory, so further comment is not necessary.

example 1 :

1)
$1 \leq \lambda < \varpi \quad \rightarrow \quad \underline{\lambda} = 1$
dir = down
pivot m_{11}

v	value	$b+a\cdot\lambda$	i_1	i_2	b	a
v_1	-1	0	1	1	-1	1
v_2	0	0	-1	1	0	0

2)

$1 \le \lambda \le 1 \rightarrow \underline{\lambda} = 1$
dir = down
pivot m_{22}

v	value	$b+a\cdot\lambda$	i_1	i_2	b	a
v_1	1	0	1	-1	1	-1
v_2	-1	0	-1	2	-1	1

3)

$-\varpi < \lambda \le 1 \rightarrow \lambda=0$
dir = down
solution found

v	value	$b+a\cdot\lambda$	i_1	i_2	b	a
v_1	1/2	1/2	1/2	-1/2	1/2	-1/2
v_2	1/2	1/2	1/2	1/2	1/2	-1/2

example 2 :

1)

$1 \le \lambda < \varpi \rightarrow \underline{\lambda} = 1$
dir = down
pivot m_{12}

v	value	$b+a\cdot\lambda$	i_1	i_2	b	a
v_1	-1	0	0	1	-1	1
v_2	1	1/2	-1	1	1/2	0

2)

$1 \le \lambda < \varpi \quad \underline{\lambda} = 1$
$-\varpi < i_1 = 1/2 \rightarrow i_1 = 1/2$
dir = up
pivot m_{21}

v	value	$b+a\cdot\lambda$	i_1	i_2	b	a
v_1	1	0	0	1	1	-1
v_2	3/2	1	-1	1	3/2	-1

3)

$-\varpi < \lambda \le 1 \rightarrow \lambda = 0$

dir = down
solution found

v	value	$b+a\cdot\lambda$	i_1	i_2	b	a
v_1	1	1	0	1	1	-1
v_2	3/2	3/2	-1	1	3/2	-1

5. Transient analysis

5.1 Introduction

The transient analysis is the solution over time of the outputs, given the inputs. Once more a valid representation and solution is searched for :

$$A(t)\cdot x(t) = b(t) \tag{5.1}$$

The problem of determining the solution over time of the circuit equations can be brought back to solving the pwl models of the leafcells. This means determining for every time step the jacobian that has to be inserted in the system matrix. More precisely the relation between u and $\dot{u}$ of the pwl model has to be solved :

$$0 = A_{11}\cdot x + A_{12}\cdot u + A_{13}\cdot i + b_1 \tag{5.2}$$
$$\dot{u} = A_{21}\cdot x + A_{22}\cdot u + A_{23}\cdot i + b_2 \tag{5.3}$$
$$v = A_{31}\cdot x + A_{32}\cdot u + A_{33}\cdot i + b_3 \tag{5.4}$$

where : A_{11} is an $m\cdot n$ matrix $m \leq n$
A_{22} is a $p\cdot p$ matrix
A_{33} is a $q\cdot q$ matrix

Depending on x and u, the solution of the *LCP* will be affected. In case an element of v turns out to be negative, the van de Panne algorithm has to be restarted, this will be explained in more detail in section 5.7. However every solution of the *LCP* will yield $i = 0$, so the pwl models simplify to :

$$0 = A_{11}\cdot x + A_{12}\cdot u + b_1 \tag{5.5}$$
$$\dot{u} = A_{21}\cdot x + A_{22}\cdot u + b_2 \tag{5.6}$$
$$v = A_{31}\cdot x + A_{32}\cdot u + b_3 \tag{5.7}$$

In general there are no closed form solutions for the calculation of u, so we must resort to numerical solutions. An integration rule defines the relation between u and $\dot{u}$ for a particular time point and can be stated as follows :

$$f(u, \dot{u}, t) = 0 \tag{5.8}$$

The solution process of the pwl models can shortly be denoted as follows. With the aid of the integration rule $\dot{u}$ is eliminated from equation 5.6, this will yield p equations which have u and x as unknown. By expressing u in x and substituting u in equation 5.5, we find the new jacobian and source vector, x is solved from the system matrix and now u, $\dot{u}$ and v in the pwl models can be solved.

For the solution over time of the pwl models various numerical integration methods are available. The integration rules are used to solve differential equations $\dot{u} = f(u, t)$, these rules have to be efficient even if the systems are not smooth. The differential equations may lack smoothness, for example because of rapidly varying differentiable terms, a situation which is often encountered in circuit analysis. We want to use methods which can solve these problems efficiently, in the simulator several linear multistep methods have been implemented, more specificly these integration rules are :

forward euler
backward euler
trapezium rule
two step backward differentiation method
two step optimal A-contractive method

In the first sections 5.2 until 5.6 an overview of the properties of linear multistep methods are presented. Hereafter the program and circuit oriented problems will be dealt with and finally some examples are given.

5.2 Linear multistep methods

Usually integration is performed for a time interval [0, T]. The linear multistep (*MS*) methods divide this interval in several time pieces so a series of time points t_0, t_1,..., t_n is obtained. In the following sections we will assume that the step size h is constant, so a uniform grid of t-values { $t = n \cdot h : n = 0,1,...$ } is obtained. The *MS* methods can be expressed as :

$$\sum_{i=0}^{p} a_i \cdot u_{n-i} - h \cdot \sum_{i=0}^{p} b_i \cdot \dot{u}_{n-i} = 0 \qquad (5.9)$$

A lot of different methods are possible, however by accuracy and stability constraints only some are useful in practice. Because the methods compute an approximation of the real solution, the parameters have to be investigated, to determine which values give the best accuracy and stability results. Perhaps the most well known methods are :

forward euler : $u_n - u_{n-1} - h \cdot \dot{u}_{n-1} = 0$
backward euler : $u_n - u_{n-1} - h \cdot \dot{u}_n = 0$
trapezium rule : $u_n - u_{n-1} - h/2 \cdot (\dot{u}_n + \dot{u}_{n-1}) = 0$

The *MS* methods are divided in two classes :

the explicit methods : $b_0 = 0$, in this class u_n is directly derived from the integration formula.

the implicit methods : $b_0 \neq 0$, in this class u_n as well as $\dot{u}_n$ have to be solved.

It is seen that the forward euler method belongs to the explicit methods and the backward euler and trapezium rule to the implicit. Although the forward euler method is straightforward, u_n is directly derived, it is not often used because of the poor accuracy and stability results. In spite of that it is often used as a start up for other methods, like for example for the trapezium rule. As for the implicit methods u_n as well as $\dot{u}_n$ have to be solved, so finding a solution takes more effort then in case of the explicit methods, on the other hand they give good accuracy and stability results.

In the following sub-sections some general properties of *MS* methods will be derived, which can mainly be found in Ruehli [18].

5.2.1 Accuracy properties

The accuracy of an integration rule is defined by the local truncation error (*LTE*).

definition: The *LTE* of an integration rule is the difference between the computed value u_n and the exact value of the solution $u(t_n)$, assuming that no previous errors have been made.

The *MS* methods are often designed to calculate exact the solution of a polynomial of order k. This is denoted in the order of the method.

definition: An *MS* method of order k implies that the local truncation error equals zero for integration of a polynomial with maximum degree k.

Dahlquist proved that only linear multistep methods which are of order up to two can be linearly stable in the entire left half plane, which is called *A*-stability. Let us therefor restrict ourselves to methods which are of order up to two, since we will investigate A-stable formulas only.

The requirement that an *MS* method should be of order k, imposes some constraints[1] on the parameters a_i and b_i. Assume we have a smooth test function $f(t)$, then the Taylor expansion of $f(t)$ at time point h is:

$$f(h) = f(0) + h \cdot f'(0) + \frac{h^2}{2} f''(0) + \frac{h^3}{3!} \cdot f'''(0) + o(h^4) \tag{5.10}$$

$$f'(h) = f'(0) + h \cdot f''(0) + \frac{h^2}{2} \cdot f'''(0) + o(h^3) \tag{5.11}$$

The linear multistep methods of order two can be expressed as:

$$a_0 \cdot u_n + a_1 \cdot u_{n-1} + a_2 \cdot u_{n-2} - h \cdot b_0 \cdot \dot{u}_n - h \cdot b_1 \cdot \dot{u}_{n-1} - h \cdot b_2 \cdot \dot{u}_{n-2} = 0 \tag{5.12}$$

By substituting u_n by $f(2h)$, u_{n-1} by $f(h)$ and u_{n-2} by $f(0)$ it follows:

$$(a_0 + a_1 + a_2) \cdot f(0) + (2 \cdot a_0 + a_1 - b_0 - b_1 - b_2) \cdot h \cdot f'(0) + \tag{5.13}$$

$$(2 \cdot a_0 + \frac{1}{2} \cdot a_1 - 2 \cdot b_0 - b_1) \cdot h^2 \cdot f''(0) + (\frac{4}{3} \cdot a_0 + \frac{1}{6} \cdot a_1 - 2 \cdot b_0 - \frac{1}{2} \cdot b_1) \cdot h^3 \cdot f'''(0) = 0$$

Now the method will be of order 2, in case the expressions before f(0), f'(0) and f''(0) equal zero, thus:

$$\begin{aligned} &a_0 + a_1 + a_2 = 0 \\ &2a_0 + a_1 - b_0 - b_1 - b_2 = 0 \\ &2a_0 + \frac{1}{2} a_1 - 2b_0 - b_1 = 0 \end{aligned} \tag{5.14}$$

and the *LTE* will be equal to:

$$LTE = [\, \frac{4}{3} a_0 + \frac{1}{6} a_1 - 2b_0 - \frac{1}{2} b_1 \,] \cdot h^3 \cdot f'''(0) \tag{5.15}$$

---

1. By Ruehli a general formula, in moment form, is derived for a k order MS method, which can simply be transformed to the above derived constraints.

$A_0 = 0$, $A_m = -m \cdot B_{m-1}$, $m = 1, 2, \ldots, k$

$$A_m = \sum_{i=0}^{p} i^m \cdot a_i \qquad B_m = \sum_{i=0}^{p} i^m \cdot b_i$$

Another constraint is the normalisation constraint, as proposed by Ruehli [18]:

$$\sum_{i=0}^{p} b_i = 0 \qquad (5.16)$$

The accuracy property together with the normalisation constraint generate four constraints, so second order two steps methods depend on two free parameters.

Introducing two new parameters, $c = -a_1$ and $d = b_1$, the integration parameters for second order accuracy can be expressed as :

$$\begin{aligned} a_0 &= 0.5\cdot(1 + c) & b_0 &= 0.25\cdot(2 + c - 2\cdot d) \\ a_1 &= -c & b_1 &= d \\ a_2 &= -0.5\cdot(1 - c) & b_2 &= 0.25\cdot(2 - c - 2\cdot d) \end{aligned} \qquad (5.17)$$

The local truncation error is often expressed as proposed by Peano :

$$LTE = - C_{k+1}\cdot h^{k+1}\cdot u^{k+1}(t) + o(h^{k+2}) \qquad (5.18)$$

where : k = the order of the method
C_{k+1} = the error constant

and $C_{k+1} = 1/3 - 1/2\cdot d$

Note that the expression of the *LTE* of a k-th order method implies that the error can be driven to zero as h is made smaller and smaller. On the other hand taking a small h will slow down the simulation, so an *MS* method exhibiting a small error constant allows "larger" step sizes.

The earlier mentioned rules exhibit an error constant of :

FE: $C_2 = -1/2$ $k = 1$
BE: $C_2 = 1/2$ $k = 1$
TR: $C_3 = 1/12$ $k = 2$

5.2.2 Stability properties

The global truncation error (*GTE*) determines the stability of the integration rule.

definition: The *GTE* of an integration rule is the difference between the computed value u_n and the exact value of the solution $u(t_n)$, assuming that only the initial condition is known exactly.

The stability of *MS* methods is determined by applying the test equation :

$$\dot{u} = \lambda\cdot u \quad \textit{with } \lambda \textit{ is constant and complex} \qquad (5.19)$$

Substitution of $\dot{u} = \lambda\cdot u$ leads to the following form for the *MS* methods :

$$\sum_{i=0}^{p} a_i\cdot u_{n-i} - h\cdot \sum_{i=0}^{p} b_i\cdot\lambda\cdot u_{n-i} = 0 \qquad (5.20)$$

By rewriting equation 5.20 and replacing $h \cdot \lambda$ by q it follows that:

$$\sum_{i=0}^{p} (a_i - q \cdot b_i) \cdot u_{n-i} = 0 \qquad (5.21)$$

Which is called the linear constant coefficient difference equation. The solutions are of the form:

$$u_n = n^{d_i} \cdot z_i^n \qquad i = 1,2,\ldots,m \qquad (5.22)$$

where : z_i *are the distinct roots of* $\sum_{i=0}^{p} (a_i - q \cdot b_i) \cdot z^{p-i}$

d_i *are the algebraic multiplications.*

Now unconditional stability is defined as:

definition: The *MS* method is said to be unconditional stable at q if all solutions $\{ u_n \}$ of 5.21 are bounded for fixed $h > 0$ as $n \to \infty$

Note that an *MS* method is unconditional stable if and only if equation 5.21 satisfies the root condition, i.e. $|z_i| \leq 1$ for all i and if $|z_i| = 1$ then z_i is simple. The set S of all q's at which the formula is stable is called the stability region.

The requirement that an *MS* method should be unconditional stable is very hard, and in practise, no *MS* method satisfies this condition. Unconditional stability has the whole complex plane as stability region. That's why other stability regions were searched for which are derived in such a way that S is adapted to the practical application of the integration rule.

Several restricted stability regions are:

A-stability: Stable in the entire left half complex plane.

A_0*-stability*: Stable on the negative real axis.

A_D*-stability*: Stable in some specified domain D, which in some sense is "large" enough.

A_α*-stability*: Derived from A_D*-stability*, where D consists of all $q \neq 0$ such that $| \pi - arg\ q | < \alpha$, $0 < \alpha < \pi/2$

Stiff-stability: Derived from A_D*-stability*, where D consists of a general rectangular domain containing most of the left half complex plane.

The stability regions for the *FE*, *BE* and *TR* rule are depicted in figure 5.1. It is seen that *FE* has a very small stability region and has almost no practical use. The *BE* and *TR* rule show good results and are both *A*-stable.

Now it is good to remind that for circuit simulation, *A*-stability is a first requisite. The right half complex plane might be unusable for the whole or partly, however this part is not necessary for the operation of an integration rule because a circuit signal can not grow unbounded. Note that in this case lambda would be positive. However reducing the stability region beyond the left half complex plane implies reducing the general use of the simulator.

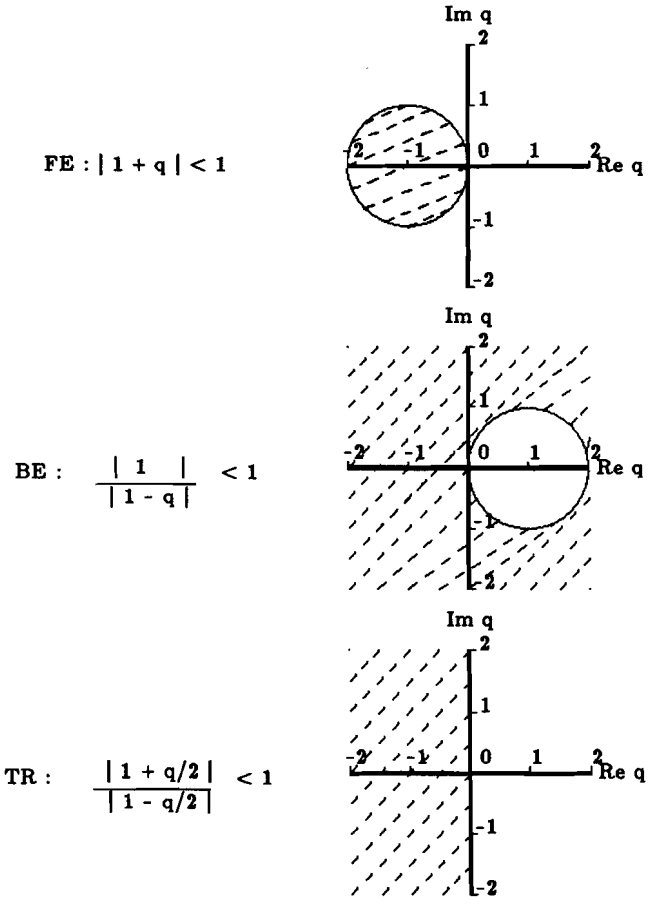


Figure 5.1. Stability regions for numerical integration rules

5.2.3 Contractivity properties

By Nevanlinna [15] a method is proposed which is said to be contractive if all solutions of the test equation $\dot{u} = \lambda \cdot u$ generated by the method are not only bounded but also non increasing.

definition: A linear multistep method is contractive at q if for solutions of

$$\sum_{i=0}^{p} (a_i - q \cdot b_i) \cdot u_{n-i} = 0$$

we have $\|U_n\| \leq \|U_{n-1}\|$ for all $n = p, p+1, \ldots$
where $U_n = (u_{n-p+1}, u_{n-p+2}, \ldots, u_n)$

The concepts related to stability can also be applied to contractivity. Thus a formula is said to be A-contractive if it is contractive for all q, Re $q \leq 0$. If we have a formula which is said to be A-contractive then by induction $\| U_n \| \leq \| U_{p-1} \|$ for all n and thus we have A-stability. In contrast to A-stability, A-contractivity is a local property to n and can be analysed easier then A-stability properties.

