Technische Universität Braunschweig

Studienarbeit

Symbolische Lösung linearer und nichtlinearer parametrisierter Gleichungssysteme

Eckhard Hennig

Betreuer: Dr.-Ing. Ralf Sommer

Braunschweig, August 1994

Eidesstattliche Erklärung

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen Hilfsmittel als die darin angegebenen benutzt habe.

Braunschweig, 30. August 1994

(Eckhard Hennig)

Danksagung

Zum Gelingen dieser Arbeit haben viele Personen und Institutionen maßgeblich beigetragen, denen ich hiermit meinen Dank aussprechen möchte:

- Richard Petti und Jeffrey P. Golden von Macsyma, Inc. (USA) für ihr Interesse an den Arbeiten, das Bereitstellen von Macsyma-Lizenzen und die Modifikation der `LINSOLVE`-Funktion,
- dem Zentrum für Mikroelektronik der Universität Kaiserslautern, insbesondere Dr. Peter Conradi und Uwe Wasenmüller, für die Unterstützung des Projekts,
- Clemens, Frank und Michael für das erstklassige WG-Leben und speziell Michael für die zeitweilige Überlassung seines Rechners,
- meinen Eltern und Großeltern für ihre stetige Unterstützung während meines Studiums
- und ganz besonders meinem Freund Dr. Ralf Sommer für die hervorragende Zusammenarbeit und die gemeinsamen Unternehmungen der vergangenen drei Jahre.

Eckhard Hennig
Braunschweig, August 1994

Zusammenfassung

Technische Dimensionierungsaufgaben erfordern häufig die Lösung von Gleichungssystemen, die ein Objekt mathematisch beschreiben, nach den Werten der funktionsbestimmenden Bauelementeparameter. Zur analytischen Lösung kleinerer Dimensionierungsprobleme bietet sich der Einsatz kommerzieller Computeralgebrasysteme wie Macsyma [MAC_94] an, die in der Lage sind, umfangreiche Gleichungen algebraisch zu manipulieren und nach ihren Variablen symbolisch aufzulösen.

Trotz ihrer hohen Leistungsfähigkeit sind diese Systeme jedoch meist bereits überfordert, wenn lineare oder schwach nichtlineare, parametrisierte Gleichungssysteme nach nur einer Teilmenge ihrer Variablen zu lösen oder zumindest symbolisch vorzuverarbeiten sind. Um solche typischerweise bei Entwurfsaufgaben entstehenden Gleichungssystem behandeln zu können, wurde im Rahmen dieser Arbeit ein auf heuristischen Algorithmen basierender, universeller symbolischer Gleichungslöser entwickelt und in Macsyma implementiert. Das Programmodul `SOLVER` erweitert die Funktionalität der Macsyma-Befehle `SOLVE` und `LINSOLVE` zur symbolischen Lösung algebraischer Gleichungen bzw. linearer Gleichungssysteme um die Fähigkeit zur selektiven Lösung nichtlinearer, parametrisierter Systeme mit Freiheitsgraden.

Das erste Kapitel der vorliegenden Arbeit beschreibt einige Anwendungsbereiche symbolischer Dimensionierungsmethoden, die aus ihnen folgenden Anforderungen an einen symbolischen Gleichungslöser sowie die verwendeten heuristischen Algorithmen zur Extraktion linearer Gleichungen und zur Komplexitätsbewertung algebraischer Funktionen. Das zweite Kapitel enthält einen Überblick über die Struktur des *Solvers* und eine Anleitung zu seiner Benutzung. Im Anhang findet sich der Quelltext des in der internen höheren Programmiersprache von Macsyma implementierten Moduls `SOLVER.MAC`.

Inhaltsverzeichnis

1 Heuristische Algorithmen **1**

1.1 Einleitung 1

1.1.1 Numerische Verfahren im Vergleich mit symbolischen Methoden 1

1.1.2 Beispiele für Anwendungsgebiete symbolischer Dimensionierungsmethoden 2

1.2 Konventionelle Gleichungslöser 7

1.3 Anforderungen an einen symbolischen Gleichungslöser 8

1.4 Extraktion und Lösung linearer Gleichungen 10

1.4.1 Intuitive Vorgehensweisen zur Suche linearer Gleichungen 10

1.4.2 Ein heuristischer Algorithmus zur Suche linearer Gleichungen 11

1.4.3 Lösung der linearen Gleichungen 13

1.5 Bewertungsstrategien zur Lösung nichtlinearer Gleichungen 14

1.5.1 Einsetzverfahren für nichtlineare Gleichungssysteme 14

1.5.2 Heuristische Verfahren zur Komplexitätsbewertung algebraischer Ausdrücke 16

1.5.3 Aufstellung der Lösungsreihenfolge 19

2 Der Solver **21**

2.1 Die Struktur des Solvers 21

2.2 Die Module des Solvers 23

2.2.1 Der Solver Preprocessor 23

2.2.2 Der Immediate Assignment Solver 25

2.2.3 Der Linear Solver 26

2.2.4 Der Valuation Solver 28

2.2.5 Der Solver Postprocessor 31

2.3 Anwendung des Solvers 32

2.3.1 Kommandosyntax 32

2.3.2 Besonderheiten der Syntax von Gleichungen 33

2.3.3 Beispielaufrufe des Solvers 33

2.4 Die Optionen des Solvers 37

2.5 Benutzerspezifische Transformationsroutinen 40

2.6 Änderung der Operatorbewertungen 44

2.7 Benutzerspezifische Bewertungsstrategien 45

Literaturverzeichnis **47**

A Beispielrechnungen **49**

A.1 Dimensionierung des Stabzweischlags 49

A.2 Dimensionierung des Transistorverstärkers 57

B Programmlistings **65**

B.1 SOLVER.MAC 65

Kapitel 1

Heuristische Algorithmen zur symbolischen Lösung von Gleichungssystemen

1.1 Einleitung

1.1.1 Numerische Verfahren im Vergleich mit symbolischen Methoden

Die exakte Dimensionierung technischer Entwürfe für vorgegebene Spezifikationen erfordert häufig die Lösung nichtlinearer Gleichungssysteme nach funktionsbestimmenden Bauelementeparametern. Meist werden zur Bestimmung der gesuchten Variablen numerische Optimierverfahren eingesetzt, die jedoch bereits bei kleinen Entwurfsproblemen mit einer vergleichsweise geringen Zahl von Unbekannten nicht immer zuverlässig funktionieren. Probleme bereiten in erster Linie die exponentiell von der Variablenzahl abhängige Komplexitätsordnung vieler gängiger Optimieralgorithmen (ein Problem, das auch unter dem Namen *curse of dimensionality* [MIC_94] bekannt ist), die Wahl geeigneter Startwerte, das unerwünschte Auffinden lokaler statt globaler Optima und das prinzipbedingte Verhalten bei der Lösung von Gleichungssystemen mit Freiheitsgraden, bei der eine in der Realität nicht vorhandene Eindeutigkeit des Lösungsvektors vorgetäuscht werden kann.

Das bestmögliche Vorgehen zur Bewältigung einer Entwurfsaufgabe wäre grundsätzlich eine vollständige analytische Lösung des zugehörigen Gleichungssystems. Analytische Funktionen, die explizit die gesuchten Elementewerte als Funktionen der Spezifikationsparameter beschreiben, müßten lediglich einmal ermittelt werden und stünden dann zur beliebig häufigen Auswertung mit geänderten Parametern, z.B. innerhalb von Entwurfsdatenbanken von CAD-Systemen, zur Verfügung. Überdies verdeutlichen analytische Dimensionierungsformeln qualitativ funktionale Zusammenhänge zwischen den Elementewerten und den Spezifikationen und offenbaren in den betreffenden Fällen die Existenz von Freiheitsgraden. Da es jedoch keine allgemeingültigen Verfahren zur Lösung beliebiger nichtlinearer Gleichungssysteme gibt und in den Spezialfällen, in denen sich symbolische Lösungen berechnen lassen, die Komplexität der Resultate den per Handrechnung zu bewältigenden Rahmen weit übersteigt, spielt die analytische Behandlung auch nur kleiner Dimensionierungsprobleme bislang eine untergeordnete Rolle.

Mit Hilfe moderner, leistungsfähiger Computeralgebrasysteme, die in der Lage sind, symbolische Gleichungssysteme algebraisch zu manipulieren und nach beliebigen Variablen aufzulösen,

werden einige der oben genannten Problemklassen, insbesondere solche, die die Lösung von großteils linearen oder multivariaten polynomialen Systemen erfordern, dennoch der analytischen Bearbeitung zugänglich. Beispiele für entsprechende Anwendungen finden sich in vielen Bereichen der Ingenieurwissenschaften, unter anderem betrifft dies den Entwurf analoger elektronischer Schaltungen, regelungstechnische Problemstellungen, die technische Mechanik und die Robotik [PFA_94].

1.1.2 Beispiele für Anwendungsgebiete symbolischer Dimensionierungsmethoden

Anhand der folgenden zwei Beispiele sollen einige Anwendungsgebiete symbolischer Methoden demonstriert werden. Gleichzeitig sollen an ihnen exemplarisch die konkreten Anforderungen herausgearbeitet werden, denen bei der Entwicklung eines Konzepts für einen universellen Lösungsalgorithmus für symbolische Gleichungssysteme Rechnung getragen werden muß.

Beispiel 1.1

Bei dem ersten Beispiel handelt es sich um eine einfache Aufgabenstellung aus der technischen Mechanik. Gegeben sei der in Abbildung 1.1 dargestellte Stabzweischlag [BRO_88, S. 112], an dem unter dem Winkel γ gegenüber der Horizontalen die Kraft F im Punkt C angreift. Die Stäbe schließen mit der Horizontalen die Winkel α und β ein, die Höhe des durch die Stäbe aufgespannten Dreiecks sei mit c bezeichnet. Die Querschnittsflächen A_1 und A_2 der beiden Stäbe seien Quadrate mit den Seitenlängen h_1 und h_2. Das Elastizitätsmodul des verwendeten Werkstoffs sei E.

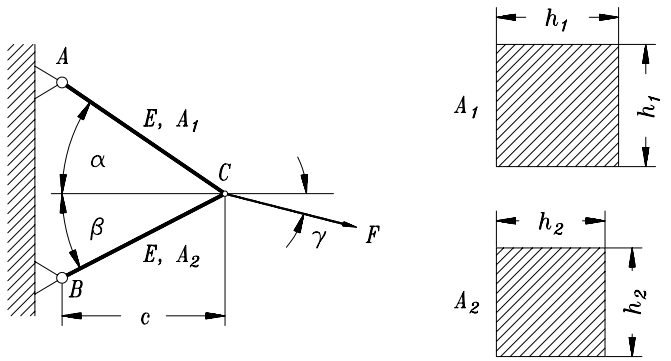

Abbildung 1.1: Belasteter Stabzweischlag

Aufgrund der Belastung durch die Kraft F verformt sich der Stabzweischlag so, daß sich die Spitze des Dreiecks gegenüber dem unbelasteten Zustand um den Vektor $(u, w)^T$ verschiebt, siehe Abbildung 1.2. Unter der Annahme, daß die Stablängenänderungen infolge der Last klein im Verhältnis zu den Ausgangslängen sind, seien nun die Querschnittsmaße h_1 und h_2 der Stäbe so zu bestimmen, daß sich für gegebene F, α, β, γ, c und E genau eine vorgeschriebene Verschiebung $(u, w)^T$ einstellt, d.h. gesucht ist

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

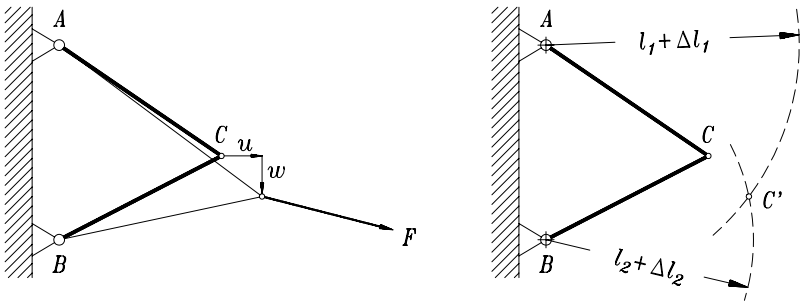

Abbildung 1.2: Elastisch verformter Stabzweischlag

Lösung: Aus den statischen Gleichgewichtsbedingungen am Punkt C folgt nach Abbildung 1.3

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

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

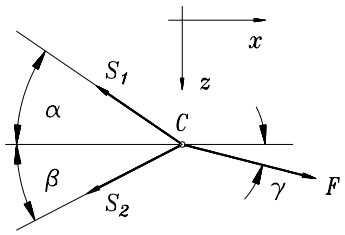

Abbildung 1.3: Kräftegleichgewicht am Punkt C

Die Materialgleichungen für die Stablängenänderungen lauten

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

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

wobei für die Stablängen l_1 und l_2 sowie für die Querschnittsflächen A_1 und A_2 gilt

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

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

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

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

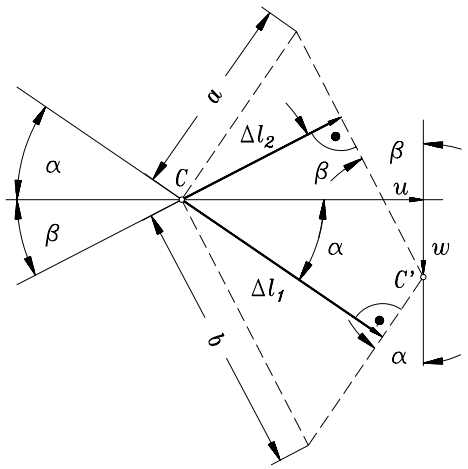

Abbildung 1.4: Verschiebungsplan

In dem in Abbildung 1.4 gezeichneten Verschiebungsplan sind wegen der kleinen Geometrieänderungen die Kreisbögen, auf denen sich die Stabenden bewegen können, durch die zur ursprünglichen Stabrichtung senkrechten Tangenten ersetzt worden. Für die Seitenlängen des gestrichelt eingezeichneten Parallelogramms gilt

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

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

Daraus ergeben sich die Verschiebungen u und v zu

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

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

Die Aufgabe besteht nun darin, das symbolische Gleichungssystem aus den Gleichungen (1.2) bis (1.13) nach den Unbekannten h_1 und h_2 explizit aufzulösen, wobei alle nicht benötigten Variablen, d.h. S_1, S_2, A_1, A_2, Δl_1, Δl_2, a und b, eliminiert werden sollen.

□

Beispiel 1.2

Das zweite Beispiel stammt aus der Elektrotechnik und betrifft die Dimensionierung analoger elektronischer Schaltungen. Für den in Abbildung 1.5 gezeichneten zweistufigen Transistorverstärker [NÜH_89] sollen symbolische Dimensionierungsformeln bestimmt werden, die die Werte der sieben Widerstände $R_1 \ldots R_7$ als Funktion der Betriebsspannung V_{CC}, der Kleinsignalverstärkung A, des Eingangswiderstandes Z_i und des Ausgangswiderstandes Z_o der Schaltung für numerisch festgelegte Arbeitspunkte der Transistoren beschreiben:

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

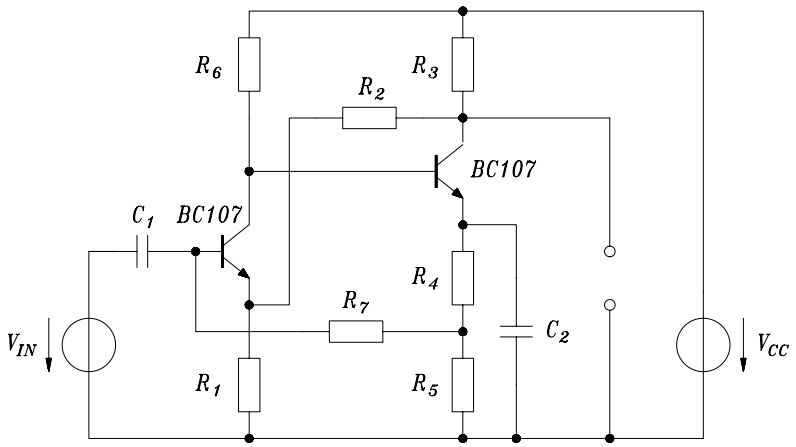

Abbildung 1.5: Zweistufiger Transistorverstärker

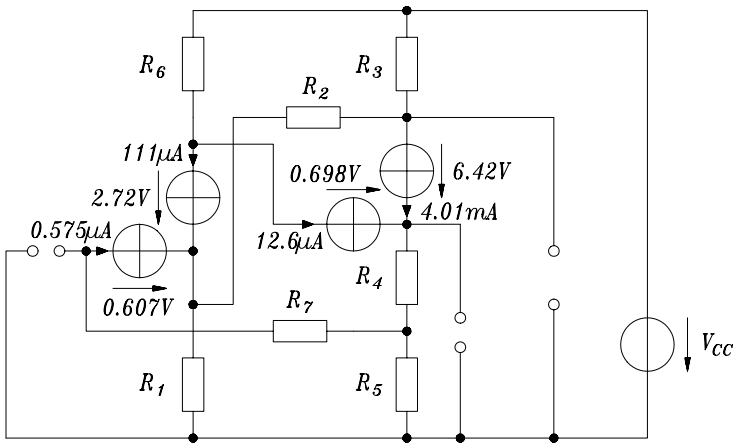

Abbildung 1.6: Arbeitspunktersatzschaltbild des Verstärkers mit festgelegten Transistor-Arbeitspunkten

Zu diesem Zweck werden mit Hilfe symbolischer Netzwerkanalyseverfahren [SOM_93a] sowie symbolischer Approximationsmethoden [HEN_93] zunächst die (stark vereinfachten) Übertragungsfunktionen A, Z_i und Z_o im Durchlaßbereich des Verstärkers als Funktionen der Elementeparameter und der Arbeitspunktwerte berechnet.

