DOE-Maxima Reference Manual

Michael Clarkson,

http://starship.python.net/crew/mike/

Version 5.9
Released August 2002

CONTENTS

Preface

This manual has been painfully and laboriously converted from the files in the manual/descrips directory of DOE-Maxima to a version of GNU Emacs TEXinfo format. If you find material that is incorrect, or needs further elaboration or clarification, please send me any corrections and I will gladly incorporate them into subsequent editions of this manual.

mailto:mike@python.net

Notational Conventions of this Manual

All **Maxima** functions, variables and keywords are set in UPPER-CASE TYPE. Function and variable definitions are set in slightly larger UPPER-CASE TYPE. Arguments to functions are set in *italic type*.

In general, arguments to functions are abbreviated as follows:

exp any expression

arg any argument

array an array

file a file name

float a floating point number

bfloat a big floating point number

fun a function

int an integer

list a list

mode a mode

poly a polynomial

prop a property

string a string

var a variable

tensor a tensor

Variables are usually followed by their default values as in the following example: FOOBAR default: [3]. This says that FOOBAR has a default value of 3, though the choice of square brackets is rather unfortunate as it is also **Maxima**'s representation for a list.

File Naming Conventions

The are a wide-variety of file naming conventions that have been used over the years with **Maxima**. In general, the following are more or less standard:

.dem demo file

.l lisp source code

.lsp lisp source code

.mac øMaxima source code

.mc øMaxima source code

.mil mail file

.o compiled binary code (øUnix)

.usg usage description

.xmp example

We will adopt the Unix naming conventions in this manual. Similarly, all file names have been converted to Unix format, relative to the top level **Maxima** directory.

CHAPTER

ONE

Fundamental Concepts

1.1 Beginning and Ending Maxima

An Init file is a file which is loaded automatically for you when you start up a **Maxima**, to customize **Maxima** for you. The init file defaults to the filename '`Maxima-init.l`' in the users home directory. It is possible to have an init file written as a BATCH file of **Maxima** commands. We hope this makes it easier for users to customize their **Maxima** environment. Here is an example init file

```
setup_autoload("share/bessel", j0, j1, jn);
showtime:all; comgrind:true;
```

SETUP_AUTOLOAD can be used to make functions in BATCH files autoloading, meaning that you can then use (for instance, here) the functions `J0`, `J1` and `Jn` from the BESSEL package directly because when you use the function the BESSEL package will be loaded in for you automatically. If the second file name or extension in the argument to SETUP_AUTOLOAD is not specified, then the standard search for second file names of `.o` `.l`, is done. See section 11.1.1 [Autoloading], page 164.

`QUIT` () [*Function*]

kills the current **Maxima**.

1.2 Entering Commands

A **Maxima** command is terminated by either a **$** or a **;**. In the first case the expression that is typed in is not displayed in its simplified form, and with the semi-colon the expression is displayed. See section 1.3 [Reader Syntax], page 3.

If an expression to be displayed is in CRE form or if it contains any subexpressions in CRE form, the symbol /R/ will follow the line label. The symbol `/`P/ follows the line label of Poisson Series expressions. The symbol `/`T/ follows the line label of Taylor Series expressions.

See section 4.2 [Canonical Rational Expressions], page 51.

1.2.1 Recalling Previous Expressions

% [*Variable*]

The last D-line computed by **Maxima** (whether or not it was printed out).

%% [*Variable*]

The value of the last computation performed while in a Maxima-BREAK. See section 12.5 [Break Points and Debugging], page 179. It also may be used in compound statements in the *n*'th statement to refer to the value of the *(n-1)*'th statement. See section 2.1.2 [Local Blocks and Variables], page 14.

```
F(N):=(INTEGRATE(X^N,X),
       SUBST(3,X,%%)-SUBST(2,X,%%));
```

is in essence equivalent to

```
F(N):=BLOCK([%%],
       %%:INTEGRATE(X^N,X),
       SUBST(3,X,%%)-SUBST(2,X,%%));
```

This will also work for communicating between the *(n-1)*th and *n*th (non-atomic) BLOCK statements. See section 2.1.2 [Local Blocks and Variables], page 14.

%TH (*i*) [*Function*]

is the *i*'th previous computation. That is, if the next expression to be computed is D(j) this is D(j-i). This is useful in BATCH files or for referring to a group of D expressions. For example, if SUM is initialized to 0 then FOR I:1 THRU 10 DO SUM:SUM+%TH(I); will set SUM to the sum of the last ten D expressions.

LABELS (*char*) [*Special Form*]

takes a *char* C, D, or E as an argument, and generates a list of all C-labels, D-labels, or E-labels, respectively. (If you've generated many E-labels via SOLVE, then FIRST(REST(LABELS(C))); reminds you what the last C-label was.) LABELS will take as argument any symbolic name, so if you have reset INCHAR, OUTCHAR, or LINECHAR, it will return the list of labels whose first character matches the first character of the arg you give to LABELS.

LABELS [*Variable*]

is a list of C, D, and E lines which are bound.

PLAYBACK (*arg*) [*Special Form*]

"plays back" input and output lines. *arg* can be one or more of:

n (a number) the last *n* expressions (Ci, Di, and Ei count as 1 each) are "played back," while if *arg* is omitted, all lines are. If a list of numbers is given, [m,n], then all lines with numbers from m to n inclusive are played-back. If m=n then [m] is sufficient for arg.

INPUT then only input lines are played back.

SLOW places PLAYBACK in a slow-mode similar to DEMO's (as opposed to the faster BATCH). This is useful in conjunction with SAVE or STRINGOUT when creating a secondary-storage file in order to pick out useful expressions.

TIME then the computation times are displayed as well as the expressions.

GCTIME or TOTALTIME then a complete breakdown of computation times are displayed, as with SHOWTIME:ALL;.

STRING "strings out" (see the STRING function) all input lines when playing back, rather than DISPLAYing them.

NOSTRING displays all input lines when playing back rather than STRINGing them.

GRIND "grind" mode can also be turned on (for processing input lines) (see GRIND).

One may include any number of options as in PLAYBACK([5,10],20,TIME,SLOW);.

1.3 Reader Syntax

Maxima statements are read line-by-line, and are terminated by a semicolon or a dollar sign. If the statement is terminated by a semicolon, the result is printed out.

```
(c1) 1 + 1;

(d1)         2
```

If the statement is terminated by a dollar sign, the result is calculated but not printed out.

```
(c2) 2 + 2$

(c3)
```

Note that although the result is not printed out, it is still assigned to the corresponding D label:

```
(c3) d2;

(d3)         4
```

Maximauses the standard mathematical operators:

+ for addition,

- for subtraction.

* for multiplication,

/ for division,

** for exponentiation,

. for non-commutative multiplication See section 6.2.2 [Non-Commutative Operations], page 90.

= defines an equation.

! for factorial,

!! for double factorial,

In addition, **Maxima** uses a quoting and evaluating operator which derive from its Lisp parenthood:

' (single quote) has the effect of preventing evaluation. E.g. 'F(X)) means do not evaluate the expression F(X). 'F(X) means return the noun form of F applied to [X] . See section 4.1.2 [Noun and Verb Forms], page 50.

'' (two single quotes) causes an extra evaluation to occur. E.g. ''c4; will re-execute line C4. ''F(X)) means evaluate the expression F(X) an extra time. ''F(X) means return the verb form of F applied to X. See section 4.1 [Evaluation], page 47.

See section 3.1 [Comparison Functions], page 33, for the inequality operators. See section 10.1.1 [Defining Operators], page 143, for how to define your own operators. See section 10.1.2 [Defining Macros], page 145, for how to define a macro.

1.3.1 Defining Variables

There are two assignment operators in **Maxima**, : and ::.

: [*Syntax*]

A:3 sets the variable A to 3.

:: [*Syntax*]

:: assigns the value of the expression on its right to the value of the quantity on its left, which must evaluate to an atomic variable or subscripted variable.

DEFINE_VARIABLE (*name, default-binding, mode, documentation*) [*Special Form*]

introduces a global variable into the **Maxima** environment. This is for user-written packages, which are often translated or compiled. E.g.: DEFINE_VARIABLE(FOO,TRUE,BOOLEAN); does the following:

1. MODE_DECLARE(FOO,BOOLEAN); sets it up for the translator.
2. If the variable is unbound, it sets it: FOO:TRUE.
3. DECLARE(FOO,SPECIAL); declares it special.
4. Sets up an assign property for it to make sure that it never gets set to a value of the wrong mode. E.g. FOO:44 would be an error once FOO is defined BOOLEAN.

See section 13.1 [Mode Declarations], page 185, for a list of the possible modes.

The optional 4'th argument is a documentation string. When TRANSLATE_FILE is used on a package which includes documentation strings, a second file is output in addition to the Lisp file which will contain the documentation strings, formatted suitably for use in manuals, usage files, or (for instance) DESCRIBE.

With any variable which has been DEFINE_VARIABLE'd with mode other than ANY, you can give a VALUE_CHECK property, which is a function of one argument called on the value the user is trying to set the variable to.

```
PUT('G5,LAMBDA([U],IF U#'G5 THEN ERROR("Don't set G5")),'VALUE\_CHECK); -
>
DEFINE_VARIABLE(G5,'G5,ANY_CHECK, "this ain't supposed to be set
by anyone but me.")
```

ANY_CHECK is a mode which means the same as ANY, but which keeps DEFINE_VARIABLE from optimizing away the assign property.

NUMERVAL (*var1*, *exp1*, *var2*, *exp2*, ...) [*Special Form*]

declares *vari* to have a numerical value of *expi*, which is evaluated and substituted for the variable in any expressions in which the variable occurs if the NUMER flag is TRUE.

See section 2.6 [Reviewing Options], page 29.

1.3.1.1 Assignment Flags

REFCHECK [*variable, default: FALSE*]

if TRUE causes a message to be printed each time a bound variable is used for the first time in a computation.

SETCHECK [*variable, default: FALSE*]

if set to a list of variables (which can be subscripted) will cause a printout whenever the variables, or subscripted occurrences of them, are bound (with : or :: or function argument binding). The printout consists of the variable and the value it is bound to. SETCHECK may be set to ALL or TRUE thereby including all variables. Note: No printout is generated when a SETCHECKed variable is set to itself, e.g. X:'X.

See section 2.4.1 [Display of Numbers], page 23.

1.3.2 Defining Functions

`:=` [*Syntax*]

To define a function in **Maxima** you use the := operator. E.g. `F`(X):=SIN(X) defines a function `F`.

`FUNDEF` (*fun*) [*Special Form*]

returns the function definition associated with *fun*. `FUNDEF`(fun); is similar to `DISPFUN`(fun); except that FUNDEF does not invoke DISPLAY.

`FUNMAKE` (*fun, [arg1, arg2, ...]*) [*Function*]

returns `f`un(arg1,...,argn) without calling the function *fun*.

`DEFINE` (*f(x1, ...), body*) [*Special Form*]

is equivalent to `f`(x1,...):=”body, but when used inside functions, it takes place at execution time rather than at the time of definition of the function which contains it.

`FUNCTIONS` [*Variable*]

a list of all user defined functions (set up by `f`(x) := ...).

1.4 Data Types

1.4.1 Constants

1.4.1.1 Logical Constants

TRUE the Boolean constant, true. (`T` in Lisp)

FALSE the Boolean constant, false. (`NIL` in Lisp)

1.4.1.2 Arithmetic Constants

INF real positive infinity.

INFINITY complex infinity, an infinite magnitude of arbitrary phase angle.

MINF real minus infinity.

%PI The **Maxima** representation for pi.

%E The **Maxima** representation for the base of natural logarithms

%I The **Maxima** representation for the square root of -1. Some commands and switches which handle %I are, LOGNEGINT, REALPART, IMAGPART, RECTFORM, POLARFORM, ABS, CARG, and CABS. See section 3.6 [Complex Variables], page 44.

%GAMMA The Euler-Mascheroni constant. The notation %GAMMA is used for consistency with standard texts which use the Greek letter gamma; it can be defined as follows:

$$\%GAMMA = \lim_{M \to \infty} \left(\sum_{N=1}^{M} \frac{1}{N} - \log(M) \right)$$

Currently %GAMMA has a NUMER property of .577215665 and a CONSTANT property. It is used along with the Polygamma simplification routines to permit their reduction to closed forms.

%PHI The constant (SQRT(5)+1)/2 = 1.618033989). If you want the Rational Function Package to know about %PHI do TELLRAT(%PHI**2-%PHI-1); ALGEBRAIC:TRUE. See also FIBTOPHI.

1.4.2 Data Type Predicates

`ATOM` (*exp*) [*Function*]

is TRUE if *exp* is atomic (i.e. a number or a name), otherwise FALSE. Thus ATOM(5) is TRUE, while ATOM(A[1]) and ATOM(SIN(X)) are FALSE (assuming A[1] and X are unbound).

`LISTP` (*exp*) [*Function*]

is TRUE if *exp* is a list, otherwise FALSE.

`MATRIXP` (*exp*) [*Function*]

is TRUE if *exp* is a matrix, otherwise FALSE.

`NUMBERP` (*exp*) [*Function*]

is TRUE if *exp* is an integer, a rational number, a floating point number or a bigfloat, otherwise FALSE.

`SYMBOLP` (*exp*) [*Function*]

returns TRUE if *exp* is a symbol or name, else FALSE. I.e., in effect, SYMBOLP(X):=ATOM(X) AND NOT NUMBERP(X).

`UNKNOWN` (*exp*) [*Function*]

returns TRUE iff *exp* contains an operator or function not known to the built-in simplifier.

1.4.3 Numerical Predicates

BFLOATP (*exp*) [*Function*]

is TRUE if *exp* is a bigfloat number, else FALSE.

CONSTANTP (*exp*) [*Function*]

is TRUE if *exp* is a constant (i.e. composed of only of numbers and %PI, %E, %I, or any variables that have been either bound to a constant or DECLAREd constant); otherwise FALSE. Any function whose arguments are constant is also considered to be a constant.

EVENP (*exp*) [*Function*]

is TRUE if *exp* is an even integer. FALSE is returned in all other cases.

FLOATNUMP (*exp*) [*Function*]

is TRUE if *exp* is a floating point number, otherwise FALSE.

INTEGERP (*exp*) [*Function*]

is TRUE if *exp* is an integer, otherwise FALSE.

NONSCALARP (*exp*) [*Function*]

is TRUE if *exp* is a non-scalar, i.e. it contains atoms declared as non-scalars, lists, or matrices.

ODDP (*exp*) [*Function*]

is TRUE if *exp* is an odd integer. FALSE is returned in all other cases.

PRIMEP (*int*) [*Function*]

returns TRUE if *int* is a prime, FALSE if not.

RATNUMP (*exp*) [*Function*]

is TRUE if *exp* is a rational number, including integers, otherwise FALSE.

RATP (*exp*) [*Function*]

is TRUE if *exp* is in CRE form or extended CRE form, otherwise FALSE.

SCALARP (*exp*) [*Function*]

is TRUE if *exp* is a number, constant, or variable DECLAREd SCALAR, or composed entirely of numbers, constants, and such variables, but not containing matrices or lists.

SUBVARP (*exp*) [*Function*]

is TRUE if *exp* is a subscripted variable, for example A[I].

TAYLORP (*exp*) [*Function*]

a predicate function which returns TRUE if and only if the expression *exp* is in Taylor Series representation.

1.4.4 Data Type Coercion

BFLOAT (*X*) [*Function*]

converts all numbers and functions of numbers to bigfloat numbers. Setting FPPREC default: [16] to *N*, sets the bigfloat precision to *N* digits. If FLOAT2BF default: [FALSE] is FALSE a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

ENTIER (*float*) [*Function*]

largest integer <= *float* where *float* is numeric. FIX (as in FIXnum) is a synonym for this, so FIX(X); is precisely the same.

FIX (*float*) [*Function*]

a synonym for ENTIER. Returns the largest integer <= *float*, where *float* is numeric.

FLOAT (*exp*) [*Function*]

converts integers, rational numbers and bigfloats in *exp* to floating point numbers.

REALPART (*exp*) [*Function*]

gives the real part of *exp*. REALPART and IMAGPART will work on expressions involving trigonometric and hyperbolic functions, as well as SQRT, LOG, and exponentiation.

IMAGPART (*exp*) [*Function*]

returns the imaginary part of *exp*.

1.4.5 Control Characters

Control characters are typed by holding down the key labeled CONTROL or CTRL and typing the indicated letter, much the same way the SHIFT key is used.

1.5 Command Line Flags

See section 2.4 [Flags Effecting the Displayed Form], page 21.

LABELS [*Variable*]

all bound C, D, and E labels.

NOLABELS [*variable, default: FALSE*]

if TRUE then no labels will be bound except for E lines generated by the SOLVE functions. This is most useful in the BATCH mode where it eliminates the need to do KILL(LABELS); in order to free up storage.

BOTHCASES [*variable, default: FALSE*]

if TRUE will cause **Maxima** to retain lower case text as well as upper case. Note, however, that the names of any **Maxima** special variables or functions must be typed in upper case. (For historical reasons this is also a function, but it should be used as a switch. Please use it as described here, not as a function.)

TTYOFF [*variable, default: FALSE*]

if TRUE stops printing output to the console.

LINEL [*Variable*]

the number of characters which are printed on a line. It is initially set by **Maxima** to the line length of the type of terminal being used (as far as is known) but may be reset at any time by the user.

LINENUM [*Variable*]

is the line number of the last expression.

LINEDISP [*variable, default: TRUE*]

Allows the use of line graphics in the drawing of equations on those systems which support them. This can be disabled by setting LINEDISP to FALSE. It is automatically disabled during WRITEFILE.

STARDISP [*variable, default: FALSE*]

if TRUE will cause multiplication to be displayed explicitly with an * between operands.

CURSORDISP [*variable, default: TRUE*]

If TRUE, causes expressions to be drawn by the displayer in logical sequence. This only works with a console which can do cursor movement, such as a vt100. If FALSE, expressions are simply printed line by line. CURSORDISP is FALSE when a WRITEFILE is in effect.

VERBOSE [*variable, default: FALSE*]

if TRUE will cause comments about the progress of some functions to be printed as the execution of it proceeds. One such function is POWERSERIES.

ABSBOXCHAR [*variable, default: !*]

is the character used to draw absolute value signs around expressions which are more than a single line high.

PAGEPAUSE [*Variable*]

This is set by **Maxima** according to what the system knows about your terminal type. If it is set to TRUE, then "more processing," which involves the printing of --More display?– or --Pause– at the bottom of your screen, or after so many lines on a printing terminal, and pausing, will be enabled. It may be set to FALSE to turn off the more processing on a display terminal. PAGEPAUSE is sometimes useful in batch files on slow lines where you just wish to watch the output run past, and can keep up with the line speed well enough.

MOREWAIT [*variable, default: FALSE*]

Controls the action of more processing. When output is suspended and a --Pause– or --More Display?– prompt is issued, one may type a space to continue the output. Typing any character other than space or rubout (delete) will continue the output, and leave the character around to be read as part of the next c-line and possibly intervening --Pause– prompts.

INCHAR [*variable, default: C*]

the alphabetic prefix of the names of expressions typed by the user.

OUTCHAR [*variable, default: D*]

the alphabetic prefix of the names of outputted expressions.

LINECHAR [*variable, default: E*]

the alphabetic prefix of the names of intermediate displayed expressions.

CHAPTER

TWO

Programming Constructs

2.1 Program Flow

2.1.1 Conditionals

`IF` [*Syntax*]

The IF statement is used for conditional execution. The syntax is:

```
IF condition THEN expression1 ELSE expression2;
```

The result of an IF statement is *expression1* if condition is TRUE and *expression2* if it is FALSE. *expression1* and *expression2* are any **Maxima** expressions (including nested IF statements), and *condition* is an expression which evaluates to TRUE or FALSE, and is composed of relational and logical operators which are as follows:

`=` equal to, relational infix

EQUAL equal to, relational infix

`#` not equal to, relational infix

`>` greater than, relational infix

`<` less than, relational infix

`>=` greater than or equal to, relational infix

`<=` less than or equal to, relational infix

AND and, logical infix

OR or, logical infix

NOT not, logical infix

2.1.2 Local Blocks and Variables

BLOCK (*[v1,...,vk], statement1, ...*) [*Syntax*]

Blocks in **Maxima** are somewhat analogous to subroutines in FORTRAN or procedures in ALGOL or PL/I. Blocks are like compound statements but also enable the user to label statements within the block and to assign dummy variables to values which are local to the block. The *vi* are variables which are local to the BLOCK and the *statementi* are any **Maxima**expressions. If no variables are to be made local then the list may be omitted. A block uses these local variables to avoid conflict with variables having the same names used outside of the block (i.e. global to the block). In this case, upon entry to the block, the global values are saved onto a stack and are inaccessible while the block is being executed. The local variables then are unbound so that they evaluate to themselves. They may be bound to arbitrary values within the block but when the block is exited the saved values are restored to these variables. The values created in the block for these local variables are lost. Where a variable is used within a block and is not in the list of local variables for that block it will be the same as the variable used outside of the block.

If it is desired to save and restore other local properties besides VALUE, for example ARRAY (except for complete arrays), FUNCTION, DEPENDENCIES, ATVALUE, MATCHDECLARE, ATOMGRAD, CONSTANT, and NONSCALAR, then the function LOCAL should be used inside of the block, with arguments being the names of the variables.

GO [*Syntax*]

The function GO may be used to transfer control to the statement of the block that is tagged with the argument to GO. To tag a statement, precede it by an atomic argument as another statement in the BLOCK. For example:

```
BLOCK( [X],
       X:1,
       LOOP,
              X:X+1,
              \dots ,
              GO(LOOP),
       \dots);
```

The argument to GO must be the name of a tag appearing within the BLOCK. One cannot use GO to transfer to a tag in a BLOCK other than the one containing the GO.

The value of the block is the value of the last statement, or the value of the argument to the function RETURN which may be used to exit explicitly from the block. Blocks typically appear on the right side of a function definition, but can be used in other places as well.

DISPFLAG [*variable, default: TRUE*]

if set to FALSE within a BLOCK will inhibit the display of output generated by the SOLVE functions called from within the BLOCK. Termination of the BLOCK with a dollar sign, $, sets DISPFLAG to FALSE.

LOCAL (*var1, var2, ...*) [*Special Form*]

causes the variables *vari* to be local with respect to all the properties in the statement in which this function is used. LOCAL may only be used in BLOCKs, in the body of function definitions, in LAMBDA expressions, or in the EV function and only one occurrence is permitted in each. LOCAL is independent of CONTEXT.

RETURN (*exp*) [*Syntax*]

may be used to exit explicitly from a BLOCK, returning its argument.

2.1.3 Throw and Catch

CATCH (*exp1, ..., expn*) [*Special Form*]

evaluates its arguments one by one; if the structure of the *expi* leads to the evaluation of an expression of the form THROW(arg), then the value of the CATCH is the value of THROW(arg). This non-local return thus goes through any depth of nesting to the nearest enclosing CATCH. There must be a CATCH corresponding to a THROW, else an error is generated. If the evaluation of the *expi* does not lead to the evaluation of any THROW, then the value of the CATCH is the value of *expn*.

```
(C1) G(L) := CATCH(
               MAP( LAMBDA([X],
                      IF X<0 THEN THROW(X) ELSE F(X)), L)
               )$
(C2) G([1,2,3,7]);
(D2)                    [F(1), F(2), F(3), F(7)]
(C3) G([1,2,-3,7]);
(D3)                                - 3
```

The function G returns a list of F applied to each element of L, if L consists only of non-negative numbers; otherwise, G “catches” the first negative element of L and "throws" it up.

THROW (*exp*) [*Function*]

evaluates *exp* and throws the value back to the most recent CATCH. THROW is used with CATCH as a structured nonlocal exit mechanism.

ERRCATCH (*exp1, exp2, ...*) [*Special Form*]

evaluates its arguments one by one and returns a list of the value of the last one if no error occurs. If an error occurs in the evaluation of any arguments, ERRCATCH “catches” the error, and immediately returns [] (the empty list). This function is useful in BATCH files where one suspects an error might occur which would otherwise have terminate the BATCH if the error weren’t caught.

2.1.4 Iteration

`DO` [*Syntax*]

The DO statement is used for performing iteration. Due to its great generality the DO statement will be described in two parts. First the usual form will be given which is analogous to that used in several other programming languages (FORTRAN, ALGOL, PL/I, etc.); then the other features will be mentioned.

There are three variants of the first form, the **FOR** statement, that differ only in their terminating conditions. They are:

```
(a)  FOR variable : initial-value STEP increment
      THRU limit DO body
(b)  FOR variable : initial-value STEP increment
      WHILE condition DO body
(c)  FOR variable : initial-value STEP increment
      UNLESS condition DO body
```

Alternatively, the **STEP** may be given after the termination condition or limit. The *initial-value*, *increment*, *limit*, and *body* can be any expressions. If the increment is 1 then STEP 1 may be omitted.

The execution of the DO statement proceeds by first assigning the initial-value to the variable (henceforth called the control-variable). Then:

1. If the control-variable has exceeded the limit of a **THRU** specification, or if the condition of the **UNLESS** is TRUE, or if the condition of the **WHILE** is FALSE then the DO terminates.
2. The body is evaluated.
3. The increment is added to the control-variable.

The process from (1) to (3) is performed repeatedly until the termination condition is satisfied. One may also give several termination conditions in which case the DO terminates when any of them is satisfied.

In general the THRU test is satisfied when the control-variable is greater than the limit if the increment was non-negative, or when the control-variable is less than the limit if the increment was negative. The increment and limit may be non-numeric expressions as long as this inequality can be determined. However, unless the increment is syntactically negative (e.g. is a negative number) at the time the DO statement is input, **Maxima** assumes it will be positive when the DO is executed. If it is not positive, then the DO may not terminate properly.

Note that the limit, increment, and termination condition are evaluated each time through the loop. Thus if any of these involve much computation, and yield a result that does not change during all the executions of the body, then it is more efficient to set a variable to their value prior to the DO and use this variable in the DO form.

The value normally returned by a DO statement is the atom DONE, as every statement in **Maxima** returns a value. However, the function RETURN may be used inside the body to exit the DO prematurely and give it any desired value. Note however that a RETURN within a DO that occurs in a BLOCK will exit only the DO

and not the BLOCK. Note also that the GO function may not be used to exit from a DO into a surrounding BLOCK.

The control variable is always local to the DO and thus any variable may be used without affecting the value of a variable with the same name outside of the DO. The control-variable is unbound after the DO terminates.

```
(C1)   FOR A:-3 THRU 26 STEP 7 DO LDISPLAY(A)$
(E1)          A = -3
(E2)          A =  4
(E3)          A = 11
(E4)          A = 18
(E5)          A = 25
```

The function LDISPLAY generates intermediate labels; DISPLAY does not.

```
(C6)   S:0$
(C7)   FOR I:1 WHILE I<=10 DO S:S+I;
(D7)          DONE
(C8)   S;
(D8)          55
```

Note that the condition in C7 is equivalent to UNLESS I > 10 and also THRU 10.

```
(C9)   SERIES:1$
(C10)  TERM:EXP(SIN(X))$
(C11)  FOR P:1 UNLESS P>7 DO
          (TERM:DIFF(TERM,X)/P,
          SERIES:SERIES+SUBST(X=0,TERM)*X^P)$
(C12)   SERIES;
                 7    6     5    4    2
(D12)           X    X     X    X    X
                -- - --- - -- - -- + -- + X + 1
                96   240   15   8    2
```

which gives 8 terms of the Taylor series for %e**sin(x).

```
(C13) POLY:0$
(C14) FOR I:1 THRU 5 DO
        FOR J:I STEP -1 THRU 1 DO
           POLY:POLY+I*X^J$
(C15) POLY;
                5      4       3       2
(D15)      5 X  + 9 X  + 12 X  + 14 X  + 15 X
(C16) GUESS:-3.0$
(C17) FOR I:1 THRU 10 DO (GUESS:SUBST(GUESS,X,.5*(X+10/X)),
         IF ABS(GUESS^2-10)<.00005 THEN RETURN(GUESS));
(D17)                  - 3.1622807
```

This example computes the negative square root of 10 using the Newton-Raphson iteration a maximum of 10 times. Had the convergence criterion not been met the value returned would have been DONE.

2.1.4.1 Additional Forms of the DO Statement

Instead of always adding a quantity to the control-variable one may sometimes wish to change it in some other way for each iteration. In this case one may use **NEXT** *expression* instead of STEP *increment*. This will cause the control-variable to be set to the result of evaluating expression each time through the loop.

```
(C1)  FOR COUNT:2 NEXT 3*COUNT THRU 20
         DO DISPLAY(COUNT)$
                    COUNT = 2
                    COUNT = 6
                    COUNT = 18
```

As an alternative to FOR variable:value ... DO ..., the syntax FOR *variable* **FROM** *value* ... DO ... may be used. This permits the FROM value to be placed after the step or next value or after the termination condition. If FROM value is omitted then 1 is used as the initial value.

Sometimes one may be interested in performing an iteration where the control-variable is never actually used. It is thus permissible to give only the termination conditions omitting the initialization and updating information as in the following example to compute the square-root of 5 using a poor initial guess.

```
(C1) X:1000
(C2)  THRU 10 WHILE X#0.0 DO X:.5*(X+5.0/X)$
(C3) X;
(D3)               2.236068
```

If it is desired one may even omit the termination conditions entirely and just give DO *body* which will continue to evaluate the body indefinitely. In this case the function RETURN should be used to terminate execution of the DO.

```
(C1) NEWTON(F,GUESS):=
  BLOCK([NUMER,Y],
        LOCAL(DF),
        NUMER:TRUE,
        DEFINE(DF(X),DIFF(F(X),X)),
        DO (Y:DF(GUESS),
            IF Y=0.0 THEN ERROR("Derivative at:",GUESS," is zero."),
            GUESS:GUESS-F(GUESS)/Y,
            IF ABS(F(GUESS))<5.0E-6 THEN RETURN(GUESS)))$
(C2) SQR(X):=X^2-5.0$
(C3) NEWTON(SQR,1000);
(D3)                    2.236068
```

Note that RETURN, when executed, causes the current value of GUESS to be returned as the value of the

DO. The BLOCK is exited and this value of the DO is returned as the value of the BLOCK because the DO is the last statement in the block.

One other form of the DO is available in **Maxima**, the **FOR** ... **IN** statement. The syntax is:

```
FOR variable IN list [end-tests] DO body
```

The members of the list are any expressions which will successively be assigned to the variable on each iteration of the body. The optional end-tests can be used to terminate execution of the DO; otherwise it will terminate when the list is exhausted or when a RETURN is executed in the body. (In fact, *list* may be any non-atomic expression, and successive parts are taken.)

```
(C1)  FOR F IN [LOG, RHO, ATAN] DO LDISP(F(1))$
(E1)                                    0
(E2)                                  RHO(1)
                                       %PI
(E3)                                   ---
                                        4
(C4) EV(E3,NUMER);
(D4)                             0.78539816
```

2.2 Reading Input

READ (*string1*, ...) [*Function*]

prints its arguments, then reads in and evaluates one expression. For example: A:READ("ENTER THE NUMBER OF VALUES").

READONLY (*string1*, ...) [*Function*]

prints its arguments, then reads in an expression (which in contrast to READ is not evaluated).

See section 11.1 [Loading Files], page 163, for information on reading in files.

2.3 Displaying Expressions

DISPLAY (*exp1*, *exp2*, ...) [*Special Form*]

displays equations whose left side is *expi* unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and FOR statements in order to have intermediate results displayed. The arguments to DISPLAY are usually atoms, subscripted variables, or function calls. See also the DISP function.

```
(C1) DISPLAY(B[1,2]);
                                    2
                          B     = X - X
                           1, 2
(D1)                              DONE
```

LDISPLAY (*exp1, exp2, ...*) [*Special Form*]

is like DISPLAY but also generates intermediate labels.

DISP (*exp1, exp2, ...*) [*Function*]

is like DISPLAY but only the value of the arguments are displayed rather than equations. This is useful for complicated arguments which don't have names or where only the value of the argument is of interest and not the name.

LDISP (*exp1, exp2, ...*) [*Function*]

is like DISP but also generates intermediate labels.

DISPFUN (*f1, f2, ...*) [*Special Form*]

displays the definition of the user defined functions *fi* which may also be the names of array associated functions, subscripted functions, or functions with constant subscripts, which are the same as those used when the functions were defined. DISPFUN(ALL); will display all user defined functions as given on the FUNCTIONS and ARRAYS lists, except subscripted functions with constant subscripts. E.g. if the user has defined a function F(x), DISPFUN(F); will display the definition.

DISPFORM (*exp*) [*Function*]

returns the external representation of *exp* with respect to its main operator. This should be useful in conjunction with PART which also deals with the external representation. Suppose *exp* is -A. Then the internal representation of *exp* is "*"(-1,A), while the external representation is "-"(A)

DISPFORM(exp,ALL) converts the entire expression (not just the top-level) to external format.

PRINT (*exp1, exp2, ...*) [*Function*]

evaluates and displays its arguments one after the other on a line starting at the leftmost position. If *expi* is unbound, or is preceded by a single quote, or is enclosed in double quotes, then it is printed literally. For example, PRINT("THE VALUE OF X IS",X). The value returned by PRINT is the value of its last argument. No intermediate lines are generated. (For printing files, see the PRINTFILE function.)

`GRIND` (*exp*) [*Special Form*]

prints out *exp* in a more readable format than the STRING command. It returns a D-line as value.

`GRIND` [*variable, default: FALSE*]

if TRUE will cause the STRING, STRINGOUT, and PLAYBACK commands to use "grind" mode instead of "string" mode. For PLAYBACK, "grind" mode can also be turned on (for processing input lines) by specifying GRIND as an option.

`STRING` (*exp*) [*Special Form*]

converts *exp* to **Maxima**'s linear notation (similar to FORTRAN's) just as if it had been typed in and puts *exp* into the buffer for possible editing (in which case *exp* is usually Ci). The STRING'ed expression should not be used in a computation.

2.4 Flags Effecting the Displayed Form

`NEGDISTRIB` [*variable, default: TRUE*]

when TRUE allows -1 to be distributed over an expression. E.g. -(X+Y) becomes -Y-X. Setting it to FALSE will allow -(X+Y) to be displayed like that. This is sometimes useful but be very careful: like the SIMP flag, this is one flag you do not want to set to FALSE as a matter of course, or necessarily for other than local use in your **Maxima**.

`NEGSUMDISPFLAG` [*variable, default: TRUE*]

when TRUE, X-Y displays as X-Y instead of as -Y+X. Setting it to FALSE causes the special check in display for the difference of two expressions to not be done. One application is that thus A+%I*B and A-%I*B may both be displayed the same way.

`DISPLAY2D` [*variable, default: TRUE*]

if set to FALSE will cause the standard display to be a string (1-dimensional) form rather than a display (2-dimensional) form. This may be of benefit for users on printing consoles who would like to conserve paper.

`DISPLAY_FORMAT_INTERNAL` [*variable, default: FALSE*]

if set to TRUE will cause expressions to be displayed without being transformed in ways that hide the internal mathematical representation. The display then corresponds to what the INPART command returns rather than the PART command. Examples:

```
User     PART       INPART
a-b;      A - B     A + (- 1) B

           A            - 1
a/b;       -         A B
           B
                      1/2
sqrt(x);   SQRT(X)    X

          4 X        4
X*4/3;    ---        - X
           3         3
```

See section 4.6 [Substituting Expressions], page 67.

DOMAIN [*variable, default: REAL*]

if set to COMPLEX, SQRT(X^2) will remain SQRT(X^2) instead of returning ABS(X). The notion of a domain of simplification is still in its infancy, and controls little more than this at the moment.

EXPOP [*variable, default: 0*]

the highest positive exponent which is automatically expanded. Thus (X+1)**3, when typed, will be automatically expanded only if EXPOP is greater than or equal to 3. If it is desired to have (X+1)**n expanded where *n* is greater than EXPOP, then executing EXPAND((X+1)**n) will work only if MAXPOSEX is not less than *n*.

EXPON [*variable, default: 0*]

the exponent of the largest negative power which is automatically expanded (independent of calls to EXPAND). For example if EXPON is 4 then (X+1)**(-5) will not be automatically expanded.

PROGRAMMODE [*variable, default: TRUE*]

when FALSE will cause SOLVE, REALROOTS, ALLROOTS, and LINSOLVE to print E-labels (intermediate line labels) to label answers. When TRUE, SOLVE, etc. return answers as elements in a list. (Except when BACKSUBST is set to FALSE, in which case PROGRAMMODE:FALSE is also used.)

NOUNDISP [*variable, default: FALSE*]

if TRUE will cause NOUNs to display with a single quote. This switch is always TRUE when displaying function definitions.

POWERDISP [*variable, default: FALSE*]

if TRUE will cause sums to be displayed with their terms in the reverse order. Thus polynomials would display as truncated power series, i.e., with the lowest power first.

SQRTDISPFLAG [*variable, default: TRUE*]

if FALSE causes SQRT to display with exponent 1/2.

2.4.1 Display of Numbers

IBASE [*variable, default: 10*]

the base for inputing numbers.

OBASE [*variable, default: 10*]

the base for display of numbers.

KEEPFLOAT [*variable, default: FALSE*]

if set to TRUE will prevent floating point numbers from being rationalized when expressions which contain them are converted to CRE form.

FLOAT2BF [*variable, default: FALSE*]

if FALSE, a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

RATPRINT [*variable, default: TRUE*]

if FALSE suppresses the printout of the message informing the user of the conversion of floating point numbers to rational numbers.

RATEPSILON [*variable, default: 2.0E-8*]

the tolerance used in the conversion of floating point numbers to rational numbers.

FPPREC [*variable, default: 16*]

Floating Point PRECision. Can be set to an integer representing the desired precision.

