TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

Thesis

Symbolic Solution of Linear and Nonlinear Parameterized Systems of Equations

Eckhard Hennig

Thesis advisor: Dr.-Ing. Ralf Sommer

Braunschweig, August 1994

Statutory declaration

I hereby certify that I wrote this work myself and that I did not use any tools other than those specified therein.

Braunschweig, August 30, 1994

(Eckhard Hennig)

Translators notes

This is the translation of the German version of Prof. Eckhard Hennig's[1] dissertation of his Solver package documentation. We would like to thank Eckart Hennig for providing us with the original German TeX file and the accompanying pictures. Using this TeX source Wolfgang Lindner translated a first version using the `solver1-en.html` by Google Translate. He used TeXShop version 5.22 for macOS 14.2.1 (Sonoma). The flow diagrams were revised with SoftMaker TextMaker[2]. Dan Stanger revised the whole template to correct the grammar and wording errors and worked on updating the Maxima code.

We hope to provide some help to the user of the solver package. The source is to be found at `https://sourceforge.net/p/maxima/code/ci/master/tree/share/algebra/solver/`

Dr. Wolfgang Lindner and Dan Stanger
Leichlingen, Germany and Newton, Massachusetts, USA

January 2024

[1] Eckhard.Hennig@reutlingen-university.de, `www.reutlingen-university.de`
[2] There is a free version, see `https://www.freeoffice.com/de/features/freeoffice-textmaker`

Acknowledgments

Many persons and institutions contributed considerably to the success of this work. I would like to express my thanks to:

- Richard Petti and Jeffrey P. Golden of Macsyma, Inc. (USA) for their interest in this work, for supplying Macsyma licenses and the modification of the `LINSOLVE`-function,
- The Center for Microelectronics of the University of Kaiserslautern, in particular Dr. Peter Conradi and Uwe Wassenmüller, for the support of the project,
- Clemens, Frank and Michael for the first-class WG life and particularly Michael for the temporary leaving of its computer,
- My parents and grandparents for their constant support during my study,
- and quite particularly my friend Dr. Ralf Sommer for outstanding co-operation and the joint activities in the last three years.

Eckhard Hennig
Braunschweig, August 1994

Summary

Engineering design tasks require frequently the solution of systems of equations, which describe an object mathematically, along the values of the function defining components and parameters. For the analytic solution of lower dimensioning problems the use of commercial computer algebra systems such as Maxima are helpful, which are able, to manipulate extensive equations algebraically and to solve them symbolically using their variables.

Despite their high capabilities these systems are however usually already overwhelmed, if linear or weakly nonlinear, parameterized sets of equations are to be solved after only a subset of their variables or be before-processed at least symbolically. In order to be able to treat such sets of equations, typically with draft tasks in the context of this work, a universal symbolic equation solver based on heuristic algorithms was developed and implemented in Maxima. The program module `SOLVER` extends the functionality of the Maxima commands `SOLVE` and `LINSOLVE` for the symbolic solution of algebraic equations or systems of linear equations by the ability for the selective solution of nonlinear, parameterized systems with some degrees of freedom.

The first chapter of this work describes some areas of application of symbolic analysis methods, the respective requirements following from them to a symbolic equation solver as well as the used heuristic algorithms for the extraction of linear equations and for the complexity valuation of algebraic functions. The second chapter contains an overview of the structure of the *Solvers* and guidance to its use. In the appendix is the source text of the module implemented in the internal higher programming language of Maxima, `SOLVER.MAC`.

Contents

Chapter 1

Heuristic algorithms for symbolic solution of systems of equations

1.1 Introduction

1.1.1 Numerical Methods compared to Symbolic Methods

Engineering design for given specifications often requires the solution of nonlinear systems of equations after defining component element parameters. Usually for the determination of the wanted variables numeric optimizing procedures are used, which do not function reliably however already for small design problems with a comparatively small number of unknowns. Above all, the complexity order of many usual optimizing algorithms is exponentially dependent on the number of variables, which cause problems (a problem, known as the *curse of dimensionality* [MIC_94] is), then the selection of suitable initial values, the unwanted finding of locally instead of global optima and the principle-conditioned behavior during the solution of systems of equations with degrees of freedom, with which a uniqueness of the solution vector is not realizable.

The best procedure for the accomplishment of a design task would basically be a complete analytic solution of the corresponding system of equations. Analytic functions, which describe explicitly the wanted values as functions of the specification parameters, would have to be determined only once and would be available afterwards to the arbitrarily frequent evaluation with modified parameters, e.g. within design data bases of CAD systems. Besides analytic dimensioning formulas clarify qualitatively functional connections between the element values and the specifications and reveal in the relevant cases the existence of degrees of freedom. Since there exists however no generally accepted procedures for the solution of nonlinear systems of equations and in those special cases, in which symbolic solutions can be calculated, the complexity of the results far exceeds by hand calculation to mastering frameworks, the analytic handling of very low dimensioning problems plays so far a subordinated role.

With the help of modern, efficient computer algebra systems, which are able to manipulate systems of symbolic equations algebraically, and solve using arbitrary variables, become some of these problem categories nevertheless accessible for an analytic handling. In particular such problems, specified above, which require the solution of linear or multivariate polynomial systems. Examples of such applications are within many fields of engineering sciences, such as the design of analog electronic circuits, regulation-technical problems, engineering mechanics and robotics [PFA_94].

1.1.2 Examples of Areas of Application of Symbolic Design Methods

On the basis of the following two examples we demonstrate some areas of application of symbolic methods. At the same time, the concrete requirements should be worked out at them, which must be considered with respect to the development of a universal solution algorithm for symbolic systems of equations .

Example 1.1.

The first example concerns a simple task of engineering mechanics. Given is the two rod truss represented in figure 1.1, [BRO_88, S. 112], at which under the angle γ opposite the horizontals the force F attacks in the point C. The rods form the angles α and β with the horizontals, the height of the triangle stretched by the rods are denoted with c. The Cross sections A_1 and A_2 of the two rods are squares with the side lengths h_1 and h_2. The modulus of elasticity of the material used is E.

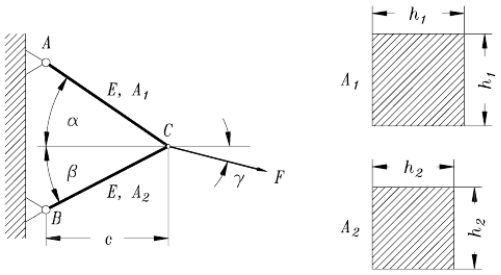


Figure 1.1:
Loaded two rod truss (in German: 'Stabzweischlag')

Due to the load by the force F the truss deforms in such a way, that the point of the triangle in relation to the unloaded status around the vector $(u, w)^T$ shifts, see figure 1.2. On the assumption that the rod lengths variations are small due to the load in relation to the original lengths, now the cross section dimensions h_1 and h_2 of the rods are to be determined in such a way, that for given F, α, β, γ, c and E there results exactly one prescribed shift $(u, w)^T$, i.e. we look for

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \mathbf{f}(F, \alpha, \beta, \gamma, c, E, u, w). \tag{1.1}$$

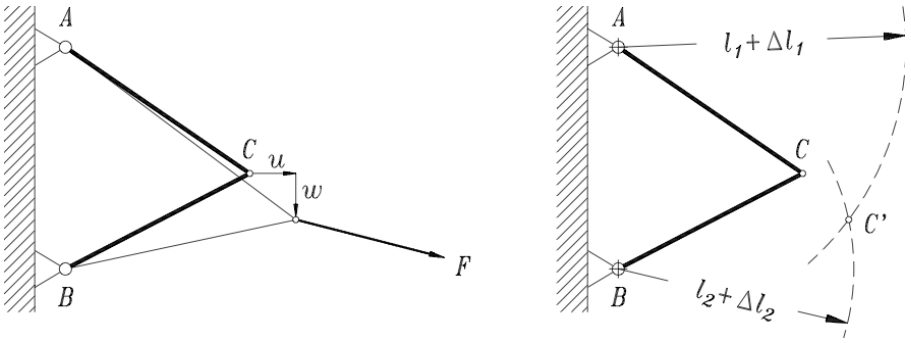
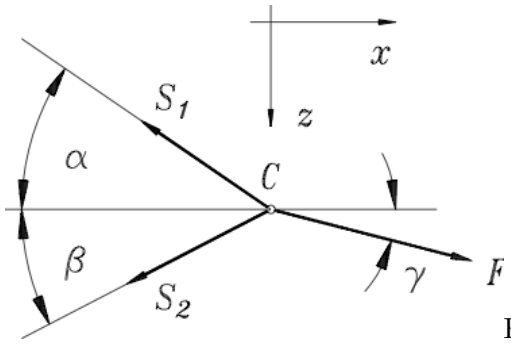


Figure 1.2:
Two rod truss elastic displacements

Solution: From the static equilibrium conditions at the point C it follows after figure 1.3

$$\sum F_{xi} = 0 \implies F\cos\gamma - S_1\cos\alpha - S_2\cos\beta = 0, \tag{1.2}$$

$$\sum F_{zi} = 0 \implies F\sin\gamma - S_1\sin\alpha + S_2\sin\beta = 0. \tag{1.3}$$



Free body diagram

Figure 1.3:
Equilibrium of forces at point C

The material equations for the variations of the rod lengths are

$$\Delta l_1 = \frac{S_1 l_1}{EA_1}, \tag{1.4}$$

$$\Delta l_2 = \frac{S_2 l_2}{EA_2}, \tag{1.5}$$

whereby for the rod lengths l_1 and l_2 and for the cross-section areas A_1 and A_2 we have

$$l_1 = \frac{c}{\cos\alpha} \tag{1.6}$$

$$l_2 = \frac{c}{\cos\beta} \tag{1.7}$$

$$A_1 = h_1^2, \tag{1.8}$$

$$A_2 = h_2^2. \tag{1.9}$$

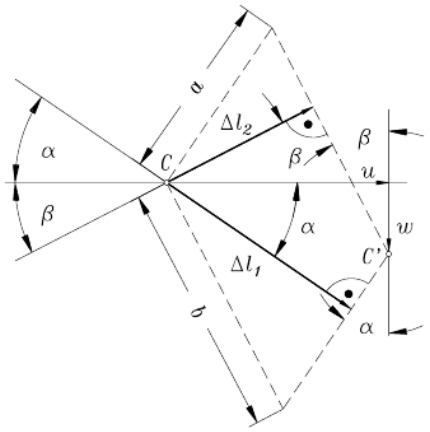


Figure 1.4:
Displacement plan

In the displacement diagram, drawn in figure 1.4, we replaced the circular arcs, on which the rod ends can move, by the tangents perpendicular to the original rod directions. This is possible, because of the small geometry variations. For the side lengths of the dashed parallelogram we have

$$a = \frac{\Delta l_2}{\cos(90° - \alpha - \beta)} = \frac{\Delta l_2}{\sin(\alpha + \beta)}, \tag{1.10}$$

$$b = \frac{\Delta l_1}{\cos(90° - \alpha - \beta)} = \frac{\Delta l_1}{\sin(\alpha + \beta)}. \tag{1.11}$$

Therefore we have for the shifts u and v

$$u = a \sin\alpha + b \sin\beta, \tag{1.12}$$

$$v = -a \cos\alpha + b \cos\beta. \tag{1.13}$$

The task is now, to solve the symbolic set of equations from the equations (1.2) through (1.13) for the unknowns h_1 and h_2 explicitly, whereby all unnecessary variables, i.e. S_1, S_2, A_1, A_2, Δl_1, Δl_2, a and b, are to be eliminated.

□

Example 1.2.

The second example originates from electro-technology and concerns the design of analog electronic circuits. For the two-stage transistor amplifier drawn in figure 1.5 [N"UH_89], there are circuit design equations to be determined, which describe the values of the seven resistances $R_1 \ldots R_7$ as function of the operating voltage V_{CC}, of the small signal amplification A, of the input resistance Z_i and the output resistance Z_o of the circuit for numerically determined

operating points of the transistors:

$$\begin{pmatrix} R_1 \\ \vdots \\ R_7 \end{pmatrix} = \mathbf{f}(V_{CC}, A, Z_i, Z_o). \tag{1.14}$$

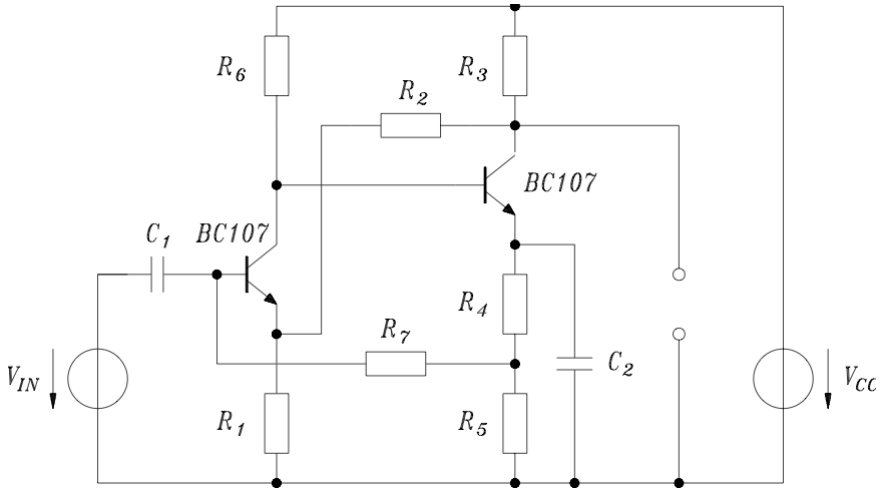


Figure 1.5:
Two-stage transistor amplifier

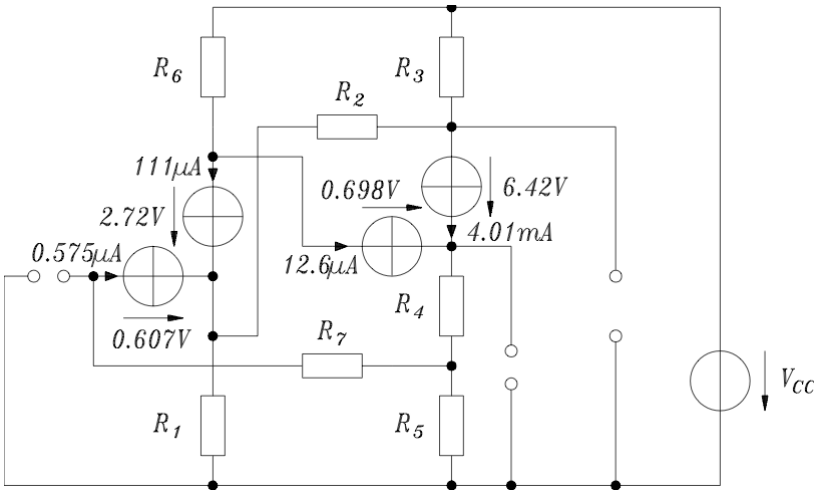


Figure 1.6:
Small signal equivalent circuit diagram of the amplifier with specified transistor operating points

For this purpose, with the help of symbolic network analysis procedures [SOM_93a] as well as symbolic approximation methods [HEN_93], at first the (highly simplified) transfer functions A, Z_i and Z_o in the pass band of the amplifier are calculated as functions of the element parameters

and the operating point values.

$$A = \frac{145303681853R_2}{145309663773R_1} \tag{1.15}$$

$$Z_i = R_7 \tag{1.16}$$

$$Z_o = \frac{1675719398828125\,R_2\,R_7 + 394048139880824192\,R_1\,R_2}{136552890630303121408\,R_1} \tag{1.17}$$

The values of the resistances $R_1 \ldots R_7$ do not only determine the small signal characteristics, but determine the operating point of the circuit. Therefore the resistances are to be determined in such a way, that the small signal and the operating point specifications are fulfilled *at the same time*. For this reason, an extensive sparse tablet set of equations (1.18) – (1.51) is added to the equations (1.15) – (1.17), which comes out of the small signal circuit diagram of the

amplifier represented in figure 1.6.

$$
\begin{aligned}
I_{\mathrm{VIN}} + I_{\mathrm{C1}} &= 0 && (1.18)\\
I_{\mathrm{R7}} + I_{\mathrm{FIX2,Q1}} - I_{\mathrm{C1}} &= 0 && (1.19)\\
I_{\mathrm{R2}} + I_{\ R1} - I_{\mathrm{FIX2,Q1}} - I_{\mathrm{FIX1,Q1}} &= 0 && (1.20)\\
I_{\mathrm{R6}} + I_{\mathrm{FIX2,Q2}} + I_{\mathrm{FIX1,Q1}} &= 0 && (1.21)\\
-I_{\mathrm{R7}} + I_{\mathrm{R5}} + I_{\mathrm{R4}} &= 0 && (1.22)\\
-I_{\mathrm{R4}} - I_{\mathrm{FIX2,Q2}} - I_{\mathrm{FIX1,Q2}} + I_{\mathrm{C2}} &= 0 && (1.23)\\
I_{\mathrm{R3}} - I_{\mathrm{R2}} + I_{\mathrm{FIX1,Q2}} &= 0 && (1.24)\\
I_{\mathrm{VCC}} - I_{\mathrm{R6}} - I_{\mathrm{R3}} &= 0 && (1.25)\\
-V_{\mathrm{VIN}} + V_{\mathrm{R1}} + V_{\mathrm{FIX2,Q1}} + V_{\mathrm{C1}} &= 0 && (1.26)\\
-V_{\mathrm{R2}} + V_{\mathrm{FIX2,Q2}} - V_{\mathrm{FIX1,Q2}} - V_{\mathrm{FIX1,Q1}} &= 0 && (1.27)\\
-V_{\mathrm{R6}} + V_{\mathrm{R3}} + V_{\mathrm{R2}} + V_{\mathrm{FIX1,Q1}} &= 0 && (1.28)\\
V_{\mathrm{R7}} + V_{\mathrm{R4}} - V_{\mathrm{R2}} - V_{\mathrm{FIX2,Q1}} - V_{\mathrm{FIX1,Q2}} &= 0 && (1.29)\\
-V_{\mathrm{VIN}} + V_{\mathrm{R7}} + V_{\mathrm{R5}} + V_{\mathrm{C1}} &= 0 && (1.30)\\
-V_{\mathrm{VIN}} + V_{\mathrm{R2}} + V_{\mathrm{FIX2,Q1}} + V_{\mathrm{FIX1,Q2}} + V_{\mathrm{C2}} + V_{\mathrm{C1}} &= 0 && (1.31)\\
V_{\mathrm{VCC}} - V_{\mathrm{VIN}} + V_{\mathrm{R6}} + V_{\mathrm{FIX2,Q1}} - V_{\mathrm{FIX1,Q1}} + V_{\mathrm{C1}} &= 0 && (1.32)\\
R1 \cdot I_{\mathrm{R1}} - V_{\mathrm{R1}} &= 0 && (1.33)\\
R2 \cdot I_{\mathrm{R2}} - V_{\mathrm{R2}} &= 0 && (1.34)\\
R3 \cdot I_{\mathrm{R3}} - V_{\mathrm{R3}} &= 0 && (1.35)\\
R4 \cdot I_{\mathrm{R4}} - V_{\mathrm{R4}} &= 0 && (1.36)\\
R5 \cdot I_{\mathrm{R5}} - V_{\mathrm{R5}} &= 0 && (1.37)\\
R6 \cdot I_{\mathrm{R6}} - V_{\mathrm{R6}} &= 0 && (1.38)\\
R7 \cdot I_{\mathrm{R7}} - V_{\mathrm{R7}} &= 0 && (1.39)\\
-I_{\mathrm{C1}} &= 0 && (1.40)\\
-I_{\mathrm{C2}} &= 0 && (1.41)\\
V_{\mathrm{VIN}} &= 0 && (1.42)\\
V_{\mathrm{VCC}} &= VCC && (1.43)\\
V_{\mathrm{FIX1,Q1}} &= 2.72 && (1.44)\\
V_{\mathrm{FIX2,Q1}} &= 0.607 && (1.45)\\
V_{\mathrm{FIX1,Q2}} &= 6.42 && (1.46)\\
V_{\mathrm{FIX2,Q2}} &= 0.698 && (1.47)\\
I_{\mathrm{FIX2,Q2}} &= 1.26 \cdot 10^{-5} && (1.48)\\
I_{\mathrm{FIX1,Q2}} &= 0.00401 && (1.49)\\
I_{\mathrm{FIX2,Q1}} &= 5.75 \cdot 10^{-7} && (1.50)\\
I_{\mathrm{FIX1,Q1}} &= 1.11 \cdot 10^{-4} && (1.51)
\end{aligned}
$$

For the determination of the wanted dimensioning regulations in the form (1.14) from the set of equations (1.15) – (1.51) all branch voltages and current flows $V_{??}$ and $I_{??}$ are to be eliminated and the remaining equations solved for the resistances $R_1 \ldots R_7$.

□

1.2 Limits of the Application of Conventional Equation Solvers

If an attempt is undertaken to use the standard routines of well-known commercial computer algebra systems like Macsyma [MAC_94] or Mathematica [WOL_91] for the solution of the systems of equations from the two examples from above, then in the case of use of Maxima/Macsyma in the example 1.1 we get usually the following typeof result, like here:

```
(COM1) Solve(
  [
    F*cos(gamma) - S1*cos(alpha) - S2*cos(beta) = 0,
    F*sin(gamma) - S1*sin(alpha) + S2*sin(beta) = 0,
    Delta_l1 = l1*S1/(E*A1),
    Delta_l2 = l2*S2/(E*A2),
    l1 = c/cos(alpha),
    l2 = c/cos(beta),
    a = Delta_l2/sin(alpha+beta),
    b = Delta_l1/sin(alpha+beta),
    u = a*sin(alpha) + b*sin(beta),
    w = -a*cos(alpha) + b*cos(beta),
    A1 = h1^2,
    A2 = h2^2
  ],
  [h1, h2]
);


(D1)  [ ]
```

```
(COM1) Solve(
  [
    F*cos(gamma) - S1*cos(alpha) - S2*cos(beta) = 0,
    F*sin(gamma) - S1*sin(alpha) + S2*sin(beta) = 0,
    Delta_l1 = l1*S1/(E*A1),
    Delta_l2 = l2*S2/(E*A2),
    l1 = c/cos(alpha),
    l2 = c/cos(beta),
    a = Delta_l2/sin(alpha+beta),
    b = Delta_l1/sin(alpha+beta),
    u = a*sin(alpha) + b*sin(beta),
    w = -a*cos(alpha) + b*cos(beta),
    A1 = h1^2,
    A2 = h2^2
  ],
  [h1, h2]
);
(D1) []
```

This behavior of the computer algebra systems is explained with the fact, that the systems of equations to be solved for the interesting variables h_1 and h_2 are regarded as over-determined, because the equation solvers cannot be given additional information about (after possibility) the *to be eliminated* variables (S_1, S_2, A_1, A_2, Δl_1, Δl_2, a, b) resp. the parameters of the system (F, α, β, γ, c, E, u, w), which should *not be eliminated under any circumstances*.

A possible way out for the equation solvers consists in letting them determine also solutions for the not interesting variables. However, this is not always feasible and usually very inefficient, because in some cases much computing time is necessary for the calculation of variables, which have no impact on the looked for unknowns. Even at all, no solution is found, if there is no analytic solution for only one not interesting variable. The latter applies for example to the following system of equations, if only the solutions for x and y are looked for:

$$x + y = 1 \tag{1.52}$$
$$2x - y = 5 \tag{1.53}$$
$$yz + \sin z = 1. \tag{1.54}$$

From the two linear equations (1.52) and (1.53) the solutions $x = 2$ and $y = -1$ are determined directly, not in the contradiction with the remaining nonlinear equation (1.54). Maxima does not detects this circumstance and gives back only error messages – in the first attempt (`COM3`) due to the apparent over-determinacy of the system, in the second attempt (`COM4`) due to the analytically not solvable third equation:

```
(COM2) Eq : [x + y = 1, 2*x - y = 5, y*z + sin(z) = 1]$
(COM3) Solve( Eq, [x, y] );
Inconsistent equations:  (3)
(COM4) Solve( Eq, [x, y, z] );
ALGSYS cannot solve - system too complicated.
```


```
(COM2) Eq : [x + y = 1, 2*x - y = 5, y*z + sin(z) = 1]$
(COM3) Solve( Eq, [x, y] );
Inconsistent equations:  (3)
(COM4) Solve( Eq, [x, y, z] );
ALGSYS cannot solve - system too complicated.
```


1.3 Requirements to an Universal Symbolic Equation Solver

For the symbolic solution of the system of equations, it is necessary that the `Solve` function in none of the two above cases aborts prematurely. In the first case, after a consistency check with equation (1.54), the solutions x and y should be returned. In the latter case it is to be desired, that beside the analytically calculated solutions, the remaining equations, for which no such solutions could be found, should be returned additionally in implicit form – so that these could be solved with numerical methods. An adequate response to the command `COM4` would then be e.g. an output of the form

$$[x = 2,\ y = -1,\ -z + \sin z = 1]\,.$$

The functions for the solution of sets of equations, provided by Maxima, are not modifiable without access to the system core in the way that they show the required behavior. The aim of this report is therefore a conceptualization and implementation of a universal symbolic equation solver, which is based on Maxima standard routines, which is able to solve sets of equations of the type stated in the examples after any subset of all variables. Or at least by elimination of not necessary variables as much as possible to do a large symbolic preprocessing of the equations, so that numeric optimizing procedures do have only be applied to a small, analytically not solvable nonlinear core of the system.