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

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

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

Die Werte der Widerstände $R_1 \ldots R_7$ bestimmen nicht nur die Kleinsignalcharakteristiken, sondern beeinflussen zudem die Arbeitspunkteinstellung der Schaltung. Daher sind die Widerstände

so zu dimensionieren, daß die Kleinsignal- und Arbeitspunktvorgaben *gleichzeitig* erfüllt werden. Den Gleichungen (1.15) – (1.17) wird aus diesem Grund das umfangreiche Sparse-Tableau-Gleichungssystem (1.18) – (1.51) hinzugefügt, das aus dem in Abbildung 1.6 dargestellten Arbeitspunktersatzschaltbild des Verstärkers hervorgeht.

$$
\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}
$$

Zur Bestimmung der gesuchten Dimensionierungsvorschriften in der Form (1.14) sind aus dem Gleichungssystem (1.15) – (1.51) alle Zweigspannungen und -ströme $V_{??}$ und $I_{??}$ zu eliminieren

und die verbleibenden Gleichungen nach den Widerständen $R_1 \ldots R_7$ aufzulösen.

□

1.2 Grenzen der Anwendung konventioneller Gleichungslöser

Wird der Versuch unternommen, zur Lösung der Gleichungssysteme aus den beiden Beispielen die Standardroutinen bekannter kommerzieller Computeralgebrasysteme wie Macsyma [MAC_94] oder Mathematica [WOL_91] einzusetzen, so ergeben sich meist, wie hier im Fall der Anwendung von Macsyma im Beispiel 1.1, Resultate der folgenden Art:

```
(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) []
```

Dieses Verhalten der Computeralgebraprogramme ist damit zu erklären, daß die zu lösenden Gleichungssysteme bezüglich der interessierenden Variablen h_1 und h_2 als überbestimmt angesehen werden, weil den Gleichungslösern keine zusätzliche Information über (nach Möglichkeit) zu *eliminierende* Variablen (S_1, S_2, A_1, A_2, Δl_1, Δl_2, a, b) bzw. über *keinesfalls* zu eliminierende Variablen, d.h. die Parameter des Systems (F, α, β, γ, c, E, u, w), übergeben werden kann.

Ein möglicher Ausweg besteht darin, die Gleichungslöser dazu zu veranlassen, auch Lösungen für die nicht interessierenden Variablen zu bestimmen. Dies ist aber nicht immer durchführbar und in der Regel sehr ineffizient, denn es wird in manchen Fällen viel Rechenzeit benötigt zur Bestimmung von Variablen, die keinen Einfluß auf die gesuchten Unbekannten haben. Es wird sogar überhaupt keine Lösung gefunden, wenn es für nur eine einzige nicht interessierende Variable keine analytische Lösung gibt. Letzteres gilt beispielsweise für das folgende Gleichungssystem, wenn nur die Lösungen für x und y gesucht sind:

$$x + y = 1 \tag{1.52}$$

$$2x - y = 5 \tag{1.53}$$

$$yz + \sin z = 1. \tag{1.54}$$

Aus den beiden linearen Gleichungen (1.52) und (1.53) lassen sich unmittelbar die Lösungen $x = 2$ und $y = -1$ bestimmen, die nicht im Widerspruch mit der verbleibenden nichtlinearen Gleichung (1.54) stehen. Macsyma erkennt diesen Umstand aber nicht und gibt lediglich Fehlermeldungen zurück — im ersten Versuch (`COM3`) wegen der scheinbaren Überbestimmtheit des Systems, im zweiten Versuch (`COM4`) aufgrund der analytisch nicht lösbaren dritten Gleichung:

```
(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 Anforderungen an einen universellen symbolischen Gleichungslöser

Für die symbolische Lösung des Gleichungsystems ist es erforderlich, daß die `Solve`-Funktion in keinem der beiden obigen Fälle vorzeitig abbricht. Im ersten Fall sollten nach einer Konsistenzprüfung mit Gleichung (1.54) die Lösungen für x und y ausgegeben werden. In letzterem Fall ist zu wünschen, daß neben den analytisch berechneten Lösungen zusätzlich die verbleibenden Gleichungen, für die keine solchen Lösungen gefunden werden konnten, in impliziter Form zurückgegeben werden, damit diese bei Bedarf mit numerischen Verfahren gelöst werden können. Eine adäquate Antwort auf das Kommando `COM4` wäre in diesem Sinne z.B. eine Ausgabe der Form

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

Die von Macsyma zur Verfügung gestellten Funktionen zur Lösung von Gleichungssystemen sind ohne Zugriff auf den Systemkern nicht in der Weise modifizierbar, daß sie das oben geforderte Verhalten zeigen. Das Ziel der vorliegenden Arbeit ist daher die Konzipierung und Implementierung eines auf den Macsyma-Standardroutinen basierenden universellen symbolischen Gleichungslösers, der in der Lage ist, Gleichungssysteme der in den Beispielen angeführten Art nach einer beliebigen Teilmenge aller Variablen aufzulösen oder zumindest durch Elimination möglichst vieler nicht benötigter Variablen eine weitgehende symbolische Vorverarbeitung der Gleichungen vorzunehmen, so daß numerische Optimierverfahren nur noch auf einem kleinen, analytisch nicht mehr lösbaren nichtlinearen Kern des Systems angewendet werden müssen.

Neben dieser generellen Zielsetzung lassen sich detaillierte Anforderungen an das zu entwickelnde Programm aus einigen allgemein bekannten Tatsachen und einer Reihe von Beobachtungen ableiten, die an den Beispielen 1.1 und 1.2 sowie dem Gleichungssystem (1.52) – (1.54) zu machen sind:

1. Meist ist nur die Lösung für einige wenige Variablen gefragt, alle anderen Unbekannten sind zu eliminieren.

2. Die zu lösenden Gleichungssysteme können einfach oder mehrfach parametrisiert sein.

3. Es ist keineswegs sichergestellt, daß die Parameter voneinander unabhängig sind, d.h. es ist möglich, daß ein Gleichungssystem nur dann eine Lösung hat, wenn gewisse arithmetische Zwangsbedingungen zwischen einigen Parametern eingehalten werden.

4. Die Gleichungssysteme enthalten oft einfache, direkte Zuweisungen der Form $x_i = \text{const.}$, siehe Gleichungen (1.6) und (1.7) oder (1.42) – (1.51).

5. Ein erheblicher Anteil der zu lösenden Gleichungen ist linear bezüglich einer nicht unmittelbar ersichtlichen Teilmenge aller Variablen, siehe die in bezug auf alle $V_{??}$ und $I_{??}$ linearen Gleichungen (1.18) – (1.32).

6. Die Systeme können Freiheitsgrade enthalten.

7. Es existieren keine allgemeingültigen Lösungsverfahren für nichtlineare Gleichungen und Gleichungssysteme.

8. Nichtlineare Gleichungen können unlösbar (widersprüchlich) sein, eindeutige Lösungen haben oder Mehrfachlösungen mit endlicher oder unendlicher Lösungsvielfalt besitzen.

9. Nicht immer sind im Fall von Mehrfachlösungen alle Elemente der Lösungsmenge mit den übrigen Gleichungen konsistent.

10. Für viele nichtlineare Gleichungen existieren gar keine analytischen Lösungen, siehe Gleichung (1.54).

Aus diesen Feststellungen ergeben sich folgende, den entsprechenden Punkten zugeordnete Forderungen:

1. Das Programm soll Gleichungssysteme nur so weit lösen, wie es zur Bestimmung einzelner Variablen unbedingt erforderlich ist. Dabei sind errechnete Lösungen auf mögliche Widersprüche mit den übrigen Gleichungen zu überprüfen.

2. Es müssen gesuchte Variablen und Parameter getrennt voneinander angegeben und verarbeitet werden können. Parameter dürfen im Gegensatz zu nicht interessierenden Variablen keinesfalls eliminiert werden.

3. Werden Abhängigkeiten zwischen Parametern erkannt, so muß auf Wunsch des Anwenders der Programmlauf unter Berücksichtigung und Speicherung der entsprechenden Zwangsbedingungen fortgeführt werden können.

4. Direkte Zuweisungen sollten unmittelbar zu Beginn des Programmablaufs gesucht und ausgeführt werden, um den Umfang des verbleibenden Gleichungssystems so weit wie möglich mit geringem Aufwand zu reduzieren.

5. Da es für lineare Gleichungen effiziente, geschlossene Lösungsverfahren gibt, ist es zweckmäßig, das Gleichungssystem wiederholt nach linearen Teilblöcken zu durchsuchen, diese zu lösen und die Resultate in den Rest der Gleichungen einzusetzen, bis keine linearen Gleichungsanteile mehr vorhanden sind.

6. Freiheitsgrade sollen automatisch in vom Programm gewählten Variablen ausgedrückt werden.

7. Die Lösung nichtlinearer Gleichungen muß mit Hilfe heuristischer Bewertungsstrategien gesteuert werden.

8. Im Fall von Mehrfachlösungen mit endlicher Vielfalt ist jeder einzelne Lösungspfad getrennt rekursiv zu verfolgen.

9. Mit den restlichen Gleichungen inkonsistente Teile von Mehrfachlösungen müssen erkannt und der zugehörige Lösungspfad verworfen werden.

10. Wie bereits zum Anfang des Abschnitts gefordert wurde, sollen analytisch nicht lösbare Gleichungen nicht zum Abbruch des Programms führen. Statt dessen soll das Gleichungssystem so weit wie möglich auf Dreiecksform gebracht und die nicht lösbaren restlichen Gleichungen zusammen mit den bis dahin ermittelten Teillösungen ausgegeben werden.

1.4 Extraktion und Lösung linearer Gleichungen

Zwar sind bei technischen Aufgabenstellungen die zu lösenden Gleichungen zumeist nichtlinear, doch enthalten die betreffenden Systeme häufig große lineare Teilblöcke. Da lineare Gleichungssysteme sehr effizient mit Hilfe der Gauß-Elimination simultan gelöst werden können, empfiehlt es sich, vor der Lösung der nichtlinearen Gleichungen zunächst den linearen Anteil des Systems separat zu verarbeiten. Selbst wenn eine vollständige analytische Auflösung des gesamten nichtlinearen Systems nach allen gesuchten Variablen nicht erreicht werden kann, ist es dennoch sinnvoll, durch Elimination der linearen Variablen und Gleichungen das System auf einen nur noch kleinen, nicht mehr analytisch lösbaren Kern zu reduzieren, dessen numerische Lösung wesentlich weniger aufwendig ist, als eine auf dem vollständigen System durchgeführte Optimierung.

Die unter Punkt 5 geforderte iterative Lösung linearer Teilsysteme des gesamten Gleichungssystems ist insofern keine triviale Aufgabe, als daß weder die betreffenden Gleichungen noch die Teilmenge der Variablen, bezüglich der diese Gleichungen linear sind, von vornherein bekannt sind. Deshalb muß eine Suchstrategie gefunden werden, die (aus Effizienzgründen möglichst große) lineare Gleichungs- und Variablenblöcke aus einem beliebigen nichtlinearen Gleichungssystem extrahiert.

1.4.1 Intuitive Vorgehensweisen zur Suche linearer Gleichungen

Zur Verdeutlichung der Aufgabenstellung wird das folgende nichtlineare Gleichungssystem herangezogen, das nach den Variablen x, y und z zu lösen sei.

$$x + 2y - z = 6 \tag{1.55}$$
$$2x + yz - z^2 = -1 \tag{1.56}$$
$$3x - y + 2z^2 = 3 \tag{1.57}$$

Auf den ersten Blick ist nur die Gleichung (1.55) linear, und zwar in bezug auf alle drei Variablen. Bei Verwendung eines einfachen Suchalgorithmus, der ausschließlich solche vollständig linearen Gleichungen findet, kann in diesem Fall maximal eine Variable nach entsprechender Auflösung von (1.55), z.B. nach x, aus den beiden restlichen Gleichungen eliminiert werden.

Eine genauere Betrachtung der Gleichungen offenbart aber eine bessere Alternative. Nach der Entfernung von Gleichung (1.56) und dem Verschieben der von z abhängigen Terme auf die rechten Seiten der Gleichungen (1.55) und (1.57) entstehen *zwei* in den Variablen x und y lineare Gleichungen:

$$x + 2y = 6 + z, \tag{1.58}$$
$$3x - y = 3 - 2z^2. \tag{1.59}$$

Deren simultane Inversion führt zu den in z parametrisierten Lösungen

$$x = -\frac{1}{7}\left(4z^2 - z - 12\right), \tag{1.60}$$

$$y = \frac{1}{7}\left(2z^2 + 3z + 15\right), \tag{1.61}$$

nach deren Einsetzen in (1.56) nur noch *eine* zu lösende nichtlineare Gleichung verbleibt:

$$2z^3 - 12z^2 + 17z + 31 = 0. \tag{1.62}$$

Angesichts der Tatsache, daß bei der zweiten Variante in nur einer Iteration zwei Unbekannte gleichzeitig bestimmt werden konnten, ist letztere Vorgehensweise der zuerst angewendeten Suche nach vollständig linearen Gleichungen trotz des zusätzlichen Umformungsaufwandes vorzuziehen. Dies gilt insbesondere dann, wenn ein Gleichungssystem überhaupt keine Gleichungen enthält, in denen alle beteiligten Variablen in rein linearer Form auftreten. Das zur Extraktion linearer Teilsysteme verwendete Verfahren sollte daher beide demonstrierten Operationen zur Beseitigung nichtlinearer Gleichungsanteile miteinander verbinden:

1. Entfernung einzelner nichtlinearer Gleichungen
2. Verschieben von in nichtlinearen Termen auftretenden Variablen auf die rechten Seiten der Gleichungen

Durch eine ausgewogene Kombination der zwei Operationen kann erreicht werden, daß die resultierenden linearen Teilsysteme maximale Größe haben und — oft zumindest annähernd — quadratisch sind.

1.4.2 Ein heuristischer Algorithmus zur Suche linearer Gleichungen

Für eine Rechnerimplementation muß eine solche vom Menschen intuitiv durchgeführte Suche nach linearen Gleichungsblöcken systematisiert und algorithmisch formuliert werden. Da der Begriff *lineares Teilgleichungssystem*, wie anhand der unterschiedlichen Lösungsmöglichkeiten ersichtlich ist, das gewünschte Resultat nicht in eindeutiger Weise definiert, wurde eine heuristische Strategie zur Nachahmung der intuitiven Vorgehensweise entwickelt. Diese Strategie wird zum Vergleich der Ergebnisse an dem schon betrachteten System (1.55) – (1.57) demonstriert.

Für das Gleichungssystem wird zunächst eine Tabelle aufgestellt, deren Zeilen den Gleichungen und deren Spalten den Variablen zugeordnet sind. Der Eintrag an der Position (i, j) der Tabelle enthält für die Gleichung i den Koeffizienten[1] bezüglich des linearen Gliedes, d.h. der ersten Potenz, der Variablen x_j. Ist wie für z in Gleichung (1.57) kein Term in erster Potenz vorhanden oder tritt dieser als Argument nicht polynomialer Funktionen auf (z.B. $\sin x$ oder $\sqrt{x}$), so wird die zugehörige Position mit einem Kreuz ($\times$) markiert.

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

[1] Macsyma stellt zur Bestimmung der Koeffizienten rationaler Ausdrücke den Befehl `RATCOEFF` zur Verfügung.

Dieser Tabelle wird eine zweite, gleich große Bewertungsmatrix zugeordnet, deren Einträge gleich Null sind, wenn der korrespondierende Eintrag in der Koeffiziententabelle eine Konstante ist, und gleich Eins, wenn der betreffende lineare Koeffizient gesuchte Variablen enthält oder nicht existiert ($\times$). Außerdem werden an den Rändern der Matrix die Zeilen- und Spaltensummen notiert, sowie unter den Summenzeichen rechts oben und links unten respektive die Zeilensumme der Spaltensummen ($\sum S$) und die Spaltensumme der Zeilensummen ($\sum Z$).

		x	y	z	$\sum S$
		0	1	2	3
1)	0	0	0	0	
2)	2	0	1	1	
3)	1	0	0	1	
$\sum Z$	3				

(1.63)

Offensichtlich entsprechen die Eins-Elemente in der Bewertungsmatrix genau den unerwünschten nichtlinearen Anteilen im Gleichungssystem. Ein lineares Teilgleichungssystem und die zugehörigen Variablen sind dann gefunden, wenn mit einer Folge der am Ende von Abschnitt 1.4.1 aufgeführten Operationen alle Einsen beseitigt wurden. Übertragen auf Manipulationen der Bewertungsmatrix entspricht dabei die 1. Operation dem Streichen der zu einer bestimmten Gleichung gehörigen Zeile. Die 2. Operation ist äquivalent zur Entfernung der einer Variablen zugeordneten Spalte. Das lineare Teilsystem besteht anschließend aus denjenigen Gleichungen und Variablen, deren Zeilen und Spalten nicht aus der Matrix entfernt wurden.

Die Reduktion der Bewertungsmatrix (1.63) zu einer Nullmatrix kann auf genau drei verschiedene Weisen erfolgen:

1. Streichen der Zeilen 2) und 3),
2. Streichen der Spalten y und z,
3. Streichen der Zeile 2) und der Spalte z.

Der Forderung nach maximaler Größe und möglichst quadratischer Form der linearen Gleichungsblöcke ist direkt auf die zu erzielenden Eigenschaften der Nullmatrix übertragbar. In diesem Sinne ist die letztgenannte der drei Optionen optimal, denn sie liefert das schon im vorigen Abschnitt als bestmögliche Lösung erkannte System (1.58) – (1.59). Die beiden anderen Möglichkeiten führen dagegen zu der unterbestimmten Gleichung (1.55) bzw. zu einem überbestimmten 3×1-System in x.

Die Suche nach einer optimalen Folge von Zeilen- und Spaltenstreichungen ist ein komplexes kombinatorisches Problem. Um den damit verbundenen Aufwand zu vermeiden, wird zur Bestimmung der im jeweiligen Schritt zu entfernenden Zeile oder Spalte ein heuristisches, lokales Entscheidungskriterium, d.h. eine *Greedy*-Strategie [FOU_92], verwendet: Es wird die Zeile oder Spalte gestrichen, die die meisten Einsen enthält, also diejenige mit der größten Zeilen- bzw. Spaltensumme. Dieses Kriterium liefert jedoch noch keine eindeutige Aussage, wenn

- zwei oder mehr Zeilen die gleiche (größte) Zeilensumme haben,
- zwei oder mehr Spalten die gleiche (größte) Spaltensumme haben,
- oder wenn die Summen der höchstbewerteten Zeile und der höchstbewerteten Spalte identisch sind.

In den ersten beiden Fällen kann aus den betreffenden Zeilen oder Spalten eine beliebige ausgewählt werden, in der Regel ist dies der Einfachheit halber die jeweils zuerst gefundene mit maximaler Bewertung.

Der dritte Fall tritt im vorliegenden Beispiel auf. In der Bewertungsmatrix (1.63) besitzen sowohl die Zeile 3) als auch die Spalte z die maximale Summe 2. Zu Beginn werde aus beiden Möglichkeiten willkürlich die Streichung der Zeile ausgewählt, so daß sich im nächsten Schritt folgende Bewertungsmatrix ergibt:

		x	y	z	$\sum S$
		0	0	1	1
1)	0	0	0	0	
3)	1	0	0	1	
$\sum Z$	1				

(1.64)

Wiederum ist nun die maximale Zeilensumme gleich der maximalen Spaltensumme. Auch hier könnte die Entscheidung nach dem Zufallsprinzip erfolgen, doch würde damit die Forderung nach möglichst quadratischer Form der linearen Teilsysteme nicht genügend berücksichtigt. Daher empfiehlt sich entweder, Zeilen und Spalten mit gleicher Bewertung *wechselweise* zu streichen, oder die Wahl zugunsten der Möglichkeit zu treffen, die das Dimensionsverhältnis n/m der $n \times m$-Bewertungsmatrix bei $n \neq m$ näher an Eins bringt. Nach beiden Kriterien erweist sich das Streichen der Spalte z als günstiger als die Entfernung der Zeile 3).

		x	y	$\sum S$
		0	0	0
1)	0	0	0	
3)	0	0	0	
$\sum Z$	0			

(1.65)

Die Streichung von Spalte z bewirkt die Beseitigung der letzten Eins in der Bewertungsmatrix. Dies äußert sich im Verschwinden von $\sum S$ und $\sum Z$, durch welches das Ende des Algorithmus markiert wird. Aus der Ergebnismatrix (1.65) ist nun abzulesen, daß die Gleichungen (1.55) und (1.57) des Beispielgleichungssystems bezüglich der Variablen x und y linear sind. Die abschließend notwendigen Umformungen der linearen Gleichungen zur Erstellung der Simultanform (1.58) – (1.59) stellen für ein Computeralgebrasystem kein Problem dar, sie werden von Macsyma automatisch von dem Befehl `LINSOLVE` zur simultanen Lösung linearer Gleichungen ausgeführt.

1.4.3 Lösung der linearen Gleichungen

Sind die unter Verwendung des beschriebenen Algorithmus extrahierten linearen Teilgleichungssysteme eindeutig lösbar oder unterbestimmt, so ist ihre Weiterverarbeitung unproblematisch. Im Fall überbestimmter Systeme treten dagegen zwangsweise Inkonsistenzen auf, die einer eingehenderen Behandlung bedürfen. Z.B. sei aus einem größeren Gleichungssystem in den Variablen x, y, z und w das überbestimmte, in x und y lineare Teilsystem (1.66) – (1.68) entnommen worden.