FPPRINTPREC [*variable, default: 0*]

The number of digits to print when printing a bigfloat number, making it possible to compute with a large number of digits of precision, but have the answer printed out with a smaller number of digits. If FPPRINTPREC is 0 (the default), or >= FPPREC, then the value of FPPREC controls the number of digits used for printing. However, if FPPRINTPREC has a value between 2 and FPPREC-1, then it controls the number of digits used. The minimal number of digits used is 2, one to the left of the point and one to the right. The value 1 for FPPRINTPREC is illegal.

PFEFORMAT [*variable, default: FALSE*]

if TRUE will cause rational numbers to display in a linear form and denominators which are integers to display as rational number multipliers.

BFTORAT [*variable, default: FALSE*]

controls the conversion of a bigfloat number to a rational numbers. If BFTORAT:FALSE, RATEPSILON will be used to control the conversion (this results in relatively small rational numbers). If BFTORAT:TRUE, the rational number generated will accurately represent the bigfloat.

BFTRUNC [*variable, default: TRUE*]

causes trailing zeroes in non-zero bigfloat numbers not to be displayed. Thus, if BFTRUNC:FALSE, BFLOAT(1); displays as 1.000000000000000B0. Otherwise, this is displayed as 1.0B0.

ZUNDERFLOW [*variable, default: TRUE*]

if FALSE, an error will be signaled if floating point underflow occurs.

2.4.2 Display of Exponentials

DEMOIVRE [*variable, default: FALSE*]

if TRUE will cause %E**(A+B*%I) to become %E**A*(COS(B)+%I*SIN(B)) if B is free of %I. A and B are not expanded. (DEMOIVRE:TRUE; is the way to reverse the effect of EXPONENTIALIZE:TRUE;)

DEMOIVRE(exp) will cause the conversion without setting the switch or having to re-evaluate the expression with EV.

%EDISPFLAG [*variable, default: FALSE*]

if TRUE, **Maxima** displays %E to a negative exponent as a quotient, i.e. %E**-X as 1/%E**X.

%EMODE [*variable, default: TRUE*]

when TRUE %E**(%PI*%I*X) will be simplified as follows: it will become COS(%PI*X)+%I*SIN(%PI*X) if X is an integer or a multiple of 1/2, 1/3, 1/4, or 1/6 and thus will simplify further. For other numerical X it will become %E**(%PI*%I*Y) where Y is X-2*k for some integer k such that ABS(Y)<1. If %EMODE is FALSE no simplification of %E**(%PI*%I*X) will take place.

%ENUMER [*variable, default: FALSE*]

when TRUE will reliably cause %E to be converted into 2.718... whenever NUMER is TRUE. The default is that this conversion will take place only if the exponent in %E**X evaluates to a number.

`%E_TO_NUMLOG` [*variable, default: FALSE*]

when set to TRUE, for *r* some rational number, and *x* some expression, %E**(r*LOG(x)) will be simplified into x**r.

`EXPTDISPFLAG` [*variable, default: TRUE*]

if TRUE, **Maxima** displays expressions with negative exponents using quotients e.g., X**(-1) as 1/X.

`RADEXPAND` [*variable, default: TRUE*]

if set to ALL will cause *n*'th roots of factors of a product which are powers of *n* to be pulled outside of the radical. E.g. if RADEXPAND is ALL, SQRT(16*X**2) will become 4*X. More particularly, consider SQRT(X**2):

1. If RADEXPAND is ALL or ASSUME(X>0) has been done, SQRT(X**2) will become X.
2. If RADEXPAND is TRUE and DOMAIN is REAL (its default), SQRT(X**2) will become ABS(X).
3. If RADEXPAND is FALSE, or RADEXPAND is TRUE and DOMAIN is COMPLEX, SQRT(X**2) will be returned.

The notion of DOMAIN with settings of REAL or COMPLEX is still in its infancy. Note that its setting here only matters when RADEXPAND is TRUE.

See section 4.6.1 [Substitution Flags], page 68, for flags that effect the substitution of expressions.

2.4.3 Display of Logarithms

`LOGEXPAND` [*variable, default: TRUE*]

causes LOG(A**B) to become B*LOG(A). If it is set to ALL, LOG(A*B) will also simplify to LOG(A)+LOG(B). If it is set to SUPER, then LOG(A/B) will also simplify to LOG(A)-LOG(B) for rational numbers a/b, a#1. (LOG(1/B) for B an integer, always simplifies.) If it is set to FALSE, all of these simplifications will be turned off.

`LOGNEGINT` [*variable, default: FALSE*]

if TRUE implements the rule LOG(-n) -> LOG(n)+%i*%pi for *n* a positive integer.

`LOGNUMER` [*variable, default: FALSE*]

if TRUE then negative floating point arguments to LOG will always be converted to their absolute value before the LOG is taken. If NUMER is also TRUE, then negative integer arguments to LOG will also be converted to their absolute value.

LOGSIMP [*variable, default: TRUE*]

if FALSE then no simplification of %E to a power containing LOG's is done.

LOGARC [*variable, default: FALSE*]

if TRUE will cause the inverse circular and hyperbolic functions to be converted into logarithmic form.

LOGARC (*exp*) [*Function*]

will cause this conversion for a particular expression without setting the switch or having to re-evaluate the expression with EV.

See section 2.4.2 [Display of Exponentials], page 24, for the definition of %E_TO_NUMLOG.

2.4.4 Display of Trig Functions

TRIGSIGN [*variable, default: TRUE*]

if TRUE permits simplification of negative arguments to trigonometric functions. E.g., SIN(-X) will become -SIN(X) only if TRIGSIGN is TRUE.

TRIGINVERSES [*variable, default: ALL*]

controls the simplification of the composition of trig and hyperbolic functions with their inverse functions: If ALL, both e.g. ATAN(TAN(X)) and TAN(ATAN(X)) simplify to X. If TRUE, the *arcfun(fun(x))* simplification is turned off. If FALSE, both the *arcfun(fun(x))* and *fun(arcfun(x))* simplifications are turned off.

HALFANGLES [*variable, default: FALSE*]

if TRUE causes half-angles to be simplified away.

See section 5.2.5.1 [Trig Expand Flags], page 80, for the definition of TRIGEXPAND, TRIGEXPANDPLUS, TRIGEXPANDTIMES,

EXPONENTIALIZE [*variable, default: FALSE*]

if TRUE will cause all circular and hyperbolic functions to be converted to exponential form. (Setting DEMOIVRE:TRUE; will reverse the effect.)

EXPONENTIALIZE(exp) will cause the conversion to exponential form of an expression without setting the switch or having to re-evaluate the expression with EV.

2.4.5 Display of Sums

See section 7.1.1.1 [Sum Flags], page 102, for the definition of SIMPSUM.

SUMEXPAND [*variable, default: FALSE*]

if TRUE, products of sums and exponentiated sums are converted into nested sums.

```
SUMEXPAND:TRUE$
SUM(F(I),I,0,M)*SUM(G(J),J,0,N); ->
                 'SUM('SUM(F(I1)*G(I2),I2,0,N),I1,0,M)
SUM(F(I),I,0,M)^2; -> 'SUM('SUM(F(I3)*F(I4),I4,0,M),I3,0,M)
```

If FALSE, they are left alone. See also CAUCHYSUM.

CAUCHYSUM [*variable, default: FALSE*]

When multiplying together sums with INF as their upper limit, if SUMEXPAND is TRUE and CAUCHYSUM is set to TRUE then the Cauchy product will be used rather than the usual product. In the Cauchy product, the index of the inner summation is a function of the index of the outer one, rather than varying independently. That is: SUM(F(I),I,0,INF)*SUM(G(J),J,0,INF) becomes SUM(SUM(F(I)*G(J-I),I,0,J),J,0,INF).

SUMHACK [*variable, default: FALSE*]

if set to TRUE then SUM(F(I),I,3,1); will yield -F(2), by the identity SUM(F(I),I,A,B) = -SUM(F(I),I,B+1,A-1) when A>B.

2.4.6 Display of Products

PRODHACK [*variable, default: FALSE*]

if set to TRUE then PRODUCT(F(I),I,3,1); will yield 1/F(2), by the identity PRODUCT(F(I),I,A,B) = 1/PRODUCT(F(I),I,B+1,A-1) when A>B.

2.4.7 Display of Simplification

ALGEBRAIC [*variable, default: FALSE*]

must be set to TRUE in order for the simplification of algebraic integers to take effect.

LISTARITH [*variable, default: TRUE*]

if FALSE causes any arithmetic operations with lists to be suppressed; when TRUE, list-matrix operations are contagious causing lists to be converted to matrices yielding a result which is always a matrix. However, list-list operations should return lists.

2.4.8 Display of Factoring

See section 5.3.1 [Factor Flags], page 83, for the definition of FACTORFLAG, DONTFACTOR, SAVEFACTORS, INTFACLIM, BERLEFACT and NEWFAC.

2.4.9 Display of Expansion

See section 5.2.1 [Expand Flags], page 77.

2.5 Ordering of the Display

`ORDERGREAT` (*V1, . . . , Vn*) [*Special Form*]

sets up aliases for the variables *V1, . . . , Vn* such that *V1* > *V2* > . . . > *Vn* > *any other variable not mentioned as an argument*.

`ORDERGREATP` (*exp1, exp2*) [*Function*]

returns TRUE if *exp2* precedes *exp1* in the set up with the ORDERGREAT function.

`ORDERLESS` (*V1, . . . , Vn*) [*Special Form*]

sets up aliases for the variables *V1, . . . , Vn* such that *V1* < *V2* < . . . < *Vn* < *any other variable not mentioned as an argument*. Thus the complete ordering scale is: numerical constants < declared constants < declared scalars < first argument to ORDERLESS < . . . < last argument to ORDERLESS < variables which begin with A < . . . < variables which begin with Z < last argument to ORDERGREAT < . . . < first argument to ORDERGREAT < declared MAINVARs. See section 10.1.3 [Declarations], page 148, for the description of MAINVAR for another ordering scheme.

`ORDERLESSP` (*exp1, exp2*) [*Function*]

returns TRUE if *exp1* precedes *exp2* in the ordering set up by the ORDERLESS command.

`UNORDER` () [*Function*]

stops the aliasing created by the last use of the ordering commands ORDERGREAT and ORDERLESS. ORDERGREAT and ORDERLESS may not be used more than one time each without calling UNORDER. See also ORDERGREAT and ORDERLESS.

2.6 Reviewing Options

`OPTIONS` () [*Special Form*]

enters the OPTIONS interpreter which is a structured list of **Maxima** commands. You type the number of the subject of interest, followed by a `;` to see the list of commands available for that subject. To move back up the list the command `BACK;` will go back up one level, and `TOP;` will get you back to the original entry list. `EXIT;` will quit out of OPTIONS. DESCRIBE may be called inside OPTIONS on the commands listed, either by number or by name.

`OPTIONS`(command); will give the various commands and switches associated with *command*.

`INFOLISTS` [*Variable*]

a list of the names of all of the information lists in **Maxima**. These are:

LABELS all bound C,D, and E labels.

VALUES all bound atoms, i.e. user variables, not øDOE-Maxima Options or Switches, (set up by :, ::, or functional binding). See section 1.3.1 [Defining Variables], page 4.

FUNCTIONS all user defined functions (set up by :=). See section 1.3.2 [Defining Functions], page 6.

ARRAYS declared and undeclared arrays (set up by :, ::, or :=. See section 6.1.1 [Defining Arrays], page 87.

MACROS any macros defined by the user. See section 10.1.2 [Defining Macros], page 145.

MYOPTIONS all options ever reset by the user (whether or not they get reset to their default value).

RULES user defined pattern matching and simplification rules (set up by TELLSIMP, TELLSIMPAFTER, DEFMATCH, or, DEFRULE.) See section 10.1.7.1 [Defining Simplification Rules], page 154.

ALIASES atoms which have a user defined alias (set up by the ALIAS, ORDERGREAT, ORDERLESS functions, or by DECLAREing the atom to be a NOUN).

DEPENDENCIES atoms which have functional dependencies (set up by the DEPENDS or GRADEF functions). See section 8.3.3 [Defining Functional Dependencies], page 114.

GRADEFS functions which have user defined derivatives (set up by the GRADEF function). See section 8.3.2 [Defining Gradients], page 114.

PROPS atoms which have any property other than those mentioned above, such as ATVALUEs, MATCHDECLAREs, etc., as well as properties specified in the DECLARE function. See section 10.1.3 [Declarations], page 148.

LET_RULE_PACKAGES list of all the user-defined let rule packages plus the special package DEFAULT_LET_RULE_PACKAGE. (DEFAULT_LET_RULE_PACKAGE is the name of the rule package used when one is not explicitly set by the user.) See section 10.1.7.2 [Substitution Rules], page 155.

`OPTIONS` [*Variable*]

all options ever reset by the user (whether or not they get reset to their default value).

`OPTIONSET` [*variable, default: FALSE*]

if TRUE, **Maxima** will print out a message whenever a **Maxima** option is reset. This is useful if the user is doubtful of the spelling of some option and wants to make sure that the variable he assigned a value to was truly an option variable.

`RESET` () [*Function*]

causes all **Maxima** options to be set to their default values. Please note that this does not include features of terminals such as LINEL which can only be changed by assignment as they are not considered to be computational features of **Maxima**.

2.7 Accessing the Underlying Lisp

`TO_LISP` () [*Function*]

enters the Lisp system under **Maxima**. This is useful on those systems where control-uparrow is not available for this function. The user can now type any Lisp S-expression and have it evaluated. Typing (CONTINUE) causes **Maxima** to be re-entered.

`?` [*Syntax*]

To access a function or variable in the Lisp, precede the function name or variable with ?.

```
(c1) ?status(?features);

(d1) (long-filenames, sun, portable, 68k, systems, string, unix, Franz,
franz)
```

2.8 Utility Functions

`CLEARSCREEN` () [*Function*]

Clears the screen. The same as typing control-L.

`GETCHAR` (*arg*, *i*) [*Function*]

returns the *i*'th character of the quoted string or atomic name *arg*. This function is useful in manipulating the LABELS list.

PAUSE () [*Function*]

Causes the display to pause, printing --Pause– and waiting for the a SPC to resume printing. Then it clears the screen and continues. PAUSE("–Something else–"); will use --Something else– as the string printed instead of --Pause–. PAUSE("–Something else–", "–And some more–"); will use --Something else– instead of --Pause– and --And some more– instead of --Continued–.

ALARMCLOCK (*arg1*, *arg2*, *arg3*) [*Special Form*]

will execute the function of no arguments whose name is *arg3* when the time specified by *arg1* and *arg2* elapses. If *arg1* is the atom TIME then *arg3* will be executed after *arg2* seconds of real-time has elapsed, while if *arg1* is the atom RUNTIME, then *arg3* will be executed after *arg2* milliseconds of cpu time. If *arg2* is negative then the *arg1* timer is shut off.

CHAPTER

THREE

Mathematical Functions

3.1 Comparison Functions

Maxima has the usual inequality operators:

< less than

> greater than

>= greater than or equal to

<= less than or equal to

not equal to

See section 2.1.1 [Conditionals], page 13.

3.2 Arithmetic Functions

ABS (*exp*) [*Function*]

returns absolute value of *exp*. ABSBOXCHAR[!] is the character used to draw absolute value signs around expressions which are more than a single line high.

CABS (*exp*) [*Function*]

returns the complex absolute value (the complex modulus) of *exp*.

MAX (*X1*, *X2*, ...) [*Function*]

yields the maximum of its arguments (or returns a simplified form if some of its arguments are non-numeric).

`MIN` (*X1, X2, ...*) [*Function*]

yields the minimum of its arguments (or returns a simplified form if some of its arguments are non-numeric).

`SIGNUM` (*X*) [*Function*]

If `X`<0 then -1 else if `X`>0 then 1 else 0. If X is not numeric then a simplified but equivalent form is returned. For example, SIGNUM(-X) gives `-`SIGNUM(X).

`POLYSIGN` (*x*) [*Function*]

same as SIGNUM but always returns a numerical result by looking at the numerical factor of the highest degree term in `x`.

`SQRT` (*x*) [*Function*]

the square root of *x*. It is represented internally by `x`**(1/2). See section 9.4 [Roots of Polynomials], page 131, for the definition of ROOTSCONTRACT. See section 2.4.2 [Display of Exponentials], page 24, for the definition of RADEXPAND. See section 2.4 [Flags Effecting the Displayed Form], page 21, for the definition of the variable SQRTDISPFLAG.

`ISQRT` (*int*) [*Function*]

takes one integer argument and returns the integer SQRT of its absolute value.

`INRT` (*x, n*) [*Function*]

takes two integer arguments, *x* and *n*, and returns the integer *n*'th root of the absolute value of *x*.

3.3 Transcendental Functions

3.3.1 Exponential Functions

`EXP` (*x*) [*Function*]

the exponential function. It is represented internally as %E**X.

`EXPT` (*A, B*) [*Function*]

if an exponential expression is too wide to be displayed as `A`**B, it will appear as `EXPT`(A,B), or as `NCEXPT`(A,B) in the case of `A`^^B.

See section 2.4.2 [Display of Exponentials], page 24, for the definition of DEMOIVRE, %EDISPFLAG, %EMODE, %ENUMER EXPTDISPFLAG, EXPTSUBST and RADEXPAND.

3.3.2 Logarithm Functions

`LOG` (*x*) [*Function*]

the natural logarithm of *x*.

`PLOG` (*X*) [*Function*]

the principal branch of the complex-valued natural logarithm with -%PI < CARG(X) <= +%PI.

See section 2.4.3 [Display of Logarithms], page 25 for the definition of LOGNEGINT, LOGEXPAND,LOGNUMER and LOGSIMP. See section 2.4.2 [Display of Exponentials], page 24, for the definition of DEMOIVRE and %E_TO_NUMLOG.

3.3.3 Trig Functions

Maxima has many trig functions defined. Not all trig identities are programmed, but it is possible for the user to add many of them using the pattern matching capabilities of the system. The trig functions defined in **Maxima** are: ACOS, ACOSH, ACOT, ACOTH, ACSC, ACSCH, ASEC, ASECH, ASIN, ASINH, ATAN, ATANH, COS, COSH, COT, COTH, CSC, CSCH, SEC, SECH, SIN, SINH, TAN, and TANH. There is a demo in the file: '`demo/trig.dem`'.

There are a number of commands especially for handling trig functions, see TRIGEXPAND, TRIGREDUCE, and the switch TRIGSIGN. Two SHARE packages extend the simplification rules built into **Maxima**, '`share1/ntrig.mc`' and '`share1/atrig1.mc`'.

See section 2.4.4 [Display of Trig Functions], page 26.

3.3.3.1 Basic Trig Functions

`SIN` (*exp*) [*Syntax*]

Sine of *exp*, in Radians.

`COS` (*exp*) [*Syntax*]

Cosine of *exp* in radians.

`TAN` (*exp*) [*Syntax*]

Tangent of *exp* in radians.

`CSC` (*exp*) [*Syntax*]

Cosecant of *exp*, in Radians.

`SEC` (*exp*) [*Syntax*]

Cosecant of *exp* in radians.

`COT` (*exp*) [*Syntax*]

Cotangent of *exp* in radians.

3.3.3.2 Inverse Trig Functions

`SIN` (*exp*) [*Syntax*]

Arc Sine of *exp*, in Radians.

`ACOS` (*exp*) [*Syntax*]

Arc Cosine of *exp* in radians.

`ATAN` (*exp*) [*Syntax*]

Arc Tangent of *exp* in radians.

`ATAN2` (*Y,X*) [*Syntax*]

yields the value of ATAN(Y/X) in the interval -%PI to %PI.

`ACSC` (*exp*) [*Syntax*]

Arc Cosecant of *exp*, in Radians.

`ASEC` (*exp*) [*Syntax*]

Arc Cosecant of *exp* in radians.

`ACOT` (*exp*) [*Syntax*]

Arc Cotangent of *exp* in radians.

3.3.3.3 Hyperbolic Trig Functions

`SINH` (*exp*) [*Syntax*]

Hyperbolic Sine of *exp*.

`COSH` (*exp*) [*Syntax*]

Hyperbolic Cosine of *exp*.

`TANH` (*exp*) [*Syntax*]

Hyperbolic Tangent of *exp*.

`CSCH` (*exp*) [*Syntax*]

Hyperbolic Cosecant of *exp*.

`SECH` (*exp*) [*Syntax*]

Hyperbolic Cosecant of *exp*.

`COTH` (*exp*) [*Syntax*]

Hyperbolic Cotangent of *exp*.

3.3.3.4 Inverse Hyperbolic Trig Functions

`ASINH` (*exp*) [*Syntax*]

Arc Hyperbolic Sine of *exp*.

`ACOSH` (*exp*) [*Syntax*]

Arc Hyperbolic Cosine of *exp*.

`ATANH` (*exp*) [*Syntax*]

Arc Hyperbolic Tangent of *exp*.

`ACSCH` (*exp*) [*Syntax*]

Arc Hyperbolic Cosecant of *exp*.

ASECH (*exp*) [*Syntax*]

Arc Hyperbolic Cosecant of *exp*.

ACOTH (*exp*) [*Syntax*]

Arc Hyperbolic Cotangent of *exp*.

3.4 Factorial and Gamma Functions

3.4.1 Factorials

FACTORIAL (*int*) [*Function*]

The factorial function. FACTORIAL(int) = int!.

! [*Syntax*]

The postfix factorial operator.

!! [*Syntax*]

The postfix double factorial operator.

FACTLIM [*variable, default -1*]

gives the highest factorial which is automatically expanded. If it is -1 then all integers are expanded.

BFFAC (*exp, int*) [*Function*]

is a bigfloat version of the factorial function. *int* is how many digits to retain and return: it's a good idea to request a couple of extra.

CBFAC (*z, fpprec*) [*Function*]

a factorial for complex bigfloats. It may be used by doing LOAD(BFAC);. There are some usage notes in the file: '`share2/bffac.usg`'.

See section 5.1.4 [Simplifying Factorials], page 74, for the definition of FACTCOMB, SUMSPLITFACT, MINFACTORIAL, and MAKEGAMMA.

3.4.2 Binomials and Generalized Factorials

`BINOMIAL` (*X*, *Y*) [*Function*]

the binomial coefficient X*(X-1)*...*(X-Y+1)/Y!. If *X* and *Y* are integers, then the numerical value of the binomial coefficient is computed. If *Y*, or the value *X*-*Y* is an integer, the binomial coefficient is expressed as a polynomial.

`GENFACT` (*X*, *Y*, *Z*) [*Function*]

is the generalized factorial of X which is: X*(X-Z)*(X-2*Z)* ...*(X-(Y-1)*Z). Thus, for integral X, GENFACT(X,X,1)=X! and GENFACT(X,X/2,2)=X!!.

3.4.3 Gamma and Related Functions

`GAMMA` (*exp*) [*Function*]

the gamma function, GAMMA(I)=(I-1)! for I a positive integer. For the Euler-Mascheroni constant, see %GAMMA. See also the MAKEGAMMA function. The variable GAMMALIM default: [1000000] controls simplification of the GAMMA function.

`GAMMALIM` [*variable, default: 1000000*]

controls simplification of the gamma function for integral and rational number arguments. If the absolute value of the argument is not greater than GAMMALIM, then simplification will occur. Note that the FACTLIM switch controls simplification of the result of GAMMA of an integer argument as well.

`CGAMMA` (*exp)* [*Function*]

The Gamma function in the complex plane. Do LOAD(CGAMMA); to use these functions: CGAMMA, CGAMMA2, and LOGCGAMMA2. These functions evaluate the GAMMA function over the complex plane using the algorithm of Kuki, CACM algorithm 421. Calculations are performed in single precision and the relative error is typically around 1.0E-7; evaluation at one point costs less than 1 msec. The algorithm provides for an error estimate, but the **Maxima** implementation currently does not use it. CGAMMA is the general function and may be called with a symbolic or numeric argument. With symbolic arguments, it returns as is; with real floating or rational arguments, it uses the **Maxima** GAMMA function; and for complex numeric arguments, it uses the Kuki algorithm.

CGAMMA2 of two arguments, real and imaginary, is for numeric arguments only; LOGCGAMMA2 is the same, but the log-gamma function is calculated. These two functions are somewhat more efficient.

`BETA` (*X*, *Y)* [*Function*]

same as GAMMA(X)*GAMMA(Y)/GAMMA(X+Y).

3.4.3.1 Polygamma Functions

`PSI` (*exp)* [*Function*]

derivative of LOG(GAMMA(X)). At this time, **Maxima** does not have numerical evaluation capabilities for PSI. Basic simplification routines for the polylogs and polygamma functions have been introduced in **Maxima**. We have decided to use subscripted notation in order to be consistent with standard reference texts for these functions. (Note that the the **Maxima** notation doesn't imply **Maxima** can produce the definitions.) The notation is: PSI[N](X) = DIFF(PSI[0](X),X,N) where PSI[0](X) = DIFF(LOG(GAMMA(X)),X)

There are closed forms for: (in SIN, COS, or ZETA functions) rational X, N = 0; N integral, > -1 and X integral or half-integral. Currently there are no numerical routines for the polygammas. The following flags permit some control over simplification: they must be set to fixnum (integer) values:

`MAXPSIPOSINT` [*variable, default: 20*]

is the largest value of the integral part of X for which a closed form will be computed.

`MAXPSINEGINT` [*variable, default: -10*]

for negative X less than this no closed forms will be computed.

The following control simplification of PSI[0](P/Q) for P and Q integral and 0 < P/Q < 1 (i.e. the fractional part of arguments)

`MAXPSIFRACNUM` [*variable, default: 4*]

is the largest P for which simplification occurs.

`MAXPSIFRACDENOM` [*variable, default: 4*]

is the largest Q for which simplification occurs.

`BFPSI` (*n, z, fpprec*) [*Function*]

gives polygammas of real arg and integer order. For digamma, BFPSI0(z,fpprec) is more direct. Note -BFPSI0(1,fpprec) provides bigfloated %GAMMA. To use this do LOAD(BFFAC);,

`LI` (*exp*) [*Function*]

Basic simplification routines for the polylogs and polygamma functions have been introduced in **Maxima**. We have decided to use subscripted notation in order to be consistent with standard reference texts for these functions. (Note that the the **Maxima** notation doesn't imply **Maxima** can produce the definitions.) The notation is: LI[N](X) = INTEGRATE(LI[N-1](T)/T,T,0,X) where LI[1](X) = -LOG(1-X).

Simplification: Closed forms for argument 1, -1 when N is +integral (involving ZETA functions); closed form for LI[2](1/2); numerical routine for LI[2](X). Fast numeric routines are now available for LI[2](x) and LI[3](x). Chebyshev expansions are used in the approximations. The extension for large real values of X is adopted following Lewin, i.e. when X is greater than unity LI[2](X) has - %I %PI LOG(X) as its imaginary part. Currently this only concerns the numerical routine.

3.5 Special Functions

3.5.1 Airy Functions

AIRY (*arg*) [*Function*]

returns the Airy function AI of real argument *arg*. The file '`share1/airy.l`' contains routines to evaluate the Airy functions **AI**(*exp*), **BI**(*exp*), and their derivatives **DAI**(*exp*), **DBI**(*exp*). The Airy equation diff(y(x),x,2)-x*y(x)=0 has two linearly independent solutions, taken to be AI(x) and BI(x). This equation is very popular as an approximation to more complicated problems in many mathematical physics settings.

Do LOAD('AIRY); to get the functions **AI**(x), **BI**(x), **DAI**(x), **DBI**(x), which compute the Ai(x), Bi(x), d(Ai(x))/dx, and d(Bi(x))/dx functions respectively. The result will be a floating point number if the argument is a number, and will return a simplified form otherwise. An error will occur if the argument is large enough to cause an overflow in the exponentials, or a loss of accuracy in SIN or COS. This makes the range of validity about -2800 to 1.e38 for AI and DAI, and -2800 to 25 for BI and DBI. The GRADEF rules are now known to **Maxima**:

```
diff(AI(x),x)=DAI(x),
diff(DAI(x),x)=x*AI(x),
diff(BI(x),x)=DBI(x),
diff(DBI(x),x)=x*BI(x).
```

The method is to use the convergent Taylor series for abs(x)<3., and to use the asymptotic expansions for x<-3. or x>3. as needed. This results in only very minor numerical discrepancies at x=3. or x=-3. For details, please see [AS64], section 10.4 (hardcover ed.) and Table 10.11. To get the floating point Taylor expansions of the functions here, do EV(TAYLOR(AI(X),X,0,9),INFEVAL); for example. Please also try BESSEL (by CFFK) for the AIRY function there. There are some usage notes in the file: '`share1/airy.usg`'..

3.5.2 Bernoulli Numbers

BERN (*int*) [*Function*]

gives the *int*'th Bernoulli number for integer *int*.

ZEROBERN [*variable, default: TRUE*]

if set to FALSE excludes the zero Bernoulli numbers. See also the BERN function.

BURN (*int*) [*Function*]

is like BERN(int), but without computing all of the uncomputed Bernoullis of smaller index. So BURN works efficiently for large, isolated N. (BERN(402) takes about 645 secs vs 13.5 secs for BURN(402). BERNs' time growth seems to be exponential, while BURNs' is about cubic. But if next you do BERN(404), it only takes 12 secs, since BERN remembers all in an array, whereas BURN(404) will take maybe 14 secs

or maybe 25, depending on whether **Maxima** needs to BFLOAT a better value of %PI.) BURN is available by doing LOAD(BFFAC);. BURN uses an observation of WGD that (rational) Bernoulli numbers can be approximated by (transcendental) zetas with tolerable efficiency.

BERNPOLY (*var, int*) [*Function*]

generates the *int*'th Bernoulli polynomial in the variable *var*.

3.5.3 Elliptic Functions

A package in the SHARE directory for numerical routines for Elliptic Functions and Complete Elliptic Integrals. (Notation of [AS64], Chapters 16 and 17). Do LOAD(ELLIPT); to use this package. At present all arguments must be floating point. You'll get nonsense otherwise. Be warned. The functions available are:

```
AM(U,M) - amplitude with modulus M
AM1(U,M1) - amplitude with complementary modulus M1
AM(U,M):=AM1(U,1-M); so use AM1 if M ~ 1
SN(U,M):=SIN(AM(U,M));
CN(U,M):=COS(AM(U,M));
DN(U,M):=SQRT(1-M*SN(U,M)^2);
```

(These functions come defined like this. Others CD, NS etc. may be similarly defined.)

```
ELLIPTK(M)      Complete elliptic integral of first kind
ELLIPTK1(M1)    Same but with complementary modulus.
ELLIPTK(M)      := ELLIPTK1(1-M); so use if M ~ 1
ELLIPTE(M)      Complete elliptic integral of second kind
ELLIPTE1(M1)    Same but with complementary modulus.
ELLIPTE(M)      := ELLIPTE1(1-M); so use if M ~ 1
```

There are some usage notes in the file: '`share/elliptic.usg`'..

3.5.4 Zeta Functions

ZETA (*int*) [*Function*]

gives the Riemann zeta function for certain integer values of *int*.

ZETA%PI [*variable, default: TRUE*]

if FALSE, suppresses ZETA(n) giving coeff*%PI**n for *n* even.

BFZETA (*exp, int*) [*Function*]

is a bigfloat version of the Riemann Zeta function. The 2nd argument is how many digits to retain and return: it's a good idea to request a couple of extra. This function is available by doing LOAD(BFFAC);.

`BGZETA` (*s, fpprec*) [*Function*]

is like BZETA, but avoids arithmetic overflow errors on large arguments, is faster on medium size arguments (say S=55, FPPREC=69), and is slightly slower on small arguments. It may eventually replace BZETA. BGZETA is available by doing `LOAD(BFAC);`.

`BHZETA` (*s, h, fpprec)* [*Function*]

gives FPPREC digits of SUM((K+H)**-S,K,0,INF). This is available by doing `LOAD(BFFAC);`.

`NZETA` (*Z*) [*Function*]

returns the complex value of the Plasma Dispersion function for complex *Z*. **NZETAR**(Z) returns `REALPART(NZETA(Z))`. **NZETAI**(Z) returns `IMAGPART(NZETA(Z))`. This function is related to the complex error function by `NZETA(Z) = %I*SQRT(%PI)*EXP(-Z**2)*(1-ERF(-%I*Z))`.Plasma Dispersion Function

3.5.5 Miscellaneous Special Functions

`BESSEL` (*Z, A*) [*Function*]

returns the Bessel function J for complex *Z* and real $A > 0.0$. Also an array BESSELARRAY is set up such that `BESSELARRAY[I] = J[I + A - ENTIER(A)](Z)`.

`EULER` (*int*) [*Function*]

gives the *int*'th Euler number for integer *int*. For the Euler-Mascheroni constant, see %GAMMA.

`ERF` (*arg*) [*Function*]

the error function, whose derivative is `2*EXP(-X**2)/SQRT(%PI)`.

`FIB` (*int*) [*Function*]

the *int*'th Fibonacci number with `FIB(0)=0`, FIB(1)=1, and `FIB(-N)=(-1)**(N+1) *FIB(N)`. PREVFIB is `FIB(X-1)`, the Fibonacci number preceding the last one computed.

`FIBTOPHI` (*exp*) [*Function*]

converts FIB(*n*) to its closed form definition. This involves the constant %PHI (`(SQRT(5)+1)/2` = 1.618033989). If you want the Rational Function Package to know about %PHI do `TELLRAT(%PHI**2-%PHI-1); ALGEBRAIC:TRUE;`.

See section 8.4 [Integration], page 116, for the definition of SPECINT which uses hypergeometric functions.

3.6 Complex Variables

A complex expression is specified in **Maxima** by adding the real part of the expression to %I times the imaginary part. Thus the roots of the equation X**2-4*X+13=0 are 2+3*%I and 2-3*%I. Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the REALPART, IMAGPART, RECTFORM, POLARFORM, ABS, CARG functions.

CARG (*exp*) [*Function*]

returns the argument (phase angle) of *exp*. Due to the conventions and restrictions, principal value cannot be guaranteed unless *exp* is numeric.

POLARFORM (*exp*) [*Function*]

returns R*%E**(%I*THETA) where R and THETA are purely real.

RECTFORM (*exp*) [*Function*]

returns an expression of the form A + B*%I, where A and B are purely real.

POLARTORECT (*magnitude-array, phase-array*) [*Function*]

converts from magnitude/phase form into real/imaginary form putting the real part in the magnitude array and the imaginary part into the phase array (<real>=<magnitude>*COS(<phase>) and <imaginary>=<magnitude>*SIN(<phase>).) This function is part of the FFT package. do LOAD(FFT); to use it. Like FFT and IFT this function accepts 1 or 2 dimensional arrays. However, the array dimensions need not be a power of 2, nor need the 2D arrays be square. The above function returns a list of the arguments.

RECTTOPOLAR (*real-array, imag-array*) [*Function*]

undoes POLARTORECT. The phase is given in the range from -%PI to %PI. This function is part of the FFT package. Do LOAD(FFT); to use it. Like FFT and IFT this function accepts 1 or 2 dimensional arrays. However, the array dimensions need not be a power of 2, nor need the 2D arrays be square. The above function returns a list of the arguments.

M1PBRANCH [*variable, default: FALSE*]

principal branch for -1 to a power. Quantities such as (-1)**(1/3) [i.e. odd rational exponent] and (-1)**(1/4) [i.e. even rational exponent] are now handled

See section 1.4.4 [Data Type Coercion], page 9.

3.7 Number Theory Functions

`DIVSUM` (*n, k*) [*Function*]

adds up all the factors of *n* raised to the *k*'th power. If only one argument is given, then *k* is assumed to be 1.

`PRIME` (*int*) [*Function*]

gives the *int*'th prime. MAXPRIME default: [489318] is the largest number accepted as argument. Note: The PRIME command does not work in Tops20 **Maxima**.

`MAXPRIME` [*variable, default: 489318*]

the largest number which may be given to the PRIME command, which returns the *n*'th prime.

`QUNIT` (*n*) [*Function*]

gives the principal unit of the real quadratic number field SQRT(n) where *n* is an integer, i.e. the element whose norm is unity. This amounts to solving Pell's equation A**2 - n*B**2=1.

```
(C1) QUNIT(17);
(D1)                SQRT(17)+4
(C2)  EXPAND(\%*(SQRT(17)-4));
(D2)                  1
```

`TOTIENT` (*n*) [*Function*]

is the number of integers less than or equal to *n* which are relatively prime to *n*.

`RANDOM` (*int*) [*Function*]

returns a random integer between 0 and *int*-1. If no argument is given then a random integer between -2**(29) and 2**(29) -1 is returned. If *int* is FALSE then the random sequence is restarted from the beginning.

`GAUSS` (*mean, sd*) [*Function*]

returns a random floating point number from a normal distribution with mean *mean* and standard deviation *sd*. This is part of the BESSEL function package, do LOAD(BESSEL); to use it.

CHAPTER

FOUR

Manipulating Expressions

See section 2.3 [Displaying Expressions], page 19.

4.1 Evaluation

EV (*exp, arg1, . . . , argn*) [*Special Form*]

is one of **Maxima**'s most powerful and versatile commands. It evaluates the expression *exp* in the environment specified by the *argi*. This is done in steps, as follows:

1. First the environment is set up by scanning the *argi* which may be as follows:

   SIMP causes *exp* to be simplified regardless of the setting of the switch SIMP which inhibits simplification if FALSE.

   NOEVAL suppresses the evaluation phase of EV (see step (4) below). This is useful in conjunction with the other switches and in causing *exp* to be resimplified without being reevaluated.

   EXPAND causes expansion.

   EXPAND(m,n) causes expansion, setting the values of MAXPOSEX and MAXNEGEX to *m* and *n* respectively.

   DETOUT causes any matrix inverses computed in *exp* to have their determinant kept outside of the inverse rather than dividing through each element.

   DIFF causes all differentiations indicated in *exp* to be performed.

   DERIVLIST(*var1, . . . , vark***)** causes only differentiations with respect to the indicated variables.

   FLOAT causes non-integral rational numbers to be converted to floating point.