Apart from this general objective, detailed requirements can be derived for the developed program from some well-known facts and a series of observations, which came from the examples 1.1 and 1.2 as well as the system of equations (1.52) – (1.54):

1. Usually only the solution for some few variables is asked for, all other unknowns are to be eliminated.
2. The sets of equations which are to be solved can be one times or more times parameterized.
3. It is not guaranteed by any means, that the parameters are independent from each other, i.e. it is possible that a system of equations has only a solution, if certain arithmetic forced conditions between some parameters are kept.
4. The systems of equations contain often simple, direct assignments of the form $x_i = \text{const.}$, see equations (1.6) and (1.7) or (1.42) – (1.51).
5. A substantial proportion of the equations to be solved, is linear with respect to a not directly evident subset of all variables, see equations (1.18) – (1.32)which are linear with respect to all $V_{??}$ and $I_{??}$.
6. The systems can contain degrees of freedom.
7. There exists no generally valid solution procedures for nonlinear equations and systems of equations.
8. Nonlinear equations can be unsolvable (contradictory), or have unique or multiple solutions with finite or infinite solution varieties.
9. Not always, all members of the multiple solution set are consistent with the remaining equations.
10. For many nonlinear equations there exists no analytic solutions, see equation (1.54).

From these statements the following requirements results corresponding to the points above:

1. The program should solve systems of equations only so far, as it is absolutely necessary for the determination of the individual variables. Calculated solutions are to be checked for possible contradictions with respect to the remaining equations.
2. Looked for variables and parameters must to be processed separately from each other. Under any circumstances, parameters may not be eliminated – in contrast to not interesting variables.

3. If dependencies between parameters are detected, then the program run must continue under consideration and storage of the appropriate force conditions, if desired by the user.

4. Direct assignments should be looked for and executed directly at the beginning of the program run, in order to reduce the scope of the remaining system of equations as far as possible and with little effort.

5. Because there exists efficient, closed solution procedures for linear equations, it is advisable to search the system of equations repeatedly for linear blocks, to solve these and put the results into the remaining equations, until no more linear parts of equations are present.

6. Degrees of freedom are to be expressed automatically in variables selected by the program.

7. The solution of nonlinear equations must be controlled with the help of heuristic evaluation strategies.

8. In case of multiple solutions with finite varieties, each individual solution path is separately recursively to be pursued.

9. Multiple solutions, which are inconsistent with the remaining equations, must be detected and the corresponding solution path rejected.

10. As was already required at the beginning of the section, equations not analytically solvable should not lead to the abort of the program. Instead the system of equations should be brought on triangle form is as far as possible and the remaining, not solvable equations returned along with the partial solutions determined up to then.

1.4 Extraction and Solution of Linear Equations

With design tasks, the equations which are to be solved, are mostly nonlinear, but the corresponding systems frequently contains large sections of linear blocks. Since linear systems of equations can be solved simultaneously very efficiently with the help of the Gauss-Elimination, it is advisable to process first the linear proportion of the system separately before the solution of the nonlinear equations. Even if a complete analytic solution of the entire nonlinear system for all searched variables cannot be achieved, it is nevertheless meaningful to reduce by elimination of the linear variables and equations the system to an only small, not any longer analytically solvable core. The numeric solution of that core is substantially less complex, than an optimization of the complete system.

Under point 5 we required an iterative solution of linear subsystems of the entire set of equations. This is a non trivial task, because neither the concerned equations nor the subset of variables, for which these equations are linear are known from the beginning. Therefore a search strategy must be found, which extracts the linear equation blocks and variable blocks (for efficiency reasons as large ones as possible) from a given nonlinear system of equations.

1.4.1 Intuitive Methodologies for the Search of Linear Equations

For the clarification of the task the following nonlinear system of equations is considered, which is to be solved after the variables x, y and z.

$$\begin{aligned} x + 2y - z &= 6 && (1.55)\\ 2x + yz - z^2 &= -1 && (1.56)\\ 3x - y + 2z^2 &= 3 && (1.57) \end{aligned}$$

At first sight only the equation (1.55) is linear, regarding all three variables. Using a simple search algorithm, which finds only such completely linear equations, in this case maximal one variable can be eliminated from the two remaining equations after a solution of (1.55) e.g. after x.

A more exact view of the equations reveals however a better alternative. After canceling of equation (1.56) and shifting the terms dependent on z on the right hand sides of equations (1.55) and (1.57), we get *two* linear equations with the variables x and y:

$$\begin{aligned} x + 2y &= 6 + z, && (1.58)\\ 3x - y &= 3 - 2z^2. && (1.59) \end{aligned}$$

Their simultaneous inversion leads to solutions parameterized in z

$$\begin{aligned} x &= -\frac{1}{7}\left(4z^2 - z - 12\right), && (1.60)\\ y &= \frac{1}{7}\left(2z^2 + 3z + 15\right), && (1.61) \end{aligned}$$

after their inserting into (1.56) only *one* nonlinear equation remains, which is to be solved :

$$2z^3 - 12z^2 + 17z + 31 = 0. \qquad (1.62)$$

In view of the fact that with the second version in only one iteration two unknowns could be determined at the same time, this latter method is to be preferred in contrast to the search for completely linear equations – despite the additional expenditure w.r.t. the algebraic transformation. This applies in particular if a system of equations does not contain any equations, in which all variables involved occurs in purely linear form. The procedure used for the extraction of linear subsystems should therefore connect both demonstrated operations for the removal of nonlinear pieces of equations:

1. canceling of individual nonlinear equations
2. shifting variables occurring in nonlinear terms to the right hand sides of the equations

By a balanced combination of these two operations it can be achieved that the resulting linear subsystems have maximal size and often are – at least approximately – square.

1.4.2 A Heuristic Algorithm for the Search of Linear Equations

For a computer implementation such a search for blocks of linear equations, executed intuitively by humans, must be systematized and formulated algorithmically. Since the term *linear piece of a system of equations* does not define the desired result in unique way, which is evident on the basis the different solution procedures, a heuristic strategy was developed for the imitation of the intuitive methodology. This strategy is demonstrated now for the comparison of the results at the already regarded system(1.55) – (1.57).

For the system of equations, at first a table is set up, whose lines are assigned to the equations and the columns corresponds to the variables. The entry at the position (i, j) of the table contains for the equation i the coefficient[1] of the linear term, i.e. the first power of the variable x_j. If as for z in equation (1.57), no term in first power is existent or if this term occurs as argument in non–polynomial functions (e.g. $\sin x$ or $\sqrt{x}$), then the corresponding position is marked with a cross ($\times$).

	x	y	z
Eq. 1)	1	2	-1
Eq. 2)	2	z	y
Eq. 3)	3	-1	$\times$

An equal large evaluation matrix is assigned to this table, whose entries are equal to zero '0', if the corresponding entry in the coefficient table is a constant, and equal unity '1', if the corresponding linear coefficient contains a searched variable or is not existent ($\times$). Moreover the row and column totals become noted at the edges of this matrix, as well as under the sigma signs on the top right and on the left down respectively the row total of the column totals ($\sum C$) and the column total of the row totals ($\sum R$).

		x	y	z	$\sum C$
		0	1	2	3
1)	0	0	0	0	
2)	2	0	1	1	
3)	1	0	0	1	
$\sum R$	3				

(1.63)

Obviously the '1' entries of the evaluation matrix correspond to the non-wanted non-linear parts of the system of equations. A linear piece in the system of equations and the corresponding variables are found, if with a sequence of operations specified at the end of section 1.4.1, all ones '1' were eliminated. Transferred to manipulations to the evaluation matrix the 1st operation corresponds to deleting the row belonging to to a certain equation. The 2nd operation is equivalent to deleting the column, which is associated to some variable. The linear subsystem consists afterwards of those equations and variables, whose rows and columns were not removed from the matrix.

The reduction of the evaluation matrix (1.63) to a zero matrix can take place in exactly three different ways:

1. canceling of the rows 2) and 3),
2. canceling of the columns y and z,

[1] Maxima has the instruction `ratcoeff` to determine the coefficients of rational terms.

3. canceling of row 2) and column z.

The requirement for maximal size and preferably square shape of the linear equation blocks is directly portable to the wanted properties of the zero matrix. In this sense, the latter of the three options is optimal, because it results in optimum solution i.e. to the system (1.58) – (1.59), already detected in the previous section. However, the other two possibilities lead to the under-determined equation (1.55) or to an over-determined 3×1-system in x.

The search for an optimal sequence of row and column cancellations is a complex combinatorial problem. In order to avoid the associated expenditure, a heuristic, local decision criterion becomes handy for the determination of the row or column, which should be removed in the respective step, i.e. a *Greedy* strategy [FOU_92], which is: That row or column is to be deleted, which contains the most ones '1', i.e. that with the largest row or column total. This criterion still does not supplies a clear decision,

- if two or more rows have the same (largest) row total,,
- or two or more columns have the same (largest) column total,
- or if the totals of the highest evaluated row and the highest evaluated column are identical.

In the first two cases any row or column of the candidates can be selected, usually – for the sake of simplicity– the first one, which is found with maximal evaluation from the concerned rows or columns.

The third case occurs in the example above. In the evaluation matrix (1.63) both row 3) and the column z have the maximal sum total 2. At the start, from both possibilities we select arbitrarily the cancellation of the row, so that in the next step the following evaluation matrix results:

		x	y	z	$\sum C$
		0	0	1	1
1)	0	0	0	0	
3)	1	0	0	1	
$\sum R$	1				

(1.64)

Once again, the maximal row total equal to the maximal column total. Now, the decision could take place according to the random principle, but thereby the requirement for a square shape of the linear subsystems would not sufficiently respected. Therefore it is preferable either to delete rows and columns with same evaluation *alternately* or favor the decision, which brings the dimension relation n/m of the $n \times m$– evaluation matrix with $n \neq m$ more near at unity '1'. According to both criteria deleting of the column z proves more favorable than cancelling of row 3).

		x	y	$\sum C$
		0	0	0
1)	0	0	0	
3)	0	0	0	
$\sum R$	0			

(1.65)

The cancellation of column z causes the removal of the last unity '1' in the evaluation matrix. This expresses itself in disappearing of $\sum S$ and $\sum Z$, by which the end of the algorithm is marked. From the result matrix (1.65) now can be read off, that the the example system of

equations (1.55) and (1.57) are linear concerning the variables x and y. The finally necessary transformations of the linear equations for the creation of the simultaneous form (1.58) – (1.59) are no problem for a computer algebra system: Maxima uses the instruction `linsolve` for the simultaneous solution of linear equations, which are then automatically executed.

1.4.3 Solution of the Linear Equations

If the linear pieces of the systems of equations, which are extracted using the described algorithm, are unique solvable or under-determined, then their subsequent treatment is unproblematic. In the case of over-determined systems inevitable occur inconsistencies, which require a more detailed handling. E.g. from a larger system of equations in the variables x, y, z and w, the following over-determined linear subsystem (1.66) – (1.68) in x and y be taken.

$$x - y = z^2 + z \tag{1.66}$$
$$x + y = w^2 + 1 \tag{1.67}$$
$$x - y = z + w \tag{1.68}$$

After the forward elimination we have the following systen of equations, which is inconsistent in the sense of linear algebra and therefore no solution exists:

$$x - y = z^2 + z \tag{1.69}$$
$$2y = w^2 - z^2 - z + 1 \tag{1.70}$$
$$0 = w - z^2 \tag{1.71}$$

In the regarded case however, z and w are the variable of the system of equations to be solved. The linear subsystem from above has solutions, if and only if these two variables fulfill the equation (1.71). This condition is therefore only regarded as a further equation of the remaining system, from which x and y were eliminated.

As consequence it results, that with the occurrence of apparent inconsistencies following the eliminations process, the right sides of the consistency conditions thereupon must be checked generally, whether they contain looked-for-variables of the entire system. If this is the case, then the corresponding conditions are again added to the initial system of equations after the solution of the linear equations. If this does not applies, i.e. does not occur not fulfillable obligation conditions between numeric values or parameters, then the set of equations has indeed no solution, and the solution process must be aborted.

1.5 Evaluation Strategies for the Solution of Nonlinear Equations

Apart from a few special cases there are closed solution procedures for general nonlinear systems of equations. This does not exclude however, that for many nonlinear systems analytic solutions or at least partial solutions can be calculated, but usually their determination is not as efficient as by Gauss-Elimination in the case of linear equations.

1.5.1 Substitution Method for Nonlinear Systems of Equations

An elementary solution procedure, which can be applied to any systems of equations, is the well-known substitution method:

1. Select an equation (most simple as possible) from the system and solve it after a variable x_j. Abort, if all equations are solved, or no further equation is analytically solvable.
2. Insert the result into the remaining equations, in order to eliminate x_j from the system.
3. Check the system, reduced by one equation and one variable, for consistency and continue with step 1.

For a demonstration of this method, the nonlinear system of equations (1.72) – (1.76) is regarded, which is to be solved after the variables a, b, c and d.

$$ab + 2c = 0 \tag{1.72}$$
$$c^2 + d - 4 = 0 \tag{1.73}$$
$$\sqrt{b + d} - 2 = 0 \tag{1.74}$$
$$\tan\left(\frac{\pi}{2a}\right) - 1 = 0 \tag{1.75}$$
$$b \operatorname{Arcosh} c - i\pi = 0 \tag{1.76}$$

Already directly at the beginning of the application of the substitution method the question arises, which concrete characteristics distinguish an equation as "as simple" as possible. From general experience and know-how, among other things the following evaluation criteria can be derived, which need not apply necessarily at the same time and also be differently weighted depending on the application.

"Simple" Equations ...

1. contain only fewof the wanted variables,
2. contain a wanted variable at exactly one position, so that the unknown itself can relatively easily be isolated,
3. have only small depths of the operation hierarchy concerning one or several variables, i.e. the *formula complexity* is small,
4. contain no transcendental or other functions, which cannot be inverted without difficulty.

On the basis of these criteria, now the simplest equation of the example system is to be determined. Regarding the first two points, this could be the equation (1.75), because it contains only the variable a and this occurs at exactly one position. On the other hand, the evaluation does not precipitate very favorably due to the criteria 3 and 4. Regarding the points 2, 3 and 4, the solution of the equation (1.73) after the variable d appears as the best selection, because all other equations contain either more variables or more only difficult resolvable functions.

As relevant criterion, at first the combination of the points 1 and 2 may be considered, so that one starts with the solution of the equation (1.75) for the variable a. If only the principal value of the `arctan` function is considered, it follows:

$$a = 2. \tag{1.77}$$

Substituted into the remaining four equations the system is now

$$\begin{aligned} 2b + 2c &= 0, && (1.78)\\ c^2 + d - 4 &= 0, && (1.79)\\ \sqrt{b+d}\, - 2 &= 0, && (1.80)\\ b\ \mathrm{Arcosh}\, c\, - i\pi &= 0. && (1.81) \end{aligned}$$

Because no inconsistencies arise, we continue with the selection of the next simplest equation. The valuation criteria speak now immediately[2] for the solution of the equation (1.79) for d:

$$d = 4 - c^2. \tag{1.82}$$

It follows

$$\begin{aligned} 2b + 2c &= 0, && (1.83)\\ \sqrt{b + 4 - c^2}\, - 2 &= 0, && (1.84)\\ b\ \mathrm{Arcosh}\, c\, - i\pi &= 0. && (1.85) \end{aligned}$$

Concerning all criteria, now equation (1.83) is the most favorable candidate, so that with the solution

$$b = -c \tag{1.86}$$

still two equations with the unknown c remain.

$$\begin{aligned} \sqrt{4 - c - c^2} - 2 &= 0, && (1.87)\\ -c\ \mathrm{Arcosh}\, c\, - i\pi &= 0. && (1.88) \end{aligned}$$

Equation (1.88) is analytically not solvable, therefore independently of the evaluation, it is equation (1.87) which must supply the missing solution for c. In this case, a multiple solution results for the first time:

$$[c = 0,\ c = -1]\,. \tag{1.89}$$

Now the importance of the consistency check shows up, which was not relevant so far. From the two solutions only the second, $c = -1$, fulfills equation (1.88). The other solution leads to the contradictory predicate

$$-i\pi = 0 \tag{1.90}$$

and must therefore be rejected. After the back substitution the consistent solutions are

$$[a = 2,\ b = 1,\ c = -1,\ d = 3]\,. \tag{1.91}$$

1.5.2 Heuristic Methods for the Complexity Evaluation of Algebraic Terms

If the evaluation of algebraic equations regarding their "simplicity" and the corresponding solution of a nonlinear system of equations, based on it by a computer algebra system should be made automatically, then the criteria formulated linguistically in the paragraph 1.5.1, must be transformed into algorithms, which supply numeric complexity evaluations for the controlling of the solution processes. The magnitude of an evaluation number b should be a measure of how

[2] Humans would probably consider the first equation to be 'simpler', but the beforehand necessary division of the equation by the factor 2 is an additionally expenditure to be considered.

difficult it is to solve an equation i after a variable x_j analytically. The larger b is, the more complex seems[3] the task.

If the evaluation for each equation is made w.r.t each variable, then a solution sequence for the equations can be generated by means of an sorting on the basis of the magnitudes of the evaluation numbers b. The solution sequence is then representable by an sorted list of the form

$$[\,(i_1, j_1, b_1),\ (i_2, j_2, b_2),\ \ldots\,]\,,$$

whereby for the evaluation numbers we have $b_k \leq b_l$ for $k < l$. As for the equations solver this list implies the following statements: first try to solve equation i_1 for variable x_{j_1}, since this appears simplest. If this does not succeed, then try instead the solution from equation i_2 for x_{j_2}, etc.. If the solution attempt is successful, then insert the solution into the remaining equations and begin from the start with the construction of a new list for a new solution sequence.

The transformation of the first criterion into an algorithm does not represent a serious challenge, because the number of variables contained in an equation is already a numeric value. The implementation using an in a computer algebra system is also no problem. In Maxima the following short instruction suffices

```
Length( ListOfVars( Equation[i] ) )
```

to determine the number of the unknowns in the i equation.

This number is however only of use as secondary criterion in connection with other evaluations, because only an order of rank of the equations is supplied, not however by pairs of equation/variables.

For reasons that will become clear later, the consideration of the criterion 2 is deferred for the moment and we continue with the points 3 and 4. These two criteria were separately listed, but they can be interconnected very easily by a single procedure. The heuristic evaluation algorithm, which is described in in the following section about the implementation of the symbolic equation solver, uses the internal representation of algebraic terms in computer algebra systems for the complexity calculation. Composite algebraic functions are administered in hierarchically organized lists of operators and operands in prefix notation, which can be mapped directly into a tree structure. The nodes of such a tree contain the operators, the operands are in the leafs. For example the figure 1.7 shows the representation of the left sides of the equations (1.75) and (1.87) as trees of operations.

An evaluation of the formula complexity regarding the depth of the operation hierarchy, i.e. the degree of the nesting of a term, is now readable at the tree structures. E.g. the complexity can be determined by counting the branches of a tree, which must be stepped starting from the root of the operation tree, in order to arrive at the instances of the regarded variables. In the case of the term in figure 1.7a) the complexity value b w.r.t. the variable a is equal to the length of the fat drawn path, i.e. $b = 4$. In order to achieve all instances of the variable c from the root of the tree in figure 1.7b), all in all seven branches must be crossed, therefore the complexity is $b = 7$.

This method for the calculation of the complexity is easy expandable in a way, that also the criterion 4 is taken into account, which gives the evaluation of a term regarding the operators

[3] Here the formulation *seems* is selected, because the evaluation is made on the basis of heuristic criteria, which cannot guarantee optimal decisions.

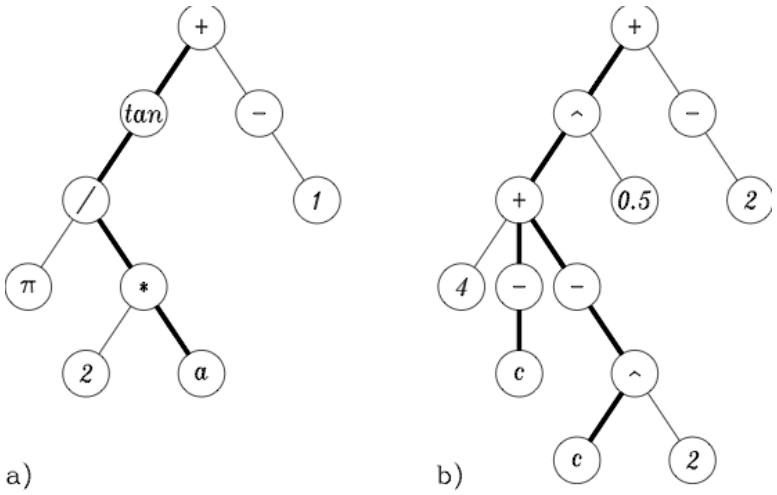


Figure 1.7:
Tree representation of algebraic expressions

contained in it. Instead of the simple counting of the branches of the tree, an additionally weighting must take place, which assigns to each operator an typical "difficulty factor", by which the evaluation of its operands are to be multiplied. The magnitude of this difficulty factor should reflect, how complex the formation of the inverse function for the computer algebra system is, i.e. the solution of the function for its operands, see [TRI_91].

At the beginning each leaf of the tree receives the weighting 1, if it contains that variable, for which the evaluation is to be calculated. Otherwise the leafs get the weighting 0. During the evaluation of the operators the operator "+" serves as reference for the weighting 1, because it is to be inverted most easily. In contrast, the reversal of the operators "$*$" and "tan" are considered arbitrarily as four or ten times as difficulty, accordingly the weightings are set. The following table shows a selection of some operators and their assigned weightings.

operator	+	−	$*$	/	$\wedge$	tan	Arcosh
weighting	1	1	4	4	10	10	12

The calculation of the complexity value takes place *bottom-up* via repeated addition of the branch weights at the operator nodes, multiplication of this sum with the operator weight and transfer of the resulting value onto the superordinate branch, until the tree root is reached. For the demonstration of this procedure the operatior tree in figure 1.8a) is considered. In the squares drawn beside the nodes, the operator weights are marked. The numbers at the branches show the total weight of the subordinated partial tree. Since the complexity of the term is to be evaluated w.r.t. the variable a, only the leaf with the symbol a gets the weight 1, all other leafs are evaluated with value 0. At the multiplication nodes above the variables, the branch weights add themselves to the sum $0 + 1 = 1$, which in accordance with the evaluation of the operator "$*$" are multiplied with the factor 4 and then passed on to right operand branch of the "/" operator. There also takes place a multiplication with 4, so that the intermediate result now equals to 16. Subsequently, in the process of the calculation the factors 10 and 1 for the "tan"- and "+" operators are added. The final result, $b = 160$, is at the root of the tree. For the operator tree in figure 1.8b) similar calculations result in a complexity value of $b = 110$ w.r.t. the variable c.