$$x - y = z^2 + z \tag{1.66}$$

$$x + y = w^2 + 1 \tag{1.67}$$

$$x - y = z + w \tag{1.68}$$

Nach der Vorwärtselimination ensteht das folgende Gleichungssystem, das im Sinne der linearen Algebra inkonsistent ist und somit keine Lösung hat:

$$\begin{aligned} x - y &= z^2 + z && (1.69)\\ 2y &= w^2 - z^2 - z + 1 && (1.70)\\ 0 &= w - z^2 && (1.71) \end{aligned}$$

Im betrachteten Fall sind jedoch z und w Variablen des zu lösenden Gleichungssystems. Das obige lineare Teilsystem hat genau dann Lösungen, wenn diese beiden Variablen die Gleichung (1.71) erfüllen. Diese Bedingung ist daher lediglich als eine weitere Gleichung des Restsystems anzusehen, aus der x und y eliminiert wurden.

Als Folgerung ergibt sich, daß beim Auftreten scheinbarer Inkonsistenzen im Anschluß an den Eliminationsvorgang generell die rechten Seiten der Konsistenzbedingungen daraufhin überprüft werden müssen, ob sie gesuchte Variablen des gesamten Systems enthalten. Ist dies der Fall, so werden die betreffenden Bedingungen nach der Lösung der linearen Gleichungen dem Ausgangsgleichungssystem wieder hinzugefügt. Trifft dies nicht zu, d.h. treten nicht erfüllbare Zwangsbedingungen zwischen numerischen Werten oder Parametern auf, dann hat das Gleichungssystem in der Tat keine Lösung, und der Lösungsvorgang muß abgebrochen werden.

1.5 Bewertungsstrategien zur Lösung nichtlinearer Gleichungen

Abgesehen von wenigen Sonderfällen existieren keine geschlossenen Lösungsverfahren für allgemeine nichtlineare Gleichungssysteme. Dies schließt jedoch nicht aus, daß sich für viele nichtlineare Systeme analytische Lösungen oder zumindest Teillösungen berechnen lassen, nur ist in der Regel deren Bestimmung nicht auf so effiziente Weise wie durch Gauß-Elimination im Fall linearer Gleichungen möglich.

1.5.1 Einsetzverfahren für nichtlineare Gleichungssysteme

Ein elementares Lösungsverfahren, das sich auf beliebige Gleichungssysteme anwenden läßt, ist das bekannte Einsetzverfahren:

1. Wähle eine (möglichst einfache) Gleichung aus dem System aus und löse sie nach einer Variablen x_j auf. Brich ab, wenn alle Gleichungen gelöst sind, oder sich keine weitere Gleichung analytisch lösen läßt.
2. Setze das Ergebnis in die restlichen Gleichungen ein, um x_j aus dem System zu eliminieren.
3. Prüfe das um eine Gleichung und eine Variable reduzierte System auf Konsistenz und fahre mit Schritt 1 fort.

Zur Demonstration des Verfahrens werde das nichtlineare Gleichungssystem (1.72) – (1.76) betrachtet, das nach den Variablen a, b, c und d zu lösen sei.

$$\begin{aligned} ab + 2c &= 0 && (1.72)\\ c^2 + d - 4 &= 0 && (1.73) \end{aligned}$$

$$\sqrt{b+d}-2 = 0 \tag{1.74}$$

$$\tan\left(\frac{\pi}{2a}\right)-1 = 0 \tag{1.75}$$

$$b\ \operatorname{Arcosh} c\ - i\pi = 0 \tag{1.76}$$

Schon unmittelbar zu Beginn der Anwendung des Einsetzverfahrens stellt sich die Frage, welche konkreten Eigenschaften eine "möglichst einfache" Gleichung auszeichnen. Aus Erfahrungswissen lassen sich u.a. folgende Beurteilungskriterien ableiten, die nicht notwendigerweise gleichzeitig zutreffen müssen und je nach Anwendung unterschiedlich gewichtet werden können:

Einfache Gleichungen

1. enthalten nur wenige der gesuchten Variablen,
2. enthalten eine gesuchte Variable an genau einer Position, so daß sich die Unbekannte relativ leicht isolieren läßt,
3. weisen bezüglich einer oder mehrerer Variablen nur geringe Tiefen der Operationenhierarchie auf, d.h. die *Formelkomplexität* ist gering,
4. enthalten keine transzendenten oder andere, schwer zu invertierende Funktionen.

Anhand dieser Kriterien soll nun die einfachste Gleichung des Beispielsystems bestimmt werden. Hinsichtlich der ersten beiden Punkte könnte dies die Gleichung (1.75) sein, denn sie enthält nur die Variable a, und diese tritt an genau einer Position auf. Dagegen fällt die Beurteilung aufgrund der Kriterien 3 und 4 nicht sehr günstig aus. In bezug auf die Punkte 2, 3 und 4 erscheint die Lösung der Gleichung (1.73) nach der Variablen d als beste Wahl, denn alle anderen Gleichungen enthalten entweder mehr Variablen oder schwieriger auflösbare Funktionen.

Als maßgebliches Kriterium möge hier zunächst die Kombination der Punkte 1 und 2 gelten, so daß mit der Lösung der Gleichung (1.75) nach der Variablen a begonnen wird. Wird nur der Hauptwert der arctan-Funktion berücksichtigt, so folgt

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

Eingesetzt in die restlichen vier Gleichungen ergibt sich das System

$$2b + 2c = 0, \tag{1.78}$$

$$c^2 + d - 4 = 0, \tag{1.79}$$

$$\sqrt{b+d}\ - 2 = 0, \tag{1.80}$$

$$b\ \operatorname{Arcosh} c\ - i\pi = 0. \tag{1.81}$$

Da keine Inkonsistenzen entstanden sind, wird mit der Auswahl der nächsten einfachsten Gleichung fortgefahren. Die Bewertungskriterien sprechen nun eindeutig[2] für die Lösung der Gleichung (1.79) nach d:

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

[2]Ein Mensch würde vermutlich die erste Gleichung für einfacher zu lösen halten, aber strenggenommen ist das vorher notwendige Teilen der Gleichung durch den Faktor 2 ein zusätzlich zu berücksichtigender Aufwand.

Daraus folgt

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

Bezüglich aller Kriterien ist jetzt die Gleichung (1.83) der günstigste Kandidat, so daß mit der Lösung

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

noch zwei Gleichungen mit der Unbekannten c verbleiben.

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

Gleichung (1.88) ist analytisch nicht lösbar, daher muß unabhängig von der Bewertung Gleichung (1.87) die noch fehlende Lösung für c liefern. In diesem Fall ergibt sich erstmals eine Mehrfachlösung:

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

An ihr zeigt sich die Bedeutung der bislang nicht relevanten Konsistenzprüfung. Von den beiden Lösungen erfüllt nur die zweite, $c = -1$, die Gleichung (1.88). Die andere Lösung führt zu der widersprüchlichen Aussage

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

und muß deshalb verworfen werden. Nach der Rücksubstitution lauten die konsistenten Lösungen somit

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

1.5.2 Heuristische Verfahren zur Komplexitätsbewertung algebraischer Ausdrücke

Sollen die Beurteilung algebraischer Gleichungen hinsichtlich ihrer "Einfachheit" und die auf ihr basierende Lösung eines nichtlinearen Gleichungssystems automatisch von einem Computeralgebrasystem vorgenommen werden, so müssen die im Abschnitt 1.5.1 sprachlich formulierten Kriterien in Algorithmen abgebildet werden, die numerische Komplexitätsbewertungen zur Steuerung des Lösungsprozesses liefern. Die Höhe einer Bewertungszahl b soll ein Maß dafür sein, wie schwierig es ist, eine Gleichung i nach einer Variablen x_j analytisch aufzulösen. Je größer b ist, um so komplexer erscheint[3] die Aufgabe.

Wird die Bewertung für jede Gleichung hinsichtlich jeder Variablen vorgenommen, so kann mittels einer Sortierung anhand der Maßzahlen eine Lösungsreihenfolge für die Gleichungen generiert werden. Die Lösungsreihenfolge ist darstellbar durch eine geordnete Liste der Form

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

wobei für die Bewertungszahlen gilt $b_k \leq b_l$ für $k < l$. Für den Gleichungslöser impliziert die Liste die folgenden Anweisungen: Versuche zunächst, die Gleichung i_1 nach der Variablen x_{j_1}

[3] Es wird hier bewußt die Formulierung *erscheint* gewählt, denn die Bewertung wird anhand heuristischer Kriterien vorgenommen, die keine optimalen Entscheidungen garantieren können.

aufzulösen, da dies am einfachsten erscheint. Gelingt dies nicht, so versuche stattdessen die Lösung von Gleichung i_2 nach x_{j_2}, usw. Ist der Lösungsversuch dagegen erfolgreich, so setze die Lösung in die verbleibenden Gleichungen ein und beginne von vorne mit der Aufstellung einer neuen Lösungsreihenfolge.

Die Umsetzung des ersten Kriteriums in einen Algorithmus stellt keine ernsthafte Herausforderung dar, denn die Anzahl der in einer Gleichung enthaltenen Variablen ist bereits ein numerischer Wert. Die programmtechnische Realisierung ist in einem Computeralgebrasystem ebensowenig ein Problem. In Macsyma genügt der kurze Befehl

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

zur Feststellung der Zahl der Unbekannten in der i-ten Gleichung. Diese Zahl ist aber lediglich als Sekundärkriterium in Verbindung mit anderen Bewertungen von Nutzen, denn es wird nur eine Rangfolge der Gleichungen, nicht aber von Gleichungs/Variablen-Paaren, geliefert.

Aus später ersichtlichen Gründen soll hier zunächst die Betrachtung des Kriteriums 2 zurückgestellt und mit den Punkten 3 und 4 fortgefahren werden. Diese beiden Kriterien wurden zwar getrennt aufgelistet, aber sie lassen sich sehr leicht ein einem einzigen Verfahren miteinander verbinden. Der in der im folgenden Kapitel beschriebenen Implementation des symbolischen Gleichungslösers verwendete heuristische Bewertungsalgorithmus nutzt zur Komplexitätsberechnung die interne Darstellung algebraischer Ausdrücke in Computeralgebrasystemen aus. Zusammengesetzte algebraische Funktionen werden in hierarchisch organisierten Listen von Operatoren und Operanden in Präfixnotation verwaltet, die sich unmittelbar in eine Baumstruktur abbilden lassen. Die Knoten eines solchen Baumes enthalten die Operatoren, die Operanden befinden sich in den Blättern. Beispielsweise zeigt die Abbildung 1.7 die Repräsentation der linken Seiten der Gleichungen (1.75) und (1.87) als Operationenbäume.

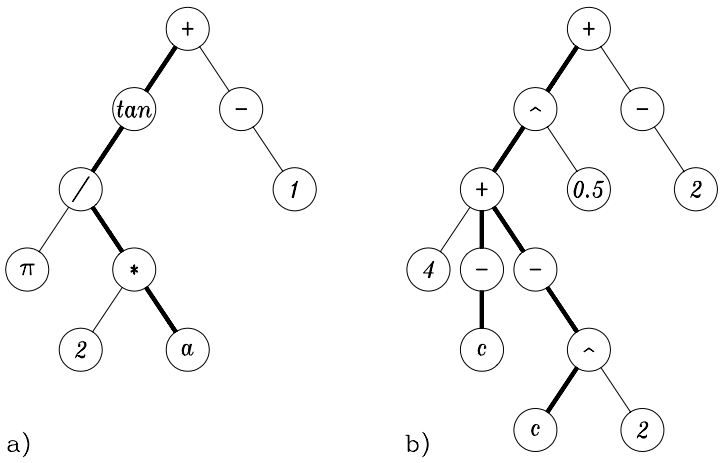

Abbildung 1.7: Baumdarstellung algebraischer Ausdrücke

Eine Bewertung der Formelkomplexität hinsichtlich der Tiefe der Operationenhierarchie, also des Grades der Verschachtelung eines Ausdrucks, ist nun an den Baumstrukturen ablesbar. Z.B. kann die Komplexität bestimmt werden, indem die Baumzweige gezählt werden, die von der Wurzel des Operationenbaumes an durchschritten werden müssen, um zu den Instanzen der

betrachteten Variablen zu gelangen. Im Fall des Ausdrucks in Abbildung 1.7a) ist der Komplexitätswert b bezüglich der Variablen a gleich der Länge des fett eingezeichneten Pfades, d.h. $b = 4$. Um von der Baumwurzel in Abbildung 1.7b) alle Instanzen der Variablen c zu erreichen, müssen insgesamt sieben Zweige durchquert werden, demnach ist die Komplexität $b = 7$.

Diese Vorgehensweise zur Berechnung der Komplexität ist mühelos so erweiterbar, daß auch das Kriterium 4, die Bewertung eines Ausdrucks hinsichtlich der in ihm enthaltenen Operatoren, gleichzeitig berücksichtigt wird. Statt der einfachen Zählung der Baumzweige muß nur zusätzlich eine Gewichtung erfolgen, indem jedem Operator ein typischer "Schwierigkeitsfaktor" zugewiesen wird, mit dem die Bewertung seiner Operanden zu multiplizieren ist. Die Höhe des Schwierigkeitsfaktors soll widerspiegeln, wie aufwendig für das Computeralgebrasystem die Bildung der Umkehrfunktion, also die Auflösung der Funktion nach den Operanden, ist [TRI_91].

Zu Beginn erhält jedes Blatt des Baumes die Gewichtung 1, wenn es die Variable enthält, bezüglich der die Bewertung berechnet werden soll. Andernfalls werden die Blätter mit der Gewichtung 0 versehen. Bei der Bewertung der Operatoren dient der Operator "+" als Referenz mit der Gewichtung 1, da er am einfachsten zu invertieren ist. Im Vergleich dazu wird die Umkehrung der Operatoren "$*$" und "tan" willkürlich als vier- bzw. zehnmal so schwierig angesehen, entsprechend werden daher die Gewichtungen angesetzt. Die folgende Tabelle zeigt eine Auswahl einiger Operatoren und der ihnen zugewiesenen Gewichtungen.

Operator	$+$	$-$	$*$	$/$	$\wedge$	tan	Arcosh
Gewichtung	1	1	4	4	10	10	12

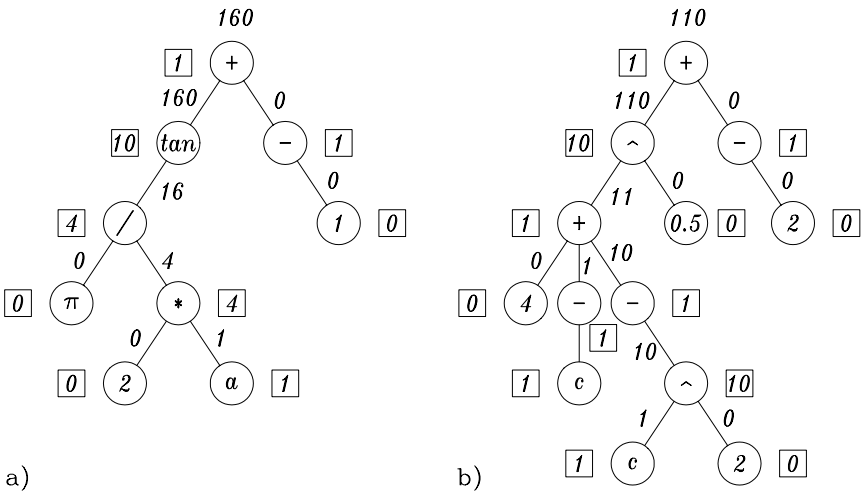

Abbildung 1.8: Bewertung der Operatoren

Die Berechnung des Komplexitätswertes erfolgt *bottom-up* durch wiederholte Addition der Zweiggewichte an den Operatorknoten, Multiplikation der Summe mit dem Operatorgewicht und Transfer des resultierenden Wertes auf den übergeordneten Zweig, bis die Baumwurzel erreicht ist. Zur Demonstration des Vorgehens werde der Operationenbaum in Abbildung 1.8a) betrachtet. In den neben den Knoten eingezeichneten Quadraten sind die Operatorengewichte markiert, die Zahlen an den Zweigen bedeuten das Gesamtgewicht des jeweils untergeordneten Teilbaums. Da die Komplexität des Ausdrucks bezüglich der Variablen a bewertet werden soll,

erhält nur das Blatt mit dem Symbol a das Gewicht 1, alle anderen Blätter werden mit 0 bewertet. An dem Multiplikationsknoten überhalb der Variablen addieren sich die Zweiggewichte zur Summe $0 + 1 = 1$, die, gemäß der Bewertung des Operators "$*$", mit dem Faktor 4 multipliziert an den rechten Operandenzweig des "/"-Operators weitergegeben wird. Dort erfolgt ebenfalls eine Multiplikation mit 4, so daß das Zwischenergebnis nun gleich 16 ist. Anschließend kommen im Verlauf der Berechnung noch die Faktoren 10 und 1 für den "tan"- und den "+"-Operator hinzu. Das endgültige Resultat, $b = 160$, findet sich über der Wurzel des Baumes. Für den Operationenbaum in Abbildung 1.8b) ergeben entsprechende Berechnungen einen Komplexitätswert von $b = 110$ bezüglich der Variablen c.

An dieser Stelle kann nun das vorläufig zurückgestellte Kriterium 2 wieder aufgegriffen werden. Um diejenigen Gleichungen eines Systems zu ermitteln, die eine oder mehrere Variablen an jeweils genau einer Position enthalten, genügt es, in der oben beschriebenen Berechnungsvorschrift für die Formelkomplexität alle Operatorengewichte zu Eins zu setzen und alle Gleichungs/Variablen-Paare zu speichern, für die sich unter diesen Bedingungen die Komplexitätsbewertung $b = 1$ ergibt. Die Methode ist damit zu begründen, daß die Suche genau jenen Gleichungen gilt, in deren Baumrepräsentation bestimmte Variablensymbole in nur einem Blatt auftreten. Das bedeutet, es gibt in diesen Fällen nur jeweils einen einzigen Pfad von der Baumwurzel zu dem betreffenden Symbol. Bei einer Gewichtung von Eins für alle Operatoren entspricht die Komplexitätsbewertung gerade der Anzahl der Pfade zu den Instanzen einer Variablen.

Das Verfahren läßt sich leicht anhand von Beispielen verifizieren. Werden alle Operatorengewichte gleich Eins gesetzt, so ergeben sich für die Ausdrücke in Abbildung 1.8a) und b) Komplexitäten von $b = 1$ für die Variable a und $b = 2$ für die Variable c. Dies stimmt mit der Tatsache überein, daß die Variable a in genau einem Blatt des Baums enthalten ist, während das Symbol c an zwei Positionen zu finden ist.

1.5.3 Aufstellung der Lösungsreihenfolge

Durch unterschiedliches Kombinieren und Gewichten der oben diskutierten Bewertungsverfahren entstehen verschiedene Strategien zur Aufstellung von Lösungsreihenfolgen. Die im in dieser Arbeit entwickelten Programm implementierte Strategie namens `MinVarPathsFirst` stellt eine Kombination aus dem Kriterium 2 und der Komplexitätsbewertung mittels Operatorgewichtung dar. Oberste Priorität in der Lösungsreihenfolge erhalten die Gleichungen, die gesuchte Variablen an jeweils genau einer Position enthalten, wobei innerhalb dieser Gruppe nach geringstem Gewicht der Operatorenbäume sortiert wird. Alle übrigen Gleichungs/Variablen-Paare werden, ebenfalls nach geringstem Baumgewicht geordnet, an das Ende der Lösungsreihenfolge gestellt. Demnach würde im Fall der beiden Ausdrücke in Abbildung 1.8 zunächst versucht werden, die Gleichung mit der tan-Funktion nach der Variablen a aufzulösen, obwohl für sie eine höhere Bewertung berechnet wurde als für den nebenstehenden Ausdruck bezüglich der Variablen c.

Kapitel 2

Der Solver

2.1 Die Struktur des Solvers

Die im Abschnitt 1.3 aufgestellten Anforderungen, insbesondere die Punkte 4, 5 und 7, legen die in Abbildung 2.1 schematisch dargestellte Strukturierung des Gleichungslösers in fünf weitgehend unabhängige Hauptmodule nahe [TRI_91]. Aufbauend auf dieser Struktur und den im vorangegangenen Kapitel beschriebenen heuristischen Algorithmen wurde im Rahmen der vorliegenden Arbeit das Macsyma-Programmpaket `SOLVER` zur Funktionalitätserweiterung der Macsyma-Funktionen `SOLVE` und `LINSOLVE` entwickelt.