   NUMER causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in *exp* which have been given numerical values to be replaced by their values. It also sets the FLOAT switch on.

   PRED causes predicates (expressions which evaluate to TRUE or FALSE) to be evaluated.

   EVAL causes an extra post-evaluation of *exp* to occur. (See step (5) below.)

   E where *E* is an atom DECLARE'd to be an EVFLAG, causes *E* to be bound to TRUE during the evaluation of *exp*.

V:expression **(or alternately** *V=expression***)** causes V to be bound to the value of *expression* during the evaluation of *exp*. Note that if V is a **Maxima** option, then expression is used for its value during the evaluation of *exp*. If more than one argument to EV is of this type then the binding is done in parallel. If V is a non-atomic expression then a substitution rather than a binding is performed.

E where *E*, a function name, has been DECLARE'd to be an EVFUN, auses *E* to be applied to *exp*.

Any other function names (e.g. SUM) cause evaluation of occurrences of those names in *exp* as though they were verbs. In addition a function occurring in *exp* (say F(args)) may be defined locally for the purpose of this evaluation of *exp* by giving *F(args):=body* as an argument to EV. If an atom not mentioned above, or a subscripted variable or subscripted expression was given as an argument, it is evaluated and if the result is an equation or assignment then the indicated binding or substitution is performed. If the result is a list then the members of the list are treated as if they were additional arguments given to EV. This permits a list of equations to be given e.g. [X=1, Y=A**2] , or a list of names of equations e.g. [E1, E2] where E1 and E2 are equations) such as that returned by SOLVE.

The *argi* of EV may be given in any order, with the exception of substitution equations, which are handled in sequence, left to right, and EVFUNs which are composed. For example, EV(exp,RATSIMP,REALPART) is handled as REALPART(RATSIMP(exp)). The SIMP, NUMER, FLOAT, and PRED switches may also be set locally in a block, or globally at the top level in **Maxima** so that they will remain in effect until being reset. If *exp* is in CRE Form then EV will return a result in CRE form provided the NUMER and FLOAT switches are not both TRUE.

2. During step (1), a list is made of the non-subscripted variables appearing on the left side of equations in the *argi* or in the value of some *argi* if the value is an equation. The variables (both subscripted variables which do not have associated array functions, and non-subscripted variables) in the expression *exp* are replaced by their global values, except for those appearing in this list. Usually, *exp* is just a label or % (as in (C2) below), so this step simply retrieves the expression named by the label, so that EV may work on it.

3. If any substitutions are indicated by the *argi*, they are carried out now.

4. The resulting expression is then re-evaluated (unless one of the *argi* was NOEVAL) and simplified according the the *argi*. Note that any function calls in *exp* will be carried out after the variables in it are evaluated and that EV(F(X)) thus may behave like F(EV(X)).

5. If one of the *argi* was EVAL, steps (3) and (4) are repeated.

```
(C1) SIN(X)+COS(Y)+(W+1)**2+'DIFF(SIN(W),W);
                          d                  2
(D1)  COS(Y) + SIN(X) + -- SIN(W) + (W + 1)
                          dW
(C2) EV(\%,SIN,EXPAND,DIFF,X=2,Y=1);
                        2
(D2)      COS(W) + W  + 2 W + COS(1) + 1.90929742
```

An alternate top level syntax has been provided for EV, whereby one may just type in its arguments, without the EV(). That is, one may write simply exp, arg1, ..., argn. This is not permitted as part of another expression, i.e. in functions, blocks, etc.

```
(C4) X+Y,X:A+Y,Y:2;
(D4)                Y + A + 2
```

Notice the parallel binding process.

```
(C5) 2*X-3*Y=3$
(C6) -3*X+2*Y=-4$
(C7) SOLVE([D5,D6]);
SOLUTION
                                  1
(E7)                       Y =  - -
                                  5
                             6
(E8)                     X = -
                             5
(D8)                [E7, E8]
(C9) D6,D8;
(D9)                - 4 =  - 4
(C10) X+1/X > GAMMA(1/2);
                     1
(D10)           X + - > SQRT(\%PI)
                     X
(C11) \%,NUMER,X=1/2;
(D11)             2.5 > 1.7724539
(C12) \%,PRED;
(D12)                   TRUE
```

EVFLAG [*Variable*]

is the list of things known to the EV function. An item will be bound to TRUE during the execution of EV if it is mentioned in the call to EV, e.g. EV(%,numer);. Initial EVFLAGs are: FLOAT, PRED, SIMP, NUMER, DETOUT, EXPONENTIALIZE, DEMOIVRE, KEEPFLOAT, LISTARITH, TRIGEXPAND, SIMPSUM, ALGEBRAIC, RATALGDENOM, FACTORFLAG, %EMODE, LOGARC, LOGNUMER, RADEXPAND, RATSIMPEXPONS, RATMX, RATFAC, INFEVAL, %ENUMER, PROGRAMMODE, LOGNEGINT, LOGABS, LETRAT, HALFANGLES, EXPTISOLATE, ISOLATE_WRT_TIMES, SUMEXPAND, CAUCHYSUM, NUMER_PBRANCH, M1PBRANCH, DOTSCRULES, and LOGEXPAND.

EVFUN [*Variable*]

is the list of functions known to the EV function which will get applied if their name is mentioned. Initial EVFUNs are FACTOR, TRIGEXPAND, TRIGREDUCE, BFLOAT, RATSIMP, RATEXPAND, RADCAN, LOGCONTRACT, RECTFORM, and POLARFORM.

4.1.1 Evaluation Flags

NOEVAL suppresses the evaluation phase of EV. This is useful in conjunction with other switches and in causing expressions to be resimplified without being reevaluated.

NOUNS when used as an option to the EV command, converts all noun forms occurring in the expression being EV'd to verbs, i.e. it evaluates them.

NUMER causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in *exp* which have been given numer values to be replaced by their values. It also sets the FLOAT switch on.

FLOAT causes non-integral rational numbers and bigfloat numbers to be converted to floating point.

EVAL causes an extra post-evaluation of *exp* to occur.

INFEVAL leads to an "infinite evaluation" mode. EV repeatedly evaluates an expression until it stops changing. To prevent a variable, say X, from being evaluated away in this mode, simply include X='X as an argument to EV. Of course expressions such as EV(X,X=X+1,INFEVAL); will generate an infinite loop. Caveat Evaluator.

PRED causes predicates (expressions which evaluate to TRUE or FALSE) to be evaluated.

DIFF causes all differentiations to be carried out. See section 8.3 [Differentiation], page 112.

DERIVLIST(*var1, . . . , vark***)** causes only differentiations with respect to the indicated variables. See section 8.3 [Differentiation], page 112.

SIMP causes *exp* to be simplified regardless of the setting of the variable SIMP, which normally inhibits simplification if set to FALSE.

4.1.2 Noun and Verb Forms

NOUNIFY (*fun*) [*Function*]

returns the noun form of the function name *fun*. This is needed if one wishes to refer to the name of a verb function as if it were a noun. Note that some verb functions will return their noun forms if they can't be evaluated for certain arguments. This is also the form returned if a function call is preceded by a quote. Verb is the opposite of noun form, i.e. a verb form which "does something" (for most functions the usual case). E.g. INTEGRATE integrates a function, unless it is DECLAREd to be a "noun," in which case it represents the INTEGRAL of the function.

VERBIFY (*fun*) [*Function*]

returns the function name *fun* in its verb form.

APPLY_NOUNS (*exp*) [*Function*]

causes the application of noun forms in an expression. For example, EXP:'DIFF(X**2/2,X); APPLY_NOUNS(EXP); gives X. This gives the same result as EV(EXP,NOUNS); except that it can be faster and use less storage. It also can be used in translated code, where EV may cause problems. Note that it is called APPLY_NOUNS, not EV_NOUNS, because what it does is to APPLY the rules corresponding to the noun-form operators, which is not evaluation.

4.2 Canonical Rational Expressions

Canonical Rational Expressions constitute a kind of representation which is especially suitable for expanded polynomials and rational functions (as well as for partially factored polynomials and rational functions when RATFAC default: [FALSE] is set to TRUE). In this CRE form an ordering of variables (from most to least main) is assumed for each expression. Polynomials are represented recursively by a list consisting of the main variable followed by a series of pairs of expressions, one for each term of the polynomial. The first member of each pair is the exponent of the main variable in that term and the second member is the coefficient of that term which could be a number or a polynomial in another variable again represented in this form. Thus the principal part of the CRE form of 3*X**2-1 is (X 2 3 0 -1) and that of 2*X*Y+X-3 is (Y 1 (X 1 2) 0 (X 1 1 0 -3)) assuming Y is the main variable, and is (X 1 (Y 1 2 0 1) 0 -3) assuming *X* is the main variable. “Main”-ness is usually determined by reverse alphabetical order.

The variables of a CRE expression needn’t be atomic. In fact any subexpression whose main operator is not + - * / or ** with integer power will be considered a variable of the expression (in CRE form) in which it occurs. For example the CRE variables of the expression X+SIN(X+1)+2*SQRT(X)+1 are X, SQRT(X), and SIN(X+1). If the user does not specify an ordering of variables by using the RATVARS function **Maxima** will choose an alphabetic one.

In general, CRE’s represent rational expressions, that is, ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. The internal form is essentially a pair of polynomials (the numerator and denominator) preceded by the variable ordering list. If an expression to be displayed is in CRE form or if it contains any subexpressions in CRE form, the symbol /R/ will follow the line label. See the RAT function for converting an expression to CRE form. An extended CRE form is used for the representation of Taylor Series. The notion of a rational expression is extended so that the exponents of the variables can be positive or negative rational numbers rather than just positive integers and the coefficients can themselves be rational expressions as described above rather than just polynomials. These are represented internally by a recursive polynomial form which is similar to and is a generalization of CRE form, but carries additional information such as the degree of truncation. As with CRE form, the symbol /T/ follows the line label of such expressions.

4.2.1 Converting To and From CRE form

RAT (*exp, var1, var2, . . .*) [*Function*]

converts *exp* to CRE form by expanding and combining all terms over a common denominator and cancelling out the greatest common divisor of the numerator and denominator as well as converting floating point numbers to rational numbers within a tolerance of RATEPSILON default: [2.0E-8]. The variables are ordered according to the *v1, . . . , vn* as in RATVARS, if these are specified. RAT does not generally simplify functions other than +, -, *, /, and ** to an integer power whereas, RATSIMP does handle these cases. Note that atoms (numbers and names) in CRE form are not the same as they are in the general form. Thus RAT(X) - X results in RAT(0) which has a different internal representation than 0.

```
(C1) ((X-2*Y)**4/(X**2-4*Y**2)**2+1)*(Y+A)*(2*Y+X)
       /(4*Y**2+X**2);
                                          4
                                   (X - 2 Y)
                (Y + A) (2 Y + X) (------------ + 1)
                                     2      2 2
                                   (X  - 4 Y )
(D1)            ----------------------------------
                                  2    2
                               4 Y  + X
\clearpage
(C2) RAT(\%,Y,A,X);
                                2 A + 2 Y
(D2)/R/                         ---------
                                 X + 2 Y
```

RATDISREP (*exp*) [*Function*]

changes its argument from CRE form to general form. This is sometimes convenient if one wishes to stop the contagion, or use rational functions in non-rational contexts. Most CRE functions will work on either CRE or non-CRE expressions, but the answers may take different forms. If RATDISREP is given a non-CRE for an argument, it returns its argument unchanged. See also TOTALDISREP.

TOTALDISREP (*exp*) [*Function*]

converts every subexpression of *exp* from CRE to general form. If *exp* is itself in CRE form then this is identical to RATDISREP but if not then RATDISREP would return *exp* unchanged while TOTALDISREP would *totally DISREP* it. This is useful for RATDISREPing expressions e.g., equations, lists, matrices, etc. which have some subexpressions in CRE form.

TELLRAT (*poly*) [*Function*]

adds to the ring of algebraic integers known to **Maxima**, the element which is the solution of the polynomial with integer coefficients. **Maxima** initially knows about %I and all roots of integers. TELLRAT(X); means substitute 0 for X in rational functions. There is a command UNTELLRAT which takes kernels and removes TELLRAT properties. When TELLRATing a multivariate polynomial, e.g. TELLRAT(X**2-Y**2);, there would be an ambiguity as to whether to substitute Y**2 for X**2 or vice versa. The system will pick a particular ordering, but if the user wants to specify which, e.g. TELLRAT(Y**2=X**2); provides a syntax which says replace Y**2 by X**2. TELLRAT and UNTELLRAT both can take any number of arguments, and TELLRAT(); returns a list of the current substitutions. Note: When you TELLRAT reducible polynomials, you want to be careful not to attempt to rationalize a denominator with a zero divisor. E.g. TELLRAT(W**3-1)$ ALGEBRAIC:TRUE$ RAT(1/(W**2-W)); will give quotient by zero. This error can be avoided by setting RATALGDENOM:FALSE.

ALGEBRAIC default: [FALSE] must be set to TRUE in order for the simplification of algebraic integers to take effect.

UNTELLRAT (*x*) [*Function*]

takes kernels and removes TELLRAT properties.

4.2.2 Operations on CRE Expressions

RATDIFF (*exp, var*) [*Function*]

differentiates the rational expression *exp* (which must be a ratio of polynomials or a polynomial in the variable var) with respect to *var*. For rational expressions this is much faster than DIFF. The result is left in CRE form. However, RATDIFF should not be used on factored CRE forms; use DIFF instead for such expressions. See section 8.3 [Differentiation], page 112 for the definition of DIFF.

```
(C1) (4*X**3+10*X-11)/(X**5+5);
                                        3
                                   4 X  + 10 X - 11
(D1)                               ----------------
                                        5
                                       X  + 5
(C2) MODULUS:3$
(C3) MOD(D1);
                                  2
                                 X  + X - 1
(D3)                       --------------------
                            4    3    2
                           X  + X  + X  + X + 1
(C4) RATDIFF(D1,X);
                           5    4    3
                          X  - X  - X  + X - 1
(D4)                -----------------------------
                     8    7    5    4    3
                    X  - X  + X  - X  + X  - X + 1
```

RATVARS (*var1, var2, ...*) [*Function*]

forms its *n* arguments into a list in which the rightmost variable *varn* will be the main variable of future rational expressions in which it occurs, and the other variables will follow in sequence. If a variable is missing from the RATVARS list, it will be given lower priority than the leftmost variable *var1*. The arguments to RATVARS can be either variables or non-rational functions (e.g. SIN(X)). The variable **RATVARS** is a list of the arguments which have been given to this function.

RATWEIGHT (*v1, w1, ..., vn, wn*) [*Function*]

assigns a weight of wi to the variable vi. This causes a term to be replaced by 0 if its weight exceeds the value of the variable RATWTLVL default: FALSE which means no truncation. The weight of a term is the sum of the products of the weight of a variable in the term times its power. Thus the weight of 3*v1**2*v2 is 2*w1+w2. This truncation occurs only when multiplying or exponentiating CRE forms of expressions.

```
(C5) RATWEIGHT(A,1,B,1);
(D5)                          [[B, 1], [A, 1]]
(C6) EXP1:RAT(A+B+1)$
(C7) \%**2;
                     2                   2
(D7)/R/             B  + (2 A + 2) B + A  + 2 A + 1
(C8) RATWTLVL:1$
(C9) EXP1**2;
(D9)/R/                      2 B + 2 A + 1
```

Note: The RATFAC and RATWEIGHT schemes are incompatible and may not both be used at the same time.

RATWEIGHTS [*Variable*]

a list of weight assignments (set up by the RATWEIGHT function), RATWEIGHTS; or RATWEIGHT(); will show you the list. KILL(...,RATWEIGHTS), and SAVE(...,RATWEIGHTS); both work.

RATWTLVL [*variable, default: FALSE*]

used in combination with the RATWEIGHT function to control the truncation of rational (CRE form) expressions (for the default value of FALSE, no truncation occurs).

SHOWRATVARS (*exp*) [*Function*]

returns a list of the RATVARS (CRE variables) of *exp*.

4.2.3 Rational Expression Flags

RATALGDENOM [*variable, default: TRUE*]

if TRUE allows rationalization of denominators with respect to radicals to take effect. To do this one must use CRE form in algebraic mode.

RATFAC [*variable, default: FALSE*]

when TRUE invokes a partially factored form for CRE rational expressions. During rational operations the expression is maintained as fully factored as possible without an actual call to the factor package. This should always save space and may save some time in some computations. The numerator and denominator are still made relatively prime, for example RAT((X**2 -1)**4/(X+1)**2); -> (X-1)**4*(X+1)**2), but the factors within each part may not be relatively prime.

In the CTENSR (Component Tensor Manipulation) Package, if RATFAC is TRUE, it causes the Ricci, Einstein, Riemann, and Weyl tensors and the Scalar Curvature to be factored automatically. This should only be set for cases where the tensorial components are known to consist of few terms.

Note that the RATFAC and RATWEIGHT schemes are incompatible and may not both be used at the same time.

4.3 Selecting Parts of Expressions

4.3.1 Selecting Top Level Expressions

`FIRST` (*exp*) [*Function*]

yields the first part of *exp* which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc. Note that FIRST and its related functions, REST and LAST, work on the form of *exp* which is displayed, not the form which is typed on input. If the variable INFLAG default: [FALSE] is set to TRUE however, these functions will look at the internal form of *exp*. Note that the simplifier re-orders expressions. Thus FIRST(X+Y) will be X if INFLAG is TRUE and Y if INFLAG is FALSE. (FIRST(Y+X) gives the same results).

`LAST` (*exp*) [*Function*]

yields the last part (term, row, element, etc.) of the exp.

`REST` (*exp*, *n*) [*Function*]

yields *exp* with its first *n* elements removed if *n* is positive and its last -*n* elements removed if *n* is negative. If *n* is 1 it may be omitted. *exp* may be a list, matrix, or other expression.

The functions FIRST LAST and REST depend on the value of the variable INFLAG.

`LHS` (*eqn*) [*Function*]

returns the left side of the equation *eqn*.

`RHS` (*eqn*) [*Function*]

returns the right side of the equation *eqn*.

`NUM` (*exp*) [*Function*]

obtains the numerator, *exp1*, of the expression *exp* = *exp1/exp2*.

`DENOM` (*exp*) [*Function*]

returns the denominator of the expression *exp*.

`RATNUMER` (*exp*) [*Function*]

obtains the numerator of the rational expression *exp*. If *exp* is in general form then the NUM function should be used instead, unless one wishes to get a CRE form.

RATDENOM (*exp*) [*Function*]

obtains the denominator of the rational expression *exp*. If *exp* is in general form then the DENOM function should be used instead, unless one wishes to get a CRE form.

NUMFACTOR (*exp*) [*Function*]

gives the numerical factor multiplying the expression *exp*, which should be a single term. If the gcd of all the terms in a sum is desired the CONTENT function may be used.

```
(C1) GAMMA(7/2);
(D1)                15 SQRT(\%PI)
                    ------------
                         8
(C2) NUMFACTOR(\%)
                     15
(D2)                 --
                      8
```

See section 1.4.4 [Data Type Coercion], page 9, for the definition of REALPART and IMAGPART.

4.3.2 Isolating and Revealing Expressions

REVEAL (*exp*, *depth*) [*Function*]

will display *exp* to the specified integer *depth* with the length of each part indicated. Sums will be displayed as Sum(n) and products as Product(n) where *n* is the number of subparts of the sum or product. Exponentials will be displayed as Expt.

```
(C1) INTEGRATE(1/(X^3+2),X)$
(C2) REVEAL(\%,2);
(D2)                    Negterm + Quotient + Quotient
(C3) REVEAL(D1,3);
                                        ATAN          LOG
(D3)                 - Quotient + ---------- + ----------
                                  Product(2)   Product(2)
```

PICKAPART (*exp*, *depth*) [*Function*]

will assign E labels to all subexpressions of *exp* down to the specified integer *depth*. This is useful for dealing with large expressions and for automatically assigning parts of an expression to a variable without having to use the part functions.

```
(C1) EXP:(A+B)/2+SIN(X^2)/3-LOG(1+SQRT(X+1));
                                                 2
                                              SIN(X )   B + A
(D1)              - LOG(SQRT(X + 1) + 1) + ------- + -----
                                                 3        2
(C2) PICKAPART(\%,1);
(E2)                    - LOG(SQRT(X + 1) + 1)
                                      2
                                   SIN(X )
(E3)                               -------
                                      3
                                   B + A
(E4)                               -----
                                     2
(D4)                            E4 + E3 + E2
```

ISOLATE (*exp, var*) [*Function*]

returns *exp* with subexpressions which are sums and which do not contain *var* replaced by intermediate expression labels (these being atomic symbols like E1, E2, ...). This is often useful to avoid unnecessary expansion of subexpressions which don't contain the variable of interest. Since the intermediate labels are bound to the subexpressions, they can all be substituted back by evaluating the expression in which they occur.

EXPTISOLATE [*variable, default: FALSE*]

if TRUE will cause ISOLATE(exp, var); to examine exponents of atoms (like %E) which contain *var*.

ISOLATE_WRT_TIMES [*variable, default: FALSE*]

if set to TRUE, then ISOLATE will also isolate with respect to products. E.g. compare both settings of the switch on ISOLATE(EXPAND((A+B+C)**2),C);.

DISOLATE (*exp, var1, var2, ...*) [*Function*]

is similar to ISOLATE(*exp, var*) except that it enables the user to isolate more than one variable simultaneously. This might be useful, for example, if one were attempting to change variables in a multiple integration, and that variable change involved two or more of the integration variables. To access this function, do LOAD(DISOL); . For example

```
DISOLATE(A*(B*(C+D) + E*(F+G)), A, B, E);
```

returns a form similar to A (B E1 + E E2), where E1 is bound to C + D and E2 is bound to F + G. There is a demo in the file: '`share2/disol.dem`'. There are some usage notes in the file: '`share2/disol.usg`'.

DISPTERMS (*exp*) [*Function*]

displays its argument in parts one below the other. That is, first the operator of *expr* is displayed, then each term in a sum, or factor in a product, or part of a more general expression is displayed separately. This is useful if *exp* is too large to be otherwise displayed. For example if P1, P2, ... are very large expressions then the display program may run out of storage space in trying to display P1+P2+... all at once. However, DISPTERMS(P1+P2+...) will display P1, then below it P2, etc. When not using DISPTERMS, if an exponential expression is too wide to be displayed as A**B it will appear as EXPT(A,B) (or as NCEXPT(A,B) in the case of A^^B).

4.3.2.1 Isolating Expressions with Boxes

BOX (*exp*) [*Function*]

returns *exp* enclosed in a box. The box is actually part of the expression.

BOX*(exp, label)* encloses *exp* in a labelled box. *label* is a name which will be truncated in display if it is too long.

BOXCHAR [*variable, default: "*]

is the character used to draw the box in the BOX and in the DPART and LPART functions.

REMBOX (*exp, arg*) [*Function*]

removes boxes from *exp* according to *arg*. If *arg* is UNLABELED then all unlabelled boxes are removed. If *arg* is the name of some label, then only boxes with that label are removed. If *arg* is omitted then all boxes labelled and unlabelled are removed.

4.3.3 Selecting Sub Expressions

PART (*exp, n1, ..., nk*) [*Function*]

deals with the displayed form of *exp*. It obtains the part of *exp* as specified by the indices n1,...,nk. First part n1 of *exp* is obtained, then part n2 of that, etc. The result is part nk of ... part n2 of part n1 of *exp*. Thus PART(Z+2*Y,2,1) yields 2. PART can be used to obtain an element of a list, a row of a matrix, etc. If the last argument to a Part function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus PART(X+Y+Z,[1,3]) -> Z+X. ALLBUT works with the PART commands (i.e. PART, INPART, SUBSTPART, SUBSTINPART, DPART, and LPART) to select all but the parts referred to by its arguments. For example,

```
EXPR: E+D+C+B+A;
PART(EXP,[2,5]); -> D+A;
PART(EXP,ALLBUT(2,5)); -> E+C+B;
```

PARTSWITCH [*variable, default: FALSE*]

if set to TRUE then END is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

PIECE [*Variable*]

holds the last expression selected when using the part functions. It is set during the execution of the function and thus may be referred to in the function itself.

INPART (*exp, n1, . . . , nk*) [*Function*]

is similar to PART but works on the internal representation of the expression rather than the displayed form, and thus may be faster since no formatting is done. Care should be taken with respect to the order of subexpressions in sums and products (since the order of variables in the internal form is often different from that in the displayed form) and in dealing with unary minus, subtraction, and division (since these operators are removed from the expression). PART(X+Y,0) or INPART(X+Y,0) yield +, though in order to refer to the operator it must be enclosed in double quotes. For example IF INPART(D9,0)="+" THEN . . .

```
(C1)  X+Y+W*Z;
(D1)                   W Z + Y + X
(C2)  INPART(D1,3,2);
(D2)                   Z
(C3)  PART(D1,1,2);
(D3)                   Z
(C4) 'LIMIT(F(X)**G(X+1),X,0,MINUS);
                                      G(X + 1)
(D4)                       LIMIT   F(X)
                           X ->0-
(C5) INPART(\%,1,2);
(D5)                             G(X + 1)
```

DPART (*exp, n1, . . . , nk*) [*Function*]

selects the same subexpression as PART, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. The box is actually part of the expression.

```
(C1) DPART(X+Y/Z**2,1,2,1);
                         Y
(D1)                     ---- + X
                           2
                        *****
                        * Z *
                        *****
```

LPART (*string, exp, n1, . . . , nk*) [*Function*]

is similar to DPART but uses a labelled box. A labelled box is similar to the one produced by DPART but it has the name *string* in the top line. See section 4.3.2.1 [Isolating Expressions with Boxes], page 58.

4.3.4 Analysing Expressions

LENGTH (*exp*) [*Function*]

gives (by default) the number of parts in the external (displayed) form of *exp*. For lists this is the number of elements, for matrices it is the number of rows, and for sums it is the number of terms. (See also DISPFORM). The LENGTH command is affected by the variable INFLAG default: FALSE. So, e.g. LENGTH(A/(B*C)); -> 2 if INFLAG is FALSE (and assuming EXPTDISPFLAG is TRUE), but 3 if INFLAG is TRUE. (The internal representation of the above is essentially A*B**-1*C**-1.)

NTERMS (*exp*) [*Function*]

gives the number of terms that *exp* would have if it were fully expanded out and no cancellations or combination of terms occurred. Note that expressions like SIN(E), SQRT(E), EXP(E), etc., count as just one term regardless of how many terms E has (if it is a sum).

ARGS (*exp*) [*Function*]

returns a list of the args of *exp*. I.e. it is essentially equivalent to SUBSTPART("[",exp,0).

Both ARGS and SUBSTPART depend on the setting of INFLAG.

COEFF (*exp, var, int*) [*Function*]

obtains the coefficient of *var****int* in *exp*. *int* may be omitted if it is 1. *var* may be an atom, or complete subexpression of *exp* e.g., X, SIN(X), A[I+1], X+Y, etc. (In the last case the expression (X+Y) should occur in *exp*). Sometimes it may be necessary to expand or factor *exp* in order to make *var****int* explicit. This is not done automatically by COEFF.

```
(C1) COEFF(2*A*TAN(X)+TAN(X)+B=5*TAN(X)+3,TAN(X));
(D1)                          2 A + 1 = 5
(C2) COEFF(Y+X*\%E**X+1,X,0);
(D2)                             Y + 1
```

BOTHCOEF (*exp, var*) [*Function*]

returns a list whose first member is the coefficient of *var* in *exp* (as found by RATCOEF if *exp* is in CRE form, otherwise by COEFF), and whose second member is the remaining part of *exp*. That is, [A,B] where exp=A*var+B.

```
(C1) ISLINEAR(EXP,VAR):=BLOCK([C],
        C:BOTHCOEF(RAT(EXP,VAR),VAR),
        IS(FREEOF(VAR,C) AND C[1]#0))$
(C2) ISLINEAR((R**2-(X-R)**2)/X,X);
(D2)                                TRUE
```

RATCOEF (*exp, var, n*) [*Function*]

returns the coefficient, C, of the expression *var***n* in the expression *exp*. *n* may be omitted if it is 1. C will be free (except possibly in a non-rational sense) of the variables in *var*. If no coefficient of this type exists, zero will be returned. RATCOEF expands and rationally simplifies its first argument and thus it may produce answers different from those of COEFF which is purely syntactic. Thus RATCOEF((X+1)/Y+X,X); -> (Y+1)/Y whereas COEFF returns 1. RATCOEF(exp, var, 0), viewing *exp* as a sum, gives a sum of those terms which do not contain *var*. Therefore if *var* occurs to any negative powers, RATCOEF should not be used. Since *exp* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

```
(C1) S:A*X+B*X+5$
(C2) RATCOEF(S,A+B);
(D2)                X
```

FREEOF (*x1, x2, . . . , exp*) [*Function*]

yields TRUE if the *xi* do not occur in *exp*, and FALSE otherwise. The *xi* are atoms or they may be subscripted names, functions (e.g. SIN(X)), or operators enclosed in double quotes. If *var* is a dummy variable of *exp*, then FREEOF(var,exp); will return TRUE. Dummy variables are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable. Example: FREEOF(I,'SUM(F(I),I,0,N)); -> TRUE.

DISPFORM(exp,ALL) converts the entire expression (not just the top-level) to external format. For example, if EXP:SIN(SQRT(X)), then FREEOF(SQRT,EXP) and FREEOF(SQRT, DISPFORM(EXP)) give TRUE, while FREEOF(SQRT,DISPFORM(EXP,ALL)) gives FALSE.

LISTOFVARS (*exp*) [*Function*]

yields a list of the variables in *exp*.

```
(C1) LISTOFVARS(F(X[1]+Y)/G**(2+A));
(D1)                           [X[1], Y, A, G]
```

LISTCONSTVARS [*variable, default: FALSE*]

if TRUE will cause LISTOFVARS to include %E, %PI, %I, and any variables DECLAREd CONSTANT in the list it returns if they appear in the expression LISTOFVARS is called on. The default is to omit these.

LISTDUMMYVARS [*variable, default: TRUE*]

if FALSE, dummy variables in the expression will not be included in the list returned by LISTOFVARS. (The meaning of dummy variables is as given in the section on FREEOF. Dummy variables are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable.) Example: LISTOFVARS('SUM(F(I),I,0,N)); -> [I,N] if LISTDUMMYVARS is TRUE, and [N] if LISTDUMMYVARS is FALSE.

PARTITION (*exp*, *var*) [*Function*]

returns a list of two expressions. They are (1) the factors of *exp* (if it is a product), the terms of *exp* (if it is a sum), or the list (if it is a list), which don't contain *var* and, (2) the factors, terms, or list which do.

```
(C1) PARTITION(2*A*X*F(X),X);
(D1)                 [ 2 A , X F(X) ]
(C2) PARTITION(A+B,X);
(D2)                 [ A + B , 0 ]
(C3) PARTITION([A,B,F(A),C],A);
(D3)                [[B,C],[A,F(A)]]
```

HIPOW (*exp*, *var*) [*Function*]

the highest explicit exponent of *var* in *exp*. Sometimes it may be necessary to expand *exp* since this is not done automatically by HIPOW. Thus HIPOW(Y**3*X**2+X*Y**4,X) -> 2.

LOPOW (*exp*, *var*) [*Function*]

the lowest exponent of *var* which explicitly appears in *exp*. Thus LOPOW((X+Y)**2+(X+Y)**A,X+Y) -> MIN(A,2).

POWERS (*exp*, *var*) [*Function*]

gives the powers of *var* occurring in *exp*. This function is a generalisation of HIPOW and LOPOW in that it returns a list of all the powers of *var* occurring in *exp*. it is still necessary to expand *exp* before applying powers (on pain of getting the wrong answer).

This function has many uses, e.g. if you want to find all the coefficients of x in a polynomial poly you can use map(lambda([pow],coeff(poly,x,pow)),powers(poly,x)); and many other similar useful hacks. There are some usage notes in the file: '`share2/powers.usg`'.

See section 8.3 [Differentiation], page 112 for the definition of DERIVDEGREE.

4.4 Manipulating Lists

APPEND (*list1*, *list2*, ...) [*Function*]

returns a single list of the elements of *list1* followed by the elements of *list2*,... APPEND also works on general expressions, e.g. APPEND(F(A,B), F(C,D,E)); -> F(A,B,C,D,E).

APPLY (*fun, list*) [*Special Form*]

gives the result of applying the function *fun* to the list of its arguments *list*. This is useful when it is desired to compute the arguments to a function before applying that function. For example, if L is the list [1, 5, -10.2, 4, 3], then APPLY(MIN, L) gives -10.2. APPLY is also useful when calling functions which do not have their arguments evaluated if it is desired to cause evaluation of them. For example, if FILESPEC is a variable bound to the list [TEST, CASE], then APPLY(CLOSEFILE,FILESPEC) is equivalent to CLOSEFILE(TEST,CASE). In general the first argument to APPLY should be preceded by a ' to make it evaluate to itself. Since some atomic variables have the same name as certain functions, the values of the variable would be used rather than the function because APPLY has its first argument evaluated as well as its second.

COPYLIST (*list*) [*Function*]

creates a copy of the list *list*.

CONCAT (*arg1, arg2, ...*) [*Function*]

evaluates its arguments and returns the concatenation of their values resulting in a name, or a quoted string the type being given by that of the first argument. Thus if X is bound to 1 and D is unbound, then CONCAT(X,2)="12" and CONCAT(D,X+1)=D2.

CONS (*exp, list*) [*Function*]

returns a new list constructed of the element *exp* as its first element, followed by the elements of list. CONS also works on other expressions, e.g. CONS(X, F(A,B,C)); -> F(X,A,B,C).

ENDCONS (*exp, list*) [*Function*]

returns a new list consisting of the elements of *list* followed by *exp*. ENDCONS also works on general expressions, e.g. ENDCONS(X, F(A,B,C)); -> F(A,B,C,X).

DELETE (*exp1, list*) [*Function*]

removes all occurrences of *exp1* from *list*. *exp1* may be a term of *list* (if it is a sum), or a factor of *list* (if it is a product).

```
(C1)  DELETE(SIN(X),X+SIN(X)+Y);
(D1)                Y + X
```

DELETE(exp1, list, int) removes the first *int* occurrences of *exp1* from *list*. Of course, if there are fewer than *int* occurrences of *exp1* in *list* then all occurrences will be deleted.

MAKELIST (*exp, var, lo, hi*) [*Function*]

returns a list as value. MAKELIST may be called as MAKELIST(exp, var, lo, hi) [*lo* and *hi* must be integers], or as MAKELIST(exp, var, list). In the first case MAKELIST is analogous to SUM, whereas in the second case MAKELIST is similar to MAP. Examples:

```
MAKELIST(CONCAT(X,I),I,1,6) -> [X1,X2,X3,X4,X5,X6]
MAKELIST(X=Y,Y,[A,B,C]) -> [X=A,X=B,X=C]
```

MEMBER (*exp, list*) [*Function*]

returns TRUE if *exp* occurs as a member of *list* (not within a member). Otherwise FALSE is returned. Member also works on non-list expressions, e.g. MEMBER(B, F(A,B,C)); -> TRUE.

REVERSE (*list*) [*Function*]

reverses the order of the members of the list (not the members themselves). REVERSE also works on general expressions, e.g. REVERSE(A=B); gives B=A.

4.4.1 Sorting Lists

SORT (*list, optional-predicate*) [*Function*]

sorts the list *list* using a suitable *optional-predicate* of two arguments (such as < or ORDERLESSP). If the *optional-predicate* is not given, then **Maxima**'s built-in ordering predicate is used.

4.5 Mapping Functions

MAP (*fun(arg1, . . . , argn), exp1, . . . , expn*) [*Special Form*]

returns an expression whose leading operator is the same as that of the *expi* but whose subparts are the results of applying *fun* to the corresponding subparts of the *expi*. *Fun* is either the name of a function of *n* arguments (where *n* is the number of *expi*), or is a LAMBDA form of *n* arguments.

MAPERROR [*variable, default: TRUE*]

if FALSE will cause all of the mapping functions (e.g. MAP(fun, exp1, exp2, . . .)) to (1) stop when they finish going down the shortest *expi* if not all of the *expi* are of the same length, and (2) apply *fun* to *[exp1, exp2, . . .]* if the *expi* are not all the same type of object. If MAPERROR is TRUE then an error message will be given in the above two instances. One of the uses of this function is to MAP a function (e.g. PARTFRAC) onto each term of a very large expression where it ordinarily wouldn't be possible to use the function on the entire expression due to an exhaustion of list storage space in the course of the computation.

```
(C1) MAP(F,X+A*Y+B*Z);
(D1)                          F(B Z) + F(A Y) + F(X)
(C2) MAP(LAMBDA([U],PARTFRAC(U,X)),X+1/(X^3+4*X^2+5*X+2));
                             1       1        1
(D2)                      ----- - ----- + -------- + X
                          X + 2   X + 1          2
                                              (X + 1)
(C3) MAP(RATSIMP, X/(X^2+X)+(Y^2+Y)/Y);
                                        1
(D3)                              Y + ----- + 1
                                    X + 1
(C4) MAP("=",[A,B],[-0.5,3]);
(D4)                            [A = - 0.5, B = 3]
```

See section 4.6.1 [Substitution Flags], page 68 for the definition of INFLAG, which effects the behaviour of the mapping functions.