Clearly, if a method is contractive then by induction $\|U_n\| \leq \|U_0\|$ for all n:

$$\sum_{i=1}^{p} |a_i - q \cdot b_i| - |a_0 - q \cdot b_0| \leq 0 \qquad (5.23)$$

An *MS* method is contractive at $q = 0$ if and only if :

$$a_0 > 0 \wedge a_j \leq 0 \tag{5.24}$$

and contractivity at $q = \infty$ if and only if :

$$b_0 - \sum_{i=1}^{p} |b_i| \geq 0 \tag{5.25}$$

The method is A_0 contractive if and only if it is contractive both at $q = 0$ and $q = \infty$, i.e. only as 5.24 and 5.25 are satisfied.

The method is said to be A-contractive if it is contractive for all q in the left half-space. It has this property if and only if it is A_0 contractive and :

$$\sum_{i=1}^{p} (a_i^2 + b_i^2 \cdot \eta)^{1/2} - (a_0^2 + b_0^2 \cdot \eta)^{1/2} \leq 0 \tag{5.26}$$

The inequality 5.26 is obtained, if the contractivity condition is applied on the imaginary axis with $q = iy$, so $n = y^2$.

Second order accuracy and normalisation require that (written in moment form) :

$$A_0=0 \quad A_1 = b_0 = 1 \quad A_2 = 2 \cdot B_1$$

$$A_m = \sum_{i=0}^{k} \theta_i^m \cdot (-a_i) \qquad B_m = \sum_{i=0}^{k} \theta_i^m \cdot b_j$$

$$\theta_i = (t_n - t_{n-i}) / h_n$$

For the two steps method the condition for contractivity at $z = 0$ is :

$$a_0 > 0 \wedge a_i < 0 \quad i = 1, 2 \tag{5.27}$$

A condition which is just slightly stronger then equation 5.24, then equation 5.26 is equivalent to :

$$F(\eta) = \sum_{i=1}^{2} (-a_i) \cdot [1 + (b_i^2/a_i^2) \cdot \eta]^{1/2} - a_0 \cdot [1 + (b_0^2/a_0^2) \cdot \eta]^{1/2} \leq 0 \tag{5.28}$$

For $n \to 0$, i.e. when analysing contractivity on the imaginary axis near $q = 0$, one finds that :

$$F(\eta) = -\frac{1}{2} \cdot F \cdot \eta + o(\eta^2) \tag{5.29}$$

$$\text{where : } F = \sum_{i=0}^{2} b_i^2/a_i$$

It follows that $F \geq 0$, however second order conditions imply $F \leq 0$. So the only possible solution that can be achieved is $F = 0$.

For given values of the a_i satisfying $A_0 = 0$ and $A_1 = 1$ we seek the unique extremum of the quadratic function $F(b_i)$, which must be a maximum, subject to the constraints $B_0 = 1$ and $B_1 = A_2/2$, in an attempt at making $F = 0$. The maximum $F = F_{max}$ is taken at :

$$b_i = (i + \frac{1}{2} A_2)(-a_i) , \quad i = 0, 1, 2 \tag{5.30}$$

and F_{max} turns out to be zero identically in the a_i's. Furthermore $F < 0$ for any other b_i's, thus the solutions found are the only two step A-contractive methods. So we find one extra constraint and the two step A-contractive methods will depend on one free parameter, for which we can choose $c = -a_1$. Comparing the results with equation 5.17 we see that the parameter $c = 2 \cdot d^{1/2}$:

$$a_0 = \frac{1}{2}(1 + c) \qquad b_0 = \frac{1}{4}(2 + c - c^2) \tag{5.31}$$
$$a_1 = -c \qquad b_1 = \frac{1}{2}c^2$$
$$a_2 = -\frac{1}{2}(1 - c) \qquad b_2 = \frac{1}{4}(2 - c - c^2)$$
$$0 \le c \le 1$$

For $c = 1$ and $c = 0$ we obtain the trapezium rule with step length h and the trapezium rule with step length $2h$ respectively.

$$u_n - u_{n-1} - \frac{h}{2} \cdot (\dot{u}_n + \dot{u}_{n-1}) = 0 \tag{5.32}$$

$$\frac{1}{2} \cdot u_n - \frac{1}{2} \cdot u_{n-2} - \frac{h}{2} \cdot (\dot{u}_n + \dot{u}_{n-2}) = 0 \tag{5.33}$$

5.2.4 Numerical damping

Another important property is the numerical damping of an *MS* method, intuitively numerical damping can be viewed as excessive stability.

definition : Numerical damping (or amplification) is the decrease (or increase) in the amplitude which results from applying the numerical method to a lossless resonance circuit (poles at $\pm i\lambda$).

The test equation that is used, is given by :

$$\dot{u} = i \cdot \lambda \cdot u \quad \textit{with } \lambda \textit{ is real} \tag{5.34}$$

This equation has no inherent damping and thus the damping of the numerical response must be attributed to the integration method.

For the analysis of the test circuit we use a two step *MS* method, the characteristic equation will be :

$$\sum_{i=0}^{2} (a_i - i \cdot \lambda \cdot h \cdot b_i) \cdot u_{n-i} = 0 \tag{5.35}$$

with $u_n = z^n$ *we get*

$$\sum_{i=0}^{2} (a_i - i \cdot \lambda \cdot h \cdot b_i) \cdot z^{2-i} = 0$$

The roots of this equation are, with $q = h \cdot \lambda$:

$$z_{1,2} = \frac{-(a_1 - i \cdot q \cdot b_1) \pm (R - i \cdot I)^{1/2}}{2 \cdot (a_0 - i \cdot q \cdot b_0)} \tag{5.36}$$

where : $R = a_1^2 - 4 \cdot a_0 \cdot a_2 - q^2 \cdot (b_1^2 - 4 \cdot b_0 \cdot b_2)$

$$I = 2 \cdot q \cdot (a_1 \cdot b_1 - 2 \cdot (a_2 \cdot b_0 + a_0 \cdot b_2))$$

The complex square root in equation 5.36 has two values given by :

$$(.)^{1/2} = (R^2 + I^2)^{1/4} . e^{i \cdot (\theta + k\pi)} \tag{5.37}$$

where : $\theta = \frac{1}{2} \cdot \tan^{-1}(I/R)$ and $k = 0, 1$

We are interested in the modulus of the principal root of the oscillatory solution to measure the amplitude decay as a function of $t = nh$, the two amplitude functions being :

$$u_n = | z_{1,2} |^n = \frac{(A^2 + q^2 \cdot B^2)^{n/2}}{[2 (a_0^2 + q^2 b_0^2)]^n} \tag{5.38}$$

where : $A = -a_0 \cdot a_1 - q^2 \cdot b_0 \cdot b_1 \pm (R^2 + I^2)^{1/4} \cdot (a_0 \cdot \cos\theta + q \cdot b_0 \cdot \sin\theta)$
$B = (a_1 \cdot b_0 - a_0 \cdot b_1) \pm (R^2 + I^2)^{1/4} \cdot (a_0 \cdot \sin\theta + q \cdot b_0 \cdot \cos\theta)$

Do we apply the test equation to the *BE* method then we find :

$$z = \frac{1}{1 - i \cdot \lambda \cdot h} \tag{5.39}$$

which correspondents to an exponential decay of :

$$u_n = |z|^n = (\frac{1}{(1 + \lambda^2 \cdot h^2)^{1/2}})^n \approx (1 - \lambda^2 \cdot h^2)^{n/2}$$

$$u_n \approx (e^{-h^2 \cdot \lambda^2}) \quad h^2 > \lambda^2 \tag{5.40}$$

This shows why the *BE* method is not suitable for large time steps h, since the numerical damping depends exponentially on *h*.

In figure 5.2 the response is depicted for a lossless resonance circuit, the same experiment is performed in Ruehli. The amplitude for the *BE* method is so strongly damped that the solution vanishes after very few cycles. The performance of the *BDF*2 method is also poor. Do we look at the optimal *ACT*2 method we see that the numerical damping is better. The *TR* rule exhibits no damping, it shows an error with an oscillatory behaviour.

The damping is dependent of the step size, here a step size of 0.5 sec. is used which yields about 12 points per cycle. Reducing the step size to 0.1 sec.(about 60 points per cycle) will yield a much better result with respect to the numerical damping for the *ACT*2 and *BDF*2 method, while the numerical damping of the *BE* method remains poor, see figure 5.3.

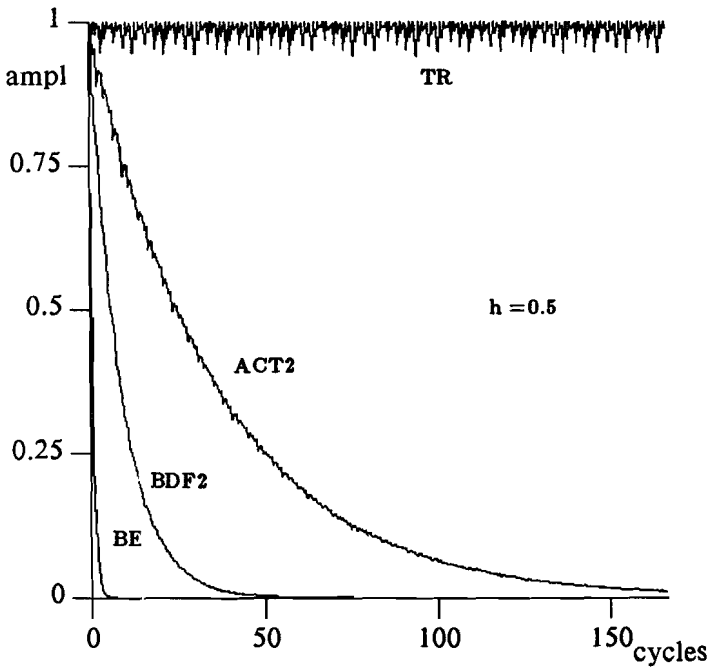


Figure 5.2. The damping for a lossless resonance circuit for h = 0.5

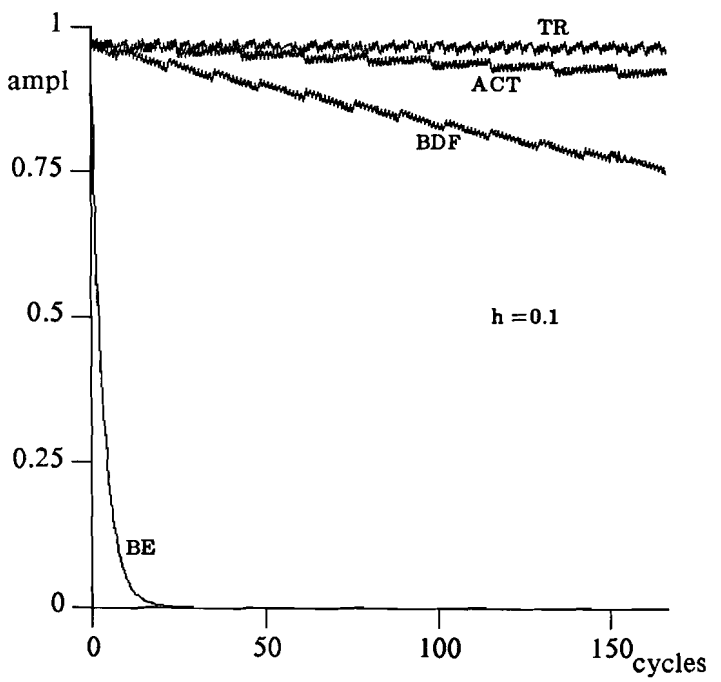


Figure 5.3. The damping for a lossless resonance circuit for h = 0.1

5.3 Two step backward differentiation method

A particular class of the *MS* methods is called the backward differentiation methods which are defined as:

$$\sum_{i=0}^{p} a_i \cdot u_{n-i} - h \cdot b_0 \cdot \dot{u}_n = 0 \qquad (5.41)$$

By the normalisation constraint it is found that $b_0 = 1$. The two steps backward differentiation method (*BDF*2) is characterised by:

$$a_0 \cdot u_n + a_1 \cdot u_{n-1} + a_2 \cdot u_{n-2} - h \cdot \dot{u}_n = 0 \qquad (5.42)$$

The three accuracy constraints, see equation 5.14 determine the three parameters of *BDF*2:

$a_0 = 3/2$
$a_1 = -2$
$a_2 = 1/2$

and the error constant of the *LTE* is:

$C_3 = 1/3$

The stability properties can be found from equation 5.21:

$$(3/2 - q) \cdot z^2 - 2 \cdot z + 1/2 = 0 \qquad (5.43)$$

For all q we search the regions where the roots satisfy $|z_i| \leq 1$.

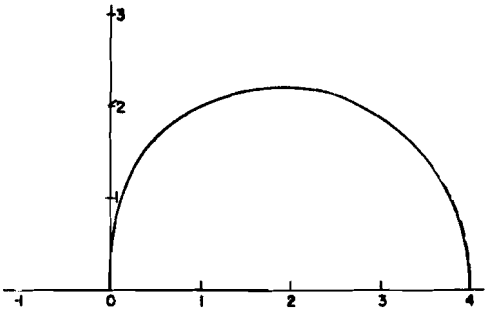


Figure 5.4. The stability region for the *BDF*2 method

As seen from the picture, the *BDF*2 method is *A*-stable and therefor applicable in the simulator.

5.4 Two step A-contractive methods

The two step *A*-contractive methods are restricted by 5 constraints, so they depend on one free parameter, see equation 5.31. An *A*-contractive method having certain optimality property in terms of a bound for the global truncation error, see Nevanlinna [15] is given for $c = 2/3$:

$$\frac{5}{6} \cdot u_n - \frac{2}{3} \cdot u_{n-1} - \frac{1}{6} \cdot u_{n-2} - h \cdot \frac{5}{9} \cdot \dot{u}_n - h \cdot \frac{2}{9} \cdot \dot{u}_{n-1} - h \cdot \frac{2}{9} \cdot \dot{u}_{n-2} = 0 \qquad (5.44)$$

Two step A-contractive methods

This method is of order 2, A-contractive and thus A-stable, for this reasons it is actually implemented in the simulator.

The stability properties can be found from equation 5.21.

$$(\frac{5}{6} - \frac{5}{9}\cdot q)\cdot z^2 + (-\frac{2}{3} - \frac{2}{9}\cdot q)\cdot z + (-\frac{1}{6} - \frac{2}{9}\cdot q) = 0 \tag{5.45}$$

5.5 Integration with variable time step

Until now we investigated the integration rules under the presumption that a uniform grid $\{ t_n \}$, $t_n = n\cdot h$ is used. In practice this presumption does not hold and a variable grid $\{ t_n \}$, whose now time step is $h_n = t_n - t_{n-1}$ and for which we define $t_n - t_{n-j} = h_n\cdot\theta_j$, is used.