Die Aufgabe des Moduls *Solver Preprocessor* sind allgemeine, vorbereitende Arbeiten wie die Prüfung der Kommandosyntax und -semantik und der Aufbau internen Datenstrukturen, aber auch die Durchführung einer einleitenden Konsistenzprüfung. Mit ihr soll festgestellt werden, ob das Gleichungssystem unmittelbar als widersprüchlich zu erkennende Aussagen der Form Zahl = Zahl oder Zwangsbedingungen zwischen Parametern enthält.

Der *Immediate Assignment Solver* durchsucht das Gleichungssystem vor dem Aufruf des *Linear Solvers* nach direkten Zuweisungen der Form *var* = const. oder const. = *var* und führt diese sofort aus, damit der für die nachfolgenden Programmodule notwendige Rechenaufwand möglichst klein gehalten wird.

Der *Linear Solver* ist ein Prä- und Postprozessor für die Macsyma-Funktion `LINSOLVE` zur simultanen Lösung linearer Gleichungssysteme. Das Modul extrahiert lineare Teilgleichungssysteme nach dem in Abschnitt 1.4.2 beschriebenen heuristischen Algorithmus und löst die Gleichungen durch Aufruf von `LINSOLVE`. Die ermittelten Lösungen werden vor dem Verlassen des *Linear Solvers* in die verbleibenden Gleichungen eingesetzt.

Der *Valuation Solver* ist das Kernmodul des *Solvers*. Seine Aufgaben sind die Anwendung von Bewertungsstrategien zur Generierung der Lösungsreihenfolgen und das Lösen der nichtlinearen Gleichungen mit Hilfe der Macsyma-internen `SOLVE`-Funktion. Im Fall von Mehrfachlösungen prüft der *Valuation Solver* jede Einzellösung auf Konsistenz mit dem Restgleichungssystem. Inkonsistente Lösungen werden verworfen, während gültige Lösungen in das Restgleichungssystem eingesetzt und die zugehörigen Lösungspfade separat durch rekursive Aufrufe des *Valuation Solvers* verfolgt werden.

Alle für die Aufbereitung der Lösungen zur Ausgabe an den Anwender erforderlichen Schritte werden vom *Solver Postprocessor* übernommen. Hierunter fallen die Expandierung der vom *Valuation Solver* gelieferten, hierarchisch organisierten Lösungsliste, die Rücksubstitution der

symbolischen Lösungen sowie das Herausgreifen, Auswerten und Ausgeben der vom Benutzer gefragten Variablen und zusammengesetzten Ausdrücke.

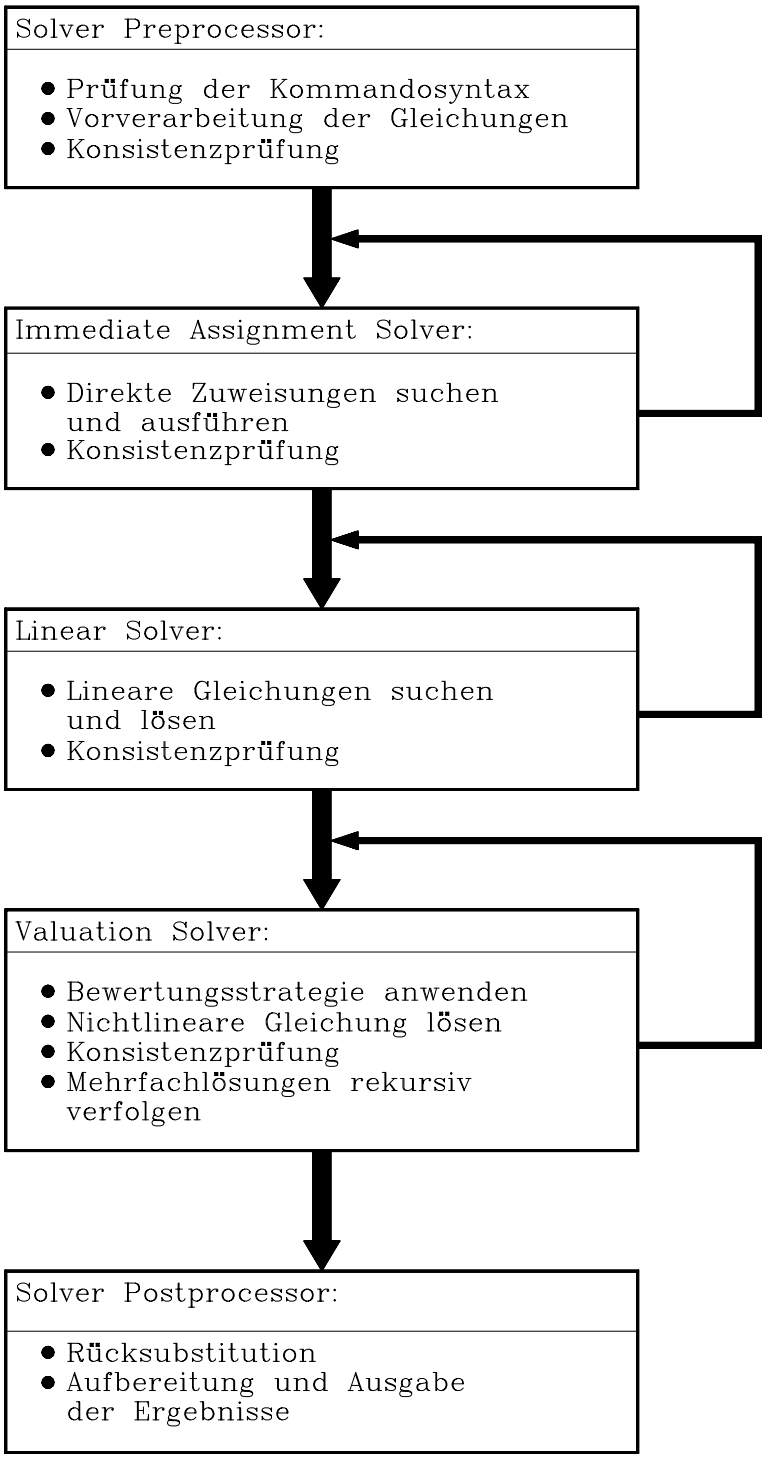

Abbildung 2.1: Struktur des Solvers

2.2 Die Module des Solvers

2.2.1 Der Solver Preprocessor

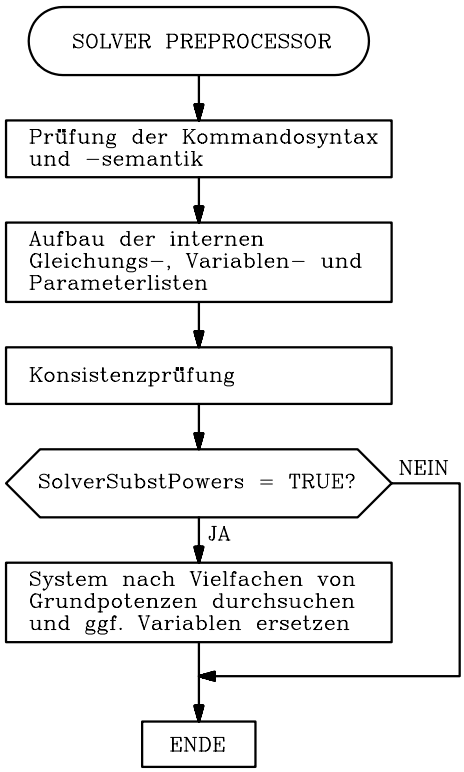

Abbildung 2.2: Ablaufdiagramm des Solver Preprocessors

Neben der Überprüfung der Kommandosyntax und -semantik sowie dem Aufbau der internen Gleichungs-, Variablen- und Parameterlisten führt der *Solver Preprocessor* eine Konsistenzprüfung der Gleichungen durch, deren Ablaufplan in Abbildung 2.3 dargestellt ist. Ziel der Konsistenzprüfung ist es, von vornherein zu erkennen, ob das Gleichungssystem aufgrund direkter Widersprüche unlösbar ist. Solche Widersprüche können in Form reiner Zahlengleichungen auftreten, z.B. $0 = 1$, aber auch in Form von Zwangsbedingungen zwischen als Parametern deklarierten Symbolen.

Um direkte Widersprüche aufzudecken, durchsucht die Routine zur Konsistenzprüfung das Gleichungssystem nach Gleichungen, die ausschließlich aus Zahlenwerten oder Zahlenwerten und Parametern bestehen, aber keine Variablen enthalten. Wird eine widersprüchliche Zahlengleichung gefunden, so bricht der *Solver Preprocessor* sofort ab. Ist eine solche Gleichung dagegen konsistent, wie beispielsweise $0 = 0$, wird sie aus dem Gleichungssystem entfernt, da sie die Lösbarkeit und Lösung des Systems nicht beeinflußt und somit redundant ist.

Etwas aufwendiger ist die Behandlung von Parameter-Zwangsbedingungen. Angenommen, es seien die Symbole A und B als Parameter eines Gleichungssystems definiert worden, das die Gleichung

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

enthält.

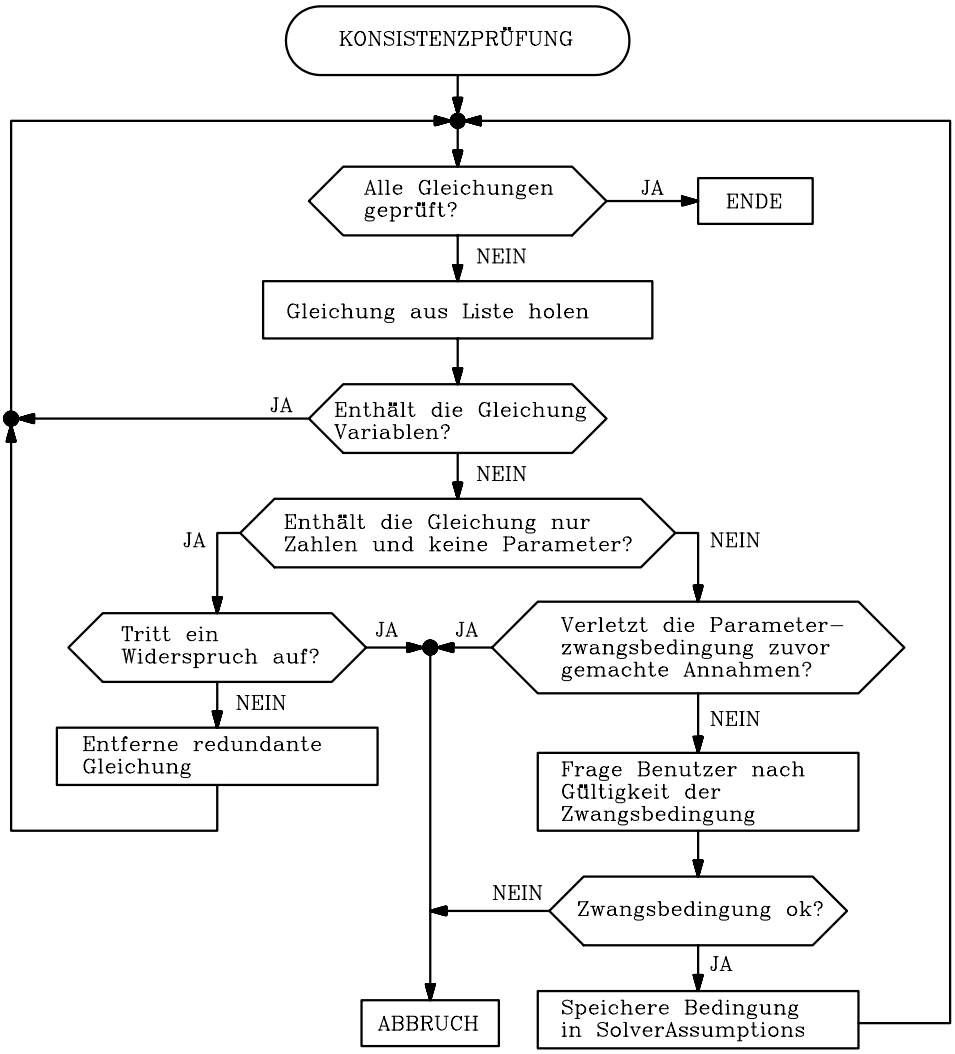

Abbildung 2.3: Ablaufdiagramm der Routine zur Konsistenzprüfung

Aus dieser Bedingung folgt, daß A und B keine *unabhängigen* Parameter sind und daher das Gleichungssystem nicht für beliebige Kombinationen ihrer Werte lösbar ist. Da sich bedingte Inkonsistenzen dieser Art bei vielen technischen Problemstellungen nicht ausschließen lassen, ist es nicht sinnvoll, den Lösungsvorgang in solchen Fällen einfach abzubrechen, es sei denn, die Parametergleichung widerspricht ihrerseits wiederum anderen Zwangsbedingungen im Gleichungssystem. Die Entscheidung, ob eine Parameter-Zwangsbedingung als zulässig betrachtet werden soll, wird daher von der Konsistenzprüfung an den Anwender delegiert. Im Fall der Gleichung (2.1) fragt das System auf folgende Weise nach der Gültigkeit der Bedingung:

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

Wird die Frage mit `p;` oder `n;` (*positive* bzw. *negative*) beantwortet, so bricht der Solver ab. Lautet die Antwort dagegen `z;` (*zero*), dann speichert die Konsistenzprüfung die Zwangsbedingung in einer globalen Liste namens `SolverAssumptions` ab, die nach dem Solver-Lauf inspiziert

werden kann. Anschließend entfernt die Konsistenzprüfung die nun redundante Gleichung aus dem System.

2.2.2 Der Immediate Assignment Solver

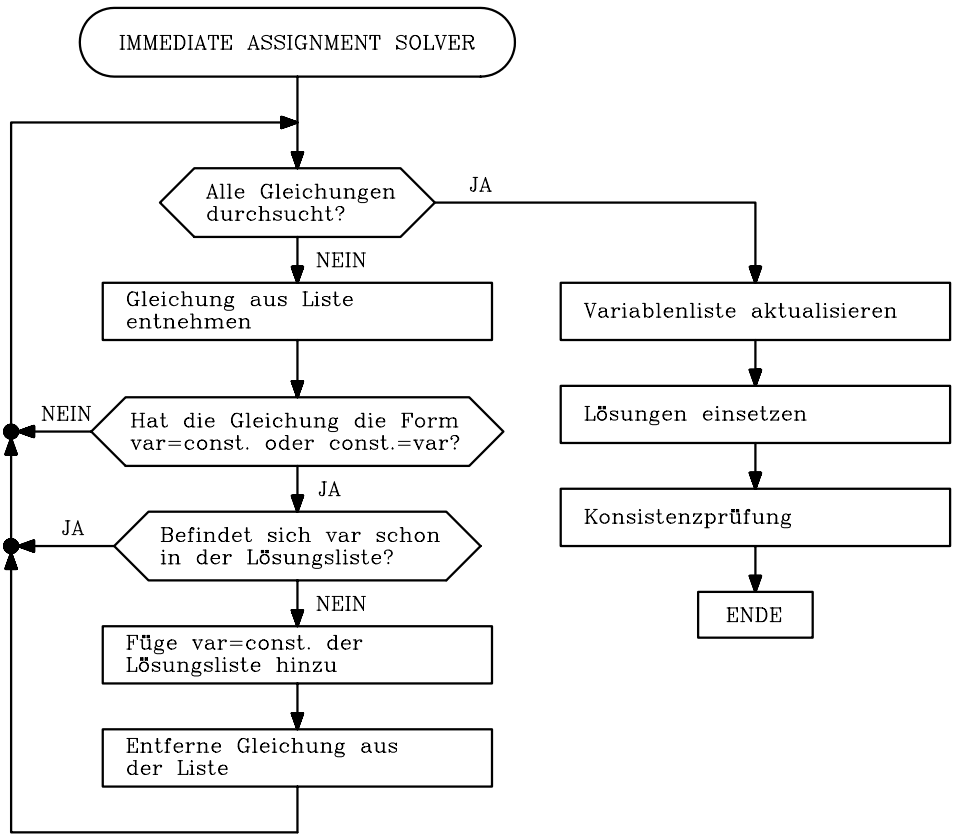

Abbildung 2.4: Ablaufdiagramm des Immediate Assignment Solvers

Die Aufgabe des *Immediate Assignment Solvers* besteht darin, das Gleichungssystem nach direkten Zuweisungen der Form $var = \text{const.}$ zu durchsuchen und solche Gleichungen unmittelbar zur Elimination der betreffenden Variablen zu benutzen. Bei maschinell aufgestellten Gleichungssystemen, wie etwa im Beispiel 1.2, kann durch eine solche Vorverarbeitung der Gleichungen oftmals der für die Extraktion und Lösung linearer Gleichungen im *Linear Solver* notwendige Rechenaufwand beträchtlich reduziert werden.

Der *Immediate Assignment Solvers* filtert in einer Schleife zunächst alle direkten Zuweisungen $var = \text{const.}$ und $\text{const.} = var$ aus dem System ab und speichert sie in der Form $var = \text{const.}$ in der Lösungsliste, wenn dort nicht bereits eine Lösung für var eingetragen ist. Im Anschluß daran wird das Gleichungssystem mit der Lösungsliste ausgewertet und wiederum auf Konsistenz geprüft.

2.2.3 Der Linear Solver

Der Ablauf des *Linear Solvers* beginnt, wie in Abschnitt 1.4.2 beschrieben, mit der Aufstellung der vollständigen Koeffizientenmatrix des symbolischen Gleichungssystems. Anhand dieser Koeffizientenmatrix wird die Bewertungsmatrix erstellt, die zur Extraktion linearer Teilgleichungssystem dient. Solange wie die Bewertungsmatrix noch Eins-Elemente enthält, d.h. $\sum S \neq 0$ und $\sum Z \neq 0$, wird die beschriebene heuristische Bewertungsstrategie angewendet, die darüber entscheidet, welche nichtlineare Gleichung entfernt wird bzw. welche in nichtlinearem Sinne auftretende Variable auf die rechten Seiten der Gleichung herübergebracht wird.

Ist die Bewertungsmatrix auf eine Nullmatrix reduziert worden, so wird eine Liste der verbliebenen (linearen) Gleichungen und eine Liste der linearen Variablen erstellt, die dem zur Lösung linearer Gleichungssysteme dienenden Macsyma-Befehl `LINSOLVE` als Funktionsparameter übergeben werden können. Zur Effizienzsteigerung wird vor dem Aufruf von `LINSOLVE` allerdings noch eine Besonderheit berücksichtigt, die bei der Beschreibung des Extraktionsalgorithmus nicht erwähnt wurde. Gelegentlich kann es vorkommen, daß gleichzeitig mit dem Streichen einer nichtlinearen Gleichung die einzige Instanz einer linearen Variablen x_j aus der gesamten Bewertungsmatrix entfernt wird, ohne daß dies unmittelbar bemerkt würde. Ebenso können Gleichungen entstehen, die zwar bezüglich der als linear erkannten Variablen nicht mehr nichtlinear sind, aber diese Variablen gar nicht mehr enthalten, d.h. die entsprechenden Koeffizienten sind gleich Null. Deshalb werden vor der Lösung des linearen Teilsystems noch einmal alle linearen Variablen daraufhin überprüft, ob sie noch in den linearen Gleichungen enthalten sind, und es wird nur die Teilmenge der extrahierten Gleichungen an `LINSOLVE` weitergegeben, die tatsächlich mindestens eine der linearen Variablen enthalten. Zwar wäre der *Linear Solver* auch in der Lage, das Gleichungssystem ohne diese zusätzlichen Maßnahmen zu lösen, doch häufig kann durch sie überflüssiger Rechenaufwand eingespart werden.

In Abschnitt 1.4.3 wurde demonstriert, daß beim Lösen überbestimmter linearer Gleichungssysteme Inkonsistenzen auftreten können, die nicht notwendigerweise die Unlösbarkeit des Gleichungssystems bedeuten. Werden solche Widerspruchsgleichungen, wie z.B. die rechte Seite von Gleichung (1.71), entdeckt[1], so werden sie — wie anfangs das gesamte Gleichungssystem — der Konsistenzprüfung unterzogen. Erweisen sich die Widerspruchsgleichungen als wahre Inkonsistenzen, so hat das Gesamtsystem keine Lösung, und der *Linear Solver* bricht mit einer entsprechenden Fehlermeldung ab. Enthalten die Widerspruchsgleichungen dagegen noch gesuchte Variablen, so werden sie aus dem linearen Gleichungssystem entfernt und als neue Zwangsbedingungen dem Restgleichungssystem hinzugefügt. Daraufhin wird `LINSOLVE` mit dem linear unabhängigen Teil der Gleichungen erneut aufgerufen (Inkonsistenzen können beim zweiten Durchlauf nicht mehr auftreten).

Wurden die linearen Gleichungen erfolgreich gelöst, so werden die Lösungen in das Restgleichungssystem eingesetzt und die Liste der gesuchten Variablen aktualisiert. Letzteres bedeutet, daß alle Variablen, für die vom *Linear Solver* eine Lösung gefunden wurde, aus der Liste entfernt werden, während neue Variablen, die in diesen Lösungen enthalten sind, der Liste hinzugefügt werden.

[1] Der Zugriff auf die Widerspruchsgleichungen ist bei Verwendung der Standardversion des `LINSOLVE`-Kommandos von außen nicht möglich. Daher war eine auf Lisp-Ebene speziell modifizierte Version des Befehls notwendig, die von Jeffrey P. Golden, Macsyma, Inc. (USA), auf Anfrage freundlicherweise zur Verfügung gestellt wurde.

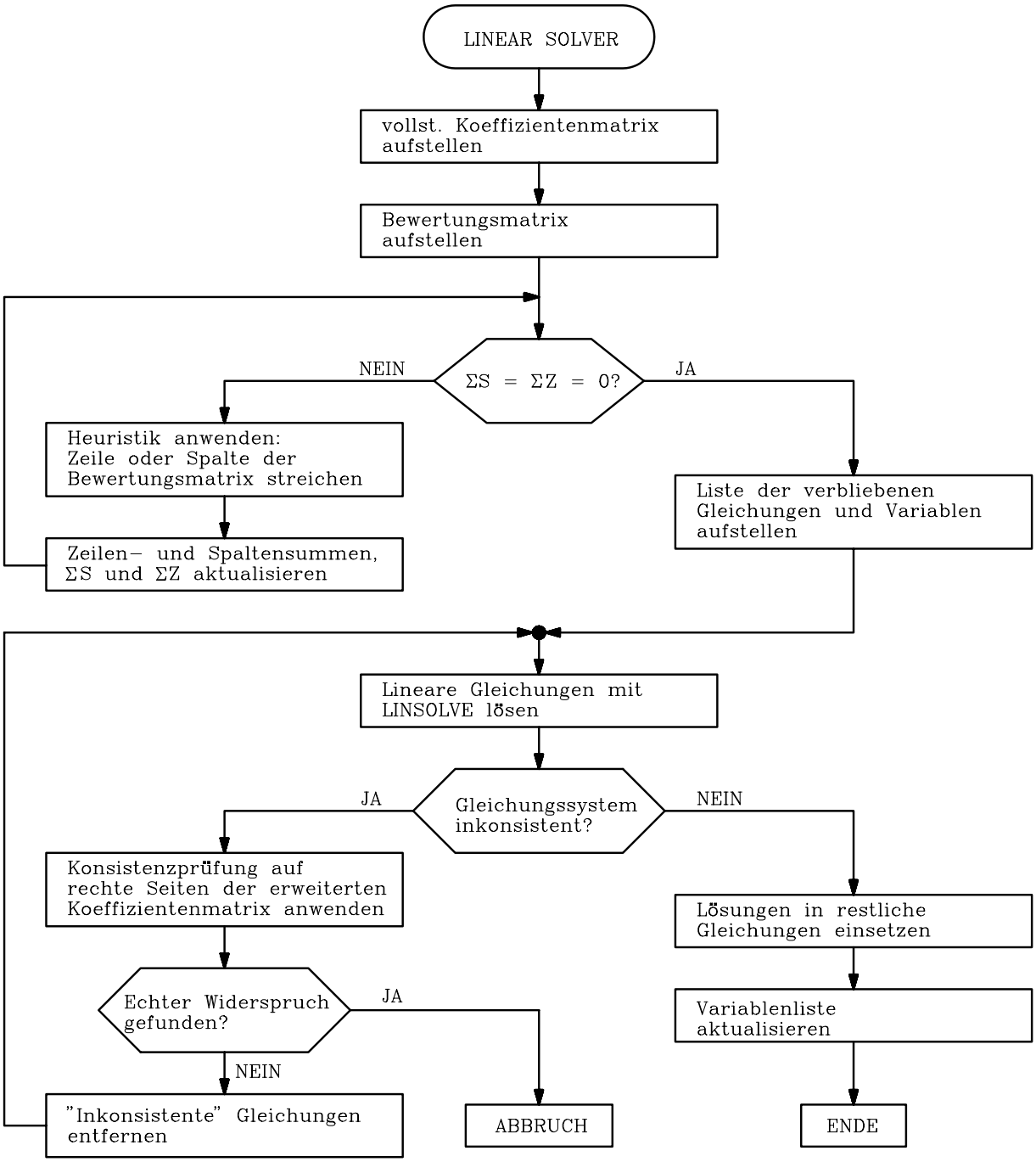

Abbildung 2.5: Ablaufdiagramm des Linear Solvers

2.2.4 Der Valuation Solver

Der *Valuation Solver* prüft zunächst, ob noch zu lösende Gleichungen und Variablen vorhanden sind. Ist dies der Fall, so werden für die restlichen Gleichungen zwei Bewertungsmatrizen aufgestellt, die als Grundlage für die Aufstellung der Lösungsreihenfolge dienen. Beide Matrizen haben die Dimensionen $n \times m$, wobei n die Anzahl der Gleichungen und m die Anzahl der momentan gesuchten Variablen ist. Die erste Matrix, die *Variablenpfadmatrix*, enhält für jede Gleichung die Anzahl der Variablenpfade (siehe Abschnitt 1.5.2) bezüglich jeder Variablen. Die zweite Matrix ist die *Bewertungsmatrix*. Ihre Elemente errechnen sich aus den heuristischen Operatorbaumbewertungen jeder Gleichung bezüglich jeder Variablen.

Auf diese zwei Matrizen wird anschließend eine — je nach Wunsch interne oder benutzerdefinierte — Bewertungsstrategie angewendet, die die Gleichungs/Variablen-Paare so ordnet, daß die ersten Elemente in dieser Liste möglichst erfolgversprechende Kandidaten für einen nachfolgenden Lösungsversuch mittels des `SOLVE`-Befehls sind.

Solange noch nicht alle Lösungsvorschläge in der Liste probiert wurden und noch keine korrekte Lösung für einen der Vorschläge berechnet werden konnte, wird anhand der ermittelten Lösungsreihenfolge die nächste Gleichung aus der Gleichungsliste ausgewählt und zu lösen versucht. Gelingt dies nicht, werden, wenn vorhanden, eine oder mehrere benutzerdefinierte Transformationsfunktionen zur Umformung der Gleichung angewendet und jeweils erneute Lösungsversuche vorgenommen. Sind auch diese ergebnislos, wird der nächste Lösungsvorschlag versucht.

Ist überhaupt keine der Gleichungen nach irgendeiner Variablen mehr lösbar, dann gibt der *Valuation Solver* zusätzlich zu allen bis zu diesem Punkt gefundenen Lösungen die ungelösten Gleichungen in impliziter Form zurück, so daß diese evtl. später mit einem numerischen Verfahren behandelt werden können. Bei einem erfolgreichen Lösungsversuch werden alle Einzellösungen (nichtlineare Gleichungen können Mehrfachlösungen haben) separat auf Konsistenz mit den restlichen Gleichungen geprüft. Diejenigen Lösungen, die zu widersprüchlichen Aussagen führen, werden verworfen und mit ihnen der zugehörige Lösungspfad.

Bleibt nach der Konsistenzprüfung keine Lösung übrig, so ist das Gleichungssystem inkonsistent, und der *Valuation Solver* bricht ab. Verbleibt genau eine Lösung, so wird diese an die Lösungsliste angehängt und in die restlichen Gleichungen eingesetzt. Abschließend wird die Liste der gesuchten Variablen aktualisiert, indem die soeben berechnete Variable aus ihr entfernt wird und gegebenenfalls in der Lösung enthaltene, neue Variablen hinzugefügt werden. Letzteres können Variablen sein, die nicht als Parameter aber auch nicht als gesucht angegeben wurden. Die Haupt-Lösungsschleife des *Valuation Solvers* beginnt sodann wieder von vorne.

Bei konsistenten Mehrfachlösungen müssen alle Lösungspfade getrennt verfolgt werden. Dazu ruft sich der *Valuation Solver* mit den verbleibenden Gleichungen und Variablen für jede Einzellösung rekursiv selbst auf und speichert die zurückerhaltenen Ergebnisse in einer hierarchisch strukturierten Lösungsliste. Deren Expandierung ist Aufgabe des im nächsten Abschnitt beschriebenen *Solver Postprocessors*.

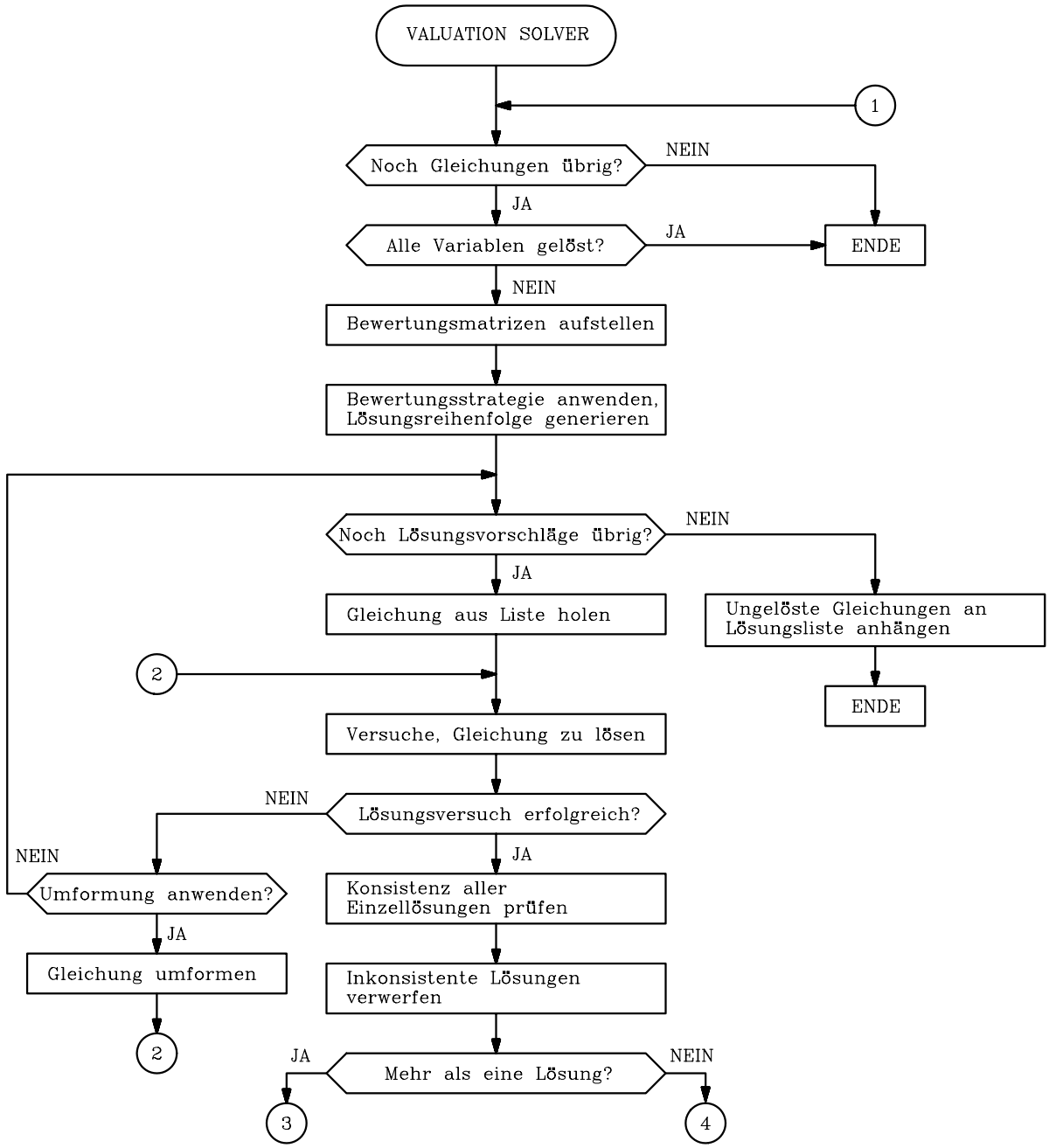

Abbildung 2.6: Ablaufdiagramm des Valuation Solvers (Teil 1)

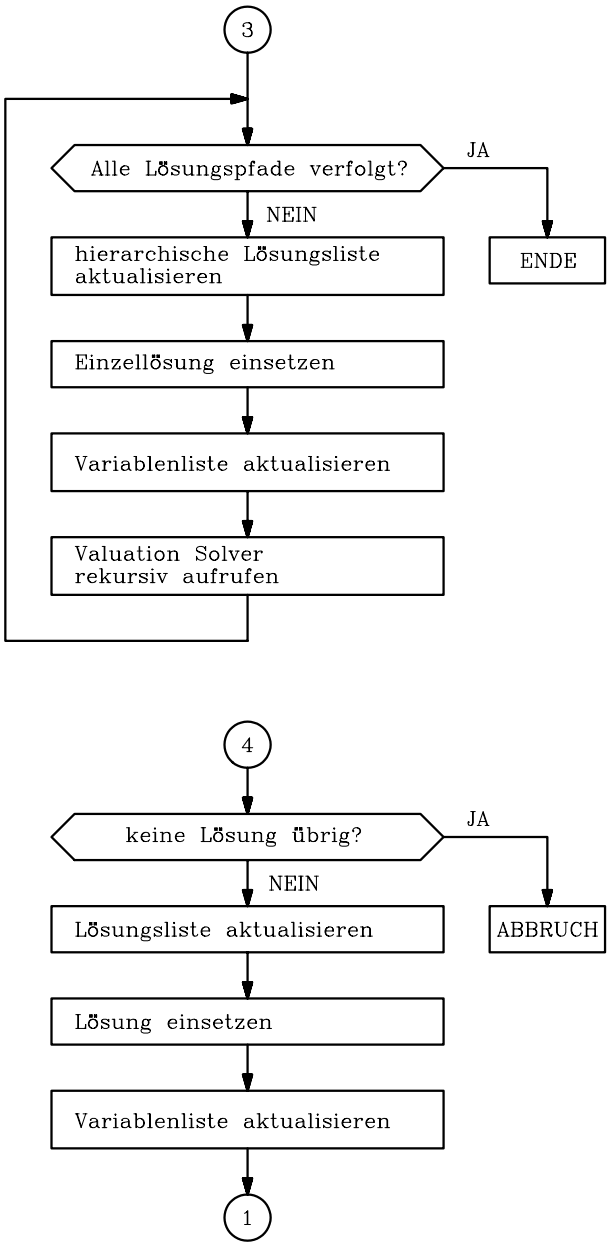

Abbildung 2.7: Ablaufdiagramm des Valuation Solvers (Teil 2)

2.2.5 Der Solver Postprocessor

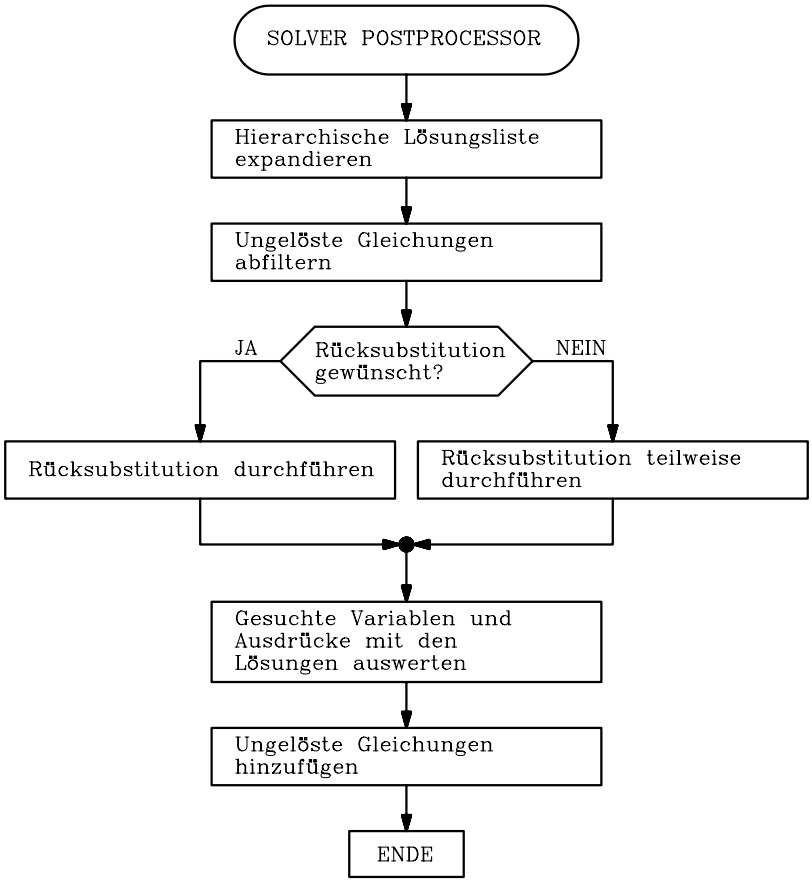

Abbildung 2.8: Ablaufdiagramm des Solver Postprocessors

Da der *Valuation Solver* im Fall von Mehrfachlösungen nichtlinearer Gleichungen wegen der rekursiven Verfolgung der Lösungspfade eine hierarchisch strukturierte Ergebnisliste als Funktionswert liefert, muß der *Solver Postprocessor* zu Beginn die Listenhierarchie auflösen, damit die Rücksubstitution durchgeführt werden kann.

Wenn nicht alle Gleichungen innerhalb eines Lösungspfades symbolisch gelöst werden konnten, dann enthält die Ergebnisliste zusätzlich zu den Variablen, für die eine Lösung gefunden wurde, die Liste der restlichen, ungelösten nichtlinearen Gleichungen. Diese werden vorläufig aus der Gleichungsliste entfernt und zwischengespeichert, um später an das Endergebnis angefügt zu werden.

Je nach Wunsch des Anwenders erfolgt im Anschluß daran die Rücksubstitution des Gleichungssystems, das bis zu diesem Zeitpunkt noch in oberer Dreiecksform vorliegt. Dazu wird die Lösungsliste solange iterativ mit sich selbst ausgewertet, bis keine Änderung der Ergebnisse mehr zu verzeichnen ist. Auch wenn die Rücksubstition vom Anwender nicht gewünscht wird[2], wird sie dennoch durchgeführt, aber nur so weit, wie notwendig ist, um alle nicht in der Befehlszeile angegebenen Variablen aus den Lösungen zu eliminieren. Soll z.B. ein Gleichungssystem

[2]Dies erfordert die Belegung der Optionsvariablen `SolverBacksubst` mit `FALSE` (siehe Abschnitt 2.4)

in den Variablen x, y, z und w nur nach den Variablen x und y gelöst werden und liefert der Lösungsprozeß die Dreiecksform

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

so führt die vollständige Rücksubstitution zu dem Ergebnis

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

Bei ausgeschalteter vollständiger Rücksubstitution werden dagegen die beiden Gleichungen

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

zurückgegeben, die die internen Variablen z und w nicht mehr enthalten, aber weiterhin untereinander durch die Variable y gekoppelt sind.

Nach Abschluß der Rücksubstitution und der Auswertung der in der Variablenliste der Kommandozeile angegebenen zusammengesetzten Ausdrücke mit den Lösungen werden die anfangs abgefilterten, ungelösten Gleichungen wieder an die Lösungsliste angehängt. Die fertiggestellte Lösungsliste wird sodann als Funktionswert an den Anwender übergeben.

2.3 Anwendung des Solvers

2.3.1 Kommandosyntax

Der Aufruf der Solvers von der Macsyma-Kommandoebene aus erfolgt mit der Syntax