FULLMAP (*fun(arg1, ..., argn), exp1, ..., expn*) [*Special Form*]

is similar to MAP but it will keep mapping down all subexpressions until the main operators are no longer the same. The user should be aware that FULLMAP is used by the **Maxima** simplifier for certain matrix manipulations; thus, the user might see an error message concerning FULLMAP even though FULLMAP was not explicitly called by the user.

```
(C1) A+B*C$
(C2) FULLMAP(G,\%);
(D2)                G(B) G(C) + G(A)
(C3)  MAP(G,D1);
(D3)                  G(B C) + G(A)
```

FULLMAPL (*fun(arg1, ..., argn), exp1, ..., expn*) [*Special Form*]

is similar to FULLMAP but it only maps onto lists and matrices.

```
(C1) FULLMAPL("+",[3,[4,5]],[[A,1],[0,-1.5]]);
(D1)                        [[A + 3, 4], [4, 3.5]]
```

MAPLIST (*fun(arg1, ..., argn), exp1, ..., expn*) [*Special Form*]

yields a list of the applications of *fun* to the parts of the *expi*. This differs from MAP(fun, exp1, exp2, ...) which returns an expression with the same main operator as *expi* has (except for simplifications and the case where MAP does an APPLY). *fun* is of the same form as in MAP.

SCANMAP (*function, exp*) [*Special Form*]

recursively applies function to *exp*, in a top down manner. This is most useful when complete factorization is desired, for example:

```
(C1) EXP:(A^2+2*A+1)*Y + X^2$
(C2) SCANMAP(FACTOR,EXP);
                                      2      2
(D2)                          (A + 1)  Y + X
```

Note the way in which SCANMAP applies the given function FACTOR to the constituent subexpressions of *exp*; if another form of *exp* is presented to SCANMAP then the result may be different. Thus, D2 is not recovered when SCANMAP is applied to the expanded form of *exp*:

```
(C3) SCANMAP(FACTOR,EXPAND(EXP));
                             2                    2
(D3)                        A  Y + 2 A Y + Y + X
```

Here is another example of the way in which SCANMAP recursively applies a given function to all subexpressions, including exponents:

```
(C4) EXPR : U*V^(A*X+B) + C$
(C5) SCANMAP('F, EXPR);
                      F(F(F(A) F(X)) + F(B))
(D5) F(F(F(U) F(F(V)                       )) + F(C))
```

SCANMAP(*function, expression, BOTTOMUP*) applies function to *exp* in a bottom-up manner. E.g., for undefined F,

```
SCANMAP(F,A*X+B) ->
        F(A*X+B) ->
        F(F(A*X)+F(B)) ->
        F(F(F(A)*F(X))+F(B)).

SCANMAP(F,A*X+B,BOTTOMUP) ->
        F(A)*F(X)+F(B) ->
        F(F(A)*F(X))+F(B) ->
        F(F(F(A)*F(X))+F(B)).
```

(In this case, you get the same answer both ways.)

SUBLIST (*list, fun*) [*Function*]

returns the list of elements of the list *list* for which the function *fun* returns TRUE. E.g., SUBLIST([1,2,3,4],EVENP); →[2,4]. SUBLIS may be used to efficiently cause the evaluation of specific noun forms in an expression.

```
EXP:(F(X)+G(C+B+A,SIN(COS(7*Q)),'H(X)))^(Q^2+P^3/2+M(A,B,C))+V$
H(X):=X^2$

EV(EXP,H) takes 62 msecs.
SUBLIS([NOUNIFY('H)=lambda([u],h(u))],exp) takes 28 msecs.
```

MAPATOM (*exp*) [*Function*]

is TRUE if and only if *exp* is treated by the *MAP*ping routines as an atom, a unit. "MAPATOMs are atoms, numbers (including rational numbers), and subscripted variables.

4.6 Substituting Expressions

SUBST (*a, b, c*) [*Function*]

substitutes *a* for *b* in *c*. *b* must be an atom, or a complete subexpression of *c*. For example, X+Y+Z is a complete subexpression of 2*(X+Y+Z)/W while X+Y is not. When *b* does not have these characteristics, one may sometimes use SUBSTPART or RATSUBST (see below). Alternatively, if *b* is of the form *e/f* then one could use SUBST(a*f,e,c) while if *b* is of the form *e**(1/f)* then one could use SUBST(a**f,e,c). The SUBST command also discerns the X**Y in X**-Y so that SUBST(A,SQRT(X),1/SQRT(X)) -> 1/A. *a* and *b* may also be operators of an expression enclosed in double quotes or they may be function names. If one wishes to substitute for the independent variable in derivative forms then the AT function (see below) should be used. Note: **SUBST** is an alias for **SUBSTITUTE**.

SUBST*(eq1, exp)* or SUBST(*[eq1,...,eqk], exp*) are other permissible forms. The *eqi* are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression *exp*.

LRATSUBST (*list, exp*) [*Function*]

is analogous to SUBST(*list_of_equations,exp*) except that it uses RATSUBST instead of SUBST. The first argument of LRATSUBST must be an equation or a list of equations identical in format to that accepted by SUBST. The substitutions are made in the order given by the list of equations, that is, from left to right. There is a demo in the file: '`share2/lrats.dem`'. There are some usage notes in the file: '`share2/lrats.usg`'.

SUBLIS (*list, exp*) [*Function*]

allows multiple substitutions into an expression in parallel. Each element of the list must be of the form *symbol = expression*. The new expression, with appropriate substitutions made, is the value returned. SUBLIS will preserve sharing where possible. eg, SUBLIS([A=B],C+D); returns a pointer to the original C+D since no substitution is needed. SUBLIS does substitutions in parallel. eg,

```
SUBLIS([A=B,B=A],SIN(A)+COS(B));
 => SIN(B) + COS(A)
```

SUBLIS_APPLY_LAMBDA [*variable, default: TRUE*]

controls whether LAMBDA's substituted are applied in simplification after SUBLIS is used or whether you have to do an EV to get things to apply. TRUE means do the application.

```
SUBLIS_APPLY_LAMBDA:TRUE$
SUBLIS([F=LAMBDA([X],X+1)],F(Y));
        => Y+1
SUBLIS_APPLY_LAMBDA:FALSE$
SUBLIS([F=LAMBDA([X],X+1)],F(Y));
         => LAMBDA([X],X+1)(Y).
```

For full documentation, see the file '`share2/sublis.nfo`'.

4.6.1 Substitution Flags

`INFLAG` [*variable, default: FALSE*]

if set to TRUE, the functions for part extraction will look at the internal form of *exp*. Note that the simplifier re-orders expressions. Thus FIRST(X+Y) will be X if INFLAG is TRUE and Y if INFLAG is FALSE. (FIRST(Y+X) gives the same results). Also, setting INFLAG to TRUE and calling PART or SUBSTPART is the same as calling INPART or SUBSTINPART. Functions affected by the setting of INFLAG are: PART, SUBSTPART, FIRST, REST, LAST, LENGTH, the FOR ... IN construct, MAP, FULLMAP, MAPLIST, REVEAL and PICKAPART.

`EXPTSUBST` [*variable, default: FALSE*]

if TRUE permits substitutions such as Y for %E**X in %E**(A*X) to take place.

`OPSUBST` [*variable, default: TRUE*]

if FALSE, SUBST will not attempt to substitute into the operator of an expression. E.g. (OPSUBST:FALSE, SUBST(X**2,R,R+R[0])); will work.

```
(C1) SUBST(A,X+Y,X+(X+Y)**2+Y);
                                2
(D1)                     Y + X + A
(C2) SUBST(-\%I,\%I,A+B*\%I);
(D2)                              A - \%I B
```

Note that (C2) is one way of obtaining the complex conjugate of an analytic expression.

`DERIVSUBST` [*variable, default: FALSE*]

controls non-syntactic substitutions such as SUBST(X,'DIFF(Y,T),'DIFF(Y,T,2));. If DERIVSUBST is set to TRUE, this gives 'DIFF(X,T).

4.6.2 Substituting in CRE Expressions

RATSUBST (*a, b, c*) [*Function*]

substitutes *a* for *b* in *c*. *b* may be a sum, product, power, etc. RATSUBST knows something of the meaning of expressions whereas SUBST does a purely syntactic substitution. Thus SUBST(A,X+Y,X+Y+Z); -> X+Y+Z whereas RATSUBST would return Z+A.

FULLRATSUBST (*a, b, c*) [*Function*]

is the same as RATSUBST except that it calls itself recursively on its result until that result stops changing. This function is useful when the replacement expression and the replaced expression have one or more variables in common.

FULLRATSUBST will also accept its arguments in the format of LRATSUBST. That is, the first argument may be a single substitution equation or a list of such equations, while the second argument is the expression being processed. Do LOAD(LRATS); to use this function. There is a demo in the file: '`share2/lrats.dem`'. There are some usage notes in the file: '`share2/lrats.usg`'.

RADSUBSTFLAG [*variable, default: FALSE*]

if TRUE permits RATSUBST to make substitutions such as U for SQRT(X) in X.

4.6.3 Partial Substitutions

SUBSTPART (*x, exp, n1, n2, ...*) [*Special Form*]

substitutes *x* for the subexpression picked out by the rest of the arguments as in PART. It returns the new value of *exp*. *x* may be some operator to be substituted for an operator of *exp*. In some cases it needs to be enclosed in double quotess; e.g. SUBSTPART("+",A*B,0); →B + A.

```
(C1) 1/(X**2+2);
                                    1
(D1)                              ------
                                   2
                                  X  + 2
(C2) SUBSTPART(3/2,\%,2,1,2);
                                    1
(D2)                              --------
                                   3/2
                                  X    + 2
(C3) A*X+F(B,Y);
(D3)                           A X + F(B, Y)
(C4) SUBSTPART("+",\%,1,0);
(D4)                         X + F(B, Y) + A
```

Also, setting the option INFLAG to TRUE and calling PART/SUBSTPART is the same as calling INPART/SUBSTINPART. See section 4.3.3 [Selecting Sub Expressions], page 58 for the definition of PARTSWITCH.

SUBSTINPART (*x, exp, n1, n2, ...*) [*Special Form*]

is like SUBSTPART but works on the internal representation of *exp*.

```
(C1) X.'DIFF(F(X),X,2);
                                    2
                                   d
(D1)                          X . (--- F(X))
                                    2
                                   dX
(C2) SUBSTINPART(D**2,\%,2);
                                       2
(D2)                                 X . D
(C3) SUBSTINPART(F1,F[1](X+1),0);
(D3)                               F1(X + 1)
```

If the last argument to a part function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus PART(X+Y+Z,[1,3]) -> Z+X.

PIECE holds the value of the last expression selected when using the part functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below. If PARTSWITCH default: [FALSE] is set to TRUE then END is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

```
(C1)  27*Y**3+54*X*Y**2+36*X**2*Y+Y+8*X**3+X+1;
            3         2       2            3
(D1)    27 Y  + 54 X Y  + 36 X  Y + Y + 8 X  + X + 1
(C2)  PART(D1,2,[1,3]);
                  2
(D2)           54 Y
(C3)  SQRT(PIECE/54);
                Y
(C4)  SUBSTPART(FACTOR(PIECE),D1,[1,2,3,5]);
                          3
(D4)            (3 Y + 2 X) + Y + X + 1
(C5) 1/X+Y/X-1/Z;
                                1   Y   1
(D5)                          - - + - + -
                                Z   X   X
(C6) SUBSTPART(XTHRU(PIECE),\%,[2,3]);
                                Y + 1   1
(D6)                            ----- - -
                                  X     Z
```

Also, setting the option INFLAG to TRUE and calling PART/SUBSTPART is the same as calling INPART/SUBSTINPART.

CHAPTER

FIVE

Simplifying Expanding and Factoring

5.1 Simplifying Expressions

`SCSIMP` (*exp, rule1, rule2, ...*) [*Function*]

(Sequential Comparative Simplification [Stoute]) takes an expression (its first argument) and a set of identities, or rules (its other arguments) and tries simplifying. If a smaller expression is obtained, the process repeats. Otherwise after all simplifications are tried, it returns the original answer. See section 10.1.7.1 [Defining Simplification Rules], page 154.

See section 4.1 [Evaluation], page 47. See section 2.4 [Flags Effecting the Displayed Form], page 21.

5.1.1 Simplifying CRE Expressions

`RATSIMP` (*exp*) [*Function*]

"rationally" simplifies (similar to RATEXPAND) the expression *exp* and all of its subexpressions including the arguments to non-rational functions. The result is returned as the quotient of two polynomials in a recursive form, i.e. the coefficients of the main variable are polynomials in the other variables. Variables may, as in RATEXPAND, include non-rational functions (e.g. SIN(X**2+1)) but with RATSIMP, the arguments to non-rational functions are rationally simplified. Note that RATSIMP is affected by some of the variables which affect RATEXPAND.

RATSIMP*(exp, v1, v2, ...)* enables rational simplification with the specification of variable ordering as in RATVARS.

```
(C1) SIN(X/(X^2+X))=\%E^((LOG(X)+1)**2-LOG(X)**2);
                                                2          2
                    X           (LOG(X) + 1)  - LOG (X)
(D1)          SIN(------) = \%E
                   2
                  X  + X
(C2) RATSIMP(\%);
                                  1          2
(D2)                        SIN(-----) = \%E X
                                X + 1
(C3) ((X-1)**(3/2)-(X+1)*SQRT(X-1))/SQRT((X-1)*(X+1));
                      3/2
                (X - 1)    - SQRT(X - 1) (X + 1)
(D3)            --------------------------------
                    SQRT(X - 1) SQRT(X + 1)
(C4) RATSIMP(\%);
                                  2
(D4)                        - -----------
                              SQRT(X + 1)
(C5)  X**(A+1/A),RATSIMPEXPONS:TRUE;
                    2
                   A  + 1
                   ------
                     A
(D5)              X
```

RATSIMPEXPONS [*variable, default: FALSE*]

if TRUE will cause exponents of expressions to be RATSIMPed automatically during simplification.

FULLRATSIMP (*exp*) [*Function*]

When non-rational expressions are involved, one call to RATSIMP followed as is usual by non-rational (general) simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. The command FULLRATSIMP makes this process convenient. FULLRATSIMP repeatedly applies RATSIMP followed by non-rational simplification to an expression until no further change occurs. For example, consider the

```
EXP:                  (X^(A/2)+1)^2*(X^(A/2)-1)^2/(X^A-1);
RATSIMP(EXP); ->      (X^(2*A)-2*X^A+1)/(X^A-1)
FULLRATSIMP(EXP); ->   X^A-1
```

The problem may be seen by looking at

```
RAT(EXP); ->  ((X^(A/2))^4-2*(X^(A/2))^2+1)/(X^A-1)
```

FULLRATSIMP*(exp, var1, var2, ...)* takes one or more arguments similar to RATSIMP and RAT.

5.1.2 Simplifying Trig Expressions

`TRIGSIMP` (*exp*) [*Function*]

employs the identities sin(x)**2 + cos(x)**2 = 1 and cosh(x)**2 - sinh(x)**2 = 1 to simplify expressions containing TAN, SEC, etc. to SIN, COS, SINH, COSH, so that further simplification may be obtained by using TRIGREDUCE on the result. There is a demo in the file: '`share/trgsmp.dem`'. There are some usage notes in the file: '`share/trgsmp.usg`'.

The file '`share1/ntrig.mc`' allows **Maxima** to compute trig functions with arguments of the form N*%PI/10 for integer N. Do LOAD(NTRIG); to access this. The main functions are **USIN** and **UCOS**.

The file '`share1/atrig1.mc`' contains several additional simplification rules for inverse trig functions. Together with rules already known to **Maxima**, the following angles are fully implemented: 0, %PI/6, %PI/4, %PI/3, and %PI/2. Corresponding angles in the other three quadrants are also available. Do LOAD(ATRIG1); to use them.

`TRIGREDUCE` (*exp*, *var*) [*Function*]

combines products and powers of trigonometric and hyperbolic SINs and COSs of *var*, into of multiples of the angle *var*. It also tries to eliminate these functions when they occur in denominators. If *var* is omitted then all variables in *exp* are used. Also see the POISSIMP function.

```
(C4) TRIGREDUCE(-SIN(X)^2+3*COS(X)^2+X);
(D4)                        2 COS(2 X) + X + 1
```

The trigonometric simplification routines will use DECLAREd information in some simple cases. Declarations about variables are used as follows, e.g.

```
(C5) DECLARE(J, INTEGER, E, EVEN, O, ODD)$
(C6) SIN(X + (E + 1/2)*\%PI)$
(D6)                         COS(X)
(C7) SIN(X + (O + 1/2) \%PI);
(D7)                       - COS(X)
```

5.1.3 Simplifying Logarithms and Exponentials

`RADCAN` (*exp*) [*Function*]

simplifies *exp*, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, RADCAN produces a regular form. Two equivalent expressions in this class will not necessarily have the same appearance, but their difference will be simplified by RADCAN to zero. For some expressions RADCAN can be quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents. See section 2.4.2 [Display of

Exponentials], page 24 for the definition of DEMOIVRE, %EDISPFLAG, %EMODE, %E_TO_NUMLOG, EXPTDISPFLAG, and RADEXPAND. There is a demo in the file: '`demo/radcan.dem`'.

`LOGCONTRACT` (*exp*) [*Function*]

recursively scans an *exp*, transforming subexpressions of the form a1*LOG(b1) + a2*LOG(b2) + c into LOG(RATSIMP(b1**a1 * b2**a2)) + c.

```
(C1) 2*(A*LOG(X) + 2*A*LOG(Y))$
(C2) LOGCONTRACT(\%);
                                          2  4
(D3)                               A LOG(X  Y )
```

If you do DECLARE(N,INTEGER); then LOGCONTRACT(2*A*N*LOG(X)); gives A*LOG(X**(2*N)). The coefficients that contract in this manner are those such as the 2 and the N here which satisfy FEATUREP(coeff,INTEGER). The user can control which coefficients are contracted by setting the option LOGCONCOEFFP default: [FALSE] to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do

```
LOGCONCOEFFP:'LOGCONFUN$
LOGCONFUN(M):=FEATUREP(M,INTEGER) OR RATNUMP(M);.
```

Then LOGCONTRACT(1/2*LOG(X)); -> LOG(SQRT(X)).

`LOGCONCOEFFP` [*variable, default: FALSE*]

controls which coefficients are contracted when using LOGCONTRACT. It may be set to the name of a predicate function of one argument.

See section 2.4.3 [Display of Logarithms], page 25 for the definition of LOGNEGINT, LOGEXPAND, LOGNUMER and LOGSIMP.

5.1.4 Simplifying Factorials

`FACTCOMB` (*exp*) [*Function*]

tries to combine the coefficients of factorials in *exp* with the factorials themselves by converting, for example, (N+1)*N! into (N+1)!.

`SUMSPLITFACT` [*variable, default: TRUE*]

if set to FALSE will cause MINFACTORIAL to be applied after a FACTCOMB.

```
(C1) (N+1)^B*N!^B;
                                        B   B
(D1)                                 (N + 1)  N!
(C2) FACTCOMB(\%);
                                           B
(D1)                                  (N + 1)!
```

`MINFACTORIAL` (*exp*) [*Function*]

examines *exp* for occurrences of two factorials which differ by an integer. It then turns one into a polynomial times the other. If *exp* involves binomial coefficients then they will be converted into ratios of factorials.

```
(C1) N!/(N+1)!;
                                          N!
(D1)                                   --------
                                       (N + 1)!
(C2) MINFACTORIAL(\%);
                                          1
(D2)                                    -----
                                        N + 1
```

`MAKEGAMMA` (*exp*) [*Function*]

transforms occurrences of binomial, factorial, and beta functions in *exp* to GAMMA functions.

5.1.5 Combining Sums of Quotients

`XTHRU` (*exp*) [*Function*]

combines all terms of *exp* (which should be a sum) over a common denominator, without expanding products and exponentiated sums as RATSIMP does. XTHRU cancels common factors in the numerator and denominator of rational expressions, but only if the factors are explicit. Sometimes it is better to use XTHRU before RATSIMPing an expression in order to cause explicit factors of the gcd of the numerator and denominator to be canceled, thus simplifying the expression to be RATSIMPed.

```
(C1) ((X+2)**20-2*Y)/(X+Y)**20+(X+Y)**-19-X/(X+Y)**20;
                                                    20
                      1           X        (X + 2)   - 2 Y
(D1)             --------- - --------- + ---------------
                        19           20              20
                  (Y + X)     (Y + X)         (Y + X)
(C2) XTHRU(\%);
                                    20
                               (X + 2)   - Y
(D2)                           -------------
                                       20
                                 (Y + X)
```

`COMBINE` (*exp*) [*Function*]

simplifies the sum *exp* by combining terms with the same denominator into a single term.

`RNCOMBINE` (*exp*) [*Function*]

transforms *exp* by combining all terms of *exp* that have identical denominators or denominators that differ from each other by numerical factors only. This is slightly different from the behavior of COMBINE, which collects terms that have identical denominators. Setting PFEFORMAT:TRUE and using COMBINE will achieve results similar to those that can be obtained with RNCOMBINE, but RNCOMBINE takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possibility of recognizing some cancellations. There is a demo in the file: '`share1/rncomb.dem`'. × There are some usage notes in the file: '`share1/rncomb.usg`'.

`SUMCONTRACT` (*exp*) [*Function*]

will combine all sums of an addition that have upper and lower bounds that differ by constants. The result will be an expression containing one summation for each set of such summations added to all appropriate extra terms that had to be extracted to form this sum. SUMCONTRACT will combine all compatible sums and use one of the indices from one of the sums if it can, and then try to form a reasonable index if it cannot use any supplied. It may be necessary to apply INTOSUM before the SUMCONTRACT.

`INTOSUM` (*exp*) [*Function*]

will take all things that a summation is multiplied by, and put them inside the summation. If the index is used in the outside expression, then the function tries to find a reasonable index, the same as it does for SUMCONTRACT. This is essentially the reverse idea of the OUTATIVE property of summations, but note that it does not remove this property, it only bypasses it. In some cases, a SCANMAP(MULTTHRU, exp) may be necessary before applying the INTOSUM.

5.2 Expanding Expressions

`EXPAND` (*exp*) [*Function*]

will cause products of sums and exponentiated sums to be multiplied out, numerators of rational expressions which are sums to be split into their respective terms, and multiplication (commutative and non-commutative) to be distributed over addition at all levels of *exp*. For polynomials, one should usually use RATEXPAND which uses a more efficient algorithm. MAXNEGEX default: [1000] and MAXPOSEX default: [1000] control the maximum negative and positive exponents, respectively, which will expand.

EXPAND*(exp, p, n)* expands *exp*, using *p* for MAXPOSEX and *n* for MAXNEGEX. This is useful in order to expand part, but not all, of an expression.

The EXPAND flag used with EV (see EV) also causes expansion; see section 4.1 [Evaluation], page 47.

EXPANDWRT (*exp, var1, var2, ...*) [*Function*]

expands *exp* with respect to the *vari*. All products involving the *vari* appear explicitly. The form returned will be free of products of sums of expressions that are not free of the *vari*. The *vari* may be variables, operators, or expressions. By default, denominators are not expanded, but this can be controlled by means of the switch EXPANDWRT_DENOM. Do LOAD(STOPEX); to use this function.

EXPANDWRT_FACTORED (*exp, var1, var2, ...*) [*Function*]

is similar to EXPANDWRT, but treats expressions that are products somewhat differently. EXPANDWRT_FACTORED will perform the required expansion only on those factors of *exp* that contain the variables in the argument list of EXPANDWRT_FACTORED. Do LOAD(STOPEX); to use this function.

5.2.1 Expand Flags

See section 5.2.5.1 [Trig Expand Flags], page 80.

MAXNEGEX [*variable, default: 1000*]

the largest negative exponent which will be expanded by the EXPAND command.

MAXPOSEX [*variable, default: 1000*]

the largest exponent which will be expanded with the EXPAND command.

EXPANDWRT_DENOM [*variable, default: FALSE*]

controls the treatment of rational expressions by EXPANDWRT. If TRUE, then both the numerator and denominator of the expression will be expanded according to the arguments of EXPANDWRT, but if EXPANDWRT_DENOM is FALSE, then only the numerator will be expanded in that way. Do LOAD(STOPEX); to use.

5.2.2 Expanding CRE Expressions

RATEXPAND (*exp*) [*Function*]

expands *exp* by multiplying out products of sums and exponentiated sums, combining fractions over a common denominator, cancelling the greatest common divisor of the numerator and denominator, then splitting the numerator (if a sum) into its respective terms divided by the denominator. This is accomplished by converting *exp* to CRE form and then back to general form. The switch RATEXPAND, default: [FALSE], if TRUE will cause CRE expressions to be fully expanded when they are converted back to general form or displayed, while if it is FALSE then they will be put into a recursive form (see RATSIMP).

RATDENOMDIVIDE [*variable, default: TRUE*]

if FALSE will stop the splitting up of the terms of the numerator of RATEXPANDed expressions from occurring.

KEEPFLOAT default: [FALSE] if set to TRUE will prevent floating point numbers from being rationalized when expressions which contain them are converted to CRE form.

```
(C1) RATEXPAND((2*X-3*Y)**3);
                3         2       2         3
(D1)      - 27 Y  + 54 X Y  - 36 X  Y + 8 X
(C2) (X-1)/(X+1)**2+1/(X-1);
                             X - 1       1
(D2)                        -------- + -----
                                    2   X - 1
                           (X + 1)
(C3) EXPAND(D2);
                         X              1           1
(D3)                -------------- - ------------ + -----
                     2              2             X - 1
                    X  + 2 X + 1   X  + 2 X + 1
(C4) RATEXPAND(D2);
                            2
                         2 X                 2
(D4)                  ---------------- + ----------------
                       3    2             3    2
                      X  + X  - X - 1    X  + X  - X - 1
```

PSEXPAND [*variable, default: FALSE*]

if TRUE will cause extended rational function expressions to display fully expanded. (RATEXPAND will also cause this.) If FALSE, multivariate expressions will be displayed just as in the rational function package. If PSEXPAND:MULTI, then terms with the same total degree in the variables are grouped together.

5.2.3 Partial Expansion

DISTRIB (*exp*) [*Function*]

distributes sums over products. It differs from EXPAND in that it works at only the top level of an expression; it doesn't recurse and it is faster than EXPAND. It differs from MULTTHRU in that it expands all sums at that level. For example,

```
DISTRIB((A+B)*(C+D))            -> A C + A D + B C + B D
MULTTHRU ((A+B)*(C+D))          -> (A + B) C + (A + B) D
DISTRIB (1/((A+B)*(C+D)))       -> 1/ ((A+B) *(C+D))
EXPAND(1/((A+B)*(C+D)),1,0)     -> 1/ (A C + A D + B C + B D)
```

MULTTHRU (*exp*) [*Function*]

multiplies a factor (which should be a sum) of *exp* by the other factors of *exp*. That is *exp* is *f1*f2* ... *fn* where at least one factor, say *fi*, is a sum of terms. Each term in that sum is multiplied by the other factors in the product. (Namely all the factors except *fi*). MULTTHRU does not expand exponentiated sums. This function is the fastest way to distribute products (commutative or noncommutative) over sums. Since quotients are represented as products, MULTTHRU can be used to divide sums by products as well.

MULTTHRU(exp1, exp2) multiplies each term in *exp2* (which should be a sum or an equation) by *exp1*. If *exp1* is not itself a sum then this form is equivalent to MULTTHRU(exp1*exp2).

```
(C1) X/(X-Y)**2-1/(X-Y)-F(X)/(X-Y)**3;
                1        X         F(X)
(D1)       - ----- + -------- - --------
               X - Y          2          3
                        (X - Y)    (X - Y)
(C2) MULTTHRU((X-Y)**3,\%);
                     2
(D2)       - (X - Y)  + X (X - Y) - F(X)
(C3) RATEXPAND(D2);
                           2
(D3)                     - Y  + X Y - F(X)
(C4) ((A+B)**10*S**2+2*A*B*S+(A*B)**2)/(A*B*S**2);
                          10  2             2  2
                 (B  + A )   S  + 2 A B S + A  B
(D4)             -------------------------------
                                 2
                              A B S
(C5) MULTTHRU(\%);
                                              10
                         2   A B   (B  + A)
(D5)                     - + --- + -------
                         S    2      A B
                              S
(notice that (B+A)**10 is not expanded)
(C6) MULTTHRU(A.(B+C.(D+E)+F));
(D6)                A . F + A . (C . (E + D)) + A . B
(compare with similar example under \fn{EXPAND})
```

5.2.4 Partial Fractions

PARTFRAC (*exp*, *var*) [*Function*]

expands the expression *exp* in partial fractions with respect to the main variable, *var*. PARTFRAC does a complete partial fraction decomposition. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out.

5.2.5 Trigonometric Expansions

`TRIGEXPAND` (*exp*) [*Function*]

expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in *exp*. For best results, *exp* should be expanded. To enhance user control of simplification, this function expands only one level at a time, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the switch TRIGEXPAND:TRUE;.

5.2.5.1 Trig Expand Flags

`TRIGEXPAND` [*variable, default: FALSE*]

if TRUE causes expansion of all expressions containing SINs and COSs occurring subsequently.

```
(C1) X+SIN(3*X)/SIN(X),TRIGEXPAND=TRUE,EXPAND;
                                2           2
(D1)                      - SIN (X) + 3 COS (X) + X
(C2) TRIGEXPAND(SIN(10*X+Y));
(D2)               COS(10 X) SIN(Y) + SIN(10 X) COS(Y)
```

`TRIGEXPANDPLUS` [*variable, default: TRUE*]

controls the sum rule for TRIGEXPAND. Thus, when the TRIGEXPAND command is used or the TRIGEXPAND switch set to TRUE, expansion of sums (e.g. SIN(X+Y)) will take place only if TRIGEXPANDPLUS is TRUE.

`TRIGEXPANDTIMES` [*variable, default: TRUE*]

controls the product rule for TRIGEXPAND. Thus, when the TRIGEXPAND command is used or the TRIGEXPAND switch set to TRUE, expansion of products (e.g. SIN(2*X)) will take place only if TRIGEXPANDTIMES is TRUE.

See section 2.4.4 [Display of Trig Functions], page 26 for the definition of TRIGSIGN, TRIGINVERSES, HALFANGLES and EXPONENTIALIZE.

5.2.6 Controlled Expansions

The file '`share1/facexp.mc`' contains several related functions that provide the user with the ability to structure expressions by controlled expansion. Do LOAD(FACEXP); to use this function. This capability is especially useful when the expression contains variables that have physical meaning, because it is often true that the most economical form of such an expression can be obtained by fully expanding the expression with respect to those variables, and then factoring their coefficients. While it is true that this procedure is not difficult to carry out using standard **Maxima** functions, additional fine-tuning may also be desirable, and these finishing touches can be more difficult to apply. The function FACSUM and its related forms provide

a convenient means for controlling the structure of expressions in this way. Another function, COLLECTTERMS, can be used to add two or more expressions that have already been simplified to this form, without resimplifying the whole expression again. This function is particularly useful when the expressions are large and address space or cpu time is in short supply.

FACSUM (*exp, arg1, arg2, ...*) [*Function*]

returns a form of *exp* which depends on the *argi*. The *argi* can be any form suitable for RATVARS, or they can be lists of such forms. If the *argi* are not lists, then the form returned will be fully expanded with respect to the *argi*, and the coefficients of the *argi* will be factored. These coefficients will be free of the *argi*, except perhaps in a non-rational sense. If any of the *argi* are lists, then all such lists will be combined into a single list, and instead of calling FACTOR on the coefficients of the *argi*, FACSUM will call itself on these coefficients, using this newly constructed single list as the new *argi* for this recursive call. This process can be repeated to arbitrary depth by nesting the desired elements in lists.

It is possible that one may wish to FACSUM with respect to more complicated subexpressions, such as LOG(X+Y). Such arguments are also permissible. With no variable specification, for example FACSUM(exp), the result returned is the same as that returned by RATSIMP(exp).

Occasionally the user may wish to obtain any of the above forms for expressions which are specified only by their leading operators. For example, one may wish to FACSUM with respect to all LOG's. In this situation, one may include among the *argi* either the specific LOG's which are to be treated in this way, or alternatively, either the expression OPERATOR(LOG) or 'OPERATOR(LOG). If one wished to FACSUM the expression exp with respect to the operators OP1, OP2, ..., OPn, one would evaluate FACSUM(EXP, OPERATOR(OP1, OP2, ..., OPn)). The OPERATOR form may also appear inside list arguments.

In addition, the setting of the switches FACSUM_COMBINE and NEXTLAYERFACTOR may affect the result of FACSUM as follows:

NEXTLAYERFACTOR [*variable, default: FALSE*]

if TRUE will force the recursive calls of FACSUM to be applied to the factors of the factored form of the coefficients of the *argi*. If FALSE then FACSUM will be applied to each coefficient as a whole whenever recursive calls to FACSUM occur as described above. In addition, inclusion of the atom NEXTLAYERFACTOR in the argument list of FACSUM has the effect of NEXTLAYERFACTOR:TRUE, but for the next level of the expression *only*. Since NEXTLAYERFACTOR is always bound to either TRUE or FALSE, it must be presented single-quoted whenever it is used in this way.

FACSUM_COMBINE [*variable, default: TRUE*]

controls the form of the final result returned by FACSUM when its argument is a quotient of polynomials. If FACSUM_COMBINE is FALSE then the form will be returned as a fully expanded sum as described above, but if TRUE, then the formed returned is a ratio of polynomials, with each polynomial in the form described above. The TRUE setting of this switch is useful when one wants to FACSUM both the numerator and denominator of a rational expression, but does not want the denominator to be multiplied through the terms of the numerator.

FACTORFACSUM (*exp, arg1, arg2, . . . , argN*) [*Function*]

returns a form of *exp* which is obtained by calling FACSUM on the factors of *exp* with the *argi* as arguments. If any of the factors of *exp* is raised to a power, both the factor and the exponent will be processed in this way.

COLLECTTERMS (*arg1, arg2, . . . argn*) [*Function*]

If several expressions have been simplified with FACSUM, FACTORFACSUM, FACTENEXPAND, FACEXPTEN or FACTORFACEXPTEN, and they are to be added together, it may be desirable to combine them using the function COLLECTTERMS. COLLECTTERMS can take as arguments all of the arguments that can be given to these other associated functions with the exception of NEXTLAYERFACTOR, which has no effect on COLLECTTERMS. The advantage of COLLECTTERMS is that it returns a form similar to FACSUM, but since it is adding forms that have already been processed by FACSUM, it does not need to repeat that effort. This capability is especially useful when the expressions to be summed are very large. There is a demo in the file: '`share1/facexp.dem`'. There are some usage notes in the file: '`share1/facexp.usg`'.

5.3 Factoring Expressions

FACTOR (*exp*) [*Function*]

factors the expression *exp*, containing any number of variables or functions, into factors irreducible over the integers.

FACTOR*(exp, p)* factors *exp* over the field of integers with an element adjoined whose minimum polynomial is *p*. There is a demo in the file: '`demo/factor.dem`'.

GFACTOR (*exp*) [*Function*]

factors the expression *exp*, containing any number of variables or functions, into factors irreducible over the Gaussian integers.

FACTOROUT (*exp, var1, var2, . . .*) [*Function*]

rearranges the sum *exp* into a sum of terms of the form *f(var1,var2,. . .)*g* where *g* is a product of expressions not containing the *vari*'s and *f* is factored.

FACTORSUM (*exp*) [*Function*]

tries to group terms in factors of *exp* which are sums into groups of terms such that their sum is factorable. It can recover the result of EXPAND((X+Y)**2+(Z+W)**2) but it can't recover × EXPAND((X+1)**2+(X+Y)**2) because the terms have variables in common.

```
(C1) (X+1)*((U+V)^2+A*(W+Z)^2),EXPAND;
      2      2                                2      2
(D1) A X Z  + A Z  + 2 A W X Z + 2 A W Z + A W  X + V  X
                     2        2    2           2
       + 2 U V X + U  X + A W  + V  + 2 U V + U
(C2) FACTORSUM(\%);
                                  2          2
(D2)                (X + 1) (A (Z + W)  + (V + U) )
```

GFACTORSUM (*exp*) [*Function*]

the same as FACTORSUM but uses GFACTOR.

5.3.1 Factor Flags

FACTORFLAG [*variable, default: FALSE*]

if FALSE suppresses the factoring of integer factors of rational expressions.

DONTFACTOR [*Variable*]

may be set to a list of variables with respect to which factoring is not to occur. (It is initially empty). Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the DONTFACTOR list.

SAVEFACTORS [*variable, default: FALSE*]

if TRUE causes the factors of an expression which is a product of factors to be saved by certain functions, in order to speed up later factorizations of expressions containing some of the same factors.

INTFACLIM [*variable, default: 1000*]

is the largest divisor which will be tried when factoring a bignum integer. If set to FALSE (this is the case when the user calls FACTOR explicitly), or if the integer is a fixnum (i.e. fits in one machine word), complete factorization of the integer will be attempted. The user's setting of INTFACLIM is used for internal calls to FACTOR. Thus, INTFACLIM may be reset to prevent **Maxima** from taking an inordinately long time factoring large integers.

BERLEFACT [*variable, default: TRUE*]

if FALSE then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used.

NEWFAC [*variable, default: FALSE*]

may be set to TRUE to use the new factoring routines.

5.4 Manipulating Polynomials

FASTTIMES (*poly1*, *poly2*) [*Function*]

multiplies the polynomials *poly1* and *poly2* by using a special algorithm for multiplication of polynomials. They should be multivariate, dense, and nearly the same size. Classical multiplication is of order N*M where N and M are the degrees. FASTTIMES is of order MAX(N,M)**1.585.

DIVIDE (*poly1*, *poly2*, *var1*, ..., *varn*) [*Function*]

computes the quotient and remainder of the polynomial *poly1* divided by the polynomial *poly2*, in a main polynomial variable, *varn*. The other variables are as in the RATVARS function. The result is a list whose first element is the quotient and whose second element is the remainder.

```
(C1) DIVIDE(X+Y,X-Y,X);
(D1)                          [1, 2 Y]
(C2) DIVIDE(X+Y,X-Y);
(D2)                        [ - 1, 2 X]
```

Note that Y is the main variable in (C2).

QUOTIENT (*poly1*, *poly2*, *var1*, ...) [*Function*]

computes the quotient of the polynomial *poly1* divided by the polynomial *poly2*.