Most circuits exhibit stiffness, which implies widely separated eigenvalues. Suppose a circuit has an exact solution of the form :

$$u(t) = 1 + e^{-\lambda_1\cdot t} - e^{-\lambda_2\cdot t} \tag{5.46}$$

and $\lambda_1 = 1$ and $\lambda_2 = 10^6$. Note that to capture the behaviour of the solution an integration of 5 sec. is needed to find the response of λ_1. To observe the response for λ_2, a time step is needed of 10^{-6} sec., for uniform time steps this implies 5 10^6 time points for the total integration. However with a variable grid the response for λ_2 can be captured in 5 steps of 10^{-6} sec. and after that 5 time steps of 1 sec. are used to capture the behaviour of λ_1. In this way a lot of computational effort can be saved. Unfortunately not all the *MS* methods can be applied with variable step size. For example the *FE* method shows growing oscillation when applied with variable step size.

So again the question rises what are the accuracy, the stability and the A-contractivity properties of the *MS* methods. In the subsequent part of this section two methods are proposed to maintain second order accuracy for the *BDF*2 and the *ACT*2 methods as found in Sangiovanni [20] and Dahlquist [7].

*The BDF*2 *method :*

In Nevanlinna [15] it is experimentally proved that the *BDF*2 method is not A-stable when an increasing step size sequence is applied. For second order methods we derived the coefficients of the two steps *MS* methods in section 5.2.1. When a variable grid is used for the time step, the computation of the coefficients becomes more complicated and depends on the time steps.

The *BDF*2 method can be expressed as :

$$a_0\cdot u_n + a_1\cdot u_{n-1} + a_2 u_{n-2} - h_n\dot{u}_n = 0 \tag{5.47}$$

Let us consider a general polynomial of order 2 :

$$u(t) = c_0 + c_1\cdot t + c_2\cdot t^2 \tag{5.48}$$

Applying the integration rule yields :

$$a_0\cdot(c_0 + c_1\cdot t_n + c_2\cdot t_n^2) + a_1\cdot(c_0 + c_1\cdot t_{n-1} + c_2\cdot t_{n-1}^2) + \tag{5.49}$$

$$a_2\cdot(c_0 + c_1\cdot t_{n-2} + c_2\cdot t_{n-2}^2) - h_n\cdot(c_1 + 2\cdot c_2\cdot t_n) = 0$$

Substituting $t_{n-1} = t_n - h_n$ and $t_{n-2} = t_n - h_n - h_{n-1}$ yields :

$$\begin{aligned} &a_0 + a_1 + a_2 = 0 \\ &a_1\cdot h_n + a_2\cdot(h_n + h_{n-1}) = -h_n \\ &a_1\cdot h_n^2 + a_2\cdot(h_n + h_{n-1}) = 0 \end{aligned} \qquad (5.50)$$

and we find the following expressions for the parameters of the *BDF* 2 method :

$$a_0 = \frac{2\cdot h_n + h_{n-1}}{h_n + h_{n-1}} \qquad (5.51)$$

$$a_1 = \frac{-h_n - h_{n-1}}{h_{n-1}}$$

$$a_2 = \frac{h_n^2}{h_{n-1}\cdot(h_n + h_{n-1})}$$

The ACT 2 methods :

The *ACT* 2 method can become instable by a bad combination of time dependence (lambda depends on t) of the problem and step size changes. For the *ACT* 2 methods an extension is defined, by the requirement that the contractivity norm be constant with respect to n, i.e. independent of the step size changes. Furthermore, if the methods are expressed in terms of a special, uniquely chosen weighted average of the current step size and the previous step size :

$$h_n^* = a_0\cdot h_n - a_2\cdot h_{n-1} \qquad (5.52)$$

then the a_i coefficients too are constant with respect to n.

The *ACT* 2 methods can be written as :

$$a_0\cdot u_n + a_1\cdot u_{n-1} + a_2\cdot u_{n-2} - h_n^*\cdot(b_{0,n}\cdot \dot{u}_n + b_{1,n}\cdot \dot{u}_{n-1} + b_{2,n}\cdot \dot{u}_{n-2}) = 0 \qquad (5.53)$$

and the formula parameters are :

$$\begin{aligned} a_0 &= \frac{1}{2}\cdot(1 + c) & b_{0,n} &= \frac{1}{4}\cdot(1 + c + u_n + v_n) \\ a_1 &= -c & b_{1,n} &= \frac{1}{2}\cdot(1 - u_n) \\ a_2 &= -\frac{1}{2}\cdot(1 - c) & b_{2,n} &= \frac{1}{4}\cdot(1 - c + u_n - v_n) \end{aligned} \qquad (5.54)$$

with : $u_n = (1 - c^2)/(1 + e_n\cdot c)^2$

$$v_n = e_n^2\cdot c\cdot(1 - c^2)/(1 + e_n\cdot c)^2$$

$$e_n = \frac{h_n - h_{n-1}}{h_n + h_{n-1}}$$

and for A-contractivity : $0 \le c \le 1$

The local truncation error relative to h_n^* satisfies :

$$E^*(t_n) = -C^*_{3,n} \cdot h_n^{*3} \cdot u'''(t_n^*) \tag{5.55}$$

where the error constant :

$$C^*_{3,n} = \frac{1}{6} \cdot (A^*_{3,n} + 3 \cdot B^*_{2,n}) \tag{5.56}$$

is expressed in the moments :

$$A^*_{3,n} = \sum_{i=0}^{2} (\theta^*_{i,n})^3 \cdot a_{i,n} \qquad B^*_{2,n} = \sum_{i=0}^{2} (\theta^*_{i,n})^2 \cdot b_{i,n} \tag{5.57}$$

$$\theta_{0,n} = 0 \qquad \theta_{1,n} = (3 + 3 \cdot \epsilon_n)/(3 + 2 \cdot \epsilon_n) \qquad \theta_{2,n} = 6/(3 + 2 \cdot \epsilon_n)$$

For the optimal *ACT* 2 method, $c = 2/3$, the parameters become :

$$\begin{aligned} a_0 &= 5/6 & b_{0,n} &= 5/12 + 1/4 \cdot (u_n + v_n) \\ a_1 &= -2/3 & b_{1,n} &= 1/2 \cdot (1 - u_n) \\ a_2 &= -1/6 & b_{2,n} &= 1/12 + 1/4 \cdot (u_n - v_n) \end{aligned} \tag{5.58}$$

$$h_n^* = 5/6 \cdot h_n + 1/6 \cdot h_{n-1}$$

$$v_n = \frac{5}{(3 + 2 \cdot e_n)^2}$$

$$u_n = \frac{2}{3} \cdot e_n^2 \cdot v_n$$

5.6 Time step control

Until now nothing has been told about the time step control, there has to be found a mechanism to derive a new time step. The time step control causes the step to increase (decrease) whenever the error was over- (under-) estimated. Variable step sizes are often used to control the magnitude of the local truncation error and also to minimize the computation time, the time step must be chosen as large as possible provided that the desired accuracy is achieved. Two methods to derive the new step size will be presented, the first one is proposed in Ruehli [18] and the second one in Sangiovanni [20]

method according to Ruehli :

The local truncation error can be compared with a user defined tolerance T for the local error. The object is to search an h_{n+1} for which $LTE_n = T$, thus :

$$h_{n+1} = (\frac{T}{|LTE_n|})^{1/k+1} \cdot h_n \tag{5.59}$$

where : T = *user defined accuracy*

This strategy causes the step to increase (decrease) whenever the error was over-

(under-) estimated. LTE_n is derived by passing the k-th degree lagrange interpolation polynomial through x_{n-k-1}, ...,x_{n-1} and evaluating it at t_n, this yields

$$x_{e,n} = \sum_{i=1}^{k+1} d_{i,n} \cdot x_{n-i} \tag{5.60}$$

$$d_{i,n} = \prod_{j=1 \; i!=j}^{k+1} (t_n - t_{n-j}) / (t_{n-i} - t_{n-j}) \qquad i = 1, \ldots, k+1$$

For the two steps second order *MS* methods with uniform step sizes, we find after evaluating equation [5.60] and noticing that $t_n = t_{n-1} + h_n$, $t_{n-2} = t_{n-1} - h_n \cdot (\theta_{2,n-1} - 1$ and $t_{n-3} = t_{n-1} - h_n \cdot (\theta_{3,n-1} - 1)$:

$$x(t_n) - x_e(t_n) \approx \frac{1}{6} \cdot \theta_{2,n} \cdot \theta_{3,n} \cdot h_n^3 \cdot x'''(t_{n-1}) \tag{5.61}$$

and,

$$LTE(t_n) \approx - \frac{6 \cdot C_{3,n}}{\theta_{2,n} \cdot \theta_{3,n}} [x(t_n) - x_e(t_n)] \tag{5.62}$$

where : $C_{3,n} = \frac{1}{6} \cdot (A_{3,n} + 3 \cdot B_{2,n})$

The *LTE* of the *BDF*2 method specialises to,

$$BDF2: \qquad LTE(t_n) = \frac{1}{\theta_{3,n}} \cdot [x(t_n) - x_e(t_n)] \tag{5.63}$$

and the *LTE* of the *ACT*2 method relative to h_n^* specialises to :

$$ACT2: \quad LTE(t_n) = \frac{-6 \cdot C_{3,n}^*}{\theta_{1,n}^* \cdot \theta_{2,n}^* \cdot \theta_{3,n}^*} \cdot [x(t_n) - x_e(t_n)] \quad \textit{relative } h_n^* \tag{5.64}$$

method according to Sangiovanni :

By Sangiovanni a method is described which is used by *SPICE*2. The *LTE* of an *MS* method is used to find the new step size as follows :

$$| LTE_{n+1} | = | \frac{C_{k+1} \cdot h_{n+1}^{k+1}}{(k+1)!} \cdot u^{k+1}(t_{n+1}) | + h^{k+2} \tag{5.65}$$

transforming equation 5.65 yields :

$$h_{n+1} \leq | \frac{(k+1)! \cdot T}{C_{k+1} \cdot u_{k+1}(t_{n+1})} |^{1/k+1} \tag{5.66}$$

where T is a user defined accuracy for the *LTE*.

The only unknown term is $u^{k+1}(t_{n+1})$, as it cannot be computed exactly, an approximation is used that is called "divided differences" :

$$DD_1(t_{n+1}) = \frac{u_{n+1} - u_n}{h_{n+1}} \tag{5.67}$$

$$DD_2(t_{n+1}) = \frac{DD_1(t_{n+1}) - DD_1(t_n)}{h_{n+1} + h_n}$$

$$\vdots$$

$$DD_{k+1}(t_{n+1}) = \frac{DD_k(t_{n+1}) - DD_k(t_n)}{h_{n-i}}$$

By Dahlquist [7] it is shown that:

$$u^{k+1}(t_{n+1}) \approx (k+1)! \cdot |DD_{k+1}(t_{n+1})| \tag{5.68}$$

and the new step size will be:

$$h_{n+1} \leq \left(\frac{T}{C_{k+1} |DD_{k+1}(t_{n+1})|} \right)^{1/k+1} \tag{5.69}$$

Given a step h_{n+1}, u_{n+1} is calculated. Then $DD_{k+1}(t_{n+1})$ is computed and h_{n+1} is checked with inequality 5.69. If h_{n+1} satisfies the test it is accepted and commonly h_{n+2} is set equal to the right hand side of 5.69. If h_{n+1} does not satisfy the test, h_{n+1} is rejected and a new h_{n+1} is given by the right hand side of 5.69.

Finally a convenient estimation for the local truncation error has to be defined. A bound for T, the user defined accuracy, is given as a combination of the absolute and relative error:

$$T = e_a + e_r \cdot |u_{n+1}| \tag{5.70}$$

5.7 The program

This section is divided in two parts. First some general concepts and formulas of the program will be stated. Thereafter the program will be presented in a C-like language.

General concepts and formulas :

The formulas that are actually implemented in the simulator, have been treated in the preceding sections, for the sake of completeness they are restated here:

FE: $u_n - u_{n-1} - h_n \cdot \dot{u}_{n-1} = 0$

BE: $u_n - u_{n-1} - h_n \cdot \dot{u}_n = 0$

TR: $u_n - u_{n-1} - \frac{h_n}{2} \cdot (\dot{u}_n + \dot{u}_{n-1}) = 0$

BDF 2: $a_{0,n} \cdot u_n - a_{1,n} \cdot u_{n-1} + a_{2,n} \cdot u_{n-2} - h_n \cdot \dot{u}_n = 0$

$$a_0 = \frac{2 \cdot h_n + h_{n-1}}{h_n + h_{n-1}} \quad a_1 = \frac{-h_n - h_{n-1}}{h_{n-1}} \quad a_2 = \frac{h_n^2}{h_{n-1} \cdot (h_n + h_{n-1})}$$

ACT 2: $\frac{5}{6} \cdot u_n - \frac{2}{3} \cdot u_{n-1} - \frac{1}{6} \cdot u_{n-2} - h_n^* \cdot (b_{0,n} \cdot \dot{u}_n + b_{1,n} \cdot \dot{u}_{n-1} + b_{2,n} \cdot \dot{u}_{n-2}) = 0$

$$b_{0,n} = 5/12 + 1/4\cdot(u_n + v_n) \quad b_{1,n} = 1/2\cdot(1 - u_n) \quad b_{2,n} = 1/12 + 1/4\cdot(u_n - v_n)$$

$$h_n^* = 5/6\cdot h_n + 1/6\cdot h_{n-1} \quad v_n = \frac{5}{(3 + 2\cdot e_n)^2} \quad u_n = \frac{2}{3}\cdot e_n^2\cdot v_n \quad e_n = \frac{h_n - h_{n-1}}{h_n + h_{n-1}}$$

The framework of the program is designed such that a general integration over time is generated. In this way it is easy to add other integration methods without changing the total program. For this purpose all the leafcell operations are performed by special functions, so changes in the integration rule or pwl models will only change (some of) these functions, and are not visible in the main program.

The simulator is event driven, meaning that during simulation every cell is assigned its own time step. Every step of the simulation the leafcell with the nearest event is handled. Here the nearest event is defined by the minimum of the so called *dynamic event* or *panne event*, meaning :

1) *dynamic event* : The leafcell has a dynamic event in case it reaches the end of a time step interval, this implies that a new time step for the leafcell has to be calculated. In case the new time step is different from the previous time step a new jacobian and source vector is generated. The *LU* decomposition is updated and finally the pwl variables are updated
2) *panne event* : The leafcell has reached a boundary of a polytope, one or more elements of v have become zero, so the leafcell has to be handled by the van de Panne algorithm. Note that a treatment by the van de Panne algorithm yields no progress in time, at this time point a new valid jacobian is searched for. The elements of x, u, $\dot{u}$ and v are recomputed and integration has to be restarted in case of a discontinuity in $\dot{u}$ was detected, otherwise integration is continued normally.

The advantage of an event driven simulation is exploiting the latency of the circuit. The circuit elements which are active have an event and are subsequently handled by the simulator. During simulation a list is composed of the fanout of the circuit elements currently being handled. The simulator will only handle those circuit elements which have an event and which belong to the list of the fanout.

For the operation of the *BE* method and the *TR* rule, we need to know the values of the leafcell variables at t_{n-1}. At time point zero and after a van de Panne event if the event cannot be solved by a simple pivot, these values are not present. They have to be preceded by an explicit integration rule, for this purpose the *FE* method is implemented. The same applies to the *BDF*2 and the *ACT*2 methods, however we need also to know the values at t_{n-2}, so these methods are preceded by the *FE* method and the *TR* rule.

