Macsyma

User's Guide

Second Edition

Macsyma User's Guide

This document corresponds to Macsyma 2.0, Macsyma 419 and successors.

Printed in the USA.

Macsyma, Inc.
telephone: 800–macsyma 800–622–7962
email: info-macsyma@macsyma.com
service@macsyma.com
Internet: URL http://www.macsyma.com

Printed in the USA.

Printing year and number: 96 3

Contents

List of Tables

Preface

Macsyma is a general purpose symbolic-numerical-graphical mathematics software product. You can use it to solve simple problems specified by one-line commands (such as finding the indefinite integral of a function), or to perform very complicated computations by means of a large Macsyma program. This document provides explanations and examples of tools for doing algebra, trigonometry, calculus, numerical analysis, and graphics. In addition to introductory information about these topics, this guide provides extended examples showing you how to use Macsyma to solve more complex problems.

If you have never used Macsyma before, you might want to start by reading *Your First Session With Macsyma*, a brief tutorial which introduces you to Macsyma's user interface and on-line help facilities. The on-line help system, includes a mathematical topic browser, hypertext topic descriptions, command templates, an interactive primer, and executable examples and demonstrations. You will also see how to enter commands to Macsyma, and how to produce a document which combines mathematical expressions and formatted text. You can run an on-line tutorial for Macsyma using the **primer** command (see page 12). Users who are new to Macsyma should read the first four chapters of this *User's Guide* carefully before going on to other chapters of interest.

Even if you have prior experience with Macsyma, you might look over the material presented in the first four chapters before continuing to other chapters that interest you. Both new and experienced users might find it helpful to look over Chapter 1, which discusses symbolic and numeric computation and how they complement each other. Starting with Chapter 5, each chapter builds on the concepts presented in the earlier chapters.

This document assumes you are already familiar with basic mathematical terminology and notation. Some additional notation conventions and terminology appear in this document to make it easier to read and understand. The notation conventions followed in this *User's Guide* are summarized in Table 1.

Although this document introduces many commands and option variables, its scope allows only a limited introduction to Macsyma's capabilities. To find out more, consult Macsyma's *Mathematics and System Reference Manual*, where you will find a comprehensive description of the mathematical functionality of Macsyma, including algebraic and calculus operations, matrix and vector algebra, vector and tensor analysis, plotting, and graphics. In addition, the *Graphics and User Interface Reference Manual* contains a complete description of the non-mathematical aspects of Macsyma, including the user interface, programming in Macsyma, file management, and interfacing to other languages.

In general, you should know that Macsyma commands are either **functions** or **special forms**. Both accept zero or more arguments, perform some computation, and return a result. For example, the function **sqrt** with an argument of 4 returns the result 2. The arguments to a function are each evaluated in order from left to right. A special form is similar to a function, except that the evaluation of some of the arguments to a special form may be delayed or may not occur at all.

By resetting **option variables** to new values, a user can change the environment in which Macsyma evaluates command lines. For example, the function **plot3d** can display different projections of the same three-dimensional plot, depending on the value of the option variable **viewpt**.

Example	Meaning
ratsimp, **integrate**	Commands and other Macsyma objects as they appear in the text.
RETURN, SPACE, C-F	Keyboard keys.
`fib(2);`	Characters entered on the command line, or Macsyma code in a file. Macsyma is not case-sensitive unless you precede an input character with a backslash (\).
diff(*exp, var, n*)	Description of the format for the **diff** command, where *exp*, *var*, and *n* describe the expected arguments.
`Division by 0.`	A system message or error.
`        3` `(x + 1)`	Results returned on the display line. Current releases of Macsyma display output in lower case by default; users can alter the display case.
testing.com	A file or directory name.
/* `comment` */	Explanatory comments in Macsyma examples. (In versions of Macsyma running under VMS, double quotes delimit interactive comments and /* */ delimit batch comments.)

Table 1: Notation conventions used in this book

Chapter 1

Introduction to Macsyma

Macsyma is an interactive symbolic-numerical-graphical mathematics software system that helps you solve complex mathematical problems. Macsyma can also help you visualize scientific data or prepare high-quality technical reports. This User's Guide provides explanations and examples of the basic methods for solving many kinds of problems in algebra, trigonometry, calculus, numerical analysis, and graphics.

Macsyma offers a wide range of capabilities, including differentiating and integrating expressions, factoring polynomials, plotting expressions, solving equations, manipulating matrices, and computing Taylor series. Macsyma also provides a programming environment in which you can define mathematical procedures tailored to your own needs.

Section 1.1 contrasts the symbolic-numerical approach to computation provided by Macsyma with the strictly numeric approach provided by computer languages such as FORTRAN. Read this section to learn about the advantages of combining symbolic and numerical computation.

Section 1.2 presents an extended example that shows how Macsyma can perform calculations that would be tedious and time-consuming when done by hand. For example, Macsyma can expand, factor, and simplify long expressions, and manipulate large matrices, allowing you to eliminate much of error-prone manual calculation.

1.1 Symbolic and Numerical Computation

Computers have traditionally been used to solve scientific problems that could be expressed in terms of numbers. FORTRAN, for example, assists scientists and mathematicians in dealing with numeric problems. When a problem can easily be expressed in terms of calculations with numbers, this approach to problem solving works well. On the other hand, some problems can be expressed best in symbolic terms or perhaps can only be expressed that way.

Macsyma can work with numbers, symbols, polynomial expressions, matrics and equations. Macsyma can return results in either numeric or symbolic form.

Unlike Macsyma, numerical systems must operate using floating-point approximations, whose precision is limited by the computer hardware. By carrying out computations in symbolic form, Macsyma can work with exact quantities rather than approximations. When you choose to convert a symbolic result to a floating-point number, you can set the precision yourself.

1.1.1 Exact and Floating Point Numbers

In scientific computation, numbers are frequently represented approximately as floating point numbers. For some purposes, it is necessary to use exact representations of numbers, or to use symbolic variables to stand for a class of possible numerical values. For example:

```
(c1) .66667;
(d1)                0.66667
(c2) 2/3;
                       2
(d2)                   -
                       3
(c3) .66667*3;
(d3)                 2.0001
(c4) (2/3)*3;
(d4)                   2
```

In many computations, radicals need to be represented exactly.

```
(c5) 1.414;
(d5)                   1.414
(c6) sqrt(2);
```

(d6) $\sqrt{2}$

```
(c7) 1.414^2;
(d7)                   1.9994
(c8) sqrt(2)^2;
(d8)                      2
```

Trigonometric identities often require that numbers be represented exactly.

```
(c9) %pi;
```

(d9) π

```
(c10) sfloat(%pi);
(d10)                  3.14159
(c11) dfloat(%pi);
(d11)                  3.14159265358978d0
(c12) bfloat(%pi), bfprecision:40;
(d12)                  3.141592653589793238462643383279502884197b0
(c13) sin(sfloat(%pi));
(d13)                  6.27833e-7
(c14) sin(%pi);
(d14)                   0
```

1.1.2 Exact and Floating Point Arithmetic

The following example illustrates how numeric and symbolic computation differ in a simple arithmetic problem.

Addition and subtraction are associative operations. To compute `a - b + c` you can group the expression either as `(a - b) + c` or as `a + (-b + c)`. For some floating-point numbers, however, the associative property will not hold. Consider this example of FORTRAN code:

```
program main
a = 12345678.0
b = 12345679.0
c = 12.0/5.0
temp1 = (a - b) + c
temp2 = a + (-b + c)
print *, temp1, temp2
end
```

Clearly, both `temp1` and `temp2` should be `1.4`, but on most systems FORTRAN returns unequal results for `temp1` and `temp2`. Typical results are `1.4` for `temp1` and `1.0` for `temp2`, showing that associativity does not hold for all floating-point calculations.

In Macsyma you can preserve precision by expressing numbers as ratios of integers. Even if you are not yet familiar with Macsyma notation, you can see that Macsyma returns the correct answer to this problem. Set a, b, and c to the same values as in the FORTRAN example. Note that in Macsyma, "**:**" is the assignment operator.

```
(c1) a:12345678;
(d1)                12345678
(c2) b:12345679;
(d2)                12345679
```

Notice that Macsyma stores the value of c as a ratio of integers.

```
(c3) c:12/5;
                       12
(d3)                   --
                        5
```

The result for *temp1* is a ratio of integers.

```
(c4) temp1:(a - b) + c;
                        7
(d4)                    -
                        5
```

The result for *temp2* is the same as *temp1*. Associativity is preserved.

```
(c5) temp2:a + (-b + c);
```

```
                              7
(d5)                          -
                              5
```

You can convert either result to a floating-point number. The (d5) below refers to the result returned on line (d5).

```
(c6) sfloat(d5);
(d6)                         1.4
```

To avoid introducing floating-point errors into your calculations, Macsyma generally does not return a floating-point result unless you specifically request it. Experienced Macsyma users learn to work with this feature to achieve exact results in their calculations. Consider the following Macsyma example working with square roots.

Macsyma simplifies square roots of integers without converting them to floating-point.

```
(c7) sqrt(32);
(d7)                      4 sqrt(2)
(c8) sqrt(2);
(d8)                       sqrt(2)
```

Since (c9) simplifies to `sqrt(32*2)` = `sqrt(64)`, an exact integer result can be returned.

```
(c9) sqrt(32)*sqrt(2);
(d9)                          8
```

Computing the square root of a floating-point number results in an approximation.

```
(c10) sqrt(32.0);
(d10)                     5.656854
(c11) sqrt(2.0);
(d11)                     1.4142135
```

Macsyma does not return `8` here because `5.656854*1.4142135` is not the same as the square root of `64`.

```
(c12) sqrt(2.0)*sqrt(32.0);
(d12)                     7.9999995
```

The square root of a rational number is maintained as an exact quantity.

```
(c13) sqrt(21555/44);
                        3 sqrt(2395)
(d13)                   ------------
                         2 sqrt(11)
```

1.1.3 Exact and Floating Point Solutions of Algebraic Equations

Symbolic mathematics combines the exactness of exact arithmetic with the generality obtained by representing an entire class of possible numbers by a symbolic variable. It also helps you to form insights through inspection of symbolic solutions, which are harder to gain from numerical solutions. The most elementary use of symbolic mathematics employs a symbolic variable to stand for an arbitrary real number (or a class of possible real numbers). For example, the symbolic case of a linear equation generalizes any particular numerical case, in the example below.

```
(c1) eqf: 3.*x+5.=7.;
(d1)                3.0x + 5.0 = 7.0
(c2) linsolve(eqf,x), keepfloat:true;
(d2)                  [x = 0.66667]
(c3) eqi: 3*x+5=7;
(d3)                  3x = 5 = 7
(c4) linsolve(eqi,x);
                          2
(d4)                 [x = - ]
                          3
(c5) eqs: a*x+b = c;
(d5)                  ax + b = c
(c6) linsolve(eqs,x);
                         c - b
(d6)                [x = ----- ]
                           a
```

1.1.4 Exact Solution of a Differential Equation

Symbolic computation can make computations involving the constant π more accurate. Consider this differential equation, taken from an example in [Va]:

$$\begin{aligned} y'(x) &= (2/\pi)xy(y - x\pi), \qquad 0 \le x \le 10 \\ y(0) &= y_0 \end{aligned}$$

In a numerical solution, if $y_0 < \pi$, the solution tends to zero as $x \to \infty$, and if $y_0 > \pi$, the solution goes to infinity. If the input data specified the value of π for y_0, then recalling that $\pi = 3.14159\ldots$ you can see that the five-digit rounded approximation 3.1416 leads to a solution that goes to infinity, while a five-digit truncated approximation 3.1415 gives a solution that goes to zero. In Chapter 6, 97, Macsyma obtains the following exact solution to this problem:

```
                  %pi y0
y = - ------------------------
            2                  2
           x                  x
      (%e    - 1) y0 - %pi %e
```

This is the exact result returned by Macsyma in its two-dimensional format. The `%pi` denotes the constant π, and `%e` is the base of the natural logarithms.

1.2 Example Problem: Modeling a Robot Arm

The best way to see Macsyma solving some simple application problems is to run some of the demonstrations which come with Macsyma. After starting Macsyma type:

`demo(begin);`	a short collection of simple computations
`demo(general);`	a longer collection of simple computations
`demo(ballistics);`	a solution for the trajectory of a cannon ball
`demo(oscillator);`	a solution for the motion of a harmonic oscillator

Macsyma is an excellent tool for verifying published computational results. The rest of this section consists of a Macsyma session which verifies published results concerning the dynamic behavior of a robot arm.

To describe the dynamic behavior of a robot arm, [Le] derives a set of differential equations of motion for a manipulator with n degrees of freedom. The derivation of a dynamic model based on the Euler–Lagrange method is simple and systematic. For $n = 2$, however, the Euler–Lagrange formulation involves about 3200 multiplications and 2500 additions, making the calculations both time-consuming and error-prone when done manually.

An experienced Macsyma user can reproduce the results published in [Le] for $n = 2$ in a single afternoon. The session below illustrates how Macsyma can reproduce these calculations. The numbered equations in the session correspond to those in [Le].

The first two commands define a general rule to convert all occurrences of $\sin^2 x$ to $1 - \cos^2 x$.

```
(c1) matchdeclare(any,true)$
(c2) tellsimp(sin(any)^2, 1 - cos(any)^2)$
```

Equation 1. Define a 4×4 homogeneous transformation matrix *tra_mat* to give the spatial relationship between adjacent links.

```
(c3) tra_mat(j):=(j1:j + 1,
          matrix([cos(th[j1]), -cos(al[j1])*sin(th[j1]),
              sin(al[j1])*sin(th[j1]), aa[j1]*cos(th[j1])],
             [sin(th[j1]), cos(al[j1])*cos(th[j1]),
              -sin(al[j1])*cos(th[j1]), aa[j1]*sin(th[j1])],
             [0, sin(al[j1]), cos(al[j1]), d[j1]],
             [0, 0, 0, 1]))$
(c4) a(j, i):=if j < i then tra_mat(j).a(j + 1, i) else ident(4)$
```

Equation 13. Define matrix U to describe the rate of change of all the points r_i on link i relative to the base coordinate frame as q_j (the generalized coordinate) changes.

```
(c5) q:matrix([0,-1, 0, 0],
              [1, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0])$
(c6) u(i, j):=if j <= i then a(0, j - 1).q.a(j - 1, i) else 4*ident(4)$
```

Define the inertia tensor matrix.

```
(c7) inertia(j):=matrix([mass[j]*aa[j]^2/3, 0, 0, -mass[j]*aa[j]/2],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [-mass[j]*aa[j]/2, 0, 0, mass[j]])$
```

Equation 15. Define matrix DU to describe the interaction effects between joints j and k.

```
(c8) du(i, j, k):=if i >= k and k >= j
          then a(0, j - 1).q.a(j - 1, k - 1).q.a(k - 1, i)
              else (if i >= j and j >= k
                then a(0, k - 1).q.a(k - 1, j - 1).q.a(j - 1, i)
                else zeromatrix(4, 4))$
```

The next set of commands defines three matrices to describe the gravity-related terms, the coriolis and centrifugal terms, and the acceleration-related terms for the robot arm respectively.

Equation 31. Define the gravity-related terms in matrix G.

```
(c9) g(i):=trigreduce(sum(mass[j]*matrix([0,-gr,0,0])
               .u(j,i).matrix([-aa[j]/2,[0],[0],[1]]),j,i,n))$
```

Equation 32. Define the coriolis and centrifugal terms in matrix H.

```
(c10) hh(i,k,m):=sum(matrix_trace(du(j,k,m).inertia(j).transpose(u(j,i))),j,max(i,k,m),n)$
(c11) h(i):=(sum(sum(hh(i,k,m)*thdot[k]*thdot[m],m,1,n),k,1,n),expand(%%))$
```

Equation 33. Define the acceleration-related terms in matrix D.

```
(c12) d(i,k) := expand(sum(matrix_trace(u(j,k).inertia(j).transpose(u(j,i))),j,max(i,k),n))$
```

Now that the matrices have been defined, the following commands calculate the results for the motion of a robot arm with 2 degrees of freedom.

```
(c13) (n:2,al[1]:al[2]:d[1]:d[2]:0,aa[1]:aa[2]:1)$
(c14) matrix([d(1,1), d(1,2)], [d(2,1), d(2,2)]);
               [                              2          2   ]
               [                     4 mass  l    mass  l    ]
               [                2         2          1       ]
               [ mass  cos(th ) l  + ---------- + --------   ]
               [     2       2           3           3       ]
(d14)  col 1 = [                                             ]
               [                        2          2         ]
               [       mass  cos(th ) l    mass  l           ]
               [           2       2           2             ]
               [       ----------------- + --------          ]
               [               2              3              ]
```

```
                  [                2         2 ]
                  [ mass  cos(th ) l    mass  l  ]
                  [     2       2          2   ]
                  [ ---------------- + -------- ]
                  [         2             3    ]
        col 2 =   [                            ]
                  [                2           ]
                  [          mass  l           ]
                  [             2              ]
                  [          --------          ]
                  [             3              ]
(c15) matrix([h(1)], [h(2)]);
      [             2           2                                  ]
      [   mass  thdot  sin(th ) l                                  ]
      [       2      2       2                                   2 ]
      [ - ----------------------- - thdot  mass  thdot  sin(th ) l  ]
      [             2                    1     2      2       2    ]
(d15) [                                                            ]
      [               2                 2                          ]
      [          thdot  mass  sin(th ) l                           ]
      [               1     2       2                              ]
      [          -----------------------                           ]
      [                     2                                      ]
(c16) matrix(g(1), g(2));
             mass  cos(th  + th ) gr l
                 2       2     1
(d16) matrix([- ------------------------- - mass  cos(th ) gr l
                         2                    2       1
            mass  cos(th ) gr l      mass  cos(th  + th ) gr l
                1       1                2       2     1
          - -------------------], [- ------------------------])
                      2                          2
```

The commands used in this example are explained more fully in subsequent chapters. Look in the index to locate information about specific commands or in the *Macsyma Reference Manual* .

The references cited in this chapter appear on page 263. Consult [Ru] or [Wa] for more information about symbolic algebra, [Va] for examples of pitfalls in numerical computation, and [Go] for using Macsyma to analyze robotic mechanisms. For an historic perspective of applications using Macsyma up until 1986, contact Macsyma Inc. for a copy of the *Bibliography of Publications Referencing Macsyma*. With so many applications written in Macsyma, the company no longer publishes comprehensive listings since that time.

Chapter 2

Getting Started

This chapter describes Macsyma's user interface and explains how to communicate with Macsyma. It describes the basic things you need to know to get started: how to enter and exit Macsyma: how to type in command lines, how to refer to the results Macsyma returns in subsequent calculations, how to get help, and how to correct errors. If you have read Chapter 1, you have already seen some sample interactions with Macsyma. Since the user interface differs slightly in different versions of Macsyma, the best information for getting started in Macsyma is found in the document called *Your First Session With Macsyma*, written for your particular Macsyma release.

2.1 Entering and Exiting Macsyma

If you have a PC window system and you have installed a Macsyma icon, enter Macsyma by clicking on the icon with your mouse. Otherwise, enter Macsyma by typing `macsyma` at the command processor or your operating system, exec, or monitor level. Depending on which version of Macsyma you have, there are other ways of entering Macsyma. Refer to the Release Notes for your version for more information.

Upon entering Macsyma, the system either prints a message telling you that Macsyma is loaded, or it prints a herald that displays the Macsyma version number and other information pertaining to your site. For example, in PC Macsyma 2.0 and its successors, the message appears something like Figure 2.1.

```
C:\MACSYMA2\system\init.lsp being loaded.
Batching the file C:\MACSYMA2\user\mac-init.mac

Batchload done.
------------------------------------------------------------
(c1)
```

Figure 2.1: The Initial Macsyma 2.0 Message

Figure 2.2 shows a typical herald displayed by Macsyma 420 and its successors.

The number 420.0 in Figure 2.2 identifies the Macsyma version. This number changes for each new release. The input prompt, (c1), marks the first command line or C-LINE. Typing commands on the C-LINE is explained in more detail in Section 2.2, page 10.

To exit Macsyma, use the **quit** command, or, in a windows environment, select another window.

```
This is Macsyma 420.0 for RS6000 (AIX 3.2.X) computers.
Copyright (c) 1982 - 1996 Macsyma Inc.  All rights reserved.
Portions copyright (c) 1982 Massachusetts Institute of Technology.
All rights reserved.
Type ``DESCRIBE(TRADE_SECRET);'' to see important legal notices.
Type ``HELP();'' for more information.
/tmp_mnt/net/hamilton/devo/rs6000/mac420/system/init.lsp being loaded

(c1)
```

Figure 2.2: An Initial Macsyma 420 Herald

The **quit** command allows you to leave Macsyma, terminating the Macsyma session. You should use this command only when you do not intend to continue this session. If you plan to return, you should simply select another window. To use this command type `quit();`. Refer to **quit** in the *Macsyma Reference Manual* for more information.

You can save a transcript of what goes on during your Macsyma session. For complete instructions see Section 9.4.1, page 160.

2.2 Typing in Command Lines

As shown in the two Figures, a C-LINE prompt appears when you invoke Macsyma. When you type a command or expression on the C-LINE, Macsyma generally answers by displaying a result on a corresponding display line, called the D-LINE. Intermediate steps in large calculations are shown on E-LINES. For an example, see page 27.

In Macsyma, C-LINEs are labeled consecutively in the form (ci), where i is the number of the current command line. Similarly, D-LINEs are labeled (di) where i is the number of the corresponding C-LINE. These labels provide an easy way of referring to previously typed commands and output expressions without retyping them.

Consider the following expression:[1]

```
(c1) (x + 1)^3;
                                3
(d1)                     (x + 1)
```

The C-LINE above ends with a semicolon (`;`). Ending the line with a semicolon tells Macsyma to display the result of the command on the screen.[2] In this example, the expression $(x + 1)^3$ is associated with the label (d1). Macsyma then prints a new C-LINE prompt, (c2) in this example, for your next command.

You terminate all C-LINEs by typing either a semicolon or a dollar sign (`$`), then pressing RETURN. Ending the command line with a dollar sign tells Macsyma not to display the result of the command. Semicolons usually end the command lines in the examples shown here, so that you can see exactly what is going on during a calculation. In practice, you will often use the dollar sign to finish lines where you do not need to see Macsyma's result (for example, when assigning values to variables).

[1] The examples in each section of this document are labeled consecutively, starting with (c1). If you are following along in Macsyma yourself, you might want to use the command `kill(labels)$` or better `initialize_macsyma()$` at the beginning of each section to reset your own c-label to (c1).

[2] In Macsyma 2.0 and its successors, the semicolon may be omitted, and the return key may be used to tell Macsyma to display the result.

After entering an expression on a C-LINE, you can manipulate it with a variety of commands. The percent sign (%) is a system variable whose value is the result of the most recent D-LINE. More generally, you can use **%th**(i) to refer to the ith previous computation. Thus, `%th(1)` is equivalent to %.

To add 5 to the most recent result, the expression shown in (d1).

```
(c2) % + 5;
                                   3
(d2)                        (x + 1)  + 5
```

Multiply the result shown on (d1) by z.

```
(c3) %th(2) * z;
                                       3
(d3)                            (x + 1)  z
```

You can also refer to any D-LINE or C-LINE explicitly, as shown below.

Multiply the expression on line (d1) by $(x + 1)$.

```
(c4) d1 * (x + 1);
                                   4
(d4)                        (x + 1)
```

Even if you suppress the display of the D-LINE by ending the command line with a dollar sign, you can reference it either explicitly or with %.

```
(c5) -y + d2$
(c6) z + %;
                                           3
(d6)                        z - y + (x + 1)  + 5
```

The previous examples show that % stands for the most recent D-LINE, and that you can use an explicit label to stand for any other command or display line. These facilities are useful to you only as long as you can see the desired expression on the screen or remember its associated label. Macsyma also provides a way for you to name an expression for later use by assigning it to a variable. For more information, see Section 3.1.3, page 20.

The system provides many commands to edit the command line if you make a mistake while typing it. Refer to your *Release Notes* and *Your First Session with Macsyma* for the editing techniques for your version of Macsyma.

Information intended for human readers of Macsyma code or output is contained in comments. Such information will not be evaluated by Macsyma. Comments can be included in your Macsyma session by enclosing them in `/*` and `*/`. A comment can be of any length.

```
/* This is a comment. */
(c7) x /* this is a comment too inside an
        expression */ : y /* */ + 2;
(d7)  y + 2
(c8) z: x^2 /* and this /* is a nested */ comment */;
       2
(d8)   x
```

2.3 Getting On-line Help

This section describes the facilities you can use to get on-line help during a Macsyma session. The Macsyma on-line help system is designed so that the user can obtain nearly all help on-line, rarely referring to the hardcopy documentation. On-line help is available through keyboard commands and, in versions of Macsyma with modern window interfaces, through menus.

The Macsyma command **help** summarizes the main facilities for obtaining on-line help. To use this command type `help();` or select the `[Help]` menu item.

The main on-line help facilities are:

- The Interactive Primer
- The Mathematics Topic Browser
- Hypertext topic descriptions
- Executable examples and demonstrations
- Function templates
- The **apropos** command
- Tips

2.3.1 The Interactive Primer

The Macsyma **primer** presents an interactive tutorial on the fundamentals of Macsyma, including an introduction to syntax, assignment, defining functions, and the simplification commands. Type `primer();` or select the menu item `[Help]|[Primer]` to run the primer.

The **primer** function accepts an optional argument of *scriptname*, which allows the user to indicate the topic in which they are interested. For example, `primer(simplification);` enters directly at the `simplification` section of the script. The possible values for *scriptname* are: `intro`, `simplification`, `scratchpad`, `syntax`, and `assignment`.

Users who are familiar with some mathematics software may want to skip the Interactive Primer, and start with the Mathematics Topic Browser.

2.3.2 The Mathematics Topic Browser

Sometimes you know what mathematical operation you wish to perform, but you do not know the name of the appropriate command, or whether Macsyma has the capability which you seek. In this case you can explore the available commands in your area of interest by using the Mathematics Topic Browser. In some versions of Macsyma, the top level topics appear along the top of the Macsyma Front End Window. In Macsyma 2.0 and its successors, the top level math topics are found in the `[MathHelp!]` menu. Each mathematics topic menu has a submenu of more narrowly defined topics to choose from. After you select a submenu, a dialog box shows you a list of the most common commands available in this topic area. You can then choose to `[Describe]` the command, to run an executable `[Example]`, or to open a function `[Template]`.

In most of the topics in the Mathematics Topic Browser, the last choice is `[Packages]`, which provides a list of the external packages in Macsyma in the given topical area. When you select `[Packages]`, the buttons in the resulting window offer a different set of choices for help information: `[Describe]`, `[Usage]` (a more in-depth description), `[Demo]`, and `[Load]`.

If you do not have a windowed version of Macsyma, type `options();` to display a hierarchical menu of topics. When you choose the topic you want to find out more about, a new menu appears, listing subtopics as menu items. Choose a subtopic to further refine your choice.

2.3.3 Hypertext Topic Descriptions

After choosing a command in the topic browser, click on the `[Describe]` button to enter the database of hypertext topic descriptions. Alternatively, if you know the name of the command or topic you need to explore, but you are unsure of the calling syntax of the command, or exactly how it is used, you can look up the command name by using the menu choice `[Help]|[Index]` or the Macsyma command **describe**(*topic*);

Recent releases of Macsyma contain nearly 2000 topic descriptions with hypertext cross references. If you click your mouse on one of the active words in the text, you will see a description of the topic named by that word.

The basic description of any topic can be found by using the **describe** command, and more in-depth information can be accessed by using the **usage** command. The **describe** command takes a single argument (the name of a topic) and displays a description of the topic, which can be a command, an option, or a concept.

If **describe** fails to find information about a feature that you think should be present, try selecting **usage** to find a function located in an external library. Also, try using **apropos** (see Section 2.3.6) to generate other candidates that **describe** might know about.

The **usage** command prints useful information about its single argument (the name of a topic), particularly if the topic is an external library file. For example, type `usage(fourier);` to find out about the **fourier** function.

2.3.4 Executable Examples and Demonstrations

We use the terms **example** and **demo** for executable sample programs which illustrate two slightly different types of topics in Macsyma.

- An **example** is a short program illustrating the basic use of one Macsyma command.
- A **demo** is a longer program illustrating either an application of Macsyma or the use of some external package or major facility in Macsyma.

Recent releases of Macsyma include about 600 examples and 200 demonstrations. In windowed versions of Macsyma, all of these programs are accessible through the menu system. To see a list of demos in your Macsyma, click your mouse on the menu item `[Help] | [Demos]` or `[Help] | [Demo a Feature]` (depending on the version of Macsyma you are using). You can start a Demonstration in several ways:

- Click on the name of a Demonstration in the `[Help]|[Demos]` or `[Help]|[Demo-a-Feature]` list.
- Click on the name of a Demonstration when it appears in the hypertext description of some topic.
- Type `demo`(*topic*); .

The **demo** command initiates an executable demonstration. The demonstration pauses after each command line until you press the RETURN key or the SPACE bar. To interrupt the **demo**, press any other key. An interrupted **demo** can be continued by typing `batcon();`. When the **demo** is finished, it prints `done`.

The **example** command takes the name of a command as an argument and presents an example of that command in a manner similar to **demo**, pausing after each command line until you press the RETURN key or the SPACE bar. For example, type `example(linsolve);` to display an example of the **linsolve** function.

You can create your own examples by placing a file called *topic*`.example` in the example directory in your Macsyma.

2.3.5 Function Templates

A Function Template is a dialog box which specifies the arguments to a Macsyma command, indicating which ones are optional. It provides blank spaces for you to fill in the arguments to define your own command. You can obtain a Function-Template for nearly all of the functions and special forms, (but not option variables) which appear in the Mathematics Topic Browser by pressing the button labeled `[Template]` in the Mathematics Topic Browser. Recent releases of Macsyma contain function templates for 500 Macsyma commands.

2.3.6 The apropos Command

Sometimes you know part of the name of the command you want to use. The **apropos** command may then be of assistance. The **apropos** command takes a symbol or a string as its argument and looks through all the Macsyma symbols with **describe** entries for ones containing that symbol or string. For example, `apropos(exp);` returns a long list of all the flags and functions that have `exp` as part of their names, such as **exp**, **expand**, **ratexpand**, **trigexpand**, and **exponentialize**. If you can only remember part of the name of a command, you can use **apropos** to find the complete name.

For Unix-based systems only, Macsyma provides an online "manual" page for Macsyma through the shell command **man**. To see it, type `man Macsyma` at the Unix prompt.

2.3.7 Tips

The on-line help system includes Tips – a quick and simple guide to Macsyma commands and examples of their behavior. The Tips interface consists of a topics browser allowing you to choose the area of interest. Tips offers a one-line description of a sample computation. Tips adds an example that consists of a problem statement, code to apply, and the output of applying that code. You can browse through a number of different problems until you see a Tip similiar to your problem. Then you try the code to produce the effect you want.

Since the user interface differs slightly in different versions, you can find the most up-to-date information for Tips in Macsyma in the documents for your particular Macsyma release.

Chapter 3

Creating Expressions

You can perform calculations in Macsyma by acting on expressions. This chapter describes the components of a Macsyma expression, which can include numbers, variables, operators, and constants. Using expressions as building blocks, you will learn how to assign values to variables, create equations, and define your own functions.

Starting with the next chapter, you will learn about the many ways that you can manipulate expressions, including expanding, simplifying, and factoring.

3.1 What is a Macsyma Expression?

The basic unit of information in Macsyma is the **expression**. An expression is made up of a combination of **operators**, **numbers**, **variables**, and **constants**. Sections 3.1.1 through 3.1.4 describe how to use these components to form expressions.

3.1.1 Operators

Macsyma uses familiar symbols to perform operations. Table 3.1 summarizes these operators in order of priority, from lowest to highest.

Operator	*Description*
+	addition
−	subtraction
*	multiplication
/	division
−	negation
^	exponentiation
.	non-commutative multiplication
^^	non-commutative exponentiation
!	factorial
!!	double factorial

Table 3.1: Mathematical Operators

Macsyma also provides some other operators that are not discussed in this section. The “:” operator (page 20) assigns values to variables, the “=” operator (page 24) creates equations, and the “:=” operator (page 25) defines functions.

Macsyma performs operations of equal priority from left to right. You can use parentheses to change the order of evaluation. Note that, as illustrated in the examples below, the application of a function has the highest priority.

Binding power is a measure of the amount of precedence an operator has over the operator tokens near it. For example, the binding power of the “*” operator is greater than that of the “+” operator, so `a+b*c` means `a+(b*c)` rather than `(a+b)*c`.

The application (in the case of the next example, **sin**) has a higher priority than “^”

```
(c1) sin(a*x^y/z!)^2;
                              y
                        2  a x
(d1)                 sin (-----)
                            z!
```

To square the argument of a function, you need an extra pair of parentheses.

```
(c2) sin((a*x^y/z!)^2);
                           2  2 y
                          a  x
(d2)                 sin(-------)
                             2
                          z!
```

Notice that “*” is commutative but “.” is not.

```
(c3) a*b - b*a + c.d - d.c;
(d3)                 c . d - d . c
```

The “^” operator distributes over the multiplication operator “*” .

```
(c4) b^2 * b^3;
                          5
(d4)                     b
```

The “^^” operator distributes over the non-commutative multiplication operator “.” .

```
(c5) b^^2 . b^^3;
                        <5>
(d5)                    b
```

Notice that “^” does not distribute over “.” and that “^^” does not distribute over “*” .

```
(c6) b^2 . b^3;
                          2    3
(d6)                     b  . b
(c7) b^^2 * b^^3;
                            <2>  <3>
(d7)                        b    b
```

The factorial operator "!" is defined as $n! = n(n-1)(n-2)\ldots 1$.

```
(c9) 8!;
(d9)                     40320
```

The double factorial operator "!!" is defined as $n!! = n(n-2)(n-4)\ldots 1$ or $n!! = n(n-2)(n-4)\ldots 2$ depending on whether n is odd or even.

```
(c8) 8!!;
(d8)                      384
```

3.1.2 Numbers

Macsyma knows about several kinds of **numbers**:

- **Integers** consist of a string of digits not containing a decimal point. For example, `15934`. Integers can grow very large, since their size is bounded only by the total virtual address space accessible to Macsyma.
- **Rational numbers** are represented as an exact ratio of two integers. For example, `3/2`. Macsyma can represent any rational number, subject only to the memory limitations of your machine.
- **Floats** and **bigfloats** are floating-point numbers. They consist of a string of digits containing a decimal point, and are optionally followed by an `e`, a `d`, or a `b` and an integer exponent. Examples of floats are `459.3`, `83.3495e6`, and `79.46d5`. Examples of bigfloats are `83.3495b6` and `3957204b15`.

  The size and precision of a floating point number is limited by each machine's hardware, so calculations involving them can be compromised by round-off errors. Bigfloats can have any number of digits you specify, but because of the performance penalty in using them you should use bigfloats only when necessary.
- **Complex numbers** are written with the imaginary unit i, which in Macsyma is written as `%i`. For example, $4i$ is written as `4*%i`. Information about predefined constants, including **%i**, appears in Section 3.1.4, page 23. Information about using complex numbers appears on page 19. See also **%i** and **constant**.
- Negative numbers are any kind of number beginning with a minus sign. For example, `-4`, `-17.4`, `-3957204b15`.

Macsyma does not limit the number of digits in an integer or rational number, but the range of nonzero floating-point numbers depends on your computer. For example, on Intel PCs a floating point number must have an absolute value between `1.2e-38` and `3.4e38` and is limited to approximately eight-digit precision. To determine the range of floating point numbers on your machine do `least_positive_float;` and `most_positive_float;`.

When you type a number, Macsyma simply returns it.

```
(c1) 15;
(d1)                15
(c2) 427.2;
(d2)                427.2
```

Macsyma preserves ratios as exact numbers rather than converting them automatically to a floating-point approximation.

```
(c3) 1/2;
                    1
(d3)                -
                    2
```

At any time, however, you can use the **sfloat** or **dfloat** commands to request a float approximation or the **bfloat** command to request a bigfloat approximation.

```
(c4) sfloat(%);
(d4)                0.5
```

Here is a very large integer, called a **bignum**; the “#” in the result is Macsyma’s way of showing that the number continues on the next line.

```
(c5) 10^506 - 10^253 - 1;
(d5)    99999999999999999999999999999999999999999999999999999999999999999999999#
9999999999999999999999999999999999999999999999999999999999999999999999999999999#
9999999999999999999999999999999999999999999999999999999999999999999999999999999#
9999999999999999999989999999999999999999999999999999999999999999999999999999999#
9999999999999999999999999999999999999999999999999999999999999999999999999999999#
9999999999999999999999999999999999999999999999999999999999999999999999999999999#
9999999999999999999999999999999999999999
```

The default precision for floats is governed by each computer’s hardware. In this example the precision shown is 8, but in some implementations the precision is 16.

```
(c6) sfloat(1/121);
(d6)                0.008264462
```

You can govern the default precision of bigfloats using the system variable **bfprecision**, whose default value is 20.

```
(c7) bfloat(1/121);
(d7)                8.2644628099173553719b-3
```

To change the default precision of bigfloats, reset **bfprecision**.

```
(c8) bfprecision:125$
(c9) bfloat(1/121);
(d9) 8.264462809917355371900826446280991735537190082644628099173553719008264462#
8099173553719008264462809917355371900826446280991736b-3
```

Generally an operation with an integer and a floating-point number results in a floating-point number.

```
(c10) 4 * 3.0;
(d10)                 12.0
```

The following commands allow you to manipulate expressions containing complex numbers:

- **realpart** and **imagpart** return the real and imaginary parts of an expression, respectively.
- **rectform** returns an expression in the form `a+b*%i`, where `a` and `b` are purely real.
- **polarform** returns an expression in the form `r*%e^(%i*theta)` where `r` and `theta` are purely real, and `%e` is the base of the natural logarithms (See page 23).

Consider the following examples.

The trigonometric functions that appear below, such as **sin** and **cos**, are described more fully in Section 4.6, page 50, and the exponential command **exp** is covered on page 24.

```
(c11) sin(exp(%i*y+x));
                             %i y + x
(d11)                  sin(%e        )
(c12) realpart(%);
                       x                 x
(d12)             sin(%e  cos(y)) cosh(%e  sin(y))
(c13) imagpart(d11);
                       x                 x
(d13)             cos(%e  cos(y)) sinh(%e  sin(y))
(c14) rectform(d11);
                            x                 x
(d14)               %i cos(%e  cos(y)) sinh(%e  sin(y))
                       x                 x
           + sin(%e  cos(y)) cosh(%e  sin(y))
```

The trigonometric command **atan2**, in (d15), returns the arctangent of y/x in the interval $(-\pi, \pi)$.

```
(c15) polarform(d11);
(d15) sqrt(cos (%e  cos(y)) sinh (%e  sin(y))
      2   x              2   x
 + sin (%e  cos(y)) cosh (%e  sin(y)))
                  x                 x                 x                 x
  %i atan2(cos(%e  cos(y)) sinh(%e  sin(y)), sin(%e  cos(y)) cosh(%e  sin(y)))
 %e
```

3.1.3 Variables

Variables are named quantities. You can either bind a value to a variable, or you can leave the variable unbound and treat it formally. Binding a value to a variable is called **assignment**. This section shows you how you can use both bound and unbound variables in expressions.

To assign a value to a variable, type the name of the variable, followed by the "**:**" operator, followed by the value. Don't confuse the "=" operator, which creates equations, with the assignment operator. The equation operator, discussed on page 24, does not assign values to variables.

When you type the name of an unbound variable, Macsyma simply returns that variable.

```
(c1) a;
(d1)                a
```

Assign the value 1234 to the variable a.

```
(c2) a:1234;
(d2)                1234
```

When you type the name of a bound variable, Macsyma returns its value.

```
(c3) a;
(d3)                1234
```

Once bound, a stands for 1234 in subsequent expressions.

```
(c4) a + a;
(d4)                2468
```

You can add bound and unbound variables.

```
(c5) a + b;
(d5)                b + 1234
```

A single quote before a bound variable supresses evaluation.

```
(c6) 'a;
(d6)                  a
```

Using the variable a in expressions does not change its original value.

```
(c7) a;
(d7)                1234
```

Confusing results often occur when you use a variable that you do not realize you have previously assigned. To remove a value from a variable use the **remvalue** command.

```
(c8) remvalue(a);
(d8)                [a]
```

```
(c9) a;
(d9)                 a
(c10) a + b;
(d10)              b + a
```

You can also create compound assignment statements by enclosing the assignments, separated by commas, in parentheses. Note that Macsyma returns the value of the last statement.

```
(c11) (a:5, b:15.3e3);
(d11)                15305.0
```

Check the values of a and b.

```
(c12) a;
(d12)                5
(c13) b;
(d13)                15305.0
```

Use the **ev** command to re-evaluate (c10); it results in the sum of the values of a and b. Note that the sum of a floating point number and an integer is a floating point number.

```
(c14) ev(c10);
(d14)                15305.0
```

An alternative way of entering `ev(c10)`, as shown below, is `''c10`.

```
(c15) ''c10;
(d15)                            b + a
```

You can also use **remvalue** to remove the values of more than one variable.

```
(c16) remvalue(a,b);
(d16)                [a, b]
```

Now both a and b are unbound.

```
(c17) ev(c10);
(d17)                b + a
```

Section 2.2 showed how you can use either the system variable % or a D-LABEL to refer to a previous expression. Another way to refer to an expression is by name, after assigning it to a variable.

```
(c18) expr1:num + x + y;
(d18)                y + x + num
(c19) num:50;
(d19)                    50
```

Notice that the value of *expr1* does not change when *num* is assigned the value of 50.

```
(c20) expr1;
(d20)               y + x + num
```

However, when you use **ev** to evaluate the expression, 50 appears in place of *num*. The commands `ev(expr1)` and `ev(d18)` are equivalent.

```
(c21) ev(expr1);
(d21)               y + x + 50
```

Assign the result shown in (d21) to a variable named *expr2*.

```
(c22) expr2:%;
(d22)               y + x + 50
```

Check its value, noting that it is different from (d20).

```
(c23) expr2;
(d23)               y + x + 50
```

Remove the value from the variable *num*.

```
(c24) remvalue(num)
(d24)                 [num]
```

Re-evaluate *expr1*, noting that the result is different from (d21). The command `''expr1` will give the same result as `ev(expr1)`.

```
(c25) ev(expr1);
(d25)               y + x + num
```

The value of *expr2* remains the same, even though *num* is unbound.

```
(c26) expr2;
(d26)               y + x + 50
```

These variables can be manipulated just as if they were expressions.

```
(c27) expr2 + 1;
(d27)               y + x + 51
(c28) expr1 * 2;
(d28)                           2 (y + x + num)
```

The system variable **values** is a list all variables that have assigned values.

```
(c29) values;
(d29)                 [expr1, expr2]
```

Use the keyword **all** to remove values from all defined variables.

```
(c30) remvalue(all);
(d30)               [expr1, expr2]
```

You can assign a value to a variable locally, so that it is bound only for the duration of the current command. To do so, include the variable assignment(s) after the command to be executed, separated by a comma from the command.

In this example the variables u and v are initially unbound.

```
(c31) expr:u + v;
(d31)                 v + u
```

Locally bind v to `5`. This command is equivalent to `ev(expr, v:5)`.

```
(c32) expr, v:5;
(d32)                 u + 5
```

After the command on (c32) executes, v reverts to being unbound.

```
(c33) v;
(d33)                   v
```

You can locally bind more than one variable at a time.

```
(c34) expr, v:a, u:4;
(d34)                 a + 4
```

Macsyma provides other methods for removing values and properties from variables. Refer to **remvalue** and **kill** in the *Macsyma Reference Manual* for more information.

3.1.4 Constants

Table 3.2 summarizes some commonly-used predefined constants.

`Constant`	Description
`%e`	Base of the natural logarithms (e)
`%i`	The square root of -1 (i)
`%pi`	The transcendental constant pi (π)
`inf`	Real positive infinity (∞)
`minf`	Real negative infinity (-∞)

Table 3.2: Some Predefined Macsyma Constants

Each term in the following sum simplifies to `1`.

```
(c1) log(%e) + sin(%pi/2);
```

```
(d1)                    2
```

You can use the base of natural logarithms explicitly.

```
(c2) u + sqrt(-1) * v;
(d2)                %i v + u
(c3) %e^%;
                     %i v + u
(d3)                %e
```

Alternatively, define **%e** using the exponential function. %**exp**.

```
(c4) exp(u + %i * v);
                     %i v + u
(d4)                %e
```

Re-evaluate the expression in (d4) with **ev**, locally binding u to `0` and v to **%pi**.

```
(c5) ev(%, u:0, v:%pi);
(d5)                - 1
```

The following notation is equivalent to the command in (c5).

```
(c6) d4, u:0, v:%pi;
(d6)                - 1
```

See Section 4.6, page 50, "Using Trigonometric Functions," for examples of complex numbers. See Section 6.3, page 89, "Taking Limits," for examples using the constants **inf** and **minf**.

3.2 Creating Equations

You can combine expressions with the "=" operator to describe equations.

```
(c1) eq:m*c^2 = e^2/r_cl;
                                  2
                         2       e
(d1)                    c  m = ----
                               r_cl
```

You can manipulate equations like any other mathematical quantities.

```
(c2) %*r_cl;
                         2           2
(d2)                    c  m r_cl = e
```

Here we solve for the *r_classical*. Macsyma also has a powerful **solve** command. Refer to **solve** in the *Macsyma Reference Manual* for more information.

```
(c3) %/m/c^2;
                                   2
                                  e
(d3)                     r_cl = ----
                                  2
                                 c  m
```

The operator "=" generally does not assign a value to a variable, it merely creates an equation. To assign a value to a variable you can use the ":" operator, described on page 20.

The variable *r_cl* is not bound by the equation operator.

```
(c4) r_cl;
(d4)                r_cl
(c5) ev(eq, m=1/2, c=1);
                           2
                    1     e
(d5)                - = ----
                    2   r_cl
```

In the second and subsequent arguments to the **ev** command, you can use the "=" operator to locally bind a value to a variable. In this case, "=" works like the ":" operator.

For more information about solving equations, see Chapter 5.

3.3 Defining Functions

To define a function, use the syntax

$$function(arg_1,\ arg_2,\ \ldots,\ arg_n) := body$$

where *function* is the name of the function, arg_i are the formal arguments in parentheses, ":=" is the function operator, and *body* is any Macsyma expression involving the variables arg_1 through arg_n.

Define the function f

```
(c1) f(p, q) := 9*p^4 - q^4 + 2*q^2;
                                     4    4      2
(d1)                f(p, q) := 9 p  - q  + 2 q
```

Find the value of f at $p = 10,864$ and $q = 18,817$. Because Macsyma is not limited to integers of a fixed size, it can return the correct answer.

```
(c2) f(10864,18817);
(d2)                       1
```

Evaluating f at $p = 10,864.0$ and $q = 18,817.0$ returns the wrong answer since the default precision for floating-point numbers is not high enough.

```
(c3) f(10864.0,18817.0);
(d3)                    0.0
```

The function l defines a line.

```
(c4) l(x) := m*x + b;
(d4)                l(x) := m x + b
```

The following (x, y) coordinates are three points on a line.

```
(c5) (x1:5201477, y1:99999,
      x2:5201478, y2:100000,
      x3:5201479, y3:100001)$
```

For optimal curve fitting with respect to least square approximation, use the following formulas for the variables m and b above. (See the LSQ package for finding best least square fits.)

```
(c6) m: (x1*y1 + x2*y2 + x3*y3 - (1/3)*(x1 + x2 + x3)*(y1 + y2 + y3)) /
    (x1^2 + x2^2 + x3^2 - (1/3)*(x1 + x2 + x3)^2)$


(c7) b: (1/3)*(y1 + y2 + y3) - (m/3)*(x1 + x2 + x3)$
```

Calculate the value for $l(5201480)$.

```
(c8) l(5201480);
(d8)                100002
```

Macsyma also allows you to deal with composite functions.

```
(c9) f(x) := x^2 + 4;
                                 2
(d9)                f(x) := x  + 4
(c10) g(x) := 1/x + 3;
                                1
(d10)               g(x) := - + 3
                                x
(c11) f(g(x));
                     1     2
(d11)               (- + 3)  + 4
                     x
(c12) g(f(x));
                        1
(d12)                 ------ + 3
                       2
                      x  + 4
```

```
(c13) f(g(2));
                               65
(d13)                          --
                                4
(c14) g(f(2));
                               25
(d14)                          --
                                8
```

You can change the definition of a function by redefining it, or remove a function definition using **remfunction**. You can also find out about the functions that are currently defined. The system variable *functions* holds a list of all defined functions. The command **dispfun** displays the definitions of the functions you specify.

Redefine g to remove its previous definition.

```
(c15) g(x) := x - a;
(d15)                g(x) := x - a
```

Check which functions are currently defined.

```
(c16) functions;
(d16)                 [l(x), f(x), g(x)]
```

Check the definitions for both of these functions.

```
(c17) dispfun(f,g);
                                  2
(e17)                f(x) := x  + 4
(e18)                g(x) := x - a
(d18)                 done
```

Remove the function definitions from f and g.

```
(c19) remfunction(f, g);
(d19)                [f, g]
```

If the function you call is undefined, Macsyma simply returns what you typed.

```
(c20) f(14);
(d20)                 f(14)
```

3.4 Using Lists

A list in Macsyma is an ordered set of elements, separated by commas and enclosed in square brackets. An element of a list can be any Macsyma expression. You can assign a list as the value of a variable, and then refer to its individual elements as **subscripted variables**.

Lists can be used as arguments to some commands, such as **solve** (Chapter 5) and **matrix** (Chapter 7). Other commands return results in a list, including **remvalue**, **remfunction**, and **solve**.

```
(c1) list1:[aa,bb,cc];
(d1)                    [aa, bb, cc]
(c2) list2:[dd,ee,ff];
(d2)                    [dd, ee, ff]
```

Add the elements of the two lists together.

```
(c3) list1+list2;
(d3)                 [dd + aa, ee + bb, ff + cc]
```

Multiply the elements of the two lists together, assigning the results to *list3*.

```
(c4) list3:list1*list2;
(d4)                [aa dd, bb ee, cc ff]
```

Find the inner product of the two lists; notice that the result is not a list.

```
(c5) list1.list2;
(d5)                cc ff + bb ee + aa dd
```

You can use the commands **first**, **last**, **part**, and **rest** to access the elements of a list. These commands work for any expression, but are particularly useful for referring to parts of a list. In addition, the **length** command returns the length of the specified list. This command is useful in finding out the number of solutions in a list returned by **solve** (see Chapter 5).

The **first** command returns the first element in the list.

```
(c6) first(list3);
(d6)                  aa dd
```

The **rest** command returns a list consisting of all the elements in the list except the first element.

```
(c7) rest(list2);
(d7)                [ee, ff]
```

The **last** command returns the last element in the specified list.

```
(c8) last(list1);
(d8)                   cc
```

Here is how to subtract the second item in *list3* from the first item in *list3*.

```
(c9) first(list3) - first(rest(list1));
(d9)                    aa dd - bb
```

Here the **rest** command returns all but the first two elements.

```
(c10) rest(list2, 2);
(d10)                  [ff]
```

The **length** function returns the number of the elements in the specified list.

```
(c11) length(list1) + length(list2);
(d11)                    6
(c12) length(list1 + list2);
(d12)                    3
```

You can also extract parts of a list with the **part** command, described in Section 4.5, page 45. This section also shows you how the **first**, **last**, and **rest** commands work on other kinds of expressions.

3.5 Using Arrays

In Macsyma an **array** is an n-dimensional data structure. Arrays enable you to refer to a collection of elements using a single name. An element of an array is indexed by a **subscripted variable**, which is a name followed by a list of subscripts enclosed in square brackets.

Macsyma supports two **array types**, declared and undeclared. Declared arrays have a fixed size allocation and are indexed by non-negative integers. Undeclared arrays, called "hashed" arrays, are of arbitrary size and indexing. If you assign a value to a subscripted variable without declaring the corresponding array, the system sets up an undeclared array. Declared arrays are similar to the fixed-size arrays that you can declare in other programming languages, such as FORTRAN.

To declare an array, use the command

(*arrayname*, [*dimension*$_1$, ..., *dimension*$_n$])

to declare the number of dimensions and indicate the maximum value of each subscript. The system then allocates space for the entire array.

If you know the size of the array you will need, it is generally best to declare it, since this is more efficient. Undeclared arrays are less efficient, but can grow to any size.

In the next example, Macsyma does not know about the array *foo*.

```
(c1) foo[5];
(d1)                   foo
                          5
```

Set up an undeclared array by assigning the value 1 to the first (0th—Macsyma arrays are 0 based) element.

```
(c2) my_factorial[0]:1;
(d2)                     1
```

Define the other elements of the undeclared array recursively.

```
(c3) my_factorial[n] := n * my_factorial[n - 1];
(d3)           my_factorial  := n my_factorial
                           n                  n - 1
```

Now, asking for the element in slot 8 returns the factorial of 8.

```
(c4) my_factorial[8];
(d4)                   40320
```

Declare a one-dimensional array whose maximum subscript value is 8.

```
(c5) array(your_factorial, 8)$
```

Define the elements of the array as factorials, similar to `my_factorial` above.

```
(c6) your_factorial[0]:1$
(c7) your_factorial[n] := n * your_factorial[n - 1]$
```

Like `my_factorial`, the array `your_factorial` holds the factorial of 8.

```
(c8) your_factorial[8];
(d8)                   40320
```

Unlike the undeclared array, however, `your_factorial` cannot handle subscripts larger than 8.

```
(c9) my_factorial[9];
(d9)                  362880
(c10) your_factorial[9];
Array YOUR_FACTORIAL has dimensions [8], but was called with [9]
(d10)                   []
```

Another example of recursion appears in Section 11.6, "Writing Recursive Functions."

The system variable **arrays** is a list of all existing arrays, declared and undeclared.

```
(c11) arrays;
(d11)              [my_factorial, your_factorial]
```

For hashed (undeclared) arrays, the command **arrayinfo**() returns the word "`hashed`," the number of dimensions, and the subscripts of every element that has a value. For declared arrays, the command returns the word "`declared`," the number of dimensions, and the maximum value of each dimension's subscript.

```
(c12) arrayinfo(my_factorial);
(d12)  [hashed, 1, [0], [1], [2], [3], [4], [5], [6], [7], [8], [9]]
(c13) arrayinfo(your_factorial);
(d13)               [declared, 1, [8]]
```

Section 7.1, page 115 describes how you can use arrays to create matrices. Refer to **arrays** in the *Macsyma Reference Manual* for more information.

Chapter 4

Algebra

You can manipulate algebraic expressions in many ways. Macsyma provides facilities for expanding, simplifying, and factoring expressions, making substitutions in expressions, extracting parts of an expression for subsequent use in other commands, and using trigonometric functions.

Sections 4.1 through 4.6 cover these topics and provide many examples using the algebraic manipulation commands. Section 4.8, "Practice Problems," presents several problems exercising these capabilities for you to try on your own. Solutions appear in Appendix B.

Although the examples in this chapter introduce many commands and option variables, the scope of this document allows only a limited introduction to Macsyma's algebraic manipulation capabilities. To learn more, consult the *Macsyma Reference Manual*.

4.1 Expanding Expressions

Macsyma provides several expansion-related commands, each of which expands its argument in a different way. The **expand** command, for example, multiplies out product sums and exponentiated sums, recursing through all levels of the expression. The **distrib** command also distributes sums over products, but works only at the top level of the expression.

This section presents the following commands:

- **expand** expands the given expression by multiplying out products of sums and exponentiated sums at all levels of the expression.**expand**
- **multthru** multiplies a term or terms through a sum or equation.
- **distrib** expands the given expression by distributing sums over products.
- **partfrac** does a complete partial fraction decomposition, expanding an expression in partial fractions with respect to a given main variable.

For a summary of the differences between **expand**, **multthru**, and **distrib**, see Table 4.1, on page 35.

Macsyma also provides several option variables that you can set to modify the way the expansion commands work. Two option variables, **logexpand** and **radexpand**, are discussed in this section.

The command **expand**(*exp*) multiplies out products of sums and exponentiated sums, splits numerators of rational expressions that are sums into their respective terms, and distributes both commutative and non-commutative multiplication over addition at all levels of the expression *exp*. Use **expand**(*exp*, *p*, *n*), to multiply out only terms with exponents e, such that $n \leq e \leq p$. Refer to **expand** in the *Macsyma Reference Manual* for more information.

```
(c1) expr1:(1/(a + b)^2 + x/(a - b)^3)^2;
                           x          1     2
(d1)                   (-------- + --------)
                               3          2
                        (a - b)    (b + a)
```

Expand *expr1*, assigning the result to *expr2*.

```
(c2) expr2:expand(expr1);
                             2
                            x
(d2) -----------------------------------------------------------
       6        5        2  4       3  3       4  2      5      6
      b  - 6 a b  + 15 a  b  - 20 a  b  + 15 a  b  - 6 a  b + a
                           2 x
        + -------------------------------------------
              5      4      2  3      3  2    4  5
          - b  + a b  + 2 a  b  - 2 a  b  - a  b + a
                             1
        + -----------------------------------
            4        3      2  2      3      4
          b  + 4 a b  + 6 a  b  + 4 a  b + a
```

With additional arguments, **expand** multiplies out specific terms only; (c3) expands certain terms of *expr1* with positive exponents.

```
(c3) expr3:expand(expr1, 2, 0);
                      2
                     x             2 x              1
(d3)             -------- + ----------------- + --------
                        6          3        2          4
                 (a - b)    (a - b)  (b + a)    (b + a)
```

Expand certain terms of *expr1* with negative exponents.

```
(c4) expand(expr1, 0, -2);
                       x              1        2
(d4)              (-------- + ---------------)
                          3    2             2
                   (a - b)    b  + 2 a b + a
```

You can set option variables to control how much and what kind of expansions are to take place. The option variable **logexpand** controls the expansion of logarithms of products and powers, and **radexpand** controls the expansion of expressions containing radicals.

The **log** command gives the natural log of its argument in base **e**.

```
(c5) log(%e)
(d5)                     1
```

When **logexpand** is **true**, Macsyma does not simplify the logarithms of products and quotients.

```
(c6) logexpand;
(d6)                   true
(c7) log(a*b);
(d7)                 log(a b)
(c8) log(a/b);
                          a
(d8)                  log(-)
                          b
```

Resetting **logexpand** to **all** tells Macsyma to simplify these logarithms. Refer to **logexpand** in the *Macsyma Reference Manual* for more information.

```
(c9) logexpand:all;
(d9)                    all
(c10) log(a*b);
(d10)                 log(b) + log(a)
(c11) log(a/b);
(d11)                 log(a) - log(b)
```

When **radexpand** is **true**, Macsyma does not simplify radicals containing products, quotients, and powers

```
(c12) radexpand;
(d12)                  true
(c13) sqrt(x^y);
                           y
(d13)                sqrt(x )
(c14) sqrt(x*y);
(d14)                sqrt(x y)
(c15) sqrt(x/y);
                           x
(d15)                 sqrt(-)
                           y
```

Resetting **radexpand** to **all** tells Macsyma to simplify these radicals

```
(c16) radexpand:all;
(d16)                   all
```

```
(c17) sqrt(x^y);
                         y/2
(d17)                   x
(c18) sqrt(x*y);
(d18)                   sqrt(x) sqrt(y)
(c19) sqrt(x/y);
                        sqrt(x)
(d19)                   -------
                        sqrt(y)
```

For more information on the kinds of option variables that affect expansion, type `apropos(expand);` or consult the *Macsyma Reference Manual.*

The command **multthru**(exp_1, exp_2) multiplies each term in the expression exp_2, which should be a sum or an equation, by the expression exp_1. Alternatively, the command **multthru**(*exp*) multiplies one of the expression *exp*'s factors, which should be a sum, by the other factors of *exp*. That is, for an expression

$$\mathtt{f}_1 * \mathtt{f}_2 * \ldots * \mathtt{f}_n$$

where at least one factor f_i is a sum of terms, the command **multthru**($\mathtt{f}_1 * \mathtt{f}_2 * \ldots * \mathtt{f}_n$) multiplies each term in the sum f_i by all the other factors in the product.

Multiply each term in the sum *expr3* by $(a-b)^4$.

```
(c20) multthru((a - b)^4, expr3);
                        2                             4
                       x        2 (a - b) x    (a - b)
(d20)                -------- + ----------- + --------
                            2            2            4
                     (a - b)      (b + a)      (b + a)
```

Using **multthru**, multiply each term in the equation by $(a-b)$.

```
(c21) a = 23*x^2 + y^2 - y^3/(a - b);
                             3
                            y        2       2
(d21)                 a = - ----- + y  + 23 x
                            a - b
(c22) multthru(a-b, %);
                             3            2                2
(d22)         a (a - b) = - y  + (a - b) y  + 23 (a - b) x
(c23) expr4:((b + a)^10*(s - t)^2 + 2*a*b*(s - t) + a^2*b^2*(s - t))
            /a/b/(s - t)^4;
                        10         2    2  2
               (b + a)   (s - t)  + a  b  (s - t) + 2 a b (s - t)
(d23)          ------------------------------------------------
                                          4
                              a b (s - t)
```

Decompose *expr4* into a sum of terms, each of which is factored into lowest terms.

```
(c24) expr5:multthru(expr4);
                                 10
                           (b + a)          a b         2
(d24)                   ------------ + -------- + --------
                                    2          3          3
                        a b (s - t)     (s - t)     (s - t)
```

The command **distrib**(*exp*) distributes sums over products in the expression *exp*. Unlike the **expand** command, **distrib** works only at the top level of an expression and expands all the sums at that level.

```
(c25) (x + 1)*((u + v)^2 + a*(w + z)^2);
                                             2          2
(d25)                  (x + 1) (a (z + w)  + (v + u) )
```

Compare the result returned by **multthru**, which expands at all levels of the expression, with the results returned by **distrib** and **expand**

```
(c26) multthru(%);
                                        2          2
(d26)              a (x + 1) (z + w)  + (v + u)  (x + 1)
(c27) distrib(d25);
                               2            2          2            2
(d27)            a x (z + w)  + a (z + w)  + (v + u)  x + (v + u)
(c28) expand(d25);
              2      2                              2      2                2
(d28) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x + 2 u v x + u  x
                                        2    2            2
                                 + a w  + v  + 2 u v + u
```

Table 4.1 summarizes the differences between **distrib**, **multthru**, and **expand**.

exp	**distrib**(*exp*)	**multthru**(*exp*)	**expand**(*exp*)
$(a+b)(c+d)$	$bd+ad+bc+ac$	$(b+a)d+(b+a)c$	$bd+ad+bc+ac$
$\frac{1}{(a+b)(c+d)}$	$\frac{1}{(b+a)(d+c)}$	$\frac{1}{(b+a)(d+c)}$	$\frac{1}{bd+ad+bc+ac}$
$(b(d+c)+1)(f+a)$	$b(d+c)f+f+ab(d+c)+a$	$(b(d+c)+1)f+a(b(d+c)+1)$	$bdf+bcf+f+abd+abc+a$

Table 4.1: A Comparison of **distrib**, **multthru** and **expand**

The **partfrac**(*exp*, *var*) command performs a complete partial fraction decomposition, expanding the expression *exp* in partial fractions with respect to the main variable *var*.

```
(c29) (1/(a + b)^2 + x/(a - b)^3)^2;
                          x         1     2
(d29)                (-------- + --------)
                             3          2
                      (a - b)    (b + a)
(c30) partfrac(%, a);
         2
        x          3 x              x              x              x
(d30) -------- - ------------ - ------------ + ------------ - ------------
             6      4              3        2            3  2           2  3
      (a - b)    8 b  (b + a)   4 b  (b + a)    2 (a - b)  b    2 (a - b)  b
                   3 x          1
             + ----------- + --------
                        4          4
               8 (a - b) b    (b + a)
```

4.2 Simplifying Expressions

Macsyma provides many commands that simplify expressions. The following simplification commands are described in this section:

- **ratsimp** simplifies an expression by combining the rational functions in the expression, then canceling out the greatest common divisor in the numerator and denominator.
- **radcan** simplifies expressions containing radicals, logarithms, and exponentials.
- **scsimp** implements the sequential comparative simplifier, which applies given identities to an expression in an effort to obtain a smaller expression.
- **combine** and **rncombine** group the terms in a sum that have the same denominator into a single term.
- **xthru** combines the terms of a sum over a common denominator without expanding them first.
- **map** can apply a given function, such as a simplification command, to each term of a very large expression. This can be useful when applying the function to the entire expression would be inefficient.