REMAINDER (*poly1*, *poly2*, *var1*, ...) [*Function*]

computes the remainder of the polynomial *poly1* divided by the polynomial *poly2*.

RESULTANT (*poly1*, *poly2*, *var*) [*Function*]

computes the resultant of the two polynomials *poly1* and *poly2*, eliminating the variable *var*. The resultant is a determinant of the coefficients of *var* in *poly1* and *poly2* which equals zero if and only if *poly1* and *poly2* have a non-constant factor in common. If *poly1* or *poly2* can be factored, it may be desirable to call FACTOR before calling RESULTANT.

RESULTANT [*variable, default: SUBRES*]

controls which algorithm will be used to compute the resultant. SUBRES for subresultant prs [the default], MOD for modular resultant algorithm, and RED for reduced prs. On most problems SUBRES should be best. On some large degree univariate or bivariate problems MOD may be better. Another alternative is the BEZOUT command which takes the same arguments as RESULTANT and returns a matrix. DETERMINANT of this matrix is the desired resultant.

MOD (*poly*) [*Function*]

converts the polynomial poly to a modular representation with respect to the current modulus which is the value of the variable MODULUS.

MOD*(poly, m)* specifies a MODULUS *m* to be used for converting *poly*, if it is desired to override the current global value of MODULUS.

POLYDECOMP (*poly, var*) [*Function*]

returns a list of polynomials [f1(var), f2(var), . . . , fn(var)] such that poly = f1(f2(. . . fn(var) . . .)). There is no other decomposition which involves more polynomials excepting linear *fi*.

HORNER (*exp, var*) [*Function*]

will convert *exp* into a rearranged representation as in Horner's rule, using *var* as the main variable if it is specified. *Var* may also be omitted in which case the main variable of the CRE form of *exp* is used. HORNER sometimes improves stability if *exp* is to be numerically evaluated. It is also useful if **Maxima** is used to generate programs to be run in FORTRAN (see also STRINGOUT).

```
(C1) 1.0E-20*X^2-5.5*X+5.2E20;
                                  2
(D1)                    1.0E-20 X  - 5.5 X + 5.2E+20
(C2) HORNER(\%,X),KEEPFLOAT:TRUE;
(D2)                   X (1.0E-20 X - 5.5) + 5.2E+20
(C3) D1,X=1.0E20;
ARITHMETIC OVERFLOW
(C4) D2,X=1.0E20;
(D4)                            6.9999999E+19
```

5.4.1 Greatest Common Divisors

GCD (*poly1, poly2, var1, . . .*) [*Function*]

computes the greatest common divisor of *poly1* and *poly2*.

GCD [*variable, default: SPMOD*]

determines which algorithm is employed. Setting GCD to EZ, EEZ, SUBRES, RED, or SPMOD selects the EZGCD, new EEZ GCD, subresultant PRS, reduced, or modular algorithm, respectively. If GCD:FALSE then GCD(poly1, poly2, var) will always return 1 for all *var*. Many functions (e.g. RATSIMP, FACTOR, etc.) cause gcd's to be taken implicitly. For homogeneous polynomials it is recommended that GCD:SUBRES be used. To take the gcd when an algebraic is present, e.g. GCD(X**2-2*SQRT(2)*X+2,X-SQRT(2));, ALGEBRAIC must be TRUE and GCD must not be EZ. SUBRES is a new algorithm, and people who have been using the RED setting should probably change it to SUBRES. The GCD flag, default: [SPMOD], if FALSE will also prevent the greatest common divisor from being taken when expressions are converted to CRE form. This will sometimes speed the calculation if gcds are not required.

EZGCD (*poly1, poly2, ...*) [*Function*]

gives a list whose first element is the g.c.d of the polynomials *poly1, poly2, ...* and whose remaining elements are the polynomials divided by the g.c.d. This always uses the EZGCD algorithm. There is a demo in the file: '`demo/ezgcd.dem`'.

LCM (*exp1, exp2, ...*) [*Function*]

returns the Least Common Multiple of its arguments. Do LOAD(FUNCTS); to access this function.

CONTENT (*poly1, var1, ..., varn*) [*Function*]

returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial *poly1* in the variable *varn* (this is the content) and whose second element is the polynomial *poly1* divided by the content.

```
(C1) CONTENT(2*X*Y+4*X**2*Y**2,Y);
(D1)            [2*X, 2*X*Y**2+Y].
```

CHAPTER

SIX

Linear Algebra

6.1 Arrays

6.1.1 Defining Arrays

ARRAY (*array, dim1, dim2, . . . , dimk*) [*Special Form*]

This sets up a *k*-dimensional array. A maximum of five dimensions may be used. The subscripts for the *i*'th dimension are the integers running from 0 to *dimi*. If the user assigns to a subscripted variable before declaring the corresponding array, an undeclared array is set up. If the user has more than one array to be set up the same way, they may all be set up at the same time, by ARRAY([list-of-names],dim1, dim2, . . . , dimk).

Undeclared arrays, otherwise known as (because hash coding is done on the subscripts), are more general than declared arrays. The user does not declare their maximum size, and they grow dynamically by hashing as more elements are assigned values. The subscripts of undeclared arrays need not even be numbers. However, unless an array is rather sparse, it is probably more efficient to declare it when possible, rather than to leave it undeclared. The ARRAY function can be also used to transform an undeclared array into a declared array. There is a demo in the file: '`demo/array.dem`'.

ARRAYS [*Variable*]

a list of all the arrays that have been allocated, both declared and undeclared. Functions which deal with arrays are: ARRAY, ARRAYAPPLY, ARRAYINFO, ARRAYMAKE, FILLARRAY, LISTARRAY, and REMARRAY.

ARRAYMAKE (*array,[i1, i2, . . .]*) [*Function*]

returns the array array[i1, i2, . . .].

6.1.2 Manipulating Arrays

LISTARRAY (*array*) [*Function*]

returns a list of the elements of a declared or hashed array. The order is row-major. Elements which you have not defined yet will be represented by #####.

ARRAYAPPLY (*array, [sub1, . . . , subk]*) [*Function*]

is like APPLY except the first argument is an array.

FILLARRAY (*array, list-or-array*) [*Function*]

fills *array* from *list-or-array*. If *array* is a floating-point (integer) array then *list-or-array* should be either a list of floating-point (integer) numbers or another floating-point (integer) array. If the dimensions of the arrays are different, *array* is filled in row-major order. If there are not enough elements in *list-or-array* the last element is used to fill out the rest of array. If there are too many elements, the remaining ones are thrown away. FILLARRAY returns its first argument.

ARRAYINFO (*array*) [*Special Form*]

returns a list of information about the array *array*. For hashed arrays it returns a list of HASHED, the number of subscripts, and the subscripts of every element which has a value. For declared arrays it returns a list of DECLARED, the number of subscripts, and the bounds that were given the the ARRAY function when it was called on *a*.

REMARRAY (*name1, name2, . . .*) [*Special Form*]

removes arrays and array associated functions and frees the storage occupied. If name is ALL then all arrays are removed. It may be necessary to use this function if it is desired to redefine the values in a hashed array.

6.2 Basic Matrix Operations

Matrix multiplication is effected by using the dot operator, ., which is also convenient if the user wishes to represent other non-commutative algebraic operations. The exponential of the . operation is ^^. Thus, for a matrix A, A.A = A^^2 and, if it exists, A^^-1 is the inverse of A.

NCEXPT (*A, B*) [*Function*]

if an (non-commutative) exponential expression is too wide to be displayed as A^^B it will appear as NCEXPT(A,B).

The operations +, -, *, ** are all element-by-element operations; all operations are normally carried out in full, including the . (dot) operation. Many switches exist for controlling simplification rules involving dot and matrix-list operations. See section 6.3 [Defining Matrices], page 91.

Options Relating to Matrices: LMXCHAR, RMXCHAR, RATMX, LISTARITH, DETOUT, DOALLMX-OPS, DOMXEXPT, DOMXMXOPS, DOSCMXOPS, DOSCMXPLUS, SCALARMATRIX, and SPARSE. There is a demo in the file: ‘`demo/matrix.dem`’.

6.2.1 Matrices Flags

RATMX [*variable, default: FALSE*]

if FALSE will cause determinants and matrix addition, subtraction, and multiplication to be performed in the representation of the matrix elements and will cause the result of matrix inversion to be left in general representation. If it is TRUE, the 4 operations mentioned above will be performed in CRE form and the result of matrix inverse will be in CRE form. Note that this may cause the elements to be expanded (depending on the setting of RATFAC) which might not always be desired.

LMXCHAR [*variable*]

The character used to display the left delimiter of a matrix (see also RMXCHAR).

RMXCHAR [*variable*]

The character used to display the right delimiter of a matrix (see also LMXCHAR).

MATRIX_ELEMENT_ADD [*variable, default: +*]

May be set to ?; may also be the name of a function, or a LAMBDA expression. In this way, a rich variety of algebraic structures may be simulated. There is a demo in the file: '`demo/matrix.dm1`'. There is a demo in the file: '`demo/matrix.dm2`'.

MATRIX_ELEMENT_MULT [*variable, default: **]

May be set to `.`; may also be the name of a function, or a LAMBDA expression. In this way, a rich variety of algebraic structures may be simulated. There is a demo in the file: '`demo/matrix.dm1`'. There is a demo in the file: '`demo/matrix.dm2`'.

MATRIX_ELEMENT_TRANSPOSE [*variable, default: FALSE*]

Other useful settings are TRANSPOSE and NONSCALARS; may also be the name of a function, or a LAMBDA expression. In this way, a rich variety of algebraic structures may be simulated. There is a demo in the file: '`demo/matrix.dm1`'. There is a demo in the file: '`demo/matrix.dm2`'.

DOALLMXOPS [*variable, default: TRUE*]

if TRUE all operations relating to matrices are carried out. If it is FALSE then the setting of the individual DOT switches govern which operations are performed.

DOMXEXPT [*variable, default: TRUE*]

if TRUE, %E**MATRIX([1,2],[3,4]); → MATRIX([%E,%E**2],[%E**3,%E**4]). In general, this transformation affects expressions of the form <base>**<power> where <base> is an expression assumed scalar or constant, and <power> is a list or matrix. This transformation is turned off if this switch is set to FALSE.

DOMXMXOPS [*variable, default: TRUE*]

if TRUE then all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this switch is FALSE they are not.

DOMXNCTIMES [*variable, default: FALSE*]

Causes non-commutative products of matrices to be carried out.

DOSCMXOPS [*variable, default: FALSE*]

if TRUE then scalar-matrix operations are performed.

DOSCMXPLUS [*variable, default: FALSE*]

if TRUE will cause SCALAR + MATRIX to give a matrix answer. This switch is not subsumed under DOALLMXOPS.

6.2.2 Non-Commutative Operations

Syntax . [*Special Form*]

The dot operator, for matrix (non-commutative) multiplication. When . is used in this way, spaces should be left on both sides of it, e.g. A . B. This distinguishes it plainly from a decimal point in a floating point number.

6.2.2.1 Flags for Non-Commutative Operations

DOT0SIMP [*variable, default: TRUE*]

Causes a non-commutative product of zero and a scalar term to be simplified to a commutative product.

DOT0NSCSIMP [*variable, default: TRUE*]

Causes a non-commutative product of zero and a nonscalar term to be simplified to a commutative product.

DOT1SIMP [*variable, default: TRUE*]

Causes a non-commutative product of one and another term to be simplified to a commutative product.

DOTASSOC [*variable, default: TRUE*]

when TRUE causes (A.B).C to simplify to A.(B.C)

DOTCONSTRULES [*variable, default: TRUE*]

Causes a non-commutative product of a constant and another term to be simplified to a commutative product. Turning on this flag effectively turns on DOT0SIMP, DOT0NSCSIMP, and DOT1SIMP as well.

DOTDISTRIB [*variable, default: FALSE*]

if TRUE will cause A.(B+C) to simplify to A.B+A.C

DOTEXPTSIMP [*variable, default: TRUE*]

when TRUE causes A.A to simplify to A^^2

DOTIDENT [*variable, default: 1*]

The value to be returned by X^^0.

DOTSCRULES [*variable, default: FALSE*]

when TRUE will cause A.SC or SC.A to simplify to SC*A, and A.(SC*B) to simplify to SC*(A.B)

6.3 Defining Matrices

MATRIX (*row1*, . . . , *rown*) [*Function*]

defines a rectangular matrix with the indicated rows. Each row has the form of a list of expressions, e.g. [A, X**2, Y, 0] is a list of 4 elements.

GENMATRIX (*array*, *i2*, *j2*, *i1*, *j1*) [*Function*]

generates a matrix from the array *array* using *array(i1,j1)* for the first (upper-left) element and *array(i2,j2)* for the last (lower-right) element of the matrix. If *j1=i1* then *j1* may be omitted. If *j1=i1=1* then *i1* and *j1* may both be omitted. If a selected element of the array doesn't exist a symbolic one will be used.

```
(C1) H[I,J]:=1/(I+J-1)$
(C2) GENMATRIX(H,3,3);
                              [   1  1]
                              [1  -  -]
                              [   2  3]
                              [       ]
                              [1  1  1]
(D2)                          [-  -  -]
                              [2  3  4]
                              [       ]
                              [1  1  1]
                              [-  -  -]
                              [3  4  5]
```

ENTERMATRIX (*m*, *n*) [*Function*]

allows one to enter a matrix element by element with **Maxima** requesting values for each of the *m***n* entries.

```
(C1) ENTERMATRIX(3,3);
Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4
1;
Row 1 Column 1:  A;
Row 2 Column 2:  B;
Row 3 Column 3:  C;
Matrix entered.
                                [ A  0  0 ]
                                [         ]
(D1)                            [ 0  B  0 ]
                                [         ]
                                [ 0  0  C ]
```

COPYMATRIX (*mat*) [*Function*]

creates a copy of the matrix *mat*. This is the only way to make a copy aside from recreating *mat* elementwise. Copying a matrix may be useful when SETELMX is used.

6.3.1 Defining Special Matrices

DIAGMATRIX (*n*, *exp*) [*Function*]

returns a diagonal matrix of size *n* by *n* with the diagonal elements all *x*. An identity matrix is created by DIAGMATRIX(n,1), or one may use IDENT(n).

EMATRIX (*m*, *n*, *exp*, *i*, *j*) [*Function*]

will create an m by *n* matrix all of whose elements are zero except for the *i*, *j* element which is *exp*.

ZEROMATRIX (*m*, *n*) [*Function*]

takes integers *m*,*n* as arguments and returns an *m* by *n* matrix of 0's.

6.4 Matrix Information

RANK (*mat*) [*Function*]

computes the rank of the matrix *mat*. That is, the order of the largest non-singular subdeterminant of *mat*. Caveat: RANK may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.

MATTRACE (*mat*) [*Function*]

computes the trace [sum of the elements on the main diagonal] of the square matrix *mat*. It is used by NCHARPOLY, an alternative to CHARPOLY. It is used by doing LOAD(NCHRPL);. There is a demo in the file: 'share1/nchrpl.dem'.

DETERMINANT (*mat*) [*Function*]

computes the determinant of *mat* by a method similar to Gaussian elimination. The form of the result depends upon the setting of the switch RATMX. There is a special routine for dealing with sparse determinants which can be used by setting the switches RATMX and SPARSE to be TRUE.

DETOUT [*variable, default: FALSE*]

if TRUE will cause the determinant of a matrix whose inverse is computed to be kept outside of the inverse. For this switch to have an effect DOALLMXOPS and DOSCMXOPS should be FALSE (see their descriptions). Alternatively this switch can be given to EV which causes the other two to be set correctly.

SPARSE [*variable, default: FALSE*]

if TRUE and if RATMX:TRUE, then DETERMINANT will use special routines for computing sparse determinants.

NEWDET (*mat, n*) [*Function*]

also computes the determinant of *mat* but uses the Johnson-Gentleman tree minor algorithm. *mat* may be the name of a matrix or array. The argument *n* is the order; it is optional if *mat* is a matrix.

PERMANENT (*mat, int*) [*Function*]

computes the permanent of the matrix *mat*. A permanent is like a determinant but with no sign changes.

6.5 Manipulating Matrices

SETELMX (*exp, i, j, mat*) [*Function*]

changes the *i, j* element of *mat* to *exp*. The altered matrix is returned as the value. MAT[I,J]:EXP may also be used, altering *mat* in a similar manner, but returning EXP as the value.

ROW (*mat, i*) [*Function*]

gives a matrix of the *i*'th row of matrix *mat*.

COL (*mat, i*) [*Function*]

gives a matrix of the *i*'th column of the matrix *mat*.

ADDROW (*mat, list1, list2, ...*) [*Function*]

appends the row(s) given by the one or more lists (or matrices) onto the matrix *mat*.

ADDCOL (*mat, list1, list2, ..., listn*) [*Function*]

appends the column(s) given by the one or more lists (or matrices) onto the matrix *mat*.

SUBMATRIX (*m1, ..., mat, n1, ...*) [*Function*]

creates a new matrix composed of the matrix *mat* with rows *mi* deleted, and columns *ni* deleted.

6.6 Operating on Matrices

ADJOINT (*matrix*) [*Function*]

computes the adjoint of a matrix using the adjoint method. Do LOAD("INVERT"); to use this function. This allows a user to compute the inverse of a matrix with bfloat entries or polynomials with floating point coefficients without converting to CRE form. The DETERMINANT command is used to compute cofactors, so if RATMX is FALSE (the default) the inverse is computed without changing the representation of the elements. The functions ADJOINT and INVERT are provided. The current implementation is inefficient for matrices of high order. The DETOUT flag if true keeps the determinant factored out of the inverse.

AUGCOEFMATRIX (*[eq1, ...], [var1, ...]*) [*Function*]

the augmented coefficient matrix for the variables *var1,...* of the system of linear equations *eq1,...*. This is the coefficient matrix with a column adjoined for the constant terms in each equation (i.e. those not dependent upon *var1,...*).

COEFMATRIX (*[eq1, ...], [var1, ...]*) [*Function*]

the coefficient matrix for the variables *var1,...* of the system of linear equations *eq1,...*.

ECHELON (*mat*) [*Function*]

produces the echelon form of the matrix *mat*. That is, *mat* with elementary row operations performed on it such that the first non-zero element in each row in the resulting matrix is a one and the column elements under the first one in each row are all zero.

```
                          [2  1 - A  -5 B ]
(D2)                      [                ]
                          [A    B      C   ]
(C3) ECHELON(D2);
                  [      A - 1        5 B      ]
                  [1   - -----      - ---      ]
                  [        2           2       ]
(D3)              [                            ]
                  [                2 C + 5 A B ]
                  [0     1         ------------]
                  [                       2    ]
                  [                2 B + A  - A]
```

`INVERT` (*matrix*) [*Function*]

finds the inverse of a matrix using the adjoint method. This allows a user to compute the inverse of a matrix with bigfloat entries or polynomials with floating point coefficients without converting to CRE form. The DETERMINANT command is used to compute cofactors, so if RATMX is FALSE (the default) the inverse is computed without changing the representation of the elements. The current implementation is inefficient for matrices of high order. The DETOUT flag if TRUE keeps the determinant factored out of the inverse.

Note: the results are not automatically expanded. If the matrix originally had polynomial entries, better appearing output can be generated by EXPAND(INVERT(mat)),DETOUT. If it is desirable to then divide through by the determinant this can be accomplished by XTHRU(%) or alternatively from scratch by

```
EXPAND(ADJOINT(mat))/EXPAND(DETERMINANT(mat)).
```

See section 1.3 [Reader Syntax], page 3, for the definiition of ^^ for another method of inverting a matrix.

There are some usage notes in the file: ‘`share1/invert.usg`’.

`TRANSPOSE` (*mat*) [*Function*]

produces the transpose of the matrix *mat*.

`TRIANGULARIZE` (*mat*) [*Function*]

produces the upper triangular form of the matrix *mat*, which needn’t be square.

`MATRIXMAP` (*fun, mat*) [*Special Form*]

will map the function *fun* onto each element of the matrix *mat*.

6.6.1 Characteristic Polynomials

`CHARPOLY` (*mat, var*) [*Function*]

computes the characteristic polynomial for matrix *mat* with respect to *var*. That is, DETERMINANT(M - DIAGMATRIX(LENGTH(mat),var)).

`NCHARPOLY` (*mat, var*) [*Function*]

finds the characteristic polynomial of the matrix *mat* with respect to *var*. This is an alternative to **Maxima**'s CHARPOLY. **NCHARPOLY** works by computing traces of powers of the given matrix, which are known to be equal to sums of powers of the roots of the characteristic polynomial. From these quantities the symmetric functions of the roots can be calculated, which are nothing more than the coefficients of the characteristic polynomial. CHARPOLY works by forming the determinant of VAR * IDENT(N) - A. Thus NCHARPOLY wins, for example, in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether. It may be used by doing LOAD(NCHRPL);.

6.6.2 Eigenvalues and Eigenvectors

`EIGENVALUES` (*mat*) [*Function*]

takes a matrix *mat* as its argument and returns a list of lists, the first sublist of which is the list of eigenvalues of the matrix, and the other sublist of which is the list of the multiplicities of the eigenvalues in the corresponding order. It is able to handle multiple eigenvalues and the eigenvectors corresponding to those eigenvalues. It will work with any square matrix (not necessarily symmetric or hermitian) and will tell whether the matrix is diagonalizable. The calculated eigenvectors and the unit eigenvectors of the matrix are the right eigenvectors and the right unit eigenvectors respectively.

You should be aware of the fact that this program uses the **Maxima** functions SOLVE and ALGSYS and if SOLVE can not find the roots of the characteristic polynomial of the matrix or if it generates a rather messy solution the EIGEN package may not produce any useful results. This package is *designed* to try to get the *exact* solutions to the eigenvalue and eigenvector problems. if the matrices you have contain floating point numbers, it may not be able to solve your problem. you should use the imsl eigenvalue and eigenvector package for numerical matrices with floating point numbers. These excellent routines will find the approximate solutions for numerical matrices with floating point numbers.

The **Maxima** function SOLVE is used to find the roots of the characteristic polynomial of the matrix. Sometimes SOLVE may not be able to find the roots of the polynomial;in that case nothing in this package except CONJUGATE, INNERPRODUCT, UNITVECTOR, COLUMNVECTOR and GRAMSCHMIDT will work unless you know the eigenvalues. In some cases SOLVE may generate very messy eigenvalues. You may want to simplify the answers yourself before you go on. There are provisions for this and they will be explained below. This usually happens when SOLVE returns an apparently imaginary expression for an eigenvalue which is supposed to be real. The EIGENVALUES command is available directly from **Maxima**. To use the other functions you must have loaded in the EIGEN package, either by a previous call to EIGENVALUES, or by doing LOAD(EIGEN). There are some usage notes in the file: '`share/eigen.usg`'.

EIGENVECTORS (*mat*) [*Function*]

takes a matrix *mat* as its argument and returns a list of lists the first sublist of which is the output of the EIGENVALUES command and the other sublists of which are the eigenvectors of the matrix corresponding to those eigenvalues respectively. This function will work directly from **Maxima**, but if you wish to take advantage of the flags for controlling it (see below), you must first load in the EIGEN package from the SHARE directory. You may do that by LOAD(EIGEN);. The flags that affect this function are:

6.6.2.1 Eigenvalue Flags

NONDIAGONALIZEABLE [*variable, default: FALSE*]

will be set to TRUE or FALSE depending on whether the matrix is nondiagonalizable or diagonalizable after an EIGENVECTORS command is executed.

HERMETIANMATRIX [*variable, default: FALSE*]

If set to TRUE will cause the degenerate eigenvectors of the hermitian matrix to be orthogonalized using the Gram-Schmidt algorithm.

KNOWNEIGVALS [*variable, default: FALSE*]

If set to TRUE the EIGEN package will assume the eigenvalues of the matrix are known to the user and stored under the global name LISTEIGVALS.

KNOWNEIGVECTS [*variable, default: FALSE*]

If set to TRUE the EIGEN package will assume that the eigenvectors of the matrix are known to the user and are stored under the global name LISTEIGVECTS. LISTEIGVECTS should be set to a list similar to the output of the EIGENVECTORS command. (If KNOWNEIGVECTS is set to TRUE and the list of eigenvectors is given the setting of the flag NONDIAGONALIZABLE may not be correct. If that is the case please set it to the correct value. The author assumes that the user knows what he is doing and will not try to diagonalize a matrix the eigenvectors of which do not span the vector space of the appropriate dimension. . .)

LISTEIGENVALS [*Variable*]

should be set to a list similar to the output of the EIGENVALUES command. The **Maxima** function ALGSYSis used to solve for the eigenvectors. Sometimes if the eigenvalues are messy, ALGSYS may not be able to produce a solution. In that case you are advised to try to simplify the eigenvalues by first finding them using EIGENVALUES command and then using whatever marvelous tricks you might have to reduce them to something simpler. You can then use the KNOWNEIGVALS flag to proceed further. There are some usage notes in the file: ‘`share/eigen.usg`’.

MULTIPLICITIES [*variable, default: NOT_SET_YET*]

will be set to a list of the multiplicities of the individual solutions returned by SOLVE or REALROOTS.

6.6.2.2 Functions in the Eigenvalue Package

`CONJUGATE` (*X*) [*Function*]

is a function in the EIGEN package on the SHARE directory. It returns the complex conjugate of its argument. This package may be loaded by LOAD(EIGEN); . Note that %I's in the expressions should be explicit, since there is no complex variable declaration in **Maxima** at the present time. This is true for all the functions in this package. There are some usage notes in the file: '`share/eigen.usg`'.

`COLUMNVECTOR` (*list*) [*Function*]

a function in the EIGEN package. Do LOAD(EIGEN) to use it. COLUMNVECTOR takes a list as its argument and returns a column vector the components of which are the elements of the list. The first element is the first component, etc. This is useful if you want to use parts of the outputs of the functions in this package in matrix calculations. There are some usage notes in the file: '`share/eigen.usg`'.

`GRAMSCHMIDT` (*X*) [*Function*]

is a function in the EIGEN package. Do LOAD(EIGEN);to use it. GRAMSCHMIDT takes as its argument a list of lists, the sublists of which are of equal length and not necessarily orthogonal with respect to the innerproduct defined above, and returns a similar list each sublist of which is orthogonal to all others. The returned results may contain integers that are factored. This is due to the fact that the function FACTOR is used to simplify each substage of the Gram-Schmidt algorithm. This prevents the expressions from getting very messy and helps to reduce the sizes of the numbers that are produced along the way. There are some usage notes in the file: '`share/eigen.usg`'.

`INNERPRODUCT` (*X*, *Y*) [*Function*]

is a function in the EIGEN package. Do LOAD(EIGEN); to use it. INNERPRODUCT takes two lists of equal length as its arguments and returns their inner (scalar) product defined by *(Complex Conjugate of X)* . *Y* (The "dot" operation is the same as the usual one defined for vectors) . There are some usage notes in the file: '`share/eigen.usg`'.

`SIMILARITYTRANSFORM` (*mat*) [*Function*]

is a function in the EIGEN package. Do LOAD(EIGEN); to use it. SIMILARITYTRANSFORM takes a matrix as its argument and returns a list which is the output of the UNITEIGENVECTORS command. In addition if the flag NONDIAGONALIZABLE is FALSE two global matrices LEFTMATRIX and RIGHTMATRIX will be generated. These matrices have the property that LEFTMATRIX . MAT . RIGHTMATRIX is a diagonal matrix with the eigenvalues of MAT on the diagonal. If NONDIAGONALIZABLE is TRUE these two matrices will not be generated. If the flag HERMITIANMATRIX is TRUE, then LEFTMATRIX is the complex conjugate of the transpose of RIGHTMATRIX. Otherwise LEFTMATRIX is the inverse of RIGHTMATRIX. RIGHTMATRIX is the matrix the columns of which are the unit eigenvectors of MAT. The other flags (see EIGENVALUES and EIGENVECTORS) have the same effects since SIMILARITYTRANSFORM calls the other functions in the package in order to be able to form RIGHTMATRIX. There are some usage notes in the file: '`share/eigen.usg`'.

`UNITEIGENVECTORS` (*mat*) [*Function*]

is a function in the EIGEN package. Do `LOAD(EIGEN);` to use it. UNITEIGENVECTORS takes a matrix *mat* as its argument, and returns a list of lists the first sublist of which is the output of the EIGENVALUES command and the other sublists of which are the unit eigenvectors of the matrix corresponding to those eigenvalues respectively. The flags mentioned in the description of the EIGENVECTORS command have the same effects in this one as well. In addition there is a flag which may be useful:

There are some usage notes in the file: '`share/eigen.usg`'.

`UNITVECTOR` (*list*) [*Function*]

is a function in the EIGEN package. Do `LOAD(EIGEN);` to use it. UNITVECTOR takes a list as its argument and returns a unit list. (i.e. a list with unit magnitude). There are some usage notes in the file: '`share/eigen.usg`'.

CHAPTER

SEVEN

Series

7.1 Sums and Products

7.1.1 Sums

`SUM` (*exp, ind, low, high*) [*Special Form*]

performs a summation of the values of *exp* as the index *ind* varies from *low* to *high*. If the upper and lower limits differ by an integer, then each term in the sum is evaluated and added together. Otherwise, if the SIMPSUM default: [FALSE] is TRUE the result is simplified. This simplification may sometimes be able to produce a closed form. If SIMPSUM is FALSE or if `'`SUM is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics. If *high* is one less than *low*, we have an empty sum and SUM returns 0 rather than erroring out. Sums may be differentiated, added, subtracted, or multiplied with some automatic simplification being performed. There is a demo in the file: ‘`demo/sum.dem`’.

`NUSUM` (*exp, var, low, high*) [*Function*]

performs indefinite summation of *exp* with respect to `v`ar using a decision procedure due to R.W. Gosper. *exp* and the potential answer must be expressible as products of n’th powers, factorials, binomials, and rational functions. The terms “definite” and “indefinite summation” are used analogously to “definite” and “indefinite integration.” To sum indefinitely means to give a closed form for the sum over intervals of variable length, not just e.g. 0 to inf. Thus, since there is no formula for the general partial sum of the binomial series, NUSUM can’t do it. Both the summand and the answer must be expressible as products of n’th powers, factorials, binomials, and rational functions. There are some usage notes in the file: ‘`share/nusum.usg`’.

```
nusum(n*n!,n,0,n);                        ==>  (n+1)! - 1
nusum(n^4*4^n/binomial(2*n,n),n,0,n);     ==>  <moby mess>
unsum(\%,n);                               ==>  n^4*4^n/binomial(2*n,n)
```

`UNSUM` (*fun, int*) [*Syntax*]

is the first backward difference *fun(int)* - *fun(int-1)*. There are some usage notes in the file: ‘`share/nusum.usg`’.

```
(C1) G(P):=P*4^N/BINOMIAL(2*N,N);
                                          N
                                        P 4
(D1)                      G(P) := ----------------
                                  BINOMIAL(2 N, N)
(C2) G(N^4);
                                     4  N
                                    N  4
(D2)                           ----------------
                               BINOMIAL(2 N, N)
(C3) NUSUM(D2,N,0,N);
                         4        3       2              N
          2 (N + 1) (63 N  + 112 N  + 18 N  - 22 N + 3) 4      2
(D3)      ---------------------------------------------- - ------
                       693 BINOMIAL(2 N, N)                3 11 7
(C4) UNSUM(\%,N);
                                     4  N
                                    N  4
(D4)                           ----------------
                               BINOMIAL(2 N, N)
```

See section 5.1.5 [Combining Sums of Quotients], page 75 for the definitions of XTHRU, SUMCONTRACT, INTOSUM, COMBINE, and RNCOMBINE. See section 7.1.3 [Operations on Sums and Products], page 103, for the definitions of BASHINDICES, and NICEINDICES and RENAME.

7.1.1.1 Sum Flags

See section 2.4.5 [Display of Sums], page 27 for the definition of SUMEXPAND, CAUCHYSUM and SUMHACK.

`SIMPSUM` [*variable, default: FALSE*]

if TRUE, the result of a SUM is simplified. This simplification may sometimes be able to produce a closed form. If SIMPSUM is FALSE or if 'SUM is used, the value is a sum noun form, which is a representation of the sigma notation used in mathematics.

`GENINDEX` [*variable, default: I*]

is the alphabetic prefix used to generate the next variable of summation.

`GENSUMNUM` [*variable, default: 0*]

is the numeric suffix used to generate the next variable of summation. If it is set to FALSE then the index will consist only of GENINDEX with no numeric suffix.

See section 2.4.5 [Display of Sums], page 27 for the definition of SUMEXPAND, CAUCYSUM and SUMHACK.

7.1.2 Products

`PRODUCT` (*exp, ind, lo, hi*) [*Special Form*]

gives the product of the values of *exp* as the index *ind* varies from *lo* to *hi*. The evaluation is similar to that of SUM. No simplification of products is available at this time. If *hi* is one less than *lo*, we have an empty product, and PRODUCT returns 1 rather than erroring out. Also see PRODHACK.

```
(C1)  PRODUCT(X+I*(I+1)/2,I,1,4);
(D1)             (X + 1) (X + 3) (X + 6) (X + 10)
```

See section 2.4.6 [Display of Products], page 27 for the definition of PRODHACK.

7.1.3 Operations on Sums and Products

`BASHINDICES` (*exp*) [*Function*]

transforms the expression *exp* by giving each summation and product a unique index. This gives CHANGEVAR greater precision when it is working with summations or products. The form of the unique index is J<number>. The quantity <number> is determined by referring to GENSUMNUM, which can be changed by the user. For example, GENSUMNUM:0 resets it.

`NICEINDICES` (*exp*) [*Function*]

will take the expression and change all the indices of sums and products to something easily understandable. It makes each index it can `I`, unless `I` is in the internal expression, in which case it sequentially tries `J`,K,L,M,N,I0,I1,I2,I3,I4,... until it finds a legal index. Sums

`NICEINDICESPREF` [*variable, default: [I,J,K,L,M,N]*]

the list which NICEINDICES uses to find indices for sums and products. This allows the user to set the order of preference of how NICEINDICES finds the "nice indices." E.g. × NICEINDICESPREF:[Q,R,S,T,INDEX]. Then if NICEINDICES finds that it cannot use any of these as indices in a particular summation, it uses the first as a base to try and tack on numbers. Here, if the list is exhausted, Q0, then Q1, etc, will be tried.

`RENAME` (*exp*) [*Function*]

returns an expression equivalent to *exp* but with the dummy indices in each term chosen from the set [!1,!2,...]. Each dummy index in a product will be different; for a sum RENAME will try to make each dummy index in a sum the same. In addition, the indices will be sorted alphanumerically.

7.2 Power Series

POWERSERIES (*exp, var, pt*) [*Function*]

generates the general form of the power series expansion for *exp* in the variable *var* about the point *pt* (which may be INF for infinity). If POWERSERIES is unable to expand *exp*, the TAYLOR function may give the first several terms of the series.

VERBOSE default: [FALSE] - if TRUE will cause comments about the progress of POWERSERIES to be printed as the execution of it proceeds.

```
(C1) VERBOSE:TRUE$
(C2) POWERSERIES(LOG(SIN(X)/X),X,0);
Can't expand
                                  LOG(SIN(X))
So we'll try again after applying the rule:
                                        d
                                       / -- (SIN(X))
                                       [ dX
                          LOG(SIN(X)) = I ----------- dX
                                       ]   SIN(X)
                                       /
In the first simplification we have returned:
                                 /
                                 [
                                 I COT(X) dX - LOG(X)
                                 ]
                                 /
                        INF
                        ====         I1  2 I1             2 I1
                        \       (- 1)   2     BERN(2 I1) X
                         >      ------------------------------
                        /                I1 (2 I1)!
                        ====
                        I1 = 1
(D2)                    -------------------------------------
                                         2
```

7.3 Taylor Series

TAYLOR (*exp, var, pt, pow*) [*Function*]

expands the expression *exp* in a truncated Taylor series (or Laurent series, if required) in the variable *var* around the point *pt*. The terms through *(var-pt)**pow* are generated. If MAXTAYORDER default: [FALSE] is set to TRUE, then during algebraic manipulation of (truncated) Taylor series, TAYLOR will try to retain as many terms as are certain to be correct.

If *exp* is of the form *f(var)/g(var)* and *g(var)* has no terms up to degree *pow* then TAYLOR will try to expand *g(var)* up to degree *2*pow*. If there are still no non-zero terms TAYLOR will keep doubling the degree of the expansion of g(var) until reaching *pow*2**n* where *n* is the value of the variable TAYLORDEPTH default: [3].

TAYLOR*(exp, [var1,pt1,ord1], [var2,pt2,ord2], ...)* returns a truncated power series in the variables *vari* about the points *pti*, truncated at *ordi*.

TAYLOR*(exp, [var1, var2, ...], pt, ord)* where each of *pt* and *ord* may be replaced by a list which will correspond to the list of variables. That is, the *n*'th items on each of the lists will be associated together.

TAYLOR*(exp, [x,pt,ord,ASYMP])* will give an expansion of *exp* in negative powers of *(x-pt)*. The highest order term will be *(x-pt)**(-ord)*. The ASYMP is a syntactic device and not to be assigned to. See also the TAYLOR_LOGEXPAND switch for controlling expansion. There is a demo in the file: '`demo/taylor.dem`'.

DEFTAYLOR (*function, exp*) [*Special Form*]

allows the user to define the Taylor series (about 0) of an arbitrary function of one variable as *exp* which may be a polynomial in that variable, or which may be given implicitly as a power series using the SUM function.