For the calculation of the jacobian and the source vector the following strategy is applied. Not the actual variables of the leafcell are used but the divided differences :

$$\bar{x}_n = \frac{x_n - x_{n-1}}{h_n} \qquad \bar{u}_n = \frac{u_n - u_{n-1}}{h_n} \qquad (5.71)$$

$$\bar{\dot{u}}_n = \frac{\dot{u}_n - \dot{u}_{n-1}}{h_n} \qquad \bar{v}_n = \frac{v_n - v_{n-1}}{h_n}$$

$$\bar{\bar{x}}_n = \bar{x}_n - \bar{x}_{n-1} \qquad \bar{\bar{b}}_n = \bar{b}_n - \bar{b}_{n-1}$$

The pwl model, together with the general two step *MS* method gets the following form :

$$0 = A_{11}\cdot\bar{x}_n + A_{12}\cdot\bar{u}_n \tag{5.72}$$
$$\bar{i}_n = A_{21}\cdot\bar{x}_n + A_{22}\cdot\bar{u}_n \tag{5.73}$$
$$\bar{v}_n = A_{31}\cdot\bar{x}_n + A_{32}\cdot\bar{u}_n \tag{5.74}$$
$$a_0\cdot\bar{u}_n + (a_0 + a_1)\cdot\frac{1}{h_n}\cdot u_{n-1} + a_2\cdot\frac{1}{h_n}\cdot u_{n-2} - b_0\cdot h_n\cdot\bar{i}_n - (b_0 + b_1)\cdot\dot{u}_{n-1} - b_2\cdot\dot{u}_{n-2} = 0 \tag{5.75}$$

Eliminating $\bar{i}_n$ from equation 5.73 with the aid of the integration rule yields :

$$\bar{u}_n = h_n\cdot b_0\cdot(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot A_{21}\cdot\bar{x} + \tag{5.76}$$
$$(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot(-(a_0 + a_1)\cdot\frac{1}{h_n}\cdot u_{n-1} - a_2\cdot\frac{1}{h_n}\cdot u_{n-2} + (b_0 + b_1)\cdot\dot{u}_{n-1} + b_2\cdot\dot{u}_{n-2})$$

This elimination is only possible in case the inverse of the matrix $a_0\cdot I - h_n\cdot b_0\cdot A_{22}$ exists. In general the time step of the simulator is very small, preventing the matrix to degenerate. The matrix will approximately be equal to the identity matrix multiplied by a_0. Eliminating $\bar{u}_n$ from equation 5.72, using equation 5.76 yields :

$$0 = [A_{11} + h_n\cdot b_0\cdot A_{12}\cdot(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot A_{21}]\cdot\bar{x} + \tag{5.77}$$
$$A_{12}\cdot(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot(-(a_0 + a_1)\cdot\frac{1}{h_n}\cdot u_{n-1} - a_2\cdot\frac{1}{h_n}\cdot u_{n-2} + (b_0 + b_1)\cdot\dot{u}_{n-1} + b_2\cdot\dot{u}_{n-2})$$

Now the jacobian and the source vector are found as :

$$J = A_{11} + h_n\cdot b_0\cdot A_{12}\cdot(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot A_{21} \tag{5.78}$$

$$\bar{b} = A_{12}\cdot(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot \tag{5.79}$$
$$(-(a_0 + a_1)\cdot\frac{1}{h_n}\cdot u_{n-1} - a_2\cdot\frac{1}{h_n}\cdot u_{n-2} + (b_0 + b_1)\cdot\dot{u}_{n-1} + b_2\cdot\dot{u}_{n-2})$$

Inserting the jacobian and the source vector in the system matrix and solving $\bar{x}$ yields x. Now the leafcell variables can be calculated :

$$\bar{u}_n = h_n\cdot b_0\cdot(a_0\cdot I - h_n\cdot b_0\cdot A_{22})^{-1}\cdot A_{21}\cdot\bar{x} + \tag{5.80}$$
$$(a_0\cdot I - h_n\cdot A_{22})^{-1}\cdot(-(a_0 + a_1)\cdot\frac{1}{h_n}\cdot u_{n-1} - a_2\cdot\frac{1}{h_n}\cdot u_{n-2} + (b_0 + b_1)\cdot\dot{u}_{n-1} + b_2\cdot\dot{u}_{n-2})$$

$$\bar{i}_n = A_{21}\cdot\bar{x}_n + A_{22}\cdot\bar{u}_n \tag{5.81}$$
$$\bar{v} = A_{31}\cdot\bar{x}_n + A_{22}\cdot\bar{u}_n \tag{5.82}$$

At this moment the step size control is not to elaborated. It is assumed that the integration rules are of first order, this implies that the local truncation error is easy to determine and depends on the second derivative of u :

$$h_{n+1} = (\frac{e_a + e_r\cdot|u_n|}{|\ddot{u}_n|})^{1/2} \tag{5.83}$$

In this estimation, the error constant is set equal to one, in practice this may be to conservative. In future this estimation has to be more elaborated, choosing a suitable step size saves computation time, and a method as described in section 5.6 may be used.

However the new step size calculated with equation 5.83 sometimes yields a value that is

to big. For example suppose the signal on a node represents a sine wave and the simulation arrived at a top value, the second derative will equal zero and a large step size will be the result. For this reason is checked, during integration, whether $\bar{u}$ changes "fast" and if so the dynamic event of the leafcell is set back to :

$$dynamic\ event = t_n + \frac{\dot{\bar{u}}_n}{\dot{\bar{u}}_i} \cdot h_{n+1} \qquad t_n < t_i \leq t_{n+1} \tag{5.84}$$

Note that the jacobian and the source vector of the concerning leafcell are not affected and remain the same during this operation.

As long as the step remains the same, the jacobian need not be recomputed. Only the source vector will change, and the new $\bar{x}$ can be computed. However in case a step changes the jacobian as well as the source vector changes. Normally the changes of the system matrix are small, to avoid recomputing a whole new *LU* decomposition for the system matrix, an update is performed on the old *LU* decomposition. The method is based on a general update of the system matrix of the form :

$$A_{n+1} = A_n + X \cdot Y_t \tag{5.85}$$

where A is an $n.n$ matrix
X, Y are $n.m$ matrices with $m \ll n$

In Bennet [1], see also section 3.2.4, an algorithm is presented to modify the L and U matrices which enables this to be done in about $2 \cdot m \cdot n^2$ operations in the general case. The matrices X and Y are derived from the pwl model as follows :

$$J_{n+1} = A_{11} + h_{n+1} \cdot A_{12} \cdot (a_0 \cdot I - h_{n+1} \cdot b_0 \cdot A_{22})^{-1} \cdot A_{21}$$
$$J_n = A_{11} + h_n \cdot A_{12} \cdot (a_0 \cdot I - h_n \cdot b_0 \cdot A_{22})^{-1} \cdot A_{21}$$

$$\Delta J = b_0 \cdot A_{12} \cdot [h_{n+1} \cdot (a_0 \cdot I - h_{n+1} \cdot b_0 \cdot A_{22})^{-1} \cdot A_{21} - h_n \cdot (a_0 \cdot - h_n \cdot b_0 \cdot A_{22})^{-1} \cdot A_{21}] \tag{5.86}$$

$$X = A_{12} \tag{5.87}$$
$$Y = b_0 \cdot [h_{n+1} \cdot (a_0 \cdot I - h_{n+1} \cdot b_0 \cdot A_{22})^{-1} - h_n \cdot (a_0 \cdot - h_n \cdot b_0 \cdot A_{22})^{-1}] \cdot A_{21} \tag{5.88}$$

The update as it is implemented in the simulator is a rank 1 update meaning that X and Y have to be $n \cdot 1$ "matrices", these vectors will mainly be sparse. Most pwl models only need a rank 1 update, in case of a rank n update the rank 1 update is performed n times, in formula form with n equals update_nbr we obtain :

$$X[i, update\_nr] = A_{12}[i, update\_nbr] \tag{5.89}$$
$$Y[upd\_nr, j] = b_0 \cdot (h_{n+1} \cdot inv_{n+1}[update\_nbr, .] - h_n \cdot inv_n[update\_nbr, .] \cdot) \cdot A_{21}[., j] \tag{5.90}$$

The operation of the program :

The transient analysis can roughly be divided in two main parts, the integration over time and the van de Panne algorithm. Furthermore the integration over time can be divided in an explicit- and an implicit integration step. Finally the implicit integration step can be divided in four parts, (1) the time step determination, (2, 3) depending on a change in the time step an update or no update for the system matrix and (4) an update for the leafcells which are influenced by the current leafcell.

For a good understanding of the program the relevant variables are explained :

cur_leaf :	current leaf that is handled by the integration rule.
next_event :	the time at which the next event of the leafcell occurs.
xbarbar_leafs :	list with leafcells having $\bar{\bar{x}} \neq 0$.
panne_event :	boolean denoting whether the current leafcell has a panne event or a dynamic event.
restart_integration :	boolean denoting whether the current leafcell is restarted with an explicit or implicit integration rule.
step_changed :	boolean denoting whether the current time step is equal to the previous time step.
update_nbr :	integer denoting how many rank 1 updates have to be performed.
rtvec, cvec :	sparse vectors containing the update for the jacobian and *LU* decomposition.
inv_n :	$(a_0 \cdot I - b_0 \cdot h_n \cdot A_{22})^{-1}$.
e_a, e_r :	absolute and relative error.

Below the program of the transient analysis is given, the function *van_de_panne(cur_leaf)* is elaborated on the next page, and after that the leafcell functions will be explained :

```
transient_analysis()
{
step 1 :  for ( all leafcells ) I_INIT( leaf );
          t_n = next_event;
          while ( t_n ≤ stop_time ) {
step 2 :      if ( panne_event ) {
                  van_de_Panne( cur_leaf );
              else {
                  if ( restart_integration ) {
step 3 :              update_module_variable x_n = x_n-1 + h_n·x̄_n
                      I_GEN_BDOT( cur_leaf, b̄_n+1 );
                      sparse_solve( x̄̄_n+1 );
                      update_circuit_variables;
                      compose xbarbar_leafs;
                      update_with_sparse_vector ẋ;
                      for ( all leafs in xbarbar_leafs ) {
                          I_UPD_PWLVARS( leaf );
                          I_CALC_PWLDER( leaf );
                          if ( leaf != cur_leaf ) I_REC_EVENT( leaf );
                      }
                  }
step 4 :          update_module_variables x_n = x_n-1 + h_n·x̄_n;
                  I_UPD_PWLVARS( cur_leaf );
                  I_STEPSIZE( cur_leaf, step_changed );
                  if ( step_changed ) {
step 5 :              for ( update_nbr-- ) {
                          I_RANK1_UPD( cur_leaf, rtvec, cvec );
                          update_jacobian( rtvec, cvec );
                          ludec_upd( rtvec, cvec);
                      }
                      I_GEN_BDOT( cur_leaf, b̄_n+1 );
                      recompute_xbar( x̄̄_n+1 );
                  }
                  else {
step 6 :              I_GEN_BDOT( cur_leaf, b̄_n+1);
                      sparse_solve( x̄̄_n+1 );
                  }
step 7 :          update_circuit_variables x_n = x_n-1 + h_n·x̄_n;
                  compose xbarbar_leafs;
                  update_with_sparse_vector x̄_n+1;
                  for ( all leafs in xbarbar_leafs ) {
                      if ( leaf != cur_leaf ) {
                          I_UPD_PWLVARS( leaf );
                          I_CALC_PWLDER( leaf );
                          I_REC_EVENT( leaf );
                      }
                      else {
                          I_CALC_PWLDER( cur_leaf );
                          I_EVENT( cur_leaf );
                      }
                  }
              }
              output_signals;
              t_n = next_event;
}         }
```

The program

```
van_de_panne( cur_leaf )
{
        I_UPD_PWLVARS( cur_leaf );
        I_INIT_PANNE( cur_leaf );
        PUSH( cur_leaf, lambda, down );
        PUSH( cur_leaf, row, up );
        start van de Panne at step 3;
        END_PANNE( cur_leaf );
        update_circuit_variables x_n = x_n-1 + h_n·x̄_n;
        compose xbarbar_leafs;
        update_with_sparse_vector x̄_n+1;
        for ( all leafs in xbarbar_leafs ) {
            if ( leaf != cur_leaf ) {
                I_UPD_PWLVARS( leaf );
                I_CALC_PWLDER( leaf );
                I_REC_EVENT( leaf );
            }
            else {
                I_CALC_PWLDER( cur_leaf );
                I_EVENT( cur_leaf );
            }
        }
}
```

The leafcell functions :

I_INIT(leaf): initialise the leafcell.
if no u variables then next_event = stop_time.
else
$h_n = 0$
next_event = 0

I_GEN_BDOT(leaf, $\bar{b}_{n+1}$): $\dot{u}_{n-1} = \dot{u}_n$

$$\dot{u}_n = A_{21}\cdot x_n + A_{22}\cdot u_n + b_2$$

$$\bar{b}_{n+1} = A_{12}\cdot inv_{n+1}\cdot \left(-(a_0 + a_1)\cdot\frac{1}{h_{n+1}}\cdot u_{n-1} - a_2\cdot\frac{1}{h_{n+1}}\cdot u_{n-2} + (b_0 + b_1)\cdot\dot{u}_{n-1} + b_2\cdot\dot{u}_{n-2} \right)$$

I_UPD_PWLVARS(leaf): $u_n = u_{n-1} + h_n\cdot\bar{u}_n$
$v_n = v_{n-1} + h_n\cdot\bar{v}_n$

I_CALC_PWLDER(leaf): $\bar{u}_{n+1}$ = see equation 5.76
$\bar{v}_{n+1} = A_{31}\cdot\bar{x}_{n+1} + A_{32}\cdot\bar{u}_{n+1}$

I_REC_EVENT(leaf): determine dynamic step :

$$\bar{\dot{u}}_{n+1} = A_{21}\cdot\bar{x}_{n+1} + A_{22}\cdot\bar{u}_{n+1}$$

$$dynamic\_step = \frac{\max|\,\bar{u}_n\,|}{\max|\,\bar{\dot{u}}_{n+1}\,|}\cdot h_{n+1}$$

determine panne step :

$$panne\_step = min\left(\frac{-v_n}{\bar{v}_{n+1}} : \bar{v}_n<0 \right)$$

$$next\_event = t_n + \min(dynamic\_step, panne\_step)$$

I_STEPSIZE(leaf, step_changed): $\dot{\bar{u}}_n = A_{21} \cdot \bar{x}_n + A_{22} \cdot \bar{u}_n$

$$h_{n+1} = \left(\frac{e_a + \max(|u_n|) \cdot e_r}{\max(|\dot{\bar{u}}_n|)} \right)^{1/2}$$

return step_changed

I_RANK1_UPD(leaf, rtvec, cvec): $cvec[i] = A_{12}[i, update\_nbr]$

$rtvec[j] = b_0 \cdot (h_{n+1} \cdot inv_{n+1}[update\_nbr, .] - h_n \cdot inv_n[update\_nbr, .]) \cdot A_{21}[., j]$

I_EVENT(leaf): $\dot{\bar{u}}_{n+1} = A_{21} \cdot \bar{x}_{n+1} + A_{22} \cdot \bar{u}_{n+1}$

determine panne step:

$$panne\_step = min\left(\frac{-v_n}{\dot{v}_{n+1}} : \dot{v}_n < 0 \right)$$

$next\_event = t_n + \min(h_{n+1}, panne\_step)$

I_INIT_PANNE(leaf): $a_1 = 0;$

$a_2 = -ub;$

$a_3 = -vb;$

END_PANNE(leaf): $a_1 = a_2 = a_3 = 0;$

5.8 Examples

In this section some examples will be treated, to show the properties and operation of the simulator. The first example can be found in Ruehli [18], it is used to denote the differences in the damping of the *MS* methods. The second example is a phase locked loop on digital level. The third and fourth examples are a seven stage ring oscillator on circuit level and on digital level respectively. On the following five pages the the circuits and the relevant signals will be pictured. Thereafter the statistics generated by the simulator will be shown and the results of the different circuits will be compared.