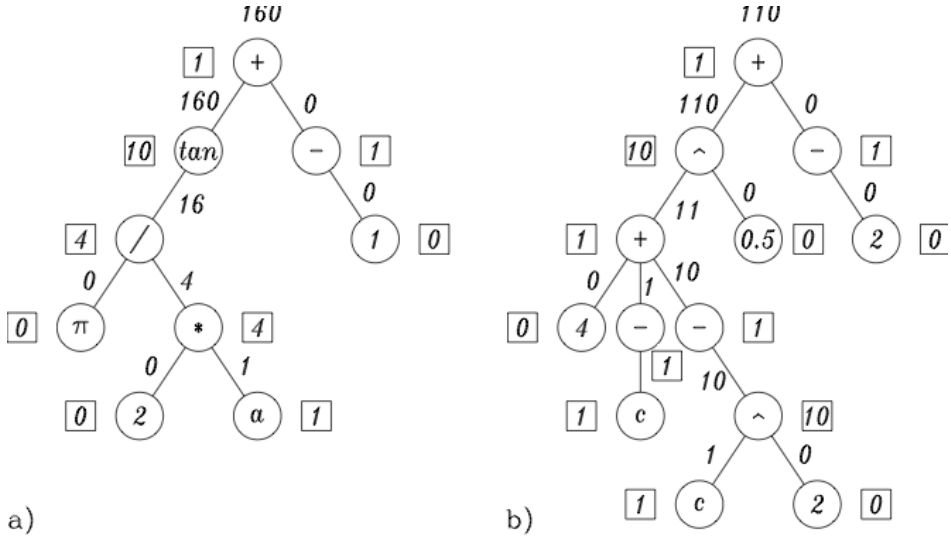


Figure 1.8:
Valuation of operators

Now, we can take up the temporarily deferred criterion 2. In order to determine those equations of a system, which contain one or more variables at exactly one position, it is sufficient to set all operator weights in the calculation formula for the term complexity to '1' and to store all pairs of equation/variables, for which under these conditions the complexity evaluation is $b = 1$. This method is justified by the fact, that the search aims at exactly those equations, in whose tree representation particular variable symbols occur in only one leaf. That means, in these cases there is only one path from the root of the tree to the symbol in question. With a weighting of '1' for all operators the complexity evaluation corresponds exactly to the number of paths to the instances of a variable.

This procedure can be verified easily on the basis of examples. If all operator weights are set equal '1', then for the terms in figure 1.8a) and b) we get complexities of $b = 1$ for the variable a and $b = 2$ for the variable c. This corresponds with the fact, that the variable a is contained in exactly one leaf of the tree, while the symbol c is to be found at two positions.

1.5.3 Order of the Sequence of Solution

By different combining and weighting of the discussed evaluation procedures, different strategies arise for the order of solution steps. The strategy named `MinVarPathsFirst`, which is implemented in the program developed in this work, represents a combination of the criterion 2 and the complexity evaluation by means of operator weighting. Highest priority in the solution step order, receive those equations, which contain wanted variables at exactly one position, whereby within this group one sorts according to smallest weight of the operator trees. All remaining pairs of equation/variables are likewise placed to the end of the solution sequence, according to smallest tree weight. Therefore in the case of the two terms in figure 1.8, at first it would be tried to solve the equation with the tan- function for the variable a, although a higher evaluation was calculated for a than for the accompanying term w.r.t. the variable c.

Chapter 2

The Solver

2.1 The Structure of the Solver

The requirements, in particular the points 4, 5 and 7, set up in chapter 1.3 and schematically represented the structuring of the equations solvers in figure 2.1 suggests to lay down this into five largely independent main modules, see [TRI_91]. Built up on this structure and the heuristic algorithms described in the preceding chapter, the Macsyma program packet `SOLVER` was developed for the functionality extension of the Macsyma functions `SOLVE` and `LINSOLVE`.

The tasks of the module *Solver Preprocessor* are general, checking the command syntax and semantics, the construction of internal data structures, and the execution of an introductory consistency check. The check determines whether the set of equations directly contains contradictory statements like number = number or constraint conditions between parameters .

The *Immediate Assignment Solver* searches the system of equations for direct assignments of the form *var* = const. or const. = *var* before the call of the *Linear Solvers* and executes these immediately, so that the cost of computation for the following program module is kept as small as possible.

The *Linear Solver* is a pre and post processor for the Maxima function `LINSOLVE` for the simultaneous solution of linear systems of equations. The module extracts pieces of linear equations according to the heuristic algorithm described in paragraph 1.4.2 and solves the equations by calling `LINSOLVE`. The resulting solutions are inserted into the remaining equations before leaving the *Linear Solvers*.

The *Valuation Solver* is the core module of the *Solvers*. Its tasks are the use of valuation strategies for the generation of the solution sequences and the solution of the nonlinear equations with the help of the Maxima build-in function `SOLVE`. In the case of multiple solutions the *Valuation Solver* checks each individual solution for consistency with the remaining system of equations. Inconsistent solutions are rejected, while valid solutions are inserted into the remaining system of equations and the corresponding solution paths are tracked separately through recursive calls of the *Valuation Solvers*.

All steps necessary for the editing of the solutions for output to the user are taken over by the *Solver Postprocessor*. These are the expansion of the hierarchically organized solution list supplied by the *Valuation Solver*, the back substitution of the symbolic solutions as well as picking out, evaluating and outputting the variables and composite terms asked for by the user.

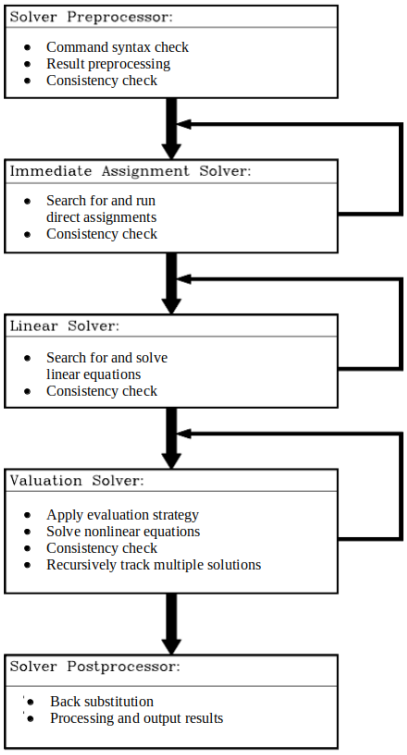


Figure 2.1:
Structure of the Solvers

2.2 The Modules of Solvers

2.2.1 The Solver Preprocessor

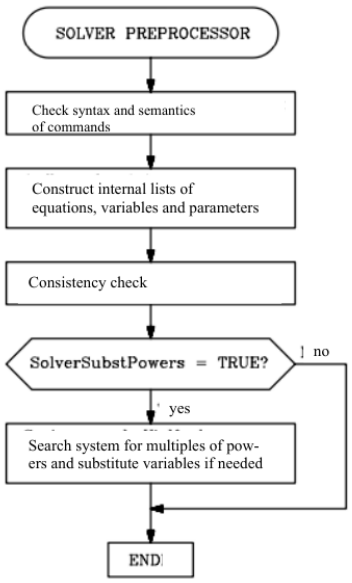


Figure 2.2:
Flow diagram of the Solver Preprocessor

Beside the examination of the command syntax and semantic as well as the construction of the internal equations, variables and parameter lists, the *Solver Preprocessor* executes a consistency check of the equations, whose flow diagram is represented in figure 2.3. Aim of the consistency check is it to detect from the beginning, whether the system of equations is unsolvable due to direct contradictions. Such contradictions can occur in form of pure number equations, e.g. $0 = 1$, or in addition, in the form of constrained conditions between symbols declared as parameters.

In order to uncover direct contradictions, the routine for the consistency check searches after equations in the system of equations, which consist exclusively of numbers or numbers with parameters, but does not contain variables. If a contradictory number equation is found, then the *Solver Preprocessor* aborts immediately. If such an equation is consistently, as for example $0 = 0$, it is removed from the system of equations, since it does not influence the solubility and solution of the system and therefore is redundant.

The handling of parameters with constrain conditions is somewhat more complex. Assumed, the symbols A and B were defined as parameters of a system of equations, which contains the equation

$$A + B = 1 \tag{2.1}$$

From this condition follows, that A and B are not *independent* parameters and therefore the system of equations is not solvable for any combinations of their values. Since conditioned inconsistencies of this type cannot be excluded with many technical problem settings, it is not meaningful to abort the solution process in such cases unless the parameterized equation

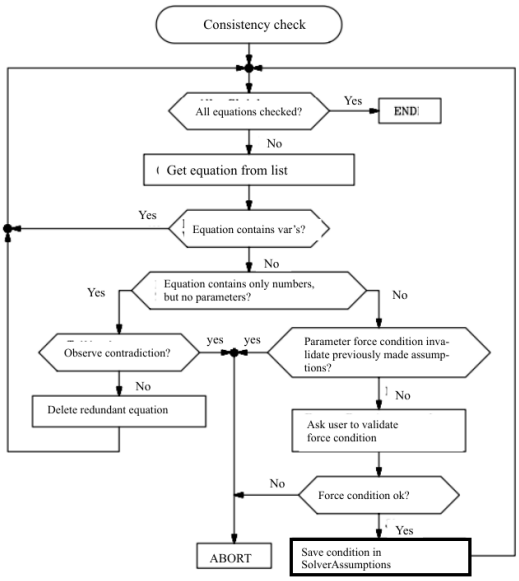


Figure 2.3:
Flow diagram of the routine for the consistency check

contradicts other constraints in the system of equations. The decision, whether a parameter constraint is to be regarded as admissible, is delegated therefore by the consistency check to the user. In the case of the equation (2.1) the system asks in the following way for the validity of the condition:

```
Is   B + A - 1   positive, negative, or zero?
```

Becomes the question answered with `p;` or `n;` (*positive* resp. *negative*), then the Solver aborts. If the response reads `z;` (*zero*), then the consistency check stores the constained condition in a global list named `SolverAssumptions`, which can be inspected after the Solver run. Subsequently, the consistency check removes the redundant equation from the system.

2.2.2 The Immediate Assignment Solver

The task of the *Immediate Assignment Solvers* is to search the system of equations for direct assignments of the form *var* = const. and to use such equations directly for the elimination of the corresponding variables. With systems of equations, set up mechanically, as for instance in the example 1.2, the cost of computation in *Linear Solver* for the extraction and solution of the linear equations can be often reduced considerably by such a preprocessing of the equations.

At first, the *Immediate Assignment Solvers* filters out all direct assignments *var* = const. and const. = *var* from the system in a loop and stores them in the form *var* = const. in the solution list, if a solution for *var* is not yet entered. Subsequently, the system of equations is analyzed by means of the solution list and checked again for consistency.

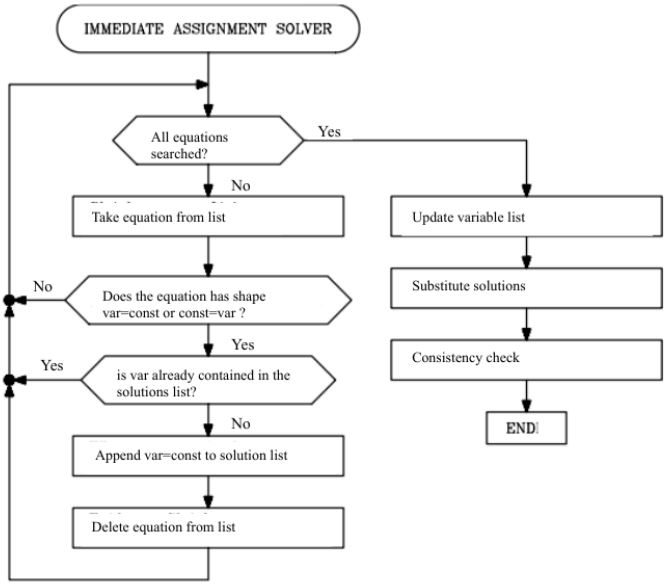


Figure 2.4:
Flow diagram of the Immediate Assignment Solver

2.2.3 The Linear Solver

The process of the *Linear Solvers* starts, as described in paragraph 1.4.2, with the construction of the complete coefficient matrix of the symbolic system of equations. Using this coefficient matrix, the valuation matrix is created, which serves for the extraction of parts of linear equations. As long as the valuation matrix contains still '1' elements, i.e. $\sum C \neq 0$ und $\sum R \neq 0$, the described heuristic evaluation strategy is applied, which decides, which nonlinear equation is removed or which in a nonlinear sense occurring variable is transferred to right hand side of the equation.

If the valuation matrix was reduced to a zero-matrix, then a list of the remaining (linear) equations and a list of the linear variables are created, which can be transferred as function parameters to the Maxima instruction `LINSOLVE`, which serves for the solution of a linear system of equations. For efficiency, before the call of `LINSOLVE` however, still another special feature is considered, which was not mentioned during the description of the extraction algorithm. Occasionally it can occur, that simultaneously with the deleting of a nonlinear equation the only instance of a linear variable x_j is removed from the entire valuation matrix, without that it was noticed directly. Likewise equations can develop, which are no longer nonlinear w.r.t as linear detected variables, but do not contain these variables any more, i.e. the associated coefficients are directly zero. Therefore, all linear variables of the linear subsystem are again checked, whether they are still contained in the linear equations, an The *Linear Solver* would also be able to solve the system of equations without these additional measures, but frequently unnecessary cost of computation can be saved by it.

In section 1.4.3, it was demonstrated, that while solving of over-determined linear systems of equations, inconsistencies can occur, which does not necessarily imply the insolubility of the system of equations. If such contradictory equations are detected, e.g. the right side of equation (1.71)[1], they are submitted a consistency checking, as at the beginning the entire system of equations. If the contradictory equations prove as true inconsistencies, then the total system does not have a solution, and the *Linear Solver* aborts with an appropriate error message. If the contradictory equations contain still looked-for variables, then they are removed from the system of equations and added as new constrained conditions to the remaining system. Thereupon `LINSOLVE` is again called with the linear independent part of the equations (inconsistencies can not occur any longer with the second run).

If the linear equations were successfully solved, then the solutions are inserted into the remaining system of equations and the list of the looked-for variables is updated. That is, all variables, for which a solution was found by the *Linear Solver*, are removed from the list, while new variables, which are contained in these solutions, are added to the list.

[1] That access to the contradictory equations from the outside is not possible using the standard `LINSOLVE` command. Therefore a particularly modified version of the instruction on Lisp level was necessary, which was made available by Jeffrey P. Golden, Macsyma, Inc. (the USA), on a kindly request.

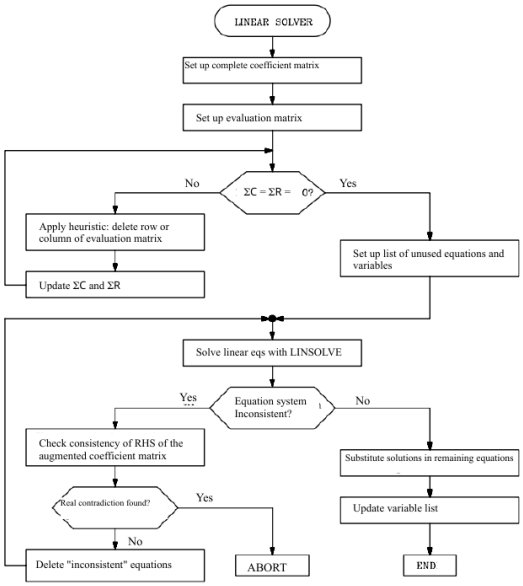


Figure 2.5:
Flow diagram of the Linear Solver

2.2.4 The Valuation Solver

The *Valuation Solver* first checks, whether equations and variables still are available, which can be solved. If this is the case, then for the remaining equations two valuation matrices are set up, which serve as basis for the solution sequence order. Both matrices have the dimensions $n \times m$, whereby n is the number of the equations and m the number of variables, looked for at the moment. The first matrix, the *path matrix of variables*, contains for each equation the number of paths for this variable (see section 1.5.2)) w.r.t each variable. The second matrix is the *valuation matrix*. Their entries are calculated by the heuristic operator tree valuations of each equation w.r.t. each variable.

To these two matrices one applies afterwards – depending on whether an internal or user-defined is desired – an valuation strategy, which arranges the pairs of equation/variables in such a way, that the first items in this list are the most promising candidates for a following solution attempt by means of the `SOLVE` command.

As long as not all proposals for solutions in the list were tried and no correct solution for one of the suggestions was calculated, on the basis the determined solution order the next equation from the equation list is selected and tried to solve. If this does not succeed, one or more user-defined transformation functions (if available) for the transformation of the equation are applied and in each case a solution attemptsare made. If these are also without result, the next proposal for a solution is tried.

If none of the equations is solvable after any variable any more, then the *Valuation Solver* returns the unresolved equations in implicit form additionally to all solutions found up to this point, so that these may be treated later with a numeric procedure. With a successful solution attempt all single solutions (nonlinear equations may have multiple solutions) are checked separately for consistency with the remaining equations. Those solutions, which lead to contradictory predicates, are rejected and with them the corresponding solution path.

If after the consistency check no solution remains, then the system of equations is inconsistent, and the *Valuation Solver* aborts. If exactly one solution remains, then this solution is appended to the solution list and substituted into the remaining equations. Finally the list of the wanted variables is updated, whilst the variable just calculated is removed from it and new variables, possibly contained in the solution, are added. These can be variables, which are not given as parameters and also not indicated as looked-for. The main solution loop of the *Valuation Solvers* begins then again from the start.

With consistent multiple solutions all solution paths must be pursued separately. Fot that the *Valuation Solver* calls itself recursively with the remaining equations and variables for each individual solution and stores the outputted results in a hierarchically structured solution list. Their expansion is task of the *Solver Postprocessors* described in the next section.

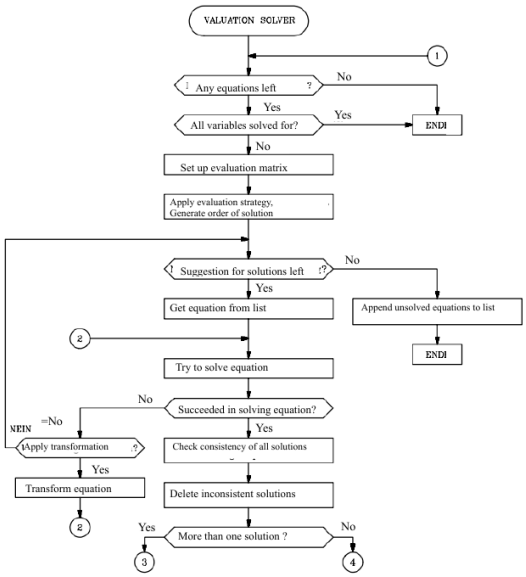


Figure 2.6:
Flow diagram of the Valuation Solver (part 1)

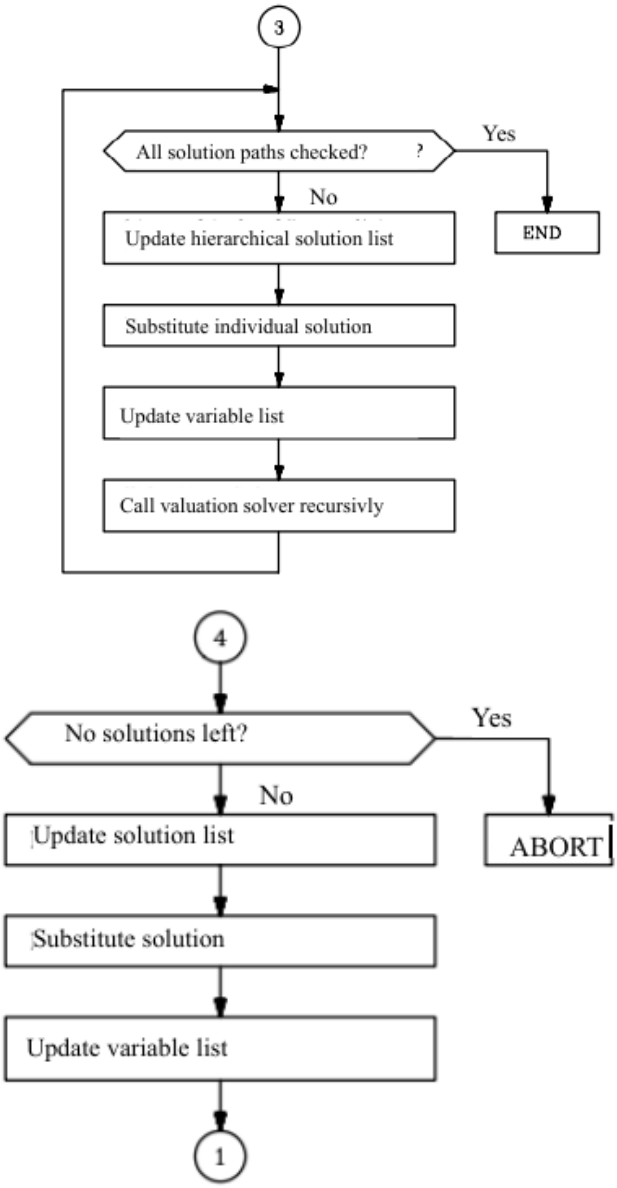


Figure 2.7:
Flow diagram of the Valuation Solver (part 2)

2.2.5 The Solver Postprocessor

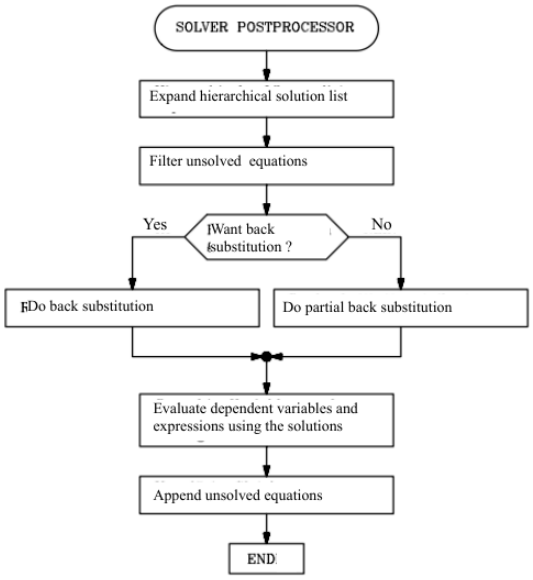


Figure 2.8:
Flow diagram of the Solver Postprocessor

The Valuation Solver supplies in the case of multiple solutions of nonlinear equations a hierarchically structured list of results (because of the recursive pursuit of the solution paths) as function value, therefore the *Solver Postprocessor* must dissolve the list hierarchy at the beginning, so that the back substitution can be executed.

If not all equations within a solution path could be solved symbolically, then the result list contains the list of the remaining, unresolved nonlinear equations additionally to the variables, for which a solution was found. These will provisionally removed from the equation list and buffered, in order to be added later to the final result.

Depending upon the desire of the user, the back substitution of the system of equations takes place afterwards, which is present up to this point still in upper triangle form. For that, the solution list is evaluated iterative with itself, until no modification of the results is to be seen any more. Even, if the back substitution is not required by the user[2], it is nevertheless executed, but only so far as necessary, in order to eliminate all not specified command line variables from the solutions. If e.g. a system of equations in the variables x, y, z and w is to be solved only after the variables x and y and the solution process gives the triangle form

$$x = f_1(y, z, w) \tag{2.2}$$
$$y = f_2(z, w) \tag{2.3}$$
$$w = f_3(z) \tag{2.4}$$
$$z = \text{const.}, \tag{2.5}$$

[2]This requires to set the option variable `SolverBacksubst` to `FALSE` (see section 2.4)

so the complete back substitution leads to the result

$$x = \text{const.} \tag{2.6}$$
$$y = \text{const.} \tag{2.7}$$

With switched off complete back substitution, the following two equations are given back

$$x = g(y) \tag{2.8}$$
$$y = \text{const.} \tag{2.9}$$

which does not contain the internal variables z und w any longer, but are further coupled by the variable y among themselves.

After termination of the back substitution and the evaluation of the composite terms in the variable list of the command line using the solutions, the at first filtered unresolved equations are again appended to the solution list. The finished solution list is then given to the user as function value.

2.3 Application of `Solver`

2.3.1 Command syntax

The call of `Solver` from the Maxima command line use the syntax

```
Solver( list_of_equations, list_of_variables, list_of_parameters )
```

or, if the system of equations, which is to be solved, does not contain any parameters, also with

```
Solver( list_of_equations, list_of_variables )
```

The list of equations is a list of Maxima objects, for which `EquationP` equals `TRUE`. The system of equations (1.55) – (1.57) is thus formulated in the following way:

```
(COM5) Equations :
[
    x + 2*y -     z =  6,
  2*x + y*z -   z^2 = -1,
  3*x -   y + 2*z^2 =  3
]$
```

The unknown variables are told to the Solver likewise in form of a list, e.g. as

```
[x, y, z]
```

for the above-mentioned systen of equations. Beside purely atomic variable symbols (`SymbolP`) the variable list may contain also composite terms in the looked-for unknowns. If for the system of equations the variables x, y and z are not explicitly searched, but rather x and the value $\sin(\pi yz)$, then the variable list reads

```
[x, sin(%pi*y*z)] .
```

The parameter list must contain exclusively atomic symbols, thus only objects with `SymbolP` equals `TRUE`. Composite terms are here neither admissible nor meaningfully.