This section also introduces the option variables **algebraic** and **ratfac**.

The command **ratsimp**(*exp*) simplifies the expression *exp* and all of its subexpressions, including the arguments to nonrational functions. With additional arguments, **ratsimp** (*exp*, var_1, var_2, ..., var_n) specifies the ordering of each variable var_i as well.

ratsimp first combines the sum of rational functions (quotients of polynomials) into one rational function, then cancels out the greatest common divisor of this rational function's numerator and denominator.

```
(c1) -(x^5 - x)/(x - 1) + x + x^2 + x^3 + x^4 + (a + b + c)^3;
```

```
                  5
                 x  - x    4    3    2                   3
(d1)           - ------ + x  + x  + x  + x + (c + b + a)
                  x - 1
```

Simplify the expression in (d1) with the **ratsimp** command.

```
(c2) ratsimp(%);
      3                2       2              2       3        2      2      3
(d2) c  + (3 b + 3 a) c  + (3 b  + 6 a b + 3 a ) c + b  + 3 a b  + 3 a  b + a
```

Simplify the following expression with Macsyma's normal ordering of variables.

```
(c3) d*(w + a)*x + c*(w + a)*x + b*d + b*c + c*d;
(d3)          d (w + a) x + c (w + a) x + cd + b d + b c
(c4) ratsimp(%);
(d4)          ((d + c) w + a d + a c) x +(c + b) d + b c
```

Resimplify (d3), ordering d first and c second.

```
(c5) ratsimp(d3, c, d);
(d5)          d ((w + a) x + c + b) + c ((w + a) x + b)
```

In (c2) and (d2) Macsyma simplified the terms containing the variable x to zero, but it returned the expanded result of $(a+b+c)^3$, which is the canonical form of the polynomial. With some practice, you will learn to use the various simplification commands to transform an expression into the form you want.

You can set the option variable **ratfac** to **true** to invoke a partially factored form for canonical rational expressions.

Bind **ratfac** to **true** for the duration of this command, and simplify the expression in (d3).

```
(c6) ratsimp(d3, c, d), ratfac:true;
(d6)             (d + c) ((w + a) x + b)
```

By setting the option variable **algebraic** to **true**, you can indicate that algebraic integers are to be simplified. By default, this variable is **false**.

```
(c7) 1/(sqrt(x) - 2);
                              1
(d7)                     -----------
                         sqrt(x) - 2
(c8) ratsimp(%);
                              1
(d8)                     -----------
                         sqrt(x) - 2
```

Locally bind **algebraic** is **true**, then use **ratsimp** to rationally simplify the expression in (d8).

```
(c9) ratsimp(%), algebraic:true;
                        sqrt(x) + 2
(d9)                    -----------
                           x - 4
```

The simplification command **radcan**(*exp*) handles expressions containing radicals, logarithms, and exponentials.

Here we simplify an expression using **radcan**.

```
(c10) (log(x^2 + x) - log(x))^a/log(x + 1)^(a/2)
      + (%e^x - 1)/(%e^(x/2) + 1)
      + sqrt(x^2 - y^2)/sqrt(x - y);
                   2    2          2               a       x
            sqrt(x  - y )   (log(x  + x) - log(x))     %e  - 1
(d10)       ------------- + ---------------------- + ---------
            sqrt(x - y)                  a/2             x/2
                              log(x + 1)             %e    + 1
(c11) radcan(%);
                                      a/2     x/2
(d11)         sqrt(y + x) + log(x + 1)    + %e    - 1
```

The command **scsimp**(*exp*, $rule_1$, ..., $rule_n$) applies each identity $rule_i$ to the expression *exp*. If the expression that results from the application of these identities is smaller, the **scsimp** repeats the process. If the resulting expression is not smaller after all the identities are applied, the command returns the original expression *exp*.

Use **scsimp** to simplify *expr8*, subjected to the constraints *eq1* and *eq2* as simplification rules

```
(c12) expr8:(k1*k4 - k1*k2 - k2*k3)/k3^2;
                     k1 k4 - k2 k3 - k1 k2
(d12)                ---------------------
                                2
                              k3
(c13) eq1:k1*k4 - k2*k3 = 0;
(d13)                  k1 k4 - k2 k3 = 0
(c14) eq2:k1*k2 + k3*k4 = 0;
(d14)                  k3 k4 + k1 k2 = 0
(c15) scsimp(expr8, eq1, eq2);
                           k4
(d15)                      --
                           k3
```

The command **combine**(*exp*) groups terms with the same denominator into a single term.

Using **multthru**, write this expression as a sum of fractions.

```
(c16) ((b + a)^10*(s - t)^2 + 2*a*b*(s - t) + a^2*b^2*(s - t)) /a/b/(s - t)^4;
```

```
                        10         2    2  2
                 (b + a)   (s - t)  + a  b  (s - t) + 2 a b (s - t)
(d16)            ------------------------------------------------
                                             4
                                a b (s - t)
(c17) m_expr:multthru(%);
                              10
                        (b + a)          a b          2
(d17)                   ----------- + -------- + --------
                                  2          3          3
                        a b (s - t)    (s - t)    (s - t)
```

The command **xthru**(*exp*) combines all terms of the expression *exp*, which should be a sum, over a common denominator, without expanding products and exponentiated sums the way **multthru** does.

Combine the terms of the expression *m_expr* over a common denominator.

```
(c18) xthru(m_expr);
                             10
                        (b + a)   (s - t) + a b (a b + 2)
(d18)                   --------------------------------
                                          3
                               a b (s - t)
```

Combine all terms with the same denominator into a single term.

```
(c19) cd_expr:x/a + y/a + z/(2 * a);
                         z    y    x
(d19)                   --- + - + -
                        2 a   a    a
(c20) combine(cd_expr);
                         z    y + x
(d20)                    -- + -----
                        2 a     a
(c21) rncombine(cd_expr);
                         z + 2y + 2 x
(d21)                    ------------
                             2 a
```

The command **map**(*function*, *exp*) applies a *function*, such as a simplification command, to each term of an expression *exp*. The **map** command is useful when applying a function to a large expression would be inefficient. Refer to **map** in the *Macsyma Reference Manual* for more information.

```
(c22) map(f, cd_expr);
                           z       y      x
(d22)                    f(---) + f(-) + f(-)
                           2 a     a      a
```

4.3 Factoring Expressions

Macsyma provides a number of commands and option variables to factor expressions. This section presents the following commands:

- **factor** factors an expression into factors irreducible over the integers.
- **cfactor** factors an expression with respect to one variable, using complex numbers, including radicals.
- **factorsum** tries to separate the terms in a sum into groups that can have common factors, then factors them.

The section also presents examples of the option variable **dontfactor**.

To factor irreducibly, you can use the command **factor**(*exp*), where *exp* is an expression containing any number of variables or functions.

Factor an expression irreducibly over the integers. Macsyma does not necessarily return a simplified result.

```
(c1) x^28 + 1;
                                    28
(d1)                               x   + 1
(c2) factor(%);
                    4        24     20     16     12     8    4
(d2)          (x  + 1) (x   - x   + x   - x   + x  - x  + 1)
```

This command does not return x^{28}, as might be expected.

```
(c3) % - 1;
                    4        24     20     16     12     8    4
(d3)          (x  + 1) (x   - x   + x   - x   + x  - x  + 1) - 1
```

You can resimplify the result before applying it to other commands.

```
(c4) ratsimp(%);
                            28
(d4)                       x
```

Macsyma finds only those factors whose coefficients are polynomials in the coefficients of the input polynomial.

```
(c5) a*x^2 - 4*a;
                             2
(d5)                     a x  - 4 a
(c6) factor(%);
(d6)                    a (x - 2) (x + 2)
```

The **factor** command does not perform algebraic extensions during the factoring process; $x^2 - 2$ is irreducible over the integers.

```
(c7) a*x^2 - 2*a;
                            2
(d7)                    a x  - 2 a
```

```
(c8) factor(%);
                                2
(d8)                    a (x  - 2)
```

The **cfactor** command factors with complex numbers and radicals.

```
(c9) cfactor((a*x^2 - 2*a), x);
(d9)                    a(x - sqrt(2)) (x + sqrt(2))
```

When you solve a polynomial using the function **solve** (discussed in Chapter 5), Macsyma calls the function **factor** to factor it, if necessary. If the factors are of degree 4 or less, Macsyma applies standard formulas to generate the solution.

The command **factorsum**(*exp*) tries to separate the terms of *exp*, which should be a sum, into groups that can have common factors, then factors them.

Here we enter an expression to be factored.

```
(c10) (x + 1)*((u + v)^2 + a*(w + z)^2);
                                             2          2
(d10)                (x + 1) (a (z + w)  + (v + u) )
```

Then we expand the expression, assigning the result to the variable *expr7*.

```
(c11) expr7:expand(%);
           2      2                            2      2                2
(d11)  a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x + 2 u v x + u  x
               2    2            2
          + a w  + v  + 2 u v + u
```

Notice that the **factor** command does not return the expression shown in (d10).

```
(c12) factor(expr7);
                          2                2    2            2
(d12)          (x + 1) (a z  + 2 a w z + a w  + v  + 2 u v + u )
```

The command **factorsum** can convert the expression back into the original factored form.

```
(c13) factorsum(expr7);
                                            2          2
(d13)                (x + 1) (a (z + w)  + (v + u) )
```

You can set **dontfactor** to a list of variables for which factoring is disabled in subsequent commands.

```
(c14) expr9:(x^2*y^2 +2*x*y^2 + y^2 - x^2 - 2*x - 1)/36/(y^2 + 2*y + 1);
                  2  2        2    2    2
                 x  y  + 2 x y  + y  - x  - 2 x - 1
(d14)            ----------------------------------
                              2
                     36 (y  + 2 y + 1)
```

Set **dontfactor** to a list containing the variable x, which is to be left unfactored.

```
(c15) dontfactor:[x];
(d15)                      [x]
(c16) factor(expr9);
                             2
                           (x  + 2 x + 1) (y - 1)
(d16)                      ----------------------
                                36 (y + 1)
```

Reset **dontfactor** to an empty list, so that the factoring of x is enabled again. Note the difference between (d18) and (d16).

```
(c17) dontfactor:[]$
(c18) factor(expr9);
                                 2
                          (x + 1)  (y - 1)
(d18)                     ----------------
                             36 (y + 1)
```

4.4 Making Substitutions

Macsyma allows you to perform many kinds of substitution, such as substituting one expression for another in a third. This section presents the following commands:

- **ev** evaluates a given expression in the specified environment.
- **subst** makes replacements by substitution, with some restrictions on the expression being replaced.
- **ratsubst** is similar to **subst**, but without the restrictions on the expression being replaced.

The **ev**(*exp*, *arg*$_1$, ..., *arg*$_n$) command allows you to re-evaluate an expression, specifying a replacement for some variable in that expression. Refer to **ev** in the *Macsyma Reference Manual* for more information.

```
(c1) expr11:z*%e^z;
                                         z
(d1)                                 z %e
```

Here **ev** replaces every occurrence of z with x^2 in *expr11* by temporarily binding z to x^2, then re-evaluating *expr11*.

```
(c2) ev(expr11,z=x^2);
                                              2
                                         2   x
(d2)                                    x  %e
```

In the next example, (c3) shows an equivalent notation for performing this replacement

```
(c3) expr11, z=x^2;
                                              2
                                         2   x
(d3)                                    x  %e
```

The variable *expr11* itself does not change.

```
(c4) expr11;
                                             z
(d4)                                     z %e
```

To make replacements by substitution, you can use the command **subst***(a, b, c)*, where a is the expression you want to substitute for expression b in expression c. The argument b must be an **atom** (*i.e.* a number, a string, or a symbol) or a complete subexpression of c. When b does not have these characteristics, use **ratsubst***(a, b, c)* instead.

Substitute x^2 for z in *expr11*; the actual value of *expr11* does not change.

```
(c5) subst(x^2, z, expr11);
                             2
                        2   x
(d5)                   x  %e
```

A form which is equivalent to the one above uses the "=" operator.

```
(c6) subst(z = x^2, expr11);
                             2
                        2   x
(d6)                   x  %e
```

To change the value of *expr11*, reassign it.

```
(c7) expr11:subst(a^2*b^3, z, expr11);
                                2  3
                        2  3   a  b
(d7)                   a  b  %e
```

```
(c8) expr11;
                                 2  3
                         2  3   a  b
(d8)                    a  b  %e
```

We set *expr12* to an expression which contains four atoms, a, b, c, and d, and two complete subexpressions, d and $c + b + a$.

```
(c9) expr12:(a + b + c)/d;
                        c + b + a
(d9)                    ---------
                            d
```

The **subst** command cannot replace $a + b$ because it is not a complete subexpression of *expr12*.

```
(c10) subst(d, a + b, expr12);
                        c + b + a
(d10)                   ---------
                            d
```

Use **ratsubst** to perform the replacement.

```
(c11) ratsubst(d, a + b, expr12);
                          d + c
(d11)                     -----
                            d
```

The **subst** command makes only syntactic substitutions.

```
(c12) expr13:expand((1 + sin(x))^4);
                  4            3            2
(d12)          sin (x) + 4 sin (x) + 6 sin (x) + 4 sin(x) + 1
(c13) subst(1 - cos(x)^2, sin(x)^2, expr13);
                   4            3                            2
(d13)           sin (x) + 4 sin (x) + 4 sin(x) + 6 (1 - cos (x)) + 1
(c14) ratsubst(1 - cos(x)^2, sin(x)^2, expr13);
                        2                  4            2
(d14)         (8 - 4 cos (x)) sin(x) + cos (x) - 8 cos (x) + 8
```

You can specify the subexpression to replace explicitly, as shown above. Macsyma also provides commands for indicating the parts of an expression by their position. These commands appear in the next section.

4.5 Extracting Parts of an Expression

You have many commands available for extracting parts of Macsyma expressions for use in other contexts. This section presents the following commands:

- **part** returns the subexpression you specify, according to its position in the expression. Table 4.2, page 46, summarizes how to use this command.
- **dpart** is similar to **part** except that it returns the entire expression, with the selected subexpression displayed inside a box.
- **substpart** substitutes the characters you specify for the indicated subexpression, then returns the new value of the expression.
- **pickapart** assigns intermediate display lines (E-LINES) to all subexpressions of an expression, down to a specified depth.
- **reveal** displays an expression to the specified integer depth, indicating the length of each part.
- **lhs** and **rhs** return the left and right sides of the given equation, respectively.
- **first** and **last** return the first and last part of the specified expression, respectively.
- **rest** returns the expression with one or more of its leading elements removed.
- **coeff** and **ratcoef** return the coefficient of a given variable in the specified expression.

In addition, the system variable **piece** holds the last expression selected with one of the part extraction commands. The variable **piece** is set during the execution of the part extraction command and thus can be used within the command itself.

To extract part of an expression you can use **part**(*exp, n*), where *exp* refers to an equation or expression, and *n* is an integer that represents the part you want. Table 4.2 summarizes the rules for using the **part** command.

Here **part** is used to extract the second part of *eq3*.

```
(c1) eq3:x^2 + 2*x + 2 = y^2 + 1
                     2            2
(d1)                x  + 2 x + 2 = y  + 1
(c2) part(%, 2);
                        2
(d2)                   y  + 1
```

Now we can extract the first part of the result shown in (d2).

```
(c3) part(%, 1);
                          2
(d3)                     y
```

Notice in Table 4.5 that part `0` is always the operator and the arguments are the successive parts. The equation `a=b` is interpreted by the **part** command as if it were in the functional notation "="`(a,b)`, similar to `f(x,y,z)`.

Command	`exp:f(x, y, z);`	`exp:a=b;`
`part(exp, 0)`	`f`	`=`
`part(exp, 1)`	`x`	`a`
`part(exp, 2)`	`y`	`b`
`part(exp, 3)`	`z`	

Table 4.2: Using the **part** Command

With additional arguments, **part**$(exp, num_1, \ldots, num_n)$ allows you to obtain the part of the expression *exp* specified by part num_1, then find the num_2 part of the resulting expressions, and so on.

This command is equivalent to (c2) and (c3) above.

```
(c4) part(eq3,2,1);
                            2
(d4)                       y
```

The command **dpart**(exp, num_1, ..., num_n) is similar to **part**, except that instead of simply returning the specified subexpression, this command returns the entire expression with the selected subexpression displayed inside a box.

In addition, the system variable **piece** holds the last expression selected with one of the part selection commands, such as **part**, **dpart**, and **substpart** (below).

Select a part of the expression *big_expr1* to be highlighted with a box in the output

```
(c5) big_expr1:(x^3 + 3*x^2 + 3*x + 1)/(d^(x^3 + 3*x^2 + 3*x + 1) + n);
                        3      2
                       x  + 3 x  + 3 x + 1
(d5)                 -------------------------
                            3      2
                           x  + 3 x  + 3 x + 1
                     n + d
(c6) dpart(big_expr1, 2, 2, 2);
                            3      2
                           x  + 3 x  + 3 x + 1
(d6)                ------------------------------
                          ************************
                          *  3      2            *
                          * x  + 3 x  + 3 x + 1  *
                          ************************
                    n + d
```

Return the value of **piece**, whose value is the subexpression selected by **dpart** above.

```
(c7) piece;
                         3      2
(d7)                    x  + 3 x  + 3 x + 1
```

The command **substpart**(x, *exp*, num_1, ..., num_n) substitutes x for a subexpression then returns the new value of the expression *exp*. You can indicate the subexpression for **substpart** just as you do for **part**, by specifying the arguments *exp*, num_1, ..., num_n.

Note that x can be some operator to be substituted for an operator of *exp*. In some cases, you might need to enclose x in double quotes; for example, the command `substpart("+", a*b, 0);` returns `b + a`.

Here we factor the part of *big_expr1* selected by **dpart**, then substitute back into the expression.

```
(c8) substpart(factor(piece), big_expr1, 2, 2, 2);
                      3      2
                     x  + 3 x  + 3 x + 1
(d8)                 -------------------
                                    3
                             (x + 1)
                        n + d
```

The command **pickapart**(*exp*, *depth*) assigns E-LABELs to all subexpressions of the expression *exp* down to the specified integer *depth*. You will find this command useful for dealing with large expressions, and in order to assign parts of expressions to variables without having to use the **part** command.

Display subexpressions of *big_expr2* down to the second level, assigning each subexpression to an E-LABEL.

```
(c9) big_expr2:log(a*x^2 + b*x + c)^4
           - 1/(1 + 1/y)^(1/2)
           + exp(-%i*cos(12*a - b + c))
           + a0*sin(a*x^2 + b)^2
           - x*y;
                  1           4    2                    2    2
(d9) - x y - ----------- + log (a x  + b x + c) + a0 sin (a x  + b)
                  1
             sqrt(- + 1)
                  y
                                                 - %i cos(c - b + 12 a)
                                             + %e
(c10) pickapart(big_expr2, 2);
(e10)                       x y
                             1
(e11)                  -----------
                             1
                       sqrt(- + 1)
                             y
                           2
(e12)                log(a x  + b x + c)
                          2    2
(e13)                  sin (a x  + b)
(e14)             - %i cos(c - b + 12 a)
                   e14              4
(d14)            %e    + a0 e13 + e12  - e11 - e10
```

The command **reveal**(*exp*, *depth*) displays the expression *exp* to the specified integer *depth*, indicating the length of each part.

Use **reveal** to break down the expression *big_expr2* to a depth of 6.

```
(c15) reveal(big_expr2, 6);
```

Note that **reveal** displays sums as `Sum(n)`, products as `Product(n)`, quotients as `Quotient`, and exponentials as `Expt`.

```
                         1               4
(d15) - x y - ------------------ + log (a Expt + b x + c)
              sqrt(Quotient + 1)
                            2                      - %i cos(Sum(3))
                 + a0 sin (Product(2) + b) + %e
```

The command **lhs** extracts the left hand side of an equation, and the command **rhs** extracts the right.

Use **rhs** to extract the right side of the equation below.

```
(c16) eq4:eq3 - 1;
                         2              2
(d16)                   x  + 2 x + 1 = y
(c17) rhs(%);
                            2
(d17)                      y
```

Use **lhs** to extract the left side of *eq3*.

```
(c18) lhs(eq4);
                           2
(d18)                     x  + 2 x + 1
```

If the expression is not an equation, **lhs** returns the entire expression and **rhs** returns `0`.

```
(c19) lhs(%);
                                      2
(d19)                     x  + 2 x + 1
(c20) rhs(%);
(d20)                       0
```

The command **first**(*exp*) returns the first part of the expression *exp*, and the command **last**(*exp*) returns the last part of the expression *exp*. The command **rest** **rest**(*exp*) returns the expression with its leading element removed. With an additional argument, **rest**(*exp*, *num*) returns the expression *exp* with the first *num* terms removed.

Display the first term in the expression *big_expr2*.

```
(c21) first(big_expr2);
(d21)                     - x y
```

Display the last term in the expression *big_expr2*.

```
(c22) last(big_expr2);
                     - %i cos(c - b + 12 a)
(d22)              %e
```

Display all the terms in the expression *big_expr2* except the first.

```
(c23) rest(big_expr2);
             1           4    2                    2    2
(d23) - ----------- + log (a x  + b x + c) + a0 sin (a x  + b)
             1
        sqrt(- + 1)
             y
                                                 - %i cos(c - b + 12 a)
                                             + %e
```

Display all the terms in the expression *big_expr2* except the first two.

```
(c24) rest(big_expr2, 2);
          4    2                  2    2          - %i cos(c - b + 12 a)
(d24) log (a x  + b x + c) + a0 sin (a x  + b) + %e
```

The commands **first**, **last**, and **rest** are also useful for dealing with lists, as shown on page 28. The commands **realpart** and **imagpart** return the real and imaginary parts of the specified expression, respectively. Examples of these commands appear on page 19.

The commands **coeff** and **ratcoefratcoef** (also known as **ratcoeff**) take as arguments an expression, a variable in the expression for which you want the coefficient, and optionally, a power to which the variable is raised. Both functions return the coefficient.

```
(c25) c_expr:a*x^2+b*x+c;
                                 2
(d25)                         a x  + b x + c
(c26) coeff(c_expr, x);
(d26)                         b
(c27) coeff(c_expr, x, 2);
(d27)                         a
(c28) ratcoef(c_expr, x);
(d28)                         b
(c29) ratcoef(c_expr, x, 2);
(d29)                         a
```

The function **ratcoef** expands and rationally simplifies the expression before finding the coefficient, and thus can produce answers different from **coeff**, which is purely syntactic.

```
(c30) rc_expr:(a*x+b)^2;
```

```
                                      2
(d30)                       (a x + b)


(c31) coeff(rc_expr, x);
(d31)                       0
(c32) ratcoef(rc_expr, x);
(d32)                       2 a b
```

The function **coeff** is acting on the expression $(ax+b)^2$ where $(ax+b)$ is seen as a single entity. On the other hand, **ratcoef** first expands the expression (as in (d33)) and then looks for coefficients of x in the expanded expression.

```
(c33) expand(rc_expr);
                             2 2             2
(d33)                       a  x  + 2 a b x + b
```

4.6 Using Trigonometric Functions

This section outlines macsyma's trigonometric functions. Table 4.6 summarizes the names of the circular and hyperbolic trigonometric functions and their inverses. The derivatives of all these functions are also provided.

Circular functions	Inverse circular functions	Hyperbolic functions	Inverse hyperbolic functions
sin	asin	sinh	asinh
cos	acos	cosh	acosh
tan	atan	tanh	atanh
cot	acot	coth	acoth
sec	asec	sech	asech
csc	acsc	csch	acsch

Table 4.3: Trigonometric Functions and their Inverses

4.6.1 Evaluating Trigonometric Functions

Macsyma always numerically evaluates trigonometric functions (such as **sin** and **cos**) that have floating-point arguments. To avoid introducing approximations prematurely, it does not do so automatically for trigonometric functions that have integer arguments. In cases where Macsyma can return an exact value, however, a number can result.

This section presents the following commands:

- **exponentialize** converts the given expression containing trigonometric functions to an exponential with complex variables.
- **demoivre** converts the given exponential with complex variables to a trigonometric function.

Several option variables control the evaluation of trigonometric functions. **numer**, **exponentialize**, and **%emode** are introduced in this section.

You can evaluate trigonometric functions with integer arguments numerically by setting the option variable **numer** to **true**.

Macsyma does not return a floating-point approximation for (c1).

```
(c1) sin(1);
(d1)                    sin(1)
```

But since (c2) simplifies to `0`, a number results.

```
(c2) sin(0);
(d2)                       0
```

Use **numer** to obtain the numeric value for `sin(1)`[1].

```
(c3) sin(1), numer:true;
(d3)                  0.841471
```

The system is aware of special values of trigonometric functions at points $n\pi/m$, where $m = 1, 2, 3, 4, 6, 12$ and n is an integer.

Define a function f in terms of the trigonometric function **sin**.

```
(c4) f(z):= sin(z)^2 + 1;
                                  2
(d4)                f(z) := sin (z) + 1
```

Evaluate f at $z = x + 1$.

```
(c5) f(x + 1);
                       2
(d5)                sin (x + 1) + 1
```

Similarly, you can evaluate the following expression at the point $x = \pi/3$.

```
(c6) cos(x)^2 - sin(x)^2;
                          2         2
(d6)                   cos (x) - sin (x)
(c7) %, x=%pi/3;
                            1
(d7)                      - -
                            2
```

You need not convert these expressions to floating point immediately. Macsyma can perform many symbolic operations on expressions involving trigonometric functions. For example, you can differentiate and integrate them (see Chapter 6 for details).

[1]Some implementations of Macsyma return 0.8414709848078965.

When you set the option variable **exponentialize** to **true**, subsequent computations convert trigonometric functions to exponentials with complex variables. You can also use the command **exponentialize**(*exp*), which performs the same transformation on a given expression.

```
(c8) exponentialize:true;
(d8)                    true
(c9) sin(x);
                                %i x     - %i x
                          %i (%e     - %e      )
(d9)                    -  ----------------------
                                    2
(c10) exponentialize:false;
(d10)                   false
(c11) tan(x)+%i*cos(y)-sin(z);
(d11)               - sin(z) + %i cos(y) + tan(x)
```

Find the real and imaginary parts of the expression *t_expr*.

```
(c12) t_expr:exponentialize(%);
            %i z     - %i z          %i y     - %i y
      %i (%e     - %e      )   %i (%e     + %e      )
(d12) ---------------------- + ----------------------
                   2                      2
                                                 %i x     - %i x
                                           %i (%e     - %e      )
                                         - ----------------------
                                                %i x     - %i x
                                              %e     + %e
(c13) imagpart(t_expr);
(d13)                  cos(y)
(c14) realpart(t_expr);
                       sin(x)
(d14)                  ------ - sin(z)
                       cos(x)
```

To convert back to exponentials with complex variables back to trigonometric functions, use the command **demoivre**(*exp*).

```
(c15) demoivre(t_expr);
                                              sin(x)
(d15)               - sin(z) + %i cos(y) + ------
                                              cos(x)
```

Exponentials with complex arguments of the form $n\pi/m$, where $m = 1, 2, 3, 4, 6, 12$ and n is an integer, are transformed into algebraic numbers if the option variable **%emode** is set to **true** (the default).

```
(c16) %emode;
```

```
(d16)              true
(c17) t_expr, x = %pi;

                        %i z     - %i z          %i y     - %i y
                  %i (%e     - %e      )   %i (%e     + %e      )
(d17)             ---------------------- + ----------------------
                             2                          2
```

4.6.2 Expanding and Simplifying Trigonometric Expressions

You can expand expressions involving trigonometric functions. This section presents the following commands:

- **trigexpand** expands expressions that contain trigonometric and hyperbolic functions of sums of angles and of multiple angles.
- **trigreduce** combines products and powers of the trigonometric and hyperbolic functions for a specified variable and tries to eliminate these functions when they occur in the denominator.
- **trigsimp** converts expressions containing functions such as **tan** and **sec** to contain **sin**, **cos**, **sinh**, and **cosh** instead, so that **trigreduce** can further simplify the expressions.

The option variables **trigexpand**, **trigexpandplus**, **trigexpandtimes**, and **halfangles** are also described in this section.

The command **trigexpand**(*exp*) converts trigonometric functions with arguments of the form nx (where n is an integer) to the form x in the expression *exp*. Setting the option variable **trigexpand** to **true** causes full expansion of all expressions containing sines and cosines.

Expand an expression involving trigonometric functions

```
(c1) t_expr1:sin(2*x + y) + cos(2*a);
(d1)             sin(y + 2 x) + cos(2 a)
(c2) trigexpand(t_expr1);
                                                  2          2
(d2)       cos(2 x) sin(y) + sin(2 x) cos(y) - sin (a) + cos (a)
```

Setting the option variable **trigexpand** to **true** causes the full expansion of sines and cosines in *t_expr1*

```
(c3) trigexpand(t_expr1), trigexpand:true;
         2         2                                          2
(d3) (cos (x) - sin (x)) sin(y) + 2 cos(x) sin(x) cos(y) - sin (a)
                                                                2
                                                           + cos (a)
```

The option variable **trigexpandplus** controls the sum rule for **trigexpand**. By default, **trigexpandplus** is **true**, so Macsyma expands sums like `sin(x+y)`. Similarly, the option variable **trigexpandtimes** controls the product rule for **trigexpand**. By default, **trigexpandtimes** is **true**, so that Macsyma expands products like `sin(2*y)`.

To simplify half angles in trigonometric expressions, set the option variable **halfangles** to **true**.

```
(c4) t_expr2:sin(2*x) + cosh(y-z) + tan(yz/2);
```

```
                                  yz
(d4)             cosh(z - y) + tan(--) + sin(2 x)
                                   2
(c5) trigexpand(t_expr2);
                                                     yz
(d5)  - sinh(y) sinh(z) + cosh(y) cosh(z) + tan(--) + 2 cos(x) sin(x)
                                                      2
```

Locally bind **trigexpandtimes** to **false** to inhibit the expansion of products in *t_expr2*.

```
(c6) trigexpand(t_expr2), trigexpandtimes:false;
                                                        yz
(d6)     - sinh(y) sinh(z) + cosh(y) cosh(z) + tan(--) + sin(2 x)
                                                         2
```

Locally bind **trigexpandplus** to **false** to inhibit the expansion of sums in *t_expr2*.

```
(c7) trigexpand(t_expr2), trigexpandplus:false;
                               yz
(d7)          cosh(z - y) + tan(--) + 2 cos(x) sin(x)
                                2
```

Expand the expression, inhibiting expansion of sums and products, and simplifying half angles.

```
(c8) trigexpand(t_expr2),
     trigexpandtimes:false,
     trigexpandplus:false,
     halfangles:true;
                            1 - cos(yz)
(d8)           cosh(z - y) + ----------- + sin(2 x)
                              sin(yz)
```

Using the command **trigreduce**(*exp*, *var*), you can perform the inverse operation of **trigexpand** by converting products and powers of trigonometric functions into functions with multiple angles. If you do not specify a variable *var*, **trigreduce** uses all the variables in the expression.

```
(c9) t_expr3:trigexpand(sin(2*z) + sin(2*y));
(d9)             2 cos(z) sin(z) + 2 cos(y) sin(y)
(c10) trigreduce(t_expr3);
(d10)              sin(2 z) + sin(2 y)
(c11) trigreduce(t_expr3, z);
(d11)              sin(2 z) + 2 cos(y) sin(y)
(c12) trigreduce(t_expr3, y);
(d12)              2 cos(z) sin(z) + sin(2 y)
```

The command **trigsimp**(*exp*) uses the identity rules

$\sin^2 x + \cos^2 x = 1 \qquad \text{and} \qquad \cosh^2 x - \sinh^2 x = 1$

to convert expressions containing functions such as **tan** and **sec** to contain **sin**, **cos**, **sinh**, and **cosh** instead, so that **trigreduce** can further simplify the expressions.

Consider the expression below:

```
(c13) t_expr4:(1-sin(x))*(sec(x) + tan(x))
         -cos(x) + (cosh(x)^2 - sinh(x)^2)^3 - 1;
                                                2          2    3
(d13) (1 - sin(x)) (tan(x) + sec(x)) + (cosh (x) - sinh (x))  - cos(x) - 1
```

You can use either of two methods to simplify this expression.

Method 1

```
(c14) trigsimp(t_expr4);
(d14)                   0
```

Method 2

```
(c15) ratsubst(cosh(x)^2 - 1, sinh(x)^2, t_expr4);
(d15)      (1 - sin(x)) tan(x) - sec(x) sin(x) + sec(x) - cos(x)
(c16) subst(sin(x)/cos(x), tan(x), %);
           (1 - sin(x)) sin(x)
(d16)      ------------------- - sec(x) sin(x) + sec(x) - cos(x)
                  cos(x)
(c17) trigreduce(ratsimp(%));
(d17)                   0
```

Alternatively, you can reduce *t_expr4* as follows.

```
(c18) t_expr4, exponentialize, ratsimp;
(d18)                   0
```

4.7 Evaluating Summations

This section explains how you can perform summations using the following commands:

- **sum** performs a summation of the given values as an index ranges over specified values from low to high.
- **nusum** performs the indefinite summation of an expression with respect to a specified variable.
- **sumcontract** combines all sums of an addition that have upper and lower bounds that differ by constants.
- **intosum** takes all factors by which a summation is multiplied, then puts them inside the summation.

This section also provides examples of the option variables **simpsum**. To obtain a sum, you can use the command **sum sum**(*exp*, *index*, *low*, *high*) where *exp* is any Macsyma expression, *index* is the index of summation, and *low* and *high* are the lower and upper limits of the sum, respectively.

Obtain the sum of i^2 for $i =$ `1, 2, ..., 5`

```
(c1) sum(i^2, i, 1, 5);
(d1)                    55
```

You can obtain the same result using the **for** statement (see page 166).

```
(c2) for i thru 5 do (s:s + i^2);
(d2)                   done
(c3) s;
(d3)                  s + 55
```

You can evaluate the unbound variable s at $s =$ `0` as follows.

```
(c4) ev(%,s = 0);
(d4)                    55
```

Alternatively, you can re-evaluate (d3). Macsyma evaluates all the variables in it and re-executes all function calls.

```
(c5) ev(d3);
(d5)                  s + 110
```

If you set **simpsum** to **true**, and the lower and upper limits of the sum do not differ by an integer, **sum** simplifies its result. Sometimes this can produce a closed form.

To obtain an indefinite summation of a Macsyma expression use the command **nusum**(*exp*, *var*, *low*, *high*). This command performs indefinite summation of the expression *exp* with respect to the variable *var*, where *low* and *high* are the lower and upper limits of the sum, respectively.

Macsyma also lets you define expressions containing sums that are not evaluated or summable in closed form. Do this by placing a single quote before **sum** or by supplying an undefined function as the *exp* argument to **sum**.

Prevent evaluation of the sum using the single quote.

```
(c6) y='sum(a[i]*x^i, i, 0, 6);
                      6
                     ====
                     \         i
(d6)                  >    a  x
                     /      i
                     ====
                     i = 0
(c7) fro:ev(%, sum);
```

```
                   6       5       4       3       2
(d7)      y = a  x  + a  x  + a  x  + a  x  + a  x  + a  x + a
               6       5       4       3       2       1      0
(c8) sum((x^i + y^i)*(x^i - y^i), i, 1, n);
             n
            ====
            \      i    i    i    i
(d8)         >    (x  - y ) (y  + x )
            /
            ====
            i = 1
```

Macsyma can perform the summation with **simpsum** set to **true**.

```
(c9) ev(%, sum, simpsum:true);
                    2 (n + 1)    2          2 (n + 1)    2
              abs(x)          - x    abs(y)          - y
(d9)          -------------------- - --------------------
                      2                       2
                     x  - 1                  y  - 1
```

Consider the following expression:

```
(c10) expr:i/(4*i^2 - 1)^2;
                          i
(d10)                -----------
                         2     2
                     (4 i  - 1)
```

Even with **simpsum** set to **true**, Macsyma cannot perform this summation.

```
(c11) sum(expr, i, 1, n), simpsum:true;
                 n
                ====
                \          i
(d11)            >    -----------
                /         2     2
                ====  (4 i  - 1)
                i = 1
```

Use the **nusum** command to successfully sum the expression *expr*.

```
(c12) nusum(expr, i, 1, n);
```

```
                          n (n + 1)
(d12)                   ------------
                                   2
                        2 (2 n + 1)
```

You can also use **sum** in conjunction with other Macsyma commands.

```
(c13) tobesum:'sum(cos(k*x)*k, k, 1, n);
                 n
                ====
                \
(d13)            >    k cos(k x)
                /
                ====
                k = 1
```

The following sequence of commands yields the desired result.

```
(c14) exponentialize(sum(sin(k*x), k, 1, n));
           n
          ====           %i k x     - %i k x
          \        %i (%e       - %e        )
(d14)      >    (- -------------------------)
          /                  2
          ====
          k = 1
(c15) ev(%, sum, simpsum:true);
           - %i (n + 1) x     - %i x           %i (n + 1) x     %i x
      %i (%e               - %e      )   %i (%e             - %e    )
(d15) ------------------------------- - ---------------------------
                     - %i x                         %i x
             2 (%e       - 1)               2 (%e     - 1)
(c16) ev(demoivre(%), ratsimp, trigreduce);
               sin(n x + x)     sin(n x)        sin(x)
(d16)          ------------ - ------------ - ------------
               2 cos(x) - 2   2 cos(x) - 2   2 cos(x) - 2
(c17) tobesum = diff(%, x);
```

```
         n
        ====
        \                  2 sin(x) sin(n x + x)   (n + 1) cos(n x + x)
(d17)    >    k cos(k x) = --------------------- + --------------------
        /                                2         2 cos(x) - 2
        ====                  (2 cos(x) - 2)
        k = 1
                                                  2
            2 sin(x) sin(n x)    n cos(n x)       2 sin (x)        cos(x)
          - ----------------- - ------------ - --------------- - ------------
                           2    2 cos(x) - 2                2   2 cos(x) - 2
             (2 cos(x) - 2)                    (2 cos(x) - 2)
```

The command **sumcontract**(*expr*) combines all sums of an addition expression *expr* that have upper and lower bounds that differ by constants. This results in an expression that contains one summation for each set of such summations, and also includes all appropriate extra terms that had to be extracted to form this sum. **sumcontract** combines all compatible sums and uses one of the indices from one of the sums if it can, then tries to form a reasonable index if it cannot use any of those supplied.

The command **intosum**(*expr*) take all factors by which a summation is multiplied, then puts them inside the summation. If the index is used in the outside expression, **intosum** tries to find a reasonable index, as it does for **sumcontract**.

The following example uses the power series method to solve a differential equation, illustrating some of the applications of the summation commands.

```
(c18) depends(y, x)$
(c19) eq:diff(y, x, 2) + diff(y, x) - 2*x*y;
                        2
                       d y   dy
(d19)                  --- + -- - 2 x y
                         2   dx
                       dx
```

The origin is a regular point, so the solution can be expressed as a convergent series of this form.

```
(c20) pseries:y = 'sum(a[n]*x^n, n, 0, inf);
                  inf
                  ====
                  \         n
(d20)              >    a  x
                  /      n
                  ====
                  n = 0
```

Substitute the equation above into the original differential equation and carry out the derivative.

```
(c21) ev(eq, pseries, diff);
```

```
              inf       inf                 inf
             ====      ====                ====
             \       n \           n - 1   \                    n - 2
(d21)  - 2x >   a  x + >    n a  x      +  >    (n - 1) n a  x
             /    n    /        n          /                 n
             ====      ====                ====
             n = 0     n = 0               n = 0
```

You can re-express the previous line as follows.

```
(c22) subst(1, n, part(%, 2, 1)) + substpart(2, %, 2, 3);
            inf          inf                     inf
            ====         ====                  ====
            \         n  \           n - 1     \                    n - 2
(d22) - 2x >     a  x +  >    n a  x       +  >    (n - 1) n a  x
            /      n     /        n            /                 n
            ====         ====                   ====
            n = 0        n = 2                  n = 0
                                                                       + a
                                                                          1
(c23) subst(2, n, part(%, 3, 1)) + substpart(3, %, 3, 3);
             inf         inf                  inf
            ====         ====                 ====
            \         n  \           n - 1    \                    n - 2
(d23) - 2x >     a  x +  >    n a  x      +  >    (n - 1) n a  x
            /      n     /        n           /                 n
            ====         ====                 ====
            n = 0        n = 2                n = 3
                                                                    + 2 a   + a
                                                                         2     1
```

Let $n = 2 + m$ in the second summation, $n = 3 + m$ in the third summation above.

```
(c24) part(%, [1, 4, 5])
    + changevar(part(%, 2), n = 2 + m, m, n)
    + changevar(part(%, 3), n = 3 + m, m, n);
```

```
          inf           inf
          ====          ====
          \         n   \       2                    m + 1
(d24) - 2x >    a  x  +  >    (m  + 5 m + 6) a      x
          /      n      /                     m + 3
          ====          ====
          n = 0         m = 0
                                   inf
                                   ====
                                   \                     m + 1
                                 +  >    (m + 2) a      x      + 2 a  + a
                                   /             m + 2              2    1
                                   ====
                                   m = 0
(c25) sol:sumcontract(intosum(%));
      inf
      ====
      \        2                    n + 1                   n + 1
(d25)  >    ((n  + 5 n + 6) a      x      + (n + 2) a      x
      /                      n + 3                   n + 2
      ====
      n = 0

                                                    n + 1
                                            - 2 a  x     ) + 2 a  + a
                                                 n              2    1
```

Equate the coefficient of each power of x to zero.

```
(c26) a2:solve(part(%, [2, 3]), a[2]);
                                a
                                 1
(d26)                   [a  = - --]
                          2     2
(c27) part(sol, 1, 1);
        2                    n + 1                   n + 1         n + 1
(d27) (n  + 5 n + 6) a      x      + (n + 2) a      x      - 2 a  x
                      n + 3                   n + 2             n
(c28) solve(%, a[n + 3]);
                           (n + 2) a      - 2 a
                                    n + 2      n
(d28)           [a      = - ---------------------]
                  n + 3          2
                                n  + 5 n + 6
(c29) first(%);
```

```
                         (n + 2) a      - 2 a
                                  n + 2      n
(d29)            a      = - ---------------------
                  n + 3           2
                                 n  + 5 n + 6
```

The first five coefficients are:

```
(c30) [ev(%, n = 3),
       ev(%, n = 2),
       ev(%, n = 1),
       ev(%, n = 0),
       first(a2)];
              5 a  - 2 a          4 a  - 2 a          3 a  - 2 a
                 5      3            4      2            3      1
(d30) [a  = - -----------, a  = - -----------, a  = - -----------,
        6          30        5          20        4          12
                                          2 a  - 2 a          a
                                             2      0          1
                                      a  = - -----------, a  = - --]
                                       3          6        2     2
```

So the solution is:

```
(c31) trunc(ev(fro, %, expand));
                          2       3       3       4       4       5
                       a  x    a  x    a  x    a  x    a  x    a  x
                        1       0       1       0       1       0
(d31) y = a  + a  x - ----- + ----- + ----- - ----- + ----- + -----
           0    1       2       3       6       12      8       60
                                        5          6          6
                                   3 a  x    7 a  x    17 a  x
                                      1         0          1
                                 - ------- + ------- + -------- + . . .
                                      40        360        720
```

Recall that this problem could also have been done with the **ode** command:

```
(c32) ode(eq, y, x, odeseries);
MANYTERM RELATION
                              2       3       3       4       4       5
                          a  x    a  x    a  x    a  x    a  x    a  x
                           1       0       1       0       1       0
(d32) [y = a  + a  x - ----- + ----- + ----- - ----- + ----- + -----
            0    1       2       3       6       12      8       60
                                                              5
                                                        3 a  x
                                                             1
                                                      - ------- + . . .]
                                                            40
```

4.8 Practice Problems

Using the commands that you have learned about in this chapter, solve the following problems. Answers appear on page 267.

Problem 1. Write a function that performs the following substitution:

```
   2            2
sin (x) = 1 - cos (x)
```

Your function should work for all x. Test the function on the expression `sin(y)^3`.

Problem 2. Write a function that performs the following substitution:

```
   2      1 - cos(2 x)
sin (x) = ------------
               2
```

Your function should work for all x, but it should not work for high terms. Test the function on the expressions `sin(y)^2` and `sin(y)^3`.

Problem 3. Show that the expression below reduces to zero.

```
      2    2              2    2              2    2           2    2
(sqrt(r  + a ) + a) (sqrt(r  + b ) + b)    sqrt(r  + b ) + sqrt(r  + a ) + b + a
---------------------------------------  - -------------------------------------
                 2                            2    2           2    2
                r                          sqrt(r  + b ) + sqrt(r  + a ) - b - a
```

Problem 4. Express

```
    1          3     2
(-------- - --------)  + (b + a) (d + c)
        4          3
 (y + x)    (z + y)
```

as

```
          6               9         1
- ---------------- + -------- + -------- + b d + a d + b c + a c
         4        3          6          8
  (y + x)  (z + y)    (z + y)    (y + x)
```

Problem 5. Consider the expression

```
(d + c) ((w + a) x + b)
```

Show that this expression can be expressed as each of the following:

a) `d w x + c w x + a d x + a c x + b d + b c`

b) `(d + c) (w + a) x + b (d + c)`

c) `d (w + a) x + c (w + a) x + b d + b c`

d) `((d + c) w + a d + a c) x + b d + b c`

e) `d ((w + a) x + b) + c ((w + a) x + b)`

f) `(d + c) w x + a (d + c) x + b (d + c)`

Problem 6. Consider the expression

```
d (w + a) x + c (w + a) x + b d + b c
```

Show that this expression can be expressed as each of the following:

a) `(d + c) (w x + a x + b)`

b) `(d + c) ((w + a) x + b)`

Problem 7. Consider the expression

```
log((b + a) d + (b + a) c) z + log((b + a) d + (b + a) c) y
                    + log((b + a) d + (b + a) c) x + w
```

Show that this expression can be expressed as each of the following:

a) `log((b + a) (d + c)) (z + y + x) + w`

(b) `log((b + a) d + (b + a) c) (z + y + x) + w`

Problem 8. Express

```
      1
------------
   2       2
log (x) - x
```

as

a)

```
         1                        1
--------------------- - ---------------------
2 log(x) (log(x) + x)   2 (x - log(x)) log(x)
```

b)

```
       1                  1
---------------- - ----------------
2 x (log(x) - x)   2 x (log(x) + x)
```

Problem 9. Express

```
   x + 1
-----------
sqrt(x) - 1
```

as

```
sqrt(x) (x + 1) + x + 1
-----------------------
     x - 1
```

Problem 10. Evaluate

```
 n
====
\
 >    k sin(k x)
/
====
k = 1
```

Problem 11 . Evaluate

```
 m
====
\      3  n
 >    n  3
/
====
n = 1
```

Chapter 5

Solving Equations

Macsyma has powerful capabilities for solving and obtaining roots of equations. This chapter presents the following commands:

- **solve** solves a system of simultaneous linear or nonlinear polynomial equations for the specified variable and returns a list of the solutions.
- **linsolve** solves a system of simultaneous linear equations for the specified variables and returns a list of the solutions.
- **taylor_solve** computes a series solution to a univariate system of equations.
- **nroots** finds the number of real roots in the specified real univariate polynomial in a given interval.
- **allroots** finds all the real and complex roots of a specified real polynomial, which must be univariate and can be an equation.
- **realroots** finds all of the real roots of a univariate polynomial within a specified tolerance.
- **roots** finds all of the roots of a univariate polynomial with real or complex coefficients and can be an equation.

This section also discusses the system variable **multiplicities** and the option variables **globalsolve**, **solvetrigwarn**, **algexact**, **solveexplicit**, **solveradcan**, and **rootsepsilon**.

Although the examples in this chapter introduce many commands and option variables, the scope of this document allows only a limited introduction to Macsyma's equation solving capabilities. For more information, consult the *Macsyma Reference Manual*.

The command **solve**(*exp*, *var*) solves the expression *exp* for a single variable *var*. You can use the form **solve**(*exp*) if *exp* is univariate. If *exp* is not an equation, Macsyma sets the expression equal to zero in order to solve it.

When you solve a polynomial with **solve**, Macsyma uses the **factor** command to factor it, if necessary. If the factors are of degree 4 or less, Macsyma applies standard formulas to generate the solution.

The system variable **multiplicities** contains a list of the multiplicities of the individual solutions returned by **solve**.

```
(c1) neq:x^3 - 5*x^2 + 7*x - 3 = 0;
                    3      2
(d1)               x  - 5 x  + 7 x - 3 = 0
```

```
(c2) ans:solve(neq, x);
(d2)                  [x = 3, x = 1]
```

Macsyma returns only two answers, but there should be three; the system variable **multiplicities** indicates that two of the solutions are $x = 1$.

```
(c3) multiplicities;
(d3)                  [1, 2]
```

Verify the second solution in (d2) by re-evaluating the equation *neq* with $x = 1$.

```
(c4) ev(neq, last(ans));
(d4)                   0 = 0
```

Since *e0* is not an equation, Macsyma sets it equal to zero to solve it.

```
(c5) e0:-x0^3 + 2*x0^2 + x0 - 2;
                           3       2
(d5)                   - x0  + 2 x0  + x0 - 2
```

You need not specify the variable to solve for, since *e0* is univariate.

```
(c6) sol_4_x0:solve(e0);
(d6)              [x0 = 2, x0 = - 1, x0 = 1]
```

5.1 Solving Linear Equations

To solve a system of simultaneous linear equations, you can use the command **linsolve**([eqn_1, ..., eqn_n], [var_1, ..., var_n]), where the first list gives the equations to be solved and the second list gives the unknowns to solve for. Note that **linsolve** does not check to make sure the equations are linear.

Consider this system of linear equations.

```
(c1) list_of_eqs:[eq1:3*zz + b*yy + a*xx - 2, eq2:4*zz + 2*yy - a = 12,
             eq3:90*zz - 12*yy - 3*a = 45]$
(c2) list_of_vars:[xx, yy, zz]$
```

Solve these equations for the variables *xx*, *yy*, and *zz*.

```
(c3) linsolve(list_of_eqs, list_of_vars);
              (13 a + 150) b + 9 a + 41       13 a + 150       3 a + 39
(d3)  [xx = - -------------------------, yy = ----------, zz = --------]
                         38 a                     38              38
```

In general the "=" operator does not assign a value to a variable, so *xx*, for example, is unbound.

```
(c4) xx;
```

```
(d4)                        xx
```

If you set the option variable **globalsolve** to **true**, Macsyma can bind the variables being solved for to their corresponding solutions. If a variable has multiple solutions, however, setting **globalsolve** to **true** has no effect.

To bind xx, yy, and zz to the solutions that **linsolve** returns, set **globalsolve** to **true**.

```
(c5) linsolve(list_of_eqs, list_of_vars), globalsolve:true;
              (13 a + 150) b + 9 a + 41       13 a + 150       3 a + 39
(d5)  [xx : - -------------------------, yy : ----------, zz : --------]
                        38 a                      38              38
(c6) xx;
                       (13 a + 150) b + 9 a + 41
(d6)                 - -------------------------
                                 38 a
```

5.2 Solving Non-Linear Equations

To solve a system of simultaneous (non)linear polynomial equations, use the command **solve** ([eq_1, ..., eq_n], [var_n, ..., var_n]), where the first list gives the equations to be solved and the second list gives the variables for which to solve. In the next example we set the variable **eqs** to a system of three nonlinear equations.

```
(c1) eqs:[x*y*z = 42,
       -z + y + x = -2,
       -3*z + 2*y + 3*x = -a];
(d1)     [x y z = 42, - z + y + x = - 2, - 3 z + 2 y + 3 x = - a]
```

Solve these nonlinear equations for the variables x, y, and z.

```
(c2) solve(eqs, [x, y, z]);
                  4      3       2                   2
             sqrt(a  - 20 a  + 148 a  - 312 a - 432) - a  + 10 a - 24
(d2)  [[x = --------------------------------------------------------,
                                   2 a - 12


                                   3       2                  2
                 sqrt(a - 6) sqrt(a  - 14 a  + 64 a + 72) + a  - 10 a + 24
y = a - 6, z =  ---------------------------------------------------------],
                                   2 a - 12


             4      3       2                   2
        sqrt(a  - 20 a  + 148 a  - 312 a - 432) + a  - 10 a + 24
[x = - --------------------------------------------------------,
                                   2 a - 12
```

```
                                    3       2                 2
                    sqrt(a - 6) sqrt(a  - 14 a  + 64 a + 72) - a  + 10 a - 24
y = a - 6,z =- ---------------------------------------------------------]]
                                      2 a - 12
```

When you use **solve** on an equation containing trigonometric functions, Macsyma prints out a warning that some solutions might be lost. To inhibit this warning message, set the option variable **solvetrigwarn** to **false**.

Solve the trigonometric equation below for x; notice that Macsyma displays a warning message.

```
(c3) eq:asin(cos(3*x))*(x - 1);
(d3)              (x - 1) asin(cos(3 x))
(c4) solve(eq, x);
SOLVE is using arc-trig functions to get a solution.
Some solutions may be lost.
                          %pi
(d4)                 [x = ---, x = 1]
                           6
```

Solve the equation again, this time inhibiting the warning message.

```
(c5) solve(eq, x), solvetrigwarn:false;
                          %pi
(d5)                 [x = ---, x = 1]
                           6
```

You can set the option variable **solveexplicit** to **true** to inhibit **solve** from returning implicit solutions (that is, solutions of the form $f(x)$ = 0).

Setting the option variable **solveradcan** to **true** makes the **solve** command slower, but allows it to solve certain problems containing exponentials and logs.

```
(c6) eq:-2^(x + 1) + 4^x + 2;
                        x + 1    x
(d6)                 - 2      + 4  + 2
```

The **solve** command returns an implicit solution.

```
(c7) solve(eq, x);
                       x    x + 1
(d7)                 [4  = 2      - 2]
```

Inhibit the implicit result by binding **solveexplicit** to **true**.

```
(c8) solve(eq, x), solveexplicit:true;
(d8)                    []
```

The **solve** command can return an explicit result to this problem if you set **solveradcan** to **true**.

```
(c9) solve(eq, x), solveradcan:true;
                     log(1 - %i)      log(%i + 1)
(d9)            [x = -----------, x = -----------]
                       log(2)           log(2)
```

By setting the option variable **algexact** to **true**, you can request Macsyma to make every attempt to return exact solutions. Although this option does not guarantee that an exact solution will be found, Macsyma will return an approximate solution only when all else fails.

Solve the equations for x and y.

```
(c10) equations:[4*x^2 - y^2 = 12, x*y - x = 2];
                          2    2
(d10)              [4 x  - y  = 12, x y - x = 2]
```

The first solution, $x = 2$ and $y = 2$, is exact, but the other solutions are floating-point approximations

```
(c11) solutionsn:solve(equations, [x, y]);
(d11) [[x = 2, y = 2], [x = 0.52025944 %i - 0.13312405,
y = 0.076783754 - 3.6080031 %i], [x = - 0.52025944 %i - 0.13312405,
y = 3.6080031 %i + 0.076783754], [x = - 1.7337519, y = - 0.1535676]]
```

You can iteratively check the solutions that **solve** found (see the description of **for** on page 166)

```
(c12) for sol in solutionsn do print(ev(equations, sol, ratsimp, keepfloat:true));
[12.0 - 1.554312234475219e-15 %i = 12, 2.0 = 2]
[1.554312234475219e-15 %i + 12.0 = 12, 2.0 = 2]
[11.99999886045595 = 12, 1.999999906150018 = 2]
[12 = 12, 2 = 2]
(d12)                  done
```

Locally bind *algexact* to **true**, telling Macsyma to try harder to find exact solutions; since the results are long, suppress the display of the D-LINE.

```
(c13) solutionse:solve(equations, [x, y]), algexact:true$
```

You can iteratively check the solutions that **solve** found. Note that the result of `solutionse` given here is exact, while the result of `solutionsn` is only approximate.

```
(c14) for sol in solutionse do disp(ev(equations,sol,radcan));
                 [12 = 12, 2 = 2]
                 [12 = 12, 2 = 2]
                 [12 = 12, 2 = 2]
                 [12 = 12, 2 = 2]
(d14)                  done
```

5.3 Finding Numerical Roots

Macsyma can find the numerical roots of a univariate polynomial. The **allroots**(*poly*) command finds all the real and complex roots of the real polynomial *poly*, which must be univariate and can be an equation.

The **roots**(*poly*) command finds all the real and complex numerical roots of the real polynomial *poly*, which must be univariate and can be an equation. **roots** also accepts polynomials whose coefficients can have real or complex coefficients as well as constants such as %e or %pi. In many practical situations, **roots** is preferred over **allroots**.

The **realroots**(*poly*, *bound*) command finds all of the real roots of univariate polynomial *poly* within a tolerance of *bound*. You can specify *bound* as small as you like to achieve any desired accuracy. (If you do not specify *bound*, Macsyma uses the value of the option variable **rootsepsilon**, whose default is `1.0e-7`.)

The **nroots**(*poly*, *high*, *low*) command finds the number of real roots in the real univariate polynomial *poly* in the half open interval from *low* to *high*. If you do not specify *low* and *high*, **nroots** assumes that *low* is **minf** and *high* is **inf**.

The following example uses **nroots**, **realroots**, **allroots** and **roots** to find the numerical roots of an equation.

Macsyma has a routine for finding the roots of a univariate polynomial.

```
(c1) eq987:x^5/987 - 3*x + 1;
                          5
                         x
(d1)                    --- - 3 x + 1
                        987
```

Find the roots of *eq987* in the interval **inf** to **minf**.

```
(c2) nroots(eq987);
(d2)                      3
```

Find the real roots of *eq987*.

```
(c3) realroots(eq987), numer:true;
(d3)             [x = - 7.457738, x = 0.3333347, x = 7.290858]
```

For more accuracy, specify a smaller tolerance.

```
(c4) realroots(eq987, 1.0e-3), numer:true;
(d4)            [x = - 7.4575195, x = 0.3334961, x = 7.2905273]
```

Find all the real and complex roots of the real polynomial.

```
(c5) e_987:allroots(eq987);
(d5) [x = 0.3333347, x = - 7.457738, x = 7.379005 %i - 0.08322733,
                  x = - 7.379005 %i - 0.08322733, x = 7.290858]
```

Find all the real and complex roots of the real polynomial using **roots**.

```
(c6) roots(eq987);
(d6)  [x = 7.3790049600837d0 * %i - 0.08322732941358d0,
x =  - 7.3790049600837d0 * %i - 0.08322732941358d0,
x =  - 7.45773790754828d0, x = 7.29085784320352d0,
x = 0.33333472317194d0]
```

To find numerical roots of equal expressions, see **newton** in the *Macsyma Reference Manual.*

5.4 Finding Approximate Symbolic Solutions

The example in Section 5.3 found the numerical roots of the polynomial *eq987*. Sometimes, however, a symbolic solution can provide more insight into a problem. You can solve this problem symbolically using the command

taylor_solve (*eq*, *dep_var*, *ind_var*, *point*, *truncation*)

This command attempts to compute a series solution `y:`*dep_var* (dependent variable) to `e(y,x):`*eq*. The solution `y` will be a series at `p:`*point* in `v:`*ind_var* (independent variable) truncated to order `t:`*truncation*.

Note: The **taylor_solve** command does not currently handle differential equations and multivariate systems of equations.

Using **solve** doesn't work.

```
(c1) solve(eq:x^5*e-3*x+1,x);
                                   5
(d1)                    [0 = e x  - 3 x + 1]
```

You can obtain an approximate solution by using **taylor_solve**. The `k0` symbols in one of the solutions are the undetermined coefficients that satisfy the last equation, $k0^4 = 3$.

```
(c2) sol:taylor_solve(eq,x,e,0,[2]);
                 1    e                  k0    1             4
(d2)/T/     [[x = - + --- + . . .], [x = ---- - -- + . . ., k0  = 3]]
                 3   729                 1/4   12
                                        e
```

The next command re-expresses the solutions in a clearer form by substituting the undetermined coefficients back into the solution returned by **taylor_solve**. See Chapter 11 for a description of the **for** statement.

```
(c3) (temp1:first(sol),temp2:last(sol),undcoef:solve(last(last(sol)),k0),
for c in undcoef do temp1:endcons(ev(first(temp2),c),temp1),trunc(expand(temp1)));
                                  1/4                         1/4
          1    e              1   3    %i                 1   3
(d3) [x = - + --- + . . ., x = - -- + ------- + . . ., x = - -- - ---- + . . .,
          3   729             12      1/4                12    1/4
                                     e                        e
```

$$x = - \frac{1}{12} - \frac{3^{1/4}\ \%i}{e^{1/4}} + \ldots,\ x = - \frac{1}{12} + \frac{3^{1/4}}{e^{1/4}} + \ldots]$$

Macsyma also provides two option variables that you can use in conjunction with the **taylor_solve** command to control order selection and coefficient selection. These are introduced in practice problem 7, on page 75.