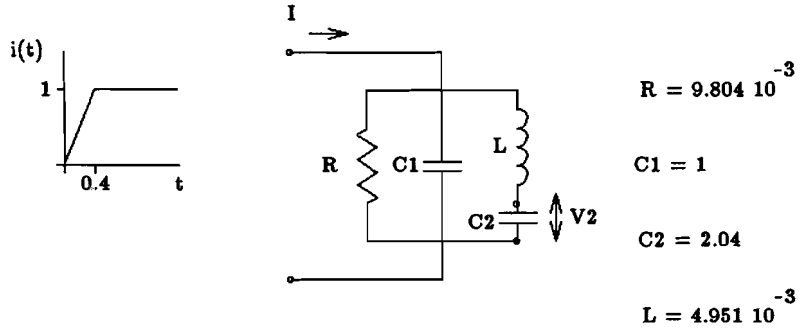


Figure 5.5. The Landman circuit

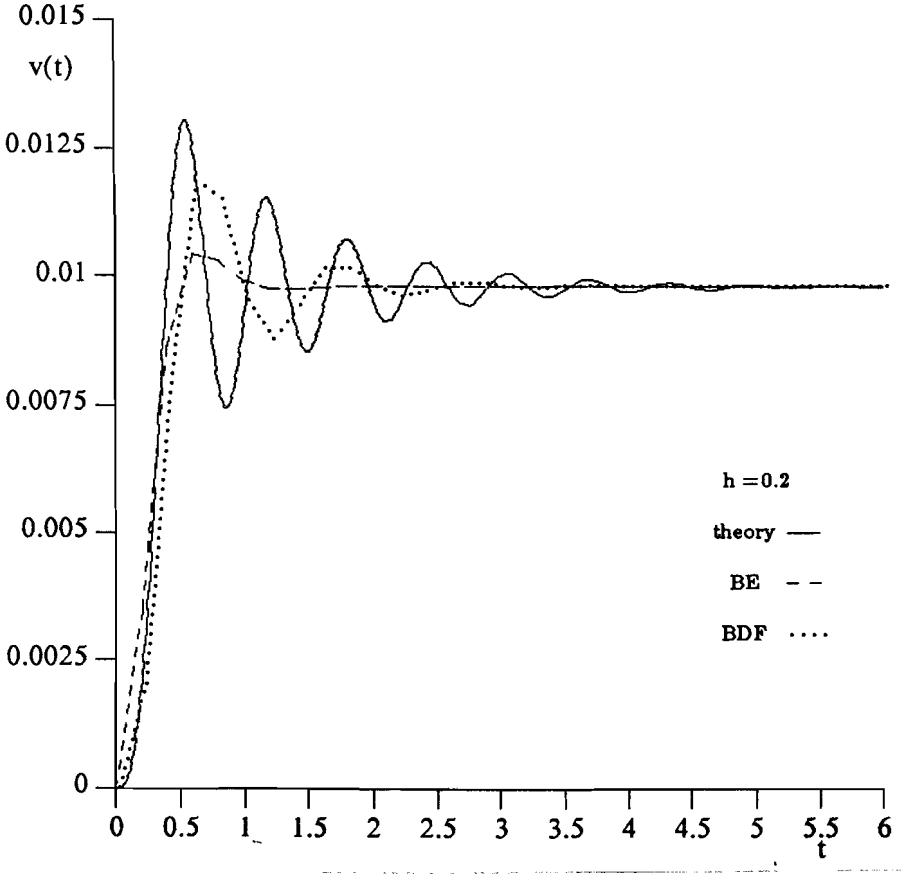


Figure 5.6. The response of the Landman circuit with different integration rules

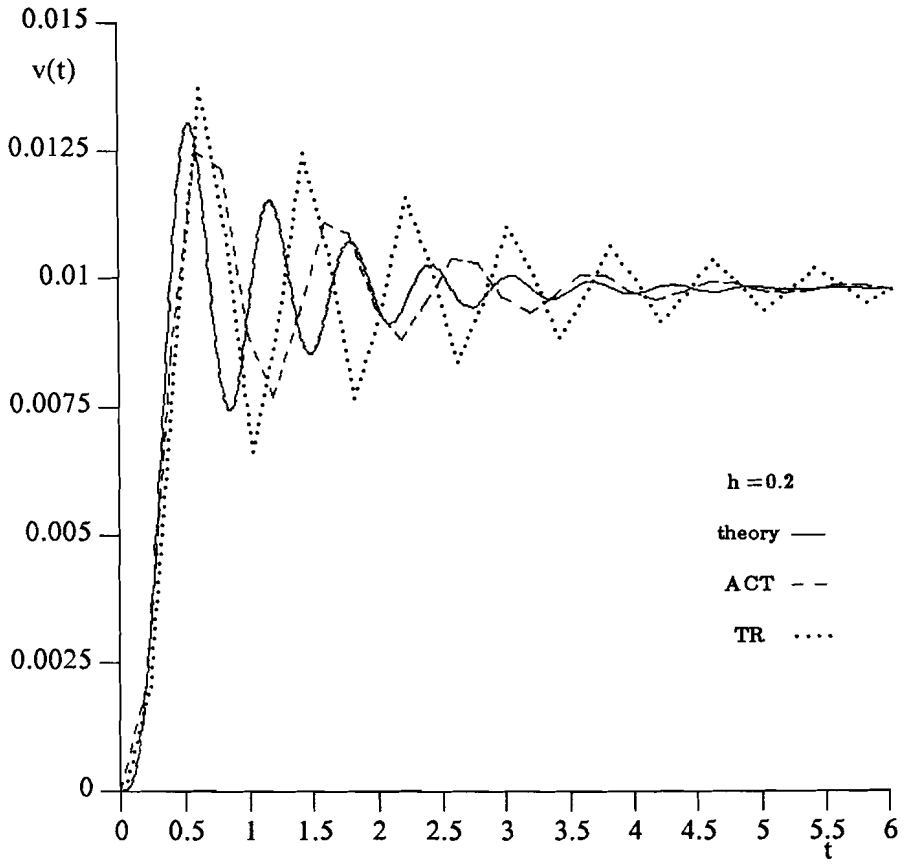


Figure 5.7. The response of the Landman circuit for different integration rules

The Landman circuit is derived from Ruehli [19], as an example for the damping of the different integration rules. The Landman circuit exhibits equations which are stiff with an eigenvalue ratio of 100. The large time step chosen in the example give a clear view of the numerical damping of the integration rules.

For the trapezium rule, the propagation of the errors is clearly visible, while the overdamping of the *BDF*2 method and the *BE* method is also clearly visible. Figure *5.7* shows the good response for the *ACT*2 method and shows the desirable properties for practical computation with an appropriate choice of the time step.