2.3.2 Special Features of the Syntax of Equations

At this point we refer to some differences of the command syntax in comparison with the admissible invocations of the Maxima build-in `SOLVE` function. The latter may also get the equations in *Expression* form, i.e. as expressions `EquationP = FALSE`, which are implicitly understood as equations of the form *Expression* = 0:

```
[
    x + 2*y -     z - 6,  ← not admissable
  2*x + y*z -   z^2 + 1,
  3*x -   y + 2*z^2 - 3
]$
```

This form of representation of the equation is used internally, e.g. by the *Valuation Solver*, but is not permitted as call of the Solver in the command line. Furthermore, the `SOLVE` function permits omitting the brackets around the arguments, if only one equation and/or only one variable is to be transferred. This is also not admissible with the use of Solver.

2.3.3 Example Calls of Solver

Example 2.1.

For the input of the system of equations `COM5` a correct call of the Solver is the instruction in the command line `COM7`. The specification of the calculated solutions is done in form of a list of solution lists, see output line `D7`.

```
(COM6) MsgLevel : 'DETAIL$    /* see section 2.4 */

(COM7) Solver( Equations, [x, y, z] );
```

Output of *Solver Preprocessors*:

```
The variables to be solved for are [X, Y, Z]
Checking for inconsistencies...
... none found.
```

Output of *Immediate Assignment Solvers*:

```
Searching for immediate assignments.
No immediate assignments found.
```

Output of *Linear Solvers*:

```
Searching for linear equations...
  ...with respect to: [X, Y, Z]
Found 2 linear equations in 2 variables.
The variables to be solved for are [X, Y]
                                       2
The equations are [- Z + 2 Y + X - 6, 2 Z  - Y + 3 X - 3]
Solving linear equations.
                           2                 2
```

```
                          4 Z  - Z - 12      2 Z  + 3 Z + 15
The solutions are [X = - -------------, Y = ---------------]
                                7                   7
Searching for linear equations...
  ...with respect to: [Z]
No linear equations found.
```

Output of *Valuation Solvers*:

```
Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [Z]
Trying to solve equation 1 for Z
```

Here a complexity valuation Is not needed,
because there is only one equation and one variable left.

```
Valuation: (irrelevant)
                    3       2
The equation is 2 Z  - 12 Z  + 17 Z + 31 = 0
Checking if equation was solved correctly.
                          SQRT(13) %I - 7      SQRT(13) %I + 7
The solutions are [Z = - ---------------, Z = ---------------, Z = - 1]
                                 2                   2
Solution is correct.
```

Individual consistency check w.r.t. multiple solutions:

```
The solution is not unique. Tracing paths separately.
Solution 1 for Z
Checking for inconsistencies...
... none found.
Solution 2 for Z
Checking for inconsistencies...
... none found.
Solution 3 for Z
Checking for inconsistencies...
... none found.
                                      SQRT(13) %I - 7      SQRT(13) %I + 7
Consistent solutions for Z : [Z = - ---------------, Z = ---------------,
                                             2                   2

                                                                  Z = - 1]
```

Recursive pursuit of all three solution paths:

```
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
All variables solved for. No equations left.
```

Output of *Solver Postprocessors*:

```
Postprocessing results.

            27 SQRT(13) %I - 41         17 SQRT(13) %I - 87
(D7) [[X = -------------------, Y = - -------------------,
                    14                          14

      SQRT(13) %I - 7           27 SQRT(13) %I + 41       17 SQRT(13) %I + 87
Z = - ---------------], [X = - -------------------, Y = -------------------,
             2                           14                        14

    SQRT(13) %I + 7
Z = ---------------], [X = 1, Y = 2, Z = - 1]]
           2
```

For an better overview, here is the result again in TEX-output:

$$\left[x = \frac{27\sqrt{13}\,i - 41}{14},\ y = -\frac{17\sqrt{13}\,i - 87}{14},\ z = -\frac{\sqrt{13}\,i - 7}{2}\right] \tag{2.10}$$

$$\left[x = -\frac{27\sqrt{13}\,i + 41}{14},\ y = \frac{17\sqrt{13}\,i + 87}{14},\ z = \frac{\sqrt{13}\,i + 7}{2}\right] \tag{2.11}$$

$$[x = 1,\ y = 2,\ z = -1] \tag{2.12}$$

□

Example 2.2.

As second example, the following system of equations parameterized in a and b

$$3ax + y^2 = 1 \tag{2.13}$$

$$bx - y = -1 \tag{2.14}$$

is to be solved for the variables x and y as well as the composite term x/y . Since the system of equations does not contain any direct assignments and in this case a repeated search for linear equations is not meaningful, the *Immediate Assignment Solver* and the repetition loop of the *Linear Solvers* are switched off with the instruction `COM8`:

```
(COM8) SolverImmedAssign : SolverRepeatLinear : FALSE$

(COM9) ParEq : [ 3*a*x + y^2 =  1, b*x - y = -1 ]$

(COM10) Solver( ParEq, [x, y, x/y], [a, b] );

                                    X
The variables to be solved for are [X, -, Y]
                                    Y
```

```
The parameters are [A, B]
Checking for inconsistencies...
... none found.
Trying to solve for [X, Y]
                                     X
in order to solve for the expression -
                                     Y
Searching for linear equations...
  ...with respect to: [X, Y]
Found 1 linear equations in 2 variables.
The variables to be solved for are [X, Y]
The equations are [- Y + B X + 1]
Solving linear equations.
The solutions are [Y = B X + 1]

Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [X]
Trying to solve equation 1 for X
Valuation: (irrelevant)
                 2  2
The equation is B  X  + (2 B + 3 A) X = 0
Checking if equation was solved correctly.
                         2 B + 3 A
The solutions are [X = - ---------, X = 0]
                             2
                            B
Solution is correct.
The solution is not unique. Tracing paths separately.
Solution 1 for X
Checking for inconsistencies...
... none found.
Solution 2 for X
Checking for inconsistencies...
... none found.
                                    2 B + 3 A
Consistent solutions for X : [X = - ---------, X = 0]
                                        2
                                       B
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
All variables solved for. No equations left.
Postprocessing results.

              2 B + 3 A        B + 3 A  X   2 B + 3 A                    X
(D10) [[X = - ---------, Y = - -------, - = ----------], [X = 0, Y = 1, - = 0]]
                  2              B     Y    2                             Y
```

```
B                                B + 3 A B
```

□

2.4 The Options of `Solver`

Using the Maxima command line (`Macsyma toplevel`) or a program file, the behavior of the Solver can be influenced by the individual setting of a set of option variables, which are listed and described in the following. Behind the names of the option variables, the standard assignments (values/symbols) are given in angle parentheses, which are set automatically on the first start of the module `SOLVER`.

`MsgLevel <SHORT>` (*message level*) controls the scope of the displayed messages during the program run. The assignments `OFF`, `SHORT` and `DETAIL` are admissible.
Becomes `MsgLevel : OFF`, then all program outputs are completely suppressed. In the case of `SHORT` only status informations are returned whilst the program is running. The keyword `DETAIL` causes additionally the output of all of the Solver modules intermediate results and also of messages of the decisions made due to the heuristics.

`SolverImmedAssign <TRUE>` switches the *Immediate Assignment Solver* on (`TRUE`) resp. off (`FALSE`). If the module is switched on, then before the call of *Linear Solvers* the system of equations is searched for direct allocations of the form $var = \text{const.}$ or $\text{const.} = var$, which can be inserted immediately into the remaining equations.

`SolverRepeatImmed <TRUE>` determines whether the *Immediate Assignment Solver* is called repeatedly (`TRUE`), until no more direct assignments are found, or whether only one call takes place (`FALSE`).

`SolverSubstPowers <FALSE>` (*substitute powers*) controls the handling of variables, which occur in powers p_k of integer multiples $p_k = kp_0$, $k \in \mathbb{N}$, of a basic power $p_0 \in \mathbb{N} \setminus \{1\}$. If the system of equations contains e.g. the variable x exclusively in the powers x^2, x^4, x^6, ..., e.g. $p_0 = 2$, then by `SolverSubstPowers:TRUE` the term $x^{p_0} = x^2$ is substituted with the new variable symbol $X2$, that therfore only occurs in the powers $X2$, $X2^2$, $X2^3$, ... In this way the degree of the equations which are to be solved is reduced as well as the solution variety. However, if necessary, a rework of the solutions are necessary.

`SolverInconsParams <ASK>` (*inconsistent parameter handling*) influence the behavior of the routine for the consistency check. Admissible parameters are `ASK`, `BREAK` and `IGNORE`. If during the consistency check a dependency between the parameters is discovered, then with `SolverInconsParams : ASK` the users is asked for the validity of the appropriate constrained condition. With a positive response, this is stored for later evalutaion in the list `SolverAssumptions`. If the dependency is not admissible or if the option variable is set with `BREAK`, then the solution process is aborted. If the parameter is set to `IGNORE`, then the consistency check is caused to accept basically all constrained conditions between parameters as valid as long as these do not contradict directly already made conditions.

`SolverLinear <TRUE>` switches the *Linear Solver* on (`TRUE`) resp. off (`FALSE`). Switching off is recommended if the system of equations, which is to be solved, does not contain linear equations or their number is very small in relation to the number of nonlinear equations.

In these cases much computing time can be saved through bypass the *Linear Solvers*, since the algorithm to search for linear equations is quite complex.

`SolverRepeatLinear <TRUE>` cause repeated calls of *Linear Solvers*. If the variable is set to `FALSE`, then *Linear Solver* is executed only once.

`SolverFindAllLinearVars <TRUE>` decides whether *Linear Solver* that looks for maximal large pieces of linear equations regarding all available variables (`TRUE`), or whether only subsystems in the variables are to be extracted, which are immediately looked for during the solution process (`FALSE`). The setting of the variables plays especially a role, if underdetermined systems of equations are to be solved.
Here `SolverFindAllLinearVars : FALSE` should to be set, because otherwise no solutions for the originally interesting variables could possibly be found due to the too small number of equations. With `FALSE` it is guaranteed that at first after these variables is solved and the degrees of freedom are expressed in the other unknowns.

`SolverValuationStrategy <MinVarPathsFirst>` contains the name of the function, which is generate a solution order from the variable path matrix and the valuation matrix (see section 2.7). The call of the function within the *Valuation Solvers* is done with: des *Valuation Solvers* mit:

```
SolveOrder : Apply(
  SolverValuationStrategy, [ VarPathMatrix, ValuationMatrix ]
)
```

As function value a list of the form

```
[ [i1, j1, b1], [i2, j2, b2], ... ]
```

is expected, which was described in section 1.5.2.
To observe here is the option variable `SolverMaxLenValOrder`.

`SolverDefaultValuation <10>` determines the valuation factor for operators, which were not explicitly assigned such a factor with the `SetProp` instruction (see section 2.6).

`SolverMaxLenValOrder <5>` (*maximum length of valuation order*) determines the maximal length of the solution order. If the last proposal for solution in the list does not lead to success, then the *Valuation Solver* aborts the solution process, even if not all pairs of equation/variables were tried.

`SolverTransforms <[]>+` contains a list of the names of functions, which can be applied after a unsuccessful solution attempt to the corresponding equation, in order to increase the solution chances with a renewed attempt. Thereby the functions are executed in the order of their occurring in the list. After each function call the next solution attempt takes place directly. If this fails, then the next transformation in the list is applied, as long as the equation could be solved or no further transformation is available. Since the possibilities for manipulations with the transformations are quite extensive, a more accurate specification of their definition and application is in section 2.5.

`SolverPostprocess <TRUE>` switches the *Solver Postprocessor* on (`TRUE`) resp. off (`FALSE`). If switched to status off, the hierarchical result list of the Solver is returned without rework, e.g. without expansion, back substitution and extraction of the variables requested by the user. This control variable serves primarily for Debug purposes.

`SolverBacksubst <TRUE>` (*backsubstitution*) determines whether the *Solver Postprocessor* is to execute a back substitution of the system of equations brought on triangle form (`TRUE`) or not (`FALSE`). If the calculated symbolic solutions are very extensive, then it is often meaningful to execute no complete back substitution, but to output some of the looked-for variables as functions of other calculated unknowns.

`SolverDispAllSols <FALSE>` (*display all solutions*) instructs the Solver, if set to `TRUE`, to output *all* found solutions at the termination of the solution process and not only for the variables indicated by the user.

`SolverRatSimpSols <TRUE>` (*perform rational simplifications on solutions*) instructs the *Solver Postprocessor* to simplify the results with the instruction `FullRatSimp` before the output.

`SolverDumpToFile <FALSE>` (`TRUE`) instructs the *Valuation Solver*, to write all found solutions (after each successful solution attempt) as well as the remaining equations and variables into a Maxima batch file, whose name is saved in the option variable `SolverDumpFile`. This option is intended for extensive problems, where Maxima is inclined to system crashes because of acute lack of main and swap memory. Within a crash, at least a part of the results can be saved by the storage of the intermediate results.

`SolverDumpFile <"SOLVER.DMP">` contains the name of the file, into which the intermediate results are to be written.

2.5 Definition and Integration of User Specific Transformation Routines

Often the *Valuation Solver* encounters equations during the solution process, which it is not able to solve, although already some simple rearrangements or simplifications could help, in order to receive the desired result. E.g. the `SOLVE` function is not able to solve the equation

$$x + \sin^2 x + \cos^2 x = 1 \tag{2.15}$$

correctly for x, since it does not know that the square terms of sine and cosine can be combined easyly into '1'.

In order to be able to execute rearrangements or simplifications of the equations depending upon the application case, the dynamic integration of user-defined transformation functions is bulid in the Solver, which are applied to unresolved equations if necessary. The *Valuation Solver* transfers to these functions among other things the equation and the variable, after which the equation is to be solve. The transformation function has now the task to transform the equation and return it as its function value again to the Solver, so that a renewed solution attempt can begin.

The call of a transformation function from the Solver works in the following way:

```
Transform : CopyList( SolverTransforms ),
  ⋮
SOLVER LOOP
     ⋮
   Trans : Pop( Transform ),
   TransEq : Apply( Trans, [ Equation, Variable, Solution ] )
     ⋮
LOOP END
```

Thereby the additionally transferred parameter `Solution` contains the result of the failed solution attempt, on the basis of which possibly helpful conclusions can be drawn. The following instruction shows an example of the definition of a simple, user specific transformation routine, which tries to make an equation solvable by simplifying of trigonometric functions:

```
TransformTrig( Equation, Variable, Solution ) :=
  TrigSimp( Equation )$
```

For the integration of the function their name had to be inserted into the global list `SolverTransforms`:

```
SolverTransforms : [ 'TransformTrig ]$
```

Also the application of a transformation routine may be unsuccessful, therefore it exists the possibility to remember the Solver the failure of the simplifying attempt by return of an empty list (`[]`) as function value. In this way it is prevented that a further call of the `SOLVE` function with the same equation takes place. Instead the Solver tries directly, to execute the next transformation in the list, if available,. For an alternative to the rearranging of the equation and an afterward solution by the *internal* Solver, a transformation routine is also allowed, to determine the solutions of the transferred equation *itself* and return them in the form

```
[ Variable = Solution_1, Variable = Solution_2, ... ]
```

A possible area of application for such functions would be the application of numeric procedures for the solution of nonlinear equations, which contain only one variable and no parameters. Or the equation can be manipulated by hand, whilst the transformation routine calls a *Maxima-Break*. The method for the definition and integration of transformation functions with success return, wil be demonstrated in the following completely worked example. To solve is the system of equations (2.16) – (2.18) for the variables x, y and z. This task is quite simple in principle, but nevertheless it causes substantial difficulties to the Solver.

$$z - \sin x = 0 \tag{2.16}$$

$$y + z^2 + \cos x^2 = 5 \tag{2.17}$$

$$y + x = 1 \tag{2.18}$$

At first, the system of equations is input as an equation list as well as the variables:

```
(COM11) TrigEq :
[
          z - sin(x) = 0,
  y + z^2 + cos(x)^2 = 5,
               y + x = 1
]$

(COM12) Var : [x, y, z]$
```

The first solution attempt is to be executed without transformation functions:

```
(COM13) SolverTransforms : []$

(COM14) Solver( TrigEq, Var );

The variables to be solved for are [X, Y, Z]
Checking for inconsistencies...
... none found.

Searching for linear equations...
  ...with respect to: [X, Y, Z]
Found 2 linear equations in 2 variables.
The variables to be solved for are [Y, Z]
The equations are [Z - SIN(X), Y + X - 1]
Solving linear equations.
The solutions are [Y = 1 - X, Z = SIN(X)]

Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [X]
Trying to solve equation 1 for X
Valuation: (irrelevant)
                   2         2
The equation is SIN (X) + COS (X) - X - 4 = 0
Checking if equation was solved correctly.
                          2         2
The solutions are [X = SIN (X) + COS (X) - 4]
Solution is not correct.
Cannot solve equation. Giving up.

Postprocessing results.
Cannot determine an explicit solution for X

                                        2         2
(D14)        [[Y = 1 - X, Z = SIN(X), [SIN (X) + COS (X) - X - 4]]]
```

In this case the Solver is not able to solve the equation with the squared sine and cosine terms for the variable x and therefore returns the unresolved equation in implicit form as last

item of the solution list. To simplifyf the equations two transformation functions are now to be defined and merged in. The first transformation serves for the simplification of logarithm terms, the second for the simplification of terms, which contain trigonometric functions. Both functions check whether their application was successful and returns an empty list, if the transformation did not cause any modification of the equation.

```
(COM15) TransformLog( Equation, Variable, Solution ) := BLOCK(
  [ Eq ],
  Eq : LogContract( Equation ),
  IF Eq = Equation THEN
    RETURN( [] )
  ELSE
    RETURN( Eq )
)$

(COM16) TransformTrig( Equation, Variable, Solution ) := BLOCK(
  [ Eq ],
  Eq : TrigSimp( Equation ),
  IF Eq = Equation THEN
    RETURN( [] )
  ELSE
    RETURN( Eq )
)$
```

The transformation function `TransformLog` may be applied basically as first, therefore the integration of the routines takes place in the following order:

```
(COM17) SolverTransforms : [ 'TransformLog, 'TransformTrig ]$
```

Now a new run of Solver may be started.

```
(COM18) Solver( TrigEq, Var );

The variables to be solved for are [X, Y, Z]
Checking for inconsistencies...
... none found.

Searching for linear equations...
  ...with respect to: [X, Y, Z]
Found 2 linear equations in 2 variables.
The variables to be solved for are [Y, Z]
The equations are [Z - SIN(X), Y + X - 1]
Solving linear equations.
The solutions are [Y = 1 - X, Z = SIN(X)]

Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [X]
```

```
Trying to solve equation 1 for X
Valuation: (irrelevant)
The equation is SIN (X) + COS (X) - X - 4 = 0
Checking if equation was solved correctly.
                          2         2
The solutions are [X = SIN (X) + COS (X) - 4]
Solution is not correct.
```

Here the Solver recognize as in the preceded case that it cannot solve the equation and applies the first transformation in the list to it.

```
Applying transformation TRANSFORMLOG
Transformation failed.
```

Since the equation does not contain logarithmic functions, the simplifying attempt is not successful. Therefore the second transformation routine is tried.

```
Applying transformation TRANSFORMTRIG
The transformation yields - X - 3 = 0
Retrying with transformed equation.
Checking if equation was solved correctly.
The solutions are [X = - 3]
Solution is correct.

Solution 1 for X
Checking for inconsistencies...
... none found.
Consistent solutions for X : [X = - 3]
Checking for remaining equations.
All variables solved for. No equations left.
Postprocessing results.

(D18)                    [[X = - 3, Y = 4, Z = - SIN(3)]]
```

The transformation function `TransformTrig` succeeds to simplfy the equation, so that the Solver is now able, to determine the correct solution for the variable x.

2.6 Modification of the Operator Valuations

To give the user of the Solver the possibility for an individual intervention of the heuristic complexity valuation of algebraic terms, the valuations of the arithmetic operators are stored as *Macsyma Properties* under the keyword `Valuation`, and not as immutable constants. The definition of a valuation factor takes place with the instruction

`SetProp(` *Operator*, `'Valuation,` *Valuation factor* `)$`

The standard mappings of the valuation factors are pre-defined in the Solver:

```
SetProp( 'sin,   'Valuation, 10 )$
SetProp( 'cos,   'Valuation, 10 )$
SetProp( 'tan,   'Valuation, 10 )$
SetProp( 'asin,  'Valuation, 12 )$
SetProp( 'acos,  'Valuation, 12 )$
SetProp( 'atan,  'Valuation, 12 )$
SetProp( 'sinh,  'Valuation, 12 )$
SetProp( 'cosh,  'Valuation, 12 )$
SetProp( 'tanh,  'Valuation, 12 )$
SetProp( 'asinh, 'Valuation, 12 )$
SetProp( 'acosh, 'Valuation, 12 )$
SetProp( 'atanh, 'Valuation, 12 )$
SetProp( "+",    'Valuation,  1 )$
SetProp( "-",    'Valuation,  1 )$
SetProp( "*",    'Valuation,  4 )$
SetProp( "/",    'Valuation,  4 )$
SetProp( "^",    'Valuation, 10 )$
SetProp( 'exp,   'Valuation, 10 )$
SetProp( 'log,   'Valuation, 10 )$
SetProp( 'sqrt,  'Valuation, 10 )$
```

E.g. if the valuation of the tanh–operator should be modified on 20, then this can be done at any time by

```
SetProp( 'tanh, 'Valuation, 20 )$
```

A quality factor can queried with the `Get` command:

```
Get( Operator, 'Valuation );
```

Example: The valuation of "$*$"– operator is determined through:

```
(COM19) Get( "*", 'Valuation );

(D19)                                4
```

If the result of the query has the value `FALSE`, then this means that no valuation factor for the operator concerned was defined.

2.7 Definition and Integration of User Specific Valuation Strategies

By means of a reassignment of the procedure variable `SolverValuationStrategy` the *Valuation Solver* is caused to use a user-defined valuation strategy to the determination of the solution order instead of the internal function. The function are given by their call two valuation matrices,

```
SolveOrder : Apply(
  SolverValuationStrategy, [ VarPathMatrix, ValuationMatrix ]
)
```

which are the *variable path matrix* and the *complexity valuation matrix*, whose row dimension equal the number of the equations and their column dimension equals the number of variables which are be determined.

The entry at the position (i, j) of the *variable path matrix* corresponds to the number of paths to the instances of the variable x_j in equation i (see section 1.5.2). The *complexity evaluation matrix* contains at the position (i, j) the complexity valuation of the equation i w.r.t. the variable x_j. Thus, for the system of equations (1.55) – (1.57) the path matrix reads

$$\begin{array}{c} \\ (1.55) \\ (1.56) \\ (1.57) \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

and the valuation matrix using the standard operator valuation factors are determined to

$$\begin{array}{c} \\ (1.55) \\ (1.56) \\ (1.57) \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{pmatrix} 1 & 4 & 1 \\ 4 & 4 & 14 \\ 4 & 1 & 40 \end{pmatrix} \end{array}$$

From these matrices the valuation strategy must generate a solution order (see section 1.5.2)) in the form

```
[ [i1, j1, b1], [i2, j2, b2], ...  ]
```

The maximal length of the list should not be larger than the value of the option variable `SolverMaxLenValOrder`.