For more information on Macsyma's Taylor series facilities, see Section 6.4.

5.5 Practice Problems

Using the commands that you have learned about in this chapter, solve the following problems. Answers appear on page 272.

Problem 1. Find the four roots of the equation below:

$$x^4 - 7\ x^3 + 18\ x^2 - 20\ x + 8 = 0$$

Problem 2. Consider the following equation:

$$x^5 - \frac{x^2}{2940} - \frac{3}{9604000} = 0$$

a) Find the number of real roots.

b) Find all real roots.

c) Find all numerical roots.

Problem 3. Solve the three equations below for $(x,\ y,\ z)$:

$$z + y + x = 3$$
$$y\ z + x\ z + x\ y = - 18$$
$$z^3 + y^3 + x^3 = 189$$

Problem 4. Solve the two equations below for (x, y):

$$x^2\ y + y = 1$$
$$y - 2\ x = 4$$

Problem 5. Solve the three equations below for $(x,\ y,\ z)$

$$- z + 12\ y + a\ x = 3\ b$$

$$-5z - 4cy + x = 0$$
$$4y + x = c$$

a) with **globalsolve** set to **false**,

b) with **globalsolve** set to **true**, then remove the values bound to x, y, z.

Problem 6. Solve the equations below for (x, y, z):

$$y^2 + x^2 = 1$$
$$-4xz = 0$$

Problem 7. Macsyma provides two methods for selecting only certain solutions returned by **taylor_solve**. When set to **true**, the option variable **taylor_solve_choose_iorder** allows you you to supply the order of the particular solution you want interactively. Similarly, when the option variable **taylor_solve_choose_coef** is set to **true**, you can choose which of the multiple solutions for a coefficient equation you want.

Consider the following equation:

$$e^2 x^6 - e x^4 - x^3 + 2 x^2 + x - 2 = 0$$

For small e,

a) Determine the two-term expansions for the equation above.

b) Solve the equation with **taylor_solve_choose_order** set to **true**.

c) Solve the equation with **taylor_solve_choose_coef** set to **true**.

Chapter 6

Calculus

This chapter explains how to perform calculus operations using Macsyma, including differentiating and integrating expressions, taking limits, computing Taylor and Laurent series, solving ordinary differential equations, performing summations, and taking Laplace transforms.

Although the examples in this chapter introduce many commands and option variables, the scope of this document allows only a limited introduction to Macsyma's calculus capabilities. To learn more, consult the *Macsyma Reference Manual*.

6.1 Differentiating Expressions

Macsyma provides a facility for differentiating expressions. This section presents the following commands:

- **diff** differentiates an expression with respect to the given variables.
- **gradef** defines the gradients for each derivative of a function with respect to the function's arguments.
- **depends** declares functional dependencies for variables to be used by **diff**.
- **at** evaluates an expression with the variables assuming the values as specified for them in an equation or list of equations.

In addition, the option variable **derivabbrev** controls the display of derivatives as subscripts.

Use the command **diff**(*exp*, *var*, *num*) to differentiate the expression *exp num* times with respect to the variable *var*. Alternatively, you can use the command **diff**(*exp*, var_1, num_1, ..., var_n, num_n) to differentiate the expression *exp* with respect to each variable var_i num_i times.

```
(c1) expr:cosh(x*y);
(d1)                 cosh(x y)
```

The command **diff**(*exp*, *var*) differentiates an expression *exp* once with respect to a variable *var*.

```
(c2) diff(expr, x);
(d2)                   y sinh(x y)
```

Differentiate the expression *expr* twice with respect to *x*.

```
(c3) diff(expr, x, 2);
                             2
(d3)                        y  cosh(x y)
```

Differentiate the expression *expr* twice with respect x and once with respect to y.

```
(c4) diff(expr, x, 2, y, 1);
                          2
(d4)                   x y  sinh(x y) + 2 y cosh(x y)
```

To inhibit evaluation, precede the command with a single quote; Macsyma returns the "noun form".

```
(c5) 'diff(expr, x, 3);
                             3
                            d
(d5)                       --- (cosh(x y))
                             3
                           dx
```

To evaluate the derivative, use **ev**.

```
(c6) ev(%, diff);
                             3
(d6)                        y  sinh(x y)
```

You can differentiate predefined functions, such as f, as shown below:

```
(c7) f(y, z) := y^z;
                                        z
(d7)                        f(y, z) := y
(c8) diff(f(y, z), y, 2, z, 1);
                z - 2                    z - 2      z - 2
(d8)           y      log(y) (z - 1) z + y      z + y      (z - 1)
```

The command **diff**(*exp*) returns the total differential of the expression *exp*. The total differential is the sum of the derivatives of *exp* with respect to each of its variables times the function **del** of the variable. Macsyma does not offer any further simplification of **del**.

```
(c9) diff(expr);
(d9)            x sinh(x y) del(y) + y sinh(x y) del(x)
```

To declare functional dependencies for variables to be used by **diff**, use the command **depends**($funlist_1$, $varlist_1$, ..., $funlist_n$, $varlist_n$) where each list of functions $funlist_i$ depends on the corresponding list of variables $varlist_i$.

The system variable **dependencies** is a list of all the dependencies declared with **depends**. Use the command **remove**(*object*, *feature*) to remove a *feature*, such as **dependency**, from an *object*.

```
(c10) diff(u*v, x, 1, y, 2);
```

```
(d10)                    0
```

The functions u and v depend on the variables x and y.

```
(c11) depends([u, v], [x, y]);
(d11)              [u(x, y), v(x, y)]
(c12) diff(u*v, x, 1, y, 2);
          2        2            3             2      2         3
      du d v      d u  dv      d v       du  d v    d u dv    d u
(d12) -- --- + 2 ----- -- + u ------ + 2 -- ----- + --- -- + ------ v
      dx   2     dx dy dy          2     dy dx dy     2 dx        2
         dy                   dx dy                 dy       dx dy
```

Alternatively, you can state the dependencies explicitly as follows.

```
(c13) diff(w(x), x);
                      d
(d13)                 -- (w(x))
                      dx
```

Check to see the dependencies that are currently in effect.

```
(c14) dependencies;
(d14)              [u(x, y), v(x, y)]
```

The command **remove** can eliminate a previously declared dependency.

```
(c15) remove([u, v], dependency);
(d15)                   done
```

When you know only the first derivative of a function and you want to obtain derivatives of higher order, you can define gradients. The command **gradef**($f(x_1, \ldots, x_n), g_1, \ldots, g_n$) defines the gradients $g_1, \ldots, g_n$ for each derivative of the function f with respect to the function's arguments x_1 , . . . , x_n.

Define the gradients for each derivative of the function h with respect to its arguments, n and x.

```
(c16) gradef(h(n, x), 'diff(h(n, x), n), 2*n*h(n - 1, x));
(d16)                  h(n, x)



(c17) diff(h(n, x), n, 2);
                         2
                        d
(d17)                   --- (h(n, x))
                          2
                        dn
```

```
(c18) diff(h(n, x), x, 3);
(d18)             8 (n - 2) (n - 1) n h(n - 3, x)
```

The following example converts the Laplacian from Cartesian coordinates to cylindrical coordinates. Set up dependencies for the function u.

```
(c19) depends(u, [r, t, z]);
(d19)                  [u(r, t, z)]
```

Another acceptable format for defining gradients with **gradef** is shown below.

```
(c20) gradef(r, x, x/r);
(d20)                    r


(c21) diff(r, x);
                         x
(d21)                    -
                         r


(c22) (gradef(r, y, y/r), gradef(t, x, -y/r^2), gradef(t, y, x/r^2))$
(c23) diff(u, x, 2) + diff(u, y, 2) + diff(u, z, 2)$


(c24) ratsubst(r^2 - x^2, y^2, %);
                     2     2        2
                  2 d u   d u    2 d u     du
                 r  --- + --- + r  --- + r --
                      2     2        2     dr
                    dz    dt       dr
(d24)             ----------------------------
                               2
                              r


(c25) laplacian_cyl:multthru(%);
                             2
                          d u
                          ---         du
                   2       2    2     --
                  d u   dt     d u    dr
(d25)             --- + --- + --- + --
                    2    2      2    r
                  dz    r     dr
```

You can set the variable **derivabbrev** to **true** to display derivatives as subscripts.

```
(c26) derivabbrev:true$
```

```
(c27) laplacian_cyl;
                        u             u
                         t t           r
(d27)            u    + ---- + u    + --
                  z z     2     r r   r
                         r
```

6.2 Integrating Expressions

This section describes how you can integrate Macsyma expressions. Section 6.2.1 explains how you can perform indefinite integration, and Section 6.2.2 explains how you can perform definite integration.

6.2.1 Indefinite Integration

This subsection presents the following command:

- **integrate** finds the indefinite (or definite) integral of an expression with respect to a variable.

This section also introduces the option variables **logabs**, **intanalysis**, and **laplace_call**, which allow you to control some aspects of integration. See Section 6.1, page 85, for a detailed summary of the **intanalysis** option variable.

Use the command **integrate**(*exp*, *var*) to find the indefinite integral of the expression *exp* with respect to the variable *var*.

This section also introduces the option variable **logabs** which controls the use of absolute values in some integrals containing **log**.

```
(c1) i_expr1:1/y^(3/4)/(y - 1);
                                   1
(d1)                         ------------
                                       3/4
                             (y - 1) y
```

Find the indefinite integral of the expression *i_expr1* with respect to y.

```
(c2) integrate(i_expr1, y);
                      1/4                  1/4          1/4
(d2)          - log(y    + 1) - 2 atan(y   ) + log(y    - 1)
```

Macsyma uses different algorithms for **integrate** and **diff**, so you can check the result of integration with **diff**.

```
(c3) radcan(diff(%, y));
```

```
                                 1
(d3)                        -----------
                             7/4    3/4
                            y    - y
```

Macsyma can integrate expressions involving trigonometric functions and other predefined functions (including user-defined functions).

```
(c4) sin(x)^2*cos(x)^3;
                             3       2
(d4)                      cos (x) sin (x)


(c5) integrate(%, x);
                             5           3
                        3 sin (x) - 5 sin (x)
(d5)                  - ---------------------
                                 15


(c6) integrate(%th(2), x), exponentialize:true;
                   5 %i x        3 %i x
              %i %e         %i %e                 %i x           - %i x
(d6) - (- ----------- - ----------- + 2 %i %e     - 2 %i %e
                   5             3
                                                 - 3 %i x         - 5 %i x
                                            %i %e             %i %e
                                          + ------------- +  -------------)/32
                                                  3                 5
```

Define a function f, then integrate it.

```
(c7) f(x) := 1/sqrt(a*x^2 + b);
                                     1
(d7)                  f(x) := --------------
                                       2
                              sqrt(a x  + b)
```

Macsyma sometimes asks you about the sign of a quantity during integration; suitable responses are `p;` (positive), `n;` (negative), and `z;` (zero).

```
(c8) integrate(f(x), x);
Is   a   positive or negative?
p;
Is   b   positive or negative?
p;
```

```
                        sqrt(a) x
                  asinh(---------)
                        sqrt(b)
(d8)              ----------------
                      sqrt(a)
```

You can use the **assume** or **assume_pos** command to answer Macsyma's queries in advance.

```
(c9) assume(a > 0, b > 0);
(d9)                [a > 0, b > 0]
(c10) integrate(f(x), x);
                        sqrt(a) x
                  asinh(---------)
                         sqrt(b)
(d10)             ----------------
                      sqrt(a)
```

You can remove assumptions set up with the **assume** command using **forget**.

```
(c11)  forget(a > 0, b > 0);
(d11)               [a > 0, b > 0]
```

Note that **integrate** knows only about explicit dependencies; it is not affected by dependencies set up with the **depends** command.

To inhibit evaluation, precede the command with a single quote; Macsyma returns the "noun form".

```
(c12) i_expr2:'integrate(x/(a - x), x);
                     /
                     [   x
(d12)                I ----- dx
                     ] a - x
                     /
```

To evaluate the integral, use **ev**.

```
(c13) assume(a > x);
(d13)               [a > x]
(c14) ev(i_expr2, integrate);
(d14)             - a log(x - a) - x
```

The option variable **logabs** is **false** by default. When logs are generated during the integration of certain expressions, such as `integrate(1/x, x);`, this setting causes Macsyma to return the answers in terms of `log(...)`. Setting **logabs** to **true** causes Macsyma to return these answers in terms of `log(abs(...))`, where **abs** is the absolute value command.

```
(c15) ev(i_expr2, integrate, logabs:true);
(d15)             - x - a log(a - x)
```

When Macsyma cannot integrate an expression, it returns the unintegrated expression as a noun form.

```
(c16) integrate(sin(sin(x)), x);
                         /
                         [
(d16)                    I sin(sin(x)) dx
                         ]
                         /
```

6.2.2 Definite Integration

This section discusses exact symbolic and numerical definite integration. For floating point numerical integration, see Section 6.2.3. This subsection presents the commands

- **integrate** finds the indefinite (or definite) integral of an expression with respect to a variable.
- **ldefint** returns the definite integral of the specified expression by using the command **limit** (see page 89) to evaluate the indefinite integral of the expression.
- **changevar** makes the specified change of variable in all integrals occurring in the given expression.

This section also introduces the option variables **intanalysis** and **laplace_call**, which allow you to control some aspects of integration.

You can use the command **integrate**(*exp*, *var*, *high*, *low*) to find the definite integral of the expression *exp* with respect to the variable *var* from *low* to *high*.

```
(c1) integrate(log(1/x)/(1 + x)^2, x, 0, 1);
(d1)                    log(2)
```

You can use the option variable **intanalysis** to customize Macsyma's definite integrator. When **intanalysis** is **true**, Macsyma tries to determine whether an integral is absolutely convergent before performing the integration by checking for poles in the interval of integration. When **false**, **intanalysis** turns off this time-consuming check, assuming that absolute convergence is assumed until proven otherwise. Since this procedure can lead to wrong answers if poles exist, you should use the **false** setting with caution.

Table 6.1 summarizes the differences between setting **intanalysis** to **true** or **false**.

In the following example, the option variable **showtime** is set to **true** so that you can compare the relative speeds of the calculations, depending on the setting of **intanalysis**.

```
(c2) showtime:true$
Time= 1 msec
```

The first command below uses **intanalysis** set to **true**, the second uses it set to **false**. Notice that in this case both settings return the same answer, but a setting of **true** takes more time because the function **defint** checks for poles in the interval of integration. A setting of **false** bypasses this time consuming check, but can lead to incorrect results if there are poles. Use the **false** setting with caution.

```
(c3) integrate(1/(x^2 - a), x, 0, inf), intanalysis:true;
Is   a   positive, negative, or zero?
```

Setting of **intanalysis**	*Pros*	*Cons*
true	You can have greater confidence in the result of integration.	The calculation of the integral takes longer with the convergence check.
	The information returned by convergence testing might be of interest.	Macsyma might make several queries during integration about the signs of quantities.
	The range of parameters for which the result is valid can be returned.	An expression that can be integrated might be returned as a noun form instead, if Macsyma cannot determine its convergence.
		The algorithm for determination of convergence might not work in all cases.
false	Integration is faster.	Results must be used with care, since they could be wrong.
	Macsyma makes fewer queries about the signs of quantities.	All convergence information is lost.
	All integratable expressions are integrated.	Often the result returned is purely symbolic, giving no indication of the range of its validity.

Table 6.1: Summary of the **intanalysis** Option Variable

```
p;
Principal Value
Time= 24682 msecs
(d3)                    0
(c4) integrate(1/(x^2 - a), x, 0, inf), intanalysis:false;
Is   a   positive or negative?
p;
Time= 21211 msecs
(d4)                    0
```

When Macsyma cannot determine if there are poles in the interval of integration, it prints a message and continues with the integration as though **intanalysis** were **false**.

```
(c5) integrate(exp((%i*a - b)*x), x, 0, inf), intanalysis:true;
INTEGRATE could not determine whether there are poles in the
interval of integration.
Continuing...
Is a positive, negative, or zero?
p;
Time= 18253 msecs
                         b       %i a
(d5)                  ------- + -------
                       2    2    2    2
                      b  + a    b  + a
(c6) integrate(exp((%i*a - b)*x), x, 0, inf), intanalysis:false;
Is   a   positive, negative, or zero?
p;
Time= 12570 msecs
                         b       %i a
(d6)                  ------- + -------
                       2    2    2    2
                      b  + a    b  + a
(c7) showtime:false$
```

If you are convinced that a function is best integrated through indefinite integration, then take limits at the end points using the command **ldefint**(*exp*, *var*, *low*, *high*). The function **ldefint** integrates the expression *exp* indefinitely with respect to the variable *var* and takes limits at the endpoints *low* and *high*. See page 89.

```
(c8) ldefint(exp((%i*a - b)*x), x, 0, inf);
                         1
(d8)                 --------
                      b - %i a
(c9) forget(b > 0);
(d9)                  [b > 0]
```

You can set the option variable **laplace_call** to control the extent to which the integrator attempts to use Laplace transform techniques to solve problems. By default, **laplace_call** is **true**.

- When **laplace_call** is **false**, Macsyma does not use Laplace transform techniques.
- When **laplace_call** is **true**, Macsyma applies Laplace transform techniques only to actual Laplace transforms, that is, problems of the form `integrate(exp(-s*x)*f(x),x,0,inf)`.
- When **laplace_call** is **all**, even problems that are not in Laplace transform format are forced into it, and the Laplace transform techniques are applied. The transformation

  ```
  integrate(f(x), x, 0, inf)
   limit(integrate(exp(-s*x)*f(x), x, 0, inf), s, 0, plus)
  ```

  is applied whenever both sides are defined.

For more information about Laplace transforms, see Section 6.7, page 107. For information about the **limit** command, see Section 6.3, page 89.

Consider the following examples, which illustrate the use of the various settings of **laplace_call**. In the next two commands, notice that Macsyma cannot solve the problem with **laplace_call** set to **false**.

```
(c10) integrate(t*exp(-t)/(t + 1), t, 0, inf), laplace_call:false;
                  inf
                 /        - t
                 [    t %e
(d10)            I    ------- dt
                 ]     t + 1
                 /
                  0
(c11) integrate(t*exp(-t)/(t + 1), t, 0, inf), laplace_call:true;
(d11)             1 - %e gamma(0, 1)
```

In the next three commands, notice the settings of the option variables **intanalysis** and **laplace_call** required to solve the problem.

```
(c12) integrate((cos(t*x) - cos(x))/x, x, 0, inf);
Is   t   positive, negative, or zero?
p;
Integral is not absolutely convergent. Maybe you
want to try the computation with INTANALYSIS:FALSE.
(d12)                  []
(c13) integrate((cos(t*x) - cos(x))/x, x, 0, inf), intanalysis:false;
                inf
                /
                [    cos(t x) - cos(x)
(d13)           I    ----------------- dx
                ]            x
                /
                0
(c14) integrate((cos(t*x) - cos(x))/x, x, 0, inf), laplace_call:all,
      intanalysis:false;
```

```
(d14)                - log(t)
```

The command **changevar**(*exp*, *f(x, y)*, *y*, *x*) makes the change of variable given by *f(x, y)* `= 0` in all integrals occurring in the expression *exp* with integration with respect to x. The variable y is the new variable.

```
(c15) 'integrate(f(tau)*g(t - tau), tau, 0, t);
                  t
                 /
                 [
(d15)            I  g(t - tau) f(tau) dtau
                 ]
                 /
                 0
(c16) changevar(%, u = t - tau, u, tau);
                      0
                     /
                     [
(d16)              - I  f(t - u) g(u) du
                     ]
                     /
                     t
```

Some Macsyma implementations support a more efficient way of doing numerical integration with **romberg**. Refer to **romberg** in the *Macsyma Reference Manual* for more information.

6.2.3 Numerical Integration

Macsyma has three commands for floating point numerical integration.

- **quadratr** extrapolated Gaussian quadrature.
- **romberg** Romberg integration.
- **quanc8** Newton-Cotes quadrature.

Two other functions, **simpson** and **traprule** implement Simpson's rule and the trapezoidal rule, and are useful mainly for instructional purposes.

The **quadratr** command is the most robust integrator. It can handle integrands which become singular or are ill-behaved. It tends to be slower on most problems.

The **romberg** command is best for well-behaved integrands.

The **quanc8** command is often useful for highly oscillatory integrands.

For more information on these functions, refer to the *Macsyma Reference Manual* .

Here are some examples. Note that **quadratr** gives the best answer in all cases.

```
(c1) integrate(log(x), x, 0, 1);
(d1)                                   -1
(c2) quadratr(log(x), x, 0, 1);
```

```
(d2)                          - 0.99999960994452d0
(c3) errcatch(romberg(log(x), x, 0, 1));
LOG(0) has been generated.
(d3)                                  [ ]
(c4) quadratr(sin(1/x), x, 0.0001, 1);
(d4)                            0.50638436625272d0
(c5) quanc8(sin(1/x), x, 0.0001, 1);
(d5)                                  0.4083
(c6) errcatch(romberg(sin(1/x), x, 0.0001, 1));
ROMBERG failed to converge.
(d6)                                   [ ]
```

6.3 Taking Limits

This section presents the following commands:

- **limit** finds the limit of an expression as a given real variable approaches some value, such as infinity.
- **tlimit** is similar to **limit**, except that it uses Taylor series (see page 91) when possible.

This section also includes an example using the option variable **tlimswitch**. In addition, the command **ldefint**, which integrates an expression indefinitely with respect to a variable and takes limits at the endpoints, was introduced in Section 6.2.2, page 86.

The command **limit**(*exp*, *var*, *value*, *direction*) finds the limit of the expression *exp* as the real variable *var* approaches the given *value* from some *direction*. If you do not specify *direction*, **limit** computes a bidirectional limit. However, you can specify a *direction* with **plus** to indicate a limit from above, or **minus** to indicate a limit from below.

If the limit is undefined, **limit** returns **und**. If the limit is indefinite, but bounded, **limit** returns **ind**. Both **und** and **ind** are **special symbols**.

Take the bidirectional limit of the expression as x goes to infinity.

```
(c1) limit((x + exp(x) + exp(2*x))^(1/x), x, inf);
                          2
(d1)                    %e
(c2) expr:1/(exp(1/x) + 1);
                         1
(d2)                 ---------
                       1/x
                     %e    + 1
```

Take the limit of the expression *expr* as x goes to zero from above.

```
(c3) limit(expr, x, 0, plus);
(d3)                     0
```

Take the limit of the expression *expr* as x goes to zero from below.

```
(c4) limit(expr, x, 0, minus);
(d4)                     1
```

The bidirectional limit of the expression *expr* as x goes to zero is undefined.

```
(c5) limit(expr, x, 0);
(d5)                    und
```

The bidirectional limit of the expression `sin(1/x)` as x goes to zero is indefinite, but bounded.

```
(c6) limit(sin(1/x), x, 0);
(d6)                    ind
```

To inhibit evaluation, precede the command with a single quote. Macsyma returns the "noun form".

```
(c7) 'limit(tan(x)^cos(x),x,%pi/2);
                                cos(x)
(d7)            limit    tan(x)
                      %pi
                 x -> ---
                       2
```

To evaluate the limit, use **ev**.

```
(c8) ev(%,limit);
(d8)                     1
```

You can also take a limit using the command **tlimit**(*exp*, *var*, *value*, *direction*), which instructs Macsyma to use the **taylor** command when possible (see page 91). Alternatively, you can set the option variable **tlimswitch** to **true** to indicate that the **limit** command should try to use **taylor**.

```
(c9) tlimit((sin(tan(x)) - tan(sin(x)))/x^7, x, 0);
                         1
(d9)                   - --
                         30
(c10) limit((sin(tan(x)) - tan(sin(x)))/x^7, x, 0), tlimswitch:true;
                         1
(d10)                  - --
                         30
```

6.4 Computing Taylor and Laurent Series

Macsyma has a very powerful package for computing Taylor series, or more generally, Laurent series. This section presents the following commands:

- **taylor** computes a Taylor series by expanding an expression in a given variable around a specified point.
- **taylorinfo** returns information about the Taylor expansion of the specified expression.

Other Taylor-related commands include **taylor_solve**, introduced in Section 5.4, and **tlimit** in Section 6.3.

At the end of this section, examples of multivariate Taylor series expansion and asymptotic Taylor series expansion are also presented.

To compute a Taylor series, use the command **taylor**(*exp, var, point, power*), where *exp* is the expression you want to expand in the variable *var* around the point *point*. Macsyma generates the terms through (*var* - *point*)$^{\text{power}}$. Display lines containing Taylor series representations are labeled with `/T/`.

You can manipulate the truncated Taylor series; Macsyma recalculates automatically.

```
(c1) taylor(sin(x)/x, x, 0, 4);
                           2    4
                          x    x
(d1)/T/               1 - -- + --- + . . .
                          6    120
(c2) %^2;
                           2      4
                          x    2 x
(d2)/T/               1 - -- + ---- + . . .
                          3     45
```

The answer in (d1) above gives an indirect value of `1` for the limit of `sin(x)/x` at `0`. You can also use `limit(sin(x)/x, x, 0);` to obtain the same value.

The **taylorinfo**(*exp*) command returns information about the Taylor expansion of the expression *exp*, if *exp* is a Taylor series; otherwise, **taylorinfo** returns **false**.

This command indicates that the last expression is a Taylor series expansion, is in x, is about 0, and is up to the fourth order.

```
(c3) taylorinfo(%);
(d3)                    [[x, 0, 4]]
```

The next example computes the Taylor series of an expression with respect to x about b to the second order.

```
(c4) taylor(exp(1/sin(a*x)), x, b, 2);
```

```
                                  1
                    1          --------
                 --------      sin(a b)
                 sin(a b)   (%e         cos(a b) a) (x - b)
(d4)/T/       %e          - ------------------------------
                                        2
                                     sin (a b)
          1                         1
       --------                  --------
       sin(a b)    3             sin(a b)    2
 + ((%e         sin (a b) + 2 %e          cos (a b) sin(a b)
        1
     --------
     sin(a b)    2        2         2       4
 + %e         cos (a b)) a ) (x - b) /(2 sin (a b)) + . . .
```

Macsyma also has capabilities for finite Laurent series expansion.

```
(c5) taylor(cot(%pi*z)/(4*z^2 - 1)^5, z, 1/2, 2);
                                             3                 3
               %pi           5 %pi        %pi  + 45 %pi   5 %pi  + 105 %pi
(d5)/T/ - ------------- + ------------- - ------------- + ----------------
                    1 4             1 3             1 2               1
          1024 (z - -)    1024 (z - -)    3072 (z - -)      3072 (z - -)
                    2               2               2                 2
                                      5         3                 1
        5         3             (2 %pi  + 35 %pi  + 378 %pi) (z - -)
   2 %pi  + 75 %pi  + 1050 %pi                                    2
 - -------------------------- + -----------------------------------
            15360                               3072
          7          5           3                   1 2
   (17 %pi  + 630 %pi  + 7350 %pi  + 66150 %pi) (z - -)
                                                     2
 - ---------------------------------------------------- + . . .
                        322560
```

Macsyma also allows you to perform multivariate Taylor series expansion, as shown below.

```
(c6) expr:sin(a*x + b*y)^-1;
                            1
(d6)                  --------------
                      sin(b y + a x)
 (c7) taylor(expr, x, 0, 1, y, 1, 2);
```

```
                                      2          2      2          2
          1     (cos(b) b) (y - 1)   ((2 cos (b) + sin (b)) b ) (y - 1)
(d7)/T/ ------ - ------------------ + ----------------------------------
        sin(b)          2                              3
                     sin (b)                      2 sin (b)
                     3                  2      2            2
               ((6 cos (b) + 5 cos(b) sin (b)) b  a) (y - 1)
 + . . . + (- ---------------------------------------------
                                   4
                              2 sin (b)
          2          2
   ((2 cos (b) + sin (b)) b a) (y - 1)   cos(b) a
 + ---------------------------------- - -------- + . . .) x + . . .
                3                           2
             sin (b)                     sin (b)
```

As shown in the examples below, Macsyma supports two alternative syntaxes for specifying the terms $(var - point)^{\text{power}}$.

```
(c8) taylor(expr, x, 0, 3, y, 0, 3);
                                    2
         1    b y               a   a                   a             2
(d8)/T/ --- + --- + . . . + (- - ----- + . . .) x + (----- + . . .) x
        b y    6                6  2  2                3  3
                                  b  y                b  y
                                  3
                                 a            3
                           + (- ----- + . . .) x  + . . .
                                4  4
                               b  y
(c9) taylor(expr, [x, y], 0, 3);
            1       a x + b y
(d9)/T/ --------- + ---------
        a x + b y       6
             3  3        2    2       2    2       3  3
          7 a  x  + 21 b a  y x  + 21 b  a y  x + 7 b  y
        + ---------------------------------------------- + . . .
                                360
```

Macsyma also allows you to perform asymptotic Taylor series expansion, as shown below.

```
(c10) exp:1/(1 - 1/x);
                             1
(d10)                      -----
                               1
                           1 - -
                               x



(c11) taylor(exp, [x, 0, 5, 'asymp]);
                         1   1    1    1    1
(d11)/T/             1 + - + -- + -- + -- + -- + . . .
                         x    2    3    4    5
                             x    x    x    x
(c12) subst(e, 1/x, exp);
                             1
(d12)                      -----
                           1 - e
(c13) taylor(%, e, 0, 5);
                              2    3    4    5
(d13)/T/             1 + e + e  + e  + e  + e  + . . .
(c14) subst(1/x, e, %);
                     1   1    1    1    1
(d14)                - + -- + -- + -- + -- + 1
                     x    2    3    4    5
                         x    x    x    x
```

To find the leading behavior of the integral

```
                          x
                         /     - t
                         [   %e
                         I  ------- dt
                         ]  sqrt(t)
                         /
                        0
```

as $x \to 0+$, you can use the **taylor** command.

```
(c15) taylor(exp(-t)/sqrt(t), t, 0, 4);
                                         3/2    5/2    7/2
                     1                  t      t      t
(d15)/T/          ------- - sqrt(t) + ---- - ---- + ---- + . . .
                  sqrt(t)              2      6      24
```

The command **trunc**(*exp*) displays an expression as if its sums were truncated Taylor series.

```
(c16) trunc(expand(integrate(%, t, 0, x)));
Is   x   positive, negative, or zero?
p;
                             3/2    5/2    7/2    9/2
                          2 x      x      x      x
(d16)          2 sqrt(x) - ------ + ---- - ---- + ---- + . . .
                             3      5      21     108
```

6.5 Solving Ordinary Differential Equations (ODEs)

6.5.1 Symbolic Solutions of ODEs

Macsyma has a symbolic differential equation solver. This section presents the following commands:

- **ode** solves first and second order ordinary differential equations.
- **ic1** sets an initial condition for first order initial value problems.
- **ic2** sets an initial condition for second order initial value problems.
- **bc2** sets a boundary condition for second order boundary value problems.

In addition, this section discusses the system variables **method**, **intfactor**, **odeindex**, and **yp**, and the option variable **odetutor**. See **odelinsys**, Section 6.7 for solving linear systems of ODEs.

The command **ode**(*equation*, *depvar*, *indvar*) solves first and second order ordinary differential equations, *equation* with dependent variable(s) *depvar* and independent variable(s) *indvar*. The **ode** command knows about several methods for solving ODEs. It attempts another method if a previous method fails to return a solution. When successful, **ode** returns either an explicit or implicit solution for the dependent variable.

If you set the option variable **odetutor** to **true**, **ode** prints out the name of each method as it attempts to solve an ODE. For more information about the **ode** command, and the methods it employs for solving ODEs, consult the *Macsyma Reference Manual*.

When solving an ODE, Macsyma sets several system variables that you can use to retrieve information about the solution:

- **method** denotes the method of solution used. See page 97 for an example of this variable.
- **intfactor** denotes any integrating factor used. See page 98 for an example of this variable.
- **odeindex** denotes the index for Bernouilli's method, or for the generalized homogeneous method. See page 99 for an example of this variable.
- **yp** denotes the particular solution for the variation of parameters technique. See page 99 for an example of this variable.

Macsyma provides commands to help you solve initial value problems and boundary value problems:

- The command **ic1**(*solution*, *xvalue*, *yvalue*) sets the initial condition for first order initial value problems. The argument *solution* is a general solution to a first order differential equation; *xvalue* is an equation for the independent variable, in the form $x = x_0$; and *yvalue* is an equation for the dependent variable, in the form $y = y_0$.

The **ic1** command returns an equation obtained by restricting the general solution *solution* according to the initial condition specified by *xvalue* and *yvalue*. See page 97 for an example of this command.

- The command **ic2**(*solution*, *xvalue*, *yvalue*, *derivativevalue*) sets the initial condition for second order initial value problems. The argument *solution* is a general solution to a second order differential equation; *xvalue* is an equation for the independent variable, in the form $x = x_0$; and *yvalue* is an equation for the dependent variable, in the form $y = y_0$; *derivativevalue* is an equation for the derivative of the dependent variable with respect to the independent variable evaluated at the point *xvalue*.

  The **ic2** command returns an equation obtained by restricting the general solution *solution* according to the initial conditions specified by *xvalue*, *yvalue*, and *derivativevalue*. See page 98 for an example of this command.

- The command **bc2**(*solution*, *xvalue1*, *yvalue1*, *xvalue2*, *yvalue2*) sets the boundary condition for second order boundary value problems. The argument *solution* is a general solution to a second order differential equation; *xvalue1* is an equation for the independent variable, in the form $x = x_0$; and *yvalue1* is an equation for the dependent variable, in the form $y = y_0$; *xvalue2* and *yvalue2* are equations for these variables at another point.

  The **bc2** command returns an equation obtained by restricting the general solution *solution* according to the conditions specified by *xvalue1*, *yvalue1*, *xvalue2*, and *yvalue2* together. See page 98 for an example of this command.

```
(c1) depends(y, x)$
```

Set **odetutor** to **true** to keep track of what is happening.

```
(c2) odetutor:true$
```

Macsyma can solve odes with symbolic coefficients.

```
(c3) eq:x*diff(y, x) + a*x*y^2 + 2*y + b*x = 0;
                     dy        2
(d3)               x -- + a x y  + 2 y + b x = 0
                     dx
```

The **%c** symbol in the result represents an arbitrary constant for first order solutions.

```
(c4) sol:ode(%, y, x);
Trying     ode2
Trying     nonlin
Trying     riccati
Trying     schmidt
                 b     1      1           b
FOUND    [sqrt(- -) - ---, - --- - sqrt(- -)]
                 a    a x    a x          a


(d4) y =
```

```
                          b     2 a sqrt(- b/a) x            b
  (2 %c b x + 2 %c sqrt(- -)) %e                  - a sqrt(- -) x - 1
                          a                                  a
- -------------------------------------------------------------------
                           b      2 a sqrt(- b/a) x
            2 %c a sqrt(- -) x %e                  - a x
                           a
```

As always, after you find a solution the variable **method** is set to the name of the method that succeeded.

```
(c5) method;
(d5)                 riccati
```

It is a good idea to check the solution.

```
(c6) ev(eq, sol, diff, radcan);
(d6)                  0 = 0
(c7) diff(y, x) = 2/%pi*x*y*(y - %pi);
                   dy   2 x y (y - %pi)
(d7)               -- = ---------------
                   dx         %pi
(c8) sol:ode(%, y, x);
Trying     ode2
                                          2
                  log(y - %pi) - log(y)   x
(d8)              --------------------- = -- + %c
                            2             2
(c9) method;
(d9)                separable
```

Use **ic1** for the initial value problem of a first order ODE.

```
(c10) ic1(sol, x = 0, y = y0);
                                                            2
           log(y - %pi) - log(y)   log(y0 - %pi) - log(y0) + x
(d10)      --------------------- = ---------------------------
                     2                          2


(c11) solve(logcontract(%),y);
                                %pi y0
(d11)             [y = - ------------------------]
                             2                  2
                            x                  x
                         (%e   - 1) y0 - %pi %e
```

Macsyma assumes the right side of the equation is zero if you do not specify it.

```
(c12) diff(y, x, 2) + y*diff(y, x)^3;
                          2
                         d y      dy 3
(d12)                    --- + y (--)
                           2      dx
                         dx
```

The **%k1** and **%k2** symbols in the result represent arbitrary constants for second order solutions.

```
(c13) sol:ode(%, y, x);
Trying     ode2
                    3
                   y  + 6 %k1 y
(d13)              ------------ = x + %k2
                        6
(c14) method;
(d14)                   freeofx
```

Use **ic2** to set the initial conditions for a second order ODE.

```
(c15) ratsimp(ic2(sol, x = 0, y = 0, 'diff(y,x) = 2));
                             3
                         2 y  - 3 y
(d15)                  - ---------- = x
                              6
```

Use **bc2** to set the boundary conditions for a second order ODE.

```
(c16) bc2(sol, x = 0, y = 1, x = 1, y = 3);
                        3
                       y  - 10 y       3
(d16)                  --------- = x - -
                           6           2
(c17) y + (2*x*y - %e^(-2*y))*'diff(y,x) = 0;
                                - 2 y  dy
(d17)               (2 x y - %e     ) -- + y = 0
                                       dx
(c18) ode(%,y,x);
Trying     ode2
                         2 y
(d18)               x %e    - log(y) = %c
(c19) method;
(d19)                    exact
```

When you solve an equation by means of an integrating factor, the variable **intfactor** is set to the integrating factor.

```
(c20) intfactor;
                         1
(d20)                    -
                         y
(c21) x^2*diff(y,x) + 2*x*y - y^3 = 0;
                   2 dy    3
(d21)             x  -- - y  + 2 x y = 0
                     dx
(c22) ode(%, y, x);
Trying     ode2
                              1
(d22)             y = ------------------
                             2         2
                      sqrt(---- + %c) x
                              5
                           5 x
(c23) method;
(d23)                bernoulli
```

When you find a solution for Bernoulli's equation or for a generalized homogeneous equation, the variable **odeindex** is set to the index of the equation.

```
(c24) odeindex;
(d24)                    3
```

After finding the solution to the homogeneous equation, the method of variation of parameters allows you to find the nonhomogeneous solution to the equation.

```
(c25) diff(y, x, 2) - 5*diff(y, x) + 6*y = 2*exp(x);
                  2
                 d y     dy              x
(d25)            --- - 5 -- + 6 y = 2 %e
                   2     dx
                 dx
(c26) ode(%, y, x);
Trying     ode2
                             3 x         2 x      x
(d26)             y = %k1 %e    + %k2 %e    + %e
(c27) method;
(d27)              variationofparameters
```

The variable **yp** denotes the particular solution for the technique of variation of parameters.

```
(c28) yp;
                          x
(d28)                   %e
```

The following example illustrates the use of **ode**'s optional **series** keyword to obtain a series solution.

```
(c29) eq:diff(y, x, 2)*2*x*(1 - x) + (1 - x)*diff(y, x) + 3*y;
                                  2
                                 d y           dy
(d29)               2 (1 - x) x --- + (1 - x) -- + 3 y
                                   2           dx
                                 dx


(c30) ode(eq, y, x);
                                             log(x)               log(x)
                             log(x)          ------               ------
                             ------             2                    2
                                2     3 log(%e        + 1)   3 log(%e       -  1)
(d30) y =  %k1 (x - 1) %e          (------------------- - -------------------
                                             4                    4
                                                                          log(x)
                                                                          ------
                                        2(3 x - 2)                          2
                                 - -------------------------) + %k2 (x - 1) %e
                                     3 log(x)       log(x)
                                     --------       ------
                                        2             2
                                 4 %e         - 4 %e
```

The keyword **series** indicates that you want a series solution; **%%n** is a dummy variable to sum over.

```
(c31) ode(eq, y, x, odeseries);
                 inf
                 ====             %%n
                 \               x
          3%k1  >        -------------------
                 /              3         1
                 ====     (%%n - -) (%%n - -)
                 %%n = 0        2         2
(d31)     [y = --------------------------------- + %k2 (1 - x) sqrt(x)]
                                4
(c32) ode(eq, y, x, odeseries, ode2);
```

```
                  inf
                  ====            %%n
                  \          x
             3%k1  >      -------------------
                  /              3         1
                  ====    (%%n - -) (%%n - -)
                  %%n = 0        2         2
(d32) [y = --------------------------------- + %k2 (1 - x) sqrt(x),
                                  4
                                        log(x)                log(x)
                         log(x)         ------                ------
                         ------           2                      2
                           2     3 log(%e       + 1)   3 log(%e       - 1)
y = 2 %k1 (x - 1) %e          (------------------- - -------------------
                                        8                        8
                                                  log(x)
                                                  ------
       3 x - 2                                      2
 - -------------------------) + %k2 (x - 1) %e      ]
      3 log(x)       log(x)
      --------       ------
     2                  2
  4 %e          - 4 %e
```

6.5.2 Symbolic Approximate Solution of an ODE

Using a perturbation calculation, the following computation finds an approximate solution for a nonlinear oscillator equation (Duffing's equation):

```
 2
d y      3     2
--- - e y  + w0  y = 0,   where e << 1
  2
dx
```

This problem can be solved directly with the **lindstedt** command, which is one of the perturbation methods available in Macsyma. We will work out a solution without the **lindstadt** command as an illustration of how to combine Macsyma commands to solve more complicated problems. By assumption, e `<< 1`. Thus, the solution must be close to that of a simple harmonic oscillator. The solution obtained will be correct to first order in e; higher approximations are left as a practice problem.

```
(c33) assume(w > 0)$
(c34) depends(y, x)$
```

Find an approximate solution to the following equation by perturbation methods, where $e << 1$.

```
(c35) duffing_eq:diff(y, x, 2) + w0^2*y - e*y^3;
                         2
                        d y      3     2
(d35)                   --- - e y  + w0  y
                          2
                        dx
```

Assuming y_0 is the solution to the differential equation when $e = 0$, since $e << 1$, the solution can be written as follows (to second order in e).

```
(c36) try_y:y = y[0] + e*y[1] + e^2*y[2];
                              2
(d36)                  y = y  e  + y  e + y
                            2       1      0
```

Expand $w0^2$ as follows, where a_1 and a_2 are chosen to eliminate the secular term.

```
(c37) try_w0:w0^2 = w^2 + a[1]*e + a[2]*e^2;
                                    2   2     2
(d37)                             w0 = w + a e + a e
                                             2     1
```

Substituting *try_y* and *try_w0* into the differential equation *duffing_eq* then differentiate.

```
(c38) ev(duffing_eq, try_y, try_w0, diff);
           2                2       2                  2             3
(d38) (y   e  + y  e + y ) (w  + a  e  + a  e) - e (y  e  + y  e + y )
        2       1      0         2       1          2       1      0
                         2                2              2
                        d          2     d              d
                   + (--- (y )) e  + (--- (y )) e + --- (y )
                          2   2            2   1          2   0
                        dx               dx             dx
```

Since $e << 1$, you can neglect terms in e^3 or higher.

```
(c39) duffing_eq_pert:ratsubst(0, e^3, %);
                                     2
           2               2    d                              2       2
(d39) (y   e   + y   e + y ) w   + (--- (y ) + y   a   + (a   - 3 y ) y ) e
        2       1       0           2   2      0  2      1       0   1
                                    dx
                        2                            2
                       d                    3       d
                   + (--- (y ) + y   a   - y ) e + --- (y )
                         2   1      0  1    0         2   0
                        dx                           dx
```

Equate coefficients of like powers of e.

```
(c40) duffing_e[0]:ratcoef(duffing_eq_pert, e, 0);
                                  2
                            2    d
(d40)                   y   w   + --- (y )
                         0         2   0
                                 dx
(c41) duffing_e[1]:ratcoef(duffing_eq_pert, e, 1);
                                2
                          2    d                   3
(d41)                 y   w   + --- (y ) + y   a   - y
                       1         2   1      0  1    0
                               dx
(c42) duffing_e[2]:ratcoef(duffing_eq_pert, e, 2);
                          2
                    2    d                            2
(d42)           y   w   + --- (y ) + y   a   + (a   - 3 y ) y
                 2         2   2      0  2      1       0   1
                          dx
```

Solve y_0 first, then use the result to solve for y_1, then use previous result to solve for y_2.

```
(c43) sol[0]:ode(duffing_e[0], y[0], x);
Trying     ode2
(d43)               y  = %k1 sin(w x) + %k2 cos(w x)
                     0
(c44) sol[0]:trigreduce(ic2(sol[0], x = 0, y[0] = a*sin(b),
                           'diff(y[0], x) = a*w*cos(b)));
(d44)                   y  = a sin(w x + b)
                         0
```

Substitute y_0 into the equation *duffing_ e*[1].

```
(c45) duffing_e[1]:ev(duffing_e[1], sol[0], trigreduce);
```

```
         3                          3
        a  sin(3 w x + 3 b)   3 a  sin(w x + b)                         2
(d45) ------------------- - ----------------- + a  a sin(w x + b) + y  w
                4                    4            1                    1
                                                         2
                                                        d
                                                      + --- (y )
                                                          2   1
                                                        dx
```

The terms enclosed in boxes give rise to the secular terms, as shown below.

```
(c46) dpart(%, [2, 3]);
                             """""""""""""""""""""
         3                   "     3             "
        a  sin(3 w x + 3 b)  "  3 a  sin(w x + b)"   """"""""""""""""""""
(d46) ------------------- + "- -----------------" + "a  a sin(w x + b)"
            4                "          4        "   " 1                "
                             """""""""""""""""""""   """"""""""""""""""""
                                                                    2
                                                               2   d
                                                         + y   w  + --- (y )
                                                             1        2   1
                                                                    dx
(c47) ode(duffing_e[1], y[1], x);
Trying     ode2
                  3                                  3
(d47) y  = - (- a  sin(3 w x + 3 b) + (8 a  a - 6 a ) sin(w x + b)
       1                                  1
              3                                              2
 + w x (12 a  cos(w x + b) - 16 a  a cos(w x + b)))/(32 w ) + %k1 sin(w x) + %k2 cos(w x)
                                   1
```

Extract the secular terms.

```
(c48) ratcoef(%, x);
                              3
                         (3 a  - 4 a  a) cos(w x + b)
                                    1
(d48)             0 = - ----------------------------
                                     8 w
```

Choose a to eliminate the secular terms.

```
(c49) linsolve(%, a[1]), globalsolve:true;
                                           2
                                        3 a
(d49)                             [a  : ----]
                                    1    4
```

Recall that **yp** yields the particular solution.

```
(c50) sol[1]:y[1] = ev(yp);
                                  3
                                 a  sin(3 w x + 3 b)
(d50)                       y  = -------------------
                             1             2
                                       32 w
```

This is correct to first order in e.

```
(c51) ev(try_y, sol[0], sol[1], y[2] = 0);
                              3
                             a  e sin(3 w x + 3 b)
(d51)                    y = --------------------- + a sin(w x + b)
                                           2
                                       32 w
(c52) w^2 = w0^2 - a[1]*e;
                                            2
                              2      2   3 a  e
(d52)                        w  = w0  - ------
                                           4
```

6.6 Numerical Solutions of ODEs

This section introduces several commands for finding numerical solutions of ODEs.

- **runge_kutta** uses 4^{th} order Runge-Kutta method.
- **ode_numsolve** uses 4^{th} order and adaptive 5^{th} order Runge-Kutta methods.
- **ode_stiffsys** for stiff systems of ODEs.

Here are some examples.

```
(c1) eq : 'diff(y,t) = erf(y^2);
                             dy        2
(d1)                         -- = erf(y )
                             dt
```

```
(c2) ic : 'at(y,t = 0) = 1;
                                  |
(d2)                             y|       = 1
                                  |t = 0
(c3) runge_kutta(eq,'y,'t,ic,0,1,0.1);
(d3) [t = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
y = [1.0, 1.08754, 1.18054, 1.27719, 1.37586, 1.47543, 1.57532,
1.6753, 1.77529, 1.87529, 1.97529],
dy
-- = [0.8427, 0.9056, 0.95127, 0.97894, 0.99257, 0.99792, 0.99955,
dt
0.99993, 0.99999, 1.0, 1.0]]
```

We can plot the result.

```
(c4) graph(assoc('t,%),[assoc('y,%),assoc('diff('y,'t),%)],[0,2])$
```

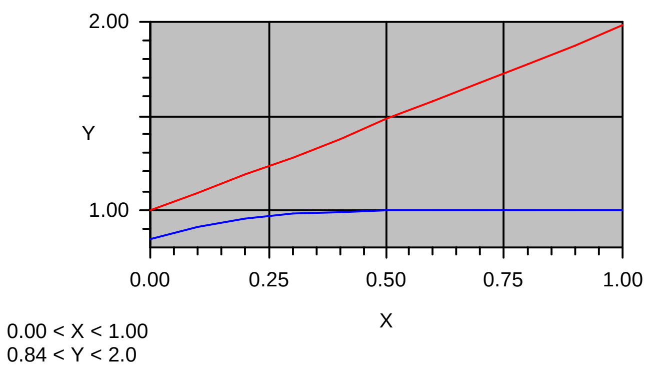

Figure 6.1: Plot of the Numerical Solution to the ODE

```
(c5) ode_numsol(eq,'y,'t,ic,0,1,0.1);
                         dy      dy
(d5)  [[[y] = [y], [--] = [--]], [t = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5,
                         dt      dt
0.6, 0.7, 0.8, 0.9, 1.0], y = [1.0, 1.08754, 1.18054, 1.27719,
1.37586, 1.47543, 1.57532, 1.6753, 1.77529, 1.87529, 1.97529],
dy
-- = [0.8427, 0.9056, 0.95127, 0.97894, 0.99257, 0.99792, 0.99955,
dt
0.99993, 0.99999, 1.0, 1.0]]]
```

Again, we can plot the result.

```
(c6) graph(assoc('t,odns_result),[assoc('y,odns_result),
     assoc('diff('y,'t),odns_result)],[0,2])$
```

This second **graph** command generates exactly the same plot as shown in Figure 6.1

6.7 Computing Laplace Transforms

This section covers the commands available for computing Laplace transforms and inverse Laplace transforms. The section presents the following commands:

- **laplace** takes the Laplace transform of an expression.
- **atvalue** assigns a boundary value to a function or a derivative.
- **ilt** takes the inverse Laplace transform of an expression.
- **odelinsys** solves differential equations for the specified dependent variables, where the functional relationships are explicitly indicated in both the equations and the variables.

In addition, the variable **laplace_call**, which controls the extent to which the integrator attempts to use Laplace transform techniques to solve problems, was introduced in Section 6.2.2, page 86.

The command **laplace**(*exp*, *ovar*, *lvar*) takes the Laplace transform of the expression *exp* with respect to the variable *ovar* and transform parameter *lvar*. Note that the expression *exp* can involve only the functions **exp**, **log**, **sin**, **cos**, **sinh**, **cosh**, and **erf** (the error function). The expression *exp* can also be a linear constant coefficient differential equation, in which case the **atvalue** of the dependent variable is used.

To assign a boundary value to a function or a derivative, use the command **atvalue**(*form*, $equation_1$, ..., $equation_n$, *value*), which assigns the value *value* to *form* at the points specified by the equations $equation_1$ though $equation_n$. The argument *form* can be a function of the form

$$f(var_1,\ \ldots,\ var)$$

or a derivative of the form

$$\mathbf{diff}(f(var_1,\ldots,var_n),var_i,n_i,var_j,n_j,\ldots)$$

in which the functional arguments appear. Symbols of the form `@1`, `@2`, ... represent the functional variables var_1.

You can also apply the inverse Laplace transform to obtain the solutions of the differential equations *eq1* and *eq2*. The command **ilt**(*exp*, *ovar*, *lvar*) takes the inverse Laplace transform of the expression *exp* with respect to the variable *ovar* and transform parameter *lvar*. The expression *exp* must be a ratio of polynomials whose denominator has only linear and quadratic factors.

By using the **laplace** and **ilt** commands with **solve**, you can solve a single differential or convolution integral equation, or a set of them.

Consider the following example. Let $n_a(t)$, $n_b(t)$, and $n_c(t)$ represent the number of nuclei of three radioactive substances, which decay according to the scheme

$$a \to \lambda a \to b \to \lambda b \to c$$

The functions $n_a(t)$, $n_b(t)$, and $n_c(t)$ are known to obey the system of differential equations as given below:

```
(c1) eq1:diff(na(t),t) = -la*na(t);
                         d
(d1)                     -- (na(t)) = - la na(t)
                         dt
(c2) eq2:diff(nb(t), t) = -lb*nb(t) + la*na(t);
                       d
(d2)                   -- (nb(t)) = la na(t) - lb nb(t)
                       dt
(c3) eq3:diff(nc(t), t) = lb*nb(t);
                          d
(d3)                      -- (nc(t)) = lb nb(t)
                          dt
```

Assuming that $N_a(0) = N_0$ and $N_b(0) = N_c(0) = 0$ solve the problem by Laplace transform methods: Set the initial conditions with **atvalue**.

```
(c4) (atvalue(na(t), t = 0, n0),
      atvalue(nb(t), t = 0, 0),
      atvalue(nc(t), t = 0, 0))$
(c5) assume(s > 0)$
```

Apply the laplace transform to the differential equations *eq1*, *eq2*, and *eq3*.

```
(c6) eq1_lt:laplace(eq1, t, s);
(d6)      s laplace(na(t), t, s) - n0 = - la laplace(na(t), t, s)
(c7) eq2_lt:laplace(eq2, t, s);
(d7) s laplace(nb(t), t, s) = la laplace(na(t), t, s) - lb laplace(nb(t), t, s)
(c8) eq3_lt:laplace(eq3, t, s);
(d8)         s laplace(nc(t), t, s) = lb laplace(nb(t), t, s)
```

Solve for the transformed equations.

```
(c9) solve([eq1_lt, eq2_lt, eq3_lt],
        ['laplace(na(t), t, s),
         'laplace(nb(t), t, s),
         'laplace(nc(t), t, s)]);


                                  n0
(d9) [[laplace(na(t), t, s) = ------,
                              s + la


                              la n0
laplace(nb(t), t, s) = ------------------------,
                        2
                       s  + (lb + la) s + la lb
```

```
                              la lb n0
laplace(nc(t), t, s) = ---------------------------]]
                         3              2
                        s  + (lb + la) s  + la lb s


(c10) ilt(%, s, t);
                                                - la t          - lb t
                     - la t           la n0 %e          la n0 %e
(d10) [[na(t) = n0 %e      , nb(t) = -------------- - --------------,
                                         lb - la        lb - la




                                           - lb t          - la t
                                   la n0 %e          lb n0 %e
                           nc(t) = -------------- - -------------- + n0]]
                                       lb - la        lb - la
```

Another way to solve the problem above is with the **odelinsys** command. **odelinsys**([eqn_1, ..., eqn_n] [var_1, ..., var_n]) solves the differential equations eqn_i for the dependent variables var_1, ..., var_n. You must explicitly indicate the functional relationships in both the equations and the variables.

```
(c11) odelinsys([eq1, eq2, eq3], [na(t), nb(t), nc(t)]);
                                               - la t          - lb t
                    - la t           la n0 %e          la n0 %e
(d11) [na(t) = n0 %e      , nb(t) = -------------- - --------------,
                                        lb - la          lb - la
                                    - lb t          - la t
                            la n0 %e          lb n0 %e
                    nc(t) = -------------- - -------------- + n0]
                                lb - la        lb - la
```

6.8 Practice Problems

Using the commands that you have learned about in this chapter, solve the following problems. Answers appear on page 276.

Problem 1. Define the Legendre polynomials using

- the Rodrigues formula

$$p(l, x) = \frac{\frac{d^{l}}{dx^{l}}\left((x^{2} - 1)^{l}\right)}{2^{l}\ l!}$$

- the recurrence relation

$$p_{0} = 1$$

$$p_{1} = x$$

$$p_{n} = \frac{(2\ n - 1)\ p_{n - 1}\ x}{n} - \frac{(n - 1)\ p_{n - 2}}{n}$$

Problem 2. Assuming that $x < 1$, show that

$$\int \frac{x}{1 - x}\, dx = - x - \log(1 - x)$$

Problem 3. Assuming that $a > 0$, show that

$$\int_{0}^{\mathrm{inf}} \%e^{- a\ x}\ (\sin(x) + \cos(x))\, dx = \frac{a}{a^{2} + 1} + \frac{1}{a^{2} + 1}$$

Problem 4. Assuming that $m > 0$ and $n > 0$, show that

```
 inf
/
[    cos(m x) - cos(n x)
I    ------------------- dx = log(n) - log(m)
]              x
/
 0
```

Problem 5. Evaluate the following integral numerically:

```
 8.0
/
[     cos(2.0 x) - cos(3.0 x)
I     ----------------------- dx
]                x
/
 0.01
```

Problem 6. Show that

```
     x
    /
    [
y = I  f(x - t) dt
    ]
    /
     0
```

is a solution of

```
dy
-- = f(x)
dx
```

(Hint: You might find it helpful to change the variable $x - t$ to u in the integral solution.)

Problem 7. Evaluate the limit of the following expression as x approaches $\pi/2$.

```
  log(cos(x))
---------------
log(1 - sin(x))
```

Problem 8. Evaluate the limit of the following expression as x approaches zero.

```
sin(x) - atan(x)
----------------
  2
 x  log(x + 1)
```

Problem 9. Show that as e goes to zero

```
 1
/                     3    5
[  sin(e t)          e    e
I  -------- dt = e - -- + --- + . . .
]     t              18   600
/
 0
```

Hint: You might want to use the **trunc** command to convert your result into the format shown above.

Problem 10. Show that the following

```
         2
        d     (g - 1) x                    2     g x
f = 4 (--- (%e          sech(x))) + (g - 2)  g %e
         2
       dx
```

is a solution for

```
                                                              3
             2                       df             2      d f
0 =  - (24 sech (x) tanh(x) + a) f - 4 -- (1 - 3 sech (x)) + ---
                                       dx                      3
                                                             dx
```

subject to the constraint

```
 3
g  - 4 g - a = 0
```

Problem 11. Solve

```
        2  dy    2
(x y + x ) -- + y  + 3 x y = 0
           dx
```

Problem 12. Solve

$$x\,(y^2 - 3\,x)\,\frac{dy}{dx} + 2\,y^3 - 5\,x\,y = 0$$

Problem 13. Solve the following by

- the default method
- the series method

$$\frac{d^2 y}{dx^2} + \frac{\frac{dy}{dx}}{x} + y = 0$$

Problem 14. Solve the following differential equations:

$$3\left(\frac{d^2}{dx^2}(f(x))\right) - 2\left(\frac{d}{dx}(g(x))\right) = \sin(x)$$

$$a\left(\frac{d^2}{dx^2}(g(x))\right) + \frac{d}{dx}(f(x)) = a\cos(x)$$

with the conditions

$$\frac{df}{dx} = 0 \qquad \text{at} \qquad x = 0$$

$$\frac{dg}{dx} = 1 \qquad \text{at} \qquad x = 0$$

Problem 15. Determine a first-order solution for small but finite u for the following differential equation (from [Na], pages 103-105).

$$u\,w0^{2} + \left(\frac{u^{2}}{1 - u^{2}} + 1\right)\frac{d^{2}u}{dxx^{2}} + \frac{u\left(\frac{du}{dxx}\right)^{2}}{1 - u^{2}} + \frac{g\,u}{l\,\mathrm{sqrt}(1 - u^{2})}$$

Chapter 7

Matrices

A matrix is a two-dimensional, ordered set of elements. Macsyma provides you with a large group of commands for performing matrix operations, as well as many option variables you can set for more flexibility and control over matrix operations. A simple example using matrices appeared in Chapter 1.

The sections below cover such topics as creating, transposing, and inverting matrices, extracting parts from matrices, adding rows and columns, calculating determinants, finding characteristic polynomials, and producing the echelon form of matrices.

The scope of this document allows only a limited introduction to Macsyma's matrix manipulation capabilities. Macsyma has more advanced matrix algebra capabilities. It includes nearly all the numerical linear algebra in MATLAB. For more information, consult the *Macsyma Reference Manual*.

7.1 Creating a Matrix

Macsyma provides commands for creating many kinds of matrices, including identity matrices. This section presents the following commands:

- **entermatrix** creates a matrix element by element, prompting you for each of the values in the matrix.
- **matrix** creates a rectangular matrix with the indicated rows.
- **ident** produces an identity matrix.
- **zeromatrix** creates a rectangular matrix of zeros.
- **diagmatrix** and **diag_matrix** produce diagonal matrices whose elements you specify.
- **coefmatrix** returns the coefficient matrix for the given variables of a system of linear equations.
- **augcoefmatrix** returns the augmented coefficient matrix for the given variables of a system of linear equations.
- **copymatrix** creates a copy of the specified matrix.
- **genmatrix** generates a matrix from an array.

This section illustrates how to use the matrix creation commands to produce various types of matrices. Subsequent sections explain how you can manipulate these matrices in many ways.

The command **entermatrix**(m, n) allows you to create an m by n matrix interactively. Macsyma prompts you for the value to be stored in each of the $m \times n$ entries. The function **entermatrix** also prompts you for the type of matrix to be created. The choices are diagonal, symmetric, antisymmetric, or general. Macsyma uses its knowledge of the different types of matrices so, for instance, if you choose to create a diagonal matrix, **entermatrix** asks only about the nonzero elements.

Create an antisymmetric four by four matrix *mat*.

```
(c1) mat:entermatrix(4,4);
Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4
3;
Row 1 Column 2:  0;
Row 1 Column 3:  0;
Row 1 Column 4:  %i*l;
Row 2 Column 3:  0;
Row 2 Column 4:  %i*v;
Row 3 Column 4:  %i*v;
Matrix entered.
             [    0       0       0     %i l ]
             [                               ]
             [    0       0       0     %i v ]
(d1)         [                               ]
             [    0       0       0     %i v ]
             [                               ]
             [ - %i l  - %i v  - %i v    0   ]
```

You can also create a matrix, with the command **matrix**(row_1, ..., row_n), where each row_i is a list of matrix elements. In the next example we create a three by three matrix.

```
(c2) t:matrix([(ex^2 - ey^2 - ez^2)/2, ex*ey, ex*ez],
             [ex*ey, (ey^2 - ex^2-  ez^2)/2, ey*ez],
             [ex*ez, ey*ez, (ez^2 - ex^2 - ey^2)/2]);
        [     2     2     2                                   ]
        [ - ez  - ey  + ex                                    ]
        [ -----------------          ex ey             ex ez  ]
        [     2                                               ]
        [                                                     ]
        [                          2     2     2              ]
(d2)    [                       - ez  + ey  - ex              ]
        [   ex ey              -----------------       ey ez  ]
        [                              2                      ]
        [                                                     ]
        [                                         2     2     2 ]
        [                                       ez  - ey  - ex  ]
        [   ex ez                  ey ez        --------------- ]
        [                                              2        ]
```

A shorter syntax for the **matrix** command is simpler to use:

```
(c3) [a, b, c; d, e, f; g, h, i];
                              [ a  b  c ]
                              [         ]
(d3)                          [ d  e  f ]
                              [         ]
                              [ g  h  i ]
```

To produce an n by n identity matrix, where each of the diagonal elements is a "1"and each of the other elements is a "0," use the command **ident**(n). You can also produce any diagonal matrix of size n by n, with all diagonal elements x and other elements zero, with the command **diagmatrix**(n, x). Notice that **diagmatrix**(n, 1) is the same as **ident**(n).