In order to display the information given to DEFTAYLOR one can use POWERSERIES(F(X),X,0).

```
(C1) DEFTAYLOR(F(X),X**2+SUM(X**I/(2**I*I!**2),
        I,4,INF));
(D1)                          [F]
(C2) TAYLOR(\%E**SQRT(F(X)),X,0,4);
                      2         3         4
                     X    3073 X    12817 X
(D2)/R/     1 + X + -- + ------- + -------- +  . . .
                     2     18432     307200
```

TAYLORINFO (*exp*) [*Function*]

returns FALSE if *exp* is not a Taylor series. Otherwise, a list of lists is returned describing the particulars of the Taylor expansion. For example,

```
(C3) TAYLOR((1-Y^2)/(1-X),X,0,3,[Y,A,INF]);
              2                       2
(D3)/R/ 1 - A  - 2 A (Y - A) - (Y - A)
                     2                         2
              + (1 - A  - 2 A (Y - A) - (Y - A) ) X
           2                        2   2
 + (1 - A  - 2 A (Y - A) - (Y - A) ) X
                     2                         2   3
              + (1 - A  - 2 A (Y - A) - (Y - A) ) X
     + . . .
(C4) TAYLORINFO(D3);
(D4)                      [[Y, A, INF], [X, 0, 3]]
```

7.3.1 Taylor Series Operations

TAYTORAT (*exp*) [*Function*]

converts *exp* from TAYLOR form to CRE form, i.e. it is like RAT(RATDISREP(exp)) although much faster.

REVERT (*exp, var*) [*Function*]

does reversion of Taylor series. *var* is the variable the original Taylor expansion *exp* is in. Do LOAD(REVERT); to access this function. Try REVERT(*exp, var);* The expression must be a Taylor series about *var*=0.

REVERT2 (*exp, var, hipower*) [*Function*]

The REVERT2 function truncates the returned polynomial to *hipower* order in the variable *var*.

There is a demo in the file: ‘`share2/revert.dem`’. There are some usage notes in the file: ‘`share2/revert.usg`’.

TAYLOR_SIMPLIFIER (*exp*) [*Function*]

A function of one argument which TAYLOR uses to simplify coefficients of power series.

TRUNC (*exp*) [*Function*]

causes *exp* which is in general representation to be displayed as if its sums were truncated Taylor series. E.g. compare EXP1:X**2+X+1; with EXP2:TRUNC(X**2+X+1); . Note that IS(EXP1=EXP2); -> TRUE.

7.3.2 Taylor Series Flags

MAXTAYORDER [*variable, default: TRUE*]

if TRUE, then during algebraic manipulation of (truncated) Taylor series, TAYLOR will try to retain as many terms as are certain to be correct.

MAXTAYDEPTH [*variable, default: 3*]

If there are still no non-zero terms TAYLOR will keep doubling the degree of the expansion of *g(var)* until reaching *pow*2**n* where *n* is the value of the variable TAYLORDEPTH default: [3].

TAYLOR_LOGEXPAND [*variable, default: TRUE*]

controls expansions of logarithms in TAYLOR series. When TRUE all LOG's are expanded fully so that zero-recognition problems involving logarithmic identities do not disturb the expansion process. However, this scheme is not always mathematically correct since it ignores branch information. If TAYLOR_LOGEXPAND is set to FALSE, then the only expansion of LOG's that will occur is that necessary to obtain a formal power series.

TAYLOR_ORDER_COEFFICIENTS [*variable, default: TRUE*]

controls the ordering of coefficients in the expression. The default is that coefficients of Taylor series will be ordered canonically.

TAYLOR_TRUNCATE_POLYNOMIALS [*variable, default: TRUE*]

When FALSE polynomials input to TAYLOR are considered to have infinite precision; otherwise (the default) they are truncated based upon the input truncation levels.

7.4 Pade Approximates

PADE (*taylor-series, num-deg-bound, denom-deg-bound*) [*Function*]

returns a list of all rational functions which have the given *taylor-series* expansion, where the sum of the degrees of the numerator and the denominator is less than or equal to the truncation level of the power series, i.e. are best approximants, and which additionally satisfy the specified degree bounds. Its first argument must be a univariate Taylor Series, the second and third are positive integers specifying degree bounds on the numerator and denominator.

PADE's first argument can also be a Laurent series, and the degree bounds can be INF which causes all rational functions whose total degree is less than or equal to the length of the power series to be returned. Total degree is *num-degree* + *denom-degree*. Length of a power series is truncation level + 1 - minimum(0, *order of series*).

7.5 Poisson Series

INTOPOIS (*exp*) [*Function*]

converts *exp* into a Poisson encoding.

OUTOFPOIS (*exp*) [*Function*]

converts *exp* from Poisson encoding to general representation. If *exp* is not in Poisson form, it will make the conversion, i.e. it will look like the result of OUTOFPOIS(INTOPOIS(A)). This function is thus a canonical simplifier for sums of powers of SIN's and COS's of a particular type.

`POISPLUS` (*A, B)* [*Function*]

is functionally identical to `I`NTOPOIS(A+B).

`POISTIMES` (*A, B)* is functionally identical to `I`NTOPOIS(A*B) [*Function*]

.

`POISEXPT` (*A, B*) (B a positive integer) [*Function*]

is functionally identical to `I`NTOPOIS(A**B).

`POISSIMP` (*exp*) [*Function*]

converts *exp* into a Poisson series for *exp* in general representation.

`POISSUBST` (*A, B, C*) [*Function*]

substitutes A for B in C. C is a Poisson series. (1) Where B is a variable U, V, W, X, Y, or Z then A must be an expression linear in those variables (e.g. 6*U+4*V). (2) Where B is other than those variables, then A must also be free of those variables, and furthermore, free of sines or cosines.

POISSUBST*(A, B, C, D, N)* is a special type of substitution which operates on A and B as in type (1) above, but where D is a Poisson series, expands COS(D) and SIN(D) to order N so as to provide the result of substituting A+D for B in C. The idea is that D is an expansion in terms of a small parameter. For example, POISSUBST(U,V,COS(V),E,3); -> COS(U)*(1-E**2/2) - SIN(U)*(E-E**3/6).

`PRINTPOIS` (*exp*) [*Function*]

prints a Poisson series in a readable format. In common with OUTOFPOIS, it will convert *exp* into a Poisson encoding first, if necessary.

`POISMAP` (*series, sinfun, cosfun*) [*Function*]

will map the functions `s`infun on the SIN terms and *cosfun* on the COS terms of the Poisson series *series*. *sinfun* and *cosfun* are functions of two arguments which are a coefficient and a trigonometric part of a term in series respectively.

`POISDIFF` (*A, B*) [*Function*]

differentiates *A* with respect to *B*. B must occur only in the trig arguments or only in the coefficients.

`POISINT` (*A, B*) [*Function*]

integrates in a similarly restricted sense (to POISDIFF). Non-periodic terms in B are dropped if B is in the trig arguments.

POISTRIM*()* is a reserved function name which (if the user has defined it) gets applied during Poisson multiplication. It is a predicate function of 6 arguments which are the coefficients of the U, V,..., Z in a term. Terms for which POISTRIM is TRUE (for the coefficients of that term) are eliminated during multiplication.

`POISLIM` [*variable, default: 5*]

determines the domain of the coefficients in the arguments of the trig functions. The initial value of 5 corresponds to the interval [-2**(5-1)+1, 2**(5-1)], or [-15,16], but it can be set to [-2**(n-1)+1, 2**(n-1)].

7.6 Continued Fractions

`CF` (*exp*) [*Special Form*]

converts *exp* into a continued fraction. *exp* is an expression composed of arithmetic operators and lists which represent continued fractions. A continued fraction a+1/(b+1/(c+...)) is represented by the list [a,b,c,...]. a,b,c, ... must be integers. *exp* may also involve SQRT(n) where *n* is an integer. In this case CF will give as many terms of the continued fraction as the value of the variable CFLENGTH default: [1] times the period. Thus the default is to give one period. (CF binds LISTARITH to FALSE so that it may carry out its function.) There is a demo in the file: '`demo/cf.dem`'.

`CFDISREP` (*list*) [*Function*]

converts the continued fraction represented by list into general representation.

```
(C1) CF([1,2,-3]+[1,-2,1]);
(D1)                  [1, 1, 1, 2]
(C2) CFDISREP(\%);
                            1
(D2)                1 + ---------
                              1
                        1 + -----
                                1
                            1 + -
                                2
```

`CFEXPAND` (*x*) [*Function*]

gives a matrix of the numerators and denominators of the next-to-last and last convergents of the continued fraction *x*.

```
(C1) CF(SQRT(3));
(D1)                 [1, 1, 2, 1, 2, 1, 2, 1]
(C2) CFEXPAND(\%);
                              [71  97]
(D2)                          [      ]
                              [41  56]
(C3) D2[1,2]/D2[2,2],NUMER;
(D3)                         1.7321429
```

CFLENGTH [*variable, default: 1*]

controls the number of terms of the continued fraction the function CF will give, as the value CFLENGTH[1] times the period. Thus the default is to give one period.

CHAPTER

EIGHT

Calculus

8.1 Limits

`LIMIT` (*exp, var, val, dir*) [*Function*]

finds the limit of *exp* as the real variable *var* approaches the value *val* from the direction *dir*. *Dir* may have the value PLUS for a limit from above, MINUS for a limit from below, or may be omitted (implying a two-sided limit is to be computed). For the method see [Wan71]. LIMIT uses the following special symbols: INF (positive infinity) and MINF (negative infinity). On output it may also use **UND** (undefined), **IND** (indefinite but bounded) and **INFINITY** (complex infinity). There is a demo in the file: '`demo/limit.dem`'.

`LHOSPITALLIM` [*variable, default: 4*]

is the maximum number of times L'Hospital's rule is used in LIMIT. This prevents infinite looping in cases like LIMIT(COT(X)/CSC(X),X,0).

`LIMSUBST` [*variable, default: FALSE*]

prevents LIMIT from attempting substitutions on unknown forms. This is to avoid bugs like LIMIT(F(N)/F(N+1),N,INF); giving 1. Setting LIMSUBST to `TRUE` will allow such substitutions. Since LIMIT is often called upon to simplify constant expressions, for example, `I`NF-1, LIMIT may be used in such cases with only one argument, e.g. LIMIT(INF-1);.

`TLIMSWITCH` [*variable, default: FALSE*]

when `TRUE` will cause the LIMIT package to use Taylor series when possible.

`TLIMIT` (*exp, var, val, dir*) [*Function*]

is just the function LIMIT with TLIMSWITCH set to `TRUE`.

8.2 Residues

`RESIDUE` (*exp, var, val*) [*Function*]

computes the residue in the complex plane of the expression *exp* when the variable *var* assumes the value *val*. The residue is the coefficient of *(var-val)**(-1)* in the Laurent series for *exp*.

```
(C1) RESIDUE(S/(S**2+A**2),S,A*\%I);
                          1
(D1)                      -
                          2
(C2) RESIDUE(SIN(A*X)/X**4,X,0);
                          3
                         A
(D2)                   - --
                         6
```

8.3 Differentiation

`DIFF` (*exp, var1, int1, var2, int2, ...*) [*Function*]

differentiates *exp* with respect to each *vari*, *inti* times. If just the first derivative with respect to one variable is desired, then the form DIFF(exp,var) may be used. If the noun form of the function is required (as, for example, when writing a differential equation), 'DIFF should be used and this will display in a two dimensional format.

DIFF(*exp*) gives the total differential, that is, the sum of the derivatives of *exp* with respect to each of its variables times the function **DEL** of the variable. No further simplification of DEL is offered.

```
(C1) DIFF(EXP(F(X)),X,2);
                                2
                      F(X)  d             F(X)  d         2
(D1)                \%E     (--- F(X)) + \%E     (-- (F(X)))
                              2                   dX
                             dX
(C2) DERIVABBREV:TRUE$
(C3) 'INTEGRATE(F(X,Y),Y,G(X),H(X));
                                H(X)
                               /
                               [
(D3)                           I     F(X, Y) dY
                               ]
                               /
                                G(X)
(C4) DIFF(\%,X);
           H(X)
          /
          [
(D4)      I     F(X, Y)  dY + F(X, H(X)) H(X)  - F(X, G(X)) G(X)
          ]            X                     X                   X
          /
           G(X)
```

See section 4.2.2 [Operations on CRE Expressions], page 53 for the definition of RATDIFF, to differentiate polynomials in CRE form. .

`GENDIFF` (*exp, var, n*) [*Function*]

Sometimes DIFF(exp, var, n) can be reduced even though *n* is symbolic. LOAD(GENDIF); and you can try, for example, DIFF(%E**(A*X),X,Q) by using GENDIFF rather than DIFF. Items that cannot be evaluated come out quoted. Some items are in terms of GENFACT. There are some usage notes in the file: '`share2/gendif.usg`'.

`DERIVDEGREE` (*exp, dv, iv*) [*Function*]

finds the highest degree of the derivative of the dependent variable *dv* with respect to the independent variable *iv* occurring in *exp*.

```
(C1) 'DIFF(Y,X,2)+'DIFF(Y,Z,3)*2+'DIFF(Y,X)*X**2$
(C2) DERIVDEGREE(\%,Y,X);
(D2)                           2
```

8.3.1 Differentiation Flags

`DERIVABBREV` [*variable, default: FALSE*]

if TRUE will cause derivatives to display as subscripts.

8.3.2 Defining Gradients

GRADEF (*fun(var1, ..., varn), exp1, ..., expn*) [*Special Form*]

defines the derivatives of the function *fun* with respect to its *n* arguments. That is, *dfun/dvari = expi* etc. If fewer than *n* gradients, say *i*, are given, then they refer to the first *i* arguments of *fun*. The *vari* are merely dummy variables as in function definition headers, and are used to indicate the *i*'th argument of *fun*. All arguments to GRADEF except the first are evaluated so that if one of *expi* is a defined function then it is invoked and the result is used.

Gradients are needed when, for example, a function is not known explicitly but its first derivatives are and it is desired to obtain higher order derivatives. GRADEF may also be used to redefine the derivatives of **Maxima**'s predefined functions, e.g. GRADEF(SIN(X),SQRT(1-SIN(X)**2));. It is not permissible to use GRADEF on subscripted functions. PRINTPROPS([fun1, fun2, ...],GRADEF) may be used to display the GRADEFs of the functions *fun1,fun2,...*.

GRADEF(*atom, var, exp*) may be used to state that the derivative of the atomic variable *atom* with respect to *var* is *exp*. This automatically does a DEPENDS(a,v). PRINTPROPS([a1,a2,...],ATOMGRAD) may be used to display the atomic gradient properties of *a1,a2,...*.

GRADEFS [*Variable*]

is a list of the functions which have been given gradients by use of the GRADEF command, i.e. GRADEF(f(x1, ..., xn), g1, ..., gn).

ATOMGRAD [*Variable*]

the atomic gradient property of an expression. May be set by GRADEF.

8.3.3 Defining Functional Dependencies

DEPENDS (*funlist1, varlist1, funlist2, varlist2, ...*) [*Function*]

declares functional dependencies for variables to be used by DIFF. × DEPENDS([F,G],[X,Y],[R,S],[U,V,W],U,T) informs DIFF that F and G depend on X and Y, that R and S depend on U,V, and W, and that U depends on T. The arguments to DEPENDS are evaluated. The variables in each *funlist* are declared to depend on all the variables in the next *varlist*. A *funlist* can contain the name of an atomic variable or array. In the latter case, it is assumed that all the elements of the array depend on all the variables in the succeeding varlist.

Initially, DIFF(F,X) is 0; executing DEPENDS(F,X) causes future differentiations of F with respect to X to give DF/DX or Y if DERIVABBREV is set to TRUE.

```
(C1) DEPENDS([F,G],[X,Y],[R,S],[U,V,W],U,T);
(D1)           [F(X, Y), G(X, Y), R(U, V, W), S(U, V, W), U(T)]
(C2) DEPENDENCIES;
(D2)           [F(X, Y), G(X, Y), R(U, V, W), S(U, V, W), U(T)]
(C3) DIFF(R.S,U);
                                dR           dS
(D3)                            -- . S + R . --
                                dU           dU
```

Since **Maxima** knows the chain rule for symbolic derivatives, it takes advantage of the given dependencies as follows:

```
(C4) DIFF(R.S,T);
                             dR dU              dS dU
(D4)                        (-- --) . S + R . (-- --)
                             dU dT              dU dT
```

If we set

```
(C5) DERIVABBREV:TRUE;
(D5)                                  TRUE
```

then re-executing the command C4, we obtain

```
(C6) ''C4;
(D6)                        (R  U ) . S + R . (S  U )
                              U  T              U  T
```

To eliminate a previously declared dependency, the REMOVE command can be used. For example, to say that R no longer depends on U as declared in C1, the user can type REMOVE(R,DEPENDENCY). This will eliminate all dependencies that may have been declared for R.

```
(C7) REMOVE(R,DEPENDENCY);
(D7)                                  DONE
(C8) ''C4;
(D8)                               R . (S  U )
                                     U    T
```

DIFF is the only **Maxima** command which uses DEPENDENCIES information. The arguments to INTEGRATE, LAPLACE, etc. must be given their dependencies explicitly in the command, e.g., INTEGRATE(F(X),X).

DEPENDENCIES [*Variable*]

the list of atoms which have functional dependencies (set up by the DEPENDS or GRADEF functions). The older command DEPENDENCIES has been replaced by the DEPENDS command.

8.3.4 Differentiating Tensors

For the tensor package, the following modifications have been incorporated:

1. the derivatives of any indexed objects in *exp* will have the variables *vi* appended as additional arguments. Then all the derivative indices will be sorted.
2. the *vi* may be integers from 1 up to the value of the variable DIMENSION default: 4. This will cause the differentiation to be carried out with respect to the *vi*th member of the list COORDINATES which should be set to a list of the names of the coordinates, e.g., [x, y, z, t]. If COORDINATES is bound to an atomic variable, then that variable subscripted by *vi* will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like X[1], X[2],... to be used. If COORDINATES has not been assigned a value, then the variables will be treated as in (1) above.

See section 16.2 [Tensors], page 213.

8.4 Integration

Maxima has several routines for handling integration. The INTEGRATE command makes use of most of them. There is also the ANTID package, which handles an unspecified function (and its derivatives, of course). For numerical uses, there is the ROMBERG function, and the IMSL version of Romberg, DCADRE. There is also an adaptive integrator which uses the Newton-Cotes 8 panel quadrature rule, called QUANC8.

Generally speaking, **Maxima** only handles integrals which are integrable in terms of the elementary functions (rational functions, trigonometrics, logs, exponentials, radicals, etc.) and a few extensions (error function, dilogarithm). It does not handle integrals in terms of unknown functions such as *g(x)* and *h(x)*.

`INTEGRATE` (*exp*, *var*) [*Function*]

integrates *exp* with respect to *var* or returns an integral expression (the noun form) if it cannot perform the integration (see note 1 below). Roughly speaking three stages are used:

1. INTEGRATE sees if the integrand is of the form F(G(X))*DIFF(G(X),X) by testing whether the derivative of some subexpression (i.e. G(X) in the above case) divides the integrand. If so, it looks up F in a table of integrals and substitutes G(X) for X in the integral of F. This may make use of gradients in taking the derivative. If an unknown function appears in the integrand it must be eliminated in this stage or else INTEGRATE will return the noun form of the integrand.
2. INTEGRATE tries to match the integrand to a form for which a specific method can be used, e.g. trigonometric substitutions.
3. If the first two stages fail, it uses the Risch algorithm.

Functional relationships must be explicitly represented in order for INTEGRATE to work properly. INTEGRATE is not affected by DEPENDENCIES set up with the DEPENDS command.

INTEGRATE(*exp, var, low, high*) finds the definite integral of *exp* with respect to *var* from *low* to *high*, or returns the noun form if it cannot perform the integration. The limits should not contain *var*. Several methods are used, including direct substitution in the indefinite integral and contour integration. Improper integrals may use the names INF for positive infinity and MINF for negative infinity. If an integral form is desired for manipulation (for example, an integral which cannot be computed until some numbers are substituted for some parameters), the noun form 'INTEGRATE may be used and this will display with an integral sign. The function LDEFINT uses LIMIT to evaluate the integral at the lower and upper limits.

Sometimes during integration the user may be asked what the sign of an expression is. Suitable responses are **POS**;, **ZERO**;, or **NEG**;.

```
(C1) INTEGRATE(SIN(X)**3,X);
                     3
                  COS (X)
(D1)              ------- - COS(X)
                     3
(C2) INTEGRATE(X**A/(X+1)**(5/2),X,0,INF);
IS  A + 1  POSITIVE, NEGATIVE, OR ZERO?
POS;
IS  2 A - 3  POSITIVE, NEGATIVE, OR ZERO?
NEG;
                              3
(D2)             BETA(A + 1, - - A)
                              2
(C3) GRADEF(Q(X),SIN(X**2));
(D3)                                Q(X)
(C4) DIFF(LOG(Q(R(X))),X);
                             d             2
                            (-- R(X)) SIN(R (X))
                             dX
(D4)                        --------------------
                                  Q(R(X))
(C5) INTEGRATE(\%,X);
(D5)                           LOG(Q(R(X)))
```

The fact that **Maxima** does not perform certain integrals does not always imply that the integral does not exist in closed form. In the example below the integration call returns the noun form but the integral can be found fairly easily. For example, one can compute the roots of X**3+X+1 = 0 to rewrite the integrand in the form 1/((X-A)*(X-B)*(X-C)) where A, B and C are the roots. **Maxima** will integrate this equivalent form although the integral is quite complicated.

```
(C6) INTEGRATE(1/(X^3+X+1),X);
                            /
                            [     1
(D6)                        I ---------- dX
                            ]  3
                            / X  + X + 1
```

ABCONVTEST [*variable, default: FALSE*]

when TRUE causes INTEGRATE to test for absolute convergence.

INTEGRATION_CONSTANT_COUNTER [*Variable*]

is a counter which is updated each time a constant of integration (called by **Maxima**, e.g., INTEGRATIONCONSTANT1) is introduced into an expression by indefinite integration of an equation.

LOGABS [*variable, default: FALSE*]

when doing indefinite integration where LOGs are generated, e.g. INTEGRATE(1/X,X), the answer is given in terms of LOG(ABS(...)) if LOGABS is TRUE, but in terms of LOG(...) if LOGABS is FALSE. For definite integration, the LOGABS:TRUE setting is used, because here evaluation of the indefinite integral at the endpoints is usually needed.

DEFINT (*exp, var, low, high*) [*Function*]

DEFinite INTegration, the same as INTEGRATE(exp, var, low, high). There is a demo in the file: '`demo/defint.dem`'.

LDEFINT (*exp, var, ll, ul*) [*Function*]

yields the definite integral of *exp* by using LIMIT to evaluate the indefinite integral of *exp* with respect to *var* at the upper limit *ul* and at the lower limit *ll*.

TLDEFINT (*exp, var, ll, ul*) [*Function*]

is just LDEFINT with TLIMSWITCH set to TRUE.

RISCH (*exp, var*) [*Function*]

integrates *exp* with respect to *var* using the transcendental case of the Risch algorithm. (The algebraic case of the Risch algorithm has not been implemented.) This currently handles the cases of nested exponentials and logarithms which the main part of INTEGRATE can't do. INTEGRATE will automatically apply RISCH if given these cases.

```
(C1) RISCH(X^2*ERF(X),X);
            2     2
          - X     X             3           2
        \%E     (\%E   SQRT(\%PI) X  ERF(X) + X  + 1)
(D1)    ----------------------------------------
                      3 SQRT(\%PI)
(C2) DIFF(\%,X),RATSIMP;
                                2
(D2)                           X  ERF(X)
```

There is a demo in the file: '`demo/risch.dem`'.

ERFFLAG [*variable, default: TRUE*]

if FALSE prevents RISCH from introducing the ERF function in the answer if there were none in the integrand to begin with.

DBLINT (*'F(X,Y), 'R(X), 'S(X), a, b*) [*Function*]

a double-integral routine which was written in top-level **Maxima** and then translated and compiled to machine code. Do LOAD(DBLINT); to access this package. It uses the Simpson's Rule method in both the x and y directions to calculate

```
/B /S(X)
|  |
|  |    F(X,Y) DY DX .
|  |
/A /R(X)
```

The function *F(X,Y)* must be a translated or compiled function of two variables, and *R(X)* and *S(X)* must each be a translated or compiled function of one variable, while *a* and *b* must be floating point numbers. The routine has two global variables which determine the number of divisions of the x and y intervals: **DBLINT_X** default: 10 and **DBLINT_Y** default: 10, and can be changed independently to other integer values (there are 2*DBLINT_X+1 points computed in the x direction, and 2*DBLINT_Y+1 in the y direction).

The routine subdivides the X axis and then for each value of X it first computes *R(X)* and *S(X)*; then the Y axis between R(X) and S(X) is subdivided and the integral along the Y axis is performed using Simpson's Rule; then the integral along the X axis is done using Simpson's Rule with the function values being the Y-integrals. This procedure may be numerically unstable for a great variety of reasons, but is reasonably fast: avoid using it on highly oscillatory functions and functions with singularities (poles or branch points in the region).

The Y integrals depend on how far apart *R(X)* and *S(X)* are, so if the distance S(X)-R(X) varies rapidly with X, there may be substantial errors arising from truncation with different step-sizes in the various Y integrals. One can increase DBLINT_X and DBLINT_Y in an effort to improve the coverage of the region, at the expense of computation time. The function values are not saved, so if the function is very time-consuming, you will have to wait for re-computation if you change anything (sorry).

It is required that the functions *F*, *R*, and *S* be either translated or compiled prior to calling DBLINT. This will result in orders of magnitude speed improvement over interpreted code in many cases! The file '`share1/dblint.dem`' can be run in batch or demo mode to illustrate the usage on a sample problem; the file '`share1/dblnt.dm1`' is an extension of the demo, which also makes use of other numerical aids, FLOATDEFUNK and QUANC8.

Note: Simpson's Rule specifies that

```
/X[2*N]
|
|       F(X) DX = H/3* (F(X[0]) +
|
/X[0]                    4*(F(X[1])+F(X[3])+...+F(X[2*N-1])) +
                         2*(F(X[2])+F(X[4])+...+F(X[2*N-2])) +
                         F(X[2*N]))
```

in one dimension, where H is the distance between the equally spaced X[N]'s, and DBLINT_X=N. The error in this formulation is of order H**5*N*DIFF(F(X),X,4) for some X in (X[0],X[2*N]).

There are some usage notes in the file: 'share1/dblint.usg'. There is a demo in the file: 'share1/dblint.dem'. There is a demo in the file: 'share1/dblint.dm1'.

INTSCE (*exp, var*) [*Function*]

'share1/intsce.mc' contains a routine, written by Richard Bogen, for integrating products of sines, cosines and exponentials of the form EXP(A*X+B) *COS(C*X)**N*SIN(C*X)**M. *exp* may be any expression, but if it is not in the above form then the regular integration program will be invoked if the switch ERRINTSCE default: [TRUE] is TRUE.There are some usage notes in the file: 'share1/intsce.usg'.

ERRINTSCE [*variable, default: TRUE*]

If a call to the INTSCE routine is not of the form EXP(A*X+B)*COS(C*X)**N*SIN(C*X)**M then the regular integration program will be invoked if the switch ERRINTSCE default: [TRUE] is TRUE. If it is FALSE then INTSCE will error out.

ANTIDIFF (*G, X, U(X)*) [*Function*]

is a routine for evaluating integrals of expressions involving an arbitrary unspecified function and its derivatives. It may be used by LOAD(ANTID);, after which, the function ANTIDIFF may be used. E.g. ANTIDIFF(G,X,U(X)); where G is the expression involving U(X) (U(X) arbitrary) and its derivatives, whose integral with respect to X is desired.

The functions **NONZEROANDFREEOF** and **LINEAR** are also defined, as well as ANTID. **ANTID** is the same as ANTIDIFF except that it returns a list of two parts, the first part is the integrated part of the expression and the second part of the list is the non-integrable remainder.

SPECINT (*EXP(-p*var)*expr, var*) [*Function*]

The Hypergeometric Special Functions Package HYPGEO is still under development. At the moment it will find the Laplace Transforms or rather, the integral from 0 to INF of some special functions or combinations of them. The factor, EXP(-P*var) must be explicitly stated. *var* is the variable of integration and *expr* may be any expression containing special functions (at your own risk). Special function notation follows:

%J[index (exp)] Bessel Funct of the 1st Kind

%K[index (exp)] Bessel Funct of the 12nd Kind

%I[( )] Modified Bessel

%HE[( )] Hermite Polynomial

%P[( )] Legendre Function of the 1st Kind

%Q[( )] Legendre of the 2nd Kind

HSTRUVE[( )] Struve H Function

LSTRUVE[( )] Struve L Function

%F[([],[], exp)] Hypergeometric Function

GAMMA() Gamma Function

GAMMAGREEK()

GAMMAINCOMPLETE() Incomplete Gamma Function

SLOMMEL() Slommel Function

%M[()] Whittaker Funct of the 1st Kind

%W[()] Whittaker Funct of the 1st Kind

Not available in **Maxima** at this time. There is a demo in the file: '`share1/hypgeo.dem`'. There are some usage notes in the file: '`share1/hypgeo.usg`'.

8.5 Change of Variable

`CHANGEVAR` (*exp*, *f(x,y)*, *x*, *y*) [*Function*]

makes the change of variable given by *f(x,y) = 0* in all integrals occurring in *exp* with integration with respect to *x*; *y* is the new variable.

```
(C1) 'INTEGRATE(\%E**SQRT(A*Y),Y,0,4);
                     4
                    /
                    [    SQRT(A) SQRT(Y)
(D1)                I (\%E               ) dY
                    ]
                    /
                     0
(C2) CHANGEVAR(D1,Y-Z^2/A,Z,Y);
                                 2 SQRT(A)
                                /
                                [              Z
                              2 I           Z \%E  dZ
                                ]
                                /
                                 0
(D4)                           ----------------------
                                         A
```

CHANGEVAR may also be used to changes in the indices of a sum or product. However, it must be realized that when a change is made in a sum or product, this change must be a shift, i.e. `I` = J+ ..., not a higher degree function. E.g.

```
(C3) SUM(A[I]*X^(I-2),I,0,INF);
                                 INF
                                 ====
                                 \          I - 2
(D3)                              >    A  X
                                 /      I
                                 ====
                                 I = 0
(C4) CHANGEVAR(\%,I-2-N,N,I);
                                INF
                                ====
                                \                N
(D4)                             >      A       X
                                /        N + 2
                                ====
                                N = - 2
```

8.6 Laplace Transforms

LAPLACE (*exp, ovar, lvar*) [*Function*]

takes the Laplace transform of *exp* with respect to the variable *ovar* and transform parameter *lvar*. *exp* may only involve the functions EXP, LOG, SIN, COS, SINH, COSH, and ERF. It may also be a linear, constant coefficient differential equation in which case ATVALUE of the dependent variable will be used. These may be supplied either before or after the transform is taken. Since the initial conditions must be specified

at zero, if one has boundary conditions imposed elsewhere he can impose these on the general solution and eliminate the constants by solving the general solution for them and substituting their values back. *exp* may also involve convolution integrals.

Functional relationships must be explicitly represented in order for LAPLACE to work properly. That is, if F depends on X and Y it must be written as F(X,Y) wherever F occurs as in × LAPLACE('DIFF(F(X,Y),X),X,S). LAPLACE is not affected by DEPENDENCIES set up with the DEPENDS command.

```
(C1) LAPLACE(\%E**(2*T+A)*SIN(T)*T,T,S);
                        A
                   2 \%E  (S - 2)
(D1)              ---------------
                             2     2
                   ((S - 2)  + 1)
```

ILT (*exp, lvar, ovar*) [*Function*]

takes the inverse Laplace transform of *exp* with respect to *lvar* and parameter *ovar*. *exp* must be a ratio of polynomials whose denominator has only linear and quadratic factors. By using the functions LAPLACE and ILT together with the SOLVE or LINSOLVE functions the user can solve a single differential or convolution integral equations, or a set of them.

```
(C1) 'INTEGRATE(SINH(A*X)*F(T-X),X,0,T)+B*F(T)=T**2;
                T
               /
               [                                      2
(D1)           I (SINH(A X) F(T - X)) dX + B F(T) = T
               ]
               /
               0
(C2) LAPLACE(\%,T,S);
              A LAPLACE(F(T), T, S)
(D2)          ---------------------
                      2    2
                     S  - A
                                              2
                  + B LAPLACE(F(T), T, S) = --
                                               3
                                              S
(C3) LINSOLVE([\%],['LAPLACE(F(T),T,S)]);
SOLUTION
                                          2      2
                                       2 S  - 2 A
(E3)       LAPLACE(F(T), T, S) = --------------------
                                     5         2     3
                                  B S  + (A - A  B) S
(D3)                          [E3]
(C4) ILT(E3,S,T);
IS  A B (A B - 1)  POSITIVE, NEGATIVE, OR ZERO?
POS;
                                          2
                       SQRT(A) SQRT(A B  - B) T
                  2 COSH(------------------------)
                                    B
(D4)  F(T) =  - -------------------------------
                               A
               2
            A T             2
          + ------- + ------------------
            A B - 1    3  2      2
                      A  B  - 2 A  B + A
```

DELTA (*exp*) [*Function*]

is the Dirac Delta function. Currently only LAPLACE knows about the DELTA function:

```
(C1) LAPLACE(DELTA(T-A)*SIN(B*T),T,S);
Is A positive, negative or zero?
POS;
                                          - A S
(D1)                          SIN(A B) \%E
```

8.7 Specifying Boundary Conditions

ATVALUE (*form, list, value*) [*Function*]

enables the user to assign the boundary value *value* to form *form*, at the points specified by *list*.

```
(C1) ATVALUE(F(X,Y),[X=0,Y=1],A**2)$
```

The *form* must be a function, *f(v1,v2,...)*, or a derivative, *DIFF(f(v1, v2, ...), vi, ni, vj, nj, ...)* in which the functional arguments explicitly appear (*ni* is the order of differentiation with respect *vi*).

The list of equations determine the boundary at which the value is given; *list* may be a list of equations, as above, or a single equation, *vi = exp*.

The symbols @1, @2, ... will be used to represent the functional variables *v1, v2,* ... when ATVALUEs are displayed. PRINTPROPS([f1, f2, ...], ATVALUE) will display the ATVALUEs of the functions *f1,f2,...* as specified in previously given uses of the ATVALUE function. If the list contains just one element, then the element can be given without being in a list. If a first argument of ALL is given, then ATVALUEs for all functions which have them will be displayed.

AT (*exp, list*) [*Function*]

will evaluate *exp* (which may be any expression) with the variables assuming the values as specified for them in the list of equations or the single equation similar to that given to the ATVALUE function. If a subexpression depends on any of the variables in list but it hasn't had an ATVALUE specified and it can't be evaluated, then a noun form of the AT will be returned which will display in a two-dimensional form.

CHAPTER

NINE

Solving

9.1 Solving Expressions

SOLVE (*exp, var*) [*Function*]

solves the algebraic equation *exp* for the variable *var* and returns a list of solution equations in *var*. If *exp* is not an equation, it is assumed to be an expression to be set equal to zero. *Var* may be a function (e.g. F(X)), or other non-atomic expression except a sum or product. It may be omitted if *exp* contains only one variable. *exp* may be a rational expression, and may contain trigonometric functions, exponentials, etc. The following method is used:

Let *E* be the expression and *X* be the variable. If *E* is linear in X then it is trivially solved for *X*. Otherwise, if *E* is of the form A*X**N+B then the result is (-B/A)**(1/N) times the Nth roots of unity.

If *E* is not linear in *X* then the GCD of the exponents of *X* in *E* (say N) is divided into the exponents and the multiplicity of the roots is multiplied by N. Then SOLVE is called again on the result. If *E* factors then SOLVE is called on each of the factors. Finally SOLVE will use the quadratic, cubic, or quartic formulas where necessary.

In the case where *E* is a polynomial in some function of the variable to be solved for, say F(X), then it is first solved for F(X) (call the result C), then the equation F(X)=C can be solved for *X* provided the inverse of the function F is known.

SOLVE*([eq1, ..., eqn], [v1, ..., vn])* solves a system of simultaneous (linear or non-linear) polynomial equations by calling LINSOLVE or ALGSYS, and returns a list of the solution lists in the variables. In the case of LINSOLVE this list would contain a single list of solutions. It takes two lists as arguments. The first list represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted. For linear systems if the given equations are not compatible, the message INCONSISTENT will be displayed (see the SOLVE_INCONSISTENT_ERROR switch); if no unique solution exists, then SINGULAR will be displayed. There is an example in the file: '`example/solve.xmp`'. There is a demo in the file: '`demo/solve.dem`'.

FUNCSOLVE (*eqn, g(t)*) [*Function*]

gives *[g(t) = ...]* or *[]*, depending on whether or not there exists a rational function g(t) satisfying *eqn*, which must be a first order, linear polynomial in (for this case) *g(t)* and *g(t+1)*.

```
(C1) FUNCSOLVE((N+1)*FOO(N)-(N+3)*FOO(N+1)/(N+1) =
     (N-1)/(N+2),FOO(N));
                                    N
(D1)                FOO(N) = ---------------
                             (N + 1) (N + 2)
```

Warning: this is a very rudimentary implementation: many safety checks and obvious generalizations are missing.

9.1.1 Solve Flags

BREAKUP [*variable, default: TRUE*]

if FALSE will cause SOLVE to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default. BREAKUP:TRUE only works when PROGRAMMODE is FALSE.

GLOBALSOLVE [*variable, default: FALSE*]

if set to TRUE then variables which are SOLVEd for will be set to the solution of the set of simultaneous equations.

SOLVEDECOMPOSES [*variable, default: TRUE*]

if TRUE, will induce SOLVE to use POLYDECOMP in attempting to solve polynomials.

SOLVEEXPLICIT [*variable, default: FALSE*]

if TRUE, inhibits SOLVE from returning implicit solutions i.e. of the form F(x)=0.

SOLVEFACTORS [*variable, default: TRUE*]

if FALSE then SOLVE will not try to FACTOR the expression. The FALSE setting may be desired in some cases where factoring is not necessary.

SOLVENULLWARN [*variable, default: TRUE*]

if TRUE the user will be warned if he calls SOLVE with either a null equation list or a null variable list. For example, SOLVE([],[]); would print two warning messages and return [].