```
Solver( Gleichungsliste, Variablenliste, Parameterliste )
```

oder, wenn das zu lösende Gleichungssystem keine Parameter enthält, auch mit

```
Solver( Gleichungsliste, Variablenliste ) .
```

Die Gleichungsliste ist eine Liste von Macsyma-Objekten, für die `EquationP` gleich `TRUE` ist. Das Gleichungssystem (1.55) – (1.57) wird somit auf folgende Weise angegeben:

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

Die gesuchten Variablen werden dem Solver ebenfalls in Form einer Liste mitgeteilt, z.B. als

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

für das obenstehende Gleichungssystem. Neben rein atomaren Variablensymbolen (`SymbolP`) darf die Variablenliste auch zusammengesetzte Ausdrücke in den gesuchten Unbekannten enthalten. Werden für das Gleichungssystem nicht explizit die Variablen x, y und z, sondern vielmehr x und der Wert $\sin(\pi yz)$ gesucht, so lautet die Variablenliste

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

Die Parameterliste darf dagegen ausschließlich atomare Symbole enthalten, also nur Objekte, für die `SymbolP` gleich `TRUE` ist. Zusammengesetzte Ausdrücke sind hier weder zulässig noch sinnvoll.

2.3.2 Besonderheiten der Syntax von Gleichungen

An dieser Stelle sei auf einige Unterschiede der Kommandosyntax im Vergleich mit den zulässigen Aufrufsformen der Macsyma-internen `SOLVE`-Funktion hingewiesen. Letzterer dürfen die Gleichungen auch in *Expression*-Form übergeben werden, d.h. als Ausdrücke mit `EquationP` = `FALSE`, die implizit als Gleichungen der Form *Expression* $= 0$ aufgefaßt werden:

```
[
    x + 2*y -     z - 6,  ← nicht zulässig
  2*x + y*z -   z^2 + 1,
  3*x -   y + 2*z^2 - 3
]$
```

Diese Form der Gleichungsdarstellung wird zwar intern, z.B. vom *Valuation Solver*, benutzt, doch ist sie beim Aufruf des Solvers in der Kommandozeile nicht erlaubt. Die `SOLVE`-Funktion gestattet ferner das Weglassen der Listenklammern um die Argumente, wenn nur eine einzige Gleichung und/oder nur eine einzige Variable übergeben werden sollen. Auch dies ist bei der Benutzung des Solvers nicht zulässig.

2.3.3 Beispielaufrufe des Solvers

Beispiel 2.1

Für das in `COM5` eingegeben Gleichungssystem ist ein korrekter Aufruf des Solvers der Befehl in der Kommandozeile `COM7`. Die Angabe der berechneten Lösungen erfolgt in Form einer Liste von Lösungslisten, siehe Ausgabezeile `D7`.

```
(COM6) MsgLevel : 'DETAIL$    /* siehe Abschnitt 2.4 */

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

Ausgaben des *Solver Preprocessors*:

```
The variables to be solved for are [X, Y, Z]
Checking for inconsistencies...
```

```
... none found.
```

Ausgaben des *Immediate Assignment Solvers*:

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

Ausgaben des *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.
```

Ausgaben des *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
```

Hier wird keine Komplexitätsbewertung benötigt,
denn es sind nur eine Gleichung und eine Variable übrig.

```
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.
```

Einzelne Konsistenzprüfung bei Mehrfachlösungen:

```
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]
```

Rekursive Verfolgung aller drei Lösungspfade:

```
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.
```

Ausgaben des *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
```

Zur besseren Übersicht folgt das Ergebnis noch einmal im TEX-Satz:

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

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

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

□

Beispiel 2.2

Als zweites Beispiel soll das in a und b parametrisierte Gleichungssystem

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

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

nach den Variablen x und y sowie dem zusammgesetzten Ausdruck x/y gelöst werden. Da das Gleichungssystem keine direkten Zuweisungen enthält und eine wiederholte Suche nach linearen Gleichungen in diesem Fall nicht sinnvoll ist, werden der *Immediate Assignment Solver* und die Wiederholungsschleife des *Linear Solvers* mit dem Befehl `COM8` abgeschaltet:

```
(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 Die Optionen des Solvers

Von der Macsyma-Kommandoebene (`Macsyma toplevel`) oder von Programmdateien aus kann das Verhalten des Solvers durch die individuelle Belegung einer Reihe von Optionsvariable beeinflußt werden, die im folgenden aufgelistet und erläutert werden. Die hinter den Namen der Optionsvariablen in spitzen Klammern angegebenen Werte bzw. Symbole stellen die im Programm vorgesehenen Standardbelegungen dar, die beim erstmaligen Laden des Moduls `SOLVER` automatisch entsprechend gesetzt werden.