Define a three by three identity matrix.

```
(c4) ident(3);
                [ 1  0  0]
                [        ]
(d4)            [ 0  1  0]
                [        ]
                [ 0  0  1]
```

Define a three by three diagonal matrix with diagonal element x.

```
(c5) dmat:diagmatrix(3, x);
                [ x  0  0]
                [        ]
(d5)            [ 0  x  0]
                [        ]
                [ 0  0  x]
```

Use **diag_matrix** to create a three by three diagonal matrix with diagonal elements x, y, and z.

```
(c6) diag_matrix([x, y, z]);
                              [ x  0  0 ]
                              [         ]
(dd)                          [ 0  y  0 ]
                              [         ]
                              [ 0  0  z ]
```

To produce an m by n matrix of zeros use the command **zeromatrix**(m, n).

```
(c7) zmat:zeromatrix(4, 2);
```

```
                  [ 0  0 ]
                  [      ]
                  [ 0  0 ]
(d7)              [      ]
                  [ 0  0 ]
                  [      ]
                  [ 0  0 ]
```

The command **coefmatrix**([eqn_1, ..., eqn_n], [var_1, ..., var_n]) creates the coefficient matrix for the variables var_1, ..., var_n of the system of linear equations eqn_1, ..., eqn_n. Similarly, the command **augcoefmatrix**([eqn_1, ..., eqn_n], [var_1, ..., var_n]) creates the augmented coefficient matrix, which is the coefficient matrix with a column adjoined for the constant terms of each equation. (Constant terms are those that are not dependent on var_i.)

Both **coefmatrix** and **augcoefmatrix** accept eqn_i as polynomials; they need not be equations.

Define a system of linear equations *eqs*.

```
(c8) eqs:[3*zz + b*yy + a*xx - 2,
      4*zz + 2*yy - a = 12,
      90*zz - 12*yy - 3*a = 45]$
```

Define *coefm* as the coefficient matrix for the variables *xx*, *yy*, *zz* of *eqs*.

```
(c9) coefm:coefmatrix(eqs, [xx, yy, zz]);
                   [ a   b    3  ]
                   [             ]
(d9)               [ 0   2    4  ]
                   [             ]
                   [ 0  - 12  90 ]
```

Define *augm* as the augmented coefficient matrix for the variables *xx*, *yy*, *zz* of *eqs*; notice the last column contains the constant terms of each equation

```
(c10) augm:augcoefmatrix(eqs, [xx, yy, zz]);
                    [ a    b    3      - 2     ]
                    [                          ]
(d10)               [ 0    2    4   - a - 12   ]
                    [                          ]
                    [ 0  - 12   90  - 3 a - 45 ]
```

The **copymatrix**(*matrix*) command produces a copy of the matrix *matrix*. This command is useful in conjunction with the **setelmx**, to make a new copy of a matrix when it is changed. **Note**: This is the only way to produce a new copy of a matrix aside from copying the matrix element by element. See the example on page 125.

```
(c11) dmat1:copymatrix(dmat);
                                    [ x  0  0 ]
                                    [         ]
(d11)                               [ 0  x  0 ]
                                    [         ]
                                    [ 0  0  x ]
```

The command **genmatrix**(*array*, i_2, j_2, i_1, j_1) command generates a matrix from the specified array *array*, using *array*(i_1, j_1) for the upper left corner element in the matrix and *array*(i_2, j_2) for the lower right corner element in the matrix. If $i_1 = j_1$, then you do not need to specify j_1. If $i_1 = j_1 = 1$, then you do not need to specify either i_1 or j_1.

Define an array h as shown below.

```
(c12) h[i, j] := 1/(i + j - 1)$
```

Use h to generate a four by four matrix with upper left corner 1/(1 + 1 - 1) = 1, and lower corner 1/(4 + 4 - 1) = 1/7

```
(c13) hilbert:genmatrix(h, 4, 4);
                           [     1  1  1  ]
                           [ 1   -  -  -  ]
                           [     2  3  4  ]
                           [              ]
                           [ 1   1  1  1  ]
                           [ -   -  -  -  ]
                           [ 2   3  4  5  ]
(d13)                      [              ]
                           [ 1   1  1  1  ]
                           [ -   -  -  -  ]
                           [ 3   4  5  6  ]
                           [              ]
                           [ 1   1  1  1  ]
                           [ -   -  -  -  ]
                           [ 4   5  6  7  ]
```

Macsyma also has a built-in command **hilbert** which gererates Hilbert matrices. For more information, refer to the *Macsyma Reference Manual*.

7.2 Extracting From and Adding to a Matrix

This section describes how you can extract data from a matrix or add rows, columns, or single elements of data to a matrix for subsequent use.

The first subsection describes how you can extract rows, columns, or elements from a matrix, and the second subsection describes how you can add rows and columns to the matrix. The last subsection describes how you can change an element in the matrix. *This is the only matrix operation that actually alters the input matrix.*

It is important to note that none of the commands described in the first two subsections actually changes the input matrix. For example, you can extract a row of data in the matrix with the **row** command and perform some operation on it, but the original matrix still contains the extracted row. Similarly, the command **addrow** returns a new matrix with the specified additional row(s); it does not change the original matrix.

Starting with Macsyma 419 and Macsyma 2.0, Macsyma has compact ways to specify submatrices and to assign values to them. See the Release Notes for these versions for more information.

7.2.1 Extracting Rows, Columns, and Elements

You can extract entire rows or columns of a matrix for later use. This section presents the following commands:

- **row** returns a matrix consisting of a specified row in another matrix.
- **col** returns a matrix consisting of a specified column in another matrix.
- **submatrix** makes a new matrix composed of specified rows and columns from another matrix.
- **minor** returns the minor of the given matrix by removing the specified column and row.

The command **row**(*matrix, i*) returns a matrix of the i^{th} row of the matrix *matrix*. Similarly, the command **col**(*matrix, i*) returns a matrix of the i^{th} column of the matrix *matrix*.

Define a matrix *hilbert*.

```
(c1) h[i, j] := 1/(i + j - 1)$
(c2) hilbert:genmatrix(h, 4, 4);
                  [    1  1  1  ]
                  [ 1  -  -  -  ]
                  [    2  3  4  ]
                  [             ]
                  [ 1  1  1  1  ]
                  [ -  -  -  -  ]
                  [ 2  3  4  5  ]
(d2)              [             ]
                  [ 1  1  1  1  ]
                  [ -  -  -  -  ]
                  [ 3  4  5  6  ]
                  [             ]
                  [ 1  1  1  1  ]
                  [ -  -  -  -  ]
                  [ 4  5  6  7  ]
```

Extract row four from the matrix *hilbert*.

```
(c3) row(hilbert, 4);
```

```
                  [ 1  1  1  1 ]
(d3)              [ -  -  -  - ]
                  [ 4  5  6  7 ]
```

Here is a shorter syntax for extracting a row.

```
(c4) hilbert[4, ..];
                  [ 1  1  1  1 ]
(d4)              [ -  -  -  - ]
                  [ 4  5  6  7 ]
```

Extract column three from the matrix *hilbert*.

```
(c5) col(hilbert, 3);
                      [ 1 ]
                      [ - ]
                      [ 3 ]
                      [   ]
                      [ 1 ]
                      [ - ]
                      [ 4 ]
(d5)                  [   ]
                      [ 1 ]
                      [ - ]
                      [ 5 ]
                      [   ]
                      [ 1 ]
                      [ - ]
                      [ 6 ]
```

Here is the shorter syntax for extracting a column.

```
(c6) hilbert[.., 3];
```

```
                       [ 1 ]
                       [ - ]
                       [ 3 ]
                       [   ]
                       [ 1 ]
                       [ - ]
                       [ 4 ]
(d6)                   [   ]
                       [ 1 ]
                       [ - ]
                       [ 5 ]
                       [   ]
                       [ 1 ]
                       [ - ]
                       [ 6 ]
```

To extract a submatrix, indicate the rows and columns you wish to extract using the sequence operator "..".

Extract a submatrix from the matrix hilbert.

```
(c7) hilbert[2..4, 2..3];
                                [ 1  1 ]
                                [ -  - ]
                                [ 3  4 ]
                                [      ]
                                [ 1  1 ]
(d7)                            [ -  - ]
                                [ 4  5 ]
                                [      ]
                                [ 1  1 ]
                                [ -  - ]
                                [ 5  6 ]
```

To create a matrix from a part of an existing matrix *matrix*, use the command **submatrix**(row_1, ..., row_n, *matrix*, col_1, ..., col_n), where the new matrix contains all rows except each row_i and all columns except each col_i. Both row_1, ..., row_n and col_1, ..., col_n are optional; you can extract only rows or only columns if you wish.

Return a new matrix containing all but the third column of *hilbert*.

```
(c8) submatrix(hilbert, 3);
```

```
                    [     1   1  ]
                    [ 1   -   -  ]
                    [     2   4  ]
                    [            ]
                    [ 1   1   1  ]
                    [ -   -   -  ]
                    [ 2   3   5  ]
(d8)                [            ]
                    [ 1   1   1  ]
                    [ -   -   -  ]
                    [ 3   4   6  ]
                    [            ]
                    [ 1   1   1  ]
                    [ -   -   -  ]
                    [ 4   5   7  ]
```

The command **minor**(*matrix*, i, j) returns the minor of the given matrix by removing the specified row i and column j.

Create a minor of the matrix *hilbert* by removing the third row and second column.

```
(c9) minor(hilbert, 3, 2);
                    [     1   1  ]
                    [ 1   -   -  ]
                    [     3   4  ]
                    [            ]
                    [ 1   1   1  ]
(d9)                [ -   -   -  ]
                    [ 2   4   5  ]
                    [            ]
                    [ 1   1   1  ]
                    [ -   -   -  ]
                    [ 4   6   7  ]
```

To extract an element from a matrix, use the notation *matrixname*$[i, j]$, where *matrixname* is the name of the matrix, i is the row, and j is the column of the desired element.

Extract the element in row two, column three, of the matrix in (d6).

```
(c10) %[2, 3];
                               1
(d10)                          -
                               5
```

7.2.2 Adding Rows and Columns to a Matrix

You can use the **addrow** and **addcol** commands to add rows or columns to a matrix, respectively. The command **addrow**(*matrix, list-or-matrix*$_1$, ..., *list-or-matrix*$_n$) appends the rows specified in each *list*$_i$ or *matrix*$_i$ to the matrix *matrix*. Similarly, the command **addcol**(*matrix, list-or-matrix*$_1$, ..., *list-or-matrix*$_n$) appends the columns specified in each *list*$_i$ or *matrix*$_i$ to the matrix *matrix*.

Use **zeromatrix** to define the matrix *empty*.

```
(c1) empty:zeromatrix(3,3);
                             [ 0  0  0 ]
                             [         ]
(d1)                         [ 0  0  0 ]
                             [         ]
                             [ 0  0  0 ]
```

Add two rows to the matrix *empty*, as specified in the lists below.

```
(c2) addrow(empty, [a41, a42, a43], [a51, a52, a53]);
                          [  0    0    0  ]
                          [               ]
                          [  0    0    0  ]
                          [               ]
(d2)                      [  0    0    0  ]
                          [               ]
                          [ a41  a42  a43 ]
                          [               ]
                          [ a51  a52  a53 ]
```

Add a column to the matrix above.

```
(c3) addcol( [b14, b24, b34, b44, b54]);
                       [  0    0    0   b14 ]
                       [                    ]
                       [  0    0    0   b24 ]
                       [                    ]
(d3)                   [  0    0    0   b34 ]
                       [                    ]
                       [ a41  a42  a43  b44 ]
                       [                    ]
                       [ a51  a52  a53  b54 ]
```

7.2.3 Changing the Elements in a Matrix

This section describes how you can change the elements stored in a matrix with the **setelmx** command. **setelmx**(*x, i, j, matrix*) changes the i, j element of the matrix *matrix* to x. This command returns the altered matrix.

Alternatively, you can use the notation *matrix*$[i, j]$:x to change the i, j element to x. In this case, Macsyma returns the value x rather than the altered matrix.

Create a two by two matrix

```
(c1) abcd:matrix([a, b], [c, d]);
                  [ a  b ]
(d1)              [      ]
                  [ c  d ]
```

Change the element in the second row, second column, to `z`. The result is the changed matrix *abcd*.

```
(c2) setelmx(z, 2, 2, abcd);
                      [ a  b ]
(d2)                  [      ]
                      [ c  z ]
```

Change the element in the first row, first column, to `0`. The result is the new value `0`.

```
(c3) abcd[1,1]:0;
(d3)                     0
(c4) abcd;
                      [ 0  b ]
(d4)                  [      ]
                      [ c  z ]
```

You can create a copy of a matrix with **copymatrix**, or you can assign a matrix to one or more variables. For example:

```
(c5) abcd_copy:copymatrix(abcd);
                      [ 0  b ]
(d5)                  [      ]
                      [ c  z ]
(c6) abcd_variable:abcd;
                      [ 0  b ]
(d6)                  [      ]
                      [ c  z ]
```

You should be aware of an important difference between these two approaches.

- Two matrices created with **copymatrix** are two distinct matrices. When you change the elements in one with **setelmx** or "**:**", the elements of the other do not change.
- When you assign a matrix as a value to two or more variables, and then change the elements in this matrix, the values of all variables change. Internally, both variables point to the same matrix.

This distinction is illustrated in the following example. Notice the current contents of the matrix *abcd*.

```
(c7) abcd;
                    [ 0  b ]
(d7)                [      ]
                    [ c  z ]
```

Reset the element in the second row, second column, of the matrix*abcd* to *d*.

```
(c8) setelmx(d, 2, 2, abcd);
                    [ 0  b ]
(d8)                [      ]
                    [ c  d ]
```

The copied matrix *abcd_copy*, created with **copymatrix**, does not reflect this change.

```
(c9) abcd_copy
                    [ 0  b ]
(d9)                [      ]
                    [ c  z ]
```

The variable *abcd_variable*, whose value is that of the matrix *abcd*, does reflect the change.

```
(c10) abcd_variable;
                    [ 0  b ]
(d10)               [      ]
                    [ c  d ]
```

7.3 Arithmetic Operations on Matrices

You can use the operators "`+`", "–", "`*`", "`.`", and "`^`" on matrices. The operations work on the corresponding elements of each matrix. (These operators were introduced in Section 3.1.1, page 15.) You can use the "`.`" operator to perform matrix multiplication. In addition, you can use the operator "`^^`" to raise a matrix to a power. The notation *matrix*`^^-1` inverts a matrix, as does the command **invert**. `m.m` is equivalent to `m^^2` for a matrix `m`.

Examples of arithmetic operations on matrices appear below.

Define two matrices.

```
(c1) mat_wxyz:matrix([w,x],[y,z]);
                    [ w  x ]
(d1)                [      ]
                    [ y  z ]
```

```
(c2) mat_1234:matrix([1,2],[3,4]);
                                 [ 1  2 ]
(d2)                             [      ]
                                 [ 3  4 ]
```

Add together the elements of the matrices *mat_1234* and *mat_wxyz*.

```
(c3) mat_1234 + mat_wxyz;
                                 [ w + 1  x + 2 ]
(d3)                             [              ]
                                 [ y + 3  z + 4 ]
```

Subtract the elements of the matrices *mat_1234* and *mat_wxyz*.

```
(c4) mat_1234 - mat_wxyz;
                                 [ 1 - w  2 - x ]
(d4)                             [              ]
                                 [ 3 - y  4 - z ]
```

You will probably do matrix multiplication with the "**.**" operator, since matrix multiplication is generally non-commutative.

```
(c5) mat_1234 * mat_wxyz;
                                   [  w   2 x ]
(d5)                               [          ]
                                   [ 3 y  4 z ]
(c6) mat_1234 . mat_wxyz;
                             [  2 y + w    2 z + x  ]
(d6)                         [                      ]
                             [ 4 y + 3 w  4 z + 3 x ]
```

Raise the elements of the matrix *mat_1234* to the third power.

```
(c7) mat_1234^3;
                                    [ 1   8  ]
(d7)                                [        ]
                                    [ 27  64 ]
```

To invert a matrix `m`, you can use the notation `m^^-1`. Another way to invert a matrix is with the command **invert**(*matrix*).

```
(c8) mat_wxyz^^-1;
```

```
                   [      z             x     ]
                   [  ---------   - --------- ]
                   [  w z - x y     w z - x y ]
(d8)               [                          ]
                   [       y           w      ]
                   [ - ---------   ---------  ]
                   [   w z - x y   w z - x y  ]
(c9) invert(mat_1234);
                       [ - 2   1 ]
                       [         ]
(d9)                   [  3     1 ]
                       [  -    - - ]
                       [  2     2 ]
```

To compute a matrix inverse while keeping the overall factor of the matrix determinant outside the matrix result, set the option variable **detout** to **true**. An example of **detout** appears on page 135.

To compute the trace of a square matrix Macsyma has a built-in command **matrux_trace**. For more information, refer to the *Macsyma Reference Manual*.

7.4 Producing the Echelon Form of a Matrix

Computing the echelon form of a matrix results in a matrix where the first nonzero element in each row is a `1`, and all the column elements under the first `1` in each row are `0`.

The command **echelon**(*matrix*) uses elementary row operations to produce the echelon form of the matrix *matrix*.

```
(c1) eqs:[3*zz + b*yy + a*xx - 2,
      4*zz + 2*yy - a = 12,
      90*zz - 12*yy - 3*a = 45]\$
(c2) augm:augcoefmatrix(eqs, [xx, yy, zz]);
                    [ a   b    3      - 2     ]
                    [                         ]
(d2)                [ 0   2    4    - a - 12  ]
                    [                         ]
                    [ 0  - 12  90  - 3 a - 45 ]
(c3) echelon(augm);
```

```
                      [    b  3        2     ]
                      [ 1  -  -       - -    ]
                      [    a  a        a     ]
                      [                      ]
                      [             a + 12   ]
(d3)                  [ 0  1  2   - ------   ]
                      [                2     ]
                      [                      ]
                      [           3 a + 39   ]
                      [ 0  0  1  - --------  ]
                      [                38    ]
```

7.5 Calculating Determinants

You can calculate the determinant of a matrix. This section presents the following commands:

- **determinant** computes the determinant of a given matrix by a method similar to Gaussian elimination.
- **rank** computes the order of the largest nonzero subdeterminant of the given matrix.

To calculate the determinant of the square matrix *matrix*, you can use the command **determinant**(*matrix*). This command uses a method similar to Gaussian elimination.

The command **rank**(*matrix*) computes the rank of the matrix *matrix*. The rank is the order of the largest nonzero subdeterminant of *matrix*.

```
(c1) t:matrix([(ex^2 - ey^2 - ez^2)/2, ex*ey, ex*ez],
              [ex*ey, (ey^2 - ex^2 - ez^2)/2, ey*ez],
              [ex*ez, ey*ez, (ez^2 - ex^2 - ey^2)/2]);


        [     2     2     2                                     ]
        [ - ez  - ey  + ex                                      ]
        [ -----------------        ex ey              ex ez     ]
        [         2                                             ]
        [                                                       ]
        [                          2     2     2                ]
(d1)    [                     - ez  + ey  - ex                  ]
        [   ex ey             -----------------       ey ez     ]
        [                             2                         ]
        [                                                       ]
        [                                          2     2     2 ]
        [                                        ez  - ey  - ex  ]
        [   ex ez                   ey ez        ---------------  ]
        [                                               2        ]
(c2) rank(t);
(d2)                        3
```

```
(c3) determinant(t - lambda*ident(3));


          2     2     2              2     2     2
      - ez  + ey  - ex             ez  - ey  - ex                2    2
(d3) ((----------------- - lambda) (--------------- - lambda) - ey  ez )
               2                            2
     2     2     2                               2     2     2
  - ez  - ey  + ex                              ez  - ey  - ex
 (----------------- - lambda) - ex ey (ex ey (--------------- - lambda)
         2                                            2
                                              2     2     2
          2                2               - ez  + ey  - ex
 - ex ey ez ) + ex ez (ex ey  ez - ex ez (----------------- - lambda))
                                                    2


(c4) factor(%);
                        2     2     2                2     2     2 2
          (2 lambda - ez  - ey  - ex ) (2 lambda + ez  + ey  + ex )
(d4)    - --------------------------------------------------------
                                     8
```

7.6 Eigenanalysis of Matrices

You can find eigenvalues, characteristic polynomials, and eigenvectors of matrices. This section presents the following commands:

- **charpoly** computes the characteristic polynomial for a given matrix with respect to the specified variable.
- **eigenvalues** returns the eigenvalues of the specified matrix.
- **eigenvectors** returns the eigenvalues and the corresponding eigenvectors of the specified matrix.

The command **charpoly**(*matrix, var*); returns the characteristic polynomial of a matrix for the variable *var*. You can use the **solve** command to solve for the eigenvalues of a matrix.

```
(c1) t:matrix([(ex^2 - ey^2 - ez^2)/2, ex*ey, ex*ez],
              [ex*ey, (ey^2 - ex^2 - ez^2)/2, ey*ez],
              [ex*ez, ey*ez, (ez^2 - ex^2 - ey^2)/2]);
```

```
          [     2     2     2                                     ]
          [ - ez  - ey  + ex                                      ]
          [ -----------------        ex ey              ex ez     ]
          [         2                                             ]
          [                                                       ]
          [                         2     2     2                 ]
(d1)      [                     - ez  + ey  - ex                  ]
          [        ex ey        -----------------        ey ez    ]
          [                             2                         ]
          [                                                       ]
          [                                        2     2     2 ]
          [                                      ez  - ey  - ex  ]
          [        ex ez             ey ez       --------------- ]
          [                                             2        ]


(c2) solve(charpoly(t, ei), ei);
                      2     2     2            2     2     2
                    ez  + ey  + ex           ez  + ey  + ex
(d2)          [ei = ---------------, ei = - ---------------]
                           2                        2
(c3) multiplicities;
(d3)                  [1, 2]
```

The command **eigenvectors**(*matrix*) returns a list containing several sublists. The first sublist contains the eigenvalues of the matrix *matrix*, and the remaining sublists contain the eigenvectors corresponding to each of the eigenvalues in the first sublist.

```
(c4) eig_info:eigenvectors(t);
          2     2     2       2     2     2
        ez  + ey  + ex      ez  + ey  + ex                  ey   ez
(d4) [[[---------------, - ---------------], [1, 2]], [1, --, --],
               2                  2                        ex   ex
                                 ex            ey
                       [1, 0, - --], [0, 1, - --]]
                                 ez            ez
```

You can find eigenvalues and eigenvectors of general floating point matrices using **eigens_by_schur.** See the *Macsyma Reference Manual* for more information.

7.7 Transposing a Matrix

Use the command **transpose**(*matrix*) to produce the transpose of the matrix *matrix*. Note that transposing the transpose of a matrix returns the matrix itself.

The next matrix could be thought of as defining the stress in an electric field.

```
(c1) t:matrix([(ex^2 - ey^2 - ez^2)/2, ex*ey, ex*ez],
              [ex*ey, (ey^2 - ex^2 - ez^2)/2, ey*ez],
              [ex*ez, ey*ez, (ez^2 - ex^2 - ey^2)/2]);


              [     2     2     2                                     ]
              [ - ez  - ey  + ex                                      ]
              [ -----------------         ex ey             ex ez     ]
              [        2                                              ]
              [                                                       ]
              [                        2     2     2                  ]
(d1)          [                     - ez  + ey  - ex                  ]
              [   ex ey             -----------------       ey ez     ]
              [                            2                          ]
              [                                                       ]
              [                                        2     2     2  ]
              [                                      ez  - ey  - ex   ]
              [   ex ez                   ey ez      ---------------  ]
              [                                             2         ]


(c2) solve(charpoly(t, ei), ei);
                        2     2     2            2     2     2
                      ez  + ey  + ex           ez  + ey  + ex
(d2)           [ei = ---------------, ei = - ---------------]
                            2                        2
(c3) multiplicities;
(d3)                   [1, 2]


(c4) eig_info:eigenvectors(t);
             2     2     2
           ez  + ey  + ex       ex     ez
(d4) [[---------------, 1, [[--, 1, --]]],
                  2             ey     ey
                        2     2     2
                      ez  + ey  + ex                 ex            ey
                   [- ---------------, 2, [[1, 0, - --], [0, 1, - --]]]]
                             2                       ez            ez
```

Pick out one eigenvector using the part command and transpose it using the **transpose** command.

```
(c5) transpose(part(eig_info, 1, 3, 1));
                                [ ex ]
                                [ -- ]
                                [ ey ]
                                [    ]
(d5)                            [ 1  ]
                                [    ]
                                [ ez ]
                                [ -- ]
                                [ ey ]
```

Macsyma also uses the postfix operator ^‘ as a shorter way to indicate transpose.

```
(c6) part(eig_info, 1, 3, 1)^‘;
                                [ ex ]
                                [ -- ]
                                [ ey ]
                                [    ]
(d6)                            [ 1  ]
                                [    ]
                                [ ez ]
                                [ -- ]
                                [ ey ]
```

Construct a matrix whose columns are eigenvectors.

```
(c7) aa:addcol(%,part(eig_info, 2, 3,1), part(eig_info, 2, 3, 2));


                          [ ex            ]
                          [ --    1     0 ]
                          [ ey            ]
                          [               ]
(d7)                      [ 1     0     1 ]
                          [               ]
                          [ ez    ex    ey ]
                          [ --  - --  - -- ]
                          [ ey    ez    ez ]
```

```
(c8) aa^^-1;
       [                            2                            ]
       [        ex ey              ey                ey ez       ]
       [  ---------------    ---------------    ---------------  ]
       [    2     2     2      2     2     2      2     2     2  ]
       [  ez  + ey  + ex     ez  + ey  + ex     ez  + ey  + ex   ]
       [                                                         ]
       [       2     2                                           ]
       [     ez  + ey              ex ey              ex ez      ]
(d8)   [  ---------------   - ---------------  - --------------- ]
       [    2     2     2       2     2     2      2     2     2 ]
       [  ez  + ey  + ex      ez  + ey  + ex     ez  + ey  + ex  ]
       [                                                         ]
       [                          2     2                        ]
       [        ex ey           ez  + ex              ey ez      ]
       [ - ---------------   ---------------   - --------------- ]
       [     2     2     2     2     2     2       2     2     2 ]
       [   ez  + ey  + ex    ez  + ey  + ex      ez  + ey  + ex  ]
(c9) aa_1:aa^^-1, detout;


              [          2            3            2     ]
              [     ex ey            ey           ey  ez ]
              [                                          ]
              [      2     3           2                 ]
              [ ey ez  + ey      - ex ey       - ex ey ez ]
              [                                          ]
              [          2                 2          2  ]
              [   - ex ey      ey ez + ex ey      - ey ez  ]
(d9)          --------------------------------------------
                             2        3     2
                        ex ez  +   ey  + ex  ey
```

Show that the matrix of eigenvalues defines a similarity transformation which diagonalizes the matrix t.

```
(c10) aa_1 . t . aa, ratsimp;
              [   2     2     2                                      ]
              [ ez  + ey  + ex                                       ]
              [ ---------------           0                 0        ]
              [       2                                              ]
              [                                                      ]
              [                     2     2     2                    ]
(d10)         [                   ez  + ey  + ex                     ]
              [       0         - ---------------           0        ]
              [                          2                           ]
              [                                                      ]
              [                                       2     2     2  ]
              [                                     ez  + ey  + ex   ]
              [       0                  0        - --------------- ]
              [                                            2         ]
```

Chapter 8

Plotting

This chapter provides a brief introduction to Macsyma's extensive plotting capabilities. Macsyma provides commands that allow you to produce two and three-dimensional plots of functions and expressions. You can produce hidden-line plots and contour plots, and by varying option variables, you can control the plot's scale, perspective, and coordinate system.

The plot illustrations in this chapter were produced using Macsyma 2.0.

Not all plotting features are available on all versions of Macsyma. Consult the *Release Notes* for information about what is available on your system.

Although the examples in this chapter introduce many commands and option variables, this document has only a limited introduction to Macsyma's plotting capabilities. Consult the *Macsyma Reference Manual* for more information.

8.1 Creating Two-Dimensional Plots

This section presents the following commands:

- **plot** plots an expression in the y direction which is a function of the independent variable, which is plotted in the x direction.
- **paramplot** plots one expression as the x coordinate against another expression as the y coordinate where both expressions are functions of an independent parameter.
- **graph** plots points specified by a list of abscissas and a list of ordinates.

See also **contourplot**, **implicit_plot**, and **plot2_vect** in the *Macsyma Reference Manual* .

This section also introduces the option variable **plotnum**.

You can use the command **plot**(*exp*, *var*, *bound1* , *bound2*) to produce a two-dimensional plot of the expression *exp*, where *var* is the variable to plot against, and *bound1* and *bound2* are the limits of the values of the independent variable. If you wish, you can provide a list of values to plot rather than specifying the range with *bound1* and *bound2*.

The following example plots an expression for x between -3 and 3. The result is shown in Figure 8.1.

```
(c1) gauss_deriv(x):=-2*x*exp(-x^2);
```

```
                                              2
                                           - x
(d1)                 gauss_deriv(x) := - 2 x %e
(c2) plot(gauss_deriv(x), x, -3, 3);
```

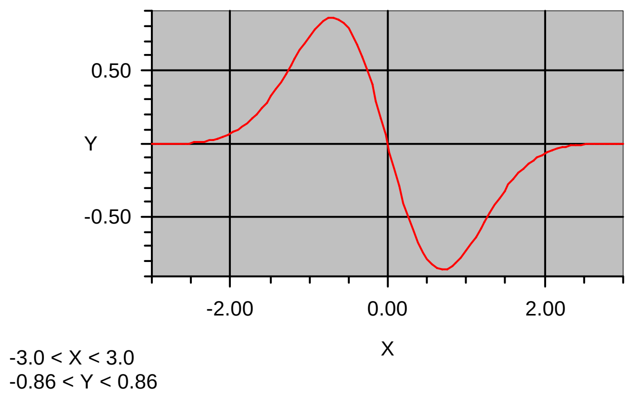

Figure 8.1: Two-Dimensional Plot with **plot**

You can also provide a list of values to be plotted, rather than a range. The next example replots the expression *gauss_deriv* for a list of points. The result is shown in Figure 8.2.

```
(c3) plot(gauss_deriv, x, [-3, -2, -1, 0, 1, 2, 3])$
```

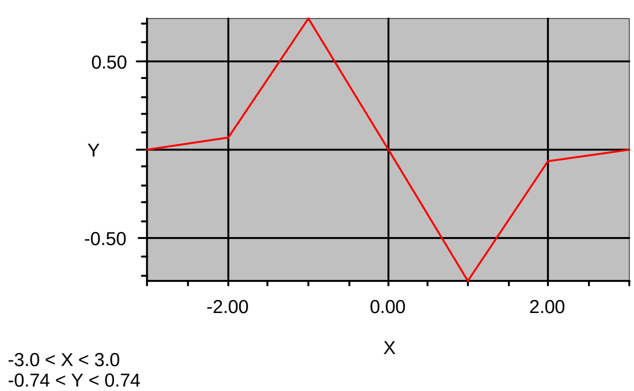

Figure 8.2: Two-Dimensional Plot from a List of Values

When you specify the plotting range with *bound1* and *bound2*, *var* takes on the number of values specified by **plotnum** within the range. By default, the value of **plotnum** is 100. You can reset this variable to control the number of points in the plot.

The next example plots 200 points in the expression $gauss\_deriv \times \sin(10x)$ for x between -3 and 3. The result is shown in Figure 8.3.

```
(c4) plot(gauss_deriv(x)*sin(10*x), x, -3, 3), plotnum:200$
```

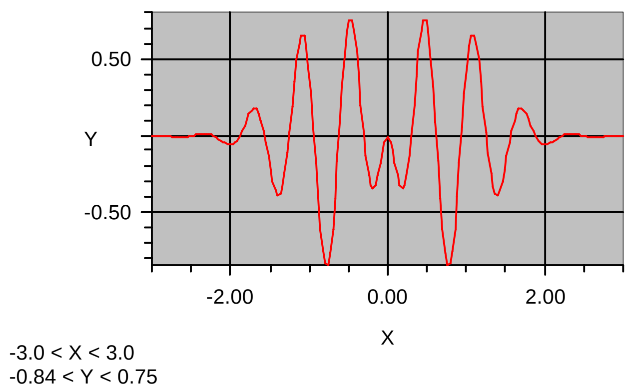

Figure 8.3: Plot with **plotnum** Set Higher

The command **paramplot**(*x-expr*, *y-expr*, *var*, *bound1*, *bound2*) plots the expression or list of expressions *x-expr* as the x coordinate against the expression or list of expressions *y-expr* as the y coordinate. The arguments *bound1* and *bound2* give the range of the plot.

In the following example **paramplot** plots *x_expr* against *y_expr*. The result is shown in Figure 8.4.

```
(c5) x_expr:cos(x);
(d5)                    cos(x)
(c6) y_expr:sin(3*x);
(d6)                   sin(3 x)
(c7) paramplot(x_expr, y_expr, x, -5*%pi, 5*%pi)$
```

Plotting more points yields a more accurate representation. Compare the result of the next plot, shown in Figure 8.5, with the plot in Figure 8.4.

```
(c8) paramplot(x_expr, y_expr, x, -5*%pi, 5*%pi), plotnum:1000$
```

The two-dimensional plotting commands recognize several types of coordinate systems. You can specify the coordinate system with an optional argument to the plotting command. An example using polar coordinates appears below. Other coordinate systems are described in the *Macsyma Reference Manual* .

You can redisplay the plot in Figure 8.5 in polar coordinates. The result is shown in Figure 8.6.

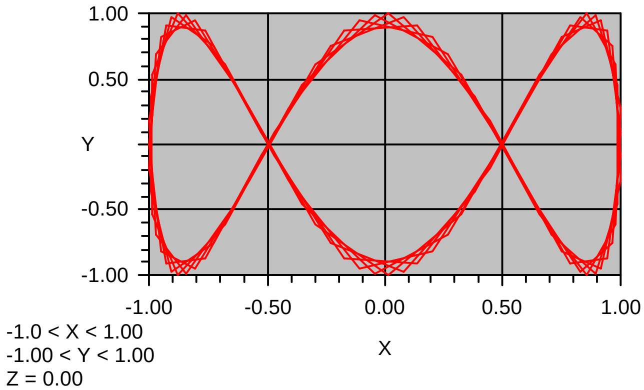

Figure 8.4: Two-Dimensional Plot with **paramplot**

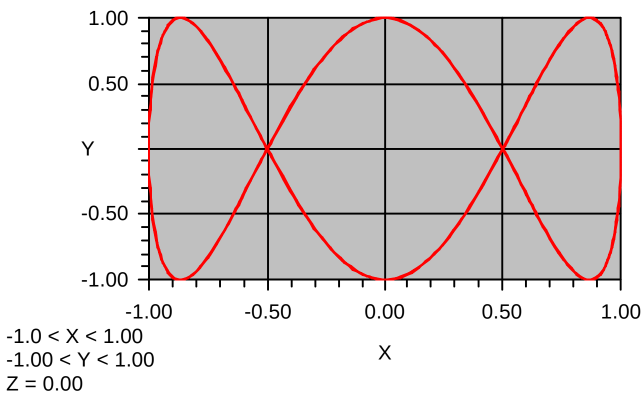

Figure 8.5: **Paramplot** with More Points Plotted

```
(c9) paramplot(x_expr, y_expr, x, -5*%pi, 5*%pi, polar), plotnum:1000$
```

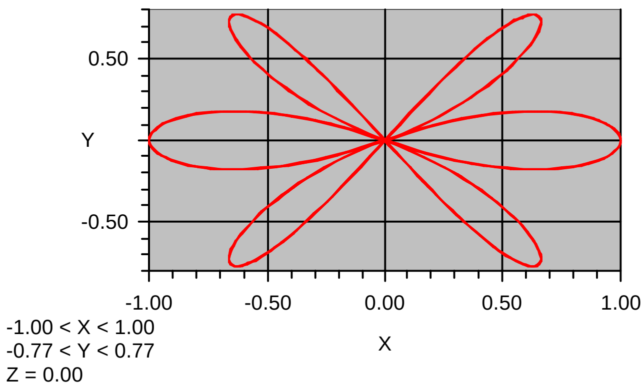

Figure 8.6: Plot with Polar Coordinates

The command **graph**(*x-list*, *y-list*) plots points specified by the list (or list of lists) *x-list* and *y-list*.

The example below uses **graph** to plot lists of points produced by the numerical solution of coupled differential equations. The result is shown in Figure 8.7.

```
(c10)eqns: ['diff(x,t)=-3.0*(x-y),'diff(y,t)=-x*z + 30.*x-y,'diff(z,t)=x*y - z];
          dx                    dy                        dz
(d10)    [-- =  - 3.0 (x - y), -- =  - x z - y + 30.0 x, -- = x y - z]
          dt                    dt                        dt
(c11) ic: ['at(x,t=0)=0.,'at(y,t=0)=1.,'at(z,t=0)=0.];
           |              |              |
(d11)    [x|      = 0.0, y|      = 1.0, z|    =0.0]
           |t = 0         |t = 0         |t=0
(c12) sol_xyz: runge_kutta(eqns,[x,y,z],t, ic,0., 25.0, 0.01)$
(c13)graph(assoc('x,sol_xyz), assoc('y,sol_xyz))$
```

You can use the command **graph3d**(*x-list*, *y-list*, *z-list*) to create a three-dimensional plot specified by the points in the lists *x-list*, *y-list*, and *z-list*. (These arguments can also be lists of lists.)

The function **runge_kutta** produces three lists of points to plot. The example below uses **graph** to plot the points from these three lists. The result is shown in Figure 8.8.

```
(c14)graph3d(assoc('x,sol_xyz), assoc('y,sol_xyz), assoc('z,sol_xyz))$
```

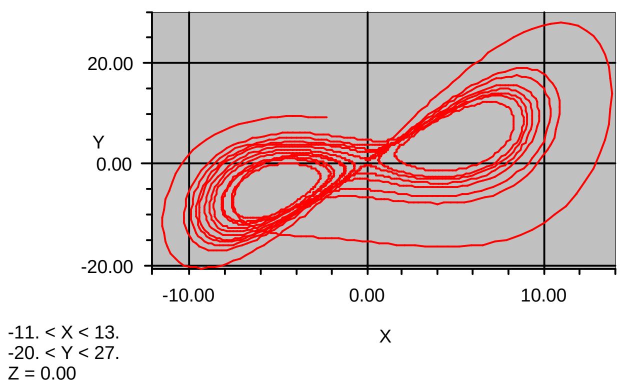

Figure 8.7: Two-Dimensional Plot with **graph**

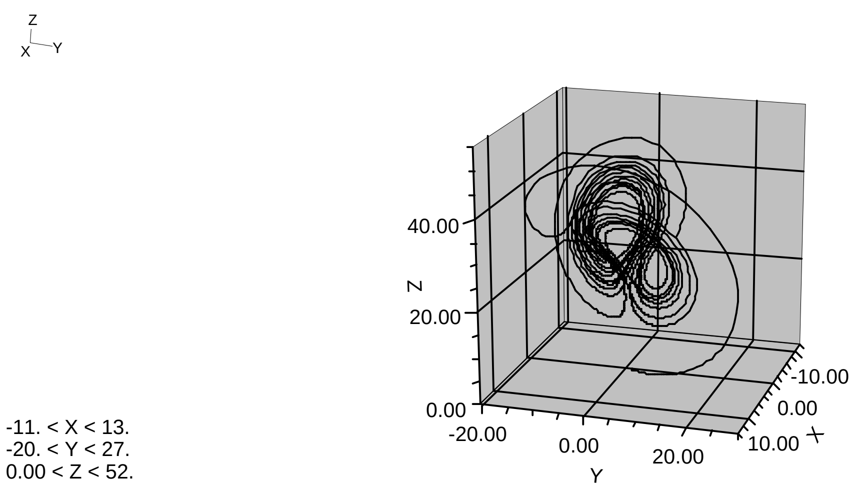

Figure 8.8: Three-Dimensional Plot with **graph3d**

8.2 Creating Three-Dimensional Plots

This section presents the following commands:

- **plot3d** plots an expression three-dimensionally with respect to the specified x and y variables.
- **contourplot** calculates the same plots as **plot3d**, but displays the points as a contour plot.
- **graph3d** produces a three-dimensional plot specified by lists of x, y, and z point.
- **plotsurf** plots parametric surfaces embedded in three-dimensional space.

See also **contourplot3d**, **paramplot3d**, in the *Macsyma Reference Manual* .

To produce three-dimensional plots, use **plot3d**(*exp*, *xvar*, *xbound1*, *xbound2*, *yvar*, *ybound1*, *ybound2*) where *exp* is the expression to be plotted, *xvar* and *yvar* are the variables to plot against, *xbound1* and *xbound2* are the limits of the plot's x values, and *ybound1* and *ybound2* are the limits of the plot's y values.

The following example plots an expression for x between 0 and 2 and for y between -2 and 2. The result is shown in Figure 8.9.

```
(c1) p2_expr:y^2 + cos(4*x) + 2*x;
                         2
(d1)                    y  + cos(4 x) + 2 x
(c2) plot3d(p2_expr, x, 0, 2, y, -2, 2)$
```

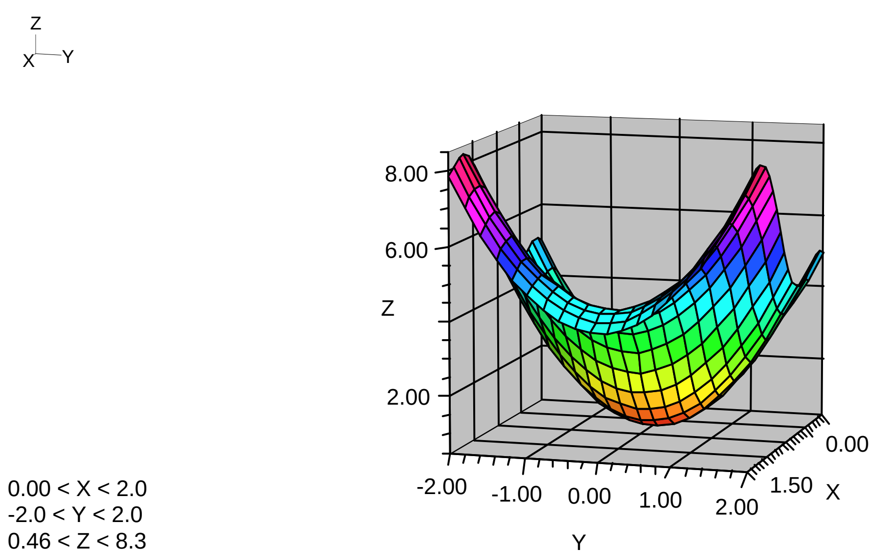

Figure 8.9: Three-Dimensional Plot with **plot3d**

The variable **equalscale**, which is discussed in more detail in Section 8.3, indicates whether or not the scale of the plot should be the same in both directions. The previous plot looks different with equal scaling, as shown in Figure 8.10.

```
(c3) plot3d(p2_expr, x, 0, 2, y, -2, 2), equalscale:true$
```

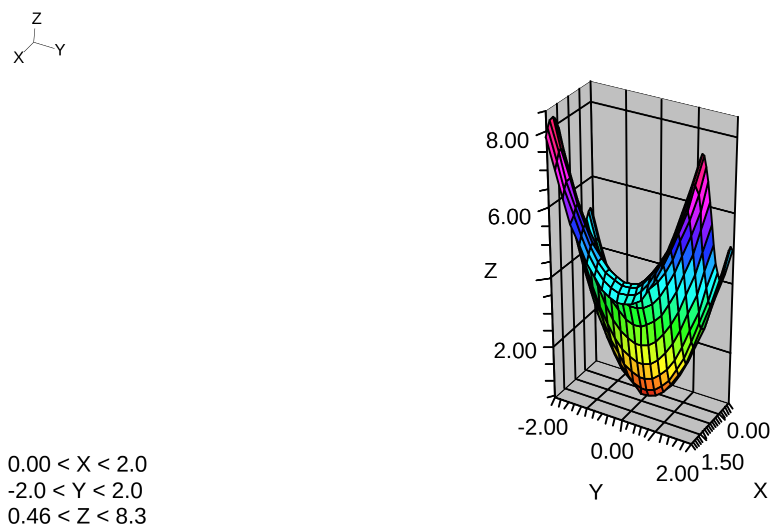

Figure 8.10: Plot with **equalscale** set to **true**

The command **contourplot**(*exp*, *xvar*, *xbound1*, *xbound2*, *yvar*, *ybound1*, *ybound2*) calculates the same points as **plot3d**, but produces a contour plot of the expression *exp* instead.

The next example produces a contour plot of the expression in (c1). Compare Figure 8.10 with Figure 8.11.

```
(c4) contourplot(p2_expr, x, 0, 2, y, -2, 2)$
```

Contour plots can be drawn in three dimensions. The next example uses the command **contourplot3d** to produce a 3D plot of the same expression. The result is shown in Figure 8.12.

```
(c5) contourplot3d(p2_expr, x, 0, 2, y, -2, 2), contours:40$
```

You can use the command **plotsurfplotsurf** to plot two-dimensional surfaces embedded in three-dimensional space. Each surface is represented in parametric form as $[x(s,t), y(s,t), z(s,t)]$, where s and t are continuous real parameters, and $x(s,t)$, $y(s,t)$ and $z(s,t)$ are real-valued continuous functions. **plotsurf** plots a grid of **plotnum0** $\times$ **plotnum1** plot points, and interpolates quadrilaterals between the plot points. Although the function **plotsurf** has four distinct calling syntaxes, only one is illustrated here. To learn about the others, please refer to the *Macsyma Reference Manual* .

plotsurf($[[x1, y1, z1]$**,...,**$[xn, yn, zn]]$**,**s, slo, shi, t, tlo, thi**)**, where

- The first argument represents n surfaces with n triples of expressions. Each expression evaluates to a floating-point number and may reference the variables s and t.
- s and t are the parameters used to specify the surface.
- *slo* and *shi* give the lower and upper limits of the parameter s.
- *tlo* and *thi* give the lower and upper limits of the parameter t.

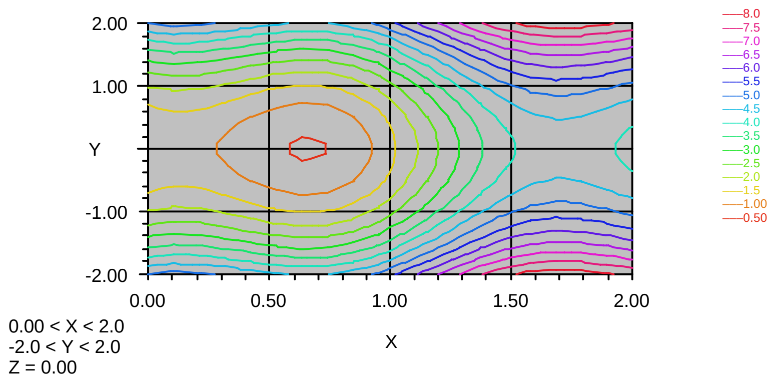

Figure 8.11: A Contour Plot

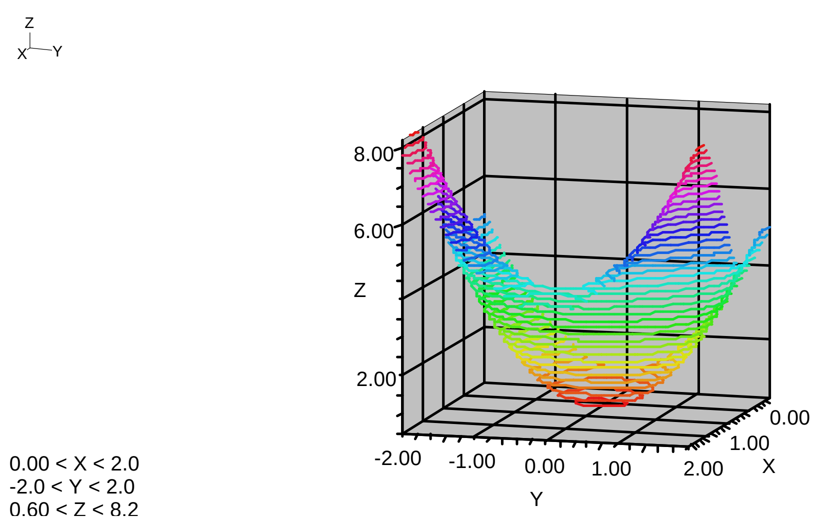

Figure 8.12: A Contour Plot in Three Dimensions

```
(c6)block([equalscale:true,plotnum0:17,plotnum1:25,  title:"A Torus"],
  plotsurf([[(3+sin(th))*cos(ph), (3+sin(th))*sin(ph), cos(th)]],
           th,0,2*%pi,ph,0,2*%pi,ph,0,2*%pi))$
```

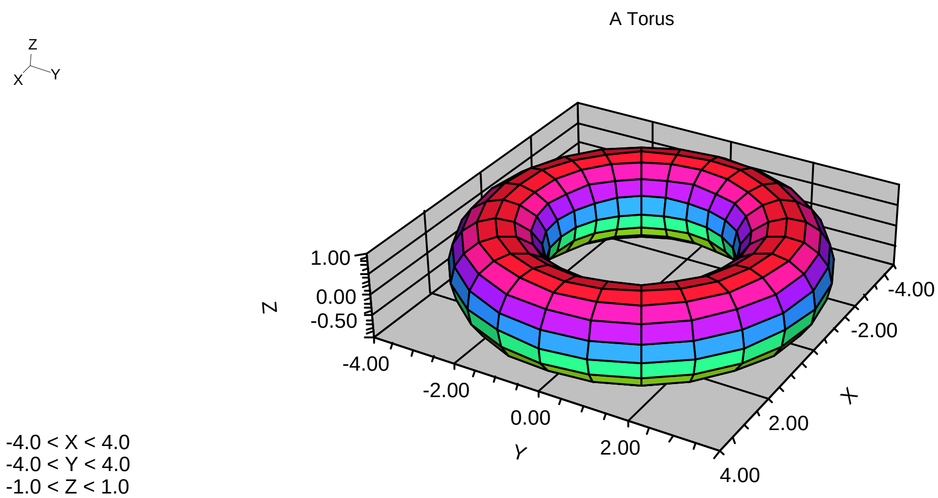

Figure 8.13: A Torus

Setting **plot_tesselation**:3 (the default is 4) causes **plotsurf** to triangulate each quadrilateral in the plotted surface. One important feature of triangular tesselation is that each element is a planar figure. This eliminates certain viewing anomalies which can occur when using nonplanar polygons.

```
(c7)block([equalscale:true,plotnum0:17,plotnum1:25, plot_tesselation:3
 title:"A Torus Tesselated with Triangles"
  plotsurf([[(3+sin(th))*cos(ph), (3+sin(th))*sin(ph), cos(th)]],
          th,0,2*%pi,ph,0,2*%pi,ph,0,2*%pi))$
```

8.3 Changing the Appearance of a Plot

This section describes the commands you can use to change the appearance of a plot. These facilities include changing the scaling, projection, and axes of a plot.

Use the command **replot**(*plotname*) to replot the plot *plotname*. To replot the most recent plot, use the command `replot();`. Examples of this command appear in the subsections below.

8.3.1 Changing a Plot's Scale

Macsyma chooses the scale of plots automatically. In general, the scale is as large as possible, while still allowing everything to fit on your screen. This section introduces the following option variables, which allow you to override the default scale settings.

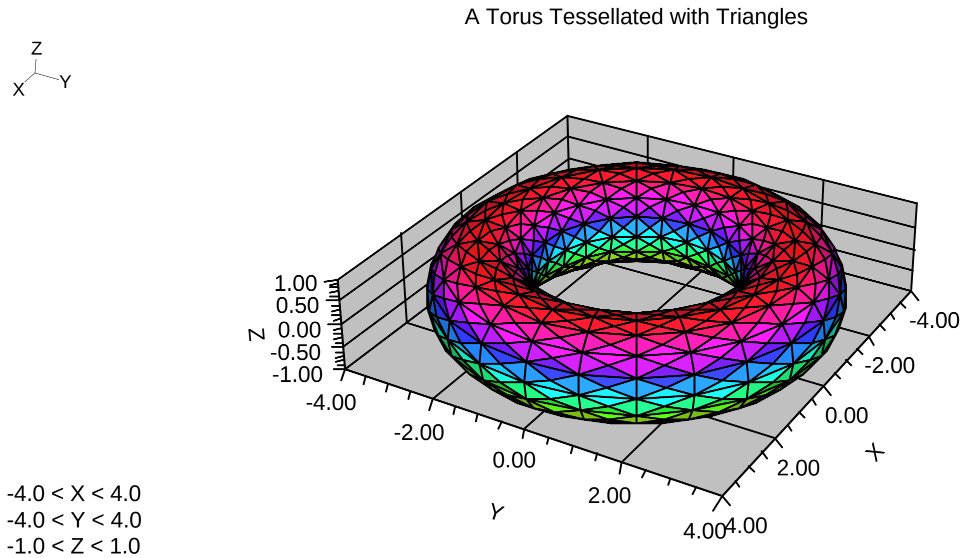

Figure 8.14: A Torus Tesselated with Triangles

- **equalscale** indicates whether or not the scale of the plot should be the same in both directions.
- **xmin**, **ymin**, and **zmin**, when set, override the calculated minimum of the plot's x, y, and z values to determine the minimum coordinate values plotted on the axes.
- **xmax**, **ymax**, and **zmax**, when set, override the calculated maximum of the plot's x, y, and z values to determine the maximum coordinate values plotted on the axes.
- **plot_size** adjusts the size of the plot.

By default, the option variable **equalscale** is **false**. When **equalscale** is **false**, Macsyma can rescale the plot to fill the display. When **true**, the scales of the plot are the same in both directions. Thus, if the display is rectangular, and **equalscale** is **false**, a circle appears as an ellipse; but if **equalscale** is **true**, it appears as a circle.

Examples using **equalscale** appeared in Section 8.2, page 143.

In choosing the scale for a plot, Macsyma looks at the minimum and maximum values it has calculated for x, y, and (for three-dimensional plots) z. By default, the option variables **xmin**, **ymin**, **zmin**, **xmax**, **ymax**, and **zmax** are unbound. If you set any of them to a numeric value, this value overrides the plot's corresponding calculated value.

The option variable **plot_size** (default: 75) controls the size of the plot on the screen.

The following example produces a polar plot of x from 0 to 200. The result is shown in Figure 8.15.

```
(c1) plot(x, x, 0, 200, polar), plotnum:800$
```

Figure 8.16 shows the change in appearance when the same plot is drawn with **equalscale** set to **true**.

```
(c2) plot(x, x, 0, 200, polar), plotnum:800, equalscale:true$
```

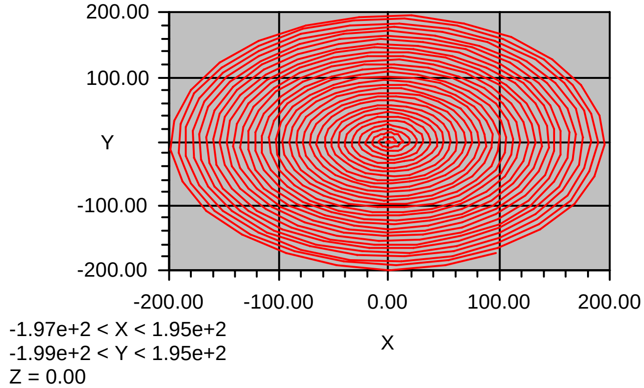

Figure 8.15: Polar Plot of a Spiral

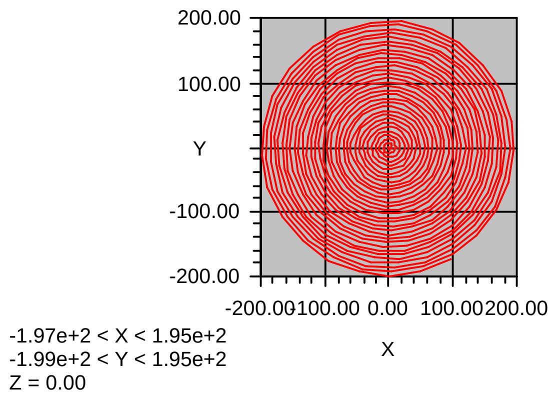

Figure 8.16: Polar Plot of a Spiral with **equalscale** set to **true**

The next example shows how **ymin** and **ymax** can be used to clip a plot. The result is shown in Figure 8.17.

```
(c3) plot(tan(x), x, 0, 10 ), plotnum:800, ymin:-50, ymax:50$
```

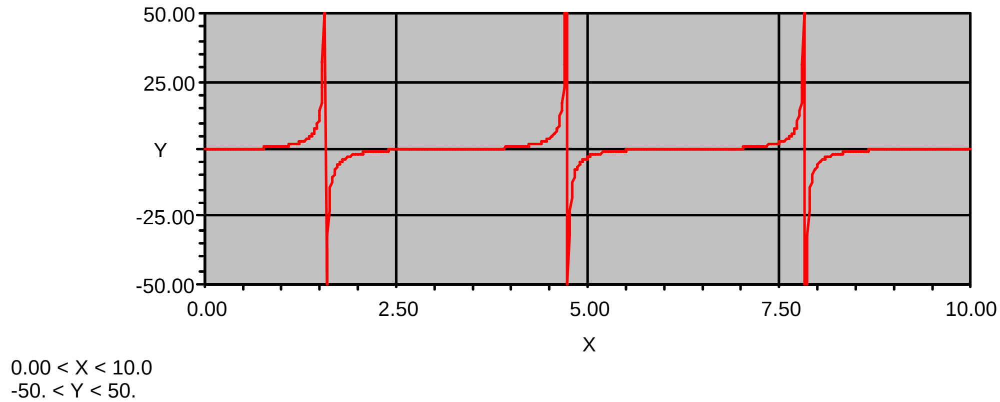

Figure 8.17: Plot with Specified Minimum and Maximum Values

In Macsyma 2.0 and successors, the Front End contains a Camera View dialog box which lets you adjust the scale of a plot and other viewing parameters. For more information, click on `Graphic_Menu` in the Front End Help. In Macsyma 419, this same feature is found in the Graphics Viewer. Click on `[Help]` in the Graphics Viewer for more information.

8.3.2 Changing the Viewpoint of a Three-Dimensional Plot

To change the viewpoint of a plot, set the option variable **viewpt** to a list of three numbers that define the coordinates of the viewpoint.

By default, **viewpt** is unbound. Macsyma determines the perspective view as follows. First, Macsyma determines the two points `min:[`*xmin, ymin, zmin*`]` and `max:[`*xmax, ymax, zmax*`]`. Then, Macsyma calculates **viewpt** as `max + 3*(max - min)`.

The example below replots the plot shown in Figure 8.9, page 143, with a new view point. The result is shown in Figure 8.18.

```
(c4) plot3d(y^2 + cos(4*x)+ 2*x, x, 0, 2, y, -2, 2), viewpt:[100,100,-100]$
```

The option variable **plot_roll** (default: 0) enables you to rotate around the viewing direction.

In Macsyma 2.0 and Macsyma 419, the Camera View dialog box allows you to adjust the viewpoint, roll angle, plot size, and other viewing parameters, after a plot has been generated.

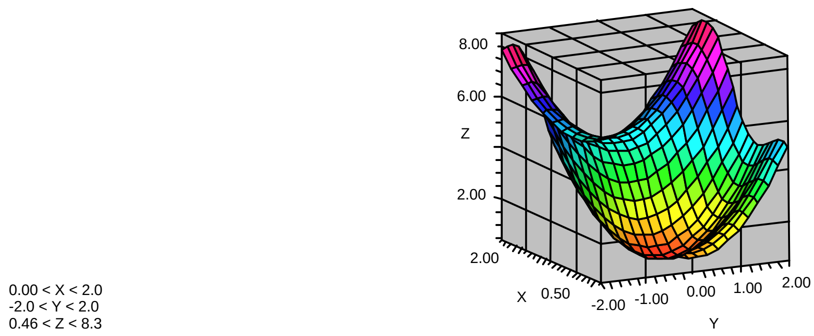

Figure 8.18: Three-Dimensional Plot with **viewpt** Changed

8.3.3 Changing Plot Titles and Axes Labels

This section describes the plotting option variables that you can use to change the titles and axes labels of your plot.

The option variable **title** accepts as an argument a string, which then appears as the title in a plot drawn by Macsyma.

You can label the x and y axes of a Macsyma plot by setting the option variables **xlabel** and **ylabel**. Each option variable accepts as a value a string, which will then appear as the label for the x-axis (the axis for the first variable) or the y-axis (the axis for the second variable) in a 2D or 3D plot drawn by Macsyma.

The line at the bottom of each plot is called the dataline. It displays the maximum and minimum of x and y (and z in 3D plots). Setting the option variable **plotbounds** to **false** suppresses display of the dataline.

The following example displays a plot with default axes. The result is shown in Figure 8.19

```
(c5) expr:40*(x^2*sin(1/x));
                                1   2
(d5)                    40 sin(-) x
                                x
(c6) plot(expr, x, 1.2e-3, 1.0e-2), plotnum:500$
```

Now replot the same plot with a title, labeled x and y axes, no plot bounds information. The result is shown in Figure 8.20.

```
(c7)plot(expr, x, 1.2e-3, 1.0e-2), plotnum:500, title:"A Plot with Modified Labels",
    xlabel:"The First Axis", ylabel:"The Second Axis", plotbounds:false$
```

You can control many aspects of plot axes, labels, grid lines, lighting, and other plot decorations from within the Macsyma Graphics Viewer in Macsyma 419 and from the Front End in Macsyma 2.0.