SOLVERADCAN [*variable, default: FALSE*]

if TRUE then SOLVE will use RADCAN which will make SOLVE slower but will allow certain problems containing exponentials and logs to be solved.

SOLVETRIGWARN [*variable, default: TRUE*]

if set to FALSE will inhibit printing by SOLVE of the warning message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.

SOLVE_INCONSISTENT_ERROR [*variable, default: TRUE*]

If TRUE, SOLVE and LINSOLVE give an error if they meet up with a set of inconsistent linear equations, e.g. SOLVE([A+B=1,A+B=2]). If FALSE, they return [].

9.2 Solving Linear Equations

LINSOLVE (*[exp1, exp2, ...], [var1, var2, ...]*) [*Function*]

solves the list of simultaneous linear equations for the list of variables. The *expi* must each be polynomials in the variables and may be equations. If GLOBALSOLVE default: [FALSE] is set to TRUE, then variables which are SOLVEd for will be set to the solution of the set of simultaneous equations.

BACKSUBST [*variable, default: TRUE*]

if set to FALSE will prevent back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions.

```
(C1) X+Z=Y$
(C2) 2*A*X-Y=2*A**2$
(C3) Y-2*Z=2$
(C4) LINSOLVE([D1,D2,D3],[X,Y,Z]),GLOBALSOLVE:TRUE;
SOLUTION
(E4)                            X : A + 1
(E5)                             Y : 2 A
(E6)                            Z : A - 1
(D6)                          [E4, E5, E6]
```

LINSOLVEWARN [*variable, default: TRUE*]

if FALSE will cause the message Dependent equations eliminated to be suppressed.

LINSOLVE_PARAMS [*variable, default: TRUE*]

If TRUE, LINSOLVE also generates the *%Ri* symbols used to represent arbitrary parameters described in the manual under ALGSYS. If FALSE, LINSOLVE behaves as before, i.e. when it meets up with an under-determined system of equations, it solves for some of the variables in terms of others.

9.3 Solving Simultaneous Equations

ALGSYS ([*exp1, exp2, . . .], [var1, var2, . . .]*) [*Function*]

solves the list of simultaneous polynomials or polynomial equations (which can be non-linear), for the list of variables. The symbols %R1, %R2, etc. will be used to represent arbitrary parameters when needed for the solution (the variable %RNUM_LIST holds these). In the process described below, ALGSYS is entered recursively if necessary.

The method is as follows:

1. First the equations are FACTORed and split into subsystems.
2. For each subsystem Si, an equation *E* and a variable *var* are selected (the *var* is chosen to have lowest nonzero degree). Then the resultant of *E* and Ej with respect to *var* is computed for each of the remaining equations Ej in the subsystem Si. This yields a new subsystem S'i in one fewer variables (var has been eliminated). The process now returns to (1).
3. Eventually, a subsystem consisting of a single equation is obtained. If the equation is multivariate and no approximations in the form of floating point numbers have been introduced, then SOLVE is called to find an exact solution. The user should realize that SOLVE may not be able to produce a solution, or if it does the solution may be a very large expression.

   If the equation is univariate and is either linear, quadratic, or bi-quadratic, then again SOLVE is called if no approximations have been introduced. If approximations have been introduced or the equation is not univariate and neither linear, quadratic, or bi-quadratic, then if the switch REALONLY default: [FALSE] is TRUE, the function REALROOTS is called to find the real-valued solutions. If REALONLY:FALSE then ALLROOTS is called which looks for real and complex-valued solutions. If ALGSYS produces a solution which has fewer significant digits than required, the user can change the value of ALGEPSILON default: [10**8] to a higher value. If ALGEXACT default: [FALSE] is set to TRUE, SOLVE will always be called.
4. Finally, the solutions obtained in step (3) are re-inserted into previous levels and the solution process returns to (1).

The user should be aware of several caveats:

When ALGSYS encounters a multivariate equation which contains floating point approximations (usually due to its failing to find exact solutions at an earlier stage), then it does not attempt to apply exact methods to such equations and instead prints the message:

```
ALGSYS cannot solve - system too complicated.
```

Interactions with RADCAN can produce large or complicated expressions. In that case, the user may use PICKAPART or REVEAL to analyze the solution. Occasionally, RADCAN may introduce an apparent %I into a solution which is actually real-valued. There is a demo in the file: '`demo/algsys.dem`'.

9.3.1 Algsys Flags

ALGEPSILON [*variable, default: 10**8*]

The value of epsilon used by ALGSYS.

ALGEXACT [*variable, default: FALSE*]

affects the behavior of ALGSYS as follows: If ALGEXACT is TRUE, ALGSYS always calls SOLVE and then uses REALROOTS on SOLVE's failures. If ALGEXACT is FALSE, SOLVE is called only if the eliminant was not univariate, or if it was a quadratic or biquadratic. Thus ALGEXACT:TRUE doesn't guarantee only exact solutions, just that ALGSYS will first try as hard as it can to give exact solutions, and only yield approximations when all else fails.

%R [*Variable*]

The arbitrary parameters used by ALGSYS.

%RNUM_LIST [*Variable*]

When %R variables are introduced in solutions by the ALGSYS command, they are added to %RNUM_LIST in the order they are created. This is convenient for doing substitutions into the solution later on. It's recommended to use this list rather than doing CONCAT('%R,J).

REALONLY [*variable, default: FALSE*]

if TRUE causes ALGSYS to return only those solutions which are free of %I.

9.4 Roots of Polynomials

ALLROOTS (*poly*) [*Function*]

finds all the real and complex roots of the real polynomial *poly* which must be univariate and may be an equation, e.g. poly=0. For complex polynomials an algorithm by Jenkins and Traub is used (Algorithm 419, Comm. ACM, vol. 15, (1972), p. 97). For real polynomials the algorithm used is due to Jenkins (Algorithm 493, TOMS, vol. 1, (1975), p.178). ALLROOTS may give inaccurate results in case of multiple roots. (If *poly* is real and you get inaccurate answers, you may want to try ALLROOTS(%I*poly);).

ALLROOTS rejects non-polynomials. It requires that the numerator after RATting should be a polynomial, and it requires that the denominator be at most a complex number. As a result of this ALLROOTS will always return an equivalent (but factored) expression, if POLYFACTOR is TRUE.

POLYFACTOR [*variable, default: FALSE*]

when TRUE causes ALLROOTS to factor the polynomial over the real numbers if the polynomial is real, or over the complex numbers, if the polynomial is complex.

NROOTS (*poly, low, high*) [*Function*]

finds the number of real roots of the real univariate polynomial *poly* in the half-open interval *(low,high]*. The endpoints of the interval may also be MINF, INF respectively for minus infinity and plus infinity. The method of Sturm sequences is used.

```
(C1) POLY1:X**10-2*X**4+1/2$
(C2) NROOTS(POLY1,-6,9.1);
RAT REPLACED 0.5 BY 1/2 = 0.5
(D2)                                4
```

NTHROOT (*poly, int*) [*Function*]

where *poly* is a polynomial with integer coefficients and *int* is a positive integer returns *q*, a polynomial over the integers, such that *q***n=p* or prints an error message indicating that *poly* is not a perfect *n'*th power. This routine is much faster than FACTOR or even SQFR.

REALROOTS (*poly, bound*) [*Function*]

finds all of the real roots of the real univariate polynomial *poly* within a tolerance of *bound* which, if less than 1, causes all integral roots to be found exactly. The parameter *bound* may be arbitrarily small in order to achieve any desired accuracy. The first argument may also be an equation. REAL-ROOTS sets MULTIPLICITIES, useful in case of multiple roots. REALROOTS(poly) is equivalent to REALROOTS(poly,ROOTSEPSILON).

ROOTSEPSILON [*variable, default: 1.0E-7*]

a real number used to establish the confidence interval for the roots found by the REALROOTS function.

POLY_DISCRIMINANT (*exp, var*) [*Function*]

computes the discriminant of the polynomial *exp*, with respect to *var*, which is the square of the product of the differences of all pairs of roots.

ROOTSCONTRACT (*exp*) [*Function*]

converts products of roots into roots of products. For example, ROOTSCONTRACT(SQRT(X)*Y**(3/2)); ->SQRT(X*Y**3). When RADEXPAND is TRUE and DOMAIN is REAL (their defaults), ROOTSCON-TRACT converts ABS into SQRT, e.g. ROOTSCONTRACT(ABS(X)*SQRT(Y)); -> SQRT(X**2*Y).

ROOTSCONMODE [*variable, default: TRUE*]

affects ROOTSCONTRACT as follows:

```
Problem                 ROOTSCONMODE       Applying ROOTSCONTRACT

X^(1/2)*Y^(3/2)         FALSE              (X*Y^3)^(1/2)
X^(1/2)*Y^(1/4)         FALSE              X^(1/2)*Y^(1/4)
X^(1/2)*Y^(1/4)         TRUE               (X*Y^(1/2))^(1/2)
X^(1/2)*Y^(1/3)         TRUE               X^(1/2)*Y^(1/3)
X^(1/2)*Y^(1/4)         ALL                (X^2*Y)^(1/4)
X^(1/2)*Y^(1/3)         ALL                (X^3*Y^2)^(1/6)
```

(The above examples and more may be tried out by typing EXAMPLE(ROOTSCONTRACT);.)

When ROOTSCONMODE is FALSE, ROOTSCONTRACT contracts only with respect to rational number exponents whose denominators are the same. The key to the ROOTSCONMODE:TRUE examples is simply that 2 divides into 4 but not into 3. ROOTSCONMODE:ALL involves taking the LCM (least common multiple) of the denominators of the exponents. ROOTSCONTRACT uses RATSIMP in a manner similar to LOGCONTRACT.

9.5 Interpolation

INTERPOLATE (*fun, var, a, b*) [*Function*]

finds the zero of *fun* as *var* varies. The last two args give the range to look in. The function must have a different sign at each endpoint. If this condition is not met, the action of the of the function is governed by INTPOLERROR default: [TRUE]). If INTPOLERROR is TRUE then an error occurs, otherwise the value of INTPOLERROR is returned (thus for plotting INTPOLERROR might be set to 0.0). Otherwise (given that **Maxima** can evaluate the first argument in the specified range, and that it is continuous) INTERPOLATE is guaranteed to come up with the zero (or one of them if there is more than one zero). The accuracy of INTERPOLATE is governed by INTPOLABS default: [0.0] and INTPOLREL default [0.0] which must be non-negative floating point numbers. INTERPOLATE will stop when the first *arg* evaluates to something less than or equal to INTPOLABS, or if successive approximants to the root differ by no more than INTPOLREL * <one of the approximants>. The default values of INTPOLABS and INTPOLREL are 0.0 so INTERPOLATE gets as good an answer as is possible with the single precision arithmetic.

The first argument *fun* may be an equation. The order of the last two args is irrelevant. Thus INTERPOLATE(SIN(X)=X/2,X,%PI,.1); is equivalent to INTERPOLATE(SIN(X)=X/2,X,.1,%PI);. The method used is a binary search in the range specified by the last two args. When it thinks the function is close enough to being linear, it starts using linear interpolation.

An alternative syntax has been added to INTERPOLATE, this replaces the first two arguments by a function name. The function must be TRANSLATEd or COMPILEd function of one argument. No checking of the result is done, so make sure the function returns a floating point number.

```
F(X):=(MODE_DECLARE(X,FLOAT),SIN(X)-X/2.0);
INTERPOLATE(SIN(X)-X/2,X,0.1,\%PI);      time= 60 msec
INTERPOLATE(F(X),X,0.1,\%PI);            time= 68 msec
TRANSLATE(F);
INTERPOLATE(F(X),X,0.1,\%PI);            time= 26 msec
INTERPOLATE(F,0.1,\%PI);                 time=  5 msec
```

NEWTON (*exp, var, X0, eps*) [*Function*]

The file 'share/newton.mc' contains a function which will do interpolation using Newton's method. It may be accessed by LOAD(NEWTON); . The Newton method can do things that INTERPOLATE will refuse to handle, since INTERPOLATE requires that everything evaluate to a flonum. Thus NEWTON(x**2-a**2,x,a/2,a**2/100); will say that it can't tell if flonum*a**2<a**2/100. Doing ASSUME(a>0);, and then doing NEWTON again works. You get x=a+<small flonum>*a, which is symbolic all the way. × INTERPOLATE(x**2-a**2,x,a/2,2*a); complains that .5*a is not a flonum.

9.5.1 Interpolation Flags

INTERPOLERROR [*variable, default: TRUE*]

When INTERPOLATE is called, it determines whether or not the function to be interpolated satisfies the condition that the values of the function at the endpoints of the interpolation interval are opposite in sign. If they are of opposite sign, the interpolation proceeds. If they are of like sign, and INTPOLERROR is TRUE, then an error is signaled. If they are of like sign and INTPOLERROR is not TRUE, the value of INTPOLERROR is returned. Thus for plotting, INTPOLERROR might be set to 0.0.

INTERPOLABS [*variable, default: 0.0*]

The accuracy of the INTERPOLATE command is governed by INTPOLABS default: [0.0] and INTPOLREL default: [0.0] which must be non-negative floating point numbers. INTERPOLATE will stop when the first arg evaluates to something less than or equal to INTPOLABS or if successive approximants to the root differ by no more than INTPOLREL * <one of the approximants>. The default values of INTPOLABS and INTPOLREL are 0.0, so INTERPOLATE gets as good an answer as is possible with the single precision arithmetic we have.

INTERPOLREL [*variable, default: 0.0*]

The accuracy of the INTERPOLATE command is governed by INTPOLABS default: [0.0] and INTPOLREL default: [0.0] which must be non-negative floating point numbers. INTERPOLATE will stop when the first arg evaluates to something less than or equal to INTPOLABS or if successive approximants to the root differ by no more than INTPOLREL * <one of the approximants>. The default values of INTPOLABS and INTPOLREL are 0.0, so INTERPOLATE gets as good an answer as is possible with the single precision arithmetic we have.

9.6 Solving Ordinary Differential Equations

ODE (*eqn*, *y*, *x*) [*Function*]

is a pot-pourri of Ordinary Differential solvers combined in such a way as to attempt more and more difficult methods as each fails. For example, the first attempt is with ODE2, so therefore, a user using ODE can assume he has all the capabilities of ODE2 at the very beginning, and if he/she has been using ODE2 in programs they will still run if he substitutes ODE, as the returned values, and calling sequence are identical. In addition, ODE has a number of user features which can assist an experienced ODE solver if the basic system cannot handle the equation. These will be covered completely toward the end of this description, but, essentially, he can make transforms of the dependent and independent variables, find the invariant in the normal form, compute the normal form, the Schwartzian derivative, the adjoint or try various particular solutions of the equation. These features are used to some extent in ODE's attempts to find a general solution. The user can also control a primitive learning capability of the program, i.e., it will remember his attempts at trial particular solutions as long as he does not read in a fresh version. The program is called with ODE(eqn, y, x); where equation is of the same form as required for ODE2; i.e.:

```
'diff(y,x,2)*F(x) + 'diff(y,x)*G(x) +y*H(x) = K(x,y)
```

The term K(x,y) is only permissible (for now) in first order equations; e.g. 'diff(y,x)=K(x,y). It's presence in a second order equation; i.e.,trying to solve a nonlinear higher order will return FALSE.

9.6.1 First Order Equations

Nearly all of the known methods for handling first order equations are present in ODE. The more basic methods; e.g. Bernoulli, Generalized Homogeneous, Linear, Separable, are contained in ODE2 written by J.P.Golden. [Lew79]. This program is also used by many of the other methods after they have done the necessary transformations required by their algorithm. The following additional methods are applied if ODE2 fails to solve the equation:

DIFFSOL This is the method of Laplace Transforms. Basically, it converts the DE to the form DIFF(Y(X),X) + F(Y(X),X) = 0, sets the ATVALUEs Y(X) at 0 to %K1 and DIFF(Y(X),X) at 0 to %K2 and then calls DESOLVE.

NONLIN This will solve an equation nonlinear in Y' according to the method of Ince. The variable P is substituted for DIFF(Y,X) and the resulting equation is solved for P. The two solutions thus obtained are then reconverted to ODE's in Y, X and solved by a recursive call to ODE. The two solutions are returned.

NONLIN1 This method is used for the special case in which the coefficients of Y' and Y are polynomials in X and Y and are homogeneous (have the same powers of the variables). The transformation %v=y/x is made and the resulting equation solved by ODE2.

RICCATI There are two Schmidt algorithms used by the Riccati solver: the first, called SCHMIDT, finds in systematic way solutions of the form P(X) where P is a polynomial in X. The coefficients of X**n in P may be symbolic but n must be a constant. The second, called RICSOL, is a heuristic which substitutes various likely looking expressions to see if they are solutions. RICSOL can solve equations having functions of X; eg, COS(x), as coefficients in the DE.

EULER MULTIPLIER Some limited search for an integrating factor is performed by ODE2. This method goes to additional lengths (as described by Schmidt) to find other possible factors. This is the last method we use because it can consume lots of time, especially if it can't find a multiplier.

9.6.2 Second Order Equations

After a failing attempt to solve by ODE2, we then try DIFFSOL, which is merely a new name for Bogen's Laplace Transform method, in which the appropriate notation has been substituted and the ATVALUEs for the dependent variable and its derivatives have been set to %k1, %k2, etc.

Failing that, the following methods are used in the solution and the variable, METHOD, will be set accordingly:

Invariant constant If the invariant of the DE in the normal form is a constant the substitution: y=v*exp(-1/2*integral(G(x)dx)) is made and the equation is solved by ODE2 using the Constant Coefficients rule. The answer is retransformed.

Solution of adjoint If the adjoint of the equation is solvable the answer is returned after retransformation.

Change of independent variable If the value of G'(x)+2*H(x)*G(x)/G(x)**3/2 is a constant, the equation is transformed into a new equation via: z= integral(G(x)**1/2 dx)*c where c is a constant chosen to simplify the result. The resulting equation, which now has constant coefficients, is solved by ODE2 and the result retransformed.

Try a solution. Next, the fact that y=R(x) may be a solution is used by scanning through the list, TRYLIST default: [], and changing the dependent variable via: y= R * v and solving the resulting equation for v and retransforming. Note that, in a pinch, the user may CONS a solution of his own into TRYLIST (if he thinks he knows one).

The program will try some more sophisticated methods at this point. The first, solution by factorization of the differential operator, uses the Riccati equation solver (see Lafferty in [Lew79]).

Failing the above, we are left with the unfortunate alternative of SERIES solutions. But first, we can see if we have a Hypergeometric or a Whittaker. This is done by looking at the singularities of the equation. If it possesses three singularities, all regular, then we have a Hypergeometric and solve by means of a Riemann P-Symbol. If two singularities, one irregular and at infinity (or at zero with the regular one at infinity), we transform such that the singularities are at [zero, infinity] and generate a Whittaker solution. Both the Whittaker and Hypergeometric solutions are then fed to the Hypergeometric series reduction routine to generate a closed form, if possible. Otherwise, the program returns the series in the form: %F[m,n]([a],[b],arg), where a and b are the lists of factorial function arguments and arg is an expression. The user has the some control of this process in that he can suppress the generation of the hypergeometric by setting the flag **CLOSEDFORM** default: [TRUE] to FALSE, and further, he can cause the result to display as a sum by setting the flag **SUMFORM** default: [FALSE] to TRUE.

Failing to find either of the preceding cases, the program will default to the SERIES solver (see Lafferty in [Lew79]). This will default to the truncated Taylor form (see Fateman in [Lew79]). if no complete solution is obtainable from the recurrence relation. The complete solution can be obtained in any of the above forms; i.e., closed, hypergeometric or sum by setting the various flags.

9.6.2.1 Assistance For Experienced Users

There are a number of transformation routines which are available in the environment of ODE for experienced users to manipulate their ODE with the intention of solving it by one of the methods above. These are described here. For further info and theory see Rainville-IDE and Ince. By the environment of ODE is meant that ODE should have been run at least once in the current **Maxima**, otherwise the routines will not autoload.

DEPTRAN(*EQN, Y, X, V, F*) will transform using Y=V*F, X=X.

INDTRAN(*EQN, Y, X, Z, F*) will transform using Y=Y, Z=F(x).

INVARIANT(*EQN, Y, X*) will compute the invariant in the normal form of EQN and return it as a function of X.

NORMALFORM(*EQN, Y, X, V*) will transform EQN to its normalform in terms of the new dependent variable V. Note that V is related to Y by Y = V*exp(-1/2*integrate(P,X)) where P is the coefficient of dy/dx in the original equation.

SCHWARTZIAN(*S, X*) will compute the Schwartzian derivative of s with respect to x, denoted in the literature by {s, x}. It is defined by the third order ODE: {s, x} = s"'/s' - 3/2*(s"/s')**2. An attempt is made to simplify the result trigonometrically.

DADJOINT(*EQN, Y, X, W*) will return the Adjoint equation of *EQN*. The definition is as follows: A(EQ)= W" + P*W' + (Q-P')*W = 0 where P and Q are the coefficients in the original equation and ' denotes differentiation with respect to X.

TRANSFORM(*EQN, Y, X, V, Z, [RELATIONS]*) is a general transform package which can handle two variable transforms as well as the single ones described above. It is somewhat by the current capabilities of SOLVE. The relations can be any two equations relating the old and new variables; example:

```
TRANSFORM(EQ,Y,X,V,Z,[V=Y*SIN(X),Z=COS(X)]);
```

Some simplification of the resulting equation is done, both trigonometric and algebraic with the intention of a future call to ODE.

9.6.2.2 ODE Options

Optional arguments can be given to ODE following the third mandatory argument which can have the following values:

ANY equivalent to no value, i.e., run the methods as in the current version a solution is found.

ALL run all appropriate (a test is made for degree) methods even if one or more return a solution. Return a list of solutions including FALSE.

SERIES run SERIES in closedform mode.

SOLVEHYPER solve as a hypergeometric using P-symbols

WHITTAKER solve as a confluent hyper using tables of Kummer solutions.

ODE2 run ODE2 on it.

DIFFSOL solve by Laplace transforms.

DESOL solve using more advanced methods.

SOLFAC solve by factoring the operator.

RICCATI run the Riccati solver.

NONLIN solve for nonlinear first order in Y'.

NONLIN1 solve for nonlinear first order in Y.

Example: ODE('diff(y,x,2)=0, y, x, series, ode2); will apply SERIES and ODE2 in that order and return a list of the two solutions obtained.

A user may have his own favorite method which he may want to include in the list. This can be done easily for the ALL case or for a specific call, but not for the ANY or default case.

Three demos are available. They should be run in the following order: 'ode/ode.dm1', 'ode/ode.dm2', 'ode/ode.dm3'. The first is a small sample of the first order capability, especially Riccati equations. The second shows some of the second order capability, Legendre and Bessel equations. The demos also show how some of the variables and switches can be used to help see what is happening. There is a demo in the file: 'ode/ode.dem'. There is a demo in the file: 'ode/ode.dm1'. There is a demo in the file: 'ode/ode.dm2'. There is a demo in the file: 'ode/ode.dm3'. There are some usage notes in the file: 'ode/ode.usg'.

ODE2 (*exp, dvar, ivar*) [*Function*]

takes three arguments: an ODE of first or second order (only the left hand side need be given if the right hand side is 0), the dependent variable *dvar*, and the independent variable *ivar*. When successful, it returns either an explicit or implicit solution for the dependent variable. %C is used to represent the constant in the case of first order equations, and %K1 and %K2 the constants for second order equations. If ODE2 cannot obtain a solution for whatever reason, it returns FALSE, after perhaps printing out an error message.

The methods implemented for first order equations in the order in which they are tested are: linear, separable, exact - perhaps requiring an integrating factor, homogeneous, Bernoulli's equation, and a generalized homogeneous method. For second order: constant coefficient, exact, linear homogeneous with non-constant coefficients which can be transformed to constant coefficient, the Euler or equidimensional equation, the method of variation of parameters, and equations which are free of either the independent or of the dependent variable so that they can be reduced to two first order linear equations to be solved sequentially.

In the course of solving ODEs, several variables are set purely for informational purposes: METHOD denotes the method of solution used e.g. LINEAR, INTFACTOR denotes any integrating factor used, ODEINDEX denotes the index for Bernoulli's method or for the generalized homogeneous method, and YP denotes the particular solution for the variation of parameters technique. There is a demo in the file: 'demo/ode2.dem'. There are some usage notes in the file: 'share/ode2.usg'.

9.6.2.3 Boundary Value Problems

IC1 (*exp, var, var*) [*Function*]

In order to solve initial value problems (IVPs) and boundary value problems (BVPs), the routine IC1 is

available in the ODE2 package for first order equations, and **IC2** and **BC2** for second order IVPs and BVPs, respectively. Do LOAD(ODE2); to access these. They are used as in the following examples:

```
(C3) IC1(D2,X=\%PI,Y=0);
                          COS(X) + 1
(D3)                Y = - ----------
                                3
                               X
(C4) 'DIFF(Y,X,2) + Y*'DIFF(Y,X)^3 = 0;
                        2
                       d Y      dY 3
(D4)                   --- + Y (--)  = 0
                         2      dX
                       dX
(C5) ODE2(\%,Y,X);
                   3
                  Y  - 6 \%K1 Y - 6 X
(D7)              ------------------ = \%K2
                           3
(C8) RATSIMP(IC2(D7,X=0,Y=0,'DIFF(Y,X)=2));
                     3
                  2 Y  - 3 Y + 6 X
(D9)              - ---------------- = 0
                            3
(C10) BC2(D7,X=0,Y=1,X=1,Y=3);
                   3
                  Y  - 10 Y - 6 X
(D11)             --------------- = - 3
                         3
```

In order to see more clearly which methods have been implemented, a demonstration file is available. There is a demo in the file: '`demo/ode2.dem`'. There are some usage notes in the file: '`share/ode2.usg`'.

DESOLVE (*[eq1, . . . , eqn], [fun1, . . . , funn]*) [*Function*]

where the *eqi* are differential equations in the dependent variables *var1,. . . ,varn*. The functional relationships must be explicitly indicated in both the equations and the variables. For example

```
(C1) 'DIFF(F,X,2)=SIN(X)+'DIFF(G,X);
(C2) 'DIFF(F,X)+X^2-F=2*'DIFF(G,X,2);
```

is not the proper format. The correct way is:

```
(C3) 'DIFF(F(X),X,2)=SIN(X)+'DIFF(G(X),X);
(C4) 'DIFF(F(X),X)+X^2-F(X)=2*'DIFF(G(X),X,2);
```

The call is then DESOLVE([D3,D4],[F(X),G(X)]);. If initial conditions at 0 are known, they should be supplied before calling DESOLVE by using ATVALUE.

```
(C11) 'DIFF(F(X),X)='DIFF(G(X),X)+SIN(X);
                        d          d
(D11)                   -- F(X) = -- G(X) + SIN(X)
                        dX         dX
(C12) 'DIFF(G(X),X,2)='DIFF(F(X),X)-COS(X);
                         2
                        d           d
(D12)                   --- G(X) = -- F(X) - COS(X)
                          2         dX
                        dX
(C13) ATVALUE('DIFF(G(X),X),X=0,A);
(D13)                                A
(C14) ATVALUE(F(X),X=0,1);
(D14)                                1
(C15) DESOLVE([D11,D12],[F(X),G(X)]);
                  X                              X
(D16) [F(X)=A \%E   - A+1, G(X) = COS(X) + A \%E   - A + G(0) - 1]
/* VERIFICATION */
(C17) [D11,D12],D16,DIFF;
                   X       X      X                 X
(D17)         [A \%E   = A \%E , A \%E   - COS(X) = A \%E   - COS(X)]
```

If DESOLVE cannot obtain a solution, it returns FALSE.

9.7 Integral Equations

IEQN (*ieqn, unk, tech, n, guess*) [*Function*]

is an integral equation solving routine. Do LOAD(INTEQN); to access it.

ieqn is the integral equation

unk is the unknown function;

tech is the technique to be tried from those given above. *tech* = FIRST means: try the first technique which finds a solution; *tech* = ALL means: try all applicable techniques);

n is the maximum number of terms to take for TAYLOR, NEUMANN, FIRSTKINDSERIES, or FREDSERIES (it is also the maximum depth of recursion for the differentiation method);

guess is initial guess for NEUMANN or FIRSTKINDSERIES.

Default values for the 2'nd thru 5'th parameters are:

unk P(X), where P is the first function encountered in an integrand which is unknown to **Maxima** and X is the variable which occurs as an argument to the first occurrence of P found outside of an integral in the case of SECONDKIND equations, or is the only other variable besides the variable of integration in FIRSTKIND equations. If the attempt to search for X fails, the user will be asked to supply the independent variable;

tech FIRST

n 1

guess NONE, which will cause NEUMANN and FIRSTKINDSERIES to use F(X) as an initial guess.

The value returned by IEQN is a list of labels of solution lists. the solution lists are printed as they are found unless the option variable IEQNPRINT is set to FALSE. These lists are of the form *[solution, technique used, nterms, flag]* where *flag* is absent if the solution is exact. Otherwise it is the word APPROXIMATE or INCOMPLETE corresponding to an inexact or non-closed form solution respectively. If a series method was used, *NTERMS* gives the number of terms taken which could be less than the *n* given to IEQN if an error was encountered preventing generation of further terms. There are some usage notes in the file: 'share1/inteqn.usg'.

IEQNPRINT [*variable, default: TRUE*]

governs the behavior of the result returned by the IEQN command. If IEQNPRINT is set to FALSE, the lists returned by the IEQN function are of the form [SOLUTION, TECHNIQUE USED, NTERMS, FLAG], where FLAG is absent if the solution is exact. Otherwise, it is the word APPROXIMATE or INCOMPLETE corresponding to an inexact or non-closed form solution, respectively. If a series method was used, NTERMS gives the number of terms taken (which could be less than the n given to IEQN if an error prevented generation of further terms).

There are some usage notes in the file: 'share1/inteqn.usg'.

CHAPTER

TEN

Maxima Knowledge Database

10.1 Adding to the Database

10.1.1 Defining Operators

The term operator is used in either of two senses: syntactic meaning that it has special syntax properties in the **Maxima** language, or semantic referring to its functionality. In the syntactic sense it is something which usually consists of non-alphanumeric characters, e.g. + or * (exceptions include AND, OR, and NOT). Semantically we sometimes refer to the operator of an expression, meaning that thing which is in the operator part of the expression, such as the + in A+B or SIN in SIN(x). Note: + in this latter example is also an operator in the syntactic sense, whereas SIN is a mathematical function.

It is possible to add new operators to **Maxima** (INFIX, PREFIX, POSTFIX, UNARY, or MATCHFIX with given precedences), to remove existing operators, or to redefine the precedence of existing operators. While **Maxima**'s syntax should be adequate for most ordinary applications, it is possible to define new operators or eliminate predefined ones that get in the user's way. The extension mechanism is rather straightforward and should be evident from the examples below.

```
(C1) PREFIX("DDX")$
(C2) DDX Y$              means                   "DDX"(Y)
(C3) INFIX("<-")$
(C4) A<-DDX Y$           means                "<-"(A,"DDX"(Y))
```

For each of the types of operator except SPECIAL, there is a corresponding creation function that will give the lexeme specified the corresponding parsing properties. Thus `(`"DDX") will make DDX a prefix operator just like - or NOT. Of course, certain extension functions require additional information such as the matching keyword for a matchfix operator. In addition, binding powers and parts of speech must be specified for all keywords defined. This is done by passing additional arguments to the extension functions. If a user does not specify these additional parameters, **Maxima** will assign default values. The six extension functions with binding powers and parts of speech defaults (enclosed in brackets) are summarized below.

1. PREFIX(operator, rbp[180], rpos[ANY], pos[ANY])
2. POSTFIX(operator, lbp[180], lpos[ANY], pos[ANY])
3. INFIX(operator, lbp[180], rbp[180], lpos[ANY], rpos[ANY],pos[ANY])
4. NARY(operator, bp[180], argpos[ANY], pos[ANY])

5. NOFIX(operator, pos[ANY])

6. MATCHFIX(operator, match, argpos[ANY], pos[ANY])

The defaults have been provided so that a user who does not wish to concern himself with parts of speech or binding powers may simply omit those arguments to the extension functions. Thus the following are all equivalent:

```
PREFIX("DDX",180,ANY,ANY)$
PREFIX("DDX",180)$
PREFIX("DDX")$
```

It is also possible to remove the syntax properties of an operator by using the functions REMOVE or KILL. Specifically, REMOVE("DDX",OP) or KILL("DDX") will return DDX to operand status; but in the second case all the other properties of DDX will also be removed.

PREFIX(*op*) A PREFIX operator is one which signifies a function of one argument, which argument immediately follows an occurrence of the operator. PREFIX("x") is a syntax extension function to declare *x* to be a PREFIX operator.

`POSTFIX` (*op*) [*Function*]

POSTFIX operators like the PREFIX variety denote functions of a single argument, but in this case the argument immediately precedes an occurrence of the operator in the input string, e.g. `3!` . The POSTFIX("x") function is a syntax extension function to declare `x` to be a POSTFIX operator.

`INFIX` (*op*) [*Function*]

INFIX operators are used to denote functions of two arguments, one given before the operator and one after, e.g. `A**2` . INFIX(fun) is a syntax extension function to DECLARE *fun* to be an INFIX operator.

`MATCHFIX` (*op*) [*Function*]

MATCHFIX operators are used to denote functions of any number of arguments which are passed to the function as a list. The arguments occur between the main operator and its matching delimiter. MATCHFIX(fun,...) is a syntax extension function which declares *fun* to be a MATCHFIX operator.

`NOFIX` (*op*) [*Function*]

NOFIX operators are used to denote functions of no arguments. The mere presence of such an operator in a command will cause the corresponding function to be evaluated. For example, when one types `exit;` to exit from a **Maxima** break, `exit` is behaving similar to a NOFIX operator. The function NOFIX("x") is a syntax extension function which declares `x` to be a NOFIX operator.

`NARY` (*op*) [*Function*]

An NARY operator is used to denote a function of any number of arguments, each of which is separated by an occurrence of the operator, e.g. `A+B` or `A+B+C`. NARY(fun) is a syntax extension function to declare

fun to be an NARY operator. Functions may be DECLAREd to be NARY. DECLARE(J,NARY); tells the simplifier to simplify J(J(A,B),J(C,D)) to J(A, B, C, D).

See section 10.1.3 [Declarations], page 148.

10.1.2 Defining Macros

The **Maxima** macro facility allows the user to define forms which are similar in appearance to **Maxima** functions but whose evaluation and simplification can be more carefully controlled. This high degree of control provides users with many powerful capabilities (the ability to write definitions of operators which act as extended control structures, which implicitly quote certain of their arguments, or which establish special environments for the evaluation of their arguments, to name just a few). Additionally, macros are highly efficient from a compilation standpoint– references to macros can be made to yield extremely good code when compiled.

A macro is a definition of a transformation between some syntactic construct typed in by the user and a form which can be interpreted by the **Maxima** evaluator. Suppose a user had several lists (X, Y, and Z) which he often wanted to print out the values of in the following way:

```
FOR I THRU LENGTH(X) DO PRINT('X[I]=X[I]);
FOR I THRU LENGTH(Y) DO PRINT('Y[I]=Y[I]);
\dots etc
```

This could get tedious to type. He might, for instance, want to define a **Maxima** operator which did most the work for him, so that he could type

```
SHOWME(X);
```

and have the same thing happen as if he had typed out the FOR loop. Without using macros, trying to write a definition for SHOWME which allows this syntax is tedious at best, and in any case would require the user to type

```
SHOWME('X);
```

in order to get a hold of the name of the form being displayed. (Try it yourself as an exercise.) Let's see how you would do this as a macro. It's quite simple, really. SHOWME will be an operator with just one thing in its argument position, so we'll want to write

```
SHOWME(LISTNAME) ::= \dots something \dots
```

The **::=** is used instead of := to indicate that what follows is a macro definition, rather than an ordinary functional definition. Proceeding, we must next write a definition. The definition should do something to create the form which we really want to evaluate, returning that form as a value. In this case, we shall want

```
FOR I THRU LENGTH(LISTNAME) DO PRINT('LISTNAME[I]=LISTNAME[I])
```

to get returned by SHOWME. Note that we don't want to do this action – we only want to return the new

("expanded") form – the **Maxima** interpreter will take care of any evaluation later. One way to do this might be to use FUNMAKE and SUBSTPART but that can be pretty hard to do, especially with a form like FOR. An easier way is to use the BUILDQ function. Using BUILDQ, the definition would look like:

```
SHOWME(LISTNAME)::=
      BUILDQ([LISTNAME],
             FOR I THRU LENGTH(LISTNAME)
                 DO PRINT('LISTNAME[I]=LISTNAME[I]))$
```

What happens when the **Maxima** evaluator encounters a macro? First it calls the macro definition on the appropriate parts of the form – for example, in the case of SHOWME(X), the SHOWME macro would be called with LISTNAME bound to X. Note that no evaluation of arguments is done prior to application of the macro – that will come later. The SHOWME definition will then create a new form which looks like

```
FOR I THROUGH LENGTH(X) DO PRINT('X[I]=X[I])
```

and return that to the evaluator. The evaluator will then take the new form, and evaluate that in place of the original SHOWME form, finally returning the value of this second evaluation as the value of the SHOWME. So we might actually type:

```
X:['A,'B]$

SHOWME(X)$

X  = A
 1

X  = B
 2
```

A symbol can have either a macro property or a function property but not both at the same time. Defining an atom with ::= will remove the atom's function properties and vice versa.

10.1.2.1 Useful Functions and Variables for Macros

MACROS [*Variable*]

is a variable (similar to FUNCTIONS) which lists at any time all macros defined in the current environment. It is one of the variables which can be found in INFOLISTS.

DISPFUN(*macro*) and GRIND(*macro*) will display either the function property or the macro property if either exist. STRINGOUT(FUNCTIONS); may be used to STRINGOUT all functions and macros in the current environment.