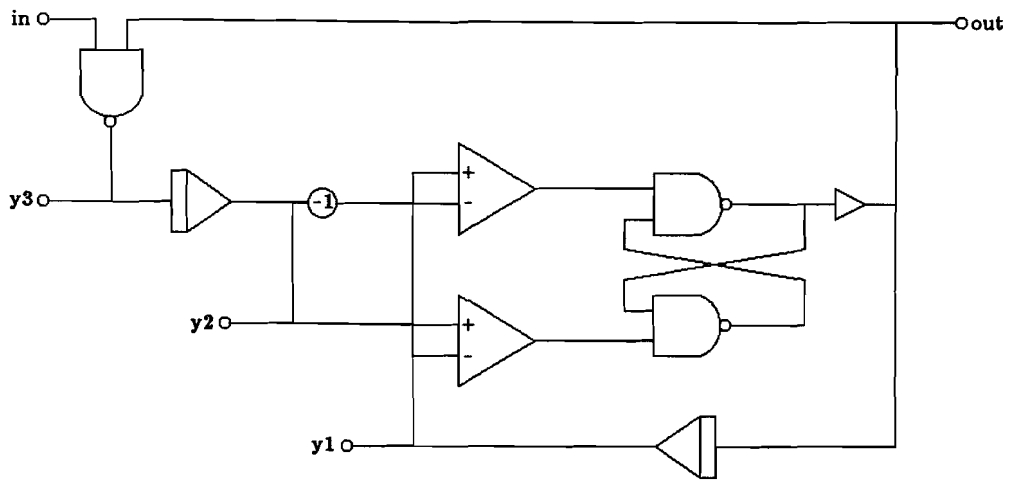


Figure 5.8. The phase locked loop

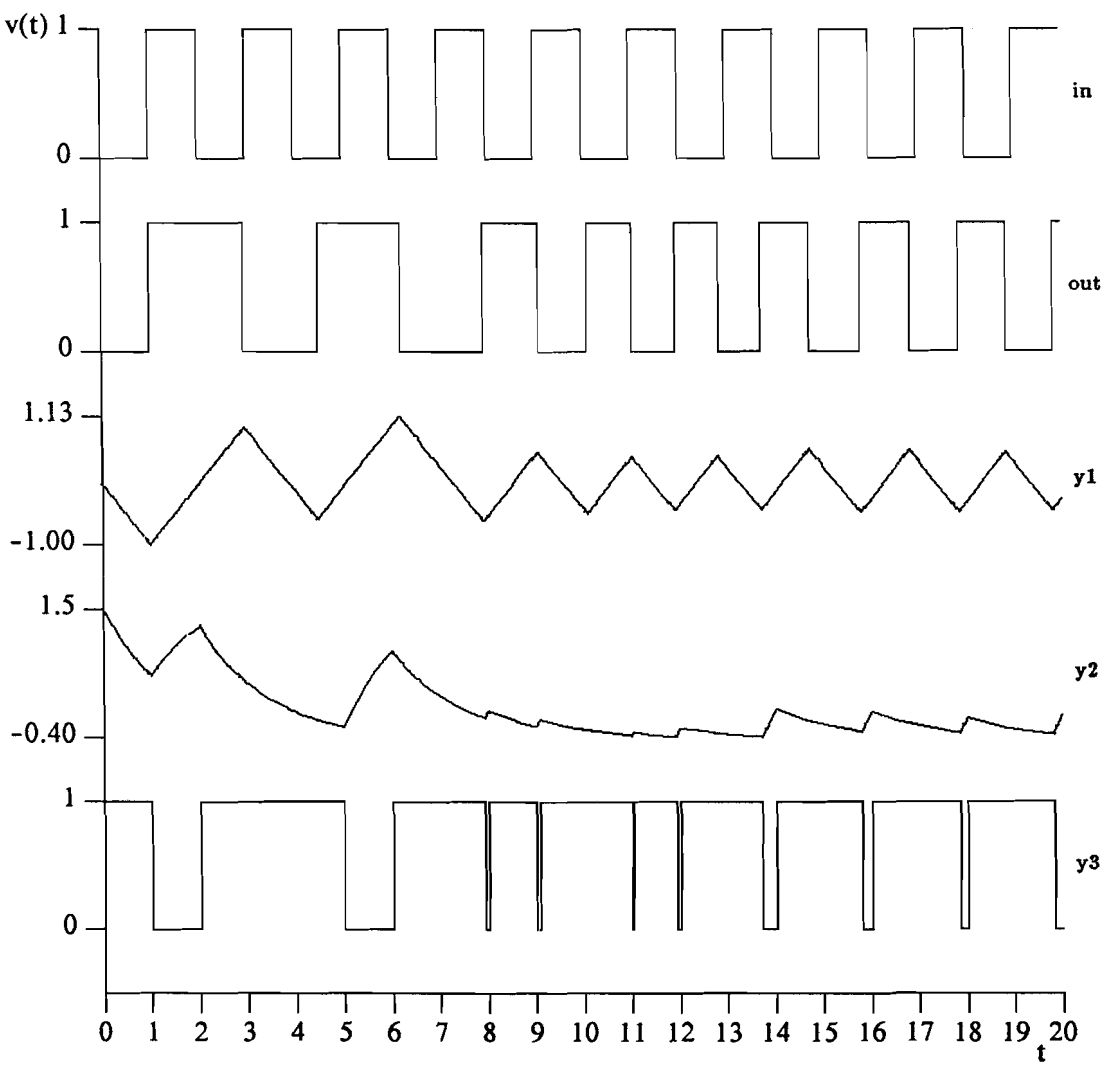


Figure 5.9. The response of the phase locked loop

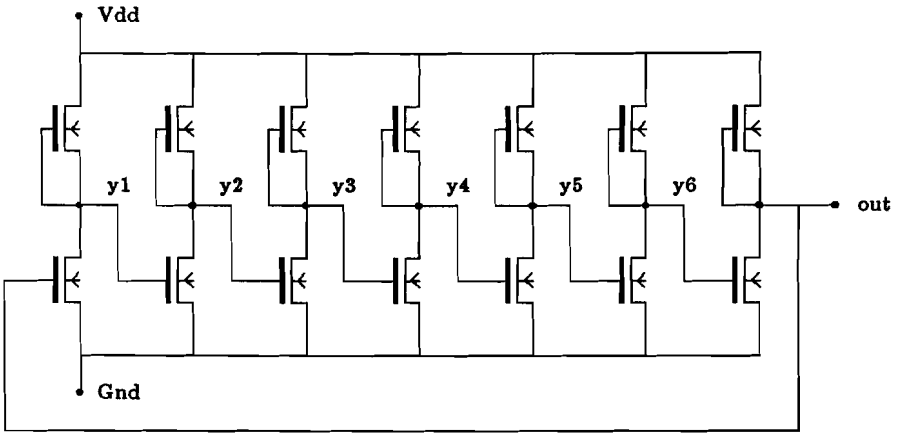


Figure 5.10. The seven stage ring oscillator, analog

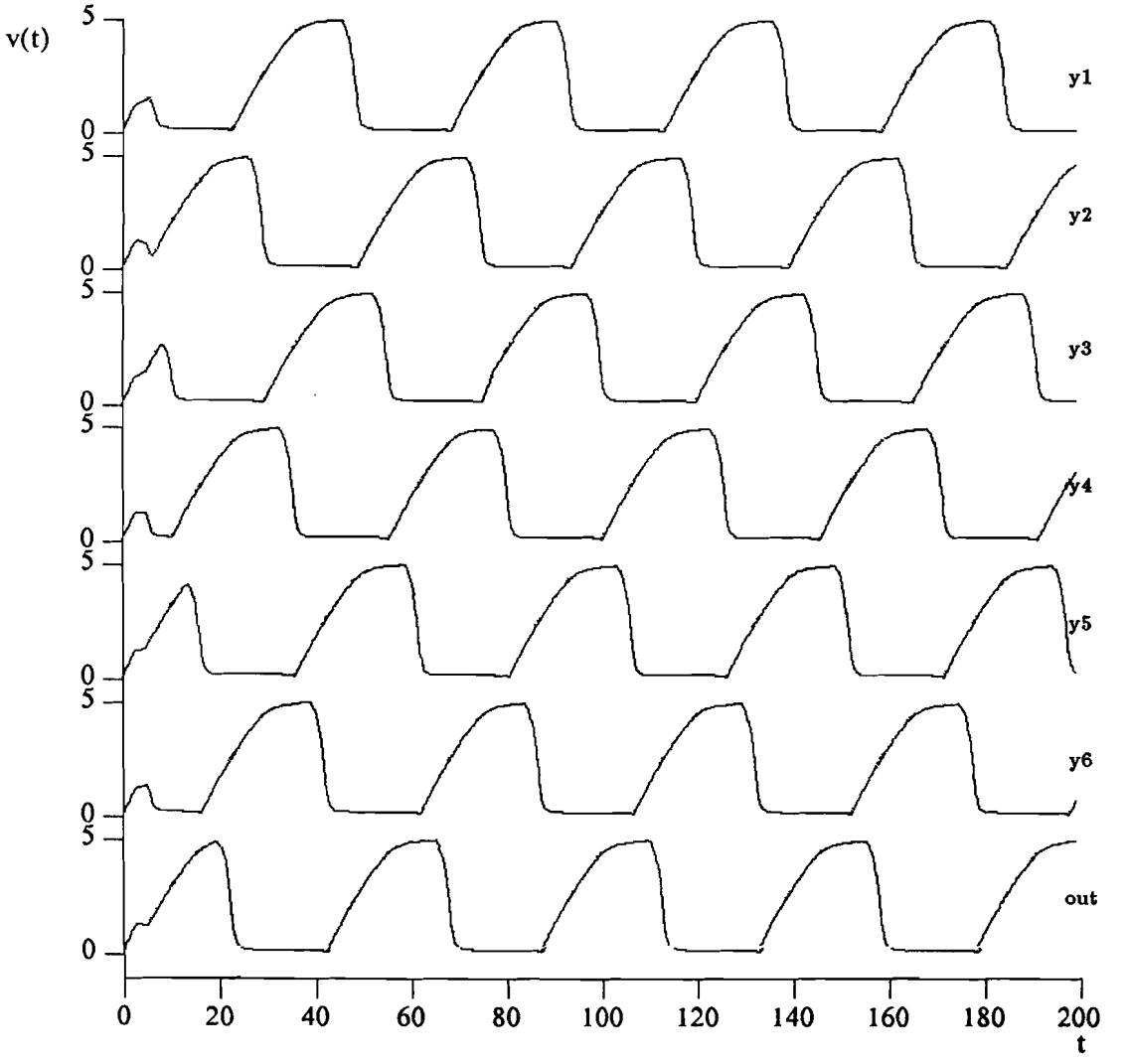


Figure 5.11. The response of the seven stage ring oscillator

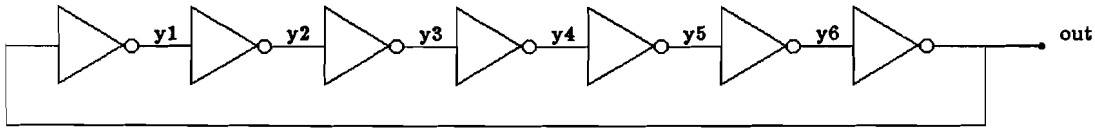


Figure 5.12. The seven stage ring oscillator, digital

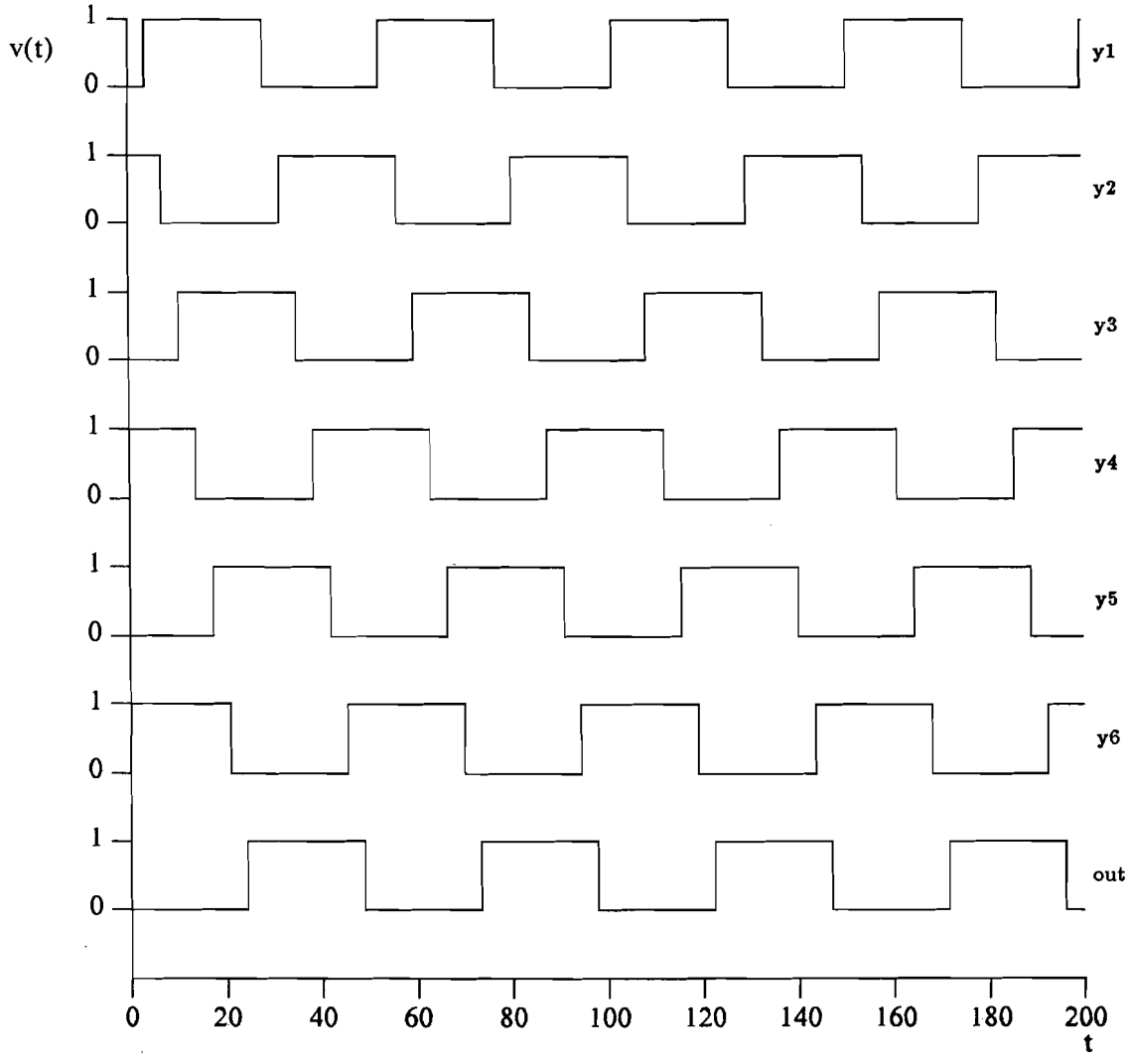


Figure 5.13. The response of the seven stage ring oscillator

STATISTICS					
properties	**1**	**2**	**3**	**4**	**5**
matrix information					
Number of modules	5	683	15	15	10
Number of leafcells	4	602	14	14	9
Matrix size	10	618	65	14	10
Number of nonzero matrix elements	27	2271	468	29	36
Percentage of nonzero matrix elements	27.0 %	0.6 %	11.1 %	14.8 %	36.0 %
Number of fill-ins generated	9	1653	228	15	25
lu decomposition					
Number of full lu decompositions	1	1	1	1	1
Number of lu-crout decompositions	0	1160	192	114	34
Number of dyadic updates	257	2634	1732	128	375
Average update vector size	1.2	1.0	1.0	0.5	1.1
Average number of changed matrix elements	13.2	13.9	165.5	1.3	10.3
Average percentage of changed elements	49.0 %	0.6 %	35.4 %	4.6 %	28.5 %
forward backward substitution					
Number of sparse forw.-backw. substitutions	3350	6112	4593	364	848
Number of multiplications / fb-subst.	22.2	19.7	278.7	1.9	3.3
changes xdot					
Average number of changes in xdot	9.0	0.0	58.8	0.0	0.7
Average number of leafcells reached	3.0	0.0	12.6	0.0	1.0
pivots					
Number of pivots done for DC solution	0	340	21	3	6
Number of pivots during transient analysis	0	2204	192	118	224
events and timing					
Number of pwl events	0	580	192	57	136
Number of dynamic events	3602	707	5863	64	322
simulation CPU time (seconds)	68	10731	1504	18	13

1 = Landman

2 = greatest common diviser

3 = analog seven stage ring oscillator

4 = digital seven stage ring oscillator

5 = phase locked loop

At this moment the statistics are in a preliminary stage. Not all the features of the simulator can be derived from the table. As far as the statistics are sufficient the following properties can be derived :

- The dynamic events generated by the simulator differ from circuit to circuit. As expected, a large number of events is generated for the Landman circuit and the analog seven stage ring oscillator, while for the other circuits which are digital the generated events are considerably less.
- The full *LU* decomposition is only performed once at the beginning of the simulation. Thereafter only partial *LU* decompositions are performed, by the *LU* decomposition according to Crout and the rank 1 update (dyadic update). It is noticed that the average percentage of changed matrix elements is a measure for the computation time of the *LU* decomposition update. With respect to the sparse forward backward substitution we notice that the number of multiplications is a measure for the effort

the simulator has to perform to find a solution for the system matrix every time a forward backward substitution is performed. The greatest numbers are found for the Landman circuit, the analog oscillator and the phase locked loop. This is related with the large number of non zero matrix elements for these circuits.

- It is noticed that the van de Panne algorithm finds fast a DC solution, only a small number of pivots has to be performed. The number of pivots done during the transient analysis per pwl event are also small. Especial attention may be drawn to the analog seven stage ring oscillator, which shows that every pwl event is solved by exactly one pivot.
- The required CPU time needed for the analog oscillator and the GCD circuit are somewhat disappointing. Leaving out the checks which are performed during simulation will certainly speed up simulation. Beside of that improvements can be made for some updates which are performed during transient analysis.

6. Input nodes

6.1 Introduction

For the construction of the circuit, every building block is transformed to a pwl model. For the input nodes it is possible to create models that perform the function chosen by the user. The use of pwl models for input nodes implies that the user has to change the circuit structure with the aid of the schematic entry program, every time he wants to test the circuit in a different way. By creating the possibility to assign functions to an input node in the task file, the user can change input functions fast and efficient without affecting the circuit.

6.2 The functions

The input functions that are implemented have been derived from SPICE. Below the functions are listed, for every function the values, the example for the task file and use by the transient analysis and the van de Panne algorithm are denoted :

constant value : *function*,
assigns a constant value to a node.

example,
/in = 5[V];

use,
not evaluated by van de Panne and transient analysis.

sine wave : *function*,
$f(x) = VO + VA \cdot e^{-(t-TD)\cdot THETA} \cdot \sin(2 \cdot \pi \cdot FREQ \cdot (t+TD))$

example,
/in = *SINE*(*VA*, *VO*, *FREQ*, *TD*, *THETA*)[*A*];

where : *VA* = amplitude
VO = offset
FREQ = frequency
TD = delay time
THETA = damping factor

use,
only evaluated by transient analysis.

pulse function : *function*,
multi function for clock or pulse generation, see figure 6.1.

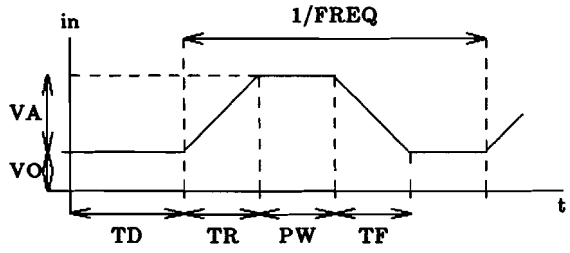


Figure 6.1. The pulse function

example,
/in = *PULSE*(*VA*, *VO*, *FREQ*, *TD*, *TR*, *PW*, *TF*)[*X*];

where : *VA* = amplitude
VO = offset
FREQ = frequency
TD = delay time
TR = rise time
PW = pulse width
TF = fall time

use,
this function is evaluated by the van de Panne algorithm as well as the transient analysis.

input file : *function*,
this function is described in a file containing user defined time points and values, which are interconnected by line straight segments.

example,
/in = /input_filename[V];
every line of the file contains subsequently one time point and corresponding value.

use,
this function is evaluated by the van de Panne algorithm as well as the transient analysis.

6.3 Adjustments to the simulator

In this section the difference of the inputcells with the leafcells in regard to the transient analysis and the van de Panne will be treated. Because of the general construction of the simulator only leafcell functions will change, which will be called inputcell functions in the subsequent part of this section, see also section 4.3 and 5.7. The simulator does not know what type of leafcell it is dealing with, the leafcell could be a pwl model or an inputcell. However all the information about inputcells is known and the transient analysis and the van de Panne algorithm derive this information indirectly. Inputcells emulate pwl behaviour.

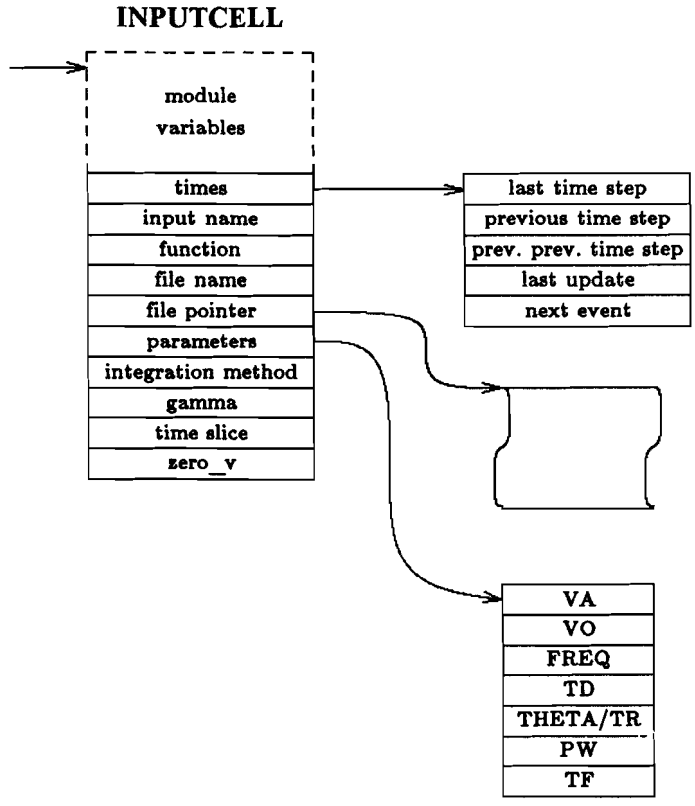


Figure 6.2. The data structure of the inputcell

The data structure for the inputcell is depicted in figure 6.2. From the figure it is seen that the inputcell has no pwl matrix and corresponding pwl variables, the input values are directly derived from the function that is applied. Because of this the update functions for the pwl model are not necessary for the inputcell.

The contribution to the system matrix of an inputcell consists of one entry. During simulation this element stays the same, only the source vector can change. So the rank 1 update in the transient analysis is not needed and therefor the variable step_changed is always false for inputcells, remind however this does not imply that the step size for an inputcell does not change.

In case of a dynamic event, the inputcell reaches the end of a time step interval, this implies that a new time step for the inputcell has to be calculated. After that a new source vector $\bar{b}$ is generated and a new $\bar{x}$ is determined.

In case of a panne event, the inputcell has reached a discontinuity and will be handled by the van de Panne algorithm. For the van de Panne algorithm the inputcell is extended with a variable "gamma". Gamma is the maximum allowed theta that the inputcell communicates to the van de Panne algorithm.

A detailed description of some inputcell operations will be shown below, subsequently the sine wave, the pulse function and the input file will be dealt with :

1) the sine wave :

transient analysis,

I_INP_STEPSIZE : $h_{n+1} = \left(\frac{e_a + |x| \cdot e_r}{\dot{x}} \right)^{1/2}$

I_INP_GEN_BDOT : $\bar{b}_{n+1} = e^{(TD - t_{n+1}) \cdot THETA} \cdot \sin(2\pi \cdot FREQ \cdot (t_{n+1} + TD))$

$-e^{(TD - t_n) \cdot THETA} \cdot \sin(2\pi \cdot FREQ \cdot (t_n + TD))$

I_INP_EVENT : $next\_event = t_n + h_{n+1}$

2)the pulse function :

transient analysis,

I_INP_STEPSIZE : h_{n+1} = the time duration of the next pulse segment
e.g. suppose t_n is at the boundary between TR and PW
then h_{n+1} = PW etc.

I_INP_GEN_BDOT : Only if the current pulse segment is TR or TF, $\bar{b}_{n+1}$ will have a value.
if TR $\bar{b}_{n+1} = VA/TR$
if TF $\bar{b}_{n+1} = -VA/TF$

I_INP_EVENT : next_event = t_n + (next pulse segment)
determine panne event
if pulse segment after next_event is TR or TF and TR = 0 or TF = 0, panne_event is true.

van de Panne algorithm,

INP_GEN_SOURCE : if active variable is lambda
if TR $\bar{b} = VO + VA - x$
if TF $\bar{b} = VO - x$
else
$\bar{b} = 0$

INP_SIGN :
```
if ( gamma = 0 )
   sign = 1
else
   sign = 0
```

INP_CALC_THETA : θ = gamma

INP_CALC_STEP :
```
if ( gamma == theta_min )
   gamma = 0
   zero_i = 1
else
   zero_i = 0
```

3) the input file :

transient analysis,

I_INP_STEPSIZE : $h_{n+1} = t_{n+1} - t_n$

I_INP_GEN_BDOT : $\bar{b}_{n+1} = (value(t_{n+1}) - value(t_n)/h_{n+1}$

I_INP_EVENT : next_event = $t_n + h_{n+1}$
read input file (t_{n+1}, $value(t_{n+1})$)
if ($t_n == t_{n+1}$) then panne_event is true.

Input nodes

van de Panne algorithm,

INP_GEN_SOURCE : if active variable is lambda
$\bar{b} = value(t_{n+1}) - x$
else
$\bar{b} = 0$

INP_SIGN : see pulse function.

INP_CALC_THETA : see pulse function.

INP_CALC_STEP : see pulse function.