The basic structure of an valuation strategy is given by the following pseudocode:

```
MyValuationStrategy( PathMat, ValMat ) := BLOCK(

  generate a solution order from the valuation strategies
  limit the length of the list to SolverMaxLenValOrder  elements

  RETURN( the solution order )
)$
```

To insert the function afterwards, the procedure variable `SolverValuationStrategy` is to assign with the function name:

```
SolverValuationStrategy : 'MyValuationStrategy$
```

Bibliography

[ADA_79] D. Adams, *The Hitchhiker's Guide to the Galaxy*, London: Pan Books, 1979

[BRO_88] E. Brommundt, G. Sachs, *Technische Mechanik — Eine Einf"uhrung*, Berlin: Springer Verlag, 1988

[FOU_92] L. R. Foulds, *Graph Theory Applications*, Berlin, New York, Tokyo: Springer Verlag, 1992

[HEN_93] E. Hennig, "Mathematische Grundlagen und Implementation eines symbolischen Matrixapproximationsalgorithmus mit quantitativer Fehlervorhersage", Diplomarbeit am Institut f"ur Netzwerktheorie und Schaltungstechnik, Technische Universit"at Braunschweig, August 1993

[MAC_94] Macsyma Inc., *Macsyma Reference Manual V14*, Arlington, 1994

[MIC_94] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Berlin, New York, Tokyo: Springer Verlag, 1994

[N"UH_89] D. N"uhrmann, *Professionelle Schaltungstechnik*, M"unchen: Franzis Verlag, 1989

[PFA_94] J. Pfalzgraf, "Some Robotics Benchmarks for Computer Algebra", Vortrag auf der GAMM-Jahrestagung 1994, Braunschweig, April 1994

[SOM_93a] R. Sommer, "Konzepte und Verfahren zum rechnergest"utzten Entwurf analoger Schaltungen", Dissertation am Institut f"ur Netzwerktheorie und Schaltungstechnik, Technische Universit"at Braunschweig, Juli 1993

[SOM_93b] R. Sommer, E. Hennig, G. Dr"oge, E.-H. Horneber, "Equation-based Symbolic Approximation by Matrix Reduction with Quantitative Error Prediction", *Alta Frequenza — Rivista di Elettronica*, Vol. 5, No. 6, Dez. 1993, S. 29 – 37

[TRI_91] H. Trispel, "Symbolische Schaltungsanalyse mit Macsyma", Studienarbeit am Institut f"ur Netzwerktheorie und Schaltungstechnik, Technische Universit"at Braunschweig, Juli 1991

[WOL_91] S. Wolfram, *Mathematica — A System for Doing Mathematics by Computer*, Addison Wesley, 1991

Appendix A

Examples

A.1 Truss Design

For the demonstration of the efficiency of the Solver finally the examples stated in the introduction 1.1 and 1.2 are solved. In the following the complete display of the outputs of the two program runs are reported. We use the following lexicon:

LEXICON:	*German*	English
	Stabzweischlags	two-bar truss
	Zweischlag	two-bar
	Stababmessungen	bar dimensions
	Zweischlagparameter	two-bar parameters

```
(COM1) Zweischlag :
[
  F*cos(gamma) - S1*cos(alpha) - S2*cos(beta) = 0,
  F*sin(gamma) - S1*sin(alpha) + S2*sin(beta) = 0,
  Delta_l1 = l1*S1/(E*A1),
  Delta_l2 = l2*S2/(E*A2),
  l1 = c/cos(alpha),
  l2 = c/cos(beta),
  a = Delta_l2/sin(alpha+beta),
  b = Delta_l1/sin(alpha+beta),
  u = a*sin(alpha) + b*sin(beta),
  w = -a*cos(alpha) + b*cos(beta),
  A1 = h1^2,
  A2 = h2^2
]$

(COM2) Stababmessungen : [h1, h2]$

(COM3) Zweischlagparameter : [alpha, beta, gamma, F, c, E, u, w]$

(COM4) SolverRepeatImmed : SolverRepeatLinear : FALSE$
```

```
(COM5) MsgLevel : 'DETAIL$

(COM6) Solver( Zweischlag, Stababmessungen, Zweischlagparameter );
The variables to be solved for are [H1, H2]
The parameters are [ALPHA, BETA, C, E, F, U, W, GAMMA]
Checking for inconsistencies...
... none found.
Searching for immediate assignments.
                  C
Assigning L1 = ----------
               COS(ALPHA)
                  C
Assigning L2 = ---------
               COS(BETA)
Checking for inconsistencies...
... none found.
Searching for linear equations...
  ...with respect to: [H1, H2, A, A1, A2, B, DELTA_L1, DELTA_L2, S1, S2]
Found 7 linear equations in 7 variables.
The variables to be solved for are [A, A1, B, DELTA_L1, DELTA_L2, S1, S2]
The equations are [F COS(GAMMA) - COS(BETA) S2 - COS(ALPHA) S1,

F SIN(GAMMA) + SIN(BETA) S2 - SIN(ALPHA) S1,

A SIN(BETA + ALPHA) - DELTA_L2, B SIN(BETA + ALPHA) - DELTA_L1,

                                                                    2
U - B SIN(BETA) - A SIN(ALPHA), W - B COS(BETA) + A COS(ALPHA), A1 - H1 ]
Solving linear equations.
                                COS(BETA) U - SIN(BETA) W
The solutions are [A = ------------------------------------------,
                       COS(ALPHA) SIN(BETA) + SIN(ALPHA) COS(BETA)

       2             SIN(ALPHA) W + COS(ALPHA) U
A1 = H1 , B = ----------------------------------------,
              COS(ALPHA) SIN(BETA) + SIN(ALPHA) COS(BETA)

DELTA_L1 = (SIN(ALPHA) SIN(BETA + ALPHA) W

 + COS(ALPHA) SIN(BETA + ALPHA) U)/(COS(ALPHA) SIN(BETA)

 + SIN(ALPHA) COS(BETA)), DELTA_L2 =

COS(BETA) SIN(BETA + ALPHA) U - SIN(BETA) SIN(BETA + ALPHA) W
-----------------------------------------------------------,
         COS(ALPHA) SIN(BETA) + SIN(ALPHA) COS(BETA)
```

```
     COS(BETA) F SIN(GAMMA) + SIN(BETA) F COS(GAMMA)
S1 = ----------------------------------------------,
       COS(ALPHA) SIN(BETA) + SIN(ALPHA) COS(BETA)

     SIN(ALPHA) F COS(GAMMA) - COS(ALPHA) F SIN(GAMMA)
S2 = ----------------------------------------------]
        COS(ALPHA) SIN(BETA) + SIN(ALPHA) COS(BETA)
Checking for inconsistencies...
... none found.
Checking for remaining equations.
3 equation(s) and 2 variable(s) left.
The variables to be solved for are [H1, H2]
Applying valuation strategy.
Trying to solve equation 3 for H2
Valuation: 10
                      2
The equation is A2 - H2  = 0
Checking if equation was solved correctly.
The solutions are [H2 = - SQRT(A2), H2 = SQRT(A2)]
Solution is correct.
The solution is not unique. Tracing paths separately.
Solution 1 for H2
Checking for inconsistencies...
... none found.
Solution 2 for H2
Checking for inconsistencies...
... none found.
Consistent solutions for H2 : [H2 = - SQRT(A2), H2 = SQRT(A2)]
Checking for remaining equations.
2 equation(s) and 2 variable(s) left.
The variables to be solved for are [A2, H1]
Applying valuation strategy.
Trying to solve equation 2 for A2
Valuation: 8
The equation is COS(ALPHA) C F SIN(GAMMA) - SIN(ALPHA) C F COS(GAMMA)

 - A2 COS(BETA) SIN(BETA) SIN(BETA + ALPHA) E W

         2
 + A2 COS (BETA) SIN(BETA + ALPHA) E U = 0
Checking if equation was solved correctly.
The solutions are [A2 = (COS(ALPHA) C F SIN(GAMMA)

 - SIN(ALPHA) C F COS(GAMMA))/(COS(BETA) SIN(BETA) SIN(BETA + ALPHA) E W

      2
 - COS (BETA) SIN(BETA + ALPHA) E U)]
Solution is correct.
```

```
Solution 1 for A2
Checking for inconsistencies...
... none found.
Consistent solutions for A2 : [A2 = (COS(ALPHA) C F SIN(GAMMA)

 - SIN(ALPHA) C F COS(GAMMA))/(COS(BETA) SIN(BETA) SIN(BETA + ALPHA) E W

      2
 - COS (BETA) SIN(BETA + ALPHA) E U)]
Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [H1]
Trying to solve equation 1 for H1
Valuation: (irrelevant)
The equation is - COS(BETA) C F SIN(GAMMA) - SIN(BETA) C F COS(GAMMA)

                                                 2
 + COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E H1  W

      2                                      2
 + COS (ALPHA) SIN(BETA + ALPHA) E H1  U = 0
Checking if equation was solved correctly.
The solutions are [H1 = - SQRT(COS(BETA) C F SIN(GAMMA)

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H1 = SQRT(COS(BETA) C F SIN(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA)

          2
 E W + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U))]
Solution is correct.
The solution is not unique. Tracing paths separately.
Solution 1 for H1
Checking for inconsistencies...
... none found.
```

```
Solution 2 for H1
Checking for inconsistencies...
... none found.
Consistent solutions for H1 : [H1 = - SQRT(COS(BETA) C F SIN(GAMMA)

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H1 = SQRT(COS(BETA) C F SIN(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA)

          2
 E W + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U))]
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
2 equation(s) and 2 variable(s) left.
The variables to be solved for are [A2, H1]
Applying valuation strategy.
Trying to solve equation 2 for A2
Valuation: 8
The equation is COS(ALPHA) C F SIN(GAMMA) - SIN(ALPHA) C F COS(GAMMA)

 - A2 COS(BETA) SIN(BETA) SIN(BETA + ALPHA) E W

         2
 + A2 COS (BETA) SIN(BETA + ALPHA) E U = 0
Checking if equation was solved correctly.
The solutions are [A2 = (COS(ALPHA) C F SIN(GAMMA)

 - SIN(ALPHA) C F COS(GAMMA))/(COS(BETA) SIN(BETA) SIN(BETA + ALPHA) E W

      2
 - COS (BETA) SIN(BETA + ALPHA) E U)]
Solution is correct.
```

```
Solution 1 for A2
Checking for inconsistencies...
... none found.
Consistent solutions for A2 : [A2 = (COS(ALPHA) C F SIN(GAMMA)

 - SIN(ALPHA) C F COS(GAMMA))/(COS(BETA) SIN(BETA) SIN(BETA + ALPHA) E W

      2
 - COS (BETA) SIN(BETA + ALPHA) E U)]
Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [H1]
Trying to solve equation 1 for H1
Valuation: (irrelevant)
The equation is - COS(BETA) C F SIN(GAMMA) - SIN(BETA) C F COS(GAMMA)

                                                 2
 + COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E H1  W

      2                                      2
 + COS (ALPHA) SIN(BETA + ALPHA) E H1  U = 0
Checking if equation was solved correctly.
The solutions are [H1 = - SQRT(COS(BETA) C F SIN(GAMMA)

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H1 = SQRT(COS(BETA) C F SIN(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA)

          2
 E W + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U))]
Solution is correct.
The solution is not unique. Tracing paths separately.
Solution 1 for H1
Checking for inconsistencies...
... none found.
```

```
Solution 2 for H1
Checking for inconsistencies...
... none found.
Consistent solutions for H1 : [H1 = - SQRT(COS(BETA) C F SIN(GAMMA)

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H1 = SQRT(COS(BETA) C F SIN(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA)

          2
 E W + COS (ALPHA) SIN(BETA + ALPHA) E U)

 + SIN(BETA) C F COS(GAMMA)/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U))]
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
All variables solved for. No equations left.
Postprocessing results.

(D6) [[H1 = - SQRT((COS(BETA) C F SIN(GAMMA) + SIN(BETA) C F COS(GAMMA))

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H2 = - SQRT((SIN(ALPHA) C F COS(GAMMA) - COS(ALPHA) C F SIN(GAMMA))

     2
/(COS (BETA) SIN(BETA + ALPHA) E U - COS(BETA) SIN(BETA) SIN(BETA + ALPHA)

 E W))], [H1 = SQRT((COS(BETA) C F SIN(GAMMA) + SIN(BETA) C F COS(GAMMA))

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),
```

```
H2 = - SQRT((SIN(ALPHA) C F COS(GAMMA) - COS(ALPHA) C F SIN(GAMMA))

     2
/(COS (BETA) SIN(BETA + ALPHA) E U - COS(BETA) SIN(BETA) SIN(BETA + ALPHA)

 E W))], [H1 = - SQRT((COS(BETA) C F SIN(GAMMA) + SIN(BETA) C F COS(GAMMA))

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H2 = SQRT((SIN(ALPHA) C F COS(GAMMA) - COS(ALPHA) C F SIN(GAMMA))

     2
/(COS (BETA) SIN(BETA + ALPHA) E U - COS(BETA) SIN(BETA) SIN(BETA + ALPHA)

 E W))], [H1 = SQRT((COS(BETA) C F SIN(GAMMA) + SIN(BETA) C F COS(GAMMA))

/(COS(ALPHA) SIN(ALPHA) SIN(BETA + ALPHA) E W

      2
 + COS (ALPHA) SIN(BETA + ALPHA) E U)),

H2 = SQRT((SIN(ALPHA) C F COS(GAMMA) - COS(ALPHA) C F SIN(GAMMA))

     2
/(COS (BETA) SIN(BETA + ALPHA) E U - COS(BETA) SIN(BETA) SIN(BETA + ALPHA)

 E W))]]
```

For the system of equations, which describe the 'Truss' the Solver gives four distinct analytical solutions, which are different only in their signs. Due to the physical boundary condition, that the lengths h_1 and h_2 cannot have negative values, only the last solutionis meaningful

$$h_1 = \sqrt{\frac{\cos\beta\, c\, F\ \sin\gamma + \sin\beta\, c\, F\ \cos\gamma}{\cos\alpha\ \sin\alpha\ \sin\left(\beta+\alpha\right)\ E\, w + \cos^2\alpha\ \sin\left(\beta+\alpha\right)\ E\, u}} \tag{A.1}$$

$$h_2 = \sqrt{\frac{\sin\alpha\, c\, F\ \cos\gamma - \cos\alpha\, c\, F\ \sin\gamma}{\cos^2\beta\ \sin\left(\beta+\alpha\right)\ E\, u - \cos\beta\ \sin\beta\ \sin\left(\beta+\alpha\right)\ E\, w}} \tag{A.2}$$

A.2 Transistor Amplifier Design

We use the following

LEXICON:	*German*	English
	Verstaerker	amplifier
	Widerstaende	resistances
	Designparameter	design parameter

```
(COM7) Verstaerker :
[
  -V_V1+V_R1+V_OC_1+V_FIX2_Q1 = 0,-V_V1+V_R2+V_OC_1+V_I1+V_FIX2_Q1 = 0,
  -V_V1+V_OC_1+V_I1+V_FIX2_Q2+V_FIX2_Q1-V_FIX1_Q2-V_FIX1_Q1 = 0,
  V_V1-V_R6+V_R3-V_OC_1-V_I1-V_FIX2_Q1+V_FIX1_Q1 = 0,
  -V_V1+V_R7+V_R4+V_OC_1+V_I1-V_FIX1_Q2 = 0,-V_V1+V_R7+V_R5+V_OC_1 = 0,
  V_OC_2-V_I1+V_FIX1_Q2 = 0,V_V_CC-V_V1+V_R6+V_OC_1+V_FIX2_Q1-V_FIX1_Q1 = 0,
  I_V1+I_OC_1 = 0,I_R7-I_OC_1+I_FIX2_Q1 = 0,I_R2+I_R1-I_FIX2_Q1-I_FIX1_Q1 = 0,
  I_R6+I_FIX2_Q2+I_FIX1_Q1 = 0,-I_R7+I_R5+I_R4 = 0,
  -I_R4+I_OC_2-I_FIX2_Q2-I_FIX1_Q2 = 0,I_R3-I_R2+I_I1+I_FIX1_Q2 = 0,
  I_V_CC-I_R6-I_R3 = 0,V_V1 = 0,I_I1 = 0,I_OC_1 = 0,V_FIX1_Q1 = 2.72,
  V_FIX2_Q1 = 0.607,I_R1*R1-V_R1 = 0,I_R7*R7-V_R7 = 0,I_R2*R2-V_R2 = 0,
  I_R6*R6-V_R6 = 0,V_FIX1_Q2 = 6.42,V_FIX2_Q2 = 0.698,I_R3*R3-V_R3 = 0,
  I_R4*R4-V_R4 = 0,I_R5*R5-V_R5 = 0,I_OC_2 = 0,V_V_CC = VCC,
  I_FIX1_Q1 = 1.11e-4,I_FIX2_Q1 = 5.75001e-7,I_FIX1_Q2 = 0.00401,
  I_FIX2_Q2 = 1.26e-5,
  A = 145303681853*R2/(145309663773*R1),
  ZIN = R7,
  ZOUT = (1675719398828125*R2*R7+394048139880824192*R1*R2)
         /(1365528906303031214O8*R1)
]$

(COM8) Widerstaende : [R1, R2, R3, R4, R5, R6, R7]$

(COM9) Designparameter : [VCC, A, ZIN, ZOUT]$

(COM10) SolverRepeatImmed : SolverRepeatLinear : TRUE$

(COM11) Solver( Verstaerker, Widerstaende, Designparameter );
The variables to be solved for are [R1, R2, R3, R4, R5, R6, R7]
The parameters are [A, VCC, ZIN, ZOUT]
Checking for inconsistencies...
... none found.
Searching for immediate assignments.
Assigning V_V1 = 0
Assigning I_I1 = 0
Assigning I_OC_1 = 0
                      68
Assigning V_FIX1_Q1 = --
```

```
                      25
                      607
Assigning V_FIX2_Q1 = ----
                      1000
                      321
Assigning V_FIX1_Q2 = ---
                      50
                      349
Assigning V_FIX2_Q2 = ---
                      500
Assigning I_OC_2 = 0
Assigning V_V_CC = VCC
                        111
Assigning I_FIX1_Q1 = -------
                      1000000
                          5
Assigning I_FIX2_Q1 = -------
                      8695637
                       401
Assigning I_FIX1_Q2 = ------
                      100000
                        25
Assigning I_FIX2_Q2 = -------
                      1984127
Assigning R7 = ZIN
Checking for inconsistencies...
... none found.
Searching for immediate assignments.
Assigning I_V1 = 0
Checking for inconsistencies...
... none found.
Searching for immediate assignments.
No immediate assignments found.
Searching for linear equations...
  ...with respect to: [R1, R2, R3, R4, R5, R6, I_R1, I_R2, I_R3, I_R4,

I_R5, I_R6, I_R7, I_V_CC, V_I1, V_OC_1, V_OC_2, V_R1, V_R2, V_R3, V_R4,

V_R5, V_R6, V_R7]
Found 18 linear equations in 18 variables.
The variables to be solved for are [I_R1, I_R2, I_R3, I_R4, I_R5, I_R6,

I_R7, I_V_CC, V_I1, V_OC_1, V_OC_2, V_R1, V_R2, V_R3, V_R4, V_R5, V_R6,

V_R7]
The equations are [1000 V_R1 + 1000 V_OC_1 + 607,

1000 V_R2 + 1000 V_OC_1 + 1000 V_I1 + 607, 200 V_OC_1 + 200 V_I1 - 1567,
```

```
- 1000 V_R6 + 1000 V_R3 - 1000 V_OC_1 - 1000 V_I1 + 2113,

50 V_R7 + 50 V_R4 + 50 V_OC_1 + 50 V_I1 - 321, V_R7 + V_R5 + V_OC_1,

50 V_OC_2 - 50 V_I1 + 321, 1000 V_R6 + 1000 V_OC_1 + 1000 VCC - 2113,

8695637 I_R7 + 5, 8695637000000 I_R2 + 8695637000000 I_R1 - 970215707,

1984127000000 I_R6 + 245238097, - I_R7 + I_R5 + I_R4,

- 198412700000 I_R4 - 798134927, 100000 I_R3 - 100000 I_R2 + 401,

I_V_CC - I_R6 - I_R3, I_R7 ZIN - V_R7]
Solving linear equations.
                            8695637000000 I_R3 + 33899288663
The solutions are [I_R1 = - --------------------------------,
                                  8695637000000

       100000 I_R3 + 401            798134927
I_R2 = -----------------, I_R4 = - ------------,
            100000                198412700000

        6939299538713499              245238097                5
I_R5 = -------------------, I_R6 = - -------------, I_R7 = - -------,
       1725324815389900000            1984127000000         8695637

         1984127000000 I_R3 - 245238097           200 V_OC_1 - 1567
I_V_CC = ------------------------------, V_I1 = - -----------------,
                 1984127000000                          200

           200 V_OC_1 - 283           1000 V_OC_1 + 607           4221
V_OC_2 = - ----------------, V_R1 = - -----------------, V_R2 = - ----,
                 200                        1000                  500

         200 V_OC_1 + 200 VCC - 1567         1000 ZIN - 2460865271
V_R3 = - --------------------------, V_R4 = ---------------------,
                    200                          1739127400

         8695637 V_OC_1 - 5 ZIN           1000 V_OC_1 + 1000 VCC - 2113
V_R5 = - ----------------------, V_R6 = - ----------------------------,
                8695637                              1000

          5 ZIN
V_R7 = - -------]
         8695637
Checking for inconsistencies...
... none found.
```

```
Searching for linear equations...
  ...with respect to: [I_R3, R1, R2, R3, R4, R5, R6, V_OC_1]
Found 6 linear equations in 5 variables.
The variables to be solved for are [R2, R4, R5, R6, V_OC_1]
The equations are [8695637000000 V_OC_1 - 8695637000000 I_R3 R1

 - 33899288663 R1 + 5278251659000, 1984127000000 V_OC_1 + 1984127000000 VCC

 - 245238097 R6 - 4192460351000, 200 V_OC_1 + 200 VCC + 200 I_R3 R3

 - 1567, - 992063500000 ZIN - 6940291602213499 R4 + 2441334613776708500,

- 992063500000 ZIN + 1725324815389900000 V_OC_1 + 6939299538713499 R5,

145309663773 A R1 - 145303681853 R2]
Solving linear equations.
Checking for inconsistencies...
... none found.
                         145309663773 A R1
The solutions are [R2 = -----------------,
                          145303681853

       992063500000 ZIN - 2441334613776708500
R4 = - --------------------------------------,
                  6940291602213499

R5 = - (- 9920635000000 ZIN + (17253248153899000000 I_R3

 + 67260493917052201) R1 - 10472721629416693000)/69392995387134990,

R6 = (17253248153899000000 VCC + (17253248153899000000 I_R3

 + 67260493917052201) R1 - 46928834978605280000)/2132501470082789,

         (8695637000000 I_R3 + 33899288663) R1 - 5278251659000
V_OC_1 = -------------------------------------------------------]
                             8695637000000
Checking for inconsistencies...
... none found.
Searching for linear equations...
  ...with respect to: [I_R3, R1, R3]
No linear equations found.
Checking for remaining equations.
3 equation(s) and 3 variable(s) left.
The variables to be solved for are [I_R3, R1, R3]
Applying valuation strategy.
Trying to solve equation 1 for I_R3
Valuation: 4
```

```
The equation is 14530966377300000 A I_R3 R1 + 58269175172973 A R1

                                                    + 122665368220302600 = 0
Checking if equation was solved correctly.
                            6474352796997 A R1 + 13629485357811400
The solutions are [I_R3 = - ------------------------------------]
                                  1614551819700000 A R1
Solution is correct.
Solution 1 for I_R3
Checking for inconsistencies...
... none found.
Consistent solutions for I_R3 : [I_R3 =

                                   6474352796997 A R1 + 13629485357811400
                                 - ------------------------------------]
                                           1614551819700000 A R1
Checking for remaining equations.
2 equation(s) and 2 variable(s) left.
The variables to be solved for are [R1, R3]
Applying valuation strategy.
Trying to solve equation 1 for R3
Valuation: 20
The equation is - R1 (- 160627551823971108760734084785394486772015696711 48#

1163833723952901566000000000 A VCC + 64411648281412414613054367998943189195#

57829438130394669732330511352796600 0 A R3 + 13560177924979641001581171437 5#

830025732935651163832398508429761039502017200000 A + 135596196971410595846#

32480674053340087555612955778449410425909386417292920000 0) - 1355961969714#

10595846324806740533400875556129557784494104259093864172929200000 R3 - 179#

22019255929527512920504145821571813245350168139179727272345362073826 00 A

   2
 R1  = 0
Checking if equation was solved correctly.
The solutions are [R3 = (14039556541800648900000 0 A R1 VCC

                            2
 - 15664635352383720279 A R1  + (- 11852193632588107801380000 A

 - 11851705716834304886180000) R1)/(562986217326206020890 A R1

 + 11851705716834304886180000)]
Solution is correct.
```

```
Solution 1 for R3
Checking for inconsistencies...
... none found.
Consistent solutions for R3 : [R3 = (14039556541800648900000 A R1 VCC

                             2
 - 15664635352383720279 A R1  + (- 1185219363258810780138000 A

 - 1185170571683430488618000) R1)/(562986217326206020890 A R1

 + 1185170571683430488618000)]
Checking for remaining equations.
1 equation(s) and 1 variable(s) left.
The variables to be solved for are [R1]
Trying to solve equation 1 for R1
Valuation: (irrelevant)
The equation is R1 (19841637776253069394020865409024 ZOUT

 - 24349822242160853396601562 5 A ZIN)

                                      2
 - 57259002716458635629644396416 A R1  = 0
Checking if equation was solved correctly.
The solutions are [R1 =

19841637776253069394020865409024 ZOUT - 24349822242160853396601562 5 A ZIN
-------------------------------------------------------------------------,
                     57259002716458635629644396416 A

R1 = 0]
Solution is correct.
The solution is not unique. Tracing paths separately.
Solution 1 for R1
Checking for inconsistencies...
... none found.
Solution 2 for R1
Checking for inconsistencies...
... none found.
Consistent solutions for R1 : [R1 =

19841637776253069394020865409024 ZOUT - 24349822242160853396601562 5 A ZIN
-------------------------------------------------------------------------,
                     57259002716458635629644396416 A

R1 = 0]
Checking for remaining equations.
All variables solved for. No equations left.
Checking for remaining equations.
```

```
All variables solved for. No equations left.
Postprocessing results.

(D11) [[R1 = - (24349822242160853396601562 5 A ZIN

 - 19841637776253069394020865409024 ZOUT)

/(57259002716458635629644396416 A), R2 =

  14938541252859419261718 75 A ZIN - 12172783911811699014736727244 8 ZOUT
- ------------------------------------------------------------------,
                      35126776412275901674720115 2

R3 = (33476318472456810570225873245155046039570859311579289355943679308595 2

     2
 ZOUT  + VCC

                                                                          2
 (10625645733711576869210053078719589996310389549602435000000000000000 A  ZIN

 - 865838820876683239612627474388382068829173989238347691708959948800000 0 A

 ZOUT) + A

 (73094113258409599088098011387867214250558868171501312134070398877696000

 ZOUT + ZIN

 (- 82164830677623835681150914833206609917142422498519656448000000000 ZOUT

 - 89698008560556767832189447109189280381589732951869815470000000000000))

 + 73091104214643137701992515955008538217110816411520639295650692857856000

         2
 ZOUT + A  (50416680420198682402077210492446131565566605185394287109375

    2
 ZIN  - 89701701283993131929871268090550778748852308577743756270000000000 0

 ZIN))/(A

 (- 34720136717154997908466361722974120960049876968457742437529293946880

 ZOUT

 - 21092632483111174683325097183189754403716708919298794191829902950400 0)
```

```
                                                                          2
 + 426088393921834232455323128456655558852046620939057643500000000 A

              992063500000 ZIN - 2441334613776708500
 ZIN), R4 = - ---------------------------------------,
                         6940291602213499

R5 = (381957713831956915040520868450162467946676879 36 ZOUT

 + A (9930399595766410870496554922774419242187 5 ZIN

 + 5996575962274855332613362021809166941397722880 00)

 + 8339540409811836894068029655629814665587863808000)

/(39733737113751639640009221921695330300641958 40 A),

R6 = (- 381957713831956915040520868450162467946676879 36 ZOUT

 + A (4687416704563305079747316874106999675781 25 ZIN

 - 2687098289520198765190831087202789799110676480000)

 + 98790378291183778132015848794220213202598400000 0 A VCC

 - 8339540409811836894068029655629814665587863808000)

/(12210490746832244925030394491952536145488422 4 A), R7 = ZIN],

                                  992063500000 ZIN - 2441334613776708500
[R1 = 0, R2 = 0, R3 = 0, R4 = - ---------------------------------------,
                                            6940291602213499

     992063500000 ZIN + 1047272162941669300
R5 = --------------------------------------,
                 6939299538713499

     1984127000000 VCC - 5396825440000
R6 = ---------------------------------, R7 = ZIN]]
                  245238097
```

Here we have more than one solution of the system ,too, but only first one is physically meaningful result.