`MsgLevel <SHORT>` (*message level*) steuert den Umfang der während des Programmlaufs ausgegebenen Bildschirmmeldungen. Zulässig sind die Belegungen `OFF`, `SHORT` und `DETAIL`. Wird `MsgLevel : OFF` gesetzt, so werden alle Programmausgaben vollständig unterdrückt. Bei einer Belegung mit `SHORT` werden lediglich Statusinformationen zur Verfolgung des Programmablaufs ausgegeben. Das Schlüsselwort `DETAIL` dagegen bewirkt zusätzlich die Ausgabe aller von den Solver-Modulen berechneten Zwischenergebnisse und sowie von Meldungen über die aufgrund der Heuristiken getroffenen Entscheidungen.

`SolverImmedAssign <TRUE>` schaltet den *Immediate Assignment Solver* ein (`TRUE`) bzw. aus (`FALSE`). Ist das Modul eingeschaltet, so wird vor dem Aufruf des *Linear Solvers* das Gleichungssystem nach direkten Zuweisungen der Form *var* = const. oder const. = *var* abgesucht, die sofort in die restlichen Gleichungen eingesetzt werden können.

`SolverRepeatImmed <TRUE>` bestimmt, ob der *Immediate Assignment Solver* wiederholt (`TRUE`) aufgerufen wird, bis keine direkten Zuweisungen mehr gefunden werden, oder ob nur ein einziger Aufruf erfolgt (`FALSE`).

`SolverSubstPowers <FALSE>` (*substitute powers*) steuert die Behandlung von Variablen, die in Potenzen p_k ganzzahliger Vielfacher $p_k = kp_0$, $k \in I\!N$, einer Grundpotenz $p_0 \in I\!N \setminus \{1\}$ auftreten. Enthält das Gleichungssystem z.B. die Variable x ausschließlich in den Potenzen x^2, x^4, x^6, ..., d.h. $p_0 = 2$, dann wird bei `SolverSubstPowers : TRUE` $x^{p_0} = x^2$ durch ein

neues Variablensymbol $X2$ substituiert, das somit nur noch in den Potenzen $X2$, $X2^2$, $X2^3$, ..., auftritt. Auf diese Weise reduziert sich der Grad der zu lösenden Gleichungen und die zu erwartende Lösungsvielfalt. Es ist jedoch gegebenenfalls eine Nachbearbeitung der Lösungen erforderlich.

`SolverInconsParams <ASK>` (*inconsistent parameter handling*) beeinflußt das Verhalten der Routine zur Konsistenzprüfung. Zulässige Belegungen sind `ASK`, `BREAK` und `IGNORE`. Wird während der Konsistenzprüfung eine Abhängigkeit zwischen Parametern entdeckt, so wird bei `SolverInconsParams : ASK` der Anwender nach der Gültigkeit der entsprechenden Zwangsbedingung gefragt. Diese wird bei positiver Antwort zur späteren Auswertung in der Liste `SolverAssumptions` abgespeichert. Ist die Abhängigkeit nicht zulässig oder ist die Optionsvariable mit `BREAK` belegt, so wird der Lösungsvorgang abgebrochen. Eine Belegung mit `IGNORE` veranlaßt die Konsistenzprüfung dazu, grundsätzlich alle Zwangsbedingungen zwischen Parametern als gültig anzuerkennen, solange diese nicht direkt bereits gemachten Annahmen widersprechen.

`SolverLinear <TRUE>` schaltet den *Linear Solver* ein (`TRUE`) bzw. aus (`FALSE`). Das Ausschalten empfiehlt sich dann, wenn das zu lösende Gleichungssystem keine linearen Gleichungen enthält oder deren Anzahl im Verhältnis zur Zahl der nichtlinearen Gleichungen sehr klein ist. In diesen Fällen kann durch das Umgehen des *Linear Solvers* viel Rechenzeit gespart werden, da der Algorithmus zur Suche nach linearen Gleichungen recht aufwendig ist.

`SolverRepeatLinear <TRUE>` veranlaßt das wiederholte Aufrufen des *Linear Solvers*. Wird die Variable auf `FALSE` gesetzt, so wird der *Linear Solver* nur ein einziges Mal durchlaufen.

`SolverFindAllLinearVars <TRUE>` entscheidet, ob der *Linear Solver* nach maximal großen linearen Teilgleichungssystem hinsichtlich aller vorhandener Variablen sucht (`TRUE`), oder ob nur Teilsysteme in den Variablen extrahiert werden sollen, die während des Lösungsprozesses augenblicklich gesucht werden (`FALSE`). Die Belegung der Variablen spielt besonders dann eine Rolle, wenn unterbestimmte Gleichungssysteme gelöst werden sollen. Hier sollte `SolverFindAllLinearVars : FALSE` gesetzt werden, denn sonst können möglicherweise aufgrund der zu geringen Zahl von Gleichungen keine Lösungen für die ursprünglich interessierenden Variablen gefunden werden. Mit der Belegung `FALSE` wird sichergestellt, daß zuerst nach diesen Variablen gelöst wird und die Freiheitsgrade in anderen Unbekannten ausgedrückt werden.

`SolverValuationStrategy <MinVarPathsFirst>` enthält den Namen der Funktion, die aus der Variablenpfadmatrix und der Bewertungsmatrix eine Lösungsreihenfolge generiert (siehe Abschnitt 2.7). Der Aufruf der Funktion geschieht innerhalb des *Valuation Solvers* mit:

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

Als Funktionswert wird eine Liste der Form

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

erwartet, wie sie in Abschnitt 1.5.2 beschrieben wurde. Zu beachten ist hierbei die Optionsvariable `SolverMaxLenValOrder`.

`SolverDefaultValuation <10>` bestimmt den Bewertungsfaktor für Operatoren, denen nicht explizit ein solcher Faktor mit dem `SetProp`-Befehl zugewiesen wurde (siehe Abschnitt 2.6).

`SolverMaxLenValOrder <5>` (*maximum length of valuation order*) bestimmt die maximale Länge der Lösungsreihenfolge. Führt auch der letzte Lösungsvorschlag in der Liste nicht zum Erfolg, so bricht der *Valuation Solver* den Lösungsprozeß ab, auch wenn noch nicht alle Gleichungs/Variablen-Paare durchprobiert wurden.

`SolverTransforms <[]>` enthält eine Liste der Namen von Funktionen, die nach einem erfolglosen Lösungsversuch auf die betreffende Gleichung angewendet werden können, um die Lösungschancen bei einem erneuten Versuch zu erhöhen. Die Funktionen werden dabei in der Reihenfolge ihres Auftretens in der Liste ausgeführt. Nach jedem Funktionsaufruf erfolgt unmittelbar der nächste Lösungsversuch. Scheitert auch dieser, so wird die nächste Transformation in der Liste angewendet, solange bis die Gleichung gelöst werden konnte oder keine weitere Transformation mehr vorhanden ist. Da die Manipulationsmöglichkeiten mit den Transformationen recht umfangreich sind, findet sich eine genauere Beschreibung ihrer Definition und Anwendung in Abschnitt 2.5.

`SolverPostprocess <TRUE>` schaltet den *Solver Postprocessor* ein (`TRUE`) bzw. aus (`FALSE`). In ausgeschaltetem Zustand wird die hierarchische Ergebnisliste des Solvers ohne Nachbearbeitung zurückgegeben, also ohne Expandierung, Rücksubstitution und Extraktion der vom Benutzer erfragten Variablen. Diese Steuervariable dient in erster Linie Debug-Zwecken.

`SolverBacksubst <TRUE>` (*backsubstitution*) bestimmt, ob der *Solver Postprocessor* eine Rücksubstitution des auf Dreiecksform gebrachten Gleichungssystems durchführen soll (`TRUE`) oder nicht (`FALSE`). Sind die berechneten symbolischen Lösungen sehr umfangreich, so ist es oft sinnvoll, keine vollständige Rücksubstitution auszuführen, sondern einige der gesuchten Variablen als Funktionen anderer berechneter Unbekannter auszugeben.

`SolverDispAllSols <FALSE>` (*display all solutions*) weist, wenn auf `TRUE` gesetzt, den Solver an, zum Abschluß des Lösungsprozesses *alle* gefundenen Lösungen auszugeben und nicht nur die für die vom Benutzer angegebenen Variablen.

`SolverRatSimpSols <TRUE>` (*perform rational simplifications on solutions*) weist den *Solver Postprocessor* an, die Ergebnisse vor der Ausgabe noch einmal mit dem Befehl `FullRatSimp` zu vereinfachen.

`SolverDumpToFile <FALSE>` veranlaßt auf Wunsch (`TRUE`) den *Valuation Solver* dazu, nach jedem erfolgreichen Lösungsversuch alle bisher gefundenden Lösungen sowie die verbleibenden Gleichungen und Variablen in eine Macsyma-Batch-Datei zu schreiben, deren Name in der Optionsvariablen `SolverDumpFile` enthalten ist. Diese Option ist für umfangreiche Probleme vorgesehen, in denen Macsyma bei akutem Mangel an Haupt- und Swap-Speicher zu Systemabstürzen neigt. Durch die Speicherung der Zwischenergebnisse kann bei einem Absturz zumindest ein Teil der Ergebnisse gerettet werden.

`SolverDumpFile <"SOLVER.DMP">` enthält den Namen der Datei, in die die Zwischenergebnisse geschrieben werden sollen.

2.5 Definition und Einbindung benutzerspezifischer Transformationsroutinen

Nicht selten stößt der *Valuation Solver* während des Lösungsprozesses auf Gleichungen, die er nicht zu lösen vermag, obwohl bereits einige einfache Umformungen oder Zusammenfassungen helfen könnten, um das gewünschte Ergebnis zu erhalten. So ist die `SOLVE`-Funktion z.B. nicht in der Lage, die Gleichung

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

korrekt nach x aufzulösen, da sie nicht bemerkt, daß sich die quadratischen Sinus- und Cosinus-Terme mühelos zu 1 zusammenfassen lassen.

Um je nach Anwendungsfall passende Umformungen der Gleichungen durchführen zu können, ist im Solver die dynamische Einbindung benutzerdefinierter Transformationsfunktionen vorgesehen, die bei Bedarf auf ungelöste Gleichungen angewendet werden. Diesen Funktionen übergibt der *Valuation Solver* unter anderem die Gleichung und die Variable, nach der die Gleichung zu lösen ist. Die Transformationsfunktion hat nun die Aufgabe, die Gleichung umzuformen und als ihren Funktionswert wieder an den Solver zurückzugeben, damit ein erneuter Lösungsversuch begonnen werden kann.

Der Aufruf einer Transformationsfunktion aus dem Solver heraus geschieht auf folgende Weise:

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

Dabei enthält der zusätzlich übergebene Parameter `Solution` das Ergebnis des gescheiterten Lösungsversuchs, anhand dessen möglicherweise hilfreiche Schlußfolgerungen gezogen werden können. Der folgende Befehl zeigt ein Beispiel für die Definition einer einfachen, benutzerspezifischen Transformationsroutine, die versucht, eine Gleichung durch Zusammenfassung trigonometrischer Funktionen lösbar zu machen:

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

Zur Einbindung der Funktion muß ihr Name in die globale Liste `SolverTransforms` eingetragen werden:

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

Da auch die Anwendung einer Transformationsroutine erfolglos sein kann, besteht die Möglichkeit, dem Solver den Mißerfolg des Umformungsversuchs durch die Rückgabe einer leeren Liste (`[]`) als Funktionswert anzuzeigen. Auf diese Weise wird verhindert, daß ein weiterer Aufruf der

SOLVE-Funktion mit derselben Gleichung erfolgt. Stattdessen versucht der Solver unmittelbar, wenn vorhanden, die nächste Transformation in der Liste auszuführen.

Als Alternative zur Umformung der Gleichung und anschließenden Lösung durch den *internen* Solver ist es einer Transformationsroutine auch gestattet, die Lösungen für die übergebene Gleichung *selbst* zu bestimmen und in der Form

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

zurückzugeben. Eine mögliches Anwendungsgebiet für solche Funktionen wäre der Einsatz numerischer Verfahren zur Lösung nichtlinearer Gleichungen, die nur eine einzige Variable und keine Parameter enthalten. Oder es kann, indem die Transformationsroutine einen *Macsyma-Break* aufruft, die Gleichung von Hand manipuliert werden.

Die Vorgehensweise zur Definition und Einbindung von Transformationsfunktionen mit Erfolgsrückmeldung soll im folgenden an einem komplett durchgerechneten Beispiel demonstriert werden. Zu lösen sei das Gleichungssystem (2.16) – (2.18) nach den Variablen x, y und z. Die Aufgabe ist zwar im Prinzip recht einfach, aber dennoch bereitet sie dem Solver erhebliche Schwierigkeiten.

$$z - \sin x = 0 \tag{2.16}$$
$$y + z^2 + \cos x^2 = 5 \tag{2.17}$$
$$y + x = 1 \tag{2.18}$$

Zunächst werden das Gleichungssystem als Gleichungsliste sowie die Variablen eingegeben:

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

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

Der erste Lösungsversuch soll ohne Transformationsfunktionen durchgeführt werden:

```
(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]]]
```

Im obenstehenden Fall vermag der Solver die Gleichung mit den quadratischen Sinus- und Cosinus-Termen nicht nach der Variablen x aufzulösen und gibt daher die ungelöste Gleichung in impliziter Form als letztes Element der Lösungsliste zurück.

Zur Vereinfachung der Gleichungen sollen nun zwei Transformationsfunktionen definiert und eingebunden werden. Die erste Transformation dient der Zusammenfassung von Logarithmus-Termen, die zweite zur Vereinfachung von Ausdrücken, die trigonometrische Funktionen enthalten. Beide Funktionen überprüfen selbst, ob ihre Anwendung erfolgreich war und geben eine leere Liste zurück, wenn die Transformation keine Änderung der Gleichung bewirkt hat.

```
(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 )
)$
```

Die Transformationsfunktion `TransformLog` möge grundsätzlich als erste angewendet werden, daher erfolgt die Einbindung der Routinen in der Reihenfolge:

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

Nun kann ein neuer Solver-Lauf begonnen werden:

```
(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.
```

An dieser Stelle stellt der Solver wie im vorhergegangenen Fall fest, daß er die Gleichung nicht lösen kann und wendet die erste Transformation in der Liste auf sie an.

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

Da die Gleichung keine Logarithmus-Funktionen enthält, ist der Umformungsversuch nicht erfolgreich. Deshalb wird die zweite Transformationsroutine versucht.

```
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)]]
```

Der Transformationsfunktion `TransformTrig` gelingt die Zusammenfassung der Gleichung, so daß der Solver anschließend in der Lage ist, die korrekte Lösung für die Variable x zu bestimmen.

2.6 Änderung der Operatorbewertungen

Um dem Anwender des Solvers die Möglichkeit zur individuellen Beeinflussung der heuristischen Komplexitätsbewertung von algebraischen Ausdrücken zu geben, sind die Bewertungen der arithmetischen Operatoren nicht als unveränderbare Konstanten, sondern als *Macsyma Properties* unter dem Schlüsselwort `Valuation` abgelegt. Die Definition eines Bewertungsfaktors erfolgt mit dem Befehl

`SetProp(` *Operator*`, 'Valuation,` *Bewertungsfaktor* `)$` .