7. Conclusions

At the moment the simulator is ready, however in a preliminary state. The size of the simulator as it is implemented at the moment is about 12000 lines and the compiled version takes about 250 Kbytes. With respect to the operation of the simulator the following properties are observed :

- The circuit is stored in a hierarchical way this storage is used to form a bordered block matrix structure, a structure which is maintained during simulation. The sparse matrix structure, the hierarchical storage of the circuit and the bordered block matrix structure reduce automatically the number of operations involved with the *LU* decomposition and the forward backward substitution.
- The van de Panne algorithm, which computes the solution of the linear complementarity problem, shows fast convergence properties.
- The van de Panne algorithm and the transient analysis have a general framework and changes e.g. in the integration rule, are executed fast and concern only small sub-functions.
- The simulator is event driven, leading automatically to simulation of only those parts of a circuit which possibly can change. This leads to considerably savings of computation time during simulation
- The choice of the integration method, backward euler, trapezium rule, two step backward differentiation method or two step *A*-contractive method, is difficult to make and depends on the application. Theoretical considerations, with respect to constant time step, give preference to the trapezium rule. The properties of the trapezium rule are : simple to apply, stability in the entire left half complex plane, second order, a small error constant and no damping. However do we apply a variable step size, as is implemented for the simulator, then also the two steps *A*-contractive method and the two steps backward differentiation method, which are more complex then the trapezium rule, show good results.
- The application of the input nodes yields a nice feature for the users. The implementation is small and straightforward, only small sub-functions are involved. A basic set of functions is available and if desired new functions can be added fast.

During simulation a lot of time consuming tests are performed, e.g. every time the *LU* decomposition changes a check for the correctness of the new *LU* decomposition is performed. It is certain that leaving out these tests, simulation will speed up considerably. Some of the sub-functions can be made faster and smarter, as there are the time step control and the application of a pivot strategy. With respect to the data structure some improvements can be added, e.g. the data structure of the input nodes can be reduced or storing only the *LU* decomposition in the sparse matrix structure will reduce storage considerably.

Concerning the integration rules, more research has to be carried out after the properties of the implemented rules. Beside of the implemented methods, other ones could be used, like for example the one leg implementation of the *ACT* methods as proposed by Ruehli [18]. Research has to be carried out after a strategy that could be used for the implementation of an automatically change of integration rule, as proposed in Sangiovanni [20].

As is noticed the simulator is working and yields good results. Although some improvements have to be carried out, the simulator can handle all feasible circuits.

8. References

[1] Bennett J.M.
Triangular factors of modified matrices,
Numerische Mathematic, Vol 7, 1974, page 927 - 937.

[2] Bokhoven W.M.G. van
Linear implicit differentiation formulas of variable step and order,
IEEE Trans. Circuits & Systems, Vol CAS-22, 1975, page 108-115.

[3] Bokhoven W.M.G. van
Piecewise-linear modelling and analysis,
Deventer: Kluwer, 1981,
Ph.D. Thesis Eindhoven University of Technology, 1981.

[4] Brayton R., el all.
A new efficient algorithm for solving differential algebraic systems using implicit backward difference formulas
Proc. IEEE, Vol 60, No 1, page 98-108 Jan.

[5] Brayton R., Conley C.
Some results on the stability of the backward differentiation methods with nonuniform time steps,
Topics in Numerical Analysis, Proc. Royal Irish Acad. Conf.,
Academic Press, NY, 1972, page 13-33.

[6] Brayton R., Tong C.
Stability of dynamical systems : A-contractive approach,
IEEE Trans. Circuits and Systems, CAS-26 (1979), page 224-234

[7] Dahlquist G., Liniger W., Nevanlinna O.
Stability of two step methods for variable integration steps,
SIAM J. Numer. Anal., Vol. 20, pp. 1071-1085, 1983.

[8] Eijndhoven J.T.J. van
to appear :
Piecewise linear analysis
Analog Circuits : Computer Aided Analysis and Diagnosis,
T. Ozawa (ed.), Moral dekker inc., New York.

[9] Eijndhoven J.T.J. van
A piecewise linear simulator for large scale integrated circuits,
Helmond: Wibro, 1984,
Ph.D. Thesis, Eindhoven University of Technology, 1984.

[10] Eijndhoven J.T.J. van, Jess J.A.G.
The solution of large piecewise linear systems,
Proc. ISCAS 1982 page 597-600.

[11] Eijndhoven J.T.J. van, Stiphout M.T. van
to appear,
Latency exploitation in circuit simulation by sparse matrix techniques,
Proc. ISCAS 1988.

[12] Gear C.W.
The automatic integration of ordinary differential equations,

Comm. ACM 14, 1971, page 176-179.

[13] Lemke C.E.
On Complementary Pivot Theory,
Mathematics of the decision sciences. Part 1, Proc 5th.
Summer Seminar, Stanford, Calif., 10July - 11 Aug 1967.

[14] Liniger W.
The A-contractive second-order multistep formulas with variable integration step,
SIAM Jr. Numer. Anal., Vol 20, page 1231-1238, 1983.

[15] Nevanlinna O., Liniger W.
Contractive methods for stiff differential equations,
Part I, BIT, vol 18, page 457-474, 1978,
Part II, BIT, vol 19, page 53-72, 1979.

[16] Odeh F., Liniger W.
On variable step and non-linear A-stability,
Proc. IEEE Intl Conf. Circuits and Computers,
Port Chester, NY, October 1-3, 1980.

[17] Panne C. van de
A complementary variant of Lemke's method for the linear complementary problem,
North Holland publishing Company,
Mathematical Programming 7, 1974, page 283-310.

[18] Ruehli A.
Circuit analysis simulation and design,
IBM Corporation, T.J. Watson Research Center,
Yorktown Heights, NY, USA, 1986.

[19] Ruehli A., Brennan P., Liniger W.
Control of numerical stability and damping in oscillatory differential equations,
Proc. of the IEEE Intl. Conf. Circuits and Computers,
G.Rabat Ed., Vol 1, page 111-114, 1980.

[20] Sangiovanni-Vincentelli A.L.
Circuit simulation,
IBM T.J. Watson research center,
Yorktown Heights, HY 10598, USA.

[21] Skelboe S.
Stability properties of implicit linear multirate formulas,
Proceedings ECTTD, 1987.

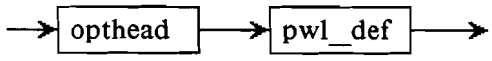
[22] Vlach M.
LU decomposition forward backward substitution of recursive bordered block diagonal matrices,
Proc. 16th Int. Symp. on circuits and Systems,
Newport Beach, Calif, 2 - 4 May 1983.

[23] Wilkinson J.H.
Rounding errors in algebraic processes,
Prentice Hall 1963.

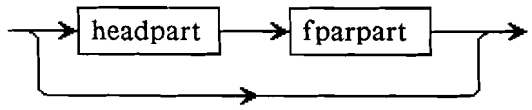
[24] Wilkinson J.H.
The algebraic eigenvalue problem,
Clarendon Press. Oxford 1965.

Appendix 1 : Leafcell description language (NDML subset)

SSS :

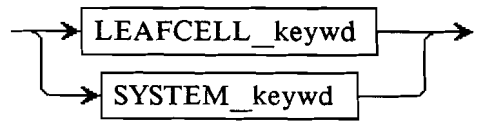


opthead :

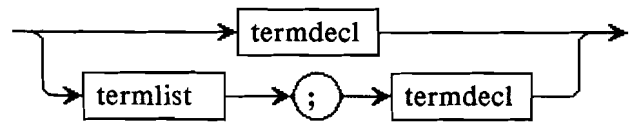


headpart :

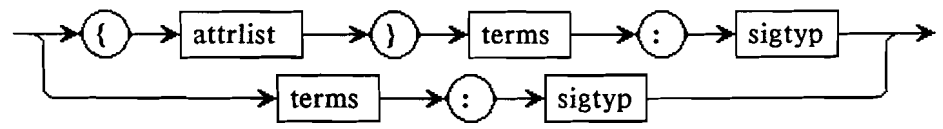
leafkey :



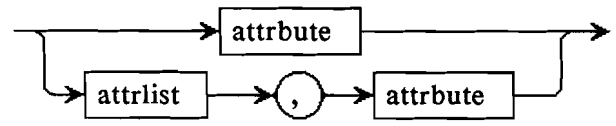
termlist :



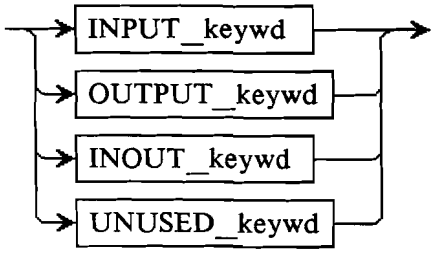
termdecl :



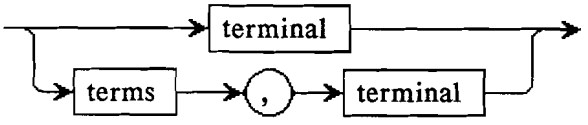
attrlist :



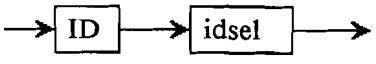
attrbute :



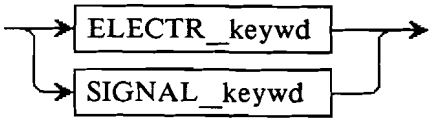
terms :



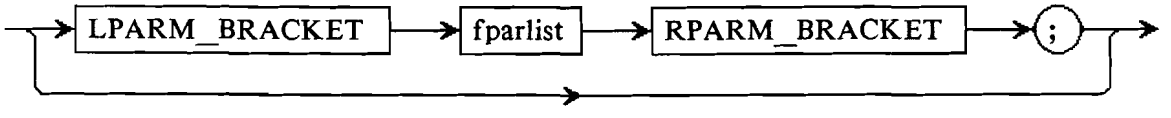
terminal :



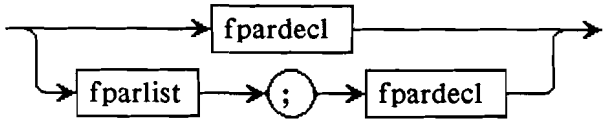
sigtyp :



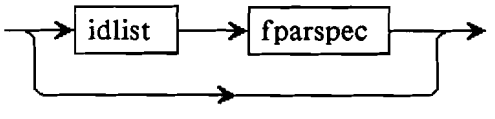
fparpart :



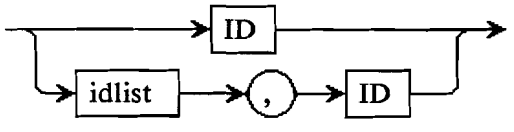
fparlist :



fpardecl :

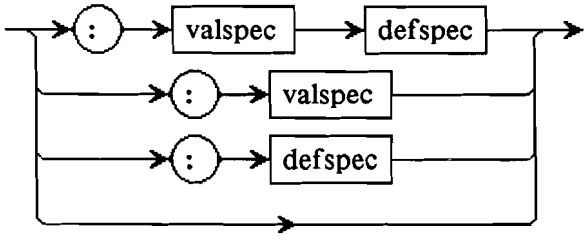


idlist :



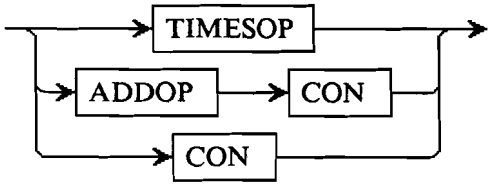
Appendix 1 : Leafcell description language (NDML subset)

fparspec :

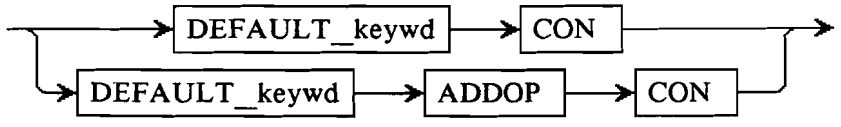


valspec :

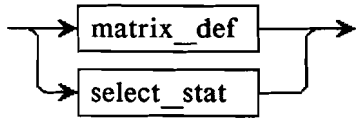
bound :



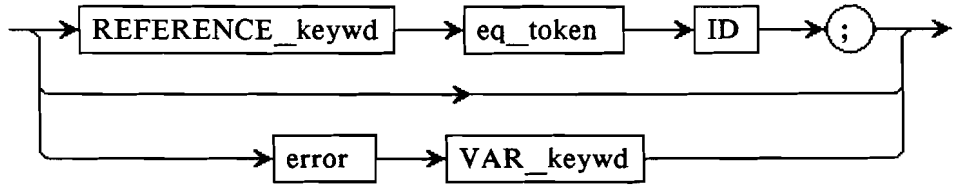
defspec :



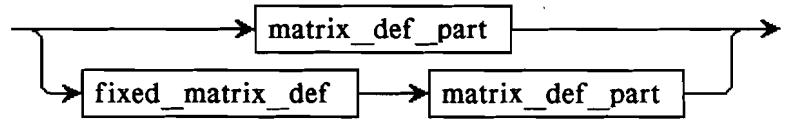
pwl_def :



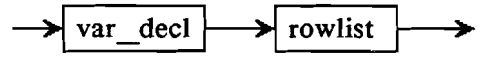
reference_def :



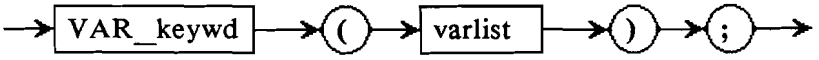
fixed_matrix_def :



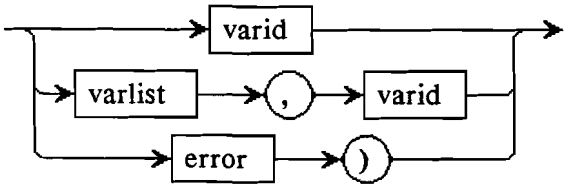
matrix_def_part :



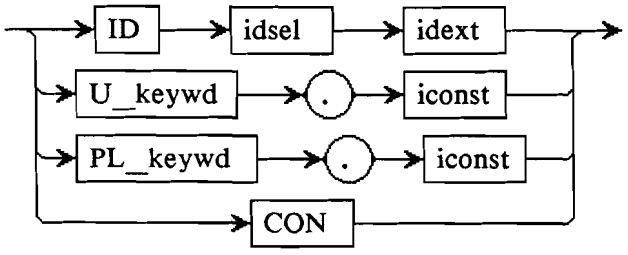
var_decl :



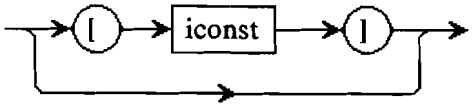
varlist :



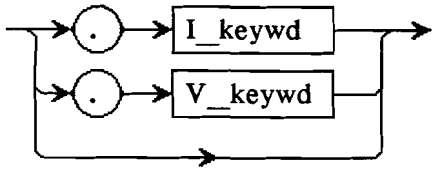
varid :



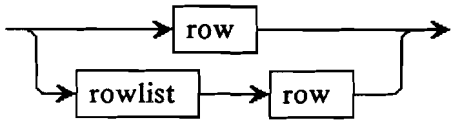
idsel :



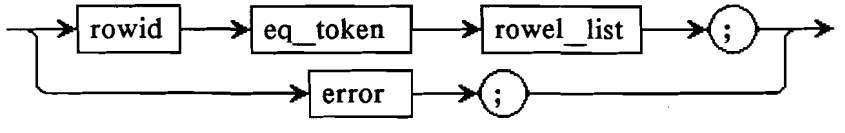
idext :



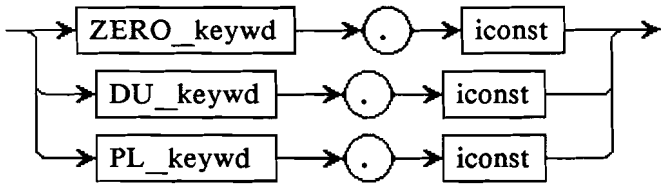
rowlist :



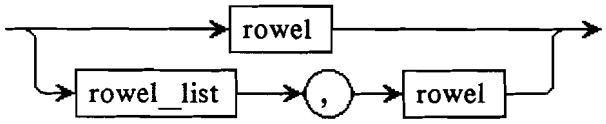
row :



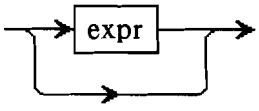
rowid :