Appendix B

Program listings

B.1 SOLVER.MAC

```
/*************************************************************************/
/*                                                                       */
/* SOLVER - THE NEXT GENERATION                                          */
/*                                                                       */
/* Copyright (C) 2000 : Eckhard Hennig, Ralf Sommer                      */
/* This library is free software; you can redistribute it and/or modify it    */
/* under the terms of the GNU Library General Public License as published     */
/* by the Free Software Foundation; either version 2 of the License, or (at   */
/* your option) any later version.                                       */
/*                                                                       */
/* This library is distributed in the hope that it will be useful, but        */
/* WITHOUT any WARRANTY; without even the implied warranty of                 */
/* MERCHANTABILITY or FITNESS for A PARTICULAR PURPOSE.  See the GNU          */
/* Library General Public License for more details.                      */
/*                                                                       */
/* You should have received a copy of the GNU Library General Public          */
/* License along with this library; if not, write to the Free Software        */
/* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA
/*************************************************************************/
/* Credits: This program is based on many ideas from Henning Trispel's work   */
/*          on the EASY-Solver in 1991.                                  */
/*                                                                       */
/*************************************************************************/
/* Author(s)    : Eckhard Hennig, Ralf Sommer                            */
/* Project start: 19.01.1994                                             */
/* Completed    : 16.07.1994                                             */
/* last change  : 29.06.1995                                             */
/* Time         : 09:26                                                  */
/*************************************************************************/
/* Changes      : ||||| ||||| ||||| |||                                  */
/*************************************************************************/
/* Modified by  : Dan Stanger dan.stanger@ieee.org to work under maxima      */
```

```
/*****************************************************************************/
load( "solver/linsolve.mac" )$
load( "solver/slvrtbox.mac" )$
load( "solver/slvrmsgs.mac" )$
load( "solver/misc.mac" )$

put( 'SOLVER, 1, 'Version )$

SetVersion(
  /* KEY    = */ 'SOLVER,
  'MODULE      = "SOLVER",
  'DESCRIPTION = "Symbolic solver for parametric systems of equations.",
  'AUTHORS     = "Eckhard Hennig, Ralf Sommer",
  'DATE        = "19.01.1994",
  'LASTCHANGE  = "29.06.1995",
  'TIME        = "09:26",
  'PLAN        = "Add a-priori transforms"
)$


/*****************************************************************************/
/* last change: 29.06.1995                                                   */
/* Time       : 09:26                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Option variable Solver_Valuate_All_Nonlin_Vars added.        */
/*              Bug in ValuationSolver removed: rhs variables of solutions   */
/*              were not added to the list of variables.                     */
/*****************************************************************************/
/* last change: 28.05.1995                                                   */
/* Time       : 11:59                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Solver break test added.                                     */
/*****************************************************************************/
/* last change: 18.05.1995                                                   */
/* Time       : 16:10                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Name conflict with AI keyword PARAMS removed.                */
/*****************************************************************************/
/* last change: 30.01.1995                                                   */
/* Time       : 21.42                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Removal of map( 'num, ... ) has revealed some side effects on */
/*              the list of equations in ValuationSolver. Bug repaired by    */
/*              inserting two additional copylist calls.                     */
/*              Arguments to both load commands above now written in lower-  */
/*              case letters to avoid problems with UNIX versions.           */
/*****************************************************************************/
/* last change: 24.01.1995                                                   */
```

```
/* Time       : 16.33                                                    */
/* By         : Eckhard Hennig                                           */
/* Description: mapping 'num to expressions caused some invalid solutions,    */
/*              therefore removed. Version property added.               */
/*************************************************************************/
/* last change: 19.01.1995                                               */
/* Time       : 13.04                                                    */
/* By         : Eckhard Hennig, Ralf Sommer                              */
/* Description: Bug in ImmediateAssignments corrected, Function AppendImmed   */
/*              removed.                                                 */
/*************************************************************************/
/* last change: 19.01.1995                                               */
/* Time       : 11.28                                                    */
/* By         : Eckhard Hennig, Ralf Sommer                              */
/* Description: SLVRTBOX and SLVRMSGS now autoloading.                   */
/*************************************************************************/
/* last change: 17.01.1995                                               */
/* Time       : 20.15                                                    */
/* By         : Eckhard Hennig, Ralf Sommer                              */
/* Description: Option variables modified (underscores inserted according to  */
/*              R. Petti's suggestions)                                  */
/*              Function SetValuation added.                             */
/*************************************************************************/
/* last change: 25.10.1994                                               */
/* Time       : 12.46                                                    */
/* By         : Eckhard Hennig                                           */
/* Description: errorMsg renamed to ErrMsg.                              */
/*************************************************************************/
/* last change: 12.09.1994                                               */
/* Time       : 11.50                                                    */
/* By         : Eckhard Hennig                                           */
/* Description: errcatch wrapped around final fullratsimp.               */
/*************************************************************************/
/* last change: 05.09.1994                                               */
/* Time       : 15.59                                                    */
/* By         : Eckhard Hennig                                           */
/* Description: mode_identity's inserted.                                */
/*************************************************************************/
/* last change: 05.09.1994                                               */
/* Time       : 15.21                                                    */
/* By         : Eckhard Hennig                                           */
/* Description: Number of linear equations is now correctly displayed.   */
/*              Bug in DumpToFile corrected.                             */
/*              Numbers of solution sets are now printed by the postprocessor.*/
/*************************************************************************/
/* last change: 30.08.1994                                               */
/* Time       : 13.45                                                    */
/* By         : Eckhard Hennig                                           */
```

```
/* Description: Single inconsistent solution paths are no longer returned.    */
/*****************************************************************************/
/* last change: 29.08.1994                                                   */
/* Time       : 17.16                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Slight modification of Immediate Assignment Solver.          */
/*              Valuation property for sqrt added.                           */
/*              New option variable SolverDefaultValuation.                  */
/*              Change in Valuation. Now making use of the above option var. */
/*****************************************************************************/
/* last change: 26.08.1994                                                   */
/* Time       : 15.29                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Bug in Linear Solver removed.                                */
/*****************************************************************************/
/* last change: 25.08.1994                                                   */
/* Time       : 12.41                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: New option variable SolverRatSimpSols.                       */
/*              Linear Solver now calls the consistency check.               */
/*              Capabilities of the transforms in ValuationSolver strongly   */
/*              enhanced.                                                    */
/*****************************************************************************/
/* last change: 24.08.1994                                                   */
/* Time       : 23.14                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Bug fixed in postprocessor: compound expressions are now cor- */
/*              rectly evaluated.                                            */
/*              Linear Solver now checks for remaining equations & variables, */
/*              and removes those "linear" eqs and vars which do not really  */
/*              belong to the "true" linear subsystem.                       */
/*****************************************************************************/
/* last change: 11.08.1994                                                   */
/* Time       : 19.09                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Variable SolverDelEq2VarPref replaced by function variable   */
/*              SolverDelEq. Heuristic algorithm for linear equation         */
/*              extraction improved.                                         */
/*****************************************************************************/
/* last change: 19.07.1994                                                   */
/* Time       : 18.01                                                        */
/* By         : Eckhard Hennig                                               */
/* Description: Bug in post processor corrected.                             */
/*****************************************************************************/


/*****************************************************************************/
/* Global variables for Solver                                               */
```

```
/*****************************************************************************/

/*****************************************************************************/
/* If Solver_Immed_Assign is true then the Solver searches the equations     */
/* for immediate assignments of the form variable = constant and immediately */
/* inserts these constraints into the remaining equations.                   */
/*****************************************************************************/

define_variable( Solver_Immed_Assign, true, boolean )$


/*****************************************************************************/
/* Solver_Repeat_Immed controls whether the search for immediate assignments */
/* is performed repeatedly until no more of them are found.                  */
/*****************************************************************************/

define_variable( Solver_Repeat_Immed, true, boolean )$


/*****************************************************************************/
/* Solver_Subst_Powers controls whether the Solver substitutes powers of a   */
/* variable by new symbols in one of the following cases:                    */
/* 1. var^n appears raised to exactly one power n                            */
/* 2. for all var^m in the equations : m=n*k , k integer                     */
/*****************************************************************************/

define_variable( Solver_Subst_Powers, false, boolean )$


/*****************************************************************************/
/* Solver_Incons_Params controls whether the Solver terminates when a non-   */
/* trivial equation containing only parameters is encountered. for example,  */
/* if A and B are defined as parameters and the solution process yields      */
/* A = B^2  then the Solver stops if Solver_Incons_Params is = 'BREAK. If set */
/* to 'ASK the Solver asks whether  A - B^2  is zero and continues if it is. */
/* If set to 'IGNORE the Solver quietly assumes that the expression is zero  */
/* if it does not directly contradict with any of the assumptions made before.*/
/*****************************************************************************/

define_variable( Solver_Incons_Params, 'ASK, any_check )$

put(
  'Solver_Incons_Params,
  lambda( [ x ],
    if not member( x, [ 'ASK, 'BREAK, 'IGNORE ] ) then
      ErrorHandler("InvIncPar", x, 'Fatal )
  ),
  'value_check
```

```
)$


/*****************************************************************************/
/* Solver_Linear controls whether the Solver tries to find and solve blocks  */
/* of linear equations before submitting the remaining equations to the      */
/* heuristic valuation solver. This is useful if the equations to be solved  */
/* are known to contain a large linear part.                                 */
/*****************************************************************************/

define_variable( Solver_Linear, true, boolean )$


/*****************************************************************************/
/* If Solver_Repeat_Linear is true then the LinearSolver will be called      */
/* repeatedly until no more linear equations are found. If false then        */
/* LinearSolver will be called only once.                                    */
/*****************************************************************************/

define_variable( Solver_Repeat_Linear, true, boolean )$


/*****************************************************************************/
/* If Solver_Find_All_Linear_Vars is true then LinearSolver will try to find */
/* linear equations with respect to all available variables and solve these  */
/* equations simultaneously. If false then LinearSolver will only search for */
/* linear equations with respect to the variables passed over in the function */
/* call.                                                                     */
/*****************************************************************************/

define_variable( Solver_Find_All_Linear_Vars, true, boolean )$


/*****************************************************************************/
/* Solver_Assumptions contains constraints on the parameters which should be */
/* checked by the user after the termination of Solver. These constraints    */
/* result from the parameter consistency check the behavior of which is      */
/* controlled by the setting of the option variable Solver_Incons_Params.    */
/* Any numerical solution of the equations obtained by assigning numerical   */
/* values to symbolic parameters should be checked for consistency with all  */
/* expressions in Solver_Assumptions.                                        */
/*****************************************************************************/

define_variable( Solver_Assumptions, [], list )$


/*****************************************************************************/
/* Solver_Del_Eq holds the name of a function which controls the behavior of */
```

```
/* the heuristic search algorithm which extract linear equations from the     */
/* entire system of equations.                                                */
/******************************************************************************/

define_variable( Solver_Del_Eq, 'MakeSquareLinearBlocks, any )$


/******************************************************************************/
/* If Solver_Valuate_All_Nonlin_Vars is true then the ValuationSolver will    */
/* valuate the equations w.r.t. all remaining variables and not only w.r.t    */
/* variables which are being searched for at the current step.                */
/******************************************************************************/

define_variable( Solver_Valuate_All_Nonlin_Vars, false, boolean )$


/******************************************************************************/
/* Solver_Valuation_Strategy holds the name of the equation valuation         */
/* strategy called by the valuation solver to determine the order by which    */
/* the equations are to be solved.                                            */
/******************************************************************************/

define_variable( Solver_Valuation_Strategy, 'MinVarPathsFirst, any_check )$

put(
  'Solver_Valuation_Strategy,
  lambda(
    [ x ],
    if not FunctionP( x ) then
      ErrorHandler( "UndefStrat", x, 'Fatal )
  ),
  'value_check
)$


/******************************************************************************/
/* Solver_Default_Valuation contains the default valuation for arithmetic     */
/* operators. Whenever an operator is encountered for which no valuation has  */
/* been defined by SetProp( <operator>, 'Valuation, <valuation> ) this value  */
/* is taken for the formula complexity calculations.                          */
/******************************************************************************/

define_variable( Solver_Default_Valuation, 10, fixnum )$


/******************************************************************************/
/* Solver_Max_Len_Val_Order limits the length of the list of candidates for   */
/* the solve calls in ValuationSolver. A low value will usually increase the  */
```

```
/* efficiency of the valuation solver since, in general, the first or second  */
/* attempt to solve an equation (hopefully) succeeds.                         */
/******************************************************************************/

define_variable( Solver_Max_Len_Val_Order, 5, fixnum )$


/******************************************************************************/
/* Solver_Transforms is a list containing the names of functions which can be */
/* applied to an equation after a failed solve call. These functions must     */
/* take three arguments: the equation to be transformed, the variable to be   */
/* solved for, and a list of (probably implicit) solutions the Solver has     */
/* already found for the equation.                                            */
/******************************************************************************/

define_variable( Solver_Transforms, [], list )$


/******************************************************************************/
/* If Solver_Postprocess is set to false no postprocessing of the results will*/
/* be done. Instead, the solutions are displayed in the internal hierarchical */
/* list format. Useful for debugging purposes.                                */
/******************************************************************************/

define_variable( Solver_Postprocess, true, boolean )$


/******************************************************************************/
/* Solver_Backsubst controls the output format of Solver. If Solver_Backsubst */
/* is true then the result will be displayed with fully evaluated right-hand  */
/* sides for each variable. If the option variable is set to false then the   */
/* right-hand sides of the solutions may still contain references to some     */
/* of the other variables which have been solved for.                         */
/******************************************************************************/

define_variable( Solver_Backsubst, true, boolean )$


/******************************************************************************/
/* With Solver_Disp_All_Sols set to true all solutions will be displayed      */
/* including those for the variables which have been solved for in the        */
/* solution process but have not been explicitly asked for.                   */
/******************************************************************************/

define_variable( Solver_Disp_All_Sols, false, boolean )$


/******************************************************************************/
```

```
/* If Solver_RatSimp_Sols is true then the Solver Postprocessor will          */
/* fullratsimp the solutions before returning them.                           */
/****************************************************************************/

define_variable( Solver_RatSimp_Sols, true, boolean )$


/****************************************************************************/
/* If Solver_Dump_To_File is true then the ValuationSolver writes the         */
/* solutions and yet unsolved equations to a file after each iteration. This  */
/* might help if the Solver crashes.                                          */
/****************************************************************************/

define_variable( Solver_Dump_To_File, false, boolean )$


/****************************************************************************/
/* Solver_Dump_File contains the name of the file to which the dump is        */
/* written.                                                                   */
/****************************************************************************/

define_variable( Solver_Dump_File, "SOLVER.DMP", any )$


/****************************************************************************/
/* Solver_Break_Test holds the name of a function which is called immediately */
/* before attempting to solve an equation. This function then decides whether */
/* Solver should try to solve the equation or whether it should stop, e.g.    */
/* because the problem has become too complex. The arguments passed to the    */
/* Solver_Break_Test are 1. the equation, 2. the variable, 3. the valuation.  */
/* The Solver stops if the Solver_Break_Test returns true, it continues if    */
/* the return value is false.                                                 */
/****************************************************************************/

define_variable( Solver_Break_Test, 'SolverJustDoIt, any )$


/****************************************************************************/
/* Solver, main program                                                       */
/****************************************************************************/

Solver( Equations, [ SolverParams ] ) := (

  mode_declare(
    [ Equations, SolverParams ], list
  ),

  block(
```

```
[
  Variables,       /* List of variables to be solved for           */
  UserVars,        /* List of variables specified by user          */
  Parameters,      /* List of symbols to be used as parameters     */
  Expressions,     /* List of compound expressions to be solved for */
  PowerSubst,      /* List of substituted symbols for powers       */

  Solutions,       /* List of solutions found by Solver            */
  RemainingEqs,    /* List of remaining equations                  */

  /* Assumptions made in linear solver should be local  */
  Solver_Assumptions : [],

  Active
],

mode_declare(
  [
    Variables, UserVars, Parameters, Expressions, PowerSubst,
    Solutions
  ], list,
  Active, boolean
),

/* No assumptions to start with */ErrorHandlerSolver_Assumptions : [],

/* Initialize list of solutions */
Solutions : [],

/* Do all necessary preprocessing */
map(
  lambda([x,y],x::y),
  [
    'Equations, 'SolverParams, 'Variables, 'Parameters,
    'Expressions, 'PowerSubst, 'UserVars
  ],
  SetupSolver( Equations, SolverParams )
),

if MsgLevel = 'DEBUG then
  display(
    Equations, SolverParams, Variables, Parameters, Expressions,
    PowerSubst, UserVars, Solutions
  ),

block(
  [],
```

```
/* Search for and apply immediate assignments */

Active : Solver_Immed_Assign,
while Active do (
  map(
    lambda([x,y],x::y),
    [ 'Active, 'Solutions, 'Equations, 'Variables ],
    ImmediateAssignments( Solutions, Equations, Variables, Parameters )
  ),

  Active : Active and Solver_Repeat_Immed,

  if MsgLevel = 'DEBUG then
    display(
      Equations, SolverParams, Variables, Parameters, Expressions,
      PowerSubst, UserVars, Solutions
    )
),

if Empty( Equations ) or Empty( Variables ) then
  return( false ),

Equations : map(
  lambda(
    [ Eq ],
    fullratsimp( lhs( Eq ) - rhs( Eq ) )
  ),
  Equations
),

/* Find and solve linear equations */

Active : Solver_Linear,
while Active do (
  map(
    lambda([x,y],x::y),
    [ 'Active, 'Solutions, 'Equations, 'Variables ],
    LinearSolver( Solutions, Equations, Variables, Parameters )
  ),

  Active : Active and Solver_Repeat_Linear,

  if MsgLevel = 'DEBUG then
    display(
      Equations, SolverParams, Variables, Parameters, Expressions,
      PowerSubst, UserVars, Solutions
    )
```

```
      ),

      if Empty( Equations ) or Empty( Variables ) then
        return( false ),


      if Solver_Valuate_All_Nonlin_Vars then
        Variables : Union(
          SetDifference( listofvars( Equations ), Parameters ),