Die im Solver vordefinierten Standardbelegungen der Bewertungsfaktoren wurden mit den Befehlen

```
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 )$
```

festgesetzt. Soll z.B. die Bewertung des tanh-Operators auf 20 geändert werden, so kann dies jederzeit durch

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

erreicht werden. Abfragen läßt sich ein Bewertungsfaktor mit dem `Get`-Kommando:

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

Beispiel: Die Bewertung des "$*$"-Operators wird ermittelt durch:

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

(D19)                                4
```

Wenn die Abfrage als Ergebnis den Wert `FALSE` liefert, dann bedeutet dies, daß kein Bewertungsfaktor für den betreffenden Operator definiert wurde.

2.7 Definition und Einbindung benutzerspezifischer Bewertungsstrategien

Mittels einer Umbelegung der Prozedurvariablen `SolverValuationStrategy` wird der *Valuation Solver* veranlaßt, anstelle der internen Funktion eine benutzerdefinierte Bewertungsstrategie zur Bestimmung der Lösungsreihenfolge zu verwenden. Der Funktion werden durch ihren Aufruf mit

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

zwei Bewertungsmatrizen, die *Variablenpfadmatrix* und die *Komplexitätsbewertungsmatrix*, übergeben, deren Zeilendimension gleich der Anzahl der Gleichungen und deren Spaltendimension gleich der Anzahl der zu bestimmenden Variablen ist.

Der Eintrag an der Position (i, j) der Variablenpfadmatrix entspricht der Anzahl der Pfade zu den Instanzen der Variable x_j in Gleichung i (siehe Abschnitt 1.5.2). Die Komplexitätsbewertungsmatrix enthält an der Position (i, j) die Komplexitätsbewertung der Gleichung i hinsichtlich der Variablen x_j. Für das Gleichungssystem (1.55) – (1.57) lautet die Pfadmatrix somit

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

und die Bewertungsmatrix wird bei Verwendung der Standard-Operatorbewertungsfaktoren berechnet zu

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

Aus diesen Matrizen muß die Bewertungsstrategie eine Lösungsreihenfolge (siehe Abschnitt 1.5.2) in der Form

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

generieren. Die maximale Länge der Liste sollte dabei nicht größer als der Wert der Optionsvariablen `SolverMaxLenValOrder` sein.

Die Grobstruktur einer Bewertungsstrategie wird durch den folgenden Pseudocode wiedergegeben:

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

  Generiere eine Lösungsreihenfolge aus den Bewertungstrategien

  Beschränke die Länge der Liste auf SolverMaxLenValOrder Elemente

  RETURN( Lösungsreihenfolge )
)$
```

Zur Einbindung der Funktion ist anschließend die Prozedurvariable `SolverValuationStrategy` mit dem Funktionsnamen zu belegen:

```
SolverValuationStrategy : 'MyValuationStrategy$
```

Literaturverzeichnis

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

[BRO_88] E. Brommundt, G. Sachs, *Technische Mechanik — Eine Einführung*, 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 für Netzwerktheorie und Schaltungstechnik, Technische Universität 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

[NÜH_89] D. Nührmann, *Professionelle Schaltungstechnik*, München: 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 rechnergestützten Entwurf analoger Schaltungen", Dissertation am Institut für Netzwerktheorie und Schaltungstechnik, Technische Universität Braunschweig, Juli 1993

[SOM_93b] R. Sommer, E. Hennig, G. Dröge, 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 für Netzwerktheorie und Schaltungstechnik, Technische Universität Braunschweig, Juli 1991

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

Anhang A

Beispielrechnungen

A.1 Dimensionierung des Stabzweischlags

Zur Demonstration der Leistungsfähigkeit des Solvers werden abschließend die in der Einleitung angeführten Beipielaufgaben 1.1 und 1.2 gelöst. Im folgenden finden sich die vollständigen Bildschirmausgaben der beiden Programmläufe.

```
(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))]]
```

Für das den Stabzweischlag beschreibende Gleichungssystem ermittelt der Solver insgesamt vier verschiedene analytische Lösungen, die sich nur durch die Kombination ihrer Vorzeichen unterscheiden. Aufgrund der physikalischen Randbedingung, daß die Längen h_1 und h_2 nicht negative Werte annehmen können, ist nur die letzte Lösung sinnvoll:

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

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

A.2 Dimensionierung des Transistorverstärkers

```
(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)
         /(13655289063030312140 8*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#

116383372395290156600000000 A VCC + 64411648281412414613054367998943189195#

57829438130394669732330511352796600 0 A R3 + 135601779249796410015811714375#

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 0 A R1 VCC

                            2
 - 15664635352383720279 A R1  + (- 11852193632588107801380000 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

 - 86583882087668323961262747438838206882917398923834769170895994880000000 A

 ZOUT) + A

 (73094113258409599088098011387867214250558868171501312134070398877696000

 ZOUT + ZIN

 (- 82164830677623835681150914833206609917142422498519656448000000000 ZOUT

 - 89698008560556767832189447109189280381589732951869815470000000000000))

 + 73091104214643137701992515955008538217110816411520639295650692857856000

         2
 ZOUT + A  (50416680420198682402077210492446131565566605185394287109375

    2
 ZIN  - 89701701283993131929871268090550778748852308577743756270000000000

 ZIN))/(A

 (- 34720136717154997908466361722974120960049876968457742437529293946880

 ZOUT

 - 21092632483111174683325097183189754403716708919298794191829902950400 0)

                                                                        2
 + 42608839392183423245532312845665555885204662093905764350000000000 A

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

```
R5 = (38195771383195691504052086845016246794667687936 ZOUT

 + A (9930399595766410870496554922774419242l875 ZIN

 + 5996575962274855332613362021809166941397722880l0)

 + 8339540409811836894068029655629814665587863808000)

/(39733737113751639640009221921695330300641958l0 A),

R6 = (- 38195771383195691504052086845016246794667687936 ZOUT

 + A (46874167045633050797473168741069996757812l ZIN

 - 26870982895201987651908310872027897991106764800l0)

 + 98790378291183778132015848794220213202598400000l A VCC

 - 8339540409811836894068029655629814665587863808000)

/(12210490746832244925030394491952536145488422l 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
```

Auch hier werden mehr als eine Lösung des Systems berechnet, doch stellt nur die erste ein physikalisch sinnvolles Ergebnis dar.

Anhang B

Programmlistings

B.1 SOLVER.MAC

```
/*****************************************************************************/
/*                                                                           */
/* SOLVER - THE NEXT GENERATION                                              */
/*                                                                           */
/* 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  : 12.09.1994                                                 */
/* Time         : 11.50                                                      */
/*****************************************************************************/
/* Changes      : ||||| |||||                                                */
/*****************************************************************************/

SetVersion(
  /* KEY    = */ 'SOLVER,
  'MODULE      = "SOLVER",
  'DESCRIPTION = "Heuristic symbolic solver for systems of equations.",
  'AUTHORS     = "Eckhard Hennig, Ralf Sommer",
  'DATE        = "19.01.1994",
  'LASTCHANGE  = "12.09.1994",
  'TIME        = "11.50",
  'PLAN        = "Add break test and a-priori transforms"
)$


/*****************************************************************************/
/* 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 SolverImmedAssignments 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( SolverImmedAssign, TRUE, BOOLEAN )$


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

Define_Variable( SolverRepeatImmed, TRUE, BOOLEAN )$


/*******************************************************************************/
/* SolverSubstPowers 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( SolverSubstPowers, FALSE, BOOLEAN )$


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

```
/* SolverInconsParams 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 SolverInconsParams 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( SolverInconsParams, 'ASK, ANY_CHECK )$

Put(
  'SolverInconsParams,
  Lambda( [ x ],
    IF NOT Member( x, [ 'ASK, 'BREAK, 'IGNORE ] ) THEN
      ErrorHandler( "InvIncPar", x, 'Fatal )
  ),
  'VALUE_CHECK
)$


/*****************************************************************************/
/* SolverLinear 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( SolverLinear, TRUE, BOOLEAN )$


/*****************************************************************************/
/* If SolverRepeatLinsolve 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( SolverRepeatLinear, TRUE, BOOLEAN )$


/*****************************************************************************/
/* If SolverFindAllLinearVars 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( SolverFindAllLinearVars, TRUE, BOOLEAN )$


/*************************************************************************/
/* SolverAssumptions 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 SolverInconsParams.       */
/* Any numerical solution of the equations obtained by assigning numerical    */
/* values to symbolic parameters should be checked for consistency with all   */
/* expressions in SolverAssumptions.                                          */
/*************************************************************************/

Define_Variable( SolverAssumptions, [], LIST )$


/*************************************************************************/
/* SolverDelEq 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( SolverDelEq, 'MakeSquareLinearBlocks, ANY )$


/*************************************************************************/
/* SolverValuationStrategy 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( SolverValuationStrategy, 'MinVarPathsFirst, ANY_CHECK )$

Put(
  'SolverValuationStrategy,
  Lambda(
    [ x ],
    IF NOT FunctionP( x ) THEN
      ErrorHandler( "UndefStrat", x, 'Fatal )
  ),
  'VALUE_CHECK
)$


/*************************************************************************/
/* SolverDefaultValuation 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( SolverDefaultValuation, 10, FIXNUM )$


/****************************************************************************/
/* SolverMaxLenValOrder 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( SolverMaxLenValOrder, 5, FIXNUM )$


/****************************************************************************/
/* SolverTransforms 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( SolverTransforms, [], LIST )$


/****************************************************************************/
/* If SolverPostprocess 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( SolverPostprocess, TRUE, BOOLEAN )$


/****************************************************************************/
/* SolverBacksubst controls the output format of Solver. If SolverBacksubst */
/* 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( SolverBacksubst, TRUE, BOOLEAN )$


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

```
/* With SolverDispAllSols 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( SolverDispAllSols, FALSE, BOOLEAN )$


/*****************************************************************************/
/* If SolverRatSimpSols is TRUE then the Solver Postprocessor will           */
/* FullRatSimp the solutions before returning them.                          */
/*****************************************************************************/

Define_Variable( SolverRatSimpSols, TRUE, BOOLEAN )$


/*****************************************************************************/
/* If SolverDumpToFile 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( SolverDumpToFile, FALSE, BOOLEAN )$


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

Define_Variable( SolverDumpFile, "SOLVER.DMP", 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                   */

  Active
],

Mode_Declare(
  [
    Variables, UserVars, Parameters, Expressions, PowerSubst,
    Solutions
  ], LIST,
  Active, BOOLEAN
),

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

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

/* Do all necessary preprocessing */
Map(
  ":",
  [
    '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 : SolverImmedAssign,
  WHILE Active DO (
    Map(
      ":",
```

```
    [ 'Active, 'Solutions, 'Equations, 'Variables ],
    ImmediateAssignments( Solutions, Equations, Variables, Parameters )
  ),

  Active : Active AND SolverRepeatImmed,

  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 ],
    Num( FullRatSimp( LHS( Eq ) - RHS( Eq ) ) )
  ),
  Equations
),

/* Find and solve linear equations */

Active : SolverLinear,
WHILE Active DO (
  Map(
    ":",
    [ 'Active, 'Solutions, 'Equations, 'Variables ],
    LinearSolver( Solutions, Equations, Variables, Parameters )
  ),

  Active : Active AND SolverRepeatLinear,

  IF MsgLevel = 'DEBUG THEN
    Display(
      Equations, SolverParams, Variables, Parameters, Expressions,
      PowerSubst, UserVars, Solutions
    )
),

IF Empty( Equations ) OR Empty( Variables ) THEN
  RETURN( FALSE ),


/* Convert equations to list of expressions */
```

```
      Equations : Map(
        Lambda( [ Eq ], LHS( Eq ) - RHS( Eq ) ),
        Equations
      ),

      /* Apply valuation strategies to solve the nonlinear equations. */

      Map(
        ":",
        [ 'Active, 'Solutions, 'Equations, 'Variables ],
        ValuationSolver( Solutions, Equations, Variables, Parameters )
      )

    ), /* END BLOCK */


    /* Return the solutions and the unsolved equations */

    IF SolverPostprocess THEN
      RETURN( PostProcess( Solutions, UserVars, Expressions, PowerSubst ) )
    ELSE
      RETURN( Solutions )
  )
)$


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

TerminateSolver() := Error( ErrorMsg["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 occurences 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 occurences 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", Intersect( 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 SolverAssumptions.       */

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 SolverSubstPowers is TRUE then substitute powers by new symbols */

IF SolverSubstPowers 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 SolverSubstPowers */

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 SolverAssumptions.                                      */
/*************************************************************************/

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

  Mode_Declare(
    [ Eqs, Params ], 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 ), Params )
      )
```

```
      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 SolverAssumptions.     */
/*****************************************************************************/

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  */
  /* SolverInconsParams is set to 'BREAK.                          */

  IF Empty( ListOfVars( Expression ) ) OR ( SolverInconsParams = '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( SolverAssumptions ) WHILE NOT AssumptionExists DO
    IF
      FullRatSimp(
        RatSubst( 0, Expression, LHS( SolverAssumptions[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
      ( SolverInconsParams = 'ASK )
      AND
      ( AskCSign( 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 SolverAssumptions. So the user */
      /* has access to all assumptions made during the solution process   */
      /* and can check any numerical solutions for consistency with the   */
      /* assumptions.                                                     */
      SolverAssumptions : Endcons(  Expression = 0, SolverAssumptions )
```

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

    RETURN( TRUE )
  )
)$


/*****************************************************************************/
/* 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 (
        PrintMsg( 'DETAIL, SolverMsg["Assign"], TotalDisrep( Left = Right ) ),

        Solutions : AppendImmed( 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 (
        PrintMsg( 'DETAIL, SolverMsg["Assign"], TotalDisrep( Right = Left ) ),

        Solutions : AppendImmed( 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 ] )
```

```
  )
)$


/*****************************************************************************/
/* AppendImmed appends an immediate assignment to the list of solutions.     */
/*****************************************************************************/

AppendImmed( Equation, Solutions ) := (

  Mode_Declare(
    Equation, ANY,
    Solutions, LIST
  ),

  BLOCK(
    IF Assoc( LHS( Equation ), Solutions ) = FALSE THEN
      RETURN( Endcons( Equation, Solutions ) )
    ELSE
      RETURN( Solutions )
  )
)$


/*****************************************************************************/
/* 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 SolverFindAllLinearVars 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(
    Intersect( Variables, NewVars ),
    SetDifference( NewVars, Append( Variables, Parameters ) )
  ),
  LinearEqs : Equations,
```

```
  Equations : []
)
ELSE (
  ActiveVars : Intersect( Variables, NewVars ),
  LinearEqs : [],
  FOR i THRU Length( Equations ) DO
    IF NOT Empty( Intersect( 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( SolverDelEq, [ 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 : Intersect(
  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 : Linsolve( 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 SolverAssumptions 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 : Map(
    'Num,
    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 ],
            Num( 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, BCoeff[2] ) THEN
        RETURN( BCoeff[1] )
    ),
```

```
    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 SolverDumpToFile 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( SolverValuationStrategy, [ VarPaths, ValMatrix ] )
  ),

  Solved : FALSE,

  UNLESS Solved OR Empty( SolveOrder ) DO (

    SolveInfo : Pop( SolveOrder ),
    k   : Mode_Identity( FIXNUM, SolveInfo[1] ),
    Eq  : Equations[ k ],
    Var : Variables[ SolveInfo[2] ],

    PrintMsg(
      'SHORT,
```

```
  SolverMsg["TrySolveEq"], k, SolverMsg["ForVar"], Var
),
PrintMsg( 'SHORT, SolverMsg["Valuation"], SolveInfo[3] ),
PrintMsg( 'DETAIL, SolverMsg["EqIs"], Eq = 0 ),

Failed     : FALSE,
TryToSolve : TRUE,
CheckSol   : TRUE,
Transform  : CopyList( SolverTransforms ),

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 : Equations[k],
  Equations[k] : [],
  Equations : Delete( [], Equations ),

  /* Check solutions for consistency with remaining equations. */
  FOR i THRU Length( Solution ) DO (

    IF SolCheck[i] THEN (

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

      IF NOT ParamConsistency(
        FullRatSimp( Ev( Equations, Solution[i] ) ),
        Parameters,
        'CONTINUE
      ) THEN (
        PrintMsg( 'SHORT, SolverMsg["Contradict"], Solution[i] ),
        Solution[i] : 'INCONSISTENT_PATH
      )

    )
    ELSE (
      PrintMsg( 'DETAIL, SolverMsg["Dropped"], Solution[i] ),
      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 Length( Solution ) = 1 THEN (
    Solutions : Append( Solutions, Solution ),
    Equations : Map(
      'Num,
      FullRatSimp( Ev( Equations, Solution ) )
    )
  )
  ELSE

    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(
                ":",
                [ 'Stat, 'RSolutions, 'REqs, 'RVars ],
                ValuationSolver(
                  [ Sol ],
                  Map( 'Num, 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.                            */

      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 )$


/*****************************************************************************/
/* 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 : SolverDefaultValuation,

        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 occurences 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 : Sort(
      SolveOrder,
      Lambda( [ a, b ], a[3] < b[3] )
    ),

  /* Find all variables with only one path in 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,
        Sort(
          SolveOrder1,
          Lambda( [ a, b ], a[3] < b[3] )
        )
      ),

    /* Append additional candidates if necessary. */
    IF Length( SolveOrder ) < SolverMaxLenValOrder 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 ),

      SolveOrder1 : Sort(
        SolveOrder1,
        Lambda( [ a, b ], a[3] < b[3] )
      ),

      SolveOrder : Append( SolveOrder, SolveOrder1 )
    ),

    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 SolverBacksubst 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 ( Assoc( Var, SolSet ) # FALSE ) OR NOT Atom( Var ) THEN

    /* There may be errors when indeterminate expressions are */
    /* encountered.                                           */

    ErrCatch(
      IF SolverBacksubst 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 SolverRatSimpSols THEN
  UsrSolSet : ErrCatch( FullRatSimp( UsrSolSet ) ),

IF UsrSolSet = [] THEN
  PrintMsg( 'SHORT, SolverMsg["SolSetDrp"] )
ELSE
```

```
          UserSolutions : Endcons( UsrSolSet[1], UserSolutions )

      ), /* END FOR SolSet */

      IF SolverDumpToFile 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
    ),

    /* ListP = TRUE indicates an additional recursion level */
    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 <SolverDumpFile>.                                                 */
/*****************************************************************************/

DumpToFile( Sols, Eqs, Vars ) := (

  Mode_Declare(
    [ Sols, Eqs, Vars ], LIST
  ),

  BLOCK(
    [ Solutions, Equations, Variables ],

    PrintMsg( 'SHORT, SolverMsg["Dump"], SolverDumpFile ),

    Apply(
      'StringOut,
      [
        SolverDumpFile,
        'Solutions = Sols,
        'Equations = Eqs,
        'Variables = Vars
      ]
    )
  )
)$
```

Estragon Manchmal frag ich mich, ob es nicht besser wäre, auseinanderzugehen.

Wladimir Du würdest nicht weit kommen.

Estragon Das wäre wirklich sehr schade ... Nicht wahr, Didi, das wäre doch sehr schade? ... Wenn man an die Schönheit des Weges denkt ... Und an die Güte der Weggefährten ... Nicht wahr, Didi?