8.4 Saving Plots

Once you have created some plots, you will want to save them so that they can be displayed later. There are two schemes for saving plots: saving plots in notebooks and saving plots in plot files.

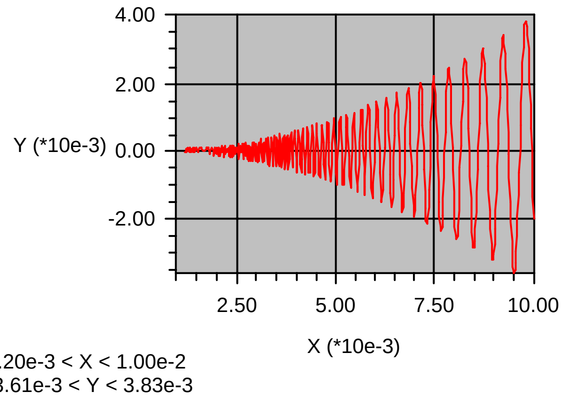

Figure 8.19: Plot with Default Axes

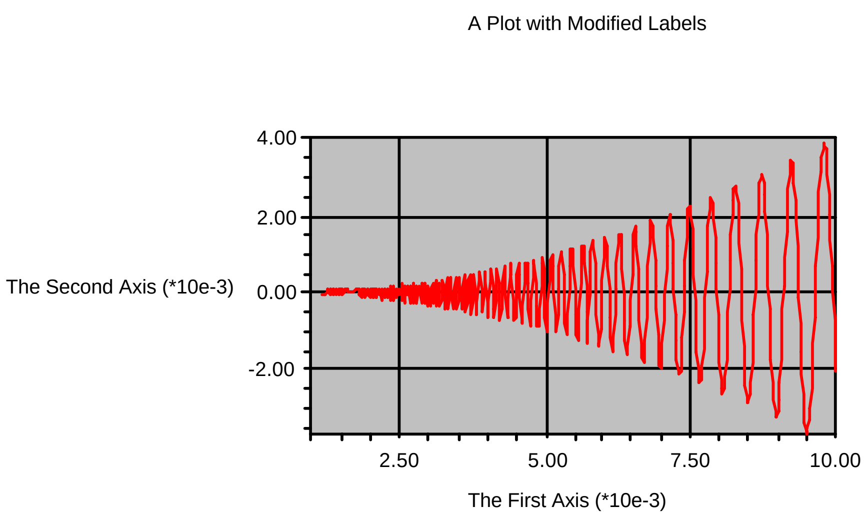

Figure 8.20: Plot With Title, Labeled Axes, and No Plot Bounds Box

8.4.1 Notebook Graphics in Macsyma 2.0 and Successors

If you are using Macsyma on a PC, you will be working in a notebook environment. In a Macsyma notebook, save a plot by saving the notebook in a file. Use the command `[File]|[Save As]`. To copy only the plot, copy it to an empty notebook using `[Edit]|[Copy Section]`, and then save it.

To paste a plot into another Windows application (*e.g.* Microsoft Word for Windows), first set the value of **Clipboard Force Metafile** to **On** in the `[File]|[Option Defaults]` menu then use `[Paste]` in the `[Edit]` menu

To move a plot within a notebook, or to another notebook, use the mouse to select the graphics section; then use `[Edit] | [Copy Section]` followed by `[Edit] | [Paste Section]`.

You can save a plots as a `.pcx`, `.bmp`, `.gif`, or `.rle` file by selecting the plot and clicking on the export button on the tool bar in the Macsyma Front End.

8.4.2 File Based Graphics in Macsyma 419 and Successors

When you create a plot on a system which uses file-based graphics, specifically, in UNIX versions 417 through 420 and successors, Macsyma stores all of the plot information in a file called `macsyma.plt`. If you wish to save the plot information for a given plot, use the function **rename_plot_file** to save it in a file with a different name. Each time you create a new plot, Macsyma stores it in the file `macsyma.plt`, overwriting the previous information in the file.

rename_plot_file(*filename*) *Function*

> Renames the current plot file to *filename*. This command allows you to store information about the most recent plot in a named file; otherwise, the next plot will overwrite the default plot file (`macsyma.plot`).

If you are using UNIX Macsyma 419, you can also save the plot from the Macsyma Graphics Viewer window, using the menu choice `[File]|[Save As]`. Later, when you wish to load the saved plot into the Macsyma Graphics Viewer window, use `[File]|[Open]`.

Macsyma allows you to edit many plot attributes in the Macsyma Graphics Viewer window without regenerating the plot. Experiment with the menu items `[Adjust View...]`, `[Render Settings...]`, and `[Other View Parameters]`.

Chapter 9

Macsyma File Manipulation

This chapter explains how to perform various Macsyma file-related operations.

Although the examples in this chapter introduce many commands and option variables, the scope of this document allows only a limited introduction to Macsyma's file manipulation facilities. To find out more, consult the *Macsyma Reference Manual* .

9.1 Specifying Pathnames

In Macsyma, certain commands require that you specify file pathnames. Operations that require a pathname include:

- Running demonstration files (Section 2.3)
- Saving, editing, and executing files of Macsyma commands (Section 9.3)
- Saving a transcript of your session in a file (Section 9.4.1)

When specifying file pathnames in the Macsyma environment, use the pathname syntax which is specific to your type of platform, enclosed in double quote marks. (DOS-Windows is an exception in that each backslash must be replaced by two backslashes.) For example, if Macsyma is stored on the directory named **macgold**, then the following commands load the Macsyma system file "functs" from the system directory, and the user file "myfile" from the user's home directory (named homedir) on subdirectory mysubdir:

platform	command
DOS-Windows	`load("c:\\macgold\\share\\functs.fas")$`
	`load("c:\\homedir\\mysubdir\\myfile.mac")$`
UNIX	`load("c:/macgold/share/functs.o")$`
	`load("/homedir/mysubdir/myfile.macsyma")$`
VMS	`load("c:[macgold.share]functs.fas")$`
	`load("c:[homedir.mysubdir]myfile.mac")$`
Symbolics	`load("c:>macgold>share>functs.bin")$ (or .ibin)`
	`load("c:>homedir>mysubdir> myfile.macsyma")$`
All Systems	`load("macsyma:share;functs.bin")$`
	`load("macsyma:share;functs")$`
	`load("functs")$`

```
load('functs)$
load(functs)$
```

When specifying pathnames in a Macsyma command file, you can make the Macsyma code more portable to other systems by using the"logical pathname" scheme described below. When Macsyma encounters a pathname specified by a logical pathname, the pathname is automatically converted to the appropriate syntax for the indicated file system.

9.1.1 Logical Pathnames

The pathnames for "All Systems" in the table above are in Macsyma's logical pathname scheme, which works across all platforms. A logical pathname uses a file name convention that is independent of any particular physical host or file server. Logical pathnames make it easy to keep software on more than one file server or operating system. Each version of Macsyma has a translation table that takes the name of the logical path and returns the physical directory and name of the installation.

This scheme uses logical names for the name of the directory where Macsyma is installed, and for filename extensions. Note that the logical directory name "macsyma" translates to the name of the directory where the Macsyma software is located. The file type extension is the logical name ".bin", whose literal interpretation varies across platforms. (See below for more information)

The logical pathname scheme is supported only for directories specified by the option variable **file_search**. This option variable contains a list of those directories to be searched if a file name is incompletely specified. You may add additional directories to this list. The directories may be either logical or literal names. To see the current value of **file_search**, enter `file_search;` . The commands on the last three lines above use Macsyma's **file_search** facility. When you load a file by its name only, Macsyma checks for a file of that name in several directories. The directories currently on the **file_search** list are:

- your home directory (denoted by the symbol **false**)
- `macsyma:library1;`
- `macsyma:library2;`
- `macsyma:matrix;`
- `macsyma:ode;`
- `macsyma:share;`
- `macsyma:tensor;`

For example, if you wish to load a file found in your home directory, you can use either

```
load("myfile.bin")$
load("myfile")$
```

If you don't provide a file specification, Macsyma will search for "binary" first, then "lisp", then "macsyma." In DOS-Windows, these files would appear with suffexes *.bin*, *.lsp*, and *.mac*.

9.1.2 Filename Extensions

Macsyma's logical pathname scheme uses three logical pathname extensions: .macsyma .lisp, and .bin. These translate into literal pathname extensions as shown in Table 9.1.

Logical	———	**Actual Extensions**	———	
Extensions	DOS-Windows	UNIX	VMS	Symbolics
`.macsyma`	`.MAC`	`.macsyma`	`.MAC`	`.macsyma`
`.lisp`	`.LSP`	`.lsp`	`.LSP`	`.lisp`
`.bin`	`.FAS`	`.o`	`.FAS`	`.bin .ibin`

Table 9.1: Literal pathname extensions from logical pathnames

9.2 Customizing Your Macsyma Init File

The first time you enter Macsyma, the system automatically attempts to load your init file. If you do not have one, Macsyma displays the name of the file it was trying to locate.

A Macsyma init file is not required, but you might want to create one to contain definitions for functions, option variable settings, assignments to variables, loading share packages, and so on. Use your system's text editor to create an init file. You should use the name that Macsyma tried to load. The default name for the Macsyma init file is `macsyma:user;mac-init.mac` .

The following sample init file contains customized commands you might like to add to your Macsyma init file.

```
/* This is an empty Macsyma initialization file.
   Macsyma commands which you place in this fie will be executed
   each time you execute the command INITIALIZE_MACSYMA(); .
*/
/*
   The following command will add a directory to the front of the
   list of directories searched when incomplete pathnames are specified.
*/
file_search:cons("c:\\mydir\\develop\\",file_search)$
/*
   The following command will cause the symbol pi to represent as
   double float %pi.
*/
pi:dfloat(%pi)$
/*
   The next command causes Macsyma to display the computation time
   elapsed for each command entered.
*/
showtime:true$
/*
   The following command suppresses the warning message displayed
   each time a floating point number is converted into a bigfloat number.
*/
float2bf:true$
/*
   The following command suppresses the warning message displayed
   each time a floating point number is converted into a rational number.
```

```
*/
ratprint:false$
```

You can use this method to define default settings for your most frequently used option variables.

Macsyma loads your init file automatically when you first enter Macsyma or when you enter Macsyma after a complete initialization, accomplished with `initialize_macsyma();` .

To load your init file at any other time, use the command **load**(*filename*), where *filename* specifies the location of your Macsyma init file. For example, you can load the init file "mac-init.mac" from your home directory as follows:

```
(c1) load("mac-init.mac")$
```

9.3 Submitting Macsyma Batch Jobs

All versions of Macsyma support running Macsyma jobs in batch mode. To run any Macsyma batch job successfully, however, you must be able to predict exactly what input Macsyma will ask for in the course of running your job, and at what point in the job Macsyma will require that input. The following sections illustrate this point and explain how to run batch jobs in DOS-Windows, Unix, and VMS.

The following **batch** command can be used in all versions of Macsyma but only for files in the Macsyma hierarchy.

```
batch("macsyma:user;testing1.mac")$
```

For Unix, a typical batch command would be:

```
batch("c:/homedir/mysubdir/testing1.mac")$
```

A typical VMS batch command would be:

```
batch("c:[homedir.mysubdir]testing1.mac")$
```

A typical DOS-Windows command is:

```
batch("c:\\homedir\\mysubdir\\testing1.mac")$
```

9.3.1 Batch Jobs in Unix

This section presents two examples for running Macsyma batch jobs in Unix. Follow the steps below to run a Macsyma batch job.

Example 1

1. Create a Macsyma command file named `testing1.com` which contains the commands you want included in your batch job. For example,

```
assume(a>0)$
int:(sin(a*x)/x)^2;
integrate(int, x, 0, inf);
quit()$
```

   The line `assume`$(a > 0)$ above answers Macsyma's query about the sign of `a` when it executes the **integrate** command. This line is necessary since a batch file cannot ask questions.

   Alternatively, you can insert `assume_pos:true$` as the first line of the command file above.

2. To execute your file `testing1.com`, create a Unix shell script, here called `testing1.bat`, containing the following line:

```
Macsyma < testing1.com > testing1.log
```

   When your batch job is completed, the file `testing1.log` contains a transcript of the Macsyma session.

3. To make your shell script executable, use the following Unix command:

```
chmod +x testing1.bat
```

4. To execute your Macsyma batch job at 9 p.m., for example, use the Unix command `at 21:00 testing1.bat`. To submit a batch job to Unix more that 24 hours in advance, add the month and date. For example, `at 21:00 feb 28 testing1.bat`

The second example shows an alternative way of submitting a Macsyma batch job in Unix. This slightly more complicated approach allows you to include only Macsyma commands in your command file and to put responses to Macsyma queries in a separate file. You can use a demo or another Macsyma command file, without making any changes to it for the batch process.

Example 2

1. Create a Macsyma command file named `testing2.mac` which contains the commands you want included in your batch job. (This filename must be lower case). Assume that `testing2.mac` has the same contents as `testing1.com` in Example 1, above, except that the line `assume`$(a > 0)$ is missing:

```
int:(sin(a*x)/x)^2;
integrate(int, x, 0, inf);
quit()$
```

2. Create a second file, `testing2.bat`, with the following Macsyma commands shown below. The first command executes the Macsyma command file; the second command answers the **integrate** command's query about the sign of `a`.

```
batch("testing2.mac");
pos;
```

3. To execute your file `testing2.bat`, create a third file called `testing2.exe` containing the following line:

```
Macsyma < testing1.com > testing2.log
```

   When your batch job is completed, the file `testing2.log` contains a transcript of the Macsyma session.

4. To make the file `testing2.exe` executable, use the Unix command

```
chmod +x testing2.exe
```

5. To execute your Macsyma batch job at 9 p.m., for example, use the Unix command

```
at 21:00 testing2.exe
```

   To submit a batch job to Unix more that 24 hours ahead of time, add the month and date to the above command after the time, as follows:

```
at 21:00 jan 31 testing2.exe
```

9.3.2 Batch Jobs in VMS

This section presents two examples for running Macsyma batch jobs in VMS. You can follow the steps below to run a Macsyma batch job.

Example 1

1. Create a Macsyma command file with the filename extension `.COM`, such as `TESTING1.COM`. The first line of the file must be `Macsyma`; the other lines can contain the commands to be executed in your batch job. For example,

```
Macsyma
assume(a>0)$
int:(sin(a*x)/x)^2;
integrate(int, x, 0, inf);
quit()$
```

   The line `assume`$(a > 0)$ above answers Macsyma's query about the sign of `a` when it executes the **integrate** command. Alternatively, you can insert `assume_pos:true$` as the first line of the command file.

2. To submit this job after 9 p.m., for example, use the VMS command

   ```
   SUBMIT/AFTER=21:00   TESTING1.COM
   ```

   When the job is completed, a file called `testing1.log`, created by the system, contains a transcript of the Macsyma session.

The second example shows an alternative way of submitting a Macsyma batch job in VMS. This slightly more complicated approach allows you to include only Macsyma commands in your command file and to put responses to Macsyma queries in a separate file. You can use a demo or another existing Macsyma command file, without making any changes to it for the batch process.

Example 2

1. Create a file with the filename extension `.MAC` to include the Macsyma commands in your batch job. Example: `testing2.mac`.

   ```
   int:(sin(a*x)/x)^2;
   integrate(int,x,0,inf);
   quit()$
   ```

2. Create a second file with the filename extension `.com`, for example `testing2.com`, and include the following:

   ```
   Macsyma
   batch("testing2.mac");
   pos;
   ```

   The line `pos;` above answers Macsyma's query about the sign of `a` when it executes the **integrate** command. **Note**: The filename in the second line above *must* be in lower case.

3. To submit this job after 9 p.m., for example, use the VMS command

   ```
   SUBMIT/AFTER=21:00   TESTING2.COM
   ```

   When the job is completed, a file called `testing2.log`, created by the system, contains a transcript of the Macsyma session.

9.3.3 Batch Jobs in DOS-Windows

Create a Macsyma command file with the filename extension `.mac`, for example `testing.mac`.

```
assume(a>0)$
int:(sin(a*x)/x)^2;
integrate(int, x, 0, inf);
```

The line `assume`$(a > 0)$ above answers Macsyma's query make about the sign of `a` when it executes the **integrate** command. Alternatively, you can insert `assume_pos:true$` as the first line of the command file.

To run the job type:

```
batch("testing.mac")
```

9.4 Saving Your Work

9.4.1 Saving an ASCII Transcript of Your Work

It is a good practice to begin a Macsyma session with the command **writefile**(*filename*) so that your entire Macsyma session will be recorded in the file *filename*. Use the command **closefile** to close a file that you have opened with the **writefile** command. For example:

Open a file to contain a transcript of this Macsyma session.

```
(c1) writefile("Macsyma1.out");
(d1)                   Macsyma1.out
```

The following C-LINEs and D-LINEs are recorded into the transcript file.

```
(c2) a:12$
(c3) b:15$
(c4) c:a*b;
(d4)                    180
```

Close the transcript file, saving it to disk.

```
(c5) closefile();
(d5)                   Macsyma1.out
```

As discussed in Section 9.1 above, you can specify *filename* as a pathname surrounded by double quotes ("), or by using the logical pathname scheme.

If you did not begin the Macsyma session with a **writefile** command, you can still make a record of the session. At any time during a Macsyma session you can issue the **writefile** command, then use the command `playback();` to redisplay all the input and output from the beginning of the session up to the current line. You can then continue your session, with subsequent commands transcripting into the file, or you can close the file with **closefile**. Try the following example.

Begin a Macsyma session without using **writefile** to enable transcripting.

```
(c1) x:100$
(c2) y:200$
(c3) z:x + y;
(d3)                    300
```

Open a file so that you can "play back" the current session into it.

```
(c4) writefile("Macsyma2.out");
(d4)                   Macsyma2.out
```

Redisplay all C-LINEs and D-LINEs since the beginning of the session.

```
(c5) playback();
(c1) x:100$
```

```
(c2) y:200$
(c3) z:x + y;
(d3)                  300
(c4) writefile("Macsyma2.out");
(d4)                 Macsyma2.out
(d5)                 done
```

Close the transcript file, saving it to disk.

```
(c6) closefile();
(d6)                 Macsyma2.out
```

9.4.2 Saving a Macsyma Notebook

Macsyma 2.0 and its successors create re-executable notebooks. After completing a Macsyma session, click on `[File]|[Save As...]` (or `[File]|[Save]` if the notebook was previously saved). To re-execute the notebook, click on `[Edit]|[Select]|[Input]`, to select all input sections. Then click on `[Edit]|[Reexecute]`.

Macsyma 419 and its successors create fancy scripts which are not re-executable. After completing a Macsyma session, click on `[File]|[Save Output As...]` (or `[File]|[Save Output]` if the notebook was previously saved).

9.4.3 Saving Your Computation Environment

To store variable settings, including the expressions associated with C-LABELS and D-LABELS, for use in a later Macsyma session, you can use the command **save**(*filename*, **all**). Execute several commands, noting that variables *eqs*, **globalsolve**, x, and y, as well as the C-LABELS and D-LABELS, are bound to new values during the session.

```
(c1) eqs:[3*x + 2*y - 2, 5*x -12*y + 14]$
(c2) globalsolve:true$
(c3) linsolve(eqs, [x, y]);
                           2       26
(d3)                [x : - --, y : --]
                           23      23
(c4) x;
                         2
(d4)                   - --
                         23
(c5) c1;
(d5)         eqs : [2 y + 3 x - 2, - 12 y + 5 x + 14]
```

Save the variables settings in a file for use in another Macsyma session.

```
(c6) save("macsess.sav", all)$
```

The file is saved in the user's home directory unless you specify another directory. Both **save** and **playback** support specifications to indicate whether all or part of the session should be saved or played back. See the *Macsyma Reference Manual* for more information.

You can restore the information in the **save** file with the **load**(*filename*) command. Begin a new Macsyma session, noting that variables have default settings.

```
(c1) globalsolve;
(d1)                  false
(c2) x;
(d2)                    x
(c3) eqs;
(d3)                   eqs
(c4) c1;
(d4)               globalsolve
```

Now load the file containing the variables settings you saved in a previous session.

```
(c5) load("macsess.sav")$
macsess.sav being loaded.
```

Notice that now the variable values have changed.

```
(c6) globalsolve;
(d6)                  true
(c7) x;
                          2
(d7)                    - --
                          23
(c8) eqs;
(d8)           [2 y + 3 x - 2, - 12 y + 5 x + 14]
(c9) c1;
(d9)        eqs : [2 y + 3 x - 2, - 12 y + 5 x + 14]
```

Chapter 10

Translating Macsyma Expressions to Other Languages

It is possible to translate Macsyma expressions to other languages. The primary facility for doing this is the **gentran** function. Gentran can be used to generate both FORTRAN and C code. For more information, see **gentran** in the *Macsyma Reference Manual* .

10.1 Translating Expressions to FORTRAN

You can convert Macsyma expressions to legal FORTRAN code using the special forms **fortran** and **gentran** and the option variables discussed below. For example, consider the following:

```
(c1) solve(x^2+3*y = 17, x);
(d1)         [x = - sqrt(17 - 3 y), x = sqrt(17 - 3 y)]
```

To convert the first solution given above into FORTRAN code, type

```
(c2) fortran(first(%))$
     x = -sqrt(17-3*y)
```

You can also convert a matrix into a series of FORTRAN assignment statements. For example,

```
(c3) m:matrix([3, 5, 4], [1, 2, 7]);
                    [ 3  5  4 ]
(d3)                [         ]
                    [ 1  2  7 ]
(c4) fortran(m)$
      m(1, 1) = 3
      m(1, 2) = 5
      m(1, 3) = 4
      m(2, 1) = 1
      m(2, 2) = 2
      m(2, 3) = 7
```

Two option variables, **fortindent** and **fortspaces**, determine the format of the FORTRAN output. The variable **fortindent** controls indentation. The default setting is 0, producing a normal indentation of 6 spaces. Resetting it to a positive value increases indentation.

Add three spaces to the normal indentation of six; compare the result to the output of (c2) above

```
(c5) fortran(part(d1, 2)), fortindent:3$
         x = sqrt(17-3*y)
```

The variable **fortspaces** is initially set to **false**. Setting it to **true** causes **fortran** to fill out to 80 columns using spaces.

10.2 Translating Macsyma Expressions to C

See the Gentran package in the *Macsyma Reference Manual* for information on translating Macsyma packages to C.

10.3 Typesetting Macsyma Expressions with TEX

TEXis a text formatter for mathematical equations. Macsyma lets you convert Macsyma expressions automatically to TEX format using the commands **tex** and **write_tex_file**.

The function **tex** accepts one or more Macsyma expressions and converts them into TEX. If **write_tex_file** has been called, and there is an open TEX file, then **tex** writes to that file. Otherwise the TEX output is sent to the terminal. See the *Macsyma Reference Manual* for more information about converting Macsyma expression to TEX.

Consider the following integral:

```
(c1) 'integrate(%e^-(x^2), x) = integrate(%e^-(x^2), x);
                  /      2
                  [   - x       sqrt(%pi) erf(x)
(d1)              I %e     dx = ----------------
                  ]                    2
                  /
```

To convert this expression to TEX code, type

```
(c2) tex(%);
% 'integrate(%e^-x^2,x) = sqrt(%pi)*erf(x)/2
$$ \int e ^ {-{ x^{2} }}}{\,dx}= {{\sqrt{\pi}\,\left({\rm erf}x
 \right)}\over{2}}$$
(d2)                    done
```

Chapter 11

Using the Macsyma Programming Language

Macsyma is a full programming language as well as providing the mathematical capabilities that you have seen in previous chapters. Macsyma accommodates many classical programming structures, including conditional statements and loops. You can write procedures for a variety of purposes, such as numerical techniques and combinatorial search.

Writing procedures in Macsyma for numerical techniques differs somewhat from writing in a purely numerical language, such as FORTRAN, however. For example, Macsyma requires no type declarations, and floating-point numbers do not result from certain calculations, though they are contagious.

The examples in this chapter cover many aspects of using Macsyma as a programming language. If you wish to learn more about programming in Macsyma, see Chapter 12 and also consult the *Macsyma Reference Manual* .

11.1 Using Conditionals

To execute expressions conditionally, use the **if** statement:

if *condition* **then** *expression1* **else** *expression2*

The **if** statement returns the value of *expression1* if the *condition* is **true** and the value of *expression2* if the *condition* is **false**.

The arguments *expression1* and *expression2* can be any kind of Macsyma expression, including another nested **if** statement, or a group of expressions known as a compound statement (See **compound statements**, page 168).

The *condition* argument is a predicate expression whose relational and logical operators always evaluates to either **true** or **false**. Table 11.1 lists Macsyma's predefined logical operators.

```
(c1) pulse(x) := if x > 2 or x <= 0 then :0 else :1$
(c2) pulse(3/2);
(d2)                 1
(c3) pulse(5);
(d3)                 0
```

Omitting the **else** clause of an **if** statement is the same as specifying **else false**.

Logical Operator	*Description*
`>`	greater than
`=`	equal to
`#`	not equal to
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`and`	logical and
`or`	logical or
`not`	logical not

Table 11.1: Predefined Logical Operators

11.2 Using Iteration

Use the **for** statement to perform iteration. Macsyma provides several variations of the **for** statement which are similar to other programming languages. (For example, FORTRAN, Algol, and PL/1). The common variants shown below differ only in their terminating conditions:

for *variable*:*initial-value* **step** *increment* **thru** *limit* **do** *body*

for *variable*:*initial-value* **step** *increment* **while** *condition* **do** *body*

for *variable*:*initial-value* **step** *increment* **unless** *condition* **do** *body*

for *variable* **in** *list* **while** *condition* **do** *body*

for *variable* **in** *list* **unless** *condition* **do** *body*

The *initial-value*, *increment*, *limit*, and *body* can be any expressions. To iterate over several statements, you can make the *body* a compound statement (See **compound statements**, page 168) or a block statement (See **block statement**, page 169). The *condition*, or predicate, is the same as that described on page 165 for the **if** statement. If *increment* is 1, you can omit the `step 1` from the statement. The statement executes until until the control variable exceeds the *limit* of the **thru** specification, or the *condition* of an **unless** clause is **true** or a **while** clause is **false**. For more details on the use of the **for** statement, see the *Macsyma Reference Manual* .

Every Macsyma command returns a value. The value normally returned by a **for** statement is `done`.

```
(c1) s:0$
(c2) for i:1 step 2 thru 7 do s:s + i;
(d2)                  done
(c3) s;
(d3)                   16
```

The command **ldisplay**(exp_1, ..., exp_n) is useful in **for** statements and blocks (see page 169) to display intermediate results. The **ldisplay** command displays equations whose left side is exp_i and whose right side is the value of the expression. Each equation has an intermediate label, which is added to the system variable **labels**, which is a list of all line labels which are currently bound to an expression.

```
(c4) s:0$
(c5) for i:1 step 2 thru 7 do ldisplay(s:s + i);
(e5)                  s = 1
(e6)                  s = 4
(e7)                  s = 9
(e8)                 s = 16
(d8)                  done
```

Macsyma provides several commands similar to **ldisplay**. These include **display**, **ldisp**, and **disp**. Unlike **display**, **ldisplay** provides a means of accessing the results through the values of the intermediate line labels generated during the execution of the **ldisplay** command. Use **ldisp** to display values only, without the equation format provided by **ldisplay**. Use **disp** or **display** if the intermediate line labels are not needed.

Another useful display command, **print**, appears in the answer to this chapter's first practice problem on page (288).

Macsyma also allows you to nest **for** loops.

```
(c9) poly:0$
(c10) for i:1 thru 5 do
      for j:i step -1 thru 1 do
          poly:poly + i*x^j$
(c11) poly;
                   5      4       3       2
(d11)           5 x  + 9 x  + 12 x  + 14 x  + 15 x
```

The remaining examples in this section illustrate the **for** statement in conjunction with various keywords. Note particularly the last example, (c21) through (d25), which simulates the **taylor** command.

You can use the keyword **in** to iterate through a list.

```
(c12) for f in [a, b] do ldisp(f);
(e12)                   a
(e13)                   b
(d13)                  done
```

Certain Macsyma commands, such as **solve**, return their results in a list. To check solutions, use the keyword **in** to look at every element in a list.

```
(c14) eq:a*x^2 + b*x + c;
                        2
(d14)                a x  + b x + c
(c15) quadraticsols:solve(eq, x);
```

```
                          2              2
                    sqrt(b  - 4 a c) + b       sqrt(b  - 4 a c) - b
(d15)      [x = - --------------------, x = --------------------]
                            2 a                          2 a
(c16) for c in quadraticsols do ldisp(ev(eq, c, ratsimp));
(e16)                 0 = 0
(e17)                 0 = 0
(d17)                 done
```

The following example illustrates the use of the **while** keyword.

```
(c18) s:0$
(c19) for i:1 while i <= 10 do s:s + i;
(d19)                 done
(c20) s;
(d20)                  55
```

Notice that you can use a **for** statement with the **unless** keyword to simulate the **taylor** command.

```
(c21) taylor(exp(sin(x)), x, 0, 7);
                       2    4    5    6     7
                      x    x    x    x     x
(d21)/T/      1 + x + -- - -- - -- - --- + -- + . . .
                      2    8    15   240   90
(c22) series:1$
(c23) term:exp(sin(x))$
(c24) for p:1 unless p > 7 do
          series:series + subst(x = 0, term:diff(term, x)/p)*x^p;
(d24)                 done
(c25) trunc(series);
                       2    4    5    6     7
                      x    x    x    x     x
(d25)         1 + x + -- - -- - -- - --- + -- + . . .
                      2    8    15   240   90
```

11.3 Compound Statements

To execute a sequence of statements in a context where only a single statement is permitted (for example, in an **if** or **for** statement), group them into a compound statement by separating the statements with commas and enclosing the entire group in parentheses.

The value returned by a compound statement is the value of the last statement in the group.

The example below uses a compound statement to rewrite the commands shown in (c21) through (c25) of Section 11.2 to simulate the **taylor** command. Macsyma returns the series because it is the value of the last statement.

```
(c1) (series:1, term:exp(sin(x)),
    for p:1 unless p > 7 do
           (term:diff(term, x)/p,
            series:series + subst(x = 0, term)*x^p), trunc(series));
                      2    4    5    6     7
                     x    x    x    x     x
(d1)          1 + x + -- - -- - -- - --- + -- + . . .
                     2    8    15   240   90
```

You can use the system variable **%%** in the n^{th} statement to refer to the value of the $(n-1)^{th}$ statement.

```
(c2) diff_at(exp, var1, var2) := (diff(exp, var1), subst(var1 = var2, %%))$
(c3) diff_at(sin(x),x,y + 3);
(d3)                    cos(y + 3)
```

11.4 Making Program Blocks

Program blocks are similar to compound statements, but they also provide a way to tag statements within the block and to assign values to variables that are local to the block. Use the **block** statement:

block($[var_1, \ldots, var_n]$, $statement_1, \ldots, statement_n$)

to establish a program block. Each var_i is a variable name with an optional assignment that is local to the **block**. Each *statement* can be any Macsyma expression. If you don't need any local variables, you can omit the variable list. For more information about the **block** statement, see the *Macsyma Reference Manual* .

Generally the value returned by **block** is the value of the last statement in the block. Alternatively, you can use the command **return**(*exp*) to explicitly exit the block and return a value. An example of the **return** command appears on page 171.

As with compound statements, you can use the system variable **%%** in the n^{th} statement of the block to refer to the value of the $(n-1)^{th}$ statement.

The following example rewrites the compound statement shown in (c1) of Section 11.3, which simulates the **taylor** command, as a function using a **block** statement. Notice that the $[var_1, \ldots, var_n]$ part of the **block** statement is not used here, and thus no local variable is declared. However, the first command in the **block** assigns a value to *series*, and so this variable remains bound after the **block** is exited.

```
(c1) mytaylor(expr, var, point, hipower) :=
    block(series:subst(point, var, expr),
              for i:1 thru hipower
              do (expr:diff(expr, var)/i,
                  series : series
                   + (var - point)^i*subst(point, var, expr)),
              trunc(series))$
```

Test the function on the following input.

```
(c2) mytaylor(exp(sin(x)), x, 0, 7);
```

```
                      2    4    5    6     7
                     x    x    x    x     x
(d2)          1 + x + -- - -- - -- - --- + -- + . . .
                     2    8    15   240   90
```

The variable *series* was assigned a value in the ($statement_1$, ..., $statement_n$) part of the **block** statement, but is not declared, and so remains bound.

```
(c3) series;
                  7    6     5    4    2
                 x    x     x    x    x
(d3)             -- - --- - -- - -- + -- + x + 1
                 90   240   15   8    2
```

The next example rewrites the function `mytaylor` again. This time *series1* is declared as a local variable in the **block**. Notice that now the variable is unbound after the **block** is exited.

```
(c4) mytaylor1(expr, var, point, hipower) :=
    block([series1],
              series1:subst(point,var,expr),
              for i:1 thru hipower
              do (expr:diff(expr,var)/i,
                  series1 : series1
                + (var - point)^i*subst(point, var, expr)),
              trunc(series1))$
```

Test the function on the following input

```
(c5) mytaylor1(exp(sin(x)), x, 0, 7);
                      2    4    5    6     7
                     x    x    x    x     x
(d5)          1 + x + -- - -- - -- - --- + -- + . . .
                     2    8    15   240   90
```

Since the variable *series1* was declared in the **block** statement, it is local to the **block** and therefore not bound outside it

```
(c6) series1;
(d6)                 series1
```

11.5 Tagging Statements

When you are writing complex procedures that involve many statements, you can tag the statements in order to transfer control among them. To do so, use the **go** statement. The command **go**(*tag*) transfers control to the statement of a block that is tagged with *tag*.

To tag a statement precede it by a symbol, as another statement in the **block**. This is illustrated in the following example.

Rewrite the `mytaylor` function defined in Section 11.4, this time using a **go** statement instead of a **for** statement to perform iteration; notice that the *tag* value is called `loop`

```
(c1) mytaylor2(expr, var, point, hipower) :=
    block([series2, i:0],
              series2:subst(point, var, expr),
              loop, if i >= hipower then return(trunc(series2)),
                i : i + 1,
                expr:diff(expr, var)/i,
                series2 : series2
                      + (var - point)^i*subst(point, var, expr),
              go(loop))$
```

Test the new function

```
(c2) mytaylor2(exp(sin(x)), x, 0, 7);
                          2    4    5    6     7
                         x    x    x    x     x
(d2)          1 + x + -- - -- - -- - --- + -- + . . .
                         2    8    15   240   90
```

Note: You can use **go** only to transfer to a *tag* within the block containing that **go** statement.

11.6 Writing Recursive Functions

Like many other programming languages, Macsyma supports recursion. You have already seen an example of a recursively defined array compute factorials (Section 3.5, on page 29). The example below illustrates a recursive function that computes factorials.

```
(c1) myfactorial(n) := if n = 0  then 1
                   else n*myfactorial(n - 1)$
```

The **trace**(*functionn*$_1$, ..., *functionn*$_n$) command is useful for monitoring the execution of one or more functions. When Macsyma encounters a function *function* during a computation, it displays the function's name and arguments upon entry, the function's name and return value upon exit, and a count of the levels of recursion.

To stop tracing a function, use the command
untrace(*functionn*$_1$, ..., *functionn*$_n$). The command `untrace();` removes tracing from all functions.

The **trace** command can monitor the execution of `myfactorial` for $n = 4$.

```
(c2) trace(myfactorial)$
(c3) myfactorial(4);
1 enter myfactorial [4]
 2 enter myfactorial [3]
```

```
  3 enter myfactorial [2]
   4 enter myfactorial [1]
    5 enter myfactorial [0]
    5 exit  myfactorial 1
   4 exit  myfactorial 1
  3 exit  myfactorial 2
 2 exit  myfactorial 6
1 exit  myfactorial 24
(d3)                    24
(c4) untrace(myfactorial)$
```

Another important use of functions is in the definition or extension of algorithms that manipulate formulas. Consider a symbolic differentiation algorithm whose four basic rules are

$$\frac{dx}{dx} = 1 \qquad \frac{dx}{dy} = 0 \text{ when } x \neq y$$
$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} \qquad \frac{d(uv)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$$

You can implement this algorithm using the recursive function given below. To implement the first two differentiation rules, you should first check to see if the expression is an **atom**. An atom is a simple data structure with no component parts, such as a number, a string, or a symbol. The command **atom**(*exp*) returns **true** if *exp* is an atom, and **false** otherwise.

```
(c5) atom(dummyvar);
(d5)                   true
(c6) atom(dummyvar + 1);
(d6)                   false
```

In implementing the first two differentiation rules, we know that if **atom** returns **false**, the expression is a composite expression.

To implement the last two rules, use the **part** command (see Section 4.5, page 45) to determine whether the leading operator obtained by `part(`*expr*`, 0)` represents addition or multiplication. Based on this value, you can apply the appropriate rule. If the leading operator is neither `+` nor `*`, the procedure returns a differential form as a result.

```
(c7) newdiff(expr,var) :=
    if atom(expr)
        then if expr = var
         then 1
         else 0
    else if part(expr, 0) = "+"
         then newdiff(part(expr, 1), var) + newdiff(part(expr, 2), var)
         else if part(expr, 0)="*"
          then part(expr, 2)*newdiff(part(expr, 1), var)
               + part(expr, 1)*newdiff(part(expr, 2), var)
          else 'newdiff(expr, var)$
```

```
(c8) newdiff((x + 1)*(x+2) + g(x), x);
(d8)               newdiff(g(x), x) + 2 x + 3
```

11.7 Functional Arguments and Formal Parameters

To pass a function as an argument to another function you need only give its name in the argument list of the call. You can then use this function in the called function by following the name of the corresponding formal parameter with a parenthesized list of arguments.

To pass a subscripted function give the name followed by the subscripts in brackets.

To pass an array, give the name of the array in the argument list. You can then reference the arrays by subscripting the corresponding formal parameter.

When you know the name of the function, you should precede the name with a single quote to prevent evaluation. In this way, you can avoid potential confusion with any variables which might have the same name.

To assign a value to a formal parameter of a function so that the corresponding actual parameter gets changed, and remains changed, when the function is exited, you can use the operator "**::**" rather than the operator "**:**". Using "**::**" in this way is discouraged, however, because the assignment takes place in the environment of the program and not in the environment of the caller.

In this example, the variable a is initially unbound.

```
(c1) a;
(d1)                     a
```

A first attempt to define a function f for binding the value 2 to its argument; using the "**:**" operator does not work.

```
(c2) f(x) := x:2$
(c3) f('a);
(d4)                     2
(c4) a;
(d4)                     a
```

Using the "**::**" operator allows the function *ff* to bind the value 2 to its argument.

```
(c5) ff(x) := x::2$
(c6) ff('a);
(d7)                     2
(c7) a;
(d7)                     2
```

For more details about functional arguments and formal parameters, see the *Macsyma Reference Manual* .

11.8 Practice Problems

Using the commands that you have learned about in this chapter, solve the following problems. Answers appear on page 288.

Solve the first six problems using Macsyma lists `[a1, a2, a3]` to represent the three-dimensional vectors

```
a = a1 e  + a2 e  + a3 e
        x       y       z
```

Problem 1. Define the functions `dot(u, v)` and `cross(u, v)` to return the dot (scalar) and cross (vector) product of two vectors `u` and `v`, respectively. (Macsyma has a dot operator "`.`", and a vector cross product operation in the vector calculus package **vect**.)

Problem 2. Use your definitions to prove that: $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) - \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) + \mathbf{c}(\mathbf{a} \cdot \mathbf{b}) = 0$ for arbitrary vectors `a`, `b`, and `c`.

Problem 3. Define the functions `my_grad`, `my_div`, and `my_curl` to return the gradient of a scalar function f, and the divergence and curl of a vector function v, respectively. (Macsyma has commands **grad**, **div**, and **curl** in the vector calculus package **vect**.)

Problem 4. Show that `my_grad(f)` at the point $(1, -2, -1)$ is $(-12, -9, -16)$ if $f = 3x^2y - y^3z^2$.

Problem 5. Show that `my_div(a)` at the point $(1, -1, 1)$ is -3 if `a` is the vector

```
  2           3  2       2
[x  z ,  -2 y  z  , x y  z]
```

Problem 6. Show that `my_curl(a)` at the point $(1, -1, 1)$ is $(0, 3, 4)$ if `a` is the vector

```
    3       2             4
[x z  , -2 x  y z , 2 y z  x]
```

Problem 7. Write a function `my_runge_kutta` to solve the following differential equations

$$\begin{aligned}
\frac{dx}{dt} &= -3(x-y) \\
\frac{dy}{dt} &= -xz - y + rx \\
\frac{dz}{dt} &= xy - z
\end{aligned}$$

subject to the constraints: $x(0) = z(0) = 0$, $y(0) = 1$, and $r > 26$. (The Macsyma commands **runge_kutta** and **ode_numsol** would normally be used to solve this problem.)

Chapter 12

Advanced Programming Topics

This chapter covers some of the more advanced aspects of Macsyma programming, including efficient use of subscripted functions, debugging functions, writing macros, translating and compiling functions, and hints for writing libraries. These topics are covered in greater depth in the *Macsyma Reference Manual*. Beginning Macsyma users or those with little programming experience can skip this chapter. This material is more appropriate for users with extensive programming experience in Macsyma.

12.1 Functional Evaluation Revisited

It is useful to briefly review the concept of a *function* in Macsyma. Loosely speaking, a Macsyma function is a procedure all of whose arguments are evaluated once prior to execution of the procedure. (This is contrasted with a *special form*, which can evaluate its arguments in a different manner.) Consider the following example:

```
(c1) f(x,n):=x^n;
                                                n
(d1)                               f(x, n) := x
(c2) f(y,3);
                                        3
(d2)                                   y
(c3) (g:f,w:v)$
(c4) g(w,4);
                                        4
(d4)                                   v
```

In `(c1)`, a function of two arguments is defined. This function is called in `(c2)`. In `(c3)`, some variables are bound to show the kinds of evaluation which occur. In `(c4)`, the call to an undefined function **g** results in the following steps:

- The identity of **g** is determined. Since there is no function definition associated with **g**, it is evaluated, resulting in **f**. The result is the intermediate form **f(w,4)**. (It is necessary to determine that **f** is indeed a function, not a special form, before evaluation of the arguments can take place.)
- Since **f** is a function, all of its arguments are evaluated; then the function definition for **f** is applied to the evaluated arguments, yielding the expected result.

12.2 Lambda Forms

12.2.1 Evaluation of lambda Forms

A **lambda** form is a "temporary" or "unnamed" function. See the *Macsyma Reference Manual* for more information. This Section explains the evaluation of **lambda** forms and the substitution of **lambda** forms for specified operators. Consider the following example:

```
(c1) lambda([var1,var2],var1^var2)(y,3);
                                        3
(d1)                                   y
(c2) f:lambda([var1,var2],var1^var2);
                                                      var2
(d2)                     lambda([var1, var2], var1    )
(c3) f(y,3);
                                        3
(d3)                                   y
(c4) (g:'f,w:v)$
(c5) g(w,4);
                                        4
(d5)                                   v
```

This example is almost identical to the previous one. A "temporary" **lambda** form is defined, used, and discarded in `(c1)`. In `(c2)`, a **lambda** form is bound to the variable `f`, and is executed in `(c3)` and `(c5)` following the previously described evaluation procedures. The symbol **g** is bound to **'f** in `(c4)` to show that the operator is evaluated, as well as the arguments.

12.2.2 Using opsubst and lambda Forms to Modify Expressions

You may want to modify an expression by manipulating some of the functional forms in place. For example, you might want to write a function which scans an expression and returns a list of all of the arguments of the **sin** function. You have several options, but a particularly efficient and elegant way is to use the **opsubst** command to scan the expression and a **lambda** form to obtain the arguments of the desired function. The following example illustrates this approach:

```
(c1) a*sin(3*x)+b*(4*sin(2*t)-6*sin(x))+c*(cos(2*y)*sin(3*w)/sin(6*s))+5;
     c sin(3 w) cos(2 y)
(d1) ------------------- + a sin(3 x)
          sin(6 s)
                              + b (4 sin(2 t) - 6 sin(x)) + 5
(c2) (arguments:[],opsubst('sin=lambda([arg],if not(member(arg,arguments))
                           then push(arg,arguments),funmake('sin,[arg])),%));
     c sin(3 w) cos(2 y)
(d2) ------------------- + a sin(3 x)
          sin(6 s)
                              + b (4 sin(2 t) - 6 sin(x)) + 5
(c3) arguments;
```

```
(d3)              [3 w, 6 s, 3 x, x, 2 t]
```

This example uses the **push** macro, which takes as arguments, an element and the name of a list, adds the element to the front of the list, and binds the resulting list to the indicated name. The test expression is defined in `(c1)`.

The compound statement in `(c2)` binds an empty list to the symbol `arguments`, uses **opsubst** to scan the expression for **sin** functions and substitutes a **lambda** form for the **sin** operator whenever it encounters one. (**opsubst** differs from **subst** in that it only performs the substitution when its target is used in the functional position of an expression.) The substitution works in the following way:

- **opsubst** scans the expression "top down" for a **sin** operator.
- When it finds one, it substitutes the **lambda** form for **sin**, thereby converting an expression of the form `sin(arg)` into the intermediate form `lambda(...)(arg)`.
- After the **lambda** form is evaluated, it checks to see if its argument is already in the list of arguments. If not, it adds the argument to the list. Finally, it reconstructs the **sin** form. (**Note:** Failure to reconstruct the **sin** form at the end might result in either the generation of an illegal expression or in greatly increased execution time.)
- The procedure repeats until the entire expression has been scanned.

In `(c3)`, we evaluate the variable *arguments*, whose value was modified as a side-effect of the **opsubst lambda** substitution.

12.3 Subscripted Functions

A subscripted function, *e.g.*, `f[n](x)`, is a Macsyma form whose functional part `f[n]` is a subscripted object. An example of a subscripted function is `f[n](x)`, where x is the argument and n is the index. In the following example, we define a subscripted function and explain how it works.

```
(c1) f[n](x):=expand((x+1)^n);
                                                  n
(d1)                       f (x) := expand((x + 1) )
                            n



(c2) f[3](y);
                                3      2
(d2)                           y  + 3 y  + 3 y + 1



(c3) f[3];
                                     3      2
(d3)                     lambda([x], x  + 3 x  + 3 x + 1)
(c4) f[4];
```

```
                                    4       3       2
(d4)                 lambda([x], x   + 4 x   + 6 x   + 4 x + 1)



(c5) arrayinfo(f);
(d5)               [hashed, 1, [3], [4]]
```

`(c1)` defines a subscripted function. `(c2)` calls the function with the specified index and functional argument. `(c3)` evaluates the *array element* created by the command `(c2)`, and the result shows that a **lambda** form has been generated and assigned to this array slot. `(c4)` evaluates the fourth element of the array, which causes the displayed **lambda** form to be generated and assigned to that element. `(c5)` shows that the array `f` has been set up as a hashed array with values specified for the indices 3 and 4. (The 1 in the list `(d5)` indicates the dimension of the array.)

This example steps used to define and are executed when a subscripted function is defined and then called:

- The function definition generates a "template" which describes the body of the function in terms of the array indices and functional parameters.
- When you first reference an specific array element, a **lambda** form is generated and is assigned to the array element, and is applied to any functional arguments.
- When you call a subscripted function whose corresponding array element is already assigned, the associated **lambda** form is invoked on the functional arguments.

Note that the call to **expand** in the function definition automatically appears when Macsyma constructs the **lambda** form. Subsequent calls to a subscripted function with previously-defined indices but new functional arguments execute swiftly because the **expand** has been done once and for all. A disadvantage of this method is that the partial evaluation of the function definition inhibits certain operations on the functional arguments. More will be said about this later on page 180.

12.3.1 Example: Incorporating A Definite Integral Into A Function Definition

This example further illustrates some other differences between regular and subscripted functions.

`(c1)` defines a function which computes a definite integral. The integral is recomputed every time the function you invoke `g`. This process is inefficient if you can compute the definite integral by evaluating a closed form result at its lower and upper limit.

```
(c1) g(n,x):=integrate(t^n,t,1,x);
                                         n
(d1)                  g(n, x) := integrate(t , t, 1, x)
(c2) g(3,y);
                                   4
                                  y    1
(d2)                              -- - -
                                  4    4
```

The next definition avoids the aforementioned shortcoming by using the *result* of the integration in the body of the definition. However, in this case, the integrator needs to know the signs of certain quantities when computing the integral for general n. However, this method is more efficient than the previous one since the integration does not need to be carried out each time the function is called.

```
(c3) gg(n,x):=''(integrate(t^n,t,1,x));
Is   n   positive, negative, or zero?
p;
Is   x   positive or negative?
p;
Is  x - 1   positive, negative, or zero?
p;
                                        n + 1
                                       x           1
(d3)                      gg(n, x) := ------ - -----
                                       n + 1    n + 1
(c4) gg(3,y);
                                     4
                                    y    1
(d4)                                -- - -
                                    4    4
```

The next command does the same thing using the subscripted function mechanism. The integral is evaluated the first time you call the subscripted function on a new index. That value of the index is bound *before* the computation, and you can compute the resulting integral without requiring sign information. The result is bound as a **lambda** form to the array index, so that you need not recompute the antiderviative when you call with this index. In addition to avoiding most of the sign queries, this method provides the correct answer when n=-1.

```
(c5) g[n](x):=integrate(t^n,t,1,x);
                                          n
(d5)                    g (x) := integrate(t , t, 1, x)
                         n
(c6) g[3](y);
                                     4
                                    y    1
(d6)                                -- - -
                                    4    4
(c7) g[3];
                                          4
                                         x    1
(d7)                         lambda([x], -- - -)
                                         4    4
(c8) g[-1];
is   x - 1   positive, negative, or zero?
p;
(d8)                         lambda([x], log(x))
```

Be careful when you test the functional arguments. Consider the following example which recursively defines a family of functions, each of which contains a removable singularity at `x=0`. The straightforward subscripted function definition generates a "`Division by 0`" error when evaluated at `x=0`.

```
(c1) f[n](x):=sin(x)/x+x*f[n-1](x);
                                        sin(x)
(d1)                     f (x) := ------ + x f      (x)
                          n         x         n - 1



(c2) f[0](x):=1;
(d2)                               f (x) := 1
                                    0



(c3) f[1](x);
                                   sin(x)
(d3)                               ------ + x
                                     x



(c4) f[1](0);
Division by 0
Returned to Macsyma Toplevel.
```

We first try to fix this error by including a straightforward test for `x=0` and returning the value `1` in this case. All other values are computed by recursion. This method also fails.

```
(c5) f1[n](x):=(if x = 0 then 1 else sin(x)/x+x*f1[n-1](x));
                                                       sin(x)
(d5)                     f1 (x) := if x = 0 then 1 else ------ + x f1      (x)
                           n                              x          n - 1



(c6) f1[0](x):=1;
(d6)                     f1 (x) := 1
                           0



(c7) f1[1](x);
                          sin(x)
(d7)                      ------ + x
                            x
```

```
(c8) f1[1](0);
Division by 0
Returned to Macsyma Toplevel.
(c9) f1[1];
                                     sin(x)
(d9)                     lambda([x], ------ + x)
                                       x
```

Inspection of the **lambda** form bound to the referenced slot shows that it is not what was intended. The problem is that the **if** was evaluated *before* the **lambda** form was generated, when x was still unbound. The *result* of the **if** statement was used to generate the body of the function definition.

The next attempt tries to get around this problem by deferring evaluation of the **if** statement until the function is invoked.

```
(c10) f2[n](x):='(if x = 0 then 1 else sin(x)/x+x*f2[n-1](x));
(d10)           f2 (x) := '(if x = 0 then 1
                  n
                                                  sin(x)
                                             else ------ + x f2     (x))
                                                    x          n - 1



(c11) f2[0](x):=1;
(d11)                             f2 (x) := 1
                                    0



(c12) f2[1](0);
(d12)                              1



(c13) f2[1];
(d13) lambda([x], if x = 0 then 1
                                                  sin(x)
                                             else ------ + x f2     (x))
                                                    x          n - 1



(c14) f2[1](x);
Error: The control stack overflowed.
```

This attempt at deferred evaluation handles the special case `x=0` correctly, but no longer handles the general recursive case since n is not bound inside the body of the **lambda** form. The error occurs because Macsyma attempts to compute `f2[n-1]`, `f2[n-2]`,.... and eventually overflows an internal storage area. The correct procedure, as documented in the *Macsyma Reference Manual* , is to defer evaluation, but to force an evaluation of the index via a **subst** command.

```
(c15) f3[n](x):=subst('nn=n,'(if x = 0 then 1 else sin(x)/x +
                                                   x*f3[nn-1](x)));
(d15)           f3 (x) := subst('nn = n,
                  n
                                               sin(x)
                      '(if x = 0 then 1 else ------ + x f3      (x)))
                                                 x          nn - 1


(c16) f3[0](x):=1;
(d16)                            f3 (x) := 1
                                   0


(c17) f3[1](x);
                                  sin(x)
(d17)                             ------ + x
                                    x


(c18) f3[1](0);
(d18)                                 1


(c19) f3[1];
                                                 sin(x)
(d19)           lambda([x], if x = 0 then 1 else ------ + x f3 (x))
                                                   x          0


(c20) f3[2];
                                                 sin(x)
(d20)           lambda([x], if x = 0 then 1 else ------ + x f3 (x))
                                                   x          1
```

This implementation handles the special case `x=0` and also the general case, but doesn't handle recursion efficiently since explicit calls to previously computed results are incorporated in the body of the **lambda** form rather than in the results. Subsequent calls to, say, `f3[3]` require `f3[2]`, `f3[1]`, and `f3[0]` to be recomputed. To implement an efficient recursion scheme, use two steps: First use **subst** and deferred evaluation to carry out the test for `x=0`; second, step call another subscripted function to carry out the recursive computations when you know that the special case has already been handled. This is done in the following example:

```
(c21) g[n](x):=subst('nn=n,'(if x=0 then 1 else %g[nn](x)));
(d21) g (x) := subst('nn = n,
       n
                           '(if x = 0 then 1 else %g  (x)))
                                                      nn
```

```
(c22) g[0](x):=1;
(d22)                              g (x) := 1
                                    0



(c23) %g[n](x):=sin(x)/x+x*%g[n-1](x);
                                   sin(x)
(d23)                   %g (x) := ------ + x %g      (x)
                          n          x          n - 1



(c24) %g[0](x):=1;
(d24)                             %g (x) := 1
                                    0
```

You call the function **g**, but **%g** actually implements the recursion. We test a few cases.

```
(c25) g[3](x);
                             sin(x)        sin(x)    sin(x)
(d25)                  x (x (------ + x) + ------) + ------
                               x             x         x



(c26) g[3];
(d26)                lambda([x], if x = 0 then 1 else %g (x))
                                                       3



(c27) %g[3];
                                   sin(x)        sin(x)    sin(x)
(d27)            lambda([x], x (x (------ + x) + ------) + ------)
                                     x             x         x
```

Here we test the functional arguments efficiently and alo implement the recursive definition. If you want a simplified result, include **expand** with your definition of **%g**.

This function handles the required tests on the functional argument and also implements the recursive definition efficiently. Should a simplified form be desired, the definition of %g could include a call to **expand**.

12.4 The Debugger

Complicated functions rarely work as expected the first time. The reason for the failure is not always obvious; it might be due to incorrect arguments, or to a bug within a function, or even to a bug inside a built-in Macsyma function. The traditional method of debugging a function is to insert **print** statements and see how far the program gets before something goes wrong. Macsyma provides a sophisticated debugger which lets you investigate the problems of functions without resorting to clumsy "trial-and-error" procedures. Although learning how to use the debugger requires some time and effort, it is well worth it for anyone who programs in Macsyma. This section will discuss only a few of the debugger utilities. For a full description, see the *Macsyma Reference Manual* .

The debugger provides a simplified Macsyma environment (a *break*) which can be entered automatically, upon detection of an error, or manually, by executing a **break** command. The break lets you investigate the cause of an error *in the environment in which the error occurs*. The debugger also lets you monitor the values passed to and returned by functions via the **trace** command. In many cases it is possible to determine the cause of the error without ever inserting debugging statements into the source code.

12.4.1 The Trace Utility

The **trace** special form lets you monitor the execution procedures. When you use **trace**, the name of the function, its argument list, and the depth of recursion are displayed. The value returned by the function is displayed when the function is exited.

The syntax of the **trace** command is **trace**(*function1, ..., functionN*) . Typing **trace**() or inspecting the system variable **trace** returns a list of functions currently traced. To untrace a function, type **untrace(***function1, ..., functionN***)** to untrace specific functions, or **untrace()** to untrace all traced functions.

Note: Functions that do not evaluate all of their arguments and functions that are implemented as simplifier functions (as opposed to evaluator functions) cannot be traced. See the *Macsyma Reference Manual* for more information on the former.

12.4.2 Tracing Simple Functions

In this example, we define functions **fun1** and **fun2** and trace them. **trace** displays the depth of recursion, the word ``Enter'', the name of the function, and the argument list when you enter each traced function. When the function is exited, the depth of recursion, the word ``Exit'', the name of the function, and the value returned by the function are displayed. If space permits, the subordinate functions are indented.

```
(c1) fun1(x):=x/fun2(x+1);

                                      x
(d1)                fun1(x) := -----------
                                fun2(x + 1)



(c2) fun2(x):=1+x;

(d2)                   fun2(x) := 1 + x



(c3) trace(fun1,fun2);

(d3)                     [fun1, fun2]
```

```
(c4) fun1(y);
1 enter fun1 [y]
 1 enter fun2 [y + 1]
 1 exit  fun2 y + 2
               y
1 exit  fun1 -----
             y + 2
                              y
(d4)                        -----
                            y + 2



(c5) untrace();
(d5)                    [fun2, fun1]
```

12.4.3 Tracing a Recursive Function

This example traces a recursive function which computes the factorial of a non-negative integer. Note that the depth of recursion is indicated each time the function is executed.

```
(c1) fact(x):=if x = 0 then 1 else x*fact(x-1);
(d1)   fact(x) := if x = 0 then 1 else x fact(x - 1)



(c2) trace(fact);
(d2)                       [fact]



(c3) fact(4);
1 enter fact [4]
 2 enter fact [3]
  3 enter fact [2]
   4 enter fact [1]
    5 enter fact [0]
    5 exit  fact 1
   4 exit  fact 1
  3 exit  fact 2
 2 exit  fact 6
1 exit  fact 24
(d3)                         24
```

```
(c4) untrace();
(d4)                       [fact]
```

12.4.4 Using the break Facility

As shown in the previous section, the **trace** facility lets you monitor the executions of functions. However, it tells nothing about the *environment* in which the function is executed. The **break** facility lets you suspend execution at any desired point and inspect, or even modify, the environment, and then resume program execution. (Note: you cannot continue execution from an error break.) You can enter an error break automatically after detecting an error condition so the user can investigate the cause of the error. This behavior is determined by the setting of the option variable **debugmode**.

Regular and error **break**s differ from Macsyma toplevel in that no input or output labels are generated automatically, and the system variables `%` and `%th(n)` are not rebound. However, the system variable `%%` is bound to the last result generated in the break. You can evaluate expressions and manually bind any variables they wishes. The following examples illustrate some of these points.

You can enter the break environment by using the command `break();` . Once in the environment, you can check or change the values of variables and then either continue the computation or abort it.

In PC Macsyma 2.0, you may be prompted to type :1 (Continue), :2 (Return) Top-Level Macsyma, etc. The response would be :1 or :2. In other versions, you may be prompted to type CONTINUE; to continue or ABORT; to abort the calculation.

```
(c1) (a:10, break(), b:20);
Macsyma Break level 1
(Type CONTINUE; to continue the computation; type ABORT; to abort it.)
_   a
      10
_   b
      b
_   abort
(c2) a;
(d2)                                10
(c3) b;
(d3)                                 b
(c4 )ev(c1);
Macsyma Break level 1
(Type CONTINUE; to continue the computation; type ABORT; to abort it.)
_   continue
Exited from Macsyma Break level 1
(d4)                                20
(c5) a;
(d5)                                10
(c6) b;
(d6)                                20
```

In the following example, the system option variables **radexpand** and **algebraic** are bound at toplevel. **radexpand** is used as a local variable in a block to demonstrate how you can inspect or modify the environment inside a break.

```
(c1) (algebraic:true,radexpand:true);
(d1)                         true
```

The following string demonstrates how % is not bound to results generated within a break. When you enter the break is entered from (c3), % still evaluates to this result.

```
(c2) "This is the previous d-line.";
(d2)            This is the previous d-line.
```

The following **block** statement locally binds **radexpand** to **false**, then enters a Macsyma break. The variable **algebraic** is rebound manually within the break. When you exit the break and continue execution, the current setting of **radexpand** (within the block) is displayed.

```
(c3) block([radexpand:false],break(),print("radexpand value:", radexpand));
Macsyma Break level 1
(Type CONTINUE; to continue the computation; type ABORT; to abort it.)
_ radexpand;
FALSE
```

Here, the Macsyma break has been entered. The default break prompt is an underscore character. The value of **radexpand** was changed when the block was entered and the local binding took effect.

The following commands show that the environment can be modified as desired (in this case, we change the settings of **radexpand** and **algebraic**). Note also the rebinding of %% but not %.

```
_ radexpand:all;
all
_ %%;
all
_ %;
This is the previous D-line.
_ algebraic;
true
_ algebraic:false;
false
_ exit;
Exited from Macsyma Break level 1
radexpand value: all
(d3)                         all
```

Finally, confirm that the change made to the block variable was indeed local, while the change to **algebraic**, which was not a block variable, persists:

```
(c4) radexpand;
(d4)                         true
(c5) algebraic;
(d5)                         false
```

Note that the **exit** command is used to return from the break and continue program execution. The **abort** command can be used to exit the break and return to toplevel.

Program execution cannot be continued from an error break. Either **exit** or **abort** will exit the error break and return to toplevel.

12.4.5 Entering the Debugger Automatically

Use of the error break is governed by the setting of the option variable **debugmode** (*default*:**false**), which can be set to **false**, **true**, **all**, or **lisp**. The results of the various settings when an error condition is detected are summarized below:

- **false**: (the default) execution aborts and the user returns to Macsyma toplevel.
- **true**: enters a Macsyma error break.
- **all**: enters a Macsyma error break and sets the system variable **backtrace** to a current list of the user-defined functions.
- **lisp**: a LISP error break is entered.

The **lisp** setting of **debugmode** is of limited use unless you are familiar with LISP and the capabilities of the LISP debugger depend on the version of LISP which comes with your Macsyma.