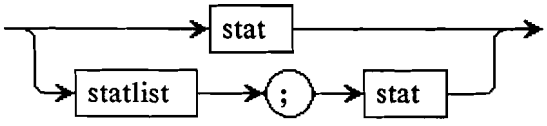
rowel_list :



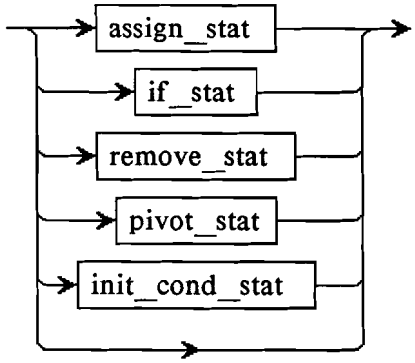
rowel :



statlist :

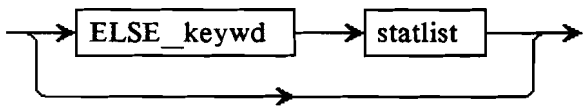


stat :

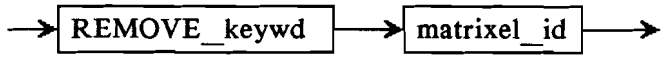


assign_stat :

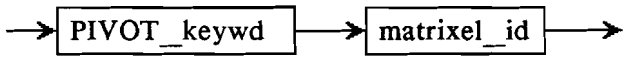
elsepart :



remove_stat :

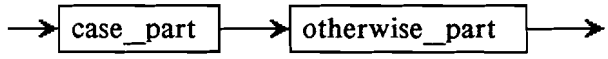


pivot_stat :

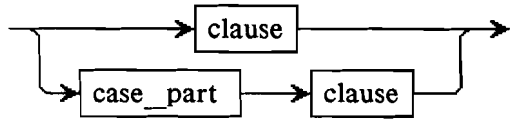


matrixel_id :

select_stat :

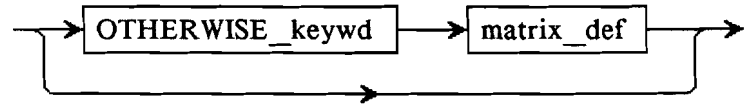


case_part :



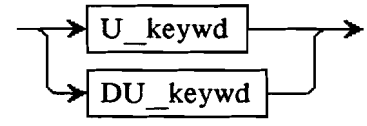
clause :

otherwise_part :



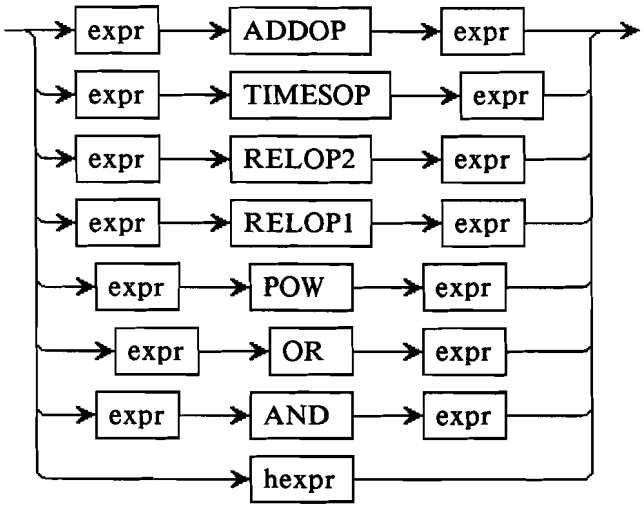
init_cond_stat :

init_var :

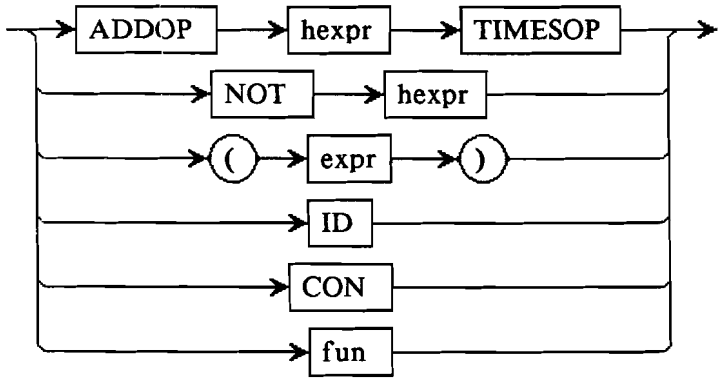


iconst :

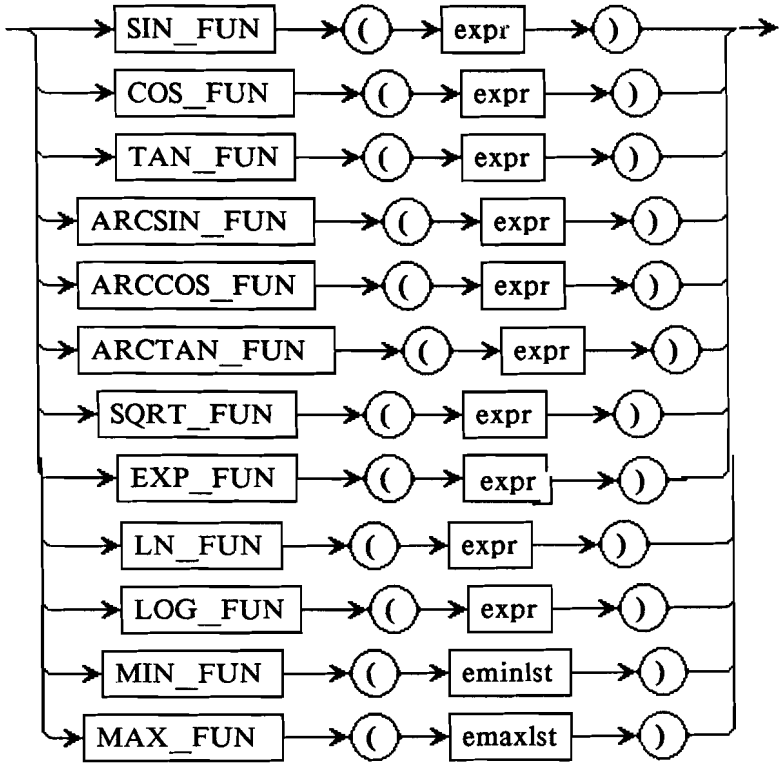
expr :



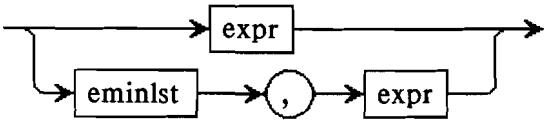
hexpr :



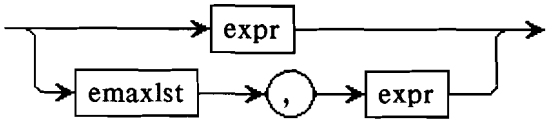
fun :



eminlst :



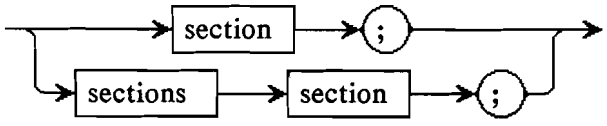
emaxlst :



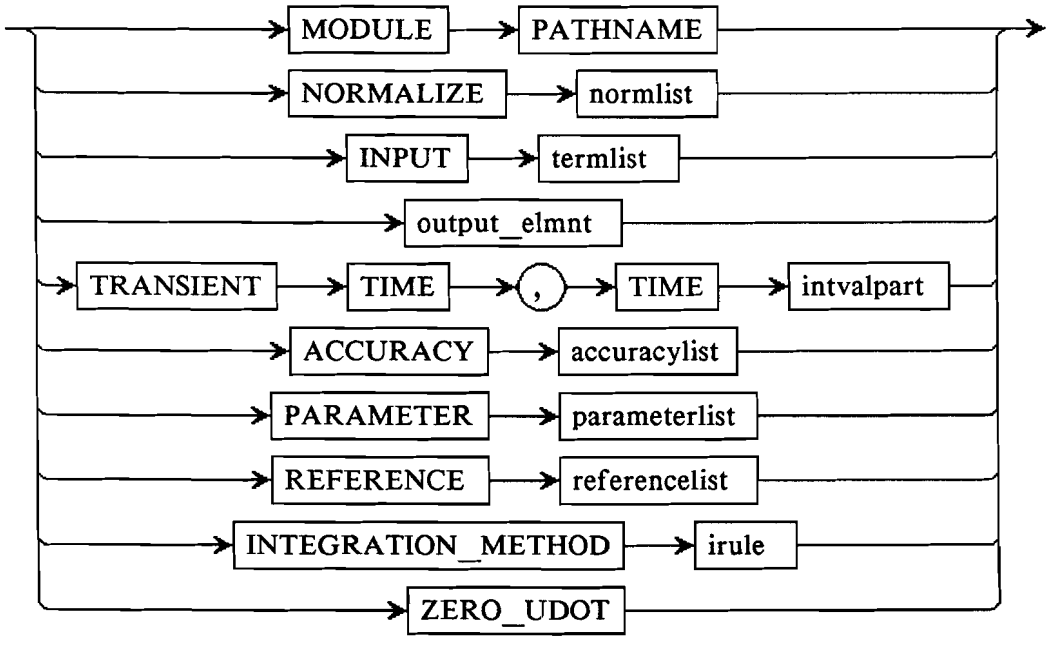
eq_token :

Appendix 2 : Simulation task description language

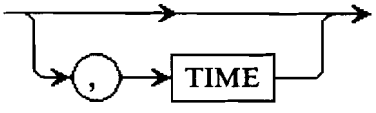
sections :



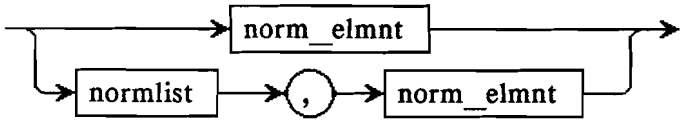
section :



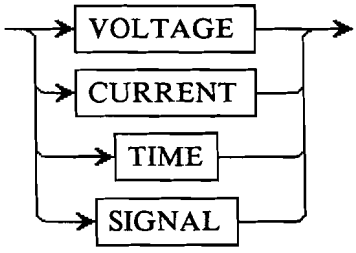
intvalpart :



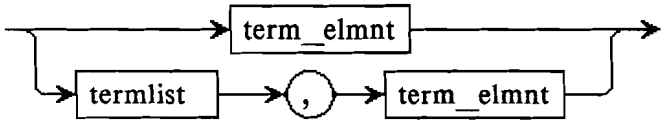
normlist :



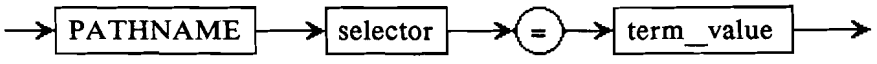
norm_elmnt :



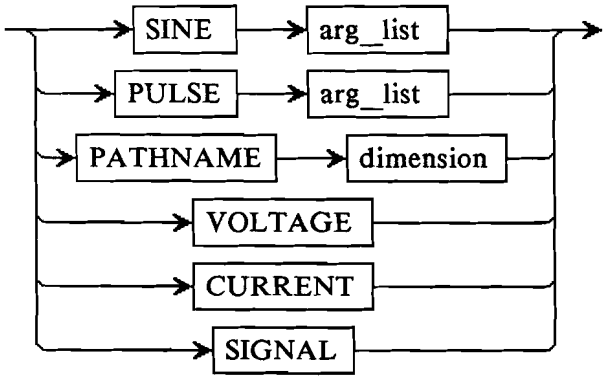
termlist :



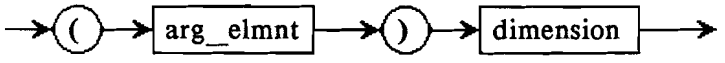
term_elmnt :



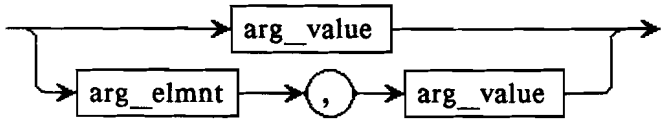
term_value :



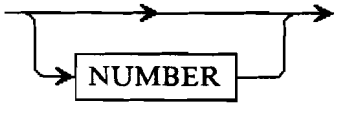
arg_list :



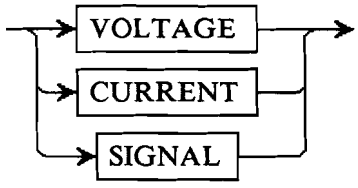
arg_elmnt :



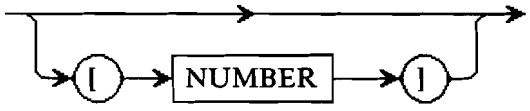
arg_value :



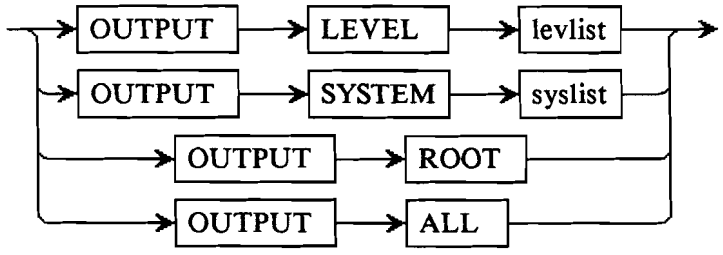
dimension :



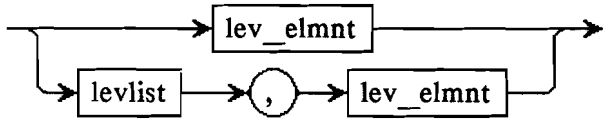
selector :



output_elmnt :

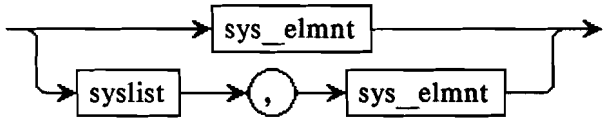


levlist :



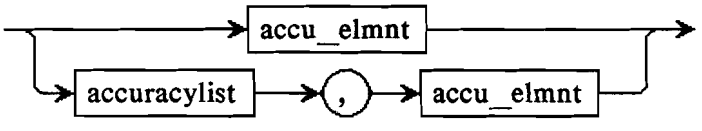
lev_elmnt :

syslist :

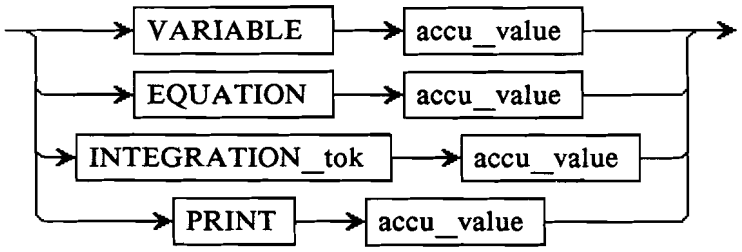


sys_elmnt :

accuracylist :

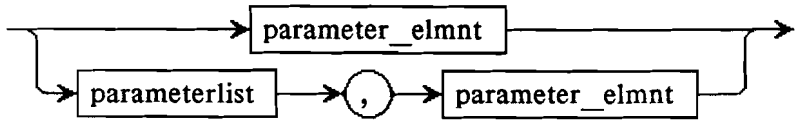


accu_elmnt :



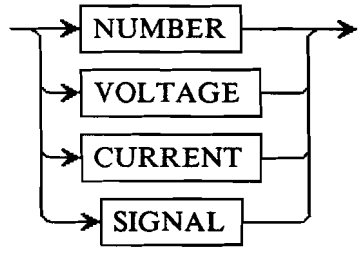
accu_value :

parameterlist :

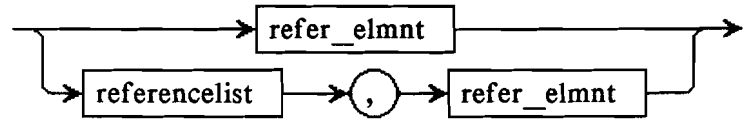


parameter_elmnt :

parameter_value :



referencelist :



refer_elmnt :

irule :