          Variables
        ),

      /* apply valuation strategies to solve the nonlinear equations. */
      map(
        lambda([x,y],x::y),
        [ 'Active, 'Solutions, 'Equations, 'Variables ],
        ValuationSolver( Solutions, Equations, Variables, Parameters )
      )

    ), /* END block */


    /* Return the solutions and the unsolved equations */

    if Solver_Postprocess then
      return( PostProcess( Solutions, UserVars, Expressions, PowerSubst ) )
    else
      return( Solutions )
  )
)$


/*****************************************************************************/
/* TerminateSolver terminates the Solver.                                    */
/*****************************************************************************/

TerminateSolver() := error( ErrMsg["SolvrTerm"] )$


/*****************************************************************************/
/* SetupSolver preprocesses the equations and optional parameters before     */
/* submitting them to the Solver. The equations are checked if they are      */
/* really equations, and all equations of the form  NUMBER = NUMBER  or      */
/* f( PARAMETERS ) = g( PARAMETERS )  are checked for consistency and then   */
/* dropped.                                                                  */
/*****************************************************************************/

SetupSolver( Equations, SolverParams ) := (
```

```
mode_declare(
  [ Equations, SolverParams ], list
),

block(

  [
     i, AllVars, Var, SubstSym, Power,
     Variables, Parameters, Expressions,
     PowerSubst, UserVars
  ],

  mode_declare(
    i, fixnum,
    [
      AllVars, Power, Variables, Parameters, Expressions,
      PowerSubst, UserVars
    ], list,
    [ Var, SubstSym ], any
  ),


  Expressions : [],
  PowerSubst  : [],
  Parameters  : [],


  /* Make sure that Equations is a list. Abort if it is not. */

  if not listp( Equations ) then
    ErrorHandler( "EqsNotLst", Equations, 'Fatal ),

  /* delete all entries from Equations which are not equations */

  Equations : sublist( Equations, 'EquationP ),

  /* Convert all floating-point numbers to rational numbers. If this was */
  /* not done then rounding errors could fool the consistency check.     */

  Equations : map( 'rat, Equations ),

  /* Process the optional arguments to Solver */

  if not Empty( SolverParams ) then (

    Variables : SolverParams[1],
```

```
  /* Make sure that Variables is a list. Abort if it is not. */

  if not listp( Variables ) then
    ErrorHandler( "VarNotLst", Variables, 'Fatal ),

  /* delete multiple occurrences of identical symbols */

  Variables : Setify( Variables ),

  PrintMsg( 'DETAIL, SolverMsg["VarsAre"], Variables ),

  if length( SolverParams ) > 1 then (

    Parameters : SolverParams[2],

    /* Make sure that Parameters is a list. Abort if it is not. */

    if not listp( Parameters ) then
      ErrorHandler( "ParNotLst", Parameters, 'Fatal ),

    /* delete multiple occurrences of identical symbols */

    Parameters : Setify( Parameters ),

    PrintMsg( 'DETAIL, SolverMsg["ParsAre"], Parameters )
  )

),


/* Check if Variables and Parameters are disjoint sets of symbols */

if not DisjointP( Variables, Parameters ) then
  ErrorHandler(
    "VarParConfl", Intersection( Variables, Parameters ), 'Fatal
  ),


/* Make a list of all variables to be solved for. This list may contain */
/* more symbols than Variables when either no SolverParams have been    */
/* given or when the user has specified only a subset of all existing   */
/* symbols.                                                             */

AllVars : SetDifference( listofvars( Equations ), Parameters ),

/* solve for all available variables if Variables is empty. */

if Empty( Variables ) then
```

```
  Variables : AllVars,


/* Check all those equations which contain only equations of the form  */
/* NUMBER = NUMBER or which contain only parameters and none of the    */
/* variables of interest for consistency. Remove consistent equations  */
/* and store the assumptions made in the list Solver_Assumptions.      */

Equations : ParamConsistency( Equations, Parameters ),

/* Abort if no equations are left after the above step. */

if Empty( Equations ) or Empty( Variables ) then (
  PrintMsg( 'SHORT, SolverMsg["NoEqOrVar"] ),
  return( [ [], SolverParams, Variables, Parameters, [], [], [] ] )
),

/* If there are compound expressions to be solved for then the solver     */
/* tries to solve for the variables contained in them first.              */
/* Subsequently the expressions are rebuilt from the solutions of         */
/* these variables and the parameters.                                    */

for i thru length( Variables ) do

  /* Search the variable list for non-atomic expressions. */

  if not atom( Var : Variables[i] ) then (

    /* append expression to the expression list */
    Expressions : endcons( Var, Expressions ),

    /* Insert the variables in the expression into the list of variables  */
    /* but keep the parameters out.                                       */
    Variables[i] : SetDifference( listofvars( Var ), Parameters ),

    PrintMsg( 'SHORT, SolverMsg["TrySolve4"], Variables[i] ),
    PrintMsg( 'SHORT, SolverMsg["Solve4Exp"], Var )
  ),

/* Make a list of all the variables the user actually wants to know */
UserVars : sublist( Variables, 'atom ),

/* Flatten the list of variables and make it a set. */
Variables : Setify( Flatten( Variables ) ),


/* If Solver_Subst_Powers is true then substitute powers by new symbols */
```

```
if Solver_Subst_Powers then (

  PrintMsg( 'SHORT, SolverMsg["SubstPwrs"] ),

  for Var in AllVars do (

    /* get all powers of Var in Equations. Convert negative powers to */
    /* positive ones.                                                 */
    Power : Setify( abs( ListOfPowers( Equations, Var ) ) ),

    /* If there is more than one power of Var then substitute each  */
    /* var^m only if all m are integer multiples of the lowest      */
    /* power > 0. The check is done by examining the modulus of all */
    /* powers with respect to the lowest power.                     */

    if
      not member( 'false, map( 'integerp, Power ) )
      and
      Power[1] # 1
    then

      if ( length( Power ) = 1 ) or

        block(
          [ Modulus : Power[1] ],
          not member(
            'false,
            map( 'ZeroP, totaldisrep( rat( rest( Power ) ) ) )
          )
        )

      then (
        /* Make a new symbol for var^power */
        SubstSym : concat( Var, "^", Power[1] ),

        /* Store a reference to the original term in an assoc list */
        PowerSubst : endcons( SubstSym = Var^Power[1], PowerSubst ),

        /* Substitute the new symbol for the original term */
        Equations  : ratsubst( SubstSym, Var^Power[1], Equations ),
        Variables  : subst( SubstSym, Var, Variables ),

        /* Notify user */
        PrintMsg( 'DETAIL, SolverMsg["subst"], Var^Power )
      )

  ) /* END for Var in AllVars */
```

```
    ), /* END if Solver_Subst_Powers */

    return(
      [
        Equations, SolverParams, Variables, Parameters, Expressions,
        PowerSubst, UserVars
      ]
    )
  )
)$


/*****************************************************************************/
/* ParamConsistency checks equations of the form NUMBER = NUMBER and equa-   */
/* tions which contain only parameters for consistency. If the system cannot */
/* determine whether a parametric expression is zero then the user is optio- */
/* nally asked to supply the required information. The assumptions made are  */
/* stored in the list Solver_Assumptions.                                    */
/*****************************************************************************/

ParamConsistency( Eqs, Pars, [ Action ] ) := (

  mode_declare(
    [ Eqs, Pars ], list,
    Action, any
  ),

  block(

    [ Eq, lhsminusrhs, i, consistent ],

    mode_declare(
      [ Eq, lhsminusrhs ], any,
      i, fixnum,
      consistent, boolean
    ),

    PrintMsg( 'SHORT, SolverMsg["ConsChk"] ),

    if Empty( Action ) then
      Action : 'BREAK
    else
      Action : Action[1],

    consistent : true,

    for i thru length( Eqs ) do (
```

```
      Eq : Eqs[i],

      /* Does the equation contain only numbers or parameters? */
      if Empty(
        SetDifference( listofvars( Eq ), Pars )
      )
      then (

        /* If so, check for consistency */
        lhsminusrhs :  expand( lhs( Eq ) - rhs( Eq ) ),

        /* Test if difference of both rhs's is zero */
        if lhsminusrhs # 0 then

          if SolverAssumeZero( lhsminusrhs ) then
            PrintMsg( 'SHORT, SolverMsg["Assum"], lhsminusrhs = 0 )
          else
            if Action = 'BREAK then (
              /* Abort if difference is non-zero */
              PrintMsg( 'SHORT, SolverMsg["Incons"], lhs( Eq ) = rhs( Eq ) ),
              TerminateSolver()
            )
            else
              return( consistent : false ),

        /* Kill the now redundant equation */
        if Action = 'BREAK then
          Eqs[i] : []

      ) /* END if Empty */

    ), /* END for i */

    if consistent then
      PrintMsg( 'SHORT, SolverMsg["NoneFnd"] ),

    if Action = 'BREAK then
      return( delete( [], Eqs ) )
    else
      return( consistent )
  )
)$


/*************************************************************************/
/* SolverAssumeZero checks whether Expression is (assumed to be) equal to     */
/* zero. If it isn't, the function returns false or asks the user for his     */
/* decision. New assumptions are appended to the list Solver_Assumptions.     */
```

```
/*****************************************************************************/

SolverAssumeZero( Expression ) := (

  mode_declare(
    Expression, any
  ),

  block(

    [ AssumptionExists, i ],

    mode_declare(
      i, fixnum,
      AssumptionExists, boolean
    ),

    /* Return false immediately if Expression is a number # 0 or if  */
    /* Solver_Incons_Params is set to 'BREAK.                        */

    if
      Empty( listofvars( Expression ) )
      or ( Solver_Incons_Params = 'BREAK )
    then
      return( false ),

    /* Do a simple check to find out whether assumption already exists:    */
    /* Zero is substituted for Expression in the stored assumption. If the */
    /* result is zero then the assumption already exists.                  */

    AssumptionExists : false,
    for i thru length( Solver_Assumptions ) while not AssumptionExists do
      if
        fullratsimp(
          ratsubst( 0, Expression, lhs( Solver_Assumptions[i] ) )
        ) = 0
      then
        AssumptionExists : true,

    /* If the expression is not yet assumed to be equal to zero, ask the user */
    /* to decide whether it is.                                                */

    if not AssumptionExists then

      if
        ( Solver_Incons_Params = 'ASK )
        and
        ( AskZeroNonzero( Expression ) # 'zero )
```

```
      then
        return( false )
      else
        /* If difference is equal to zero or assumed to be so then store    */
        /* the constraint in the global list Solver_Assumptions. So the user*/
        /* has access to all assumptions made during the solution process   */
        /* and can check any numerical solutions for consistency with the   */
        /* assumptions.                                                     */
        Solver_Assumptions : endcons(  Expression = 0, Solver_Assumptions )

    else
      PrintMsg( 'DETAIL, SolverMsg["AssmFnd"], Expression = 0 ),

    return( true )
  )
)$

AskZeroNonzero( Expression ) :=
  if equal( x, 0 ) = true then 'zero
  else block(
    [s : 'pnz],
    while s#'zero and s#'nonzero do (
      s : read( "Is", Expression, "zero or nonzero?" )
    ),
    s
  )$

/*****************************************************************************/
/* ListOfPowers returns the list of powers # 0 of a variable in a set of     */
/* equations.                                                                */
/*****************************************************************************/

ListOfPowers( Eqs, Var ) := (

  mode_declare(
    Eqs, list,
    Var, any
  ),

  delete(
    0,
    apply(
      'Union,
      map(
        lambda(
          [ Eq ],
          Powers( expand( lhs( Eq ) - rhs( Eq ) ), Var )
        ),
```

```
        Eqs
      )
    )
  )
)$


/*****************************************************************************/
/* ImmediateAssignments directly applies all immediate assignments of the    */
/* form  var = rhs...  before the actual Solver is called.                   */
/*****************************************************************************/

ImmediateAssignments( Solutions, RemainingEqs, Variables, Parameters ) := (

  mode_declare(
    [ Solutions, RemainingEqs, Variables, Parameters ], list
  ),

  block(

    [ i, Vars, AssignmentMade, Left, Right ],

    mode_declare(
      i, fixnum,
      Vars, list,
      AssignmentMade, boolean,
      [ Left, Right ], any
    ),

    PrintMsg( 'SHORT, SolverMsg["SrchImmed"] ),

    AssignmentMade : false,

    for i thru length( RemainingEqs ) do block(
      [],

      Left  : lhs( RemainingEqs[i] ),
      Right : rhs( RemainingEqs[i] ),

      /* Scan the equations for simple assignments of the form  X = Expr or   */
      /* Expr = X  and apply this assignment only if X is not a parameter and */
      /* if Expr contains only numbers or parameters.                         */
      /* Remark: Equations containing only parameters have already been remo- */
      /* ved from the equation list in SetupSolver.                           */

      if symbolp( Left ) then (

        Vars : listofvars( Right ),
```

```
    /* Check if lhs is an isolated variable and make sure that there are */
    /* only parameters on the right-hand side.                           */

    if freeof( Left, Right )
      and Empty( SetDifference( Vars, Parameters ) )
    then (

      if assoc( Left, Solutions ) = false then (
        PrintMsg(
          'DETAIL,
          SolverMsg["Assign"], totaldisrep( Left = Right )
        ),

        if member( Left, map( lhs, Solutions ) ) then (
          if assoc( Left, Solutions ) # Right then (
            PrintMsg( 'SHORT, SolverMsg["Incons"], Left = Right ),
            TerminateSolver()
          )
        )
        else
          Solutions : endcons( Left = Right, Solutions ),
        RemainingEqs[i] : [],

        AssignmentMade : true
      ),

      /* This return prevents Macsyma from entering the next if statement */
      /* which must only be executed when the current outer if statement  */
      /* has not been entered.                                            */
      return( 'DONE )
    )
  ),

  if symbolp( Right ) then (

    Vars : listofvars( Left ),

    if freeof( Right, Left )
      and Empty( SetDifference( Vars, Parameters ) )
    then (

      if assoc( Right, Solutions ) = false then (
        PrintMsg(
          'DETAIL,
          SolverMsg["Assign"], totaldisrep( Right = Left )
        ),
```

```
            if member( Right, map( lhs, Solutions ) ) then (
              if assoc( Right, Solutions ) # Left then (
                PrintMsg( 'SHORT, SolverMsg["Incons"], Left = Right ),
                TerminateSolver()
              )
            )
            else
              Solutions : endcons( Right = Left, Solutions ),
            RemainingEqs[i] : [],

            AssignmentMade : true
          )

        )
      )
    ), /* END for i thru length */

    if AssignmentMade then (

      /* delete the used equations from the list */
      RemainingEqs : delete( [], RemainingEqs ),

      if not Empty( RemainingEqs ) then (

        /* Remove all the variables from the working list which have been */
        /* determined by an immediate assignment.                         */
        Variables : SetDifference( Variables, map( 'lhs, Solutions ) ),

        /* Evaluate the remaining equations with the constraints */
        RemainingEqs : ev( RemainingEqs, Solutions ),

        /* Do a parameter consistency check */
        RemainingEqs : ParamConsistency( RemainingEqs, Parameters )
      )

      else
        PrintMsg( 'SHORT, SolverMsg["NoEqTerm"] )

    )
    else
      PrintMsg( 'SHORT, SolverMsg["NoImmed"] ),

    /* END if AssignmentMade */

    return( [ AssignmentMade, Solutions, RemainingEqs, Variables ] )
  )
)$
```

```
/*************************************************************************/
/* LinearSolver extracts blocks of linear equations from an arbitrary system  */
/* of equations by a heuristic searching strategy and solves the linear block */
/* if there is one.                                                          */
/*************************************************************************/

LinearSolver( Solutions, Equations, Variables, Parameters ) := (

  mode_declare(
    [ Solutions, Equations, Variables, Parameters ], list
  ),

  block(

    [
      CoeffMatrix, ValuationMatrix, ActiveVars,
      LinEqNos, LinVarNos, NewVars, LinSolVars,
      LinearEqs, LinearVars, LinearSolutions,
      i, j, me, mv, NumVars, NumEqs,
      MaxValVar, MaxValEq,
      EqValuation, VarValuation,
      rhsExpressions,

      linsolvewarn             : false,
      linsolve_params          : false,
      Solve_Inconsistent_Error : false,

      EqWasLast : false
    ],

    mode_declare(
      [ CoeffMatrix, ValuationMatrix ], any,
      [
        LinEqNos, LinVarNos, ActiveVars, LinearEqs, LinearVars, LinSolVars,
        EqValuation, VarValuation, LinearSolutions, NewVars, rhsExpressions
      ], list,
      [ i, j, me, mv, NumVars, NumEqs, MaxValVar, MaxValEq ], fixnum
    ),

    if Empty( Equations ) then (

      if Empty( Variables ) then
        PrintMsg( 'SHORT, SolverMsg["AllSolved"] )
      else
        PrintMsg( 'SHORT, SolverMsg["NoEqLeft"], Variables ),

      return( [ false, Solutions, Equations, Variables ] )
```

```
)
else if Empty( Variables ) then (
  PrintMsg( 'SHORT, SolverMsg["EqLeft"] ),
  return( [ false, Solutions, Equations, Variables ] )
),

PrintMsg( 'SHORT, SolverMsg["SrchLinEq"] ),

/* Make a list of all remaining variables */
NewVars : listofvars( Equations ),

if Solver_Find_All_Linear_Vars then (

  /* The following rather weird commands put the variables into the order */
  /* [ variables to be currently solved for, other variables ]. If the    */
  /* current variables are linear variables then it is more likely that   */
  /* they will have explicit solutions if they are located in the left    */
  /* half of the system of equations. Otherwise the will more likely be   */
  /* used as parameters of the null space (if there is one).              */

  ActiveVars : append(
    Intersection( Variables, NewVars ),
    SetDifference( NewVars, append( Variables, Parameters ) )
  ),
  LinearEqs : Equations,
  Equations : []
)
else (
  ActiveVars : Intersection( Variables, NewVars ),
  LinearEqs : [],
  for i thru length( Equations ) do
    if
      not Empty( Intersection( listofvars( Equations[i] ), ActiveVars ) )
    then (
      LinearEqs : endcons( Equations[i], LinearEqs ),
      Equations[i] : []
    ),
  Equations : delete( [], Equations )
),

PrintMsg( 'SHORT, SolverMsg["wrt"], ActiveVars ),

/* Set up the complete coefficient matrix w.r.t. all variables */
CoeffMatrix : ComplCoeffMatrix( LinearEqs, ActiveVars ),

/* The valuation matrix contains a 1 at each Position where a variable */
/* appears in a nonlinear form, and 0's otherwise.                     */
```

```
ValuationMatrix : matrixmap(
  lambda(
    [ x ],
    if x = 'false then
      1
    else
      0
  ),
  CoeffMatrix
),

/* EqValuation contains the number of nonlinear variables for each eq. */
EqValuation : map(
  lambda( [ Row ], apply( "+", Row ) ),
  ListMatrix( ValuationMatrix )
),

/* VarValuation contains for all vars the number of equations in which */
/* var[i] appears in a nonlinear form.                                 */
VarValuation : map(
  lambda( [ Row ], apply( "+", Row ) ),
  ListMatrix( transpose( ValuationMatrix ) )
),

LinEqNos  : makelist( i, i, 1, NumEqs : length( EqValuation ) ),
LinVarNos : makelist( i, i, 1, NumVars : length( VarValuation ) ),


/* Remove nonlinear equations and/or variables until only a linear    */
/* block remains, i.e. for all i, j VarValuation[i] = 0 and           */
/* EqValuation[j] = 0.                                                */

while
  ( apply( "+", VarValuation ) # 0 )
  and
  ( apply( "+", EqValuation ) # 0 )
do (

  /* Determine maximum equation valuation and number of corresponding */
  /* equation.                                                        */

  me : 0,
  MaxValEq : -1,
  for i thru NumEqs do
    if EqValuation[i] > MaxValEq then (
      me : i,
      MaxValEq : mode_identity( fixnum, EqValuation[i] )
    ),
```

```
  /* Determine maximum variable valuation and number of corresponding */
  /* variable.                                                        */

  mv : 0,
  MaxValVar : -1,
  for j thru NumVars do
    if VarValuation[j] > MaxValVar then (
      mv : j,
      MaxValVar : mode_identity( fixnum, VarValuation[j] )
    ),


  if apply( Solver_Del_Eq, [ MaxValEq, MaxValVar ] ) then (
    i : me,

    for j thru NumVars do (
      VarValuation[j] : VarValuation[j] - ValuationMatrix[i, j],
      ValuationMatrix[i, j] : 0
    ),

    /* Mark equation as deleted */
    EqValuation[i] : 0,
    LinEqNos[i]    : 0
  )
  else (
    j : mv,

    for i thru NumEqs do (
      EqValuation[i] : EqValuation[i] - ValuationMatrix[i, j],
      ValuationMatrix[i, j] : 0
    ),

    /* Mark variable as deleted */
    VarValuation[j] : 0,
    LinVarNos[j]    : 0
  )

), /* END while */

/* Make list of linear equations */
LinEqNos  : delete( 0, LinEqNos ),

/* Make list of linear variables */
LinVarNos : delete( 0, LinVarNos ),

if Empty( LinEqNos ) or Empty( LinVarNos ) then (
  PrintMsg( 'SHORT, SolverMsg["NoLinEqs"] ),
```

```
  return( [ false, Solutions, append( LinearEqs, Equations ), Variables ] )
),

/* Extract linear equations. append all nonlinear equations to Equations */
/* again.                                                                */
for i thru length( LinearEqs ) do
  if not member( i, LinEqNos ) then (
    Equations : endcons( LinearEqs[i], Equations ),
    LinearEqs[i] : []
  ),

LinearEqs : delete( [], LinearEqs ),


/* Extract linear variables. Since the extraction of linear equations may */
/* also have removed linear variables (the coefficients are now 0) it is  */
/* necessary to intersect the set of the linear variables with the set of */
/* those variables which actually appear in the linear equations.         */

LinearVars : Intersection(
  map( lambda( [ i ], ActiveVars[i] ), LinVarNos ),
  listofvars( LinearEqs )
),

/* By analogy, the same applies to the linear equations. Thus, keep only  */
/* those equations which still contain any of the linear variables.       */

for i thru length( LinearEqs ) do
  if DisjointP( listofvars( LinearEqs[i] ), LinearVars ) then (
    Equations : endcons( LinearEqs[i], Equations ),
    LinearEqs[i] : []
  ),

LinearEqs : delete( [], LinearEqs ),

/* Return if no linear equations are left */

if Empty( LinearVars ) or Empty( LinearEqs ) then (
  PrintMsg( 'SHORT, SolverMsg["NoLinEqs"] ),
  return( [ false, Solutions, append( LinearEqs, Equations ), Variables ] )
),

PrintMsg(
  'SHORT,
  SolverMsg["Found"], length( LinearEqs ), SolverMsg["LinEqs"],
  length( LinearVars ), SolverMsg["LinVars"]
),
PrintMsg( 'SHORT,  SolverMsg["VarsAre"],  LinearVars ),
```

```
PrintMsg( 'DETAIL, SolverMsg["EqsAre"],   LinearEqs ),
PrintMsg( 'SHORT,  SolverMsg["SolvLinEq"] ),

rhsExpressions : [],
Solve_Inconsistent_Eqn_Nos : [ 0 ],

while not Empty( Solve_Inconsistent_Eqn_Nos ) do (

  /* solve the linear equations */
  LinearSolutions : LinsolveM( LinearEqs , LinearVars ),

  /* Check for "inconsistent" equations. */
  if not Empty( Solve_Inconsistent_Eqn_Nos ) then (

    PrintMsg( 'DEBUG, SolverMsg["Incons"], Solve_Inconsistent_Eqn_Nos ),

    /* Remove "inconsistent" equations */
    for i in Solve_Inconsistent_Eqn_Nos  do
        LinearEqs[i] : [],

    LinearEqs : delete( [], LinearEqs ),

    /* append rhs = 0 to Solver_Assumptions if rhs contains only */
    /* parameters.                                               */
    rhsExpressions : ParamConsistency(
      Solve_Inconsistent_Terms, Parameters
    )

  )

),

/* append rhs's which have led to "inconsistencies" but still   */
/* contain variables to the list of equations.                  */
Equations : append( Equations, rhsExpressions ),

PrintMsg( 'DETAIL, SolverMsg["Solutions"], LinearSolutions ),

/* Insert the solutions from linsolve into the remaining equations */
Equations : ParamConsistency(
    fullratsimp( ev( Equations, LinearSolutions ) ),
    Parameters
),

/* append the linear solutions to the list of solutions */
Solutions : append( Solutions, LinearSolutions ),

/* append all variables to the working list which appear on the rhs's of  */
```

```
    /* the linear solutions. delete all variables which have been solved for. */

    /* Linear variables for which a solution has been obtained. */
    LinSolVars : map( 'lhs, LinearSolutions ),

    /* Linear variables which are free parameters of the null space. */
    LinearVars : SetDifference( LinearVars, LinSolVars ),

    /* append all those variables which are parameters of the null space of */
    /* the linear equations and which do not appear in the remaining        */
    /* equations to the list of parameters.                                 */

    for Var in LinearVars do
      if freeof( Var, Equations ) then (
        PrintMsg( 'SHORT, SolverMsg["FreeVar2Par"], Var ),
        Parameters : endcons( Var, Parameters ),
        Variables  : delete( Var, Variables )
      ),

    NewVars : SetDifference(
      listofvars( map( 'rhs, LinearSolutions ) ),
      Parameters
    ),

    Variables : Union(
      SetDifference( Variables, LinSolVars ),
      NewVars
    ),

    return( [ true, Solutions, Equations, Variables ] )
  )
)$


/*************************************************************************/
/* The following strategies decide whether the linear solver should delete a  */