Consider the following pair of functions which generates an arithmetic error when called with `x=0`.

```
(c1) fun1(x):=fun2(x);
(d1)                 fun1(x) := fun2(x)
(c2) fun2(x):=1/x;
                                1
(d2)                   fun2(x) := -
                                x
(c3) debugmode;
(d3)                      false
```

When **debugmode** is **false**, then executing the function causes an error message to be displayed, after which control returns to Macsyma toplevel.

```
(c4) fun1(0);
Division by 0
```

debugmode set to **true** enters an error break. The **backtrace** variable is not set.

```
(c5) debugmode:true;
(d5)                       true
(c6) fun1(0);
Division by 0
Macsyma Error Break level 1
(Type CONTINUE; to continue the computation; type ABORT to abort it.)
_exit;
Exited from Macsyma Error Break level 1
```

With **debugmode** set to **all**, an error break is entered and the **backtrace** variable contains a list of the functions currently entered, in reverse order. (The arguments of the functions are not evaluated.) The settings of **debugmode** to **true** and **all** are identical in all other respects.

```
(c7) debugmode:all;
(d7)                        all
(c8) fun1(0);
Macsyma Error Break level 1
(Type CONTINUE; to continue the computation; type ABORT to abort it.)
_backtrace;
[fun2(x), fun1(0)]
_exit;
Exited from Macsyma Error Break level 1
```

12.5 Handling Errors

12.5.1 General Error Handling

Sometimes a programmer anticipates an operation might fail. Rather than write code to execute that operation only if it is guaranteed to succeed, the programmer would prefer to just try it and test afterwards to see if it succeeded. A problem arises when the operation fails with an error condition since this will halt program execution. In this case, it is necessary to somehow "trap" the error so it doesn't interrupt program execution. The **errcatch** facility provides this functionality in Macsyma.

The syntax for the **errcatch** function is **errcatch(***expression***)**. If *expression* executes without error then a list containing the result of *expression* is returned. If an error occurs then an empty list is returned. The error message is displayed if the option variable **errormsg** [*default*:**true**] is **true**. The last error message is saved in the system variable **error_string**, and can be used to determine the type of error.

Note: if the error message includes a large Macsyma expression, it may not display if it is too large. (The maximum size of an expression which will be displayed is determined by the option variable **error_size**.) Instead, the error message will use one or more of the symbols **errexp1**, **errexp2**, and **errexp3**, which are bound to the offending expression(s).

The following examples show how the **errcatch** special form works.

12.5.2 Example: Catching errors

```
(c1) debugmode:false;
(d1)                       false
(c2) 1/0;
Division by 0
Returned to Macsyma Toplevel.
(c3) errcatch(1/0);
Division by 0
(d3)                         []
(c4) debugmode:true;
```

```
(d4)                           true
(c5) 1/0;
Division by 0
Macsyma Error Break level 1
(Type CONTINUE; to continue the computation; type ABORT; to abort it.)
_ exit;
Exited from Macsyma Error Break level 1
Returned to Macsyma Toplevel.
(c6) errcatch(1/0);
Division by 0
(d6)                            []
```

12.5.3 Example: Using errcatch

The next example uses the **errcatch** facility, along with the **error_string** and **errormsg** system variables, to trap errors occurring when zero is substituted into an expression. **errcatch** looks for two specific error conditions, namely, `Division by zero` and `log(0)`. If it determines that either of these errors has occurred it returns a symbol identifying the error. If some other type of error occurs, a symbol indicating an error of unknown type has occurred appears. Examples which generate these two errors, as well as an illegal substitution into a **diff** form, are shown below.

```
(c1) debugmode:false$
(c2) subst(0,x,1/x);
Division by 0
Returned to Macsyma Toplevel.
(c3) subst(0,x,log(x));
LOG(0) has been generated.
Returned to Macsyma Toplevel.
(c4) subst(0,x,'diff(f,x));
Attempt to differentiate with respect to a number:
0
Returned to Macsyma Toplevel.
```

The function **zero_eval**, defined below, performs the indicated substitution. The **subst** form is protected by an **errcatch** to prevent program interruption. The option variable **errormsg** is initialized to **false** *as a block variable*. Be careful when when setting **errormsg** to **false** to avoid overlooking valuable debugging information.

```
(c5) zero_eval(expr,var):=block
([errormsg:false,result],
 result:errcatch(subst(0,var,expr)),
 if result = []
  then (if error_string = "Division by 0"
         then 'divide_by_zero_error
         else if error_string = "LOG(0) has been generated."
               then 'log0_error
```

```
               else 'unknown_error)
  else first(result))$
(c6) zero_eval(f(x),x);
(d6)                        f(0)
(c7) zero_eval(1/x,x);
(d7)                   divide_by_zero_error
(c8) zero_eval(log(x),x);
(d8)                        log0_error
(c9) zero_eval('diff(f,x),x);
(d9)                   unknown_error
```

12.5.4 Catching Special Classes of Errors

The **errcatch** facility lets you trap all errors and you can trap mathematical errors selectively. (Mathematical errors, in this context, are errors arising from arithmetic operations or from elementary transcendental functions.) If a mathematical error occurs, then a string containing the error message is returned as the result.

This facility is enabled if the option variable **catch_mathematical_error** (*default*:**false**) is set to **true**. The operation in question must be protected with a **catch** form. The following example shows how this facility is used.

12.5.5 Example: Selectively Trapping Mathematical Errors

```
(c1) catch_mathematical_error:true$
(c2) debugmode:false$
(c3) 1/0;
Division by 0
Returned to Macsyma Toplevel.
(c4) errcatch(log(0));
LOG(0) has been generated.
(d4)                         []
(c5) catch(1/0);
(d5)                   Division by 0
(c6) log(0);
LOG(0) has been generated.
Returned to Macsyma Toplevel.
(c7) errcatch(log(0));
LOG(0) has been generated.
(d7)                         []
(c8) catch(log(0));
(d8)              LOG(0) has been generated.
```

Certain errors resulting from **taylor** and **integrate** can be caught when the associated option variables are set. These option variables are **catch_divergent**, **catch_taylor_essential_singularity**, and **catch_taylor_unfamiliar_singularity**. See the *Macsyma Reference Manual* for further information.

12.6 Macros

In advanced Macsyma programming applications, you sometimes must delay or inhibit evaluation of function arguments. The *Macsyma Reference Manual* draws a distinction between a *function* (a procedure which evaluates all of its arguments in order from left to right) and a *special form* (a procedure which can delay or omit evaluation of one or more of its arguments). The **:=** operator and the **define** command are used to define functions. The **::=** operator defines a *macro*, a utility that generates Macsyma code. Since the macro facility gives you complete control over evaluation of a macro's arguments, you can use it to implement special forms. You can use a macro when a segment of code is used repeatedly but cannot be conveniently defined as a function. Implementing this segment of code as a macro improves modularity and reduces the amount of typing. This Guide will provide only an introduction to macros. For more information, consult the *Macsyma Reference Manual* .

Loosely speaking, a macro is a piece of code that generates code. A macro looks like a function but expands into code at either translation or runtime. Recursion is allowed within a macro.

The following commands are used in writing macros:

- **buildq** delimits the body of the macro. Code substitution takes place within the body of the **buildq**. Its syntax is similar to that of **block**.
- **splice** constructs the argument list of a function. **splice** is a keyword form for **buildq** that takes its argument (which must evaluate to a list), and returns a form whose arguments are the list entries when used as an argument to a form in the body of a **buildq**,
- **macroexpand** is a useful debugging tool. It expands a form repeatedly until it is no longer a macro call.

A few examples follow.

12.6.1 Writing Simple Macros

In the next example we write a macro which takes a single argument and returns an equation consisting of the unevaluated form on the left-hand side and the evaluated form on the right-hand side.

```
(c1) showme(arg)::=buildq([arg],'arg=arg);
(d1)     showme(arg) ::= buildq([arg], 'arg = arg)
(c2) macroexpand(showme(argument));
(d2)                 'argument = argument
(c3) argument:1;
(d3)                          1
(c4) showme(argument);
(d4)                     argument = 1
(c5) argument:x+1;
(d5)                        x + 1
(c6) showme(argument);
(d6)                   argument = x + 1
```

In `(c1)`, **buildq** is used to write the macro. In `(c2)`, **macroexpand** is used to expand a macro call to **showme** to verify that the macro expands as expected. **showme** is tested in `(c4)` and `(c6)`. Note that this cannot be written as a regular Macsyma function because in that case the argument would be evaluated before it is passed to the function.

The **splice** keyword form is useful when writing recursive macros or macros which take an arbitrary number of arguments. The following commands show how to construct a call to a form **fun**. Note the difference in the argument lists of **fun** in lines `(d8)` and `(d9)`.

```
(c7) listvar:'[a,b,c];
(d7)                    [a, b, c]
(c8) buildq([listvar],fun(listvar));
(d8)                  fun([a, b, c])
(c9) buildq([listvar],fun(splice(listvar)));
(d9)                   fun(a, b, c)
```

12.6.2 Writing a Macro to Implement a Boolean Operator

This example implements an **or** operator, which we call **my_or** to avoid confusion with the built-in **or** operator. The function **my_or** takes a list of expressions, evaluates them sequentially, and returns the first non-**false** result. (This is similar to the Macsyma construct **is(***pred1* **or** *pred2* **or . . . or** *predn*) except that **my_or** can return a non-boolean value.)

The function **my_or** cannot be implemented as a Macsyma function because all of the arguments to a function are evaluated *before* the function gets hold of them. Consider the expression `(n:'n, my_or( symbolp(n), n>0))` that we want to return the value **true**. If **my_or** were a function then its arguments would be evaluated prior to the function invocation. The first would yield **true**, while the second would generate the error message `''Macsyma is unable to evaluate the predicate N > 0''`. Consequently, control would never be passed to **my_or**. What is needed is a way to evaluate only the first argument: if it is non-**false** then return that value; otherwise, evaluate the subsequent arguments sequentially. The macro facility provides the needed control over evaluation.

The macro definition of **my_or** is given below. The syntax of the command is **my_or**(*pred1, pred2, . . . , predn*).

```
my_or([args])::=
 if args = []
  then false
  else buildq
       ([tmp:first(args),args:rest(args)],
        if tmp # false
         then tmp
         else my_or(splice(args)))$
```

Note that the macro takes an arbitrary number of arguments and is recursive. When invoked, the macro expansion continues until a non-**false** argument is found or else the argument list is empty. The variable *tmp*, which is set to the first element of the argument list, is evaluated in the body of the **buildq**. The remainder of the argument list is not evaluated.

Note also that the **splice** keyword form is used to provide the required syntax for the recursive call. **splice** is used to generate the call to **my_or** with the remaining arguments.

The following example tests **my_or**. In this example **my_or** evaluates each argument in turn and finds that the third one evaluates to a non-**false** value. Consequently, it returns this value.

```
(c1) my_or([args])::=
 if args = []
  then false
  else buildq
       ([tmp:first(args),args:rest(args)],
        if tmp # false
         then tmp
         else my_or(splice(args)))$
(c2) (arg1:false,arg2:false,arg3:'howdy,arg4:'doody);
(d2)                       doody
(c3) my_or(arg1,arg2,arg3,arg4);
(d3)                       howdy
```

We can use the **macroexpand** facility to expand the macro. (This is usually the only way to debug a macro, since it cannot be traced.) Note that **macroexpand** only expands the toplevel call. To fully expand a recursive macro, it is necessary to **scanmap** the **macroexpand** call:

```
(c4) macroexpand(my_or(arg1,arg2,arg3));
(d4) if arg1 # false then arg1 else my_or(arg2, arg3)
(c5) scanmap('macroexpand,'(my_or(arg1,arg2,arg3)));
(d5) if arg1 # false then arg1
 else (if arg2 # false then arg2
 else (if arg3 # false then arg3 else false))
```

The question of how you actually write a macro has not yet been addressed. (In case you're wondering, it won't be addressed in great detail here, either.) The procedure essentially involves writing down the fully expanded form for a suitably general case and working backwards. For example, writing **my_or** consisted of the following steps:

- Write down the full expansion of **my_or** when called with three arguments. This means writing down a nested **if then else** structure. For example, `my_or(arg1,arg2,arg3)` should expand into the structure

```
if arg1 # false
   then arg1
   else (if arg2 # false
            then arg2
            else (if arg3 # false
                     then arg3
                     else false))
```

- From the structure of the problem, it is clear that a recursive approach is the most efficient one. The recursive approach is particularly useful since the macro must take an arbitrary number of arguments. The result of writing the previous expansion in terms of a recursive call to **my_or** is

```
if arg1 # false
   then arg1
   else my_or(arg2,arg3)
```

- Since the macro will be recursive, the next step is to determine the grounding condition (here, the condition is that the argument list is empty), and what action to take when the condition is satisfied. This is implemented in clause

```
if args = []
   then false
   else buildq...
```

- Finally, implement the substitution and the recursive call with a **buildq** form. The required form is

```
buildq([tmp:first(args),args:rest(args)],
      if tmp # false
         then tmp
         else my_or(splice(args)))$
```

  The local variables *tmp* and *args* are set in the **buildq** variable list. Neither is evaluated at this time. The first argument to **my_or**, *tmp*, is evaluated in the body of the **if** statement. If the **else** clause of the **if** statement is executed **my_or** is called recursively on the remaining arguments. The **splice** keyword form is needed to generate an argument list from the list of arguments contained in the variable *args*. Note that since *args* is used by **splice** it must appear in the **buildq** variable list. Note also that redefining *args* to be **rest**(*args*) *must* be done in the variable list if it were done in the body of the **buildq** then it would be evaluated, and if it were done inside the **splice** form then **splice** would not be recognized as a keyword form to **buildq**.

- Having written the macro, the next step is to test it. Since it is impossible to write a macro which works without modification, you can debug it with the **macroexpand** and **macroexpand1** facilities, along with judicious use of **scanmap** when necessary.

12.7 Localizing Information

Much of the power of Macsyma as an interactive or programming tool derives from the richness of the Macsyma environment. For example, the actions of some system functions (*e.g.*, **ratsimp**) are controlled by the settings of system option variables (*e.g.*, **algebriac**). Calling **ratsimp** on an expression can give different (but mathematically equivalent) results, depending on the setting of the option variable **algebraic**.

Unfortunately, this feature occasionally causes some difficulties for users. It is extremely frustrating to find that a program which previously worked suddenly fails to work because the global environment was unintentionally altered by an earlier operation. Good programming techniques can insulate a function from difficulties of this nature by guaranteeing that the function executes in the correct environment. Good programming techniques can also guarantee that the function runs "cleanly", that is, even though the function might require a special environment, the global environment after execution is essentially identical to the global environment prior to execution.

Macsyma provides two separate methods for localizing information. The **block** mechanism is used to localize properties and values via the block variable and **local** mechanisms. The **context** mechanism allows the user to group database information together (this grouping is referred to as a *context*) so that a given context can be made accessible or inaccessible to Macsyma.

12.7.1 Program Blocks Revisited

The **block** command was discussed briefly in Section 11.4. We discuss it in greater detail in this section.

As described in Section 11.4, a block variable is declared by naming it in the block variable list. An initialization value can be specified in the list if desired. If a block variable has the same name as a variable defined outside of the **block** then the value of the variable outside of the **block** is saved when the **block** is entered and is restored when the **block** is exited.

Note: making a variable a block variable protects its *value*, but no other properties. It does not, for example, protect the array property associated with a symbol. The following example illustrates this point.

```
(c1) array(a,3);
(d1)                        a
(c2) a[1];
(d2)                        a
                             1
(c3) variable:1;
(d3)                        1
(c4) block([a,variable],a[1]:'array,variable:'symbol);
(d4)                      symbol
(c5) a[1];
(d5)                      array
(c6) variable;
(d6)                        1
```

Note that the value of the symbol *variable* is unchanged, but the global array **a** has been modified.

12.7.2 Localizing Other Information Inside Program Blocks

As shown in the example in the previous section, the block variable mechanism only protects the *value* associated with a symbol. All other localizable properties must be localized via the **local** mechanism. (Note: not all properties can be localized. For example, a *complete array* is an example of a global object which cannot be localized.) In the following example, **local** is used to localize array and dependency information.

```
(c1) array(a,3);
(d1)                        a
(c2) a[1];
(d2)                        a
                             1
(c3) dependencies;
(d3)                        []
(c4) block
     ([x],
      local(a,y),
      a[1]:'array,depends(y,x),
      break());
Macsyma Break level 1
(Type CONTINUE; to continue the computation; type ABORT; to abort it.)
```

```
_a[1];
ARRAY
_dependencies;
[y(x)]
_exit;
Exited from Macsyma Break level 1
(d4)                       false
(c5) a[1];
(d5)                         a
                              1
(c6) dependencies;
(d6)                         []
```

In (c1) through (c3) a global array is created and the state of the Macsyma environment is displayed. (Note that the array **a**, generated in (c1), is a *declared* array rather than a *complete* array, and can therefore be localized.) The command (c4) executes a **block** form in which a and y are localized. An array slot in variablea is set, and dependency information is provided for y. A **break** is then entered and we verify that the expected properties are present inside the **block**. After the **break** and **block** are exited, we verify in (c5) and (c6) that the changes made inside the **block** were indeed local.

12.7.3 Program Contexts

The remaining localization mechanism available in Macsyma is the **context** mechanism. This mechanism allows the user to selectively activate and deactivate database information. We do not discuss this topic in this Guide except to note that the **context** mechanism is independent of the **block** mechanism. See the *Macsyma Reference Manual* for a description of **context**.

12.8 Translating Macsyma Functions

Like other interpreted languages, Macsyma supports a translator. The production of machine code from Macsyma source code is a two-step process. First, the Macsyma code is translated into LISP code; next the LISP code is compiled into machine code. The Macsyma translator produces LISP code which, in most cases, loads and executes faster than the equivalent Macsyma source code. Numerical routines often achieve a factor of three improvement in performance. Also, it is easy to make translated code "cleaner" code because of the translator warnings. The translator flags global variables that are not explicitly declared to be global (via the **special** declaration), brings them to the attention of the programmer and thus allows them to be localized, either by making them **block** variables, or via the **local** mechanism.

The primary reason that translation improves performance is that it allows assumptions to be made concerning the argument types for specified operations. In particular, it causes certain operations (such as data type checking) to be done only once, namely, at translate time. For example, the interpreted function `f(x,y):=x*y` will accept either symbolic or numeric arguments. When the function is executed, the argument types must be determined before the indicated multiplication can be carried out. If it is known that the function will be called exclusively with floating-point arguments then the user can bypass this rather expensive checking step by declaring the data types of the arguments and then translating the function. When the declarations are in place, the translator will specify that the multiplication be carried out by a routine which handles floating-point numbers. Of course, if the resulting translated function is called with arguments of an inappropriate type then a run time error will occur.

Another benefit of translation (of files) is that a translated file loads much faster than a Macsyma source file since the translated file does not need to be parsed. This is also true for interpreted Macsyma functions which are subsequently saved.

12.8.1 Translating Files and Functions

The Macsyma translator permits the user to translate interpreted functions internally via the **translate** command, or to translate Macsyma source files and save the results in a file via the **translate_file** command. A third option, in which interpreted functions are translated and the results are saved in a specified file, is available via the **compfile** command. We mention the last one for completeness, since it is used infrequently in practice. (A user is much more likely to use a text editor to store a Macsyma function in a file than to enter the function interactively and save the translated result via **compfile**).

The following examples show how to use **translate** and **translate_file**.

12.8.1.1 Example: Translating a Function Definition

The following Macsyma commands define an interpreted function and then translate it (to core):

```
(c1) f(x):=x^2;
                                     2
(d1)                     f(x) := x
(c2) translate(f);
(d2)                        [f]
(c3) properties(f);
(d3)                [function, transfun]
```

Note that **f** has both **function** and **transfun** properties, corresponding to both the interpreted and translated versions. The translated version of the function is executed by default.

12.8.1.2 Example: Translating a Macsyma File

In the following example, a file named `` `f.macsyma' ``, which defines a function identical to the one in the previous example, exists in the directory `macsyma:user;f.macsyma`. The following Macsyma commands translate the file, place the result in a file named `` `f.lisp' ``, load the translated file into the Macsyma session, and look at the resulting properties of the function. (Note: the pathnames used in this example are the Macsyma logical pathnames.)

```
(c1) printfile("macsyma:user;f.macsyma");
f(x):=x^2;
(d1)                        C:\MACSYMA2\user\f.mac
(c2) translate_file("macsyma:user;f.macsyma")
Translating the file C:\MACSYMA2\user\f.mac
Translation done.
(d2)                        C:\MACSYMA2\user\f.lsp
(c3) load("f.lisp");
C:\MACSYMA2\user\f.lsp being loaded.
(d3)                  C:\MACSYMA2\user\f.lsp
```

```
(c4) properties(f);
(d4)                [function, transfun]
```

Some operating systems, use the extensions **mac** and **lsp**, while others use **macsyma** and **lisp**. Refer to pathname extensions in the *Macsyma Reference Manual* for information for your particular version of Macsyma.

12.8.2 Data Type Declarations

As mentioned in the previous section, the greatest performance advantage of translation is due to the fact that the data type declaration permits the translator to determine which machine-level operation to call on the data, bypassing the expensive checking step. In this section we introduce the Macsyma forms that are used to declare data types. The forms that we will discuss are

- **mode_declare**, which informs the translator of the data type of a variable;
- **mode_identity**, which can be used to inform the translator of the data type of a more complicated Macsyma structure;
- the **special** property, which is used in declaring a variable global (or one that will have an unusual side-effect).

12.8.2.1 Declaring Data Types With mode_declare

As mentioned earlier, **mode_declare** is used to specify the data type of a variable or function so that the translator can select the most efficient routine to carry out the specified operation. The **mode_declare** operator also allows the translator to specify an initial value for a variable, although it is good programming practice to initialize the variables manually. (If you do not initialize the variables manually then your code might run when translated but not interpreted.)

The following example uses the **romberg** command to compute an iterated integral via numerical integration. The first approach calls **romberg** on an interpreted argument. The second sets up functions which compute the iterated integrals using **mode_declare** and nested calls to **romberg**. Using translated functions in this manner results in a speedup by roughly a factor of two in this case.

```
(c1) (showtime:all,g(x,y,z):=sin(x^2+y^2+z^2));
time= 0 msecs
                                  2    2    2
(d1)          g(x, y, z) := sin(x  + y  + z )
(c2) romberg(romberg(romberg(g(x,y,z),x,0.0,1.0),y,0.0,1.0),z,0.0,1.0);
time= 178983 msecs
(d2)                0.7316832579251946
(c3) f(x):=block
     (mode_declare([function(f),x,y,z],float),
      declare([y,z],special),
      sin(x^2+y^2+z^2))$
(c4) f1(y):=block
     (mode_declare([function(f1,f),y],float),
      declare(f,special),
      romberg(f,0.0,1.0))$
```

```
(c5) f2(z):=block
     (mode_declare([function(f2,f1),z],float),
      declare(f1,special),
      romberg(f1,0.0,1.0))$
(c6) translate(f2,f1,f);
time= 266 msecs
(d6)                    [f2, f1, f]
(c7) romberg(f2,0.0,1.0);
time= 51800 msecs
(d7)                 0.7316832579251946
```

It is necessary to define the functions **f**, **f1**, and **f2** to take single arguments because **romberg** requires that a translated or compiled function as its first argument take only one argument. Calling **romberg** on **f2** in `(c7)` binds the argument of **f2**, z, and evaluates the function **f2**. When **f2** is evaluated, it binds the argument of **f1**, y, and invokes **romberg** on **f1**. When **f1** is evaluated, it binds the argument of **f**, x, and evaluates **f**. At this point, x, y, and z evaluate to floating-point numbers, and **f** consequently returns a floating-point result.

It is also possible to use **mode_declare** to improve the handling of translated arrays. See the *Macsyma Reference Manual* for more information on this topic.

12.8.2.2 Defining Complex Data Types With mode_identity

The **mode_declare** form can be used to inform the translator of the data type of a variable. However, it is often the case that the variable is a rather complicated object, for example, a list of floating-point numbers. **mode_declare** cannot be used to declare the entries to be floating-point numbers because it can only declare "top-level" structures. What is needed is a way to declare the types of the elements of the list, which can be done via **mode_identity**. We will say little else about **mode_identity** in this Guide; for more information, consult the *Macsyma Reference Manual*.

Note: The **mode_identity** scheme is rather awkward when dealing with, say, lists of arbitrary length. For example, it cannot be used to assist in the translation of a structure such as **apply("+",list_of_numbers)**. Such operations are best implemented directly in LISP.

12.8.3 Defining Option Variables For Packages

When writing complicated programs, it is often desirable to have global option variables which control the program's behavior. These option variables can be defined using the **define_variable** form, which defines the name of the variable, its default binding, type checking, and an optional documentation string. As usual, we provide some examples and refer the reader to the *Macsyma Reference Manual* for more information.

The first example defines a boolean variable **bool** with initial value **true**.

```
(c1) bool;
(d1)                         bool
(c2) define_variable(bool,true,boolean);
(d2)                         true
(c3) bool;
(d3)                         true
(c4) bool:false;
```

```
(d4)                      false
(c5) bool:1;
Error: BOOL was declared mode BOOLEAN, has value: 1
Returned to Macsyma Toplevel.
(c6)
```

Note that **bool** can be set to either **true** or **false**, but attempting to set it to a non-boolean value results in an error.

The next example shows how the user can specify the acceptable values for an option variable. In this example, the option variable **opt** will be allowed to take on any odd positive integer value. Note that **opt** is initialized to 1.

```
(c1) define_variable(opt, 1, any_check, "This defines the variable OPT");
(d1)                        1
(c2) put('opt,lambda([arg],if not(integerp(arg)) or arg <=0 or evenp(arg)
                     then error("Illegal value for OPT: ",arg)),'value_check);
(d2) lambda([arg], if not integerp(arg) or arg <= 0
     or evenp(arg) then error(Illegal value for OPT: , arg))
(c3) opt;
(d3)                        1
(c4) opt:3;
(d4)                        3
(c5) opt:2;
Illegal value for OPT:  2
Returned to Macsyma Toplevel.
(c6) opt:'a;
Illegal value for OPT:  A
Returned to Macsyma Toplevel.
(c7)
```

Note that the value check is implemented via a **lambda** form placed on the property list of the option variable with a **value_check** indicator.

It should be noted that **define_variable** differs from a toplevel binding using **":"** in two important ways. The first difference is the type checking feature. The second is that **define_variable** will set the variable to its specified initial value only if the variable is unbound. This means that the user can set an option variable *before* loading in the package which defines it. The following example illustrates this last point.

```
(c1) bool:false;
(d1)                      false
(c2) define_variable(bool,true,boolean);
(d2)                      false
(c3) bool;
(d3)                      false
(c4) bool:1;
Error: BOOL was declared mode BOOLEAN, has value: 1
Returned to Macsyma Toplevel.
(c5)
```

12.8.3.1 The special Declaration

The Macsyma translator will complain when it finds a variable that is global with respect to the local block. If such a variable is indeed intended to be global then it should be declared **special**. The following example shows how **special** is used in this manner. In this example, the variable y is global with respect to the program block, and is explicitly declared to be **special** in the function definition of **g** but not **f**.

```
(c1) f(x):=block(x*y);
(d1)                f(x) := block(x y)
(c2) translate(f);
Warning-> y is an undefined global variable.
(d2)                       [f]
(c3) g(x):=block(declare(y,special),x*y);
(d3)      g(x) := block(declare(y, special), x y)
(c4) translate(g);
(d4)                       [g]
```

In the next example, we define a function that takes an expression and the name of an operator and uses a **opsubst lambda** construct to return a list of the arguments of the specified operator.

```
(c1) list_args(expr,op):=block([args:[]],
 opsubst(op=lambda([arg],if not(member(arg,args))
                          then push(arg,args),
                         funmake(op,[arg])),expr), args)$
(c2) translate(list_args);


This form:
LAMBDA([ARG],IF NOT MEMBER(ARG,ARGS) THEN PUSH(ARG,ARGS),
       FUNMAKE(OP,[ARG]))
has side effects on these variables:
[ARGS]
which cannot be supported in the translated code.
(at this time)
(d2)                   [list_args]
```

Translation of **list_args** fails because of the side-effects of the **lambda** form on the variable *args*. However, if *args* is declared **special** then the function translates.

```
(c3) list_args1(expr,op):=block([args:[]],
 declare(args,special),
 opsubst(op=lambda([arg],if not(member(arg,args))
                          then push(arg,args),
                         funmake(op,[arg])),expr), args)$
(c4) translate(list_args1);
(d4)                   [list_args1]
```

12.8.3.2 Customizing the Translation Environment

It is often the case that a program needs a special environment for translation. For example, if the program references a macro then the macro must be defined prior to translation. Macsyma provides a method whereby the user can execute Macsyma commands when a particular operation (batching, loading, or translating a file) is invoked. This is done via the **eval_when** form.

The **eval_when** form takes two or more arguments. The first is a keyword which specifies the condition under which the remaining forms are evaluated. The acceptable keywords are **batch**, **loadfile**, and **translate**, corresponding to batching in a Macsyma source file, **load**ing a LISP or binary file, and translating a file via **translate_file**.

For example, to automatically load the **basic** macro package prior to translation of a file, the toplevel form **eval_when(translate, load("basic"))$** could be put at the beginning of the source file. When the file is translated using the **translate_file** command, the **load** form will be evaluated, thereby defining the needed macros.

12.8.3.3 Compilation vs. Translation

Up to this point we have discussed the Macsyma translator, which takes Macsyma functions and produces LISP code, either in memory (via the **translate** command), or in a file (via the **translate_file** or **compile** commands). We now discuss (briefly) the advantages which can be gained from compilation of the translator output.

Compilation of a LISP function results in machine object code. Such code frequently runs faster than either interpreted LISP or interpreted Macsyma code. This is usually achieved at the expense of run-time debugging information, which means that run-time errors in compiled code will often result in rather mysterious LISP-level error messages.

Functions which are currently loaded into your environment can be compiled using the **compile** command, which essentially calls the compiler on the in-core result of the **translate** command. A LISP or Macsyma file can be compiled (or translated and compiled) via the **compile_file** command, which is called in the same way as the **translate_file** command. The resulting binary file usually occupies less disk space and loads faster than the translated file.

For more information on this topic, consult the *Macsyma Reference Manual* and the *Release Notes* for your version of Macsyma.

12.8.4 Additional Notes on Translation

This section mentions a few hints on the use of translation that might not be readily accessible elsewhere.

- As of 1994, the **local** command doesn't translate correctly. If you want to translate a function which contains **local**, you should do the following:
  - Use the LISP function **?mlocal** instead of **local**. (**?mlocal** takes the same arguments as **local**.)
  - Call **?munlocal** prior to all run-time exits in the block.

  For example, following function puts database information on the variable y inside a program block and removes the information automatically when the block is exited:

```
local_1(x):=block
([y],
 local(y),
 assume(y>0),
```

```
 if x = 1
  then true
  else false)$
```

The function should be written as follows to make it translate correctly:

```
local_2(x):=block
([y],
 ?mlocal(y),
 assume(y>0),
 if x = 1
  then (?munlocal(), true)
  else (?munlocal(), false))$
```

- A macro must be defined prior to use. (Otherwise, a macro is interpreted as a function call.) If the macro is defined in a file, an **eval_when** form can be used to load the file and thereby define the macro. Alternatively, the macro definition can be included in the source code prior to the time where it is first called.
- Special forms that have both the properties that some arguments are not evaluated, and also have optional arguments do not translate properly. If you want to write a special form at Macsyma level, use a macro instead.
- Subscripted functions don't translate efficiently. They should be run interpreted instead.

12.9 Hints for Efficient Programming

The typical Macsyma programmer eventually discovers that he or she uses certain procedures frequently enough to put them into a personal library. Good programming style can greatly increase the utility of such a library. The following hints can make this task easier:

- **Don't assume anything about the global environment.** If your function requires a special environment (*i.e.*, a special option setting or a property) then set it up locally. It is important to make the relevant option variables initialized block variables to guarantee that they have the correct settings, even if the desired settings are the default values, because the option variables may be changed in the global environment.
- **Localize variables and properties whenever possible.** It can be frustrating to find that a function pollutes the global environment by leaving variable bindings or properties around after it is called. It is especially frustrating when a function changes the setting of a system option variable. Use the **block** and **local** mechanisms to avoid this.
- **Never use go.** Although this construction is supported in Macsyma, it is almost never necessary to use it. Programs which use **go** are hard to read.
- **If your code contains floating point variables and floating point arithmatic, it will run much faster if the variables are declared float with mode_declare and the code is compiled.**
- **Translate functions when possible.** Translated code often run faster than interpreted code. The translation process also warns the user of the presence of possibly unintended global variables.

- **Don't use ev in programs.** To force an extra evaluation, use the **eval** or **apply** functions. To force resimplification, use the **resimplify** function. To force evaluation of noun forms, use **apply_nouns** or an **opsubst lambda** construct to force evaluation of specified noun forms. To substitute values for variables, use **subst**.

- **Avoid using "::".** It is almost always the case that an apparent need to use **"::"** can be eliminated by restructuring the program.

- **Use the part functions instead of subscript notation to reference list elements.** The syntax **part**(*object*,*n*) is preferred over *object[n]* because if a global array named *object* exists then the latter notation will refer to the array in what might appear to be a haphazard manner.

Chapter 13

Displaying Expressions

One of the complaints lodged against computer algebra systems is that it is often difficult to express results in a "natural" way. Even experienced Macsyma users complain that it can be as difficult to simplify the results as it is to derive them. This section discusses some techniques for simplifying Macsyma expressions. These techniques give the user considerable (but not complete) control over the display of expressions.

The techniques used to simplify expressions fall into two categories. The first category consists of commands that modify the structure of the expression, via selective simplification, substitution, or a change of internal representations. Examples of facilities that change the internal representation are the **rat** and **ordergreat** **orderless** commands and the **mainvar** declaration. Commands used for selective simplification include **substpart**, **map**, and **multthru**. The second category consists of commands that affect the display of an expression, in particular the ordering of terms. Facilities which fall into this category include the **trunc** command and display flags such as **exptdispflag** and **powerdisp**.

13.1 The Macsyma Display Package

The Macsyma simplifier uses a default ordering scheme to put expressions into a "standard form." (This standard representation makes it easier for the simplifier to recognize identical quantities for, say, purposes of cancellation.) The ordering of atoms in a simplified expression is:

$$\text{mainvars} > \text{other variables} > \text{scalars} > \text{constants} > \text{numbers}$$

with the ordering inside a category taken to be reverse-alphabetic. The display package generates the two-dimensional display of an expression after simplification. A product displays with factors in increasing order, while a sum displays with terms in decreasing order of importance.

Note: the characters "_" and "%" are alphabetic characters. In the Macsyma alphabet, "%" comes before "**a**" and "_" follows "**z**".

Macsyma defaults to two dimensional formatted display of mathemetics using drawn mathematical symbols and Greek letters. You can turn off the fancy display and get two dimensional character display by setting **fancy_display**:**false** (default: **true**).

13.2 Changing the Default Display

The easiest way to change the default display is to change a display option variable. These variables are documented in the chapter entitled **Displaying and Ordering Functions** in the *Macsyma Reference*

Manual. Changing the value of a display option variable doesn't affect the internal representation of an expression and can be done (or undone) at any time. However, a display option variable should be set at toplevel (rather than via a local binding) to guarantee that it is still set during the display phase. The following example illustrates the effect of the option variable **powerdisp** (*default:* **false**), which when set to **true** causes polynomials to display with terms of increasing order.

```
(c1) expr:1+x+x^2;
                                  2
(d1)                             x  + x + 1
(c2) powerdisp:true;
(d2)                                true
(c3) expr;
                                         2
(d3)                             1 + x + x
(c4) reset(powerdisp);
(d4)                                done
(c5) ev(expr,powerdisp:true);
                                  2
(d5)                             x  + x + 1
```

Note that the local binding of **powerdisp** in (c5) doesn't modify the display in (d5) because the local binding is undone prior to the generation of the display.

13.3 Rewriting Expressions

The first technique for rewriting expressions uses the **rat** command, which converts an expression into CRE (*Canonical Rational Expression*) form. The CRE form is a recursive representation that is suited for polynomials and rational functions. A multivariate polynomial is represented in CRE form by rewriting the polynomial as a polynomial in the main variable. The coefficients are then rewritten as polynomials in the next main variable. This procedure is repeated for all of the variables. The main variables can be specified as optional arguments to the **rat** command (with the right-most variable assumed to be the most important) or to the **ratvars** option variable.

To illustrate the main variable concept, consider the following representations of a multivariate polynomial. In this example, we express a given multivariate polynomial with respect to different main variables using the **rat** function.

```
(c1) poly:(x+y+z+x*y)^2;
                                              2
(d1)                          (z + x y + y + x)
(c2) rat(poly,x);
          2             2                    2            2           2
(d2)/r/ (y  + 2 y + 1) x  + ((2 y + 2) z + 2 y  + 2 y) x + z  + 2 y z + y
(c3) rat(poly,z);
         2                          2            2       2            2
(d3)/r/ z  + ((2 x + 2) y + 2 x) z + (x  + 2 x + 1) y  + (2 x  + 2 x) y + x
```

```
(c4) rat(poly,y,x,z);
        2                            2             2       2             2
(d4)/r/ z  + ((2 y + 2) x + 2 y) z + (y  + 2 y + 1) x  + (2 y  + 2 y) x + y
(c5) rat(poly),ratvars:[y,x,z];
        2                            2             2       2             2
(d5)/r/ z  + ((2 y + 2) x + 2 y) z + (y  + 2 y + 1) x  + (2 y  + 2 y) x + y
```

The commands (c4) and (c5), which are equivalent, specify that the polynomial should be rewritten using the ordering $z > x > y$.

The next example shows how selective simplification functions can be used in conjunction with **rat** to format expressions. In this example, we reconstruct the original form of an expression.

We first generate a test expression and then **expand** it.

```
(c1) (a+b*x)*exp(x)/c + (x+1)^2 + x/y;
                                         x
                        x   (b x + a) %e          2
(d1)                    - + ------------ + (x + 1)
                        y         c
(c2) expr:expand(%);
                                x       x
                     x   b x %e    a %e     2
(d2)                 - + ------- + ----- + x  + 2 x + 1
                     y      c        c
```

The command **listratvars** displays the default variable ordering in the expression. To return the expression to its original form, using **rat**, make the exponential the main variable.

```
 (c3) listratvars(expr); x (d3) [a, b, c, x, %e , y]
(c4) rat(expr,exp(x));
                              x       2
               (b x + a) y %e  + (c x  + 2 c x + c) y + c x
(d4)/R/        --------------------------------------------
                                    c y
```

Now invoke some selective simplification functions to put the expression in the desired form. The **multthru** command in (c5) multiplies the denominator through the sum in the numerator. The **substpart** command in (c6) factors the third part of the expression in place.

```
(c5) multthru(%);
                                    x      2
                    x   (b x + a) %e    c x  + 2 c x + c
(d5)                - + ------------ + ----------------
                    y         c                c
(c6) substpart(factor(piece),%,3);
```

```
                                   x
                     x    (b x + a) %e            2
(d6)                 - + ------------- + (x + 1)
                     y          c
```

Note: this example could also have been produced without using **rat** by using two calls to **substpart**.

The next example in this section is a minor modification of the previous example. The only difference between the two is the replacement of *exp(x)* with *exp(-x)*, but this turns out to be significant because the conversion to rational form puts *exp(-x)* in the denominator. In such cases, it is simpler to eliminate the expression *exp(-x)* through substitution, put the resulting expression in the desired form, and substitute back.

```
(c1) (a+b*x)*exp(-x)/c + (x+1)^2 + x/y;
                                   - x
                     x    (b x + a) %e             2
(d1)                 - + --------------- + (x + 1)
                     y          c
(c2) expand(%);
                              - x       - x
                 x    b x %e       a %e        2
(d2)             - + --------- + ------- + x  + 2 x + 1
                 y        c          c
```

The first step is to substitute a token for *exp(-x)*.

```
(c3) expr:subst(exp(-x)=u,%);
                     x    2    b u x          a u
(d3)                 - + x  + ----- + 2 x + --- + 1
                     y          c            c
```

The next step is to rewrite the expression with *u* as the main variable.

```
(c4) listratvars(%);
(d4)                       [a, b, c, u, x, y]
(c5) rat(expr,u);
                                       2
               (b x + a) y u + (c x  + 2 c x + c) y + c x
(d5)/r/        ------------------------------------------
                                   cy
```

Finally, we simplify the expression and substitute back for *exp(-x)*.

```
(c6) multthru(%);
                           2
                   x    c x  + 2 c x + c    u (b x + a)
(d6)               - + ---------------- + -----------
                   y           c                 c
```

```
(c7) map('factor,%);
                          x   u (b x + a)          2
(d7)                      - + ----------- + (x + 1)
                          y        c
(c8) subst(u=exp(-x),%);
                                         - x
                        x   (b x + a) %e              2
(d8)                    - + --------------- + (x + 1)
                        y          c
```

The final example in this section uses the fact that variables recognized by **rat** can be kernels (expressions such as **sqrt**(x) or **integrate**($f(x)$,x)) rather than just symbols. In this example, we use **rat** to rewrite an expression, collecting coefficients of the integrals in the expression. The main variables are specified as optional arguments to the **rat** command.

```
(c1) expr:a*'integrate(f(x),x)+b*'integrate(g(x),x,0,1)+c*'integrate(f(x),x)+
          d*'integrate(g(x),x,0,1)+e;
            1              1
           /              /              /             /
           [              [              [             [
(d1)     d I  g(x) dx + b I  g(x) dx + c I f(x) dx + a I f(x) dx + e
           ]              ]              ]             ]
           /              /              /             /
            0              0
(c2) listratvars(expr);
                                                     1
                                      /          /
                                      [          [
(d2)                [a, b, c, d, e, I f(x) dx, I  g(x) dx]
                                      ]          ]
                                      /          /
                                                     0
(c3) rat(expr,'integrate(f(x),x),'integrate(g(x),x,0,1));
                           1
                          /                    /
                          [                    [
(d3)/r/           (d + b) I  g(x) dx + (c + a) I f(x) dx + e
                          ]                    ]
                          /                    /
                           0
```

Note: this technique is also useful for computation. Evaluating the integrals in (d3) (via **ev**($d3, integrate$) or **apply_nouns**($d3, integrate$)) requires only half as much work as for the mathematically equivalent expression (d1).

13.4 The mainvar Declaration

The **mainvar** declaration is used to make a variable "more important" than normal variables. For example, by default $x < y$, where $<$ is the ordering predicate. However, if x is declared to be **mainvar** then in subsequent computations $y < x$. (If y is also declared **mainvar** then $x < y$ again, but x and y are more important than any other variables.)

The **mainvar** declaration affects the internal representation of expressions by changing the canonical ordering of variables in simplified expressions. For this reason the **mainvar** declaration makes expressions context-sensitive since mathematically identical expressions can have different internal simplified forms, depending on the **mainvar** declarations in force when the expressions are simplified.

The next example shows how **mainvar** declarations affect the representation of expressions. Note that a resimplification (accomplished here using the **resimplify** command) is required for the **mainvar** declaration to take effect.

```
(c1) expr:[x+y+z,x*y*z];
(d1)                      [z + y + x, x y z]
(c2) declare(x,mainvar);
(d2)                             done
(c3) resimplify(expr);
(d3)                      [x + z + y, y z x]
```

With x declared **mainvar**, the ordering is $x > z > y$. When y is also declared **mainvar**, the ordering changes to $y > x > z$.

```
(c4) declare(y,mainvar);
(d4)                             done
(c5) resimplify(expr);
(d5)                      [y + x + z, z x y]
(c6) remove([x,y],mainvar);
(d6)                             done
(c7) resimplify(expr);
(d7)                      [z + y + x, x y z]
```

Note: A few points concerning the **mainvar** declaration should be noted. First, if you save an expression containing a variable declared to be **mainvar**, then you should also save the variable itself. This guarantees that the **mainvar** property will be present when the expression is reloaded. Second, a **mainvar** declaration establishes a context which might require an explicit resimplification to obtain fully simplified results.

In the following example, identical expressions are evaluated and simplified both before and after a **mainvar** declaration. Note that the result must be explicitly resimplified to get zero.

```
(c1) expr1:x+y;
(d1)                            y + x
(c2) declare(x,mainvar);
(d2)                             done
(c3) expr2:x+y;
(d3)                            x + y
(c4) expr1-expr2;
(d4)                            y - y
```

```
(c5) resimplify(%);
(d5)                                 0
```

13.5 Inhibiting Simplification

13.5.1 Using Invisible Boxes

You will sometimes want to display results in an unsimplified form. For example, you might want to keep certain terms in a product from combining. Simplifications of this nature can be inhibited by the **box** command. The disadvantage of this method is that you must dissect the expression and explicitly protect subexpressions by wrapping them in boxes.

The next example shows how to use **box** to inhibit simplification. Note the effect of setting of the option variable **boxchar** [*default:"*"*] to a more esthetic space character.

```
(c1) s/(s+1);
                                     s
(d1)                               -----
                                   s + 1
(c2) 2*box(s/(s+1));
                                   *******
                                   *  s  *
(d2)                             2 *-----*
                                   *s+ 1 *
                                   *******
(c3) boxchar:" "$
(c4) %th(2);
                                      s
(d4)                             2  -----
                                    s + 1
```

Note: the spacing between the 2 and the rational function in (d4) cannot be controlled. It is not possible to set **boxchar** to an empty string. If the display is ambiguous due to the lack of parentheses, reasonable results can be obtained by setting **boxchar** to an unobtrusive printing character such as `‘‘.’’` .

The final example in this section shows how **box** can be used to display a rational function as a constant times the ratio of two monic polynomials. The first step is to isolate the lead coefficients of the numerator and denominator and divide through to yield monic polynomials.

```
(c1) expr:(3*s+1)/(4*s^2+6*s-3);
                                  3 s + 1
(d1)                           --------------
                                   2
                               4 s  + 6 s - 3
(c2) (tmp1:ratcoef(num(%),s,hipow(num(%),s)),
      tmp2:ratcoef(denom(%),s,hipow(denom(%),s)))$
(c3) (num:multthru(num(expr)/tmp1),denom:multthru(denom(expr)/tmp2))$
```

```
(c4) num;
                                            1
(d4)                                    s + -
                                            3
(c5) denom;
                                      2   3 s     3
(d5)                                 s + --- -   -
                                          2      4
```

Now that the various pieces have been isolated, we reconstruct the expression. Note what happens if no **box** forms are used:

```
(c6) tmp1/tmp2*num/denom;
                                             1
                                     3 (s + -)
                                             3
(d6)                              ----------------
                                      2    3 s    3
                                  4 (s   + --- - -)
                                            2     4
```

After resetting **boxchar** to an invisible character, different representations of the expression can be generated by **box**ing different subexpressions.

```
(c7) boxchar:" "$
(c8) tmp1/tmp2*box(num/denom);
                                             1
                                         s + -
                                             3
                                  3   ------------
                                      2    3 s    3
                                     s   + --- - -
                                            2     4
(d8)                              ----------------
                                         4
(c9) box(tmp1/tmp2)*box(num/denom);
                                              1
                                          s + -
                                  3           3
(d9)                              -    ------------
                                  4     2    3 s    3
                                       s   + --- - -
                                              2     4
(c10) box(box(tmp1/tmp2)*num)/denom;
```

```
                                 3       1
                                 -  (s + -)
                                 4       3
(d10)                          -------------
                                2   3 s   3
                               s  + --- - -
                                     2    4
```

Note: expressions containing **box** forms are not suitable for computation. Any **box** forms should be removed prior to computation via the **rembox** command.

13.5.2 Using Null Functions

Another way of prohibiting simplification is to use a *null function* as a placeholder. That is, the expression to be protected is given as the argument of a function named ‘‘’’. Consider the following example:

```
(c1) (s/(s+1))*(s/(s-1));
                                     2
                                    s
(d1)                         ---------------
                             (s - 1) (s + 1)
(c2) " "(s/(s+1))*" "(s/(s-1));
                                 s       s
(d2)                         (-----) (-----)
                              s - 1   s + 1
```

Note that, unlike the **box** technique, this method puts parentheses around the factors.

```
(c3) boxchar:" "$
(c4) box(s/(s+1))*box(s/(s-1));
                                 s       s
(d4)                           -----   -----
                               s - 1   s + 1
```

Finally, note that the null function can be removed with an appropriate **lambda** substitution:

```
(c5) opsubst(" "=lambda([arg],arg),d2);
                                     2
                                    s
(d5)                         ---------------
                             (s - 1) (s + 1)
```

13.6 The ordergreat and orderless Commands

Another facility which allows the user to control the ordering of symbols is the **ordergreat/orderless** facility. This facility, unlike the **mainvar** declaration, gives the user complete control over the ordering of symbols. (The ordering among symbols which have been declared to be **mainvar**s is reverse-alphabetic. The user has no control over the ordering of **mainvar**s.) However, because of the way in which the **ordergreat/orderless** mechanism is implemented, results generated using this scheme are even more context-sensitive than those generated using the **mainvar** scheme.

The syntax of **ordergreat** is `ordergreat(`*var1*`, `*var2*`, ..., `*varN*`)`. This establishes the ordering *var1* > *var2* > ... > *varN* > *any other variables*. The **orderless** function is called in the same way and establishes an ordering making its arguments less important than any other symbol. The ordering imposed by either of these commands is removed using the **unorder** command. A second call to **ordergreat** or **orderless** without first calling **unorder** results in an error.

The functions **ordergreat** and **orderless** work by establishing aliases between the specified symbols and special internal symbols which are named to be either the most important or the least important. Since aliases are handled only by the parser (for input) and the displayer (for output), an existing expression cannot be reformatted using this mechanism unless it is re-entered.

The next example demonstrates how **ordergreat** is used. The order $x > z > y$ is established.

```
(c1) [x+y+z,x*y*z];
(d1)                          [z + y + x, x y z]
(c2) ordergreat(x,z,y);
(d2)                                 done
(c3) [x+y+z,x*y*z];
(d3)                          [x + z + y, y z x]
(c4) unorder();
(d4)                              [y, z, x]
(c5) [x+y+z,x*y*z];
(d5)                          [z + y + x, x y z]
```

The next example demonstrates that the **ordergreat** and **orderless** commands make Macsyma expressions context-dependent. In this example, *expr1* and *expr2* contain variables with different internal names since the **ordergreat** command in (c2) establishes aliases for x and y. The internal names generated by the **ordergreat** command are shown in (d7).

```
(c1) expr1:x+y;
(d1)                                y + x
(c2) ordergreat(x,y);
(d2)                                 done
(c3) expr2:x+y;
(d3)                                x + y
(c4) expr1-expr2;
(d4)                          - x - y + y + x
(c5) unorder();
(d5)                                [y, x]
(c6) expr1;
(d6)                                y + x
(c7) expr2;
```

```
(d7)                          _102X + _101y
```

As mentioned earlier, an existing expression must be re-parsed before any **ordergreat** or **orderless** orderings will be used. The only practical ways to re-parse an expression are via the **stringout** and **batch** commands or via **medit**, since neither re-evaluation nor resimplification invokes the parser. The next example shows how **stringout** and **batch** can be used.

```
(c1) expr:x+y+z;
(d1)                           z + y + x
(c2) ordergreat(x,y);
(d2)                              done
```

Re-evaluating the existing expression doesn't pick up the new ordering. The **stringout** command in (c4) writes the indicated expression to a file in Macsyma format, and the **batch** command in (c5) reads the expression back in. The ordering is picked up when the Macsyma expressions are parsed.

```
(c3) expr;
(d3)                           z + y + x
(c4) stringout("expr.mac",expr)$
(c5) batch("expr.mac");
(c6) Z+Y+X;
(d6)                           x + y + z
(d7)                              done
```

The final example shows how **medit** can be used to reparse an expression.

```
(c1) expr:x+y+z;
(d1)                           z + y + x
(c2) ordergreat(x,y);
(d2)                              done
(c3) expr;
(d3)                           z + y + x
(c4) medit(expr);
In editor:
$Z+Y+X
_ $$
(d5)                           x + y + z
```

Macsyma allows considerable control over the simplifying of expressions. The **rat** command, which converts an expression into CRE form, is useful when it is desired to collect coefficients of specified variables. (The "variables" used by **rat** need not be symbols. Non-atomic kernels, such as *exp(x)* or **integrate** forms, can be also used.) The **mainvar** declaration allows the user to specify a subset of "important" variables, although he has no control over the ordering within this subset. The **ordergreat/orderless** facility allows the user to explicitly define the ordering of the main variables, but in a way that requires existing expressions to be re-parsed before they can be used consistently.

Chapter 14

The Macsyma Pattern Matcher

The Macsyma pattern matching facility provides a powerful and useful method of testing expressions to see if they contain expressions of a specified "form," and also to perform literal or semantic substitutions. The pattern matching facility also gives the user the ability to extend the Macsyma simplifier by defining substitution rules that are applied automatically during the Macsyma simplification phase. Finally, the ability to define recursive rules permits the user to implement recursive simplifications conveniently. This chapter introduces you to the Macsyma pattern matchers and discusses the circumstances under which the pattern matcher is most useful. Because of the complexity of the subject, this chapter is intended to introduce you to pattern matching. Refer to "Pattern Matching and Related Functions" in the *Macsyma Reference Manual* for full details.

14.1 Introduction to Pattern Matching Techniques

The Macsyma pattern matchers can be thought of as powerful substitution mechanisms. The "substitution" is defined in terms of a *rule*, which defines a substitution for a given class of expressions. The method by which the rule is defined determines if the rule will be applied automatically or only when you request it, and also whether or not it is applied recursively.

Modifying an expression using pattern matching techniques is an alternative to the programmatic approach. In the following example, we show how to compute the factorial of a positive integer using both programming techniques and pattern matching. First, we implement the factorial function as a recursive function.

```
(c1) fact(int):=block
(if int = 0
  then 1
  else if not(integerp(int)) or int < 0
        then 'fact(int)
        else int * fact(int-1))$
(c2) fact(-1);
(d2)                    fact(- 1)
(c3) fact(0.5);
(d3)                    fact(0.5)
(c4) fact(0);
(d4)                        1
(c5) fact(5);
```

```
(d5)                         120
```

Next, we implement the factorial function with rules as an extension to the Macsyma simplifier.

```
(c1) posintp(n):=is(integerp(n) and n>0)$
(c2) matchdeclare(int,posintp)$
(c3) tellsimpafter(fact(0),1)$
(c4) tellsimpafter(fact(int),int*fact(int-1))$
(c5) fact(-1);
(d5)                      fact(- 1)
(c6) fact(0.5);
(d6)                      fact(0.5)
(c7) fact(0);
(d7)                          1
(c8) fact(5);
(d8)                         120
```

Although this process looks the same as the implementation using a recursive function definition, it is in fact quite different. The **tellsimpafter** approach teaches the Macsyma simplifier how to simplify *fact* forms. A major advantage of this approach is that additional simplifications on *fact* can be added simply by implementing additional **tellsimpafter** rules, without affecting the existing ones. Very powerful rule-based transformations can be “taught” to Macsyma by incrementally adding small components. This modularity makes it easy to add enhancements in stages, without making it necessary to modify existing (and previously debugged) code.

A major disadvantage of pattern matching is that it is a relatively slow mechanism. Adding rules that are tested automatically, as in this example, can cause a noticeable decrease in speed, affecting every Macsyma computation. The question of when to use pattern matching instead of a programmatic approach will be discussed in Section 14.8. A second disadvantage is that the user must learn how to use the pattern matcher, which is itself a nontrivial task.

Two distinct pattern matchers are implemented in Macsyma.

- The general Macsyma pattern matcher permits you to define rules that are applied only when explicitly requested (rules defined using the **defmatch** or **deftaylor** forms) or rules that are applied automatically by the simplifier (rules defined using the **tellsimp** or **tellsimpafter** forms).
- The **let** pattern matcher uses the rational functions package and is based on a pattern matching algorithm first implemented in the REDUCE computer algebra system. It is designed to make substitutions for products in rational expressions.

In the remainder of this chapter, we refer to the two pattern matchers as “the general pattern matcher” and “the rational function pattern matcher” or “the **let** pattern matcher,” respectively. The choice of the appropriate pattern matcher depends on the class of expressions and the nature of the substitution.

14.2 An Overview of the Pattern Matcher Facilities

As mentioned earlier, the pattern matcher is a mechanism that tests an expression to see if it contains a subexpression of a given form, and if so takes a specified action (*i.e.*, perform a substitution, side-effect a variable, or construct a list of parts of the subexpression). This section introduces the different Macsyma pattern matchers, provides examples of their use, and also discusses typical problems and issues encountered in pattern matching.

The Macsyma pattern matching components fall into three general classes:

- *Predicates:* A predicate is a boolean function, *i.e.*, a function that returns **true** or **false**. For example, the predicate function **integerp** returns **true** if and only if its argument evaluates to an integer. (The -p suffix is commonly used to indicate that a function is a predicate. Not all Macsyma predicates follow this rule, however. For example, the predicate that tests to see if its argument is an atom is called **atom**, not **atomp**).
- *Part Identification:* This mechanism, implemented with the **defmatch** mechanism in the Macsyma general pattern matcher, tests an expression to see if it is of a specified form. If so, the operation returns a list of the pattern variables as they matched in that expression.
- *Rewrite Rules:* A rewrite rule is an operation that matches and substitutes subexpressions in a given expression. This mechanism is implemented using **defrule**, **tellsimp**, and **tellsimpafter** in the Macsyma general pattern matcher, and by **let** in the Macsyma rational function pattern matcher.

Semantic pattern variables are defined using the **matchdeclare** facility, which uses predicates to test expressions to see if they match a specified form. The ability to define semantic (as opposed to literal) matches greatly enhances the usefulness of the pattern matcher. More will be said about semantic pattern variables and **matchdeclare** in a later section.

The following simple examples show how the different pattern matchers are used.

14.2.1 Examples of Predicates

The predicate **integerp** returns **true** if its argument evaluates to an integer, and **false** otherwise. The predicate **atom** returns **true** if its argument evaluates to an atom (a simple data structure with no component parts), and **false** otherwise. Predicates are the simplest pattern matchers, indicating whether or not an expression is of a specified form.

```
(c1) integerp(1);
(d1)                       true
(c2) integerp(1.0);
(d2)                      false
(c3) atom(x);
(d3)                       true
(c4) atom(x+y);
(d4)                      false
```

14.2.2 An Example of a Pattern Matching Rule

The following example shows how to define and apply a rule which replaces one algebraic expression by another. We define two variations of the rule. The first version of the rule searches for the exact pattern of variables which it is told to find and replace, The second version can accept an arbitrary variable name where it expects to find the velocity variable v.

We start with the expression for Newtonian kinetic energy. We then define the rule **relativize1** which converts Newtonian kinetic energy to relativistic energy.

```
(c1) energy:1/2*m*v^2;
                                  2
                                mv
(d1)                           -----
                                 2
(c2) defrule(relativize1,1/2*m*v^2,m*c^2/sqrt(1-v^2/c^2));
                                               2         2
                                           m v         c   m
(d2)                      relativize1 : ---- -> ------------
                                             2                  2
                                                              v
                                                  sqrt (1 - --)
                                                                2
                                                              c
```

This rule changes the nature of the expression **energy**. See **apply1**, page 234.

```
(c3) apply1(energy, relativize1);
                                  2
                                c   m
(d3)                          ------------
                                        2
                                       v
                              sqrt(1 - --)
                                        2
                                       c
```

Now try the rule where the velocity is represented by the symbol u instead of the symbol v. The rule does not work because it sees the letter u where it expects to see the letter v.

```
(c4) energy: 1/2*m*u^2;
                                 2
                              m u
(d4)                          ----
                                2
(c5) apply1(energy, relativize1);
                                 2
                              m u
(d5)                          ----
                                2
```

If we declare that, in pattern definitions, the variable v stands for any atomic symbol, we can then define the rule **relativize2** with this new interpretation of v. The rule works because u is an atomic variable, which matches the symbol v in the definition of the rule **relativize2**.

```
(c6) matchdeclare(v,atom)$
(c7) defrule(relativize2,1/2*m*v^2,m*c^2/sqrt(1-v^2/c^2));
```

```
                                         2         2
                                      m v       c  m
(d7)                  relativize2 : ---- -> ------------
                                      2                2
                                                      v
                                           sqrt (1 - --)
                                                       2
                                                      c
(c8) apply1(energy, relativize1);
                               2
                             c  m
(d8)                      ------------
                                   2
                                  u
                          sqrt(1 - --)
                                    2
                                   c
```

14.2.3 An Example of a User-Defined Pattern-Testing Predicate

In the following example, the **defmatch** mechanism, which is part of the Macsyma general pattern matcher, is used to define a function called **quadratic** that tests its first argument to see if it is a quadratic polynomial with respect to the second argument.

```
(c1) matchdeclare([a,b,c],freeof(x));
(d1)                         done
(c2) defmatch(quadratic,a*x^2+b*x+c,x);
(d2)                      quadratic
(c3) quadratic(a2*z^2+a1*z+a0,z);
(d3)          [c = a0, a = a2, b = a1, x = z]
(c4) quadratic(a1*z+a0,z);
(d4)           [c = a0, a = 0, b = a1, x = z]
(c5) quadratic(a3*z^3+a2*z^2+a1*z+a0,z);
(d5)                        false
```

The **matchdeclare** form in (c1) defines the pattern variables. In this case, the **matchdeclare** form tells the pattern matcher that the coefficients cannot contain the variable given as the last argument to **quadratic**. (This association comes from the fact that the symbol x is used in both the **matchdeclare** form in (c1) and also in the **defmatch** form in (c2).)

Note that this implementation of **quadratic** allows linear or constant polynomials to match the quadratic expression pattern by allowing the appropriate coefficient to be zero. In a subsequent section, we show how to write **quadratic** to prevent linear or constant polynomials from matching the pattern.

Note also that **freeof** takes more than one argument when used interactively, but is used with only one argument in (c1). The **freeof** form in (c1) is not evaluated directly, but is instead used as a template when the rule **quadratic** is defined using the **defmatch** form in (c2). This is discussed in greater detail in Section 14.4.

14.2.4 Examples of Rewrite Rules

The first example uses **defrule** to define a rewrite rule called **fsub** to perform the substitution $f(1) \to f1$. The rule is applied manually using the **apply1** form.

```
(c1) defrule(fsub,f(1),f1);
(d1)                 fsub : f(1) -> f1
(c2) f(1)+1/f(1)+f(2);
                                     1
(d2)                 f(2) + f(1) + ----
                                    f(1)
(c3) apply1(%,fsub);
                             1
(d3)                   f1 + -- + f(2)
                             f1
```

The next example uses **tellsimpafter** to define a rewrite rule that performs the same substitution. However, this rule is applied automatically during the simplification phase.

```
(c1) tellsimpafter(f(1),f1);
(d1)                  [frule1, false]
(c2) f(1)+1/f(1)+f(2);
                             1
(d2)                   f1 + -- + f(2)
                             f1
```

The next example uses the rational function pattern matcher to perform the same substitution. The rewrite rule is defined using the **let** command, and is applied manually using the **letsimp** command.

```
(c1) let(f(1),f1);
(d1)                    f(1) --> f1
(c2) f(1)+1/f(1)+f(2);
                                     1
(d2)                 f(2) + f(1) + ----
                                    f(1)
(c3) letsimp(%);
                        2
                     f1  + f(2) f1 + 1
(d3)                 -----------------
                             f1
```

The final example shows how the direction in which the pattern matcher scans the expression can be controlled. The **matchdeclare** form in (c1) allows any expression in the position indicated by x in (d2) to match the pattern. The **apply1** and **applyb1** commands force the matcher to scan the expressions "top-down" and "bottom-up," respectively.

```
(c1) matchdeclare(x,true);
(d1)                         done
```

```
(c2) defrule(fsub2,f(f(x)),g(x));
(d2)              fsub2 : f(f(x)) -> g(x)
(c3) apply1(f(f(f(x))),fsub2);
(d3)                      g(f(x))
(c4) applyb1(f(f(f(x))),fsub2);
(d4)                      f(g(x))
```

Note that the result (d3) results from scanning the expression "top-down," while the result in (d4) results from scanning the expression "bottom-up."

14.2.5 General Pattern Matcher Issues

Finally, we mention some of the difficulties and questions that frequently arise when using the pattern matcher facilities. These topics will be covered in greater detail in subsequent sections.

- **Looping and recursive rules.** Many rules are applied recursively "until the result stops changing". A rule that substitutes $f(f(x))$ for $f(x)$, and is applied recursively, will loop until the resulting expression exceeds an internal storage space. A rule that substitutes $f(x)$ for $f(x)$ might loop if the "new" expression is determined to be different from the "old" one after the transformation has occurred. Macsyma will sometimes warn the user that a circular definition is being attempted (for example, when the original and replacement expressions in a **tellsimp/tellsimpafter** rule are identical). This may complicate debugging.
- **Specifying a semantic pattern.** The power of the pattern matcher is greatly enhanced by permitting a *semantic* (as opposed to a *literal*) match. For example, if the pattern identifies a form such as $f(x)$ with the understanding that f refers to the symbol f, but that x is a "placeholder" and can refer to expressions other than the symbol x then we refer to it as a semantic pattern. If, however, the pattern requires only that the expression $f(x)$ will match the pattern, then we refer to it as a literal pattern.
- **In what direction does the matcher scan the expression?** The transformation $f(f(y)) \rightarrow g(y)$, where y can be anything (that is, y is a semantic pattern variable that can match an arbitrary expression), yields different results if the expression is scanned "top-down" instead of "bottom-up". For example, consider the expression $f(f(f(x)))$. Applying the transformation "top-down" yields $g(f(x))$, while applying it "bottom-up" yields $f(g(x))$.
- **How "smart" is the matcher?** Can (or should) the pattern matcher match, say, $exp(x)$ in $cosh(x)$? Should it be able to find a leading factor of 1 (or -1) in a product?

14.3 Simple Pattern Testing: Predicates

The simplest type of pattern is the **predicate**. A predicate is a function that returns either **true** or **false**. Macsyma provides a large number of built-in predicates, and the user can define compound predicates using the **is** form. Consult the *Macsyma Reference Manual* for a complete list of Macsyma predicates.

In the following example, we define a predicate that takes two arguments and returns **true** if and only if the first argument is a symbol and the second argument doesn't contain that symbol.

```
(c1) symbol_and_freeofp(var,expr):=is(symbolp(var1) and freeof(var,expr))$
(c2) symbol_and_freeofp(x,1+x);
(d2)                       false
```

```
(c3) symbol_and_freeofp(x+y,x+y);
(d3)                       false
(c4) symbol_and_freeofp(x,y);
(d4)                        true
```

The rest of this section is devoted to explaining how to write predicates for pattern matching purposes. We first discuss the techniques used to write *compound predicates*, that is, predicates that are logical combinations of simple predicates.

A useful form when writing predicates is the **not**, or logical negation, operator. For convenience, **not** is implemented both as a prefix operator and as a function.

```
(c1) not true;
(d1)                       false
(c2) not(true);
(d2)                       false
```

Although most simple predicates are functions, compound predicates are constructed using the **is**, **and**, and **or** special forms. These are implemented as special forms rather than functions because you must evaluate the individual predicates sequentially.

The **is** form performs the logical evaluation of predicates which do not contain the usual logical operators **and**, **or**, and **not**. That is, the **is** form forces the logical evaluation of expressions such as inequalities that are themselves legitimate Macsyma expressions. It is good programming practice to write compound predicates using **is**, even when it is not needed to force the logical evaluation. The next example shows how to use **is**.

```
(c1) 2>1;
(d1)                       2 > 1
(c2) is(2>1);
(d2)                        true
(c3) true or false;
(d3)                        true
(c4) is(true or false);
(d4)                        true
```

Note that the **is** form is needed in (c2) to carry out the logical evaluation of the inequality. The command (c4) shows that **is** can be used, even if the logical evaluation occurs automatically.

In this example, we construct a predicate that returns **true** if its argument is a positive floating point number. Note what happens if the predicates are evaluated in the incorrect order.

```
(c1) floating_point_and_positivep(arg):=is(floatp(arg) and arg > 0)$
(c2) positive_and_floating_pointp(arg):=is(arg > 0 and floatp(arg))$
(c3) floating_point_and_positivep(1.0);
(d3)                        true
(c4) positive_and_floating_pointp(1.0);
(d4)                        true
(c5) floating_point_and_positivep(1);
(d5)                       false
```

```
(c6) positive_and_floating_pointp(1);
(d6)                     false
(c7) floating_point_and_positivep(a);
(d7)                     false
(c8) positive_and_floating_pointp(a);
Macsyma was unable to evaluate the predicate:
A > 0
Returned to Macsyma Toplevel.
```

Note that the test that $arg > 0$ must follow the floating point check or else a non-numeric input will generate a Macsyma error. The components of an **and** form are evaluated left-to-right until a component evaluates to **false**, in which case the remaining forms are not evaluated and **false** is returned. If all of the components evaluate to **true** then the value **true** is returned. An **or** form evaluates its components left-to-right until a component evaluates to **true**, in which case the remaining forms are not evaluated and **true** is returned. If none of the components evaluates to **true** then the value **false** is returned. The components of **and** and **or** must evaluate to either **true** or **false**. (Note that the construct **false or 1**, which is allowed in some languages, generates an error in Macsyma.)

As indicated earlier in this chapter, predicates are used to construct *pattern variables* for the Macsyma pattern matchers. The user defines a pattern variable using the **matchdeclare** form and associated predicates. The pattern matcher tests an expression to see if it fits the pattern by passing it to the indicated predicate. A predicate used in a **matchdeclare** form must have the expression to be tested appear at the end of the predicate argument list. Any parameters that are passed to the predicate must precede the expression being tested.

14.4 Literal vs. Semantic Matches: matchdeclare

Pattern matching provides a convenient means of implementing mathematical relationships. However, relationships of this nature must frequently be interpreted as identities. For example, the simplification $sin(x)^2 + cos(x)^2 = 1$ should hold regardless of the expression in the locations indicated by the placeholder x.

The effectiveness of a pattern matcher is greatly increased if the matcher can make semantic matches. The type of expression that can match a semantic pattern variable is determined by the **matchdeclare** property associated with a pattern variable. The **matchdeclare** property is generated by a **matchdeclare** form, which takes an even number of arguments. The first element in the pair of arguments to **matchdeclare** is a *pattern variable* (a variable that is used in a rule definition), and the second is a predicate that must evaluate to **true** for a test expression to match. If the first argument in a pair is a list of variables then each variable in the list acquires the same **matchdeclare** property. If no **matchdeclare** is specified then only a literal match can be made for that pattern variable.

In the previous section it was noted that the predicate used in a **matchdeclare** must be written so that the expression to be tested is the last argument in the predicate's argument list. When defining the pattern variable in a **matchdeclare** form, this last argument is omitted. For example, if a pattern variable named *intvar* should match integers, the predicate **integerp** should be used. Since **integerp** takes one argument, the **matchdeclare** form `matchdeclare(intvar, integerp)$` should be used.

It is important to note that the **matchdeclare** information is incorporated into the rule when the rule is defined. Simply redefining the **matchdeclare** property on a pattern variable will not affect existing rules that use the pattern variable. The rule must be redefined with the new **matchdeclare** in place before the new pattern variable will be used. However, a predicate that is specified in the **matchdeclare** form *is* used dynamically, so redefining a **matchdeclare** predicate will in fact change the behavior of rules using the corresponding pattern variables. This information is provided not so much to encourage a programmer to redefine pattern match predicates after a rule has been generated (this will usually lead to inexplicable behavior on the part of the pattern matcher), but rather to provide further insight into the pattern matching mechanism and also to point out that tracing the predicates or rules (using the Macsyma **trace** facility) can help to debug pattern matching problems.

Note: the presence of **matchdeclare** information on a symbol can be determined by inspecting the property list of the symbol. If such information is present, a **matchdeclare** indicator will appear on the property list. The **matchdeclare** information on a symbol can be inspected using the **printprops** special form.

```
(c1) matchdeclare(x,symbolp);
(d1)                        done
(c2) properties(x);
(d2)                [matchdeclare]
(c3) printprops(x,matchdeclare);
     matchdeclare(x, symbolp)
(d3)                        done
```

14.5 The General Pattern Matcher

There are three interfaces to the Macsyma general pattern matcher:

- the **defmatch** facility, which generates functions that identify subexpressions and binds them to user-specified variables;
- the **defrule** facility, which generates rewrite rules that are invoked manually;
- the **tellsimp/tellsimpafter** facility, which generates rewrite rules that are invoked automatically by the Macsyma simplifier.

This section begins with an overview of the general pattern matcher, then discusses the three interfaces in detail. The section concludes with a discussion of issues that arise when using the general pattern matcher with translated or compiled files.

14.5.1 An Overview of the General Pattern Matcher

The general pattern matcher is used extensively throughout Macsyma. The three interfaces all require that the same procedure be followed when using the general pattern matcher.

- To use the facility, the user first defines syntactic pattern variables using **matchdeclare**. Any predicates used in the **matchdeclare** calls should be defined prior to defining the rule.
- The next step is to define the rule using **defmatch**, **defrule**, **tellsimp**, or **tellsimpafter**.
- The procedure used to apply the rule depends on the method used to define it.

The process of defining a rule using the general pattern matcher generates a LISP function that, when invoked on a given expression, decides if the match can be made and, if so, takes the appropriate action. (The action may not transform the expression—a rule defined using **defmatch** identifies indicated subexpressions but doesn't transform the expression; rules defined using **deftaylor**, **tellsimp**, or **tellsimpafter** are rewrite rules and *do* transform the expression.) The resulting LISP function can be compiled for increased efficiency.

The problem of matching a subexpression to a pattern is a difficult one. The Macsyma pattern matchers don't employ full backtracking in the sense that if they search a branch of an expression tree for a match and find they cannot make one, they will not back up and search another branch. The result from the user's point of view is that the pattern matcher fails to match when the user thinks it should. This deficiency can usually be alleviated by defining less ambitious rules. In general, rules defining substitutions for sums or products will not work. In particular, the substitition $sin(x)^2 + cos(x)^2 = 1$ cannot be implemented directly.

In the remainder of this section, the three interfaces to the general pattern matcher are discussed in detail.

14.5.2 Identifying Subexpressions: defmatch

The **defmatch** facility identifies subexpressions and binds these subexpressions to user-specified variables. **defmatch** is faster than **deftaylor**, **tellsimp**, or **tellsimpafter**. It is commonly used to test an expression to see if it is of a specified form and, if so, identifies the components. For example, a quadratic polynomial equation solver can be implemented easily using **defmatch**.

The syntax of **defmatch** is **defmatch**(*name*, *pattern*, *parm1, ..., parmN*). **defmatch** generates a function called *name* that takes as its arguments an expression to be tested and the parameters *parm1* through *parmN*. To apply the rule, the user calls the function explicitly. The test pattern, indicated by *pattern*, can contain variables with **matchdeclare** properties. If the match succeeds, a list of the form $[patternvar1 = subexpr1, \ldots, patternvarK = subexprK]$ is returned where *patternvar1, ..., patternvarK* are pattern variables used in the definition of *pattern*. The pattern variables are also bound to the indicated expressions. The parameters are typically used to indicate, say, the independent variables in a polynomial, and are passed as arguments to the predicate(s) associated with the **matchdeclare** forms.

If the pattern contains no pattern variables or parameters then the function generated by **defmatch** returns **true** if the match succeeds. If the match fails, the function returns **false**. The function generated by **defmatch** is added to the system **rules** list, and can be removed using **remove** or **kill**.

A rule generated by **defmatch** can be inspected with the **disprule** command.

Note: The **defmatch** command is not recursive, since recursion does not make sense in this context. For this reason, **defmatch** is much faster than the other general pattern matcher facilities, and should be used whenever possible.

14.5.2.1 defmatch Summary

- **Defining defmatch rules: defmatch**(*name*, *pattern*, *parm1, ..., parmN*) generates a function *name* that is called by *name*(*expr*, *parm1, ..., parmN*). If the match succeeds, a list of the form $[patternvar1 = subexpr1, \ldots, patternvarK = subexprK]$ is returned. The toplevel bindings *patternvar1*:*subexpr1, ..., patternvarK*:*subexprK* are also made. If the match fails, **false** is returned. Since **defmatch** is a special form it does not evaluate any of its arguments.

- **Inspecting defmatch rules: disprule(***name***)** displays the rule named *name* that is generated by **defmatch**. The names of rules generated by **defmatch** are placed on the **rules** list.

- **Removing defmatch rules: remove(***name***, rule)** removes the rule named *name*. **kill(***name***)** removes all information associated with *name*, including the rule definition.

14.5.2.2 Examples of defmatch

The first example uses **defmatch** to identify a quadratic polynomial. The first step is to write a **defmatch** rule that identifies a quadratic polynomial (not an equation) and, if given such an expression, returns a list identifying the coefficients.

```
(c1) nonzero_and_freeof(x,expr):=is(expr#0 and freeof(x,expr))$
(c2) matchdeclare([b,c],freeof(x),a,nonzero_and_freeof(x))$
(c3) defmatch(quadratic,a*x^2+b*x+c,x)$
(c4) quadratic(3*x^2-5*x+6,x);
(d4)           [c = 6, a = 3, b = - 5, x = x]
(c5) [a,b,c];
(d5)                    [3, - 5, 6]
```

In lines (c1) through (c3), a **defmatch** function called **quadratic** is defined that tests an expression to see if it is a quadratic polynomial. If **quadratic** finds a match then a list containing the pattern variables and their values is returned. The pattern variables are also bound to these values. If the expression fails to match, **quadratic** returns **false**. In (c5), it is verified that the pattern variables are also bound at toplevel.

The next few commands show some examples for which **quadratic** fails to match. They also shows that **quadratic** can match expressions that are polynomials in non-atomic variables:

```
(c6) quadratic(a1*y+b1,y);
(d6)                       false
(c7) quadratic(a1*y^3+b1,y);
(d7)                       false
(c8) quadratic(aa*exp(2*t)+bb*exp(t)+cc,exp(t));
                                             T
(d8)         [c = cc, a = aa, b = bb, x = %e ]
```

The first argument in (c8) is a polynomial in the variable *exp(t)*.

Finally, we show why care must be taken when using **defmatch**-generated functions at toplevel. Consider the following example.

```
(c9) quadratic(3*x^2-5*x+6,x);
(d9)           [c = 6, a = 3, b = - 5, x = x]
(c10) quadratic(3*a^2-5*a+6,a);
(d10)                      false
```

Why did (c10) fail to match? Since **quadratic** is a toplevel Macsyma function, we can trace it to see what arguments it receives.

```
(c11) trace(quadratic)$
(c12) quadratic(3*a^2-5*a+6,a);
1 Enter quadratic [18, 3]
1 Exit  quadratic false
(d12)                      false
```

The **trace** information shows that the arguments to **quadratic** evaluate to integers. This is because the call to **quadratic** in (c9) bound the pattern variables a, b, and c to the values 3, -5, and 6 respectively, and these bindings were used in (c12) when the arguments to **quadratic** were evaluated. For this reason, **defmatch** rules are usually called inside a block with the pattern variables local to the block.

We can recover the expected behavior of **quadratic** by quoting the variables in the argument list.

```
(c13) quadratic(3*'a^2-5*'a+6,'a);
                            2
1 Enter quadratic [3 a  - 5 a + 6, a]
1 Exit  quadratic [c = 6, a = 3, b = - 5, x = a]
(d13)          [c = 6, a = 3, b = - 5, x = a]
```

The next example uses **defmatch** to find the homogeneous solution of a second-order constant-coefficient linear ordinary differential equation. The variables *a1* and *a2* represent the arbitrary constants of the solution. This example is adapted from the **demo** file **macsyma:ode;odetrix.demo**.

```
(c1) matchdeclare([b,c],freeof(u,x),f,freeof(u))$
(c2) defmatch(solde,'diff(u,x,2) + b*'diff(u,x) + c*u = f,u,x)$
(c3) solder(eqn,u,x):=block
     ([b,c,f,disc,r1,r2,alpha,beta],
      if solde(eqn,u,x) = false
       then return(false)
       else (disc: b^2 - 4*c, alpha: -b/2,
             if asksign(disc) = 'zero
              then return(%e^(alpha*x) * (a1 + a2*x))
              else (beta: sqrt(disc)/2,
                    if asksign(disc) = 'pos
                     then (r1: alpha + beta, r2: alpha - beta,
                           return(a1*%e^(r1*x) + a2*%e^(r2*x)))
                     else (beta: %i*beta,
                           return(%e^(alpha*x) * (a1*cos(beta*x)
                                   + a2*sin(beta*x)))))))$
```

In (c1) and (c2), a **defmatch** function is generated that matches constant-coefficient second-order linear ordinary differential equations. The scope of the match is determined both by the **matchdeclare** form in (c1) and by the structure of the test expression in (c2). Note that the expression must be an equation to match the pattern (this differs from the syntax of most built-in Macsyma solvers, where an expression is assumed to be an equation with right-hand side equal to zero if it is not an equation) and that the lead coefficient must be 1. These restrictions could be eliminated by modifying the **defmatch** form and adding another pattern variable to represent the coefficient of the second derivative term.

The function **solder** calls the **defmatch**-generated function **solde** to check to see if the expression is of the required form and, if so, identifies the various parts. If the expression is not of the required form then **solde** returns **false**. If the expression is of the required form, the solution of the characteristic polynomial is constructed and the appropriate form of the solution is generated. The **asksign** command is used to obtain the needed sign information.

Note also that the pattern variables in **solder** are local variables (either because they are block variables, or because they appear in the argument list). This is good programming practice since a **defmatch**-generated function binds the pattern variables, and failure to localize them with the **block** mechanism would leave them bound at toplevel.

We proceed to test the function.

```
(c4) eq1:'diff(y,t,2)+a*'diff(y,t)+b*y=f(t);
                  2
                 d y     dy
(d4)             --- + a -- + b y = f(t)
                   2     dt
                 dt
(c5) sol:solder(%,y,t);
          2
is   4 b - a    positive, negative, or zero?
p;
         a t
       - ---                  2
          2             sqrt(a  - 4 b) t
(d5) %e      (%i a2 sinh(----------------)
                                2
                                               2
                                        sqrt(a  - 4 b) t
                                + a1 cosh(----------------))
                                                2
(c6) ev(eq1,y=sol,diff, ratsimp);
(d6)                      0 = f(t)
```

The result (d6) doesn't simplify to $0 = 0$ because the solution (d5) is only the homogenous solution of the given differential equation.

The next few examples show that **solder** fails to match equations that don't match the pattern specified by the **defmatch** form.

```
(c7) eq3:'diff(y,t,2)+t*'diff(y,t)+y=0;
                     2
                    d y     dy
(d7)                --- + t -- + y = 0
                      2     dt
                    dt
(c8) solder(%,y,t);
(d8)                       false
```

The equation in (d7) fails to match because the coefficient of the first derivative is not a constant.

```
(c9) 'diff(y,t,2)+a*'diff(y,t)+y;
                           2
                          d y     dy
(d9)                      --- + a -- + y
                            2     dt
                          dt
(c10) solder(%,y,t);
(d10)                         false
```

The expression in (d9) fails to match because it is not an equation.

```
(c11) 'diff(y,t,2)+y*'diff(y,t)+3*y=0;
                        2
                       d y     dy
(d11)                  --- + y -- + 3 y = 0
                         2     dt
                       dT
(c12) solder(%,y,t);
(d12)                         false
```

The expression in (d11) fails to match because it is a nonlinear equation.

```
(c13) expand(2*eq1);
                   2
                  d y       dy
(d13)           2 --- + 2 a -- + 2 b y = 2 f(t)
                    2       dt
                  dT
(c14) solder(%,y,t);
(d14)                         false
```

The expression in (d13) fails because the lead coefficient is different from 1.

14.5.3 Transforming Expressions with Rewrite Rules: defrule

The **defrule** interface to the general pattern matcher defines rewrite rules that modify expressions by recursive substitutions. Rules generated by **defrule** are applied manually, and the user determines how the expression is scanned and the order in which the rules are applied.

The syntax for **defrule** is **defrule(***name***,** *pattern***,** *replacement***)**, where *name* is the name of the rule, *pattern* is an expression containing pattern variables, and *replacement* is an expression to be substituted in for an expression that matches *pattern*. As usual, if no pattern variables are specified by **matchdeclare** then only literal match will be made.

Once a rule has been implemented by **defrule**, it is applied manually by a call to **apply1**, **applyb1**, **apply2**, or **applyb2**. These special forms are called with the same syntax, but the manner in which they apply the rules is different. The syntax is **applyXX**(*expr*, *rule1, ..., ruleN*) where *expr* is an expression to be simplified and *rule1, ..., ruleN* are the names of rules, and XX is 1,2,b1 or b2.

A rule generated by **defrule** is added to the system **rules** infolist. The rule definition can be inspected by typing **disprule**(*name*), where *name* is the name of the rule. To remove a rule definition, type **remove**(*name, rule*) or **kill**(*name*). The former is preferred, since it removes only the rule associated with *name*, whereas the latter removes all information associated with *name*. To remove all rules generated using **defrule**, **defmatch**, or **tellsimp/tellsimpafter**, type **remove**(**all**, **rule**);. Typing **kill**(**rules**); is an alternative method of removing all of the rules.

14.5.3.1 defrule Summary

- **Defining a Rule: defrule**(*name*, *expr*, *repl*) generates a rule called *name* that replaces *expr* with *repl*. The scope of the match is determined by the **matchdeclare** properties present on the pattern variables when the **defrule** form is executed.
- **Displaying a Rule: disprule**(*name1*, *name2, ..., nameN*) displays the substitution rules of the named rules *name1*, *name2, ..., nameN*.
- **Removing a Rule: remove**(*name*, rule) deletes the rule *name*. To delete several rules use **remove**([*name1*, *name2, ..., nameN*], rule). The command **kill**(*name*) removes all information associated with *name*, including any rule definitions. The command **kill**(rules) deletes all rules named on the system **rules** infolist, which contains the names of rules defined using **defrule**, **defmatch**, **tellsimp**, and **tellsimpafter**.
- **Applying a Rule:** The forms **apply1**, **apply2**, **applyb1**, and **applyb2** apply rules generated using **defrule**. The syntax is **applyfn**(*expr*,*rule1, ..., ruleN*) where **applyfn** is one of the four previously mentioned special forms, *expr* is the expression to simplify, and *rule1, ..., ruleN* are the names of rules defined using **defrule**. The manner in which the rules are applied is summarized below.

Applying **defrule** rules:

- **apply1**(*expr*, *name1, ..., nameN*) repeatedly applies the rule specified by *name1* to *expr* top-down until it fails, then applies the rule to all subexpressions of the resulting expression. The recursive application of the rule ends when the rule has been applied at the maximum allowable depth. The process is repeated on the resulting expression with the remaining rules.
- **apply2**(*expr*, *name1, ..., nameN*) applies the rule specified by *name1* to *expr* top-down until it fails. The remaining rules are applied sequentially at the same level until all have failed. The procedure then repeats on the subexpressions of the result.
- **applyb1**(*expr*, *name1*, ..., *nameN*) repeatedly applies the rule specified by *name1* to *expr* bottom-up until it fails, then applies the rule to all subexpressions of the resulting expression. The recursive application of the rule ends when the rule is applied to the maximum allowable height. The process is repeated on the resulting expression with the remaining rules.
- **applyb2**(*expr*, *name1*, ..., *nameN*) applies the rule specified by *name1* to *expr* bottom-up until it fails. The remaining rules are applied sequentially at the same level until all have failed. The procedure then repeats on the subexpressions of the result.

Note: the system option variables **maxapplydepth** (*default:10000*) and **maxapplyheight** (*default:10000*) control the maximum depth and height, respectively, to which the **apply** functions search.

14.5.3.2 Examples of defrule

This example uses **defrule** to perform the syntactic substitution $\cot x \to \frac{\cos x}{\sin x}$. This is a syntactic substitution since it is defined for any expression in the position indicated by x.

```
(c1) matchdeclare(angle,true)$
(c2) defrule(cotrule,cot(angle),cos(angle)/sin(angle));
                                        cos(angle)
(d2)         cotrule : cot(angle) -> ----------
                                        sin(angle)
(c3) [cot(x),exp(cot(2*y)),a*cot(u+v+z)/b];
                         cot(2 y)  a cot(z + v + u)
(d3)       [cot(x), %e         , ----------------]
                                          b
(c4) applyb1(%,cotrule);
                         cos(2 y)
                         --------
            cos(x)       sin(2 y)  a cos(z + v + u)
(d4)       [------, %e         , ----------------]
            sin(x)                 b sin(z + v + u)
```

The next example uses **defrule** to define a finite-differencing utility for ordinary differential equations. For simplicity, we assume that the differential equations use y and x as the dependent and independent variables respectively, and h is the step size. The following differencing schemes are implemented:

- **forward**: $\frac{dy(x)}{dx} \to \frac{y(x+h)-y(x)}{h}$
- **centered**: $\frac{dy(x)}{dx} \to \frac{y(x+h)-y(x-h)}{2h}$
- **backward**: $\frac{dy(x)}{dx} \to \frac{y(x)-y(x-h)}{2h}$

The finite differencing mechanism is implemented as follows:

- The differencing scheme to be used is determined by the setting of the option variable *difference_type*, which defaults to **forward**.
- The rule **diftran** substitutes the pseudo-operator **delta[n]** (which is really a just a placeholder) for nth-order derivatives of y with respect to x.
- The rule **deltatran_n** recursively simplifies differences by first rewriting

  $$delta[n](expr) \to delta[n-1](delta[1](expr))$$

  and then differencing the $delta[1](expr)$ form.
- The rule **deltatran_0** implements the simplification $delta[0](expr) \to expr$.

The function **fin_dif** takes an expression and applies the rules as necessary.

We begin by declaring the option variable *difference_type* and setting up the rules.

```
(c1) define_variable(difference_type, 'forward, any_check)$
(c2) put('difference_type,
      lambda([foo],if not(member(foo,'[forward,central,backward]))
                   then error("not a valid differencing method: ",foo)),'value_check)$
```

```
(c3) posintp(n):=is(integerp(n) and n>0);
(d3)      posintp(n) := is(integerp(n) and n > 0)
(c4) matchdeclare(n,posintp,t,true)$
(c5) defrule(diftran,'diff(y(x),x,n),delta[n](y(x)));
                      n
                     d
(d5)        diftran : --- (y(x)) -> delta (y(x))
                       n                n
                     dx
```

The option variable *difference_type* contains the names of the differencing schemes. The rule **diftran** converts **diff** forms into expressions containing **delta**. Next, we implement the simplification rules for **delta**.

```
(c6) defrule(deltatran_0,delta[0](t),t);
(d6)            deltatran_0 : delta (t) -> t
                                   0
(c7) defrule(deltatran_n,delta[n](t),
             delta[n-1](apply('apply1,[delta[1](t),difference_type])));
(d7) deltatran_n : delta (t) ->
                       n
    delta     (apply('apply1, [delta (t), difference_type]))
         n - 1                      1
```

Note that the name of the rule implementing the differencing scheme is given in the variable *difference_type*, which is evaluated in the **apply** form.

Next, we implement the differencing schemes.

```
(c8) defrule(forward,delta[1](t), ratsimp((at(t,x=x+h)-t)/h));
                                    at(t, x = x + h) - t
(d8) forward : delta (t) -> ratsimp(--------------------)
                     1                       h
(c9) defrule(central,delta[1](t), ratsimp((at(t,x=x+h)-at(t,x=x-h))/(2*h)));
(d9) central : delta (t) ->
                     1
                       at(t, x = x + h) - at(t, x = x - h)
                ratsimp(-----------------------------------)
                                       2 h
(c10) defrule(backward,delta[1](t), ratsimp((t-at(t,x=x-h))/h));
                                      t - at(t, x = x - h)
(d10) backward : delta (t) -> ratsimp(--------------------)
                       1                       h
```

Before we go any further, we test the rules.

```
(c11) diff:'diff(y(x),x,2);


                           2
                          d
(d11)                     --- (y(x))
                            2
                          dx
(c12) del:apply1(diff,diftran);
(d12)                    delta (y(x))
                              2
(c13) apply1(del,deltatran_n),difference_type:'forward;
                y(x + 2 h) - 2 y(x + h) + y(x)
(d13)      delta (------------------------------)
               0                2
                               h
(c14) apply1(del,deltatran_n),difference_type:'backward;
                 2 y(x - h) - y(x - 2 h) - y(x)
(d14)     delta (- ------------------------------)
              0                  2
                                h
(c15) apply1(del,deltatran_n),difference_type:'central;
               y(x + 2 h) + y(x - 2 h) - 2 y(x)
(d15)     delta (--------------------------------)
              0                  2
                              4 h
(c16) apply1(%,deltatran_0);
             y(x + 2 h) + y(x - 2 h) - 2 y(x)
(d16)        --------------------------------
                                2
                             4 h
```

All of the rules work as expected. The next step is to define a function called **fin_dif** that applies the rules in the proper order. Note that we need to apply **diftran** only one time, so we invoke the rule in a separate **apply1** form.

```
(c17) FIN_DIF(expr):=block
      (expr:apply1(expr,diftran),
       apply2(expr,deltatran_n,deltatran_0))$
```

We now compile the rules for increased efficiency and test **fin_dif**.

```
(c18) compile_rule(all)$
(c19) fin_dif(a*'diff(y(x),x)+b*y(x)=g(x));
            a (y(x + h) - y(x))
(d19)       ------------------- + b y(x) = g(x)
                     h
```

```
(c20) fin_dif('diff(y(x),x,2)+'diff(y(x),x)+3*y(x)=0);


       y(x + 2 h) - 2 y(x + h) + y(x)   y(x + h) - y(x)
(d20) ------------------------------ + ---------------
                      2                       h
                     h
                                                + 3 y(x) = 0
```

14.5.4 Automatic Simplification of Expressions: tellsimp and tellsimpafter

The **tellsimp** and **tellsimpafter** commands provide a means of extending the Macsyma simplifier. Rules defined using these two special forms are applied automatically and recursively during the simplification stage. As the names suggest, rules defined using **tellsimp** are applied prior to the Macsyma simplification stage, while rules defined by **tellsimpafter** are applied after the Macsyma simplification stage. For most applications, **tellsimpafter** should be used since this allows the Macsyma simplifier to cancel any subexpressions that would otherwise have to be scanned. **tellsimp** should be used only when it is necessary to override a built-in simplification.

Since the two forms are identical except for the point in the Macsyma simplification process in which they are applied, the subsequent discussion will mention **tellsimpafter** only. However, unless otherwise noted, the remarks also apply to **tellsimp**.

The syntax for **tellsimpafter** is **tellsimpafter**(*pattern*, *repl*, *cond*), where *pattern* is an expression to search for and *repl* is the replacement expression when a match is found. The variable *cond* is an optional argument that, if present, is a predicate depending on the pattern variables that must evalute to **true** for the rule to fire. (This feature permits **tellsimpafter** rules to be defined recursively.) The value returned by a **tellsimpafter** form is a list containing the name of the rule and the name of the internal simplification function that applies the rule.

As usual, the scope of the match is determined by the **matchdeclare** properties of the associated pattern variables. If no **matchdeclare** properties are present then only literal matches will be made. Since the **matchdeclare** information is used only when the **tellsimpafter** form is executed (thereby defining the rule), the old rule must be removed and the new rule defined if it is desired to change the **matchdeclare** properties of any of the pattern variables.

The rule name generated by **tellsimpafter** is based on the main operator in *pattern*. If the main operator is a symbol, the rule name will be a symbol; otherwise, the rule name is a string. (Note that if the rule name is a string then it is case sensitive.) The following example shows how to generate a few simple rules and review their definitions.

```
(c1) tellsimpafter(a+b,c);
(d1)                [+rule1, simplus]
(c2) stringp(first(%));
(d2)                      true
(c3) tellsimpafter(f(x),g);
(d3)                 [frule1, false]
(c4) symbolp(first(%));
(d4)                      true
(c5) disprule(frule1);
```

```
(d5)                    frule1 : f(x) -> g
(c6) disprule("+rule1");
(d6)                   +rule1 : b + a -> c
(c7) disprule("+rule1");
"+rule1" is not a user rule.
Returned to Macsyma Toplevel.
```

tellsimpafter rules can be removed using the **remrule** form. The syntax for **remrule** is **remrule**(*operator*, *rulename*), where *operator* is the name of the operator to which a simplifier rule was attatched, and *rulename* is either the name of the rule to be deleted or the keyword **all**. If **all** is specified, all rules associated with the operator *operator* are deleted. If the operator is a symbol then both *operator* and *rulename* must be symbols. If the operator is not a symbol, *operator* and *rulename* must be specified as strings. Continuing the previous example, we remove the rules.

```
(c8) rules;
(d8)                     [+rule1, frule1]
(c9) remrule(f,frule1);
(d9)                            f
(c10) remrule("+","+rule1");
(d10)                           +
(c11) rules;
(d11)                           []
```

The command **remove**(*rulename*, rule) can also be used to remove rule definitions.

Note: **tellsimpafter** rules can be compiled using the **compile_rule** form.

14.5.4.1 tellsimpafter Summary

Note: all information in this section also applies to **tellsimp**.

- **Defining rules: tellsimpafter**(*pattern*, *repl*, *cond*) replaces *pattern* with *repl* when *cond* (if given) evaluates to **true**. The scope of the match depends on the **matchdeclare** properties of the pattern variables.
- **Reviewing rules: disprule**(*rule*) prints the definition of the rule given by *rule*, which is either an upper-case string or a symbol. The system infolist **rules** contains the names of all rules defined using **defmatch**, **defrule**, and **tellsimpafter**.
- **Removing rules: remrule**(*operator*, *rule*) removes the rule *rule* that is associated with the operator *operator*. Both *rule* and *operator* are either upper-case strings or symbols. **remrule**(*'operator*, all) removes all **tellsimpafter** rules associated with the operator *operator*.
- **Other methods of removing rules: remove**(*rule*, rule) removes the rule named *rule*, which is either a symbol or an upper-case string. **kill**(*rule*) removes all information associated with *rule*, which can be either a string or a symbol. **kill**(rules) removes all of the rule definitions contained in the system infolist **rules**.

14.5.4.2 Differences between tellsimp and tellsimpafter

As mentioned in the previous section, rules defined by **tellsimp** and **tellsimpafter** are invoked at different times in the simplification process. **tellsimp** rules are invoked prior to the Macsyma simplification stage, and should only be used to override default simplifications.

The Macsyma simplifier puts the argument lists of commutative operators into a canonical form. This is why, for example, $x+y+z$ and $x+z+y$ both display as $z+y+x$: the canonical representation of the expressions *"+"(x,y,z)* and *"+"(x,z,y)* is *"+"(z,y,x)*. The commutativity of "+" permits this kind of rearrangement. However, there is no guarantee that an unsimplified expression will be in canonical form, and as a consequence pattern matching will be more difficult. For this reason, a call to **tellsimp** that attempts to make a substitution for a sum or product generates a warning.

```
(c1) tellsimp(x+y,z);
Warning: Putting rules on "+" or "*" is inefficient, and may not work.
(d1)                 [+rule1, simplus]
(c2) tellsimp(x*y,z);
Warning: Putting rules on "+" or "*" is inefficient, and may not work.
(d2)                [*rule1, simptimes]
```

Since **tellsimpafter** rules are invoked on simplified expressions, this particular problem does not arise. However, since the general pattern matcher does not employ backtracking, **tellsimpafter** rules on "+" and "*" will not in general simplify subexpressions of sums or products. (Rules on "*" should be implemented with the **let** pattern matcher, which is discussed later in this chapter.) For example, some of the following expressions which contain $x+y$ do not simplify.

```
(c1) tellsimpafter(x+y,z);
(d1)                 [+rule1, simplus]
(c2) x+y;
(d2)                         z
(c3) a*(x+y);
(d3)                        a z
(c4) a*x+a*y;
(d4)                     a y + a x
(c5) x+y+z;
(d5)                     z + y + x
```

14.5.4.3 Examples of tellsimp and tellsimpafter

In the first example, we declare **"&"** to be a postfix operator and define **tellsimpafter** rules on **"&"** that implement the factorial function.

```
(c1) postfix("&")$
(c2) [0&,1&,2&,3&,4&,n&];
(d2)           [0 &, 1 &, 2 &, 3 &, 4 &, N &]
(c3) posintp(n):=is(integerp(n) and n>0)$
(c4) matchdeclare(pint,posintp)$
(c5) tellsimpafter(0&,1)$
```

```
(c6) tellsimpafter(pint&,pint * (pint-1)&)$
(c7) [0&,1&,2&,3&,4&,n&];
(d7)               [1, 1, 2, 6, 24, N &]
```

Next, we implement a simplification rule that implements the identity $x\& = gamma(x + 1)$ when x is not a positive integer.

```
(c8) not_posintp(n):=not(posintp(n))$
(c9) matchdeclare(not_pint,not_posintp)$
(c10) tellsimpafter(not_pint&,gamma(not_pint+1))$
(c11) n&;
(d11)                   gamma(n + 1)
(c12) (-1)&;
gamma(0) is undefined
Returned to Macsyma Toplevel.
```

Note that the last **tellsimpafter** rule generates an error when the resulting **gamma** expression contains an illegal argument. In order to prevent errors of this sort, we remove that rule and redefine it so that the rule does not fire when the argument is a non-positive integer. We do this by adding the condition that the rule fire only when the pattern variable is *not* an integer. The **matchdeclare** property on the pattern variable **not_pint** prevents it from matching positive integers, so this condition prevents the rule from firing when the pattern variable is a non-positive integer. (An alternative method to prevent this kind of error would be to redefine the predicate **not_posintp**.)

```
(c13) rules;
(d13)              [&rule1, &rule2, &rule3]
(c14) remrule("&","&rule3")$
(c15) tellsimpafter(not_pint&,gamma(not_pint+1),not(integerp(not_pint)))$
```

We now test the new rule.

```
(c16) [(-1)&,n&];
(d16)              [(- 1) &, gamma(n + 1)]
(c17) rules;
(d17)              [&rule1, &rule2, &rule4]
```

In the next example, we use **tellsimp** to override the default simplification of 0ˆ0, which signals an error. We will install a rule which defines the replacement $0\hat{}0 = 1$. Note that the first attempt to define the rule fails since the simplifier signals an error before the rule overriding the simplification is installed. In order to define the rule, it is necessary to turn the simplifier off by (locally) binding the option variable *simp* to **false**. It is preferable to bind *simp* as a block variable rather than as an *evflag* as this method guarantees that the binding of *simp* holds for the duration of the **tellsimp** command. The *simp* should never be bound to **false** at toplevel.

```
(c1) 0^0;
0^0 has been generated
Returned to Macsyma Toplevel.
(c2) tellsimp(0^0,1);
```

```
0^0 has been generated
Returned to Macsyma Toplevel.
(c3) block([simp:false],tellsimp(0^0,1));
(d3)                 [^rule2, simpexpt]
(c4) 0^0;
(d4)                         1
```

Note that the purpose of the **block** statement in (c3) is to localize the binding of **simp** to **false**. Rules generated by **tellsimp**, **tellsimpafter**, **defmatch**, and **defrule** cannot be localized.

The next example uses **tellsimpafter** to extend the integrator to handle a class of integrals related to the incomplete gamma function. Note that Macsyma can do the following improper integrals.

```
(c1) (assume(a>0,b>0),declare(b,integer));
(d1)                       done
(c2) integrate(t^(b-1)*exp(-t),t,0,inf);
(d2)                     (B - 1)!
(c3) integrate(t^(b-1)*exp(-t),t,a,inf);
(d3)                   gamma(b, a)
```

However, it cannot do the following definite integral, even though the integral can be deduced from the integrals in (c2) and (c3).

```
(c4) integrate(t^(b-1)*exp(-t),t,0,a);
                      a
                     /
                     [   b - 1   - t
(d4)                 I  t      %e    dt
                     ]
                     /
                      0
```

We will use **tellsimpafter** to set up a rule which computes integrals of the type (c4) as the difference of the integrals (c2) and (c3).

We first decide what level of generality is desired. It is not unreasonable to demand that there be a semantic match on the variable of integration. Furthermore, we insist that the upper limit of integration, a, be positive. We will do limited testing on the integrand, leaving most of the work to **integrate**, and will instead make the substitution if the result is free of **integrate** noun forms.

```
(c5) positivep(limit):=is(asksign(limit)='pos)$
(c6) integ(expr,var,limit):=integrate(expr,var,0,inf) -
                            integrate(expr,var,limit,inf)$
(c7) matchdeclare(t, symbolp, [expt1, expt2], true, uvar, positivep)$
(c8) block
     ([simp:false],
       tellsimpafter('integrate(t^(expt1)*%e^(expt2),t,0,uvar),
        result,
        freeof(nounify('integrate), result:integ(t^expt1*exp(expt2),t,uvar))));
```

```
(d8)            [integraterule1, simpinteg]
```

The **tellsimpafter** rule binds the pattern variables and passes them to the function **integ**, which attempts to compute the modified definite integrals. The result returned by **integ** is bound to **result**, which is then checked to see if it is independent of **integrate** noun forms. If so, **result** is substituted for the original integral.

Note: It is necessary to bind **simp** to **false** to prevent the **outative** property of **integrate** from pulling the factor $\%e\hat{}(expt2)$ out of the integral.

Note: In this case, it is necessary to refer to the exponential function as $\%e\hat{}(expt2)$ rather than $exp(expt2)$ since the former is the simplified form of the latter, and will be the form seen by the rule when it is invoked on simplified Macsyma expressions.

Finally, we test the rule.

```
(c9) integrate(t^(b-1)*exp(-t),t,0,a);
(d9)                (b - 1)! - gamma(b, a)
(c10) integrate(t^2*exp(-t),t,0,5);
                                       - 5
(d10)                    2 - 37 %e
```

The efficiency of this scheme can be improved by writing **integ** differently. For example, **integ** could incorporate the results of the integrals in its function definition rather than compute the integrals each time the rule is invoked. However, this would involve substantially more effort since very little testing is done on the integrand in this implementation of the rule. Note: In this example, it would be sufficient to implement the integration rule directly with the following form:

```
block
([simp:false],
 tellsimpafter('integrate(t^(expt1)*%e^(expt2),t,0,uvar),
               integrate(t^(expt1)*%e^(expt2),t,0,inf) -
                integrate(t^(expt1)*%e^(expt2),t,uvar,inf)));
```

This can be done this way because the integral is simple enough that the replacement rule will never contain an **integrate** noun form. However, if the class of integrals were enlarged and the replacement were to yield an **integrate** noun form then the **tellsimpafter** rule would also fire on that form, resulting in an infinite loop. In this example, the current method, namely, triggering the rule if and only if the integration succeeds, does- not result in an infinite loop.

The final example uses **tellsimpafter** to rewrite the log of a product as a sum of logs. That is, the goal is to implement the rule

$$\log\left(\prod_{a2=a3}^{a4} a1\right) \rightarrow \sum_{a2=a3}^{a4} \log a1$$

It is necessary to bind **simp** to **false** to prevent simplification of the **prod** and **sum** forms prior to the definition of the rule.

```
(c1) matchdeclare([a1,a2,a3,a4],true)$
(c2) block([simp:false],
          tellsimpafter('log('prod(a1,a2,a3,a4)),apply('sum,[log(a1),a2,a3,a4])))$
```

```
(c3) log(product(a[i],i,1,inf));
                         inf
                         ====
                         \
(d3)                      >    log(a )
                         /          i
                         ====
                         i = 1
```

Note: It is necessary to **apply** the **sum** form in order to force the evaluation of the summation index prior to the handling of the **sum** form. Failure to **apply** the **sum** form results in an incorrect rule because the **sum** form will simplify to $log(a1) * (a4 - a3 + 1)$ before the pattern variable definitions are inserted by evaluation.

14.5.4.4 Additional Information on tellsimp and tellsimpafter

Since **tellsimp** and **tellsimpafter** generate rules which are invoked automatically by the Macsyma simplifier, some additional information might be useful.

The simplifier is called on all Macsyma expressions, including the arguments of special forms. (Arguments are usually simplified before operators.) The quote operator, which prevents evaluation, does not prevent simplification. The Macsyma simplifier usually works "inside-out" or "bottom-up" when simplifying expressions. Furthermore, **tellsimp** and **tellsimpafter** rules are usually applied in the order in which they appear on the system **rules** infolist. The application mechanism for these rules is similar to the **applyb1** mechanism for the application of **defrule** rules.

Finally, it should be noted that adding **tellsimp** or **tellsimpafter** rules can greatly increase overall execution time. Compiling the rules (using the **compile_rule** form) substantially improves performance. The user should take into consideration the disadvantages as well as the obvious conveniences afforded by this mechanism.

14.5.5 Defining Taylor Expansions of Unknown Functions

The **deftaylor** facility permits the user to define the Taylor expansion of an unknown univariate function about 0. It does not allow expansions about other points. In the following example, we first compute the expansion of an unknown function $f(x)$, then define an expansion for $f(x)$ and re-compute the expansion.

```
(c1) taylor(f(x),x,0,2);
                                           |
                                    2      |
                                   d       |      2
                                  (--- (f(x))|     ) x
                                     2     |
                          |        dx      |
                d         |                |X = 0
(d1)/t/ f(0) + (-- (f(x))|     ) x + ---------------------
                dx        |                     2
                          |x = 0
                                                   + . . .
(c2) deftaylor(f(x),sum(a[n]*x^n,n,0,inf));
```

```
(d2)                        [f]
(c3) taylor(f(x),x,0,3);
                             2      3
(d3)/t/      a  + a  x + a  x  + a  x  + . . .
              0    1      2       3
(c4) taylor(f(x),x,0,5);
                        2       3       4       5
(d4)/t/ a  + a  x + a  x  + a  x  + a  x  + a  x  + . . .
         0    1      2       3       4       5
```

Note that the result (d1) is just the Taylor series of the unknown function $f(x)$ about $x = 0$. The expansion for the unknown function $f(x)$ (assumed to be about $x = 0$) is defined in (c2), and is used when **taylor** is invoked in (c3) and (c4).

Note that **taylor** never returns a noun form. When given an unknown function in (d1), **taylor** returns a general expansion rather than, say, $'taylor(f(x), x, 0, 2)$. This explains the need for **deftaylor**. A rule defining the Taylor expansion of an unknown function could not be installed by **tellsimp** or **tellsimpafter** since **taylor** never generates a noun form to trigger the rule.

14.5.6 Translating and Compiling Rules

This section discusses the role of the LISP compiler when loading a translated or compiled file that contains rule definitions. When a file of this type is loaded, a message to the effect that rules are being compiled is displayed. We provide a brief discussion of why this happens, its advantages and disadvantages, and what options the user has in this situation.

A compiled rule generally runs *much* faster than a translated or interpreted rule. However, translating and compiling a rule is not straightforward (as far as Macsyma is concerned) since it requires that the **matchdeclare** properties be present when the rule is generated. As was discussed at the beginning of this section, when the general pattern matcher defines a rule, it generates a function based on the pattern, the pattern variables, and the replacement. Since the Macsyma translator does not evaluate Macsyma forms, existing **matchdeclare** forms are not evaluated at translate time and consequently rules cannot be defined at translate time. Therefore, the translator parses a rule definition but does not evaluate the form. Evaluation takes place as the file is loaded. The following sequence of events occurs at load time:

- LISP forms are evaluated when loaded into Macsyma. In particular, **matchdeclare** and rule-writing forms are evaluated.
- If a rule-writing form is evaluated, a LISP function is generated.
- If appropriate, the resulting LISP function is compiled automatically. (By default, such functions are compiled.)

Although some of the previous versions of Macsyma did not include LISP compilers, all of the current ones do. However, in some versions the compiler is quite slow. Compilation of rules can be skipped by setting the option variable *compile_rules_in_tr_files* (*default:***true**) to **false**. This will reduce the time spent loading the file, but will increase execution time since the interpreted rules will run slower than the compiled rules.

Note: Interpreted rules can be compiled (in a manner analagous to compiling interpreted functions) using the **compile_rule** form. This speeds up rule execution significantly, especially for rules defined by **tellsimp** and **tellsimpafter**. See the *Macsyma Reference Manual* for more information.

14.6 The Rational Function Pattern Matcher

As mentioned earlier, Macsyma contains two pattern matching mechanisms. The first mechanism, the general pattern matcher, was described in the previous section. This section describes the rational function pattern matcher. It is designed to make substitutions for products in rational expressions. This alternative pattern matcher is recursive and permits the user to define substitution rules and control their application conveniently.

In order to use the rational function pattern matcher, the user first defines pattern variables with **matchdeclare** (this step can be omitted if the user wants to consider literal matches only). Next, the substitution rules are defined using the **let** form. The rules are applied by the **letsimp** form in the order in which they were defined. Since **letsimp** is recursive, it performs the indicated substitutions until the result no longer changes.

14.6.1 Defining Substitution Rules: let

The **let** form is used to define substitution rules. In its simplest form, **let** permits the user to define the pattern and its replacement. If desired, the user can also provide the name of a predicate and associated arguments that depend on any atoms or arguments of any kernels (that is, a functional expression of the form $cos(x)$ or $n!$) as arguments to **let**. When called in this fashion, **let** first matches atoms and functional arguments, then evaluates the predicate arguments according to the pattern variable bindings, and finally evaluates the predicate with the specified arguments. The rule is applied if and only if the predicate evaluates to **true**. The syntax for **let** (as well as for associated operations) is summarized below.

Finally, it is possible to isolate **let** rules so that they can be applied selectively. This is done by associating the rule with a *rule package*. By default, a **let** rule goes into the package **default_let_rule_package**. The current **let** rule package is specified by the value of the system variable *current_let_rule_package*. All **let** commands (**letrules**, **letsimp**, **let**, and **remlet**) use the package specified by the value of **current_let_rule_package** unless the package name is specified as an argument to the command. However, the only way to create a *new* **let** rule package is to call **let** and specify the name of the new package. A rule can be installed in a different rule package by including as a final optional argument to **let** the name of the package. When specifying a package name this way, it is necessary to enclose all of the preceding **let** arguments in square brackets to tell **let** that the final argument is the name of a rule package rather than the name of a predicate or a predicate argument.

If needed, **let** rules can be removed with the **remlet** command. Either **remlet**() or **remlet**(all) will remove all **let** rules in the current **let** rule package. **remlet**(*pred*) will remove the substitution rule defined for the expression *pred*. The name of a **let** rule package can be supplied as a second argument to **remlet**, which will remove the specified **let** rule in the named **let** rule package. The command **remlet(all,***name***)** will remove all of the rules in the package specified by *name* and will also delete the specified **let** package.

The names of existing **let** rule packages are contained in the system variable **let_rule_packages**.

Finally, we mention that the rules contained in a **let** rule package can be displayed by the command **letrules**, which takes either zero or one arguments. If **letrules** is called with no arguments, it displays the contents of the default rule package (specified by the system variable **current_let_rule_package**). If called with the name of an existing **let** rule package as its argument, it displays the contents of that **let** rule package. Calling **letrules** with the name of a non-existent package results in an error.

14.6.1.1 let Syntax

Defining **let** rules:

- **let**(*prod*, *subst*) defines a substitution rule, *prod* → *subst*, in the rule package specified by **current_let_rule_package**.
- **let**(*prod*, *subst*, *pred*, *arg1, ..., argN*) defines a substitution rule, *prod* → *subst*, in the rule package specified by **current_let_rule_package**, which will be executed if and only if the form *pred*(*arg1*, *arg2, ..., argn*) evaluates to **true**, where the expressions *arg1, ..., argn* contain the pattern variables used in *prod*.
- **let**([*prod*, *subst*], *name*) defines a substitution rule, *prod* → *subst*, in the rule package specified by *name*. If the specified rule package doesn't already exist, it is created.
- **let**([*prod*, *subst*, *pred*, *arg1*, ..., *argN*], *name*) defines a substitution rule, *prod* → *subst*, in the rule package specified by *name*, which will be executed if and only if the form *pred*(*arg1*, *arg2, ..., argN*) evaluates to **true**.

Displaying **let** rules:

- **letrules()** displays the **let** rules in the rule package specified by **current_let_rule_package**.
- **letrules**(*name*) displays the **let** rules in the rule package specified by *name*. An error results if *name* is not the name of an existing **let** rule package.

Removing **let** rules:

- **kill**(*name*) removes all substitution rules in the rule package specified by *name* but doesn't delete the specified rule package.
- **remlet**(*prod*) removes the substitution rule for the expression *prod* in the rule package specified by **current_let_rule_package**.
- **remlet(all)** removes all substitution rules in the rule package specified by **current_let_rule_package**.
- **remlet**(*prod*,*name*) removes the substitution rule for the expression *prod* in the rule package specified by *name*.
- **remlet(all,** *name*) removes all substitution rules in the rule package specified by a*name*. Furthermore, the rule package specified by *name* is deleted.

Note: The package **default_let_rule_package** cannot be deleted.

For example, the following example installs a substitution rule in the default **let** rule package, which is called **default_let_rule_package**. The system variable **let_rule_packages** is evaluated to show that no new package is created by this call to **let**.

```
(c1) let_rule_packages;
(d1)            [default_let_rule_package]
(c2) let(x*y^2,z);
                             2
(d2)                      x y  --> z
(c3) let_rule_packages;
(d3)            [default_let_rule_package]
```

The next example installs a **let** substitution rule into a **let** rule package called **new_let_rule_package**.

```
(c4) let([x*y^3,z],new_let_rule_package);
```

```
                             3
(d4)                      x y  --> z
(c5) let_rule_packages;
(d5)  [default_let_rule_package, new_let_rule_package]
(c6) letrules();
                             2
                          x y  --> z
(d6)                         done


(c7) letrules(default_let_rule_package);
                             2
                          x y  --> z
(d7)                         done
(c8) letrules(new_let_rule_package);
                             3
                          x y  --> z
(d8)                         done
```

Further examples of **let** are provided in the next section.

Note: **let** rules cannot be compiled.

14.6.2 Applying let Rules: letsimp

Substitution rules defined by **let** are applied by **letsimp**. The **letsimp** form takes one required argument, which is the expression to "simplify" with the substitution rules, and any number of additional arguments, which are the names of **let** rule packages. Only the first argument to **letsimp** is evaluated.

If no optional arguments are provided, the package specified by **current_let_rule_package** is used. If multiple package names are provided, they are applied left to right. That is, **letsimp(***expr*, *name1***,** *name2***)** is equivalent to **letsimp(letsimp(***expr*, *name1***),** *name2***)**.

When applying **let** rules to a quotient, **letsimp** by default simplifies the numerator and denominator independently and returns the result. Setting the system option variable **letrat** (*default*: **false**) to **true** causes **letsimp** to first simplify the numerator and denominator independently, then simplify the resulting (simplified) quotient.

The rules in the specified **let** rule package are applied repeatedly by **letsimp** until the expression is unchanged. Care should therefore be taken to write rules that will eventually terminate. Furthermore, it should be noted that the **let** rule that is defined last will be the first to be executed. In fact, if the user defines rules a, b, and c, in that order, then **letsimp**s an expression using these rules, the result will be obtained by first applying c until the result stops changing, then b, then a, and then repeating the rule sequence cba until no further changes are made. Take care to avoid defining circular **let** substitutions.

14.6.2.1 letsimp Summary

- **letsimp(***expr***)** recursively applies the rules in the **let** rule package specified by the value of **current_let_rule_package** to *expr*.
- **letsimp(***expr*, *'name1, ..., 'nameN***)** applies the rules in the **let** rule packages *name1, ..., nameN* to *expr*.

Calling **letsimp** with the name of a non-existent **let** rule package results in an error.

14.6.3 Examples of let Rules

This section contains some examples which demonstrate how to use the **let** pattern matcher.

14.6.3.1 Example 1

The first example shows how **letsimp** treats powers.

```
(c1) let(x^2,y);
                               2
(d1)                          x   --> y
(c2) [1/x^3,1/x^2,1/x,1,x,x^2,x^3];
                   1   1   1         2   3
(d2)              [--, --, -, 1, x, x , x ]
                    3   2  x
                   x   x
(c3) letsimp(%);
                    1   1  1
(d3)              [---, -, -, 1, x, y, x y]
                   x y  y  x
```

The substitution rule is defined in (c1). Since no **matchdeclare** property is associated with x, this is a literal match. Some test expressions are generated in (c2), and **letsimp** is applied to these expressions in (c3). Some of the terms with negative exponents simplify since **letsimp** simplifies the numerator and denominator separately.

```
(c4) letrules();
                               2
                              x   --> y
(d4)                            done
(c5) remlet(x^2);
(d5)                            done
(c6) letrules();
(d6)                            done
```

The call to **letrules** in (c4) displays the known substitution rules. The **remlet** call in (c5) removes the substitution defined for $x\hat{}2$, as verified by (c6).

14.6.3.2 Example 2

This example shows how **letrat** controls the behavior of **letsimp**. We start by defining a rule to perform a rational substitution for $1/x\hat{}2$.

```
(c1) let(x^-2,y);
                            1
(d1)                        -- --> y
                             2
                            x



(c2) expr:[1/x^3,1/x^2,1/x,1,x,x^2,x^3];
                   1   1   1          2   3
(d2)              [--, --, -, 1, x, x , x ]
                    3   2  x
                   x   x



(c3) letsimp(expr);
                   1   1   1          2   3
(d3)              [--, --, -, 1, x, x , x ]
                    3   2  x
                   x   x



(c4) letsimp(expr),letrat:true;
                    y     1         2   3
(d4)               [-, y, -, 1, x, x , x ]
                    x     x
```

Since $1/x\hat{}2$ does not occur in either the numerator or denominator of any of these expressions, **letsimp** will not find it unless it is permitted to look at the entire expression. This explains why the result (d3) is unchanged. In (c4), binding **letrat** to **true** enables **letsimp** to consider the quotients after it attempts to simplify the numerators and denominators, and consequently it simplifies some of the expressions.

14.6.3.3 Example 3

This example demonstrates some of the limitations of **letsimp** in simplifying functions.

```
(c1) let(x^2,y);
                             2
(d1)                        x  --> y
(c2) letsimp([x^2,f(x^2)]);
                                 2
(d2)                       [y, f(x )]
```

Note that no substitution into the argument of f was made. The **let** pattern matcher is not designed to substitute into functional forms. Substitutions into functional forms should be done using the general pattern matcher.

The next example shows that rational substitutions of functional forms can be made, as long as they aren't made to functional arguments. (The exceptions, of course, are the functions "*" and "^", which the **let** pattern matcher is designed to handle.)

```
(c3) let(f(x),g);
(d3)                      f(x) --> g
(c4) letsimp([f(x)^2,a*f(x),f(f(x))]);
                        2
(d4)                  [g , a g, f(f(x))]
```

14.6.3.4 Example 4

This example demonstrates that the **let** pattern matcher, like the general pattern matcher, is unable to make general substitutions for sums.

```
(c1) let(x+y,z);
(d1)                     y + x --> z
(c2) letsimp([x+y,(x+y)^2,(x+y)^n,a*x+a*y]);
                2            2         n
(d2)   [y + x, y  + 2 x y + x , (y + x) , a y + a x]
```

Note that the substitution failed in each case. Furthermore, the second element in the list was expanded; this happened when **letsimp** converted its argument to CRE form.

14.6.3.5 Example 5

This example illustrates the recursive nature of **letsimp**. The **matchdeclare** form is used to generate a semantic match, and a trivial predicate (**truep**) is used to trace the action of **letsimp**.

```
(c1) matchdeclare([x,y],symbolp)$
(c2) truep(foo,bar):=(print(``truep args: '',[foo,bar]),true)$
(c3) let(x*y^2,z,truep,x,y);
                   2
(d3)            x y  --> z where truep(x, y)
(c4) letsimp(u*v^2+a*b^2);
truep args:  [u, v]
truep args:  [a, b]
(d4)                         2 z
```

The pattern variables are allowed to match any symbols using the **matchdeclare** form in (c1). A trivial predicate (trivial in that it always returns **true**) is defined in (c2). (The purpose of **truep** is to show the order in which **letsimp** performs the substitutions. Since it returns **true** for any pair of arguments, the substitution will be performed whenever the pattern variables x and y match.) Note that the arguments of **truep** depend on the pattern variables which are used in the first argument to the **let** form. In (c4), a sum is simplified. The information printed by **truep** shows that in this case the individual terms are simplified, giving the expected result.

```
(c5) letsimp(u*v^2*a*b^2);
truep args:  [u, b]
truep args:  [z, v]
(d5)                          a z
```

In (c5), a complicated product is simplified. The first match is made on u and b, and the indicated substitution is made. The next call to **letsimp** matches the substituted variable z and the original variable v, yielding $a * z$. (The reader might well wonder why the result $z\hat{}2$ was not obtained instead. This is due to the canonical re-ordering of the argument list of "*" prior to calls to **letsimp**.)