/* nonlinear equation or a nonlinear variable from the system while searching */
/* for linear subblocks of equations.                                    */
/*************************************************************************/

define_variable( EqWasLast, false, boolean )$

MakeSquareLinearBlocks( ValEq, ValVar ) := (

  mode_declare(
    [ ValEq, ValVar ], fixnum
  ),
```

```
  if ValEq = ValVar then
    EqWasLast : not EqWasLast
  else
    if ValEq > ValVar then
      EqWasLast : true
    else
      EqWasLast : false
)$


DelEqBeforeVar( ValEq, ValVar ) := (

  mode_declare(
    [ ValEq, ValVar ], fixnum
  ),

  if ValEq >= ValVar then
    true
  else
    false
)$


/*****************************************************************************/
/* ComplCoeffMatrix returns a matrix whose row size is equal to the number of */
/* equations and whose column size is equal to the number of variables. The   */
/* entry at Position [i,j] is RatCoeff( equation[i], variable[j] ) if         */
/* equation[i] is linear w.r.t. variable[j] and 'false if equation[i] is      */
/* nonlinear w.r.t. variable[j].                                              */
/*****************************************************************************/

ComplCoeffMatrix( Eqs, ActiveVars ) := (

  mode_declare(
    [ Eqs, ActiveVars ], list
  ),

  block(


    apply(

      'matrix,

      /* for each equation do */
      map(

        lambda(
```

```
          [ Eq ],

          /* for each variable do */
          map(
            lambda(
              [ Var ],
              block(
                [ rc ],
                rc  : LinCoeff( Eq, Var ),

                /* Return the RatCoeff only if it contains none of the active */
                /* variables or if the equation doesn't contain var at all.   */

                if
                  ( ( rc # 0 ) and DisjointP( listofvars( rc ), ActiveVars ) )
                  or
                  freeof( Var, Eq )
                then
                  rc
                else
                  false
              )
            ),

            ActiveVars
          ) /* END lambda( [ Var ] ) */

        ), /* END lambda( [ Eq ] ) */


        /* map target: Transform all equations into homogeneous form. */

        map(
          lambda(
            [ Eq ],
            fullratsimp( expand( lhs( Eq ) - rhs( Eq ) ) )
          ),
          Eqs
        )


      ) /* END map( lambda( [ Eq ] ) ) */

    ) /* END apply */
  )
)$
```

```
/*****************************************************************************/
/* LinCoeff returns the linear coefficient of Var within Eq if Var appears   */
/* raised to the first power only.                                           */
/*****************************************************************************/

LinCoeff( Eq, Var ) := (

  mode_declare(
    [ Eq, Var ], any
  ),

  block(

    [ BCoeff ],

    mode_declare(
      BCoeff, list
    ),

    if ListOfPowers( [ Eq ], Var ) = [ 1 ] then (
      BCoeff : bothcoeff( Eq, Var ),
      if freeof( Var, second( BCoeff ) ) then
        return( first( BCoeff ) )
    ),

    return( 0 )
  )
)$


/*****************************************************************************/
/* ValuationSolver                                                           */
/*****************************************************************************/

ValuationSolver( Solutions, Equations, Variables, Parameters ) := (

  mode_declare(
    [ Solutions, Equations, Variables, Parameters ], list
  ),

  block(

    [
      VarPaths, ValMatrix, Eq, Var, Trans, TempEq, TransEq,
      SolveOrder, SolveInfo, Transform, Solution, SolCheck,
      Status, Solved, Failed, UniqueSol, TryToSolve, CheckSol,
      i, k
    ],
```

```
mode_declare(
  [ VarPaths, ValMatrix, Eq, Var, Trans, TempEq, TransEq ], any,
  [ SolveOrder, SolveInfo, Transform, Solution, SolCheck ], list,
  [ Status, Solved, Failed, UniqueSol, TryToSolve, CheckSol ], boolean,
  [ i, k ], fixnum
),


UniqueSol : true,

LOOP,

PrintMsg( 'SHORT, SolverMsg["Chk4RemEq"] ),

if Empty( Equations ) then (

  if Empty( Variables ) then
    PrintMsg( 'SHORT, SolverMsg["AllSolved"] )
  else
    PrintMsg( 'SHORT, SolverMsg["NoEqLeft"], Variables ),

  Status : false

)
else if Empty( Variables ) then (
  PrintMsg( 'SHORT, SolverMsg["EqLeft"] ),
  Status : false
)
else (

  PrintMsg(
    'SHORT,
    length( Equations ), SolverMsg["Eqs"],
    length( Variables ), SolverMsg["Vars"]
  ),
  PrintMsg( 'DETAIL, SolverMsg["VarsAre"], Variables ),
  PrintMsg( 'DEBUG, SolverMsg["EqsAre"], Equations ),

  /* Dump solutions and remaining equations to file if requested. */
  if Solver_Dump_To_File then
    DumpToFile( Solutions, Equations, Variables ),

  if ( length( Variables ) = 1 ) and ( length( Equations ) = 1 ) then
    SolveOrder : [ [ 1, 1, "(irrelevant)" ] ]

  else (
    PrintMsg( 'SHORT, SolverMsg["ValStrat"] ),
```

```
  /* Set up the valuation matrices. */
  VarPaths  : OccurrenceMatrix( Equations, Variables ),
  ValMatrix : ValuationMatrix( Equations, Variables ),

  /* Determine an order by which the equations should be solved. */
  SolveOrder : apply( Solver_Valuation_Strategy, [ VarPaths, ValMatrix ] )
),

Solved : false,

unless Solved or Empty( SolveOrder ) do (

  SolveInfo : Pop( SolveOrder ),
  if listp(SolveInfo[1]) then SolveInfo : first(SolveInfo),
  k   : mode_identity( fixnum, first( SolveInfo ) ),
  Eq  : part( Equations, k ),
  Var : Variables[ second( SolveInfo ) ],

  PrintMsg(
    'SHORT,
    SolverMsg["TrySolveEq"], k, SolverMsg["ForVar"], Var
  ),
  PrintMsg( 'SHORT, SolverMsg["Valuation"], third( SolveInfo ) ),
  PrintMsg( 'DETAIL, SolverMsg["EqIs"], Eq = 0 ),

  TryToSolve : true,
  CheckSol   : true,
  Transform  : copylist( Solver_Transforms ),

  /* Do the solver break test to check whether it is worth */
  /* attempting to solve the equation at all.              */
  Failed : apply( Solver_Break_Test, [Eq, Var, third( SolveInfo )] ),

  unless Solved or Failed do (

    /* Try to solve the selected equation */
    if TryToSolve then
      Solution : solve( Eq, Var ),

    /* Check if the equation was solved correctly */
    if CheckSol then (
      PrintMsg( 'SHORT, SolverMsg["CheckSol"] ),
      SolCheck : SolutionOK( Solution, Var ),

      PrintMsg( 'DETAIL, SolverMsg["Solutions"], Solution )
    )
    else
```

```
  SolCheck : [ false ],

/* All solutions OK? */
if member( true, SolCheck ) then (
  PrintMsg( 'SHORT, SolverMsg["SolOK"] ),
  Solved : true
)

/* If not, apply transformations */
else (
  if CheckSol then
    PrintMsg( 'SHORT, SolverMsg["SolNotOK"] ),

  /* Give up if no transformations are left */
  if Empty( Transform ) then (
    PrintMsg( 'SHORT, SolverMsg["GiveUp"] ),
    Failed : true
  )

  else (
    /* Retrieve one transformation function */
    Trans : Pop( Transform ),
    PrintMsg( 'SHORT,  SolverMsg["AppTrans"], Trans ),

    /* and apply it to the equation, the variable, and the solution */
    TransEq : apply( Trans, [ Eq, Var, Solution ] ),

    /* The transformation should return an equation as its function */
    /* value. However, if no reasonable transformation of the       */
    /* equation was possible then the SOLVE function should not be  */
    /* tried again. Hence, to signal a failure, the transformation  */
    /* must return an empty list, which will instruct the Solver to */
    /* try the next transformation instead. In addition, the        */
    /* transformation may itself take care of solving the equation. */
    /* It must then return a list of solutions:                     */
    /*   [ var = solution_1, var = solution_2, ... ]                */

    /* Did the transformation fail? */
    if TransEq = [] then (
      PrintMsg( 'SHORT, SolverMsg["TransFail"] ),

      /* Instruct the Solver to try the next transformation */
      TryToSolve : false,
      CheckSol   : false
    )

    /* Did it solve the equation by itself? */
    else if listp( TransEq ) then (
```

```
            PrintMsg( 'SHORT, SolverMsg["TransSolv"] ),
            Solution : TransEq,

            /* Instruct the Solver not to call SOLVE again */
            TryToSolve : false,
            CheckSol   : true
          )

          /* Transformation thinks it has succeeded, so try again */
          else (
            Eq : TransEq,
            PrintMsg( 'DETAIL, SolverMsg["ResTrans"], Eq = 0 ),
            PrintMsg( 'SHORT,  SolverMsg["RetryTrans"] ),

            TryToSolve : true,
            CheckSol   : true
          )

        ) /* END if Empty( Transform ) else */

      ) /* if member( true, SolCheck ) else */

    ) /* END unless Solved or Failed */

  ), /* END unless Solved or Empty( SolveOrder ) */

  if Solved then (
    if length( Solution ) > 1 then
      PrintMsg( 'SHORT, SolverMsg["NotUnique"] ),

    if member( false, SolCheck ) then
      PrintMsg( 'SHORT, SolverMsg["SolsLost"] ),

    /* Remove solved equation from list of equations. Store it in TempEq  */
    /* so it can be appended to Equations again if the consistency check  */
    /* fails.                                                             */
    TempEq : part( Equations, k ),
    Equations : delete( [], Set_Element( Equations, k, [] ) ),

    /* Check solutions for consistency with remaining equations. */
    for i thru length( Solution ) do (

      if part( SolCheck, i ) then (

        PrintMsg(
          'DETAIL, SolverMsg["Solution"], i, SolverMsg["ForVar"], Var
        ),
```

```
    if not ParamConsistency(
      fullratsimp( ev( Equations, part( Solution, i ) ) ),
      Parameters,
      'CONTINUE
    ) then (
      PrintMsg( 'SHORT, SolverMsg["Contradict"], part( Solution, i ) ),
      Set_Element( Solution, i, 'INCONSISTENT_PATH )
    )

  )
  else (
    PrintMsg( 'DETAIL, SolverMsg["Dropped"], part( Solution, i ) ),
    Set_Element( Solution, i, [] )
  )

),

/* delete all implicit or empty solutions */
Solution : delete( [], Solution ),

if Empty( Solution ) then (
  PrintMsg( 'SHORT, SolverMsg["NoValidSol"], Var ),

  Equations : endcons( TempEq, Equations ),
  Solved : false
)

/* Check if there are any consistent solutions */
else if not member(
  true,
  map(
    lambda(
      [x],
      if x = 'INCONSISTENT_PATH then
        false
      else
        true
    ),
    Solution
  )
)
then (
  PrintMsg( 'SHORT, SolverMsg["NoConsSol"], Var ),

  Solutions : endcons( 'INCONSISTENT_PATH, Solutions ),
  Solved : false
)
```

```
/* append consistent solutions to the solution list */
else (
  PrintMsg( 'DETAIL, SolverMsg["ConsSol"], Var, ":", Solution ),

  Variables : delete( Var, Variables ),

  /* If the solution is unique or if there's only one consistent */
  /* solution then ...                                           */
  if length( Solution ) = 1 then (

    /* ... store it, insert it into the remaining equations, and */
    /* add its rhs variables to the list of unknowns.            */
    Solutions : append( Solutions, Solution ),
    Equations : copylist(
      fullratsimp( ev( Equations, Solution ) )
    ),
    Variables : Union(
      Variables,
      SetDifference(
        listofvars( rhs( first( Solution ) ) ),
        Parameters
      )
    )
  )

  else /* length( Solution ) > 1, call ValuationSolver recursively */

    block(
      [ MultipleSolutions, RSolutions, RVars, REqs, Sol, Stat ],

      mode_declare(
        [ MultipleSolutions, RSolutions, RVars, REqs ], list,
        Sol, any,
        Stat, boolean
      ),

      MultipleSolutions : [],

      for Sol in Solution do (

        map(
          lambda([x,y], x::y),
          [ 'Stat, 'RSolutions, 'REqs, 'RVars ],
          ValuationSolver(
            [ Sol ],
            copylist( fullratsimp( ev( Equations, Sol ) ) ),
            Union(
              Variables,
```

```
                    SetDifference( listofvars( rhs( Sol ) ), Parameters )
                  ),
                  Parameters
                )
              ),

              MultipleSolutions : endcons( RSolutions, MultipleSolutions )

            ), /* END for Sol */

            Solutions : endcons( MultipleSolutions, Solutions ),

            UniqueSol : false
          ) /* END block */

      ) /* END if Empty( Solution ) */

    )

    else (

      /* append remaining equations to the solutions if no further */
      /* solutions could be determined.                            */

      for e in Equations do (
        if is( equal( e, 0 ) ) # 'unknown then (
          PrintMsg( SHORT, "Inconsistent equation", e = 0),
          TerminateSolver()
        )
      ),

      Solutions : endcons( [ Equations ], Solutions )
    ), /* END if Solved */

    Status : Solved
  ),

  if Status and UniqueSol then
    go( LOOP )
  else
    return( [ Status, Solutions, Equations, Variables ] )
 )
)$


/*****************************************************************************/
/* SolutionOK checks whether the result of a call to the solve function is   */
/* indeed a solution of the form  var = expression_free_of_var.              */
```

```
/*****************************************************************************/

SolutionOK( Solution, Var ) := (

  mode_declare(
    [ Solution, Var ], any
  ),

  if listp( Solution ) then

    /* List of solutions must not be empty. */
    if Empty( Solution ) then
      [ false ]

    else

      /* Check if the lhs of each solution is equal to var and make sure */
      /* that var does not appear on the rhs's.                          */
      map(
        lambda(
          [ Sol ],
          ( lhs( Sol ) = Var ) and freeof( Var, rhs( Sol ) )
        ),
        Solution
      )

  else
    [ false ]

)$


/*****************************************************************************/
/* ValuationMatrix generates a matrix of valuations with respect to each     */
/* equation and each variable.                                               */
/*****************************************************************************/

ValuationMatrix( Equations, Variables ) := (

  mode_declare(
    [ Equations, Variables ], list
  ),

  genmatrix(
    lambda( [ i, j ], Valuation( Equations[i], Variables[j] ) ),
    length( Equations ), length( Variables )
  )
)$
```

```
/*****************************************************************************/
/* Operator valuation factors for expression valuation.                      */
/*****************************************************************************/

SetProp( 'sin,   'Valuation, 10 )$
SetProp( 'cos,   'Valuation, 10 )$
SetProp( 'tan,   'Valuation, 10 )$
SetProp( 'asin,  'Valuation, 12 )$
SetProp( 'acos,  'Valuation, 12 )$
SetProp( 'atan,  'Valuation, 12 )$
SetProp( 'sinh,  'Valuation, 12 )$
SetProp( 'cosh,  'Valuation, 12 )$
SetProp( 'tanh,  'Valuation, 12 )$
SetProp( 'asinh, 'Valuation, 12 )$
SetProp( 'acosh, 'Valuation, 12 )$
SetProp( 'atanh, 'Valuation, 12 )$
SetProp( "+",    'Valuation,  1 )$
SetProp( "-",    'Valuation,  1 )$
SetProp( "*",    'Valuation,  4 )$
SetProp( "/",    'Valuation,  4 )$
SetProp( "^",    'Valuation, 10 )$
SetProp( 'sqrt,  'Valuation, 10 )$
SetProp( 'exp,   'Valuation, 10 )$
SetProp( 'log,   'Valuation, 10 )$


/*****************************************************************************/
/* With SetValuation, the operator valuation factors can be redefined.       */
/*****************************************************************************/

SetValuation( Operator, Valuation ) :=
  SetProp( Operator, 'Valuation, Valuation )$


/*****************************************************************************/
/* Valuation measures the complexity of an expression with respect to Var by */
/* weighting the operator tree representation of Expr.                       */
/*****************************************************************************/

Valuation( Expr, Var ) := (

  mode_declare(
    [ Expr, Var ], any
  ),
```

```
  block(

    [ OpFactor ],

    mode_declare(
      OpFactor, fixnum
    ),

    /* Return zero if Expr does not contain Var. */
    if freeof( Var, Expr ) then
      return( 0 )

    else
      /* Return 1 if Expr is an atom, i.e. Expr = Var. */
      if atom( Expr ) then
        return( 1 )

      /* If Expr is an algebraic expression then retrieve the valuation    */
      /* factor associated with the operator of Expr and recursively apply */
      /* the valuation function to each subexpression of Expr.             */
      else (

        if (
            OpFactor : mode_identity( fixnum, get( op( Expr ), 'Valuation ) )
           ) = false
        then
          OpFactor : Solver_Default_Valuation,

        return(
          OpFactor * apply(
            "+",
            map(
              lambda( [ SubExpr ], Valuation( SubExpr, Var ) ),
              substpart( "[", Expr, 0 )
            )
          )
        )

      ) /* END if atom */

  )
)$


/*****************************************************************************/
/* OccurrenceMatrix sets up a matrix in which the number of occurrences of   */
/* each variable in each equation is counted.                                */
/*****************************************************************************/
```

```
OccurrenceMatrix( Equations, Variables ) := (

  mode_declare(
    [ Equations, Variables ], list
  ),

  genmatrix(
    lambda( [ i, j ], Occurences( Equations[i], Variables[j] ) ),
    length( Equations ), length( Variables )
  )
)$


/*****************************************************************************/
/* Occurences counts the number of occurences of Var in Expr, i.e. the number */
/* of paths to distinct occurrences of the atom Var in the internal tree      */
/* representation of Expr.                                                    */
/*****************************************************************************/

Occurences( Expr, Var ) := (

  mode_declare(
    [ Expr, Var ], any
  ),

  if atom( Expr ) then

    if Expr = Var then
      1
    else
      0

  else
    apply(
      "+",
      map(
        lambda( [ SubExpr], Occurences( SubExpr, Var ) ),
        substpart( "[", Expr, 0)
      )
    )
)$


/*****************************************************************************/
/* MinVarPathsFirst tries to find variables which can be easily isolated.    */
/* These are variables which appear only once in an entire expression tree   */
/* (= 1 in OccurrenceMatrix).                                                */
```

```
/*****************************************************************************/

MinVarPathsFirst( OccMat, ValMat ) := (

  mode_declare(
    [ OccMat, ValMat ], any
  ),

  block(

    [
      SolveOrder, SolveOrder1, SumVarPaths,
      i, j, v, ne, nv
    ],

    mode_declare(
      [ SolverOrder, SolveOrder1, SumVarPaths ], list,
      [ i, j, v, ne, nv, Function( RowSize, ColSize, Position ) ], fixnum
    ),

    SolveOrder : [],

    ne : RowSize( OccMat ),
    nv : ColSize( OccMat ),

    SumVarPaths : map(
      lambda( [ Row ], apply( "+", Row ) ),
      ListMatrix( OccMat )
    ),

    /* Search for equations which contain only one variable in one path. */
    for i thru length( SumVarPaths ) do
      if SumVarPaths[i] = 1 then (
        SolveOrder : endcons(
          [ i, j : Position( 1, OccMat[i] ), ValMat[i, j] ],
          SolveOrder
        ),
        /* Mark eq/var Position as used. */
        OccMat[i, j] : 0,
        ValMat[i, j] : 0
      ),

    /* Sort SolveOrder by least valuation. */
    if not Empty( SolveOrder ) then
      SolveOrder : SortSolveOrder( SolveOrder ),

    /* Find all variables with only one path in the expression tree. */
    SolveOrder1 : [],
```

```
    for i thru ne do
      for j thru nv do
        if OccMat[i, j] = 1 then (
          SolveOrder1 : endcons( [ i, j, ValMat[i, j] ], SolveOrder1 ),
          OccMat[i, j] : 0,
          ValMat[i, j] : 0
        ),

    /* Sort variables by least valuation. */
    if not Empty( SolveOrder1 ) then
      SolveOrder : append(
        SolveOrder,
        SortSolveOrder( SolveOrder1 )
      ),

    /* append additional candidates if necessary. */
    if length( SolveOrder ) < Solver_Max_Len_Val_Order then (

      SolveOrder1 : [],
      for i thru ne do
        for j thru nv do
          if ( v : mode_identity( fixnum, ValMat[i, j] ) ) # 0 then
            SolveOrder1 : endcons( [ i, j, v ], SolveOrder1 ),

      SolveOrder : append(
        SolveOrder,
        SortSolveOrder( SolveOrder1 )
      ),

      /* Return only as many candidates as given by Solver_Max_Len_Val_Order */
      if ( i : length( SolveOrder ) ) > Solver_Max_Len_Val_Order then
        SolveOrder : rest(
          SolveOrder, Solver_Max_Len_Val_Order - i
        )
    ),

    return( SolveOrder )
  )
)$


/*****************************************************************************/
/* PostProcess does all the postprocessing needed to display the results.    */
/* This includes expansion of the solution list hierarchies, backsubsitution, */
/* and extraction of the variables which the user explicitly asked for.      */
/*****************************************************************************/

PostProcess( Solutions, UserVars, Expressions, PowerSubst ) := (
```

```
mode_declare(
  [ Solutions, UserVars, Expressions, PowerSubst ], list
),

block(
  [
    SolSet, UsrSolSet, UserSolutions, UnsolvedEqs, InternalSols,
    Var, TempVar, EvalVar,
    i
  ],

  mode_declare(
    [ SolSet, UsrSolSet, UserSolutions, UnsolvedEqs, InternalSols ], list,
    [ Var, TempVar, EvalVar ], any,
    i, fixnum
  ),


  PrintMsg( 'SHORT, SolverMsg["PostPr"] ),

  /* first of all, Flatten the solution list hierarchy and drop all */
  /* inconsistent solution paths.                                   */
  Solutions : sublist(
    ExpandSolutionHierarchy( Solutions ),
    lambda( [Set], last( Set ) # 'INCONSISTENT_PATH )
  ),

  /* Return an empty list if no consistent solution paths are left. */
  if Empty( Solutions ) then
    return( [] ),

  /* Do the backsubstitutions. */
  if not Empty( Solutions ) then (

    UserSolutions : [],
    i : 0,

    for SolSet in Solutions do (

      i : i + 1,
      PrintMsg( 'SHORT, SolverMsg["SolSet"], i ),

      UsrSolSet   : [],

      /* Extract the unsolved equations */
      UnsolvedEqs : sublist( SolSet, lambda( [x], not EquationP( x ) ) ),
```

```
/* and the solutions. */
SolSet : sublist( SolSet, 'EquationP ),

/* If no complete backsubstitution is requested then variables on the */
/* right-hand sides of the solutions will only be substituted if they */
/* do not belong to the variables specified in the command line.      */
if not Solver_Backsubst then
  InternalSols : sublist(
    SolSet,
    lambda( [x], not member( lhs( x ), UserVars ) )
  ),

/* Evaluate all variables and expressions with the solutions. */
for Var in append( UserVars, Expressions ) do (

  EvalVar : if member( Var, map(lhs, SolSet) ) or not atom( Var ) then

    /* There may be errors when indeterminate expressions are */
    /* encountered.                                           */

    errcatch(
      if Solver_Backsubst then
        ev( Var, SolSet, infeval )
      else (
        TempVar : ev( Var, SolSet ),
        ev( TempVar, InternalSols, infeval )
      )
    )

  else
    [],

  if EvalVar # [] then
    UsrSolSet : endcons( Var = EvalVar[1], UsrSolSet )
  else
    PrintMsg( 'SHORT, SolverMsg["NoSol"], Var )
),


/* append the unsolved equations. */
if not Empty( UnsolvedEqs ) then
  UsrSolSet : endcons( UnsolvedEqs, UsrSolSet ),

if Solver_RatSimp_Sols then
  UsrSolSet : errcatch( fullratsimp( UsrSolSet ) )
else
  UsrSolSet : [UsrSolSet],
```

```
        if UsrSolSet = [] then
          PrintMsg( 'SHORT, SolverMsg["SolSetDrp"] )
        else
          UserSolutions : endcons( UsrSolSet[1], UserSolutions )

      ), /* END for SolSet */

      if Solver_Dump_To_File then
        DumpToFile( UserSolutions, [], [] )

    ), /* END if not Empty( Solutions ) */


    return( UserSolutions )

  ) /* END block */
)$


/*****************************************************************************/
/* ExpandSolutionHierarchy transforms the hierarchically structured list of  */
/* solutions into a list of flat lists of solutions.                         */
/*****************************************************************************/

ExpandSolutionHierarchy( Solutions ) := (

  mode_declare(
    Solutions, list
  ),

  block(
    [ FlatSolutions ],

    mode_declare(
      FlatSolutions, list
    ),

    if length( Solutions ) = 0 then return ( [] )
    /* listp = true indicates an additional recursion level */
    else if listp( last( Solutions ) ) then (

      FlatSolutions : rest( Solutions, -1 ),

      return(

        map(
          lambda( [ x ], append( FlatSolutions, x ) ),
```

```
          apply(
            'append,
            map( 'ExpandSolutionHierarchy, last( Solutions ) )
          )
        )

      )

    )

    else
      return( [ Solutions ] )

  )
)$


/*****************************************************************************/
/* DumpToFile dumps the current set of solutions, equations and variables to  */
/* the file <Solver_Dump_File>.                                               */
/*****************************************************************************/

DumpToFile( Sols, Eqs, Vars ) := (

  mode_declare(
    [ Sols, Eqs, Vars ], list
  ),

  block(
    [ Solutions, Equations, Variables ],

    PrintMsg( 'SHORT, SolverMsg["Dump"], Solver_Dump_File ),

    apply(
      'StringOut,
      [
        Solver_Dump_File,
        'Solutions = Sols,
        'Equations = Eqs,
        'Variables = Vars
      ]
    )
  )
)$

tma():=trace(
TerminateSolver,
SetupSolver,
```

```
ParamConsistency,
SolverAssumeZero,
ListOfPowers,
ImmediateAssignments,
LinearSolver,
MakeSquareLinearBlocks,
DelEqBeforeVar,
ComplCoeffMatrix,
LinCoeff,
ValuationSolver,
SolutionOK,
ValuationMatrix,
SetValuation,
Valuation,
OccurrenceMatrix,
Occurences,
MinVarPathsFirst,
PostProcess,
ExpandSolutionHierarchy,
DumpToFile)$
```

ESTRAGON Sometimes I wonder if it wouldn't be better to diverge.

WLADIMIR You wouldn't get far.

ESTRAGON That would really be a shame ... Not true, Didi, That would be a real shame? ... If you think about the beauty of the path ... And about the goodness of the companions... Isn't that right, Didi?