14.6.3.6 Example 6

This example is similar to the previous example, except that we define the pattern variables to exclude the substituted variable z from matching the pattern variables.

```
(c1) not_z_p(symbol):=is(symbolp(symbol) and symbol # 'z)$
(c2) matchdeclare([x,y],not_z_p)$
(c3) let(x*y^2,z)$
(c4) letsimp(u*v^2*a*b^2);
                                2
(d4)                           z
```

Note that by defining the pattern variables in such a way as to exclude matching the symbol z, with the predicate **not_z_p**, we obtain a different result.

14.6.3.7 Example 7

This example is similar to the previous two, except that we define the pattern variables to match any symbols and exclude the substituted variable z from matching the pattern variables by means of optional arguments of the **let** form (the previous example excluded z from matching the pattern variable through the **matchdeclare** predicate). We use the Macsyma **trace** command to see how **letsimp** scans its argument:

```
(c1) matchdeclare([x,y],symbolp)$
(c2) not_z_p(symbol1,symbol2):=is(symbol1 # 'z and symbol2 # 'z)$
(c3) let(x*y^2,z,not_z_p,x,y)$
(c4) trace(not_z_p)$
(c5) letsimp(u*v^2*a*b^2);
1 Enter not_z_p [u, b]
1 Exit  not_z_p true
1 Enter not_z_p [z, v]
1 Exit  not_z_p false
1 Enter not_z_p [a, v]
1 Exit  not_z_p true
                                2
(d5)                           z
```

Note that **letsimp** matched one of the pattern variables to z the second time around, but the predicate **not_z_p** prevented **letsimp** from performing the indicated substitution. **letsimp** then successfully matched the variables a and v and made a substitution, resulting in $z\hat{}2$.

It is more efficient to obtain this behavior with **matchdeclare**, as in example six, rather than as a side condition to **let**, as in this example.

14.6.3.8 Example 8

This example uses the **let** pattern matcher to implement part of the trigonometric identity $\sin^2 x + \cos^2 x = 1$ for any x. Since it was shown in example four of this section that the **let** pattern matcher cannot make substitutions for sums, we must decide to substitute either $sin(x)\hat{}2 = 1 - cos(x)\hat{}2$ or $cos(x)\hat{}2 = 1 - sin(x)\hat{}2$.

```
(c1) matchdeclare(arg,true)$
(c2) let(sin(arg)^2,1-cos(arg)^2);
                    2                  2
(d2)           sin (arg) --> 1 - cos (arg)
(c3) [a*cos(x)^2+b*sin(x)^2,sin(1)^2,cos(1)^2,sin(f(x))^4];
          2           2        2        2        4
(d3) [b sin (x) + a cos (x), sin (1), cos (1), sin (f(x))]
(c4) letsimp(%);
             2           2                 2        2
(d4) [- b cos (x) + a cos (x) + b, 1 - cos (1), cos (1),
                                   4               2
                                cos (f(x)) - 2 cos (f(x)) + 1]
```

Note that attempting to implement the two substitutions $sin(x)^2 = 1 - cos(x)^2$ and $cos(x)^2 = 1 - sin(x)^2$ can lead to circular substitutions. The Macsyma **trigsimp** function, which is not implemented with pattern matching techniques, actually tries both substitutions and uses the one that produces the "simplest" form.

14.6.3.9 Example 9

This example uses the **let** pattern matcher to simplify derivative consequences of an equation. For example, given the relation $'diff(x,t,n) = f(x,t)$ for some n, the higher-order derivatives of x can be expressed in terms of $f(x,t)$ and its derivatives.

The following example implements the rule to simplify subsequent derivatives of x given that $'diff(x,t,2) = a * x * sin(t)$.

```
(c1) derivabbrev:true$
(c2) depends(x,t)$
(c3) bigger_integerp(reference,test):=is(integerp(test) and test >= reference)$
(c4) matchdeclare(n,bigger_integerp(2))$
(c5) let('diff(x,t,n),diff(a*x*sin(t),t,n-2));
              n
             d x
(d5)         --- --> diff(a sin(t) x, t, n - 2)
               n
             d t
```

The command (c5) defines a simplification rule for derivatives of degree greater than 1. The **matchdeclare** property on n guarantees that the rule is applied to derivatives of degree 2 or greater. We proceed to test the rule.

```
(c6) [x,'diff(x,t),'diff(x,t,2),'diff(x,t,3)^2,b*'diff(x,t,2)+c*'diff(x,t,3)];
                                2
(d6)    [x, x , x   , (x     ) , c x      + b x   ]
             t   t t    t t t       t t t      t t


(c7) letsimp(%);
                          2    2        2
(d7) [x, x , a sin(t) x, a  sin (t) (x )
          t                          t
      2                      2    2     2
 + 2 a  cos(t) sin(t) x x  + a  cos (t) x ,
                          t
a c sin(t) x  + a b sin(t) x + a c cos(t) x]
            t
```

A few comments are in order. First, we require literal matches for t and x. (The requirement on x should be clear. The requirement on t is mainly for the sake of convenience: x must have a functional dependency on the variable of differentiation.) Second, the predicate **bigger_integerp**, strictly speaking, should test to guarantee that its first argument is a positive integer. However, since the predicate is called every time a match is attempted, this would slow the process down.

14.7 Debugging Pattern Matching Routines

The information in this section is applicable to both Macsyma pattern matchers.

The following problems typically arise when using the Macsyma pattern matchers:

- A rule doesn't match an expression when the user thinks it should.
- A rule matches an expression when the user thinks it shouldn't.
- A recursive rule fails to exit, and either goes into an infinite loop or overflows one of the internal spaces.

14.7.1 Example: Failure to Match

Probably the most common problem is that the pattern matcher fails to match a subexpression. This can occur for the following reasons:

- The pattern is a complicated expression which fails to match on the first attempt, and no further search is performed since the pattern matcher does not implement full backtracking. For example, the substitution $\sin^2 x + \cos^2 x \to 1$ will often fail to match. The solution is usually to write less ambitious patterns, *i.e.*, instead of the previously mentioned substitution, try $\sin^2 x \to 1 - \cos^2 x$.
- The **matchdeclare** forms or predicates incorrectly define the desired semantic pattern variables. In many cases, this can be diagnosed by tracing the predicates.

- The pattern contains a verb form that should instead be a noun form, or vice-versa. (It is usually correct to use the noun form.) The easiest way to debug a problem of this sort is to review the rule definition using **disprule** with the system option variable *noundisp* (*default*:**false**) set to **true**. The effect of *noundisp* is described in detail later in this chapter. (An alternative method of inspecting a rule to see what form is incorporated is to look at the LISP representation of the expression returned by **disprule**. We won't say much more about this technique since it is presumes a working knowledge of LISP.)

Since the first two cases have been discussed in detail in previous sections, we provide an example of the last one only. In this example, we define **tellsimpafter** rules to transform some derivatives.

```
(c1) tellsimpafter(diff(f(x),x),g(x));
(d1)                [diffrule1, false]
(c2) diff(f(x),x);
                         d
(d2)                     -- (f(x))
                         dx
(c3) disprule(diffrule1);
(d3)         diffrule1 : diff(f(x), x) -> g(x)
(c4) tellsimpafter('diff(ff(x),x),gg(x));
(d4)            [derivativerule1, simpderiv]
(c5) diff(ff(x),x);
(d5)                      gg(x)
(c6) disprule(derivativerule1);
                             d
(d6)       derivativerule1 : -- (ff(x)) -> gg(x)
                             dx
```

Note that the expression in (d2) did not transform to $g(x)$, as one might have expected. Note also that the names of the rules are different—the rule generated in (c1) is called **diffrule1**, while the rule generated in (c2) is called **derivativerule1**. The reason for these differences is that the rule generated in (c1) is on the verb form of the derivative, while the rule generated in (c4) is on the noun form of the derivative. Since the rule **diffrule1** is applied *after* the simplification of the **diff** form in (c2), which results in a noun form, the result (c2) does not simplify with the rule **diffrule1**. (**diffrule1** will, however, simplify the result of, say, $funmake('diff, [f(x), x)]$.)

Even experienced users can be unsure of when it is appropriate to quote an operator or function in a rule definition. For some operators, it does not matter if it is quoted or not—these operators always simplify to a unique form, including the time when the rule is written. These operators are predefined and have both **alias** and **noun** properties (*i.e.*, **log**, **sin**, **cos**), and always return **noun** forms (unless the user goes out of his way to create a verb form, in which case he deserves to lose). Any user-defined function or built-in function that has a **noun** property but no corresponding **alias** property (*i.e.*, **diff**, **limit**, **integrate**, **matrix**) has both noun and verb forms, and care must be taken to see that the correct form is incorporated in the rule definition.

The option variable *noundisp* (*default*:**false**) controls the printing of noun forms. When *noundisp* is set to **true**, the display of an operator is changed if the operator appears in an "unusual" form. Consider the following example.

```
(c1) noundisp:true$
(c2) [f(x),'f(x)];
```

```
(d2)                    [f(x), 'f(x)]
(c3) [log(x),'log(x),funmake(verbify('log),[x])];
(d3)              [log(x), log(x), 'log(x)]
(c4) properties(log);
(d4) [database info, kind(log, increasing), alias, noun,
gradef, built-in simplifications, user-defined simplifications,
system function]
```

```
(c5) [diff(f(x),x),'diff(f(x),x),funmake('diff,[f(x),x])];
             d          d
(d5)        [-- (f(x)), -- (f(x)), diff(f(x), x)]
             dx         dx
(c6) properties(diff);
(d6) [system function, noun, built-in simplifications,
                                user-defined simplifications]
```

Both the noun and verb forms of **f** can appear naturally, as demonstrated in (d2). The noun form is the "unusual" one in such cases. Since **log** has a **noun** property and a corresponding **alias** property, the noun form is the usual one. The first two forms in (d3) are noun forms, and the third one, which is a verb form generated by devious methods, displays with **"** to point out the fact that this operator is the "unusual" one. Since **diff** has a **noun** form but no corresponding **alias** property, both the noun and verb forms can appear naturally. In (d5), the noun form displays in two-dimensional format while the verb form displays in linear format.

If all else fails, the user can check which form of the operator is present by means of commands such as $freeof(nounify('diff), expr))$.

In the next example, we define some rules containing unknown functions and built-in functions with the **noun** and **alias** properties and look to see which forms can incorporate noun forms and which forms cannot.

```
(c1) tellsimpafter('f(x),g(x));
(d1)                 [frule1, false]
(c2) tellsimpafter(ff(x),gg(x));
(d2)                 [ffrule1, false]
(c3) tellsimpafter('log(x),y);
(d3)               [logrule1, simpln]
(c4) noundisp:true;
(d4)                       true
(c5) disprule(frule1);
(d5)              frule1 : 'f(x) -> g(x)
(c6) disprule(ffrule1);
(d6)             ffrule1 : ff(x) -> gg(x)
(c7) disprule(logrule1);
(d7)             logrule1 : log(x) -> y
```

Note that the rule **frule1** contains the noun form of f, while the rule **ffrule1** contains the verb form of ff. The **log** operator in (d7) is not quoted since it has simplified to the "usual" noun form.

```
(c8) properties(log);
(d8) [Database Info, kind(log, increasing), alias, noun,
gradef, Built-in Simplifications,
User-defined Simplifications, System Function]
(c9) properties(diff);
(d9) [System Function, noun, Built-in Simplifications,
                                User-defined Simplifications]
```

14.7.2 Example: Incorrect Matching

A user occasionally finds that a rule successfully matches on an expression, but that the result is not what he had intended. This is usually caused by an incorrectly specified **matchdeclare** declaration, although it can also be caused by the inappropriate selection of a rule application mechanism (when using the **defrule** interface). The following example shows how such problems can be isolated.

In this example, we attempt to use **defrule** to isolate the constant term in the denominator of a rational function. (Note: it is better to use **defmatch** for jobs like this.) We begin by defining the scope of the pattern variable and the rule, and then test it on a few expressions.

```
(c1) matchdeclare(const1,freeof(s))$
(c2) defrule(rat1,s/(s+const1),const1);
                              s
(d2)            rat1 : ---------- -> const1
                          s + const1
(c3) apply1(s/(s+4), rat1);
(d3)                            4
(c4) apply1(s/(s+1), rat1);
(d4)                            0
```

When testing the rule, (d3) yields the expected answer, but (d4) yields the "wrong" answer. Recalling that **defrule** rules are applied recursively, we suspect that the problem is due in part to the recursive application of the rule. To investigate this, we redefine the rule to return a *list* rather than an atom, since the list structure might indicate that the recursive application is to blame.

```
(c5) remove(rat1, rule);
(d5)                           done
(c6) defrule(rat1,s/(s+const1),[const1]);
                             S
(d6)           rat1 : ---------- -> [const1]
                         s + const1
(c7) apply1(s/(s+4), rat1);
(d7)                           [4]
(c8) apply1(s/(s+1), rat1);
(d8)                          [[0]]
```

Note that the result in (d8) contains a nested list, suggesting that the rule was applied twice. To determine what happens in this case, we **trace** the rule **rat1**.

```
(c9) trace(rat1);
(d9)                          [rat1]
```

```
(c10) apply1(s/(s+1), rat1);
                s
1 Enter rat1 [-----]
              s + 1
1 Exit  rat1 [1]
1 Enter rat1 [[1]]
1 Exit  rat1 false
1 Enter rat1 [1]
1 Exit  rat1 [0]
1 Enter rat1 [[0]]
1 Exit  rat1 false
1 Enter rat1 [0]
1 Exit  rat1 false
(d10)                      [[0]]
```

The result of the first application of **rat1** yields the expected answer, [1]. However, since **apply1** applies the rule recursively over the result and then its subexpressions, it is first applied to [1] and, failing to match there, to the subexpression 1. Since the pattern $s/(s + const1)$ matches 1 if $const1 = 0$, 1 transforms to 0, and consequently [1] transforms to [[0]].

The problem is therefore that the matchdeclare property on **const1** allows $const1 = 0$. Strengthening the **matchdeclare** declaration on **const1** by excluding $const1 = 0$ yields the expected behavior.

```
(c11) nonzeroandfreeof(var,exp):=is(exp#0 and freeof(var,exp))$
(c12) matchdeclare(const1,nonzeroandfreeof(s))$
(c13) defrule(rat1,s/(s+const1),const1);
                           s
(d13)           rat1 : ---------- -> const1
                       s + const1
(c14) apply1(s/(s+4), rat1);
(d14)                        4
(c15) apply1(s/(s+1), rat1);
(d15)                        1
```

Of course, the best fix is to use **defmatch** rather than **defrule**, since the recursive nature of **defrule** is inappropriate for this problem.

14.8 Patterns versus Functions

This section gives a few guidelines describing when it is appropriate to use pattern matching and when it is appropriate to implement the transformation using a function. Such guidelines are necessarily vague, and may be based on assumptions which are invalid in the context of specific problems. The suggestions presented in this section should therefore be interpreted with this all-purpose disclaimer in mind.

The current implementations of the pattern matchers contain the following deficiencies:

- A functional form in a pattern must contain a fixed number of arguments. There is no mechanism equivalent to the “rest args” feature supported in the function mechanism.

- The pattern matcher is relatively slow. Furthermore, adding a large number of **tellsimp** or **tellsimpafter** rules can slow down almost all computations.
- It is not possible to incorporate **mode_declare** information into rules.

Even with these deficiencies, the pattern matchers are useful and powerful tools, and the time required to learn how to use the pattern matchers is well spent. Furthermore, the pattern matcher is often a valuable aid in designing and implementing new packages. Although a pattern matching implementation of a package might be prohibitively slow for the final version, it is often useful to construct a prototype of a new package using the pattern matcher because of the ease of implementation and the great flexibility inherent in this method. Once the prototype works and the developer has a good model in mind, he can implement the final version of the package (using programmatic techniques) in Macsyma or LISP to achieve increased speed.

14.9 Complex Example of Pattern Matching

The final example uses **defmatch** and a tricky **lambda** substitution to extend Macsyma's inverse Laplace transform capability. In particular, we want to implement the inverse of the Laplace transform of $t * f(t)$.

```
(c1) assume(s>0)$
(c2) laplace(t*f(t),t,s);
                   d
(d2)             - -- (laplace(f(t), t, s))
                   ds
(c3) ilt(%,s,t);
                 d
(d3)       ilt(- -- (laplace(f(t), t, s)), s, t)
                 ds
```

(Aside: we assume $s > 0$ in (c1) to avoid sign questions in subsequent computations.)

Our goal is to implement an **ilt** simplifier that explicitly computes the inverse transform of expressions of the form (d3). (This type of substitution can be done more directly using **defrule** or **tellsimpafter**. However, we prefer the **defmatch** approach because the recursive application of rules generated by **defrule** or **tellsimpafter** is not needed for this problem. Furthermore, **tellsimpafter** rules affect the overall performance of Macsyma.)

We proceed as follows:

- Write a **defmatch** rule that identifies the **ilt** form in which we are interested, with sufficiently general semantic pattern variables to make it useful.
- Write a simplification function that assumes it is given an **ilt** form and uses the **defmatch** rule to decide if it is of the appropriate form for inversion. If so, construct the inverse transform; otherwise, return the original form.
- Write a function that applies the simplification function to **ilt** forms in an expression using an **opsubst/lambda** construct.

```
(c1) use_nilt:false$
(c2) assume(s>0,v>0)$
```

```
(c3) laplace(t*f(t),t,s);
                   d
(d3)             - -- (laplace(f(t), t, s))
                   ds
(c4) expr:ilt(%,s,t);
                 d
(d4)       ilt(- -- (laplace(f(t), t, s)), s, t)
                 ds
(c5) symbol_and_notequal(v1,v2):=is(symbolp(v2) and v2#v1)$
(c6) matchdeclare(t,symbolp,[f,s],symbol_and_notequal(t))$
(c7) defmatch(ilt_diff_rule,'ilt('diff('laplace(f(t),t,s),s),s,t),t)$
```

The command (c1), which sets **use_nilt** to **false**, disables the extended **ilt** code. (This just reduces the time needed for **ilt** to "noun out" the extended package can't invert this transform, either.) The command (c2) informs Macsyma that the transform variables are positive. In (c3) and (c4), we demonstrate that **ilt** is unable to invert this transform. In (c5) through (c7), we define the **defmatch** procedure that recognizes expressions of the form (c4) and identifies pertinent subexpressions.

We next declare **ilt** to be **outative** so that constants will factor out of the **ilt** forms (thereby simplifying the patterns that we have to write) and test the rule on a few examples.

```
(c8) declare(ilt,outative)$
(c9) expr:ev(expr,ilt);
                 d
(d9)       - ilt(-- (laplace(f(t), t, s)), s, t)
                 ds
(c10) ilt_diff_rule(expr,t);
(d10)                    false
(c11) ilt_diff_rule(-expr,t);
(d11)              [s = s, f = f, t = t]
(c12) ilt(laplace(u*g(u),u,v),v,u);
                 d
(d12)      - ilt(-- (laplace(g(u), u, v)), v, u)
                 dv
(c13) ilt_diff_rule(-%,u);
(d13)              [s = v, f = g, t = u]
```

Note that **ilt_diff_rule** matches in (c11) but not in (c10), due to the leading minus sign. This is not a problem since the rule will be applied only to **ilt** forms.

In the next phase, we define the function **ilt_simp** that takes an **ilt** form, calls **ilt_diff_rule** on it, and returns the inverse transform of the expression if it is of the right form; otherwise, it returns the original expression. We first remove any global bindings of f, t, and s.

```
(c14) remvalue(f,t,s);
(d14)                  [f, false, s]
```

```
(c15) expr:-ilt(laplace(t*f(t),t,s),s,t);
                d
(d15)       ilt(-- (laplace(f(t), t, s)), s, t)
                ds
 (c16) ilt_simp(expr):=block
([f,s,t,match],
 if (match:ilt_diff_rule(expr,part(expr,1,1,2))) # false
  then subst(match,-t*f(t))
  else expr)$
(c17) ilt_simp(expr);
(d17)                      t f(t)
```

The function **ilt_simp**, defined in (c16), inverts the transform, as is demonstrated in (c17). Finally, we write a function called **my_ilt** that invokes **ilt_simp** using an **opsubst/lambda** substitution. (This approach, when coupled with the **outative** simplification property that was added to **ilt**, greatly simplifies the steps needed to implement simplifications of this type.) The function **my_ilt** scans an expression for an **ilt** noun form, isolates its argument list, reconstructs the **ilt** noun form, and invokes **ilt_simp** on the result. The result of **ilt_simp** is substituted for the original **ilt** form. (It is important that **my_ilt** return the original form if the **ilt** form cannot be inverted!)

```
(c18) my_ilt(expr):=
 opsubst(nounify('ilt) = lambda([[foo]],apply(ilt_simp,[funmake(nounify('ilt),foo)])),expr)$
(c19) a*t*f(t)-6*t*f(t);
(d19)              a t f(t) - 6 t f(t)
(c20) laplace(%,t,s);
        d
(d20) 6 (-- (laplace(f(t), t, s)))
        ds
                                  d
                             - a (-- (laplace(f(t), t, s)))
                                  ds
(c21) ilt(%,s,t);
           d
(d21) 6 ilt(-- (laplace(f(t), t, s)), s, t)
           ds
                            d
                   - a ilt(-- (laplace(f(t), t, s)), s, t)
                            ds
(c22) my_ilt(%);
(d22)              a t f(t) - 6 t f(t)
```

Bibliography

[Be] Bender, C. M. and Orszag, S. A., *Advanced Mathematical Methods for Scientists and Engineers*, McGraw-Hill Book Company (1978).

[Ge] Gentleman, W. M. and Johnson, S. C., "The Evaluation of Determinants by Expansion by Minors and the General Problem of Substitution," in *Mathematics of Computation* (April 1974), vol. 28, no. 126.

[Go] Golnaraghi, M., Keith, W., and Moon, F. C., "Stability Analysis of a Robotic Mechanism Using Computer Algebra" in *Applications of Computer Algebra*, Pavelle, R., ed., Kluwer Academic Publishers, Boston (1985).

[Gr] Gradshteyn, I. S. and Ryzhik, I. M., *Table of Integrals, Series, and Products*, Academic Press, Inc., New York (1980).

[Le] Lee, C. S. G., "Robot Arm Dynamics" in *Tutorial on Robotics*, Lee, C. S. G., Gonzalez, R. C., and Fu, K. S., ed., IEEE Computer Society Press (1983).

[Lh] Leu, M. C. and Hemati, N., "Automated Symbolic Derivation of Dynamic Equations of Motion for Robotic Manipulators," in *Journal of Dynamic Systems, Measurement, and Control* (Sept. 1986), vol. 108.

[Na] Nayfeh, A. H., *Problems in Perturbation*, John Wiley and Sons, New York (1985).

[Va] Vandergraft, J. S., *Introduction to Numerical Computations*, Academic Press, New York (1978).

[Wa] Wang, P. S., "Modern Symbolic Mathematical Computing Systems" in *Applications of Computer Algebra*, Pavelle, R., ed., Kluwer Academic Publishers, Boston (1985).

Appendix A

Hints for New Users of Macsyma

Often, new users of Macsyma have a number of unrealistic expectations about the nature of symbolic computing and what computer algebra systems can do. Sometimes these misconceptions can lead to poor experiences with the system. The following maxims may help you avoid "start up" problems with Macsyma.

- **Learn how symbolic algebra differs from other computational methods.**

  Algebraic manipulation is a new and different computational approach for which prior experience with computational methods is likely to be misleading. Attempt to learn the ways in which algebraic manipulation differs from other approaches. Some of these differences are covered in Chapter 1.

  For example, consider the problem of inverting an $n \times n$ matrix. In numerical analysis, you learn that the problem requires on the order of n^3 operations. Some theoreticians will even point out that, for n sufficiently large, the number of multiplications can be reduced to $n^{2.8}$.

  This kind of analysis is not altogether helpful in dealing with matrices with symbolic entities. Consider the problem of estimating the number of terms in the symbolic calculation of a matrix inverse. Since the matrix is symbolic, there is no special structure to exploit, and the general inverse can be calculated as the matrix of cofactors divided by the determinant. The number of terms in the determinant of the general $n \times n$ matrix whose elements are the symbols a_{ij} has, when fully expanded, $n!$ terms. Each cofactor element is an $(n-1) \times (n-1)$ determinant. Thus, each of the n^2 elements of the inverse has, in fully expanded form, $n! + (n-1)!$ terms. If you now subsitute numbers for the elements, the numerical inverse is of the order of $n^2 n!$ operations. For $n > 5$, this is clearly nowhere near n^3.

  When poorly posed, some algebraic manipulation problems have a combinatorial or exponential character which makes them very different from problems of numerical analysis.

- **Decide whether a numeric approach is more effective for your problem than a symbolic approach.**

  Sometimes, trying to solve the most general formulation of your problem is too hard. If you can identify one or two symbolic parameters to carry through the problem, you may find it easier to graph parametric solutions than to solve the general problem.

- **Curb exponential growth of time and memory requirements and avoid generalizing problems.**

  A common tendency for a beginning user is to needlessly generalize a problem and thus cause inevitable exponential growth. Consider the problem of obtaining determinants of matrices whose entries are formulas that vary from problem to problem. New users often consider obtaining the determinant of a general matrix and substituting the entries into the result. This might work for matrices of order $n < 5$, but it is a poor plan for dealing with exponential growth inherent in the problem.

You should be aware of the types of calculations that have exponential growth in the general case. These include matrix calculation, repeated differentiation of products or quotients (”brute force” Taylor series calculations), and solutions of systems of polynomial equations.

- **Anticipate a certain amount of trial-and-error in calculations.**

  A certain amount of trial-and-error is necessary in many calculations to determine a good sequence of operations for obtaining the solution. It pays, however, to consider carefully before trying a powerful method.

  Consider, for example, the problem of obtaining a truncated Taylor series of an expression with several variables. If you truncate all variables, the number of terms in your result might be an exponential in the number of variables and the degree of the truncation. Thus, it is not surprising that the time it takes to compute the sixth order terms is much larger than that of the fifth order.

- **Keep the number and degree of a problem’s variables to a minimum.**

  Reduce the number and degree of variables in a problem as much as possible, since the exponential growth inherent in some computations is usually a function of the number and degree of variables in the expression. This rule is especially applicable to expressions represented as polynomials or rational functions or inputs to powerful polynomial-based algorithms (such as factorization, matrix operations, and equation solving).

- **Convert from general to rational representation to avoid overhead.**

  It is occasionally useful to convert all expressions to the internal rational function form when you expect to manipulate large formulas (greater than 50 terms). Use the **rat** command to avoid the overhead of general representation. The downside of this approach is that you can lose the structure of the formulas.

- **Use pattern matching to customize your application.**

  Pattern matching is a very useful Macsyma facility which allows you to ”tune” the system to your application. Although learning to use the pattern matching facilities effectively is no small task, users with complicated problems will benefit greatly from the effort. An introduction to pattern matching appears in Chapter 14.

- **Recurse Carefully.**

  Symbolic computing makes it easy to use an important technique: the ability to formulate calculations and algorithms recursively. Recursion is a powerful alternative to iteration. New users frequently abuse recursion by omitting proper end or beginning conditions. This often leads to unending recursions, and subsequent crashes or “out of memory” errors from stack overflow. Macsyma has many debugging and tracing tools to catch errors due to faulty recursions. Information on programming techniques for debugging functions in Macsyma appears in Chapter 12 and Section 11.6.

Appendix B

Answers to Practice Problems

This appendix presents solutions to the practice problems that have been given in the preceding chapters. Note that a solution generally represents only one of the many ways in which the problem could have been solved with Macsyma.

B.1 Answers for Chapter 4

This section presents the solutions to the problems given in Section 4.8, page 63.

Problem 1. See page 63.

```
(c1) f(expr, x) := ratsubst(1 - cos(x)^2, sin(x)^2, expr)$
(c2) f(sin(y)^3, y);
                                  2
(d2)               (1 - cos (y)) sin(y)
```

Problem 2. See page 63.

```
(c1) f(expr, x) := subst((1 - cos(2*x))/2, sin(x)^2, expr)$
(c2) f(sin(y)^2, y);
                       1 - cos(2 y)
(d2)                   ------------
                            2
(c3) f(sin(y)^3, y);
                          3
(d3)                   sin (y)
```

Problem 3. See page 63.

```
(c1) expr:(sqrt(r^2 + a^2) + a)*(sqrt(r^2 + b^2) + b)/r^2
          - (sqrt(r^2 + b^2) + sqrt(r^2 + a^2) + b + a)
          /(sqrt(r^2 + b^2) + sqrt(r^2 + a^2) - b - a);
```

```
             2    2               2    2
     (sqrt(r  + a ) + a) (sqrt(r  + b ) + b)
(d1) ---------------------------------------
                          2
                         r
                             2    2          2    2
                       sqrt(r  + b ) + sqrt(r  + a ) + b + a
                     - -------------------------------------
                             2    2          2    2
                       sqrt(r  + b ) + sqrt(r  + a ) - b - a
(c2) ratsimp(%);
(d2)                      0
(c3) radcan(expr);
(d3)                      0
```

Problem 4. See page 64.

```
(c1) (b + a)*(d + c) + (1/(y + x)^4 - 3/(z + y)^3)^2;
                  1          3     2
(d1)        (-------- - --------)  + (b + a) (d + c)
                    4          3
             (y + x)    (z + y)
(c2) expand(%, 2, 0);
                6              9          1
(d2) - ----------------- + -------- + -------- + b d + a d + b c + a c
               4        3          6          8
         (y + x)  (z + y)    (z + y)    (y + x)
```

Problem 5. See page 64.

```
(c1) expr:(d+c)*((w+a)*x+b);
(d1)              (d + c) ((w + a) x + b)
```

Answers to problems 5 *a* through *f*:

```
(c2) expand(expr);
(d2)        d w x + c w x + a d x + a c x + b d + b c
(c3) multthru(expr);
(d3)               (d + c) (w + a) x + b (d + c)
(c4) distrib(expr);
(d4)          d (w + a) x + c (w + a) x + b d + b c


(c5) ratsimp(expr);
(d5)          ((d + c) w + a d + a c) x + b d + b c
(c6) ratsimp(expr,c,d);
(d6)          d ((w + a) x + b) + c ((w + a) x + b)
```

```
(c7) ratsimp(expr,b,a);
(d7)           (d + c) w x + a (d + c) x + b (d + c)
```

Problem 6. See page 64.

```
(c1) expr:d*(w + a)*x +c*(w + a)*x + b*d + b*c;
(d1)           d (w + a) x + c (w + a) x + b d + b c
(c2) factor(expr);
(d2)              (d + c) (w x + a x + b)
(c3) factorsum(expr);
(d3)              (d + c) ((w + a) x + b)
```

Problem 7. See page 64.

```
(c1) expr:log((b + a)*d + (b + a)*c)*z + log((b + a)*d + (b + a)*c)*y
          + log((b+a)*d + (b+a)*c)*x + w;
(d1) log((b + a) d + (b + a) c) z + log((b + a) d + (b + a) c) y
                     + log((b + a) d + (b + a) c) x + w


(c2) factorsum(expr);
(d2)           log((b + a) (d + c)) (z + y + x) + w
(c3) ratsimp(expr, log((a + b)*(d + c)));
(d3)            log((b + a) d + (b + a) c) (z + y + x) + w
```

Problem 8. See page 65.

```
(c1) expr:1/(log(x)^2 - x^2);
                                 1
(d1)                       ------------
                              2       2
                           log (x) - x
```

Answer to part *a*:

```
(c2) partfrac(expr, x);
                          1                         1
(d2)           --------------------- - ---------------------
               2 log(x) (log(x) + x)   2 (x - log(x)) log(x)
```

Answer to part *b*:

```
(c2) partfrac(expr, log(x));
                      1                   1
(d2)            ---------------- - ----------------
                2 x (log(x) - x)   2 x (log(x) + x)
```

Problem 9. See page 65.

```
(c1) (x + 1)/(sqrt(x) - 1);
                              x + 1
(d1)                       -----------
                           sqrt(x) - 1
(c2) ratsimp(%), algebraic:true;
                     sqrt(x) (x + 1) + x + 1
(d2)                 -----------------------
                               x - 1
```

Problem 10. See page 65.

Solution Method 1:

```
(c1) (load(nusum1), trigsimp(imagpart(
        closedform(sum(k*exp(%i*k*x),k,1,n)
                                             ) ) );
(d1)
    ((n+1)*cos(x)-n)*sin((n+1)*x) - (n+1)*sin(x)*cos((n+1)*x)
 - -----------------------------------------------------------
                    2*cos(x)-2
```

Solution Method 2:

```
(c1) 'sum(cos(k*x), k, 1, n) = 'sum(cos(k*x), k, 1, n);
               n                n
              ====             ====
              \                \
(d1)           >    cos(k x) =  >    cos(k x)
              /                /
              ====             ====
              k = 1            k = 1

(c2) lhs(%) = exponentialize(rhs(%));
                               n
                              ====
                              \        %i k x    - %i k x
                               >    (%e       + %e      )
           n                  /
          ====                ====
          \                   k = 1
(d2)       >    cos(k x) =  -----------------------------
          /                               2
          ====
          k = 1
```

```
(c3) ev(%, sum, simpsum:true);
                      %i (n + 1) x    %i x       - %i (n + 1) x     - %i x
                    %e             - %e        %e                - %e
     n              ---------------------- + --------------------------
    ====                       %i x                    - %i x
    \                        %e     - 1              %e       - 1
(d3)  >    cos(k x) = ----------------------------------------------------
    /                                      2
    ====
    k = 1


(c4) ev(demoivre(%), ratsimp, trigreduce);
     n
    ====
    \               cos(n x + x)   cos(n x)        cos(x)
(d4)  >    cos(k x) = ------------ - ------------ - ------------
    /               2 cos(x) - 2   2 cos(x) - 2   2 cos(x) - 2
    ====
    k = 1
                                                         1
                                                  + ------------
                                                    2 cos(x) - 2


(c5) diff(%, x);
       n
      ====
      \                    (n + 1) sin(n x + x)   2 sin(x) cos(n x + x)
(d5) -  >    k sin(k x) = - -------------------- + ---------------------
      /                        2 cos(x) - 2                        2
      ====                                           (2 cos(x) - 2)
      k = 1
   n sin(n x)    2 sin(x) cos(n x)      sin(x)      2 cos(x) sin(x)
 + ------------ - ----------------- + ------------ - ---------------
  2 cos(x) - 2                   2    2 cos(x) - 2                 2
                 (2 cos(x) - 2)                    (2 cos(x) - 2)
     2 sin(x)
 + ---------------
                 2
  (2 cos(x) - 2)
```

```
(c6) combine(%);
       n
      ====
      \                  - (n + 1) sin(n x + x) + n sin(n x) + sin(x)
(d6) -  >    k sin(k x) = --------------------------------------------
      /                                 2 cos(x) - 2
      ====
      k = 1
     2 sin(x) cos(n x + x) - 2 sin(x) cos(n x) - 2 cos(x) sin(x) + 2 sin(x)
   + ----------------------------------------------------------------------
                                              2
                                 (2 cos(x) - 2)
```

Problem 11. See page 66.

Solution Method 1:

```
(c1) (load(nusum1), closedform(sum(n^3*3^n,n,1,m)));
                      3    2          m
      33          3*(4m - 6m +12m -11)3
(d1) ----   +     -----------------------
       3
      2                     8
```

Solution Method 2:

```
(c1) nusum(3^n*n^3, n, 1, m);
                 3      2               m
           3 (4 m  - 6 m  + 12 m - 11) 3      33
(d1)       ------------------------------  +  --
                            8                  8
```

B.2 Answers for Chapter 5

This section presents the solutions to the problems given in Section 5.5, page 74.

Problem 1. See page 74.

```
(c1) x^4 - 7*x^3 + 18*x^2 - 20*x + 8;
                   4      3       2
(d1)              x  - 7 x  + 18 x  - 20 x + 8
(c2) solve(%, x);
(d2)                 [x = 1, x = 2]
(c3) multiplicities;
```

```
(d3)                    [1, 3]
```

Problem 2. See page 74.

```
(c1) eq:x^5 - x^2/980/3 - 3*1/10/980^2 = 0;
                             2
                      5     x       3
(d1)                 x  - ---- - ------- = 0
                          2940   9604000
```

Find the number of real roots.

```
(c2) nroots(eq);
(d2)                        1
```

Find all real roots.

```
(c3) ev(realroots(eq), numer:true);
(d3)                  [x = 0.073550195]
```

Find all numerical roots.

```
(c4) allroots(eq);
(d4) [x = 0.03009181 %i + 0.0011866146, x = 0.0011866146 - 0.03009181 %i,
x = 0.056936793 %i - 0.03796171, x = - 0.056936793 %i - 0.03796171,
x = 0.07355019]
```

Problem 3. See page 74.

```
(c1) eq1:x + y + z = 3;
(d1)                      z + y + x = 3
(c2) eq2:y*z + x*z + x*y = -18;
(d2)                 y z + x z + x y = - 18
(c3) eq3:z^3 + y^3 + x^3 = 189;
                        3    3    3
(d3)                   z  + y  + x  = 189
(c4) solve([eq1, eq2, eq3], [x, y, z]);
(d4) [[x = 0, y = 6, z = - 3], [x = 0, y = - 3, z = 6],
[x = 6, y = 0, z = - 3], [x = 6, y = - 3, z = 0], [x = - 3, y = 0, z = 6],
[x = - 3, y = 6, z = 0]]
```

Problem 4. See page 74.

```
(c1) eqs:[x^2*y + y = 1, y - 2*x = 4];
                         2
(d1)                   [x  y + y = 1, y - 2 x = 4]
```

The solutions are long, so suppress display of the D-LINE.

```
(c2) solve(eqs, [x, y]),algexact:true$
```

Look at the third solution.

```
(c3) part(%, 3);
            sqrt(73)   77  1/3              1             2
(d3) [x = (-------- - ---)    + --------------------  - -,
               12      108          sqrt(73)   77  1/3    3
                                 9 (-------- - ---)
                                       12      108
                          2/3      2/3                   1/3       1/3
          2 (9 sqrt(73) - 77)    + 8 2    (9 sqrt(73) - 77)    + 4 2
      y = -------------------------------------------------------------]
                        2/3                    1/3
                     3 2    (9 sqrt(73) - 77)
```

Problem 5. See page 74.

```
(c1) eq1:a*x + 12*y - z = 3*b;
(d1)              - z + 12 y + a x = 3 b
(c2) eq2:x - 4*c*y - 5*z = 0;
(d2)               - 5 z - 4 c y + x = 0
(c3) eq3:x + 4*y = c;
(d3)                    4 y + x = c
```

Answer to part a:

```
(c4) linsolve([eq1, eq2, eq3], [x, y, z]), globalsolve:false;
              2
           - c  - 15 c + 15 b        - 5 a c + c + 15 b
(d4) [[x = ------------------, y = - ------------------,
            - c + 5 a - 16           - 4 c + 20 a - 64
                                  2
                             - a c  + b (3 c + 3) - 3 c
                         z = --------------------------]]
                               - c + 5 a - 16
(c5) y;
(d5)                     y
```

Answer to part b:

```
(c6) linsolve([eq1, eq2, eq3], [x, y, z]), globalsolve:true$
(c7) y;
                       - 5 a c + c + 15 b
(d7)                 - ------------------
```

```
                    - 4 c + 20 a - 64
(c8) remvalue(x, y, z);
(d8)                 [x, y, z]
```

Problem 6. See page 75.

```
(c1) eq1:x^2 + y^2 = 1;
                        2    2
(d1)                   y  + x  = 1
(c2) eq2:-4*x*z = 0;
(d2)                   - 4 x z = 0


(c3) solve([eq1, eq2], [x, y, z]);
                                           2
(d3) [[x = %r9, y = - sqrt(1 - %r9 ), z = 0],
                                 2
[x = %r10, y = sqrt(1 - %r10 ), z = 0], [x = 0, y = - 1, z = %r11],
[x = 0, y = 1, z = %r12]]
```

Problem 7. See page 75.

```
(c1) eq:e^2*x^6 - e*x^4 - x^3 + 2*x^2 + x - 2 = 0;
                2  6      4    3      2
(d1)           e  x  - e x  - x  + 2 x  + x - 2 = 0
```

Answer to part *a*. This returns the six general solutions. The k0 symbols in the solutions are the undetermined coefficients that satisfy the last equation in each list; for example, $\mathrm{k0}^3 = 1$.

```
(c2) taylor_solve(eq, x, e, 0, [2]);
                          2
                  (5 k0  - 4) e              3       2
(d2)/T/ [[x = k0 - ---------------- + . . ., k0  = 2 k0  + k0 - 2],
                         2
                  3 k0  - 4 k0 - 1
                      k0        1                 3
                [x = ---- + ----------- + . . ., k0  = 1]]
                      2/3           1/3
                     e      (3 k0) e
```

Answer to part *b*. Notice that (d3) returns the second half of the general solution above.

```
(c3) taylor_solve(eq, x, e, 0, [2]), taylor_solve_choose_order:true;
                                                            2
Possible choices for the order of a series solution are:  [- -, 0]
                                                            3
Please enter your choice:
```

```
-2/3;
                       k0      1                     3
(d3)/T/         [[x = ---- + ----------- + . . ., k0  = 1]]
                       2/3           1/3
                      e      (3 k0) e
```

Answer to part c. This is but one of the many choices for coefficient that you could make.

```
(c4) taylor_solve(eq, x, e, 0, [2]),taylor_solve_choose_coef:true;
                                    1
Possible choices for the coef. of  ---- are:
                                    2/3
                                   e
      sqrt(3) %i - 1         sqrt(3) %i + 1
[k0 = --------------, k0 = - --------------, k0 = 1]
            2                      2


Please enter your choice:
k0 = 1;
Possible choices for the coef. of  1 are:  [k0 = 2, k0 = - 1, k0 = 1]
Please enter your choice:
k0 = 2;
                     16 e                  1       1
(d4)/T/    [[x = 2 - ---- + . . .], [x = ---- + ------ + . . .]]
                       3                  2/3     1/3
                                         e      3e
```

B.3 Answers for Chapter 6

This section presents the solutions to the problems given in Section 6.8, page 110.

Problem 1. See page 110.

Define the Legendre polynomials using the Rodrigues formula.

```
(c1) p(l, x) := 1/2^l/l!*diff((x^2 - 1)^l, x, l);
                                    2     l
                               diff((x  - 1) , x, l)
                               ---------------------
                                         l
                                        2
(d1)              p(l, x) := ---------------------
                                         l!
```

Define the Legendre polynomials using the recurrence relation.

```
(c2) p[0]:1$
(c3) p[1]:x$
(c4) p[n]:=ratsimp((2*n - 1)*x*p[n - 1]/n - (n - 1)*p[n - 2]/n)$
```

Problem 2. See page 110.

```
(c1) assume(x < 1)$
(c2) integrate(x/(1 - x), x), logabs:true;
(d2)                 - x - log(1 - x)
(c3) forget(x < 1)$
```

Problem 3. See page 110.

```
(c1) assume(a > 0)$
(c2) integrate(exp(-a*x)*(cos(x) + sin(x)), x, 0, inf), intanalysis:false;
                      a + 1
(d2)                  ------
                       2
                      a  + 1
```

Or, alternatively:

```
(c3) ldefint(exp(-a*x)*(cos(x) + sin(x)), x, 0, inf);
                         a       1
(d3)                  ------ + ------
                           2        2
                      a  + 1   a  + 1
(c4) forget(a > 0)$
```

Problem 4. See page 111.

```
(c1) assume(m > 0, n > 0)$
(c2) integrate((cos(m*x) - cos(n*x))/x, x, 0, inf), intanalysis:false,
                            laplace_call:all;
(d2)                  log(n) - log(m)
(c3) forget(m > 0, n > 0)$
```

Problem 5. See page 111.

```
(c1) romberg((cos(2*x) - cos(3*x))/x, x, 0.01, 8.0);
(d1)                 0.4294731
```

Problem 6. See page 111.

```
(c1) depends(y, x);
(d1)                  [y(x)]
(c2) y = 'integrate(f(x - t), t, 0, x);
```

```
                 x
                /
                [
(d2)        y = I  f(x - t) dt
                ]
                /
                 0


(c3) changevar(%, x - t = u, u, t);
                           0
                          /
                          [
(d3)               y = -  I  f(u) du
                          ]
                          /
                           x
(c4) diff(%, x);
                   dy
(d4)               -- = f(x)
                   dx
```

Problem 7. See page 111.

```
(c1) limit(log(cos(x))/log(1 - sin(x)), x, %pi/2);
                       1
(d1)                   -
                       2
```

Problem 8. See page 112.

```
(c1) expr:(sin(x) - atan(x))/x^2/log(x + 1);
                   sin(x) - atan(x)
(d1)               ----------------
                     2
                    x  log(x + 1)


(c2) tlimit(expr, x, 0);
                       1
(d2)                   -
                       6
```

Problem 9. See page 112.

```
(c1) taylor(sin(e*t)/t, e, 0, 5);
                    2  3    4  5
```

```
                      t  e     t  e
(d1)/T/          e - ----- + ----- + . . .
                        6       120
(c2) ldefint(%, t, 0, 1);
                         5     3
                        e     e
(d2)                    --- - -- + e
                        600   18
(c3) trunc(%);
                         3    5
                        e    e
(d3)               e - -- + --- + . . .
                        18   600
```

Problem 10. See page 112.

```
(c1) depends(f, x)$
(c2) eq:diff(f, x, 3) - 4*diff(f, x)*(1 - 3*sech(x)^2)
        - f*(24*sech(x)^2*tanh(x) + a);
```

```
                                                                          3
                     2                       df             2       d f
(d2)    - f (24 sech (x) tanh(x) + a) - 4 -- (1 - 3 sech (x)) + ---
                                             dx                       3
                                                                    dx
(c3) sol:f = 4*'diff(exp((g - 1)*x)*sech(x), x, 2) + g*(g - 2)^2*exp(g*x);


                    2
                   d      (g - 1) x                    2     g x
(d3)       f = 4 (--- (%e          sech(x))) + (g - 2)  g %e
                    2
                   dx
(c4) constraint:g^3 - 4*g - a = 0$
(c5) exponentialize:true$
(c6) ev(eq, ev(sol, diff), diff)$
(c7) factor(%)$
(c8) scsimp(%, constraint);
(d8)                     0
(c9) remove(f, dependency)$
```

Restore option variables to their default settings.

```
(c10) reset()$
```

Problem 11. See page 112.

```
(c1) depends(y, x);
(d1)                   [y(x)]
(c2) (3*x*y + y^2) + (x^2 + x*y)*'diff(y, x) = 0;
                            2  dy    2
(d2)               (x y + x ) -- + y  + 3 x y = 0
                               dx
(c3) ode(%, y, x);
                      2  2      3
                     x  y  + 2 x  y
(d3)                 -------------- = %c
                           2
(c4) method;
(d4)                    exact
(c5) intfactor;
(d5)                      x
```

Problem 12. See page 113.

```
(c1) depends(y, x);
(d1)                   [y(x)]
(c2) x*(y^2 - 3*x)*diff(y, x) + 2*y^3 - 5*x*y = 0;
```

```
                      2        dy      3
(d2)           x (y  - 3 x) -- + 2 y  - 5 x y = 0
                            dx


(c3) ode(%, y, x);
                       2
                    5 y  - 13 x              y
              2 log(-----------) - 30 log(-------)
                         x                sqrt(x)
              -----------------------------------
                                65
(d3)          x = %c %e
(c4) method;
(d4)                  genhom
(c5) odeindex;
                          1
(d5)                    - -
                          2
```

Problem 13. See page 113.

```
(c1) depends(y, x);
(d1)                  [y(x)]
(c2) eq:diff(y, x, 2) + 1/x*diff(y, x) + y = 0;
                           dy
                      2    --
                     d y   dx
(d2)                 --- + -- + y = 0
                       2   x
                     dx
```

Solve by the default method.

```
(c3) ode(eq, y, x);
(d3)                y = %k2 %y (x) + %k1 %j (x)
                              0            0
(c4) method;
(d4)                  bessel
```

Solve by the series method.

```
(c5) ode(eq, y, x, odeseries);
DIAGNOSIS: TYPE: singular    equal  ROOTS: R1= 0   R2= 0
SINGULARITIES: [0, inf]
```

```
                     inf
                     ====          %%n  2 %%n
                     \       (- 1)     x
(d5) [y = %k2 (log(x)  >       ---------------
                     /            %%n     2
                     ====        4    %%n!
                     %%n = 0
        inf                                          inf
        ====                          %%n  2 %%n     ====          %%n  2 %%n
        \          harm(1, %%n) (- 1)     x          \       (- 1)     x
   - 2  >       ---------------------------) + %k1  >      ---------------]
        /                %%n     2                   /            %%n     2
        ====            4    %%n!                    ====        4    %%n!
        %%n = 0                                      %%n = 0
(c6) method;
(d6)                 bessel
```

Problem 14. See page 113.

```
(c1) assume(a > 0)$
(c2) eq1:3*'diff(f(x), x, 2) - 2*'diff(g(x), x) = sin(x);
                   2
                  d                d
(d2)          3 (--- (f(x))) - 2 (-- (g(x))) = sin(x)
                   2              dx
                 dx
(c3) eq2:a*'diff(g(x), x, 2) + 'diff(f(x), x) = a*cos(x);
                    2
                   d            d
(d3)           a (--- (g(x))) + -- (f(x)) = a cos(x)
                    2            dx
                  dx
(c4) atvalue(g(x), x = 0, 1)$
(c5) atvalue('diff(f(x), x), x = 0, 0)$
(c6) atvalue('diff(g(x), x),x = 0, 1)$
```

```
(c7) desolve([eq1, eq2], [f(x), g(x)]);
                 5/2     sqrt(6) x
             27 a    sin(---------)
                         3 sqrt(a)       2     sqrt(6) x
             --------------------- - 3 a  cos(---------)
               sqrt(6) (3 a - 2)               3 sqrt(a)    3 a sin(x)
(d7) [f(x) = ------------------------------------------- - ---------- + a
                                3 a                          3 a - 2
                    3/2     sqrt(6) x            2     sqrt(6) x
                9 a    sin(---------)       27 a  cos(---------)
                            3 sqrt(a)                  3 sqrt(a)
                -------------------- + --------------------------
                     sqrt(6)                  6 a - 4
 + f(0), g(x) = ---------------------------------------------------
                                   3 a
   (3 a + 1) cos(x)   1
 - ---------------- + -]
       3 a - 2        2
```

Problem 15. See page 114.

```
(c1) depends([u, v], xx)$
(c2) eq:(1 + u^2/(1 - u^2))*diff(u, xx, 2) + u*diff(u, xx)^2/(1 - u^2)
        + w0^2*u + g/l*u/sqrt(1 - u^2);
                                            du  2
                        2         2    u (---)
               2      u          d u      dxx          g u
(d2)        u w0  + (------ + 1) ---- + -------- + --------------
                           2        2        2                2
                      1 - u       dxx     1 - u     l sqrt(1 - u )
```

Expand the equation above with **taylor**, keeping terms up to cubic terms

```
(c3) taylor(eq, u, 0, 3);
                     2      du  2                          du  2       3
         2     (l w0  + l (---)  + g) u    2        (2 l (---)  + g) u
        d u                dxx            d u   2          dxx
(d3)/T/ ---- + ----------------------- + ---- u  + -------------------
          2               l                2               2 l
        dxx                              dxx
                                                                + . . .
```

```
(c4) neq:expand(subst(w^2 - g/l, w0^2, %));
                        2      2                               3
              2 2      d u    d u     3  du  2      du  2   g u
(d4)        u w  + u  ---- + ---- + u  (---)  + u (---)  + ----
                         2      2        dxx        dxx     2 l
                      dxx    dxx
```

Since u is small, scale it with e^{ll} (for ll greater than 0).

```
(c5) u = v*e^ll;
                           ll
(d5)                  u = e   v
```

Evaluate the equation above into the original neq, then differentiate.

```
(c6) ev(neq, %, diff);
                           2          2
      ll    2    3 ll  2 d v     ll d v     5 ll  3  dv  2    3 ll    dv  2
(d6) e   v w  + e     v  ---- + e   ---- + e     v  (---)  + e     v (---)
                            2          2             dxx              dxx
                         dxx        dxx
                                             3 ll    3
                                            e     g v
                                          + ----------
                                               2 l
(c7) multthru(e^-ll, %);
                         2      2
       2    2 ll  2 d v    d v     4 ll  3  dv  2    2 ll    dv  2
(d7) v w  + e     v  ---- + ---- + e     v  (---)  + e     v (---)
                        2      2             dxx              dxx
                     dxx    dxx
                                                              2 ll    3
                                                             e     g v
                                                           + ----------
                                                                2 l
```

Choose $ll = 1/2$.

```
(c8) nneq:ev(%, ll = 1/2);
                       2      2                                  3
       2 2          d v    d v     2  3  dv  2         dv  2    e g v
(d8)   v w  + e v  ---- + ---- + e  v  (---)  + e v (---)  + ------
                       2      2         dxx           dxx       2 l
                    dxx    dxx
```

Introduce the change of variable.

```
(c9) changeofv:x = (w + e*w1)*xx;
(d9)                 x = (e w1 + w) xx
(c10) depends([v0, v1], x)$
(c11) gradef(x, xx, w + e*w1)$
(c12) v = v0 + e*v1;
(d12)                  v = e v1 + v0
```

Introduce the change of variable *changeofv* into the differential equation *nneq*.

```
(c13) ev(nneq, %, diff);
                          2                  2
                    2    d v1           2   d v0            2
(d13) e (e v1 + v0)  (e ---- (e w1 + w)  + ---- (e w1 + w) )
                            2                  2
                          dx                 dx
    2           3     dv1              dv0            2
 + e  (e v1 + v0)  (e --- (e w1 + w) + --- (e w1 + w))
                       dx               dx
                                                           2
                    dv1              dv0            2     d v1            2
 + e (e v1 + v0) (e --- (e w1 + w) + --- (e w1 + w))  + e ---- (e w1 + w)
                    dx               dx                      2
                                                           dx
    2                                                3
   d v0          2                 2   e g (e v1 + v0)
 + ---- (e w1 + w)  + (e v1 + v0) w  + ----------------
     2                                       2 l
   dx
```

Truncate the differential equation above, keeping only terms e^2 or lower.

```
(c14) nnneq:ratsubst(0, e^2, %);
             2                  2                                    2
            d v0               d v1                          2      d v0
(d14) (4 e l ---- w w1 + (2 e l ---- + 2 e l v1 + (2 e l v0  + 2 l) ----
               2                  2                                   2
             dx                 dx                                  dx
                       dv0 2             2          3
            + 2 e l v0 (---)  + 2 l v0) w  + e g v0 )/(2 l)
                        dx
```

Equate terms independent of e.

```
(c15)e_0:ratcoef(nnneq, e, 0);
                          2
                         d v0        2
(d15)                   (---- + v0) w
                           2
                         dx
```

Equate terms to the first order in en.

```
(c16) e_1:ratcoef(nnneq, e, 1);
            2                 2                        2
           d v0              d v1                  2 d v0           dv0 2
(d16) (4 l ---- w w1 + (2 l ---- + 2 l v1 + 2 l v0  ---- + 2 l v0 (---) )
             2                 2                        2            dx
           dx                dx                       dx
                                                            2       3
                                                           w  + g v0 )/(2 l)
(c17) ode(e_0, v0, x);
(d17)               v0 = %k1 sin(x) + %k2 cos(x)
```

Re-express the solution to the differential equation.

```
(c18) sol_e_0:trigreduce(ic2(%, x = 0, v0 = a*cos(b),
                 'diff(v0, x) = -a*sin(b)));
(d18)                 v0 = a cos(x + b)
```

Put the solution of *v0* into the differential equation *e_1* and carry out the derivative.

```
(c19) e_1:ev(e_1, %, diff, trigreduce, expand);
        3  2                      3
       a  w  cos(3 x + 3 b)   a  g cos(3 x + 3 b)
(d19) - -------------------- + ------------------ - 2 a w w1 cos(x + b)
                2                     8 l
             3  2                3                  2
            a  w  cos(x + b)   3 a  g cos(x + b)   d v1  2       2
          - ---------------- + ----------------- + ---- w  + v1 w
                   2                 8 l              2
                                                    dx
```

```
(c20) ode(%, v1, x);
                   3    2    3
(d20) v1 = - ((4 a  l w  - a  g) cos(3 x + 3 b) - 64 a l w w1 x sin(x + b)
       3    2                    3
 - 16 a  l w  x sin(x + b) + 12 a  g x sin(x + b)
                        3    2      3                        2
 + (- 32 a l w w1 - 8 a  l w  + 6 a  g) cos(x + b))/(64 l w ) + %k1 sin(x)
 + %k2 cos(x)
```

The secular term:

```
(c21) ratcoef(%, x, 1);
                                    3    2      3
                 (16 a l w w1 + 4 a  l w  - 3 a  g) sin(x + b)
(d21)        0 = ---------------------------------------------
                                             2
                                       16 l w
```

Choose the parameter for *w1* to eliminate the secular term

```
(c22) eq_w1:solve(%, w1);
                                2    2      2
                             4 a  l w  - 3 a  g
(d22)               [w1 = - ------------------]
                                   16 l w
(c23) ev(changeofv, eq_w1);
                                2    2      2
                          e (4 a  l w  - 3 a  g)
(d23)           x = (w - ----------------------) xx
                                  16 l w
(c24) ev(sol_e_0, %);
                                     2    2      2
                               e (4 a  l w  - 3 a  g)
(d24)         v0 = a cos((w - ----------------------) xx + b)
                                       16 l w
```

Since e was introduced for bookkeeping purposes

```
(c25) ev(%, e = 1);
                                    2    2      2
                                 4 a  l w  - 3 a  g
(d25)           v0 = a cos((w - ------------------) xx + b)
                                      16 l w
```

B.4 Answers for Chapter 11

This section presents the solutions to the problems given in Section 11.8, page 173.

Problem 1. See page 174.

```
(c1) dot(a, b) := block(sum(a[i]*b[i], i, 1, 3))$
(c2) cross(a, b) :=
    block([temp1, temp2, temp3],
          if listp(a) and listp(b)
          then (temp1:a[2]*b[3] - a[3]*b[2],
            temp2:a[3]*b[1] - a[1]*b[3],
            temp3:a[1]*b[2] - a[2]*b[1],
            return([temp1, temp2, temp3]))
          else print('undefined_operations))$
```

Problem 2. See page 174.

```
(c3) (aa:[a[1], a[2], a[3]],
      bb:[b[1], b[2], b[3]],
      cc:[c[1], c[2], c[3]])$
(c4) expand(cross(aa, cross(bb, cc)) + cc*dot(aa, bb) - bb*dot(aa, cc));
(d4)                 [0, 0, 0]
```

*Problem 3.*See page 174.

```
(c5) my_grad(a) :=
    block(if listp(a)
          then print('undefined_operations)
          else return([diff(a, x), diff(a, y), diff(a, z)]))$
(c6) my_div(a) :=
    block([temp],
          if listp(a)
          then temp:ratsimp(diff(a[1], x) + diff(a[2], y) + diff(a[3], z))
          else print('undefined_operations))$
(c7) my_curl(a) :=
    block([temp, temp1, temp2, temp3],
          if listp(a)
          then (temp1:diff(a[3], y) - diff(a[2], z),
            temp2:diff(a[1], z) - diff(a[3], x),
            temp3:diff(a[2], x) - diff(a[1], y),
            temp:[ratsimp(temp1), ratsimp(temp2), ratsimp(temp3)],
            return(temp))
          else print('undefined_operations))$
```

Problem 4. See page 174.

```
(c8) expr3*x^2*y - y^3*z^2$
(c9) my_grad(expr);
                             2      2  2       3
(d9)            [6 x y, 3 x  - 3 y  z , - 2 y  z]
(c10) ev(%, x = 1, y = -2, z = -1);
(d10)               [- 12, - 9, - 16]
```

Problem 5. See page 174.

```
(c11) a1:[x^2*z, -2*y^3*z^2, x*y^2*z]$
(c12) my_div(a1);
                      2  2              2
(d12)           - 6 y  z  + 2 x z + x y
(c13) ev(%, x = 1, y = -1, z = 1);
(d13)                   - 3
```

Problem 6. See page 174.

```
(c14) my_curl([x*z^3, -2*x^2*y*z, 2*y*z^4]);
                    4      2         2
(d14)          [2 z  + 2 x  y, 3 x z , - 4 x y z]
(c15) ev(%, x = 1, y = -1, z = 1);
(d15)               [0, 3, 4]
```

Problem 7. See page 174.

```
(c1) r:26.5$
(c2) ff(t, x, y, z):=-3.0*(x - y)$
(c3) gg(t, x, y, z):=-x*z + r*x - y$
(c4) hh(t, x, y, z):=x*y - z$
(c5) my_runge_kutta(f, g, h, t0, tfinal, x00, y00, z00, size):=
block([x_list:[], y_list:[], z_list:[],
       k1, k2, k3, k4, l1, l2, l3, l4, m1, m2, m3, m4, x0, y0, z0],
      x0:x00, y0:y00, z0:z00,
      for t:t0 thru tfinal step size do
      (x_list:endcons(x0, x_list),
       y_list:endcons(y0, y_list),
       z_list:endcons(z0, z_list),
       k1:size*apply(f, [t, x0, y0, z0]),
       l1:size*apply(g, [t, x0, y0, z0]),
       m1:size*apply(h, [t, x0, y0, z0]),
       k2:size*apply(f, [t + size/2, x0 + k1/2, y0 + l1/2, z0 + m1/2]),
       l2:size*apply(g, [t + size/2, x0 + k1/2, y0 + l1/2, z0 + m1/2]),
       m2:size*apply(h, [t + size/2, x0 + k1/2, y0 + l1/2, z0 + m1/2]),
```

```
 k3:size*apply(f, [t + size/2, x0 + k2/2, y0 + l2/2, z0 + m2/2]),
 l3:size*apply(g, [t + size/2, x0 + k2/2, y0 + l2/2, z0 + m2/2]),
 m3:size*apply(h, [t + size/2, x0 + k2/2, y0 + l2/2, z0 + m2/2]),
 k4:size*apply(f, [t + size, x0 + k3, y0 + l3, z0 + m3]),
 l4:size*apply(f, [t + size, x0 + k3, y0 + l3, z0 + m3]),
 m4:size*apply(f, [t + size, x0 + k3, y0 + l3, z0 + m3]),
 x0:x0 + (k1 + k4)/6 + (k2 + k3)/3,
 y0:y0 + (l1 + l4)/6 + (l2 + l3)/3,
 z0:z0 + (m1 + m4)/6 + (m2 + m3)/3),
[x_list, y_list, z_list])$
```

Index