MACROEXPANSION [*variable, default: FALSE*]

Controls advanced features which affect the efficiency of macros. Possible settings:

FALSE Macros expand normally each time they are called.

EXPAND The first time a particular call is evaluated, the expansion is remembered internally, so that it doesn't have to be recomputed on subsequent calls making subsequent calls faster. The macro call still GRINDs and DISPLAYs normally, however extra memory is required to remember all of the expansions.

DISPLACE The first time a particular call is evaluated, the expansion is substituted for the call. This requires slightly less storage than when MACROEXPANSION is set to EXPAND and is just as fast, but has the disadvantage that the original macro call is no longer remembered and hence the expansion will be seen if DISPLAY or GRIND is called. Macros always expand and displace at the time of translation when calls to them are translated. See section 13.2 [Translation], page 186.

`MACROEXPAND` (*macro*) [*Special Form*]

expands *macro* repeatedly until it is no longer a macro call. The final expansion is returned without doing the automatic evaluation.

`MACROEXPAND1` (*macro*) [*Special Form*]

expands *macro* exactly once if *macro* is a macro call without doing the automatic evaluation. If *macro* is not a macro call, *macro* is returned, again unevaluated. There is a demo in the file: '`demo/macex.dem`'.

10.1.2.2 More Macro Examples

```
PUSH(VALUE,STACKNAME)::=BUILDQ([VALUE,STACKNAME],
                          STACKNAME:CONS(VALUE,STACKNAME))$

POP(STACKNAME)::=BUILDQ([STACKNAME],
                    BLOCK([TEMP:FIRST(STACKNAME)],
                          STACKNAME:REST(STACKNAME),TEMP))$

A:[];            => []
PUSH('FOO,A);    => [FOO]
PUSH('BAR,A);    => [BAR, FOO]
A;               => [BAR, FOO]
POP(A);          => BAR
A;               => [FOO]
POP(A);          => FOO
A;               => []
```

There is a demo in the file: '`demo/macro.dem`'. There are some usage notes in the file: '`demo/macro.usg`'.

`BUILDQ` (*variable-list*, *expression*) [*Special Form*]

is a **Maxima** function for constructing pieces of code to be executed. It will be very useful in conjunction with macros, but may have other applications as well. BUILDQ is a generalized substitution function used to create **Maxima** forms. (Its primary use is in macro bodies.)

variable-list is similar to the variable list for a BLOCK (i.e. it can contain both atomic variables and assignment-forms). The right hand sides of any assignment-forms in the *variable-list* are evaluated left to right and the resulting variable bindings are substituted in parallel into the *expression* (actually, the process is more complicated than what SUBST would provide, as we shall see farther down). The *expression* can be any **Maxima** expression (including a nested BUILDQ). The new expression is returned as is, not evaluated. For example:

```
S:A+B$

BUILDQ([S,A:B*C,FUN:BAR],
       S^2+G(A,'A)+FUN(S) );

                  2
=>      (A + B)  + G(B C, '(B C)) + BAR(A + B)
```

The parallel nature of the substitution can probably be most easily seen in the following example:

```
BUILDQ([A:'B,B:'A], SIN(A)+COS(B));

=> SIN(B) + COS(A)
```

BUILDQ also recognizes specially the keyword SPLICE when it is used in a functional position within the expression. If SPLICE is used with only one argument and that argument is the name of one of the variables being substituted for, then the value of the variable is spliced into the expression instead of being substituted. The spliced variable must evaluate to a list.

```
L:[A,B,C]$

BUILDQ([L],F(L,SPLICE(L)));

F([A, B, C], A, B, C)
```

There is a demo in the file: '`demo/buildq.dem`'. There are some usage notes in the file: '`demo/buildq.usg`'.

10.1.3 Declarations

Maxima has built-in properties which are handled by the database. These are called features. One can do `DECLARE(N,INTEGER)` to declare that *N* is an integer. One can also DECLARE one's own features by e.g. `DECLARE(INCREASING,FEATURE);` which will then allow one to say `DECLARE(F,INCREASING);`. One can then check if F is INCREASING by using the predicate FEATUREP via `FEATUREP(F,INCREASING)`. There is an infolist FEATURES which is a list of known FEATURES.

Maxima currently recognizes and uses the following features of objects: EVEN, ODD, INTEGER, NONINTEGER, RATIONAL, IRRATIONAL, REAL, IMAGINARY, and COMPLEX. The useful features of functions include: INCREASING, DECREASING, ODDFUN (odd function), EVENFUN (even function), POSFUN, COMMUTATIVE (or SYMMETRIC), ANTISYMMETRIC, ANALYTIC, LAS-

SOCIATIVE and RASSOCIATIVE. DECLARE(F,INCREASING) is in all respects equivalent to ASSUME(KIND(F,INCREASING)). The *ai* and *fi* may also be lists of objects or features. The command FEATUREP(object,feature) may be used to determine if *object* has been DECLAREd to have *feature*. COMMUTATIVE, LASSOCIATIVE, RASSOCIATIVE, SYMMETRIC, and ANTISYMMETRIC. System features may be checked with STATUS(FEATURE,...);

DECLARE (*a1, f1, a2, f2, ...*) [*Special Form*]

gives the atom *ai* the flag *fi*. The *ai*'s and *fi*'s may also be lists of atoms and flags respectively in which case each of the atoms gets all of the properties. The possible flags and their meanings are:

CONSTANT makes *ai* a constant as is %PI.

MAINVAR makes *ai* a **MAINVAR**. The ordering scale for atoms is: numbers < constants (e.g. %E, %PI) < scalars < other variables < mainvars. E.g. compare EXPAND((X+Y)**4); with (DECLARE(X,MAINVAR), EXPAND((X+Y)**4));. (Note: Care should be taken if you elect to use the above feature. E.g. if you subtract an expression in which X is a MAINVAR from one in which X isn't a MAINVAR, resimplification e.g. with EV(expression,SIMP) may be necessary if cancellation is to occur. Also, if you SAVE an expression in which X is a MAINVAR, you probably should also SAVE X.) See section 2.5 [Ordering of the Display], page 28.

SCALAR makes *ai* a scalar.

NONSCALAR makes *ai* behave as does a list or matrix with respect to the dot operator . .

NOUN makes the function *ai* a noun so that it won't be evaluated automatically.

EVFUN makes *ai* known to the EV function so that it will get applied if its name is mentioned. Initial evfuns are FACTOR, TRIGEXPAND, TRIGREDUCE, BFLOAT, RATSIMP, RATEXPAND, and RADCAN.

EVFLAG makes *ai* known to the EV function so that it will be bound to TRUE during the execution of EV if it is mentioned. Initial evflags are: FLOAT, PRED, SIMP, NUMER, DETOUT, EXPONENTIALIZE, DEMOIVRE, KEEPFLOAT, LISTARITH, TRIGEXPAND, SIMPSUM, ALGEBRAIC, RATALGDENOM, FACTORFLAG, %EMODE, LOGARC, LOGNUMER, RADEXPAND, RATSIMPEXPONS, RATMX, RATFAC, INFEVAL, %ENUMER, PROGRAMMODE, LOGNEGINT, LOGABS, LETRAT, HALFANGLES, EXPTISOLATE, ISOLATE_WRT_TIMES, SUMEXPAND, CAUCHYSUM, NUMER_PBRANCH, M1PBRANCH, DOTSCRULES, and LOGEXPAND. See section 4.1 [Evaluation], page 47.

BINDTEST causes *ai* to signal an error if it ever is used in a computation unbound. DECLARE([var1, var2, ...], BINDTEST) causes **Maxima** to give an error message whenever any of the *vari* occur unbound in a computation.

ALPHABETIC adds to **Maxima**'s alphabet (initially A-Z,% and _). Thus, DECLARE(" ",ALPHABETIC) enables NEW VALUE to be used as a name.

ADDITIVE If DECLARE(F,ADDITIVE) has been executed, then:

1. If F is univariate, whenever the simplifier encounters F applied to a sum, F will be distributed over that sum. I.e. F(Y+X); will simplify to F(Y)+F(X).

2. If F is a function of 2 or more arguments, additivity is defined as additivity in the first argument to F, as in the case of SUM or INTEGRATE, i.e. F(H(X)+G(X),X); will simplify to F(H(X),X)+F(G(X),X).

This simplification does not occur when F is applied to expressions of the form SUM(X[I], I, lo, hi).

SYMMETRIC If DECLARE(H,SYMMETRIC); is done, this tells the simplifier that H is a symmetric function. E.g. H(X, Z, Y) will simplify to H(X, Y, Z). This is the same as COMMUTATIVE.

ANTISYMMETRIC If DECLARE(H,ANTISYMMETRIC); is done, this tells the simplifier that H is antisymmetric. E.g. H(X,Z,Y) will simplify to -H(X, Y, Z). That is, it will give (-1)**n times the result given by SYMMETRIC or COMMUTATIVE, where n is the number of interchanges of two arguments necessary to convert it to that form.

COMMUTATIVE If DECLARE(H,COMMUTATIVE); is done, this tells the simplifier that H is a commutative function. E.g. H(X, Z, Y) will simplify to H(X, Y, Z). This is the same as SYMMETRIC.

LASSOCIATIVE DECLARE(G,LASSOCIATIVE); tells simplifier that G is left-associative. E.g. G(G(A,B),G(C,D)) will simplify to G(G(G(A,B),C),D).

RASSOCIATIVE DECLARE(G,RASSOCIATIVE); tells the simplifier that G is right-associative. E.g. G(G(A,B),G(C,D)) will simplify to G(A,G(B,G(C,D))).

POSFUN POSitive FUNction, e.g. DECLARE(F,POSFUN); IS(F(X)>0); -> TRUE.

LINEAR For univariate F so declared, expansion F(X+Y) -> F(X)+F(Y), F(A*X) -> A*F(X) takes place where A is a "constant.' For functions F of 2 or more arguments, "linearity" is defined to be as in the case of SUM or INTEGRATE, i.e. F(A*X+B,X) -> A*F(X,X)+B*F(1,X) for A,B FREEOF X. (LINEAR is just ADDITIVE + OUTATIVE.)

MULTIPLICATIVE If DECLARE(F,MULTIPLICATIVE) has been executed, then:

1. If *F* is univariate, whenever the simplifier encounters *F* applied to a product, F will be distributed over that product, i.e. intoF(X*Y); will simplify to F(X)*F(Y).
2. If *F* is a function of 2 or more arguments, multiplicativity is defined as multiplicativity in the first argument to F, i.e. F(G(X)*H(X),X); will simplify to F(G(X),X)*F(H(X),X). This simplification does not occur when F is applied to expressions of the form PRODUCT(X[I],I,lower-limit,upper-limit).

OUTATIVE If DECLARE(F,OUTATIVE); has been executed, then:

1. If F is univariate, whenever the simplifier encounters F applied to a product, that product will be partitioned into factors that are constant and factors that are not and the constant factors will be pulled out. I.e. F(A*X); will simplify to A*F(X) where A is a constant. Non-atomic constant factors will not be pulled out.
2. If F is a function of 2 or more arguments, outativity is defined as in the case of SUM or INTEGRATE, i.e. F(A*G(X),X); will simplify to A*F(G(X),X) for A FREEOF X. Initially, SUM, INTEGRATE, and LIMIT are declared to be OUTATIVE.

10.1.4 Assumptions

ASSUME (*pred1, pred2, ...*) [*Special Form*]

First checks the specified predicates for redundancy and consistency with the current database. If the predicates are consistent and non-redundant, they are added to the database; if inconsistent or redundant, no action is taken. ASSUME returns a list whose entries are the predicates added to the database and the atoms REDUNDANT or INCONSISTENT where applicable.

ASSUME_POS [*variable, default: FALSE*]

When using INTEGRATE, etc. one often introduces parameters which are real and positive or one's calculations can often be constructed so that this is true. There is a switch ASSUME_POS default: FALSE such that if set to TRUE, **Maxima** will assume one's parameters are positive. The intention here is to cut down on the number of questions **Maxima** needs to ask. Obviously, ASSUME information or any contextual information present will take precedence. The user can control what is considered to be a parameter for this purpose. Parameters by default are those which satisfy SYMBOLP(x) or SUBVARP(x). The user can change this by setting the option ASSUME_POS_PRED default: [FALSE] to the name of a predicate function of one argument. E.g. if you want only symbols to be parameters, you can do

```
ASSUME_POS:TRUE$
ASSUME_POS_PRED:'SYMBOLP$
SIGN(A); -> POS,
SIGN(A[1]); -> PNZ.
```

ASSUME_POS_PRED [*variable, default: FALSE*]

may be set to one argument to control what will be considered a parameter for the assumptions that INTEGRATE will make. See also ASSUME and ASSUME_POS.

ASSUMESCALAR [*variable, default: TRUE*]

helps govern whether expressions *exp* for which NONSCALARP(exp) gives FALSE are assumed to behave like scalars for certain transformations as follows: Let *exp* represent any non-list/non-matrix, and [1,2,3] any list or matrix. exp . [1,2,3]; will give [exp, 2*exp, 3*exp] if ASSUMESCALAR is TRUE or SCALARP(exp) is TRUE or CONSTANTP(exp) is TRUE. If ASSUMESCALAR is TRUE, such expressions will behave like scalars only for the commutative operators, but not for . If ASSUMESCALAR is FALSE, such expressions will behave like non-scalars. If ASSUMESCALAR is ALL, such expressions will behave like scalars for all the operators listed above.

10.1.5 Contexts

CONTEXTS [*variable, default: [INITIAL, GLOBAL]*]

is a list of the contexts which currently exist, including the currently active context. The context mechanism makes it possible for a user to bind together and name a selected portion of his database, called a context.

Once this is done, the user can have **Maxima** assume or forget large numbers of facts merely by activating or deactivating their context. Any symbolic atom can be a context, and the facts contained in that context will be retained in storage until the user destroys them individually by using FORGET or destroys them as a whole by using KILL to destroy the context to which they belong.

Contexts exist in a formal hierarchy, with the root always being the context GLOBAL, which contains information about **Maxima** that some functions need. When in a given context, all the facts in that context are active (meaning that they are used in deductions and retrievals) as are all the facts in any context which is an inferior of that context. When a fresh **Maxima** is started up, the user is in a context called INITIAL, which has GLOBAL as a subcontext. The functions which deal with contexts are: FACTS, NEWCONTEXT, SUPCONTEXT, KILLCONTEXT, ACTIVATE, DEACTIVATE, ASSUME, and FORGET.

`CONTEXT` [*variable, default: INITIAL*]

Whenever a user assumes a new fact, it is placed in the context named as the current value of the variable CONTEXT. Similarly, FORGET references the current value of CONTEXT. To change contexts, simply bind CONTEXT to the desired context. If the specified context does not exist it will be created by an invisible call to NEWCONTEXT. The context specified by the value of CONTEXT is automatically activated.

`FACTS` (*context*) [*Function*]

If *context* is the name of a context then FACTS returns a list of the facts in the specified context. If no argument is given, it lists the current context. If *context* is not the name of a context then it returns a list of the facts known about *context* in the current context. Facts that are active, but in a different context, are not listed.

`NEWCONTEXT` (*context*) [*Function*]

creates a new (empty) context, called *context*, which has GLOBAL as its only subcontext. The new context created will become the currently active context.

`SUPCONTEXT` (*name, context*) [*Special Form*]

will create a new context (called *name*) whose subcontext is *context*. If context is not specified, the current context will be assumed. If it is specified, context must exist.

`ACTIVATE` (*context1, context2, ...*) [*Function*]

causes the specified contexts *contexti* to be activated. The facts in these contexts are used in making deductions and retrieving information. The facts in these contexts are not listed when `FACTS()`; is done. The variable ACTIVECONTEXTS is the list of contexts which are active by way of the ACTIVATE function.

`ACTIVECONTEXTS` [*Variable*]

is a list of the contexts which are active by way of the ACTIVATE function, as opposed to being active because they are subcontexts of the current context.

DEACTIVATE (*context1, context2, . . .*) [*Function*]

causes the specified contexts *contexti* to be deactivated.

KILLCONTEXT (*context1, context2, . . . , contextn*) [*Function*]

kills the specified contexts. If one of them is the current context, the new current context will become the first available subcontext of the current context which has not been killed. If the first available unkilled context is GLOBAL then INITIAL is used instead. If the INITIAL context is killed, a new INITIAL is created, which is empty of facts. KILLCONTEXT doesn't allow the user to kill a context which is currently active, either because it is a subcontext of the current context, or by use of the function ACTIVATE.

10.1.6 Properties

PROPERTIES (*atom*) [*Special Form*]

will yield a list showing the names of all the properties associated with the atom *atom*.

PROPVARS (*prop*) [*Special Form*]

yields a list of those atoms on the PROPS list which have the property indicated by *prop*. Thus PROPVARS(ATVALUE) will yield a list of atoms which have ATVALUES.

PROPS [*Variable*]

atoms which have any property other than those explicitly mentioned in INFOLISTS, such as ATVALUEs, MATCHDECLAREs, etc. as well as properties specified in the DECLARE function.

PUT (*atom, prop, ind*) [*Function*]

associates with the atom *atom* the property *prop* with the indicator *ind*. This enables the user to give an atom any arbitrary property.

QPUT (*atom, prop, ind*) [*Special Form*]

is similar to PUT but it doesn't have its arguments evaluated.

GET (*atom, ind*) [*Function*]

retrieves the user property indicated by *ind* associated with atom *atom* or returns FALSE if a doesn't have property *ind*.

```
(C1) PUT(\%E,'TRANSCENDENTAL,'TYPE);
(D1)                            TRANSCENDENTAL
(C2) PUT(\%PI,'TRANSCENDENTAL,'TYPE)$
(C3) PUT(\%I,'ALGEBRAIC,'TYPE)$
(C4) TYPEOF(EXP) := BLOCK([Q],
                       IF NUMBERP(EXP)
                       THEN RETURN('ALGEBRAIC),
                       IF NOT ATOM(EXP)
                       THEN RETURN(MAPLIST('TYPEOF, EXP)),
                       Q : GET(EXP, 'TYPE),
                       IF Q=FALSE
                       THEN ERRCATCH(ERROR(EXP,"is not nu-
meric.")) ELSE Q)$
(C5) TYPEOF(2*\%E+X*\%PI);
X is not numeric.
(D5)         [[TRANSCENDENTAL, []], [ALGEBRAIC, TRANSCENDENTAL]]
(C6) TYPEOF(2*\%E+\%PI);
(D6)            [TRANSCENDENTAL, [ALGEBRAIC, TRANSCENDENTAL]]
```

PRINTPROPS (*atom*, *ind*) [*Special Form*]

will display the property with the indicator *ind* associated with the atom *atom*. *atom* may also be a list of atoms or the atom ALL in which case all of the atoms with the given property will be used. For example, PRINTPROPS([F,G],ATVALUE). PRINTPROPS is for properties that cannot otherwise be displayed, i.e. for ATVALUE, ATOMGRAD, GRADEF, and MATCHDECLARE.

10.1.7 Rules

User defined pattern matching and simplification rules can be set up by TELLSIMP, TELLSIMPAFTER, DEFMATCH, or, DEFRULE.

10.1.7.1 Defining Simplification Rules

TELLSIMPAFTER (*pattern*, *replacement*) [*Special Form*]

defines a replacement for pattern which the **Maxima** simplifier uses after it applies the built-in simplification rules. The pattern may be anything but a single variable or a number.

TELLSIMP (*pattern*, *replacement*) [*Special Form*]

is similar to TELLSIMPAFTER but places new information before old so that it is applied before the built-in simplification rules. TELLSIMP is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier knows something about the expression, but what it returns is not to your liking. If the simplifier knows something about the main operator of the expression, but is simply not doing enough for you, you probably want to use TELLSIMPAFTER. The pattern may not be a sum, product, single variable, or number. RULES is a list of names having simplification rules added to them by DEFRULE, DEFMATCH, TELLSIMP, or TELLSIMPAFTER.

10.1.7.2 Substitution Rules

LET (*prod, repl, predname, arg1, arg2, ...*) [*Special Form*]

defines a substitution rule for LETSIMP such that *prod* gets replaced by *repl*. *prod* is a product of positive or negative powers of the following types of terms:

1. Atoms which LETSIMP will search for literally unless previous to calling LETSIMP the MATCHDECLARE function is used to associate a predicate with the atom. In this case LETSIMP will match the atom to any term of a product satisfying the predicate.
2. Kernels such as SIN(X), N!, F(X,Y), etc. As with atoms above LETSIMP will look for a literal match unless MATCHDECLARE is used to associate a predicate with the argument of the kernel.

A term to a positive power will only match a term having at least that power in the expression being LETSIMPed. A term to a negative power on the other hand will only match a term with a power at least as negative. In the case of negative powers in product the switch LETRAT must be set to TRUE. If a predicate is included in the LET function followed by a list of arguments, a tentative match (i.e. one that would be accepted if the predicate were omitted) will be accepted only if predname(arg1',...,argn') evaluates to TRUE where *argi'* is the value matched to argi. The *argi* may be the name of any atom or the argument of any kernel appearing in *prod*. *repl* may be any rational expression. If any of the atoms or arguments from *prod* appear in *repl* the appropriate substitutions will be made.

LETRAT [*variable, default: FALSE*]

when FALSE, LETSIMP will simplify the numerator and denominator of *exp* independently and return the result. Substitutions such as N!/N goes to (N-1)! will fail. To handle such situations LETRAT should be set to TRUE, then the numerator, denominator, and their quotient will be simplified in that order.

These substitution functions allow you to work with several rule packages at once. Each rule package can contain any number of LETed rules and is referred to by a user supplied name. To insert a rule into the rule package *name*, do LET([prod, repl, pred, arg1, ...], name). To apply the rules in rule package *name*, do LETSIMP(exp, name). The function LETSIMP(exp, name1, name2, ...) is equivalent to doing LETSIMP(exp, name1) followed by LETSIMP(%, name2) etc.

LETSIMP (*exp*) [*Special Form*]

will continually apply the substitution rules previously defined by the function LET until no further change is made to *exp*.

LETSIMP(*exp, rule_pkg_name*) will cause the rule package *rule_pkg_name* to be used for that LETSIMP command only, i.e. the value of CURRENT_LET_RULE_PACKAGE is not changed.

LETRULES () [*Special Form*]

displays the rules in the current rule package (initially DEFAULT_LET_RULE_PACKAGE). × LETRULES*(name)* displays the rules in the rule package named *name*. The current rule package is the value of CURRENT_LET_RULE_PACKAGE.

CURRENT_LET_RULE_PACKAGE [*variable, default: DEFAULT_LET_RULE_PACKAGE*]

the name of the rule package that is presently being used. The user may reset this variable to the name of any rule package previously defined via the LET command. Whenever any of the functions comprising the let package are called with no package name the value of CURRENT_LET_RULE_PACKAGE is used. If a call such as LETSIMP(exp,rule_pkg_name); is made, the rule package *rule_pkg_name* is used for that LETSIMP command only, i.e. the value of CURRENT_LET_RULE_PACKAGE is not changed.

DEFAULT_LET_RULE_PACKAGE [*Variable*]

the name of the rule package used when one is not explicitly set by the user with LET or by changing the value of CURRENT_LET_RULE_PACKAGE.

LET_RULE_PACKAGES [*variable, default: DEFAULT_LET_RULE_PACKAGE*]

The value of LET_RULE_PACKAGES is a list of all the user-defined let rule packages plus the special package DEFAULT_LET_RULE_PACKAGE. DEFAULT_LET_RULE_PACKAGE is the name of the rule package used when one is not explicitly set by the user.

10.1.7.3 Pattern Matching Rules

DEFMATCH (*progname, pattern, parm1, ..., parmn*) [*Special Form*]

creates a function of *n*+1 arguments with the name *progname* which tests an expression to see if it can match a particular *pattern*. The pattern is some expression containing pattern variables and parameters. The *parms* are given explicitly as arguments to DEFMATCH, while the pattern variables (if supplied) were given implicitly in a previous MATCHDECLARE function. The first argument to the created function *progname* is an expression to be matched against the *pattern* and the other *n* arguments are the actual variables occurring in the expression which are to take the place of dummy variables occurring in the *pattern*. Thus the *parms* in the DEFMATCH are like the dummy arguments to the SUBROUTINE statement in FORTRAN. When the function is called, the actual arguments are substituted. For example:

```
(C1)  NONZEROANDFREEOF(X,E):=  IF E#0 AND FREEOF(X,E)
            THEN \code{TRUE} else \code{FALSE}$
/*IS(E#0 AND FREEOF(X,E)) is an  equivalent function definition */
(C2)  MATCHDECLARE(A,NONZEROANDFREEOF(X),B,FREEOF(X))$
(C3)  DEFMATCH(LINEAR,A*X+B,X)$
```

This has caused the function LINEAR(exp, var1) to be defined. It tests *exp* to see if it is of the form A*var1+B, where A and B do not contain *var1*, and A is not zero. If the match is successful, DEFMATCHed functions return a list of equations whose left sides are the pattern variables and *parms*, and whose right sides are the expressions which the pattern variables and parameters matched. The pattern variables, but not the parameters, are set to the matched expressions. If the match fails, the function returns FALSE. Thus LINEAR(3*Z+(Y+1)*Z+Y**2,Z) would return [B=Y**2, A=Y+4, X=Z].

Any variables not declared as pattern variables in MATCHDECLARE or as parameters in DEFMATCH which occur in pattern will match only themselves so that if the third argument to the DEFMATCH in (C3)

had been omitted, then LINEAR would only match expressions linear in X, not in any other variable. A pattern which contains no parameters or pattern variables returns TRUE if the match succeeds.

MATCHDECLARE (*pattern, pred, ...*) [*Special Form*]

associates a predicate *pred* with a pattern variable *pattern*, so that the variable will only match expressions for which the predicate is not FALSE. The matching is accomplished by one of the functions described below. For example after MATCHDECLARE(Q,FREEOF(X,%E)) is executed, *Q* will match any expression not containing X or %E. If the match succeeds, then the variable is set to the matched expression. The predicate (in this case FREEOF) is written without the last argument which should be the one against which the pattern variable is to be tested. Note that *pattern* and the arguments to the predicate *pred* are evaluated at the time the match is performed.

The odd numbered argument may also be a list of pattern variables all of which are to have the associated predicate. Any even number of arguments may be given. For pattern matching, predicates refer to functions which are either FALSE, or not FALSE, i.e. any non FALSE value acts like TRUE. MATCHDECLARE(var,TRUE) will permit *var* to match any expression.

DEFRULE (*rulename, pattern, replacement*) [*Special Form*]

defines and names a replacement rule for the given pattern. If the rule named *rulename* is applied to an expression (by one of the APPLY functions below), every subexpression matching the pattern will be replaced by the replacement. All variables in the replacement which have been assigned values by the pattern match are assigned those values in the replacement which is then simplified. The rules themselves can be treated as functions which will transform an expression by one operation of the pattern match and replacement. If the pattern fails, the original expression is returned.

DISPRULE (*rulename1, rulename2, ...*) [*Special Form*]

will display rules with the names *rulenamei*, as were given by DEFRULE, TELLSIMP, or TELLSIMPAFTER, or a pattern defined by DEFMATCH. For example, the first rule modifying SIN will be called SINRULE1. DISPRULE(ALL); will display all rules.

REMRULE (*fun, rule*) [*Function*]

will remove a rule with the name *rule* from the function *fun*, which was placed there by DEFRULE, DEFMATCH, TELLSIMP, or TELLSIMPAFTER. If rulename is ALL, then all rules will be removed.

10.1.7.4 Applying Rules

APPLY1 (*exp, rule1, ..., rulen*) [*Special Form*]

repeatedly applies the first rule to *exp* until it fails, then repeatedly applies the same rule to all subexpressions of *exp*, left-to-right, until the first rule has failed on all subexpressions. Call the result of transforming *exp* in this manner *exp'*. Then the second rule is applied in the same fashion starting at the top of *exp'*. When the final rule fails on the final subexpression, the application is finished.

APPLY2 (*exp, rule1, . . . , rulen*) [*Special Form*]

differs from APPLY1 in that if the first rule fails on a given subexpression, then the second rule is repeatedly applied, etc. Only if they all fail on a given subexpression is the whole set of rules repeatedly applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule.

APPLYB1 (*exp, rule1, . . . , rulen*) [*Special Form*]

is similar to APPLY1 but works from the bottom-up instead of from the top-down. That is, it processes the smallest subexpression of exp, then the next smallest, etc.

MAXAPPLYDEPTH [*variable, default: 10000*]

the maximum depth to which APPLY1 and APPLY2 will delve.

MAXAPPLYHEIGHT [*variable, default: 10000*]

the maximum height to which APPLYB1 will reach before giving up.

10.2 Querying the Database

FEATUREP (*atom, feat*) [*Function*]

attempts to determine whether the object *atom* has the feature *feat* on the basis of the facts in the current database. If so, it returns TRUE, otherwise FALSE.

```
(C1) DECLARE(J,EVEN)$
(C2) FEATUREP(J,INTEGER);
(D2)                                TRUE
```

IS (*exp*) [*Special Form*]

attempts to determine whether *exp* (which must evaluate to a predicate) is provable from the facts in the current database. IS returns TRUE if the predicate is TRUE for all values of its variables consistent with the database, and returns FALSE if it is false for all such values. Otherwise, its action depends on the setting of the switch PREDERROR default: TRUE. IS errors out if the value of PREDERROR is TRUE, and returns UNKNOWN if PREDERROR is FALSE.

PREDERROR [*variable, default: TRUE*]

If TRUE, an error message is signalled whenever the predicate of an IF statement or an IS function fails to evaluate to either TRUE or FALSE. If FALSE, UNKNOWN is returned instead in this case. The PREDERROR : FALSE mode is not supported in translated code.

EQUAL (*expr1, expr2*) [*Function*]

when used with an IS, returns TRUE (or FALSE) if and only if expr1 and expr2 are equal (or not equal) for all possible values of their variables (as determined by RATSIMP). Thus IS(EQUAL((X+1)**2,X**2+2*X+1)) returns TRUE, whereas if X is unbound × IS((X+1)**2=X**2+2*X+1) returns FALSE. Note also that IS(RAT(0)=0) gives FALSE but

IS(EQUAL(RAT(0),0)) gives TRUE. If a determination can't be made with EQUAL then a simplified but equivalent form is returned whereas = always causes either TRUE or FALSE to be returned. All variables occurring in *exp* are presumed to be real valued. EV(exp,PRED) is equivalent to IS(exp).

```
(C1) IS(X**2 >= 2*X-1);
(D1)                                TRUE
(C2) ASSUME(A>1);
(D2)                                DONE
(C3) IS(LOG(LOG(A+1)+1)>0 AND A^2+1>2*A);
(D3)                                TRUE
```

ZEROEQUIV (*exp, var*) [*Function*]

tests whether the expression *exp* in the variable *var* is equivalent to zero. It returns either TRUE, FALSE, or DONTKNOW. For example ZEROEQUIV(SIN(2*X) - 2*SIN(X)*COS(X),X) returns TRUE and ZEROEQUIV(%E**X+X,X) returns FALSE. On the other hand, ZEROEQUIV(LOG(A*B) - LOG(A) - LOG(B),A) will return DONTKNOW because of the presence of an extra parameter. The restrictions are:

1. Do not use functions that **Maxima** does not know how to differentiate and evaluate.
2. If the expression has poles on the real line, there may be errors in the result (but this is unlikely to occur).
3. If the expression contains functions which are not solutions to first order differential equations (e.g. Bessel functions) there may be incorrect results.
4. The algorithm uses evaluation at randomly chosen points for carefully selected subexpressions. This is always a somewhat hazardous business, although the algorithm tries to minimize the potential for error.

SIGN (*exp*) [*Function*]

attempts to determine the sign of its specified expression on the basis of the facts in the current database. It returns one of the following answers: POS (positive), NEG (negative), ZERO, PZ (positive or zero), NZ (negative or zero), PN (positive or negative), or PNZ (positive, negative, or zero, i.e. nothing known).

ASKINTEGER (*exp, optional-arg*) [*Function*]

If *exp* is any valid **Maxima** expression and *optional-arg* is EVEN, ODD, INTEGER and defaults to INTEGER if not given. This function attempts to determine from the database whether *exp* is EVEN, ODD or just an INTEGER. It will ask the user if it cannot deduce it on its own, and attempt to install the information in the database if possible.

ASKSIGN (*exp*) [*Function*]

first attempts to determine whether the specified expression *exp* is positive, negative, or zero. If it cannot, it asks the user the necessary questions to complete its deduction. The user's answer is recorded in the database for the duration of the current computation (one C-line). The value of ASKSIGN is one of POS, NEG, or ZERO.

ASKEXP [*Variable*]

contains the expression upon which ASKSIGN is called. A user may enter a **Maxima** break and inspect this expression in order to answer questions asked by ASKSIGN.

10.3 Deleting From the Database

FORGET (*pred1, pred2, ...*) [*Special Form*]

removes relations established by ASSUME. The predicates may be expressions equivalent to (but not necessarily identical to) those previously ASSUMEd. FORGET(list) is also a legal form.

KILL (*arg1, arg2, ...*) [*Special Form*]

eliminates its arguments from the **Maxima** system. If *argi* is a variable, a single array element, a function, or an array, the designated item with all of its properties is removed from core. If *argi* is

LABELS then all input, intermediate, and output lines to date (but not other named items) are eliminated.

CLABELS then only input lines will be eliminated.

ELABELS then only intermediate E-lines will be eliminated.

DLABELS only the output lines will be eliminated.

n a number, then the last *n* lines (i.e. the lines with the last *n* line numbers) are deleted. If argi is of the form [m,n] then all lines with numbers between *m* and *n* inclusive are KILLed.

VALUES will kill all properties associated with every item on the VALUES list

ALL then every item on every information list previously defined, as well as LABELS is KILLed.

ALLBUT ALLBUT(name1,...) will do a KILL(ALL), except it will not KILL the names specified. Note that *namei* means a name such as U, V, F, G, not an infolist such as FUNCTIONS.)

If *argi* is the name of any of the other information lists (the elements of the **Maxima** variable INFOLISTS), then every item in that class (and its properties) is KILLed. See section 2.6 [Reviewing Options], page 29, for a listing of the INFOLISTS.

Note that KILL(VALUES) or KILL(variable) will not free the storage occupied unless the labels which are pointing to the same expressions are also KILLed. Thus if a large expression was assigned to X on line C7 one should do KILL(D7) as well as KILL(X) to release the storage occupied.

KILL removes all properties from the given argument thus KILL(VALUES) will kill all properties associated with every item on the VALUES list whereas the REMOVE set of functions (REMVALUE, REMFUNCTION, REMARRAY, REMRULE) remove a specific property. Also the latter print out a list of names or FALSE if the specific argument doesn't exist, whereas KILL always has value DONE even if the named item doesn't exist.

REMOVE (*args*) [*Special Form*]

will remove some or all of the properties associated with variables or functions. REMOVE(a1, p1, a2, p2, ...) removes the property *pi* from the atom *ai*. *Ai* and *pi* may also be lists as with DECLARE. *Pi* may be any property e.g. FUNCTION, MODE_DECLARE, etc. It may also be TRANSFUN implying that the translated Lisp version of the function is to be removed. This is useful if one wishes to have the **Maxima** version of the function executed rather than the translated version. *Pi* may also be OP or OPERATOR to remove a syntax extension given to *ai*. If *ai* is ALL then the property indicated by *pi* is removed from all atoms which have it. Unlike the more specific remove functions (REMVALUE, REMARRAY, REMFUNCTION, and REMRULE), REMOVE does not indicate when a given property is non-existent; it always returns DONE.

REM (*atom, ind*) [*Function*]

removes the property indicated by *ind* from the atom *atom*.

REMLET (*prod, rule*) [*Special Form*]

deletes the substitution rule, *prod* –> *repl*, most recently defined by the LET function. If name is supplied the rule is deleted from the rule package *rule*. REMLET() and REMLET(ALL) delete all substitution rules from the current rulepackage. If the name of a rule package is supplied, e.g. REMLET(ALL, name), the rule package *name* is also deleted. If a substitution is to be changed using the same product, REMLET need not be called, just redefine the substitution using the same product (literally) with the LET function and the new replacement and/or predicate name. Should REMLET(product); now be executed, the original substitution rule will be revived.

REMFUNCTION (*f1, f2, ...*) [*Special Form*]

removes the user defined functions *fi* from **Maxima**. If there is only one argument of ALL, then all functions are removed.

REMFUNCTION(*fun*) removes all functional properties and macro properties from *fun*. × REMFUNCTION(ALL) removes all functions and macros from the current environment. REMOVE(*name*, FUNCTION) removes the function property of *macro* if one exists, but will not remove macro properties. REMOVE(*name*, MACRO) removes the macro property of *name* if one exists, but will not remove function properties. See section 10.3 [Deleting From the Database], page 160.

KILL(FUNCTIONS) only effects the functions in the current environment and has no effects on any macros. Similarly, KILL(MACROS) only affects macros and has no effects on any defined functions.

REMVALUE (*name1, name2, ...*) [*Special Form*]

removes the values of user variables (which can be subscripted) from the system. If name is ALL then the values of all user variables are removed. Values are those items given names by the user as opposed to those which are automatically labeled by **Maxima** as Ci, Di, or Ei.

10.4 Renaming Elements in the Database

ALIAS (*newname1, oldname1, newname2, oldname2, ...*) [*Special Form*]

provides an alternate name for a (user or system) function, variable, array, etc. Any even number of arguments may be used.

ALIASES [*Variable*]

atoms which have a user defined alias (set up by the ALIAS, ORDERGREAT, ORDERLESS functions or by DECLAREing the atom a NOUN).

CHAPTER

ELEVEN

Input and Output

A file is simply an area on a particular storage device which contains data or text. Files on the disks are figuratively grouped into directories. A directory is just a list of all the files stored under a given user name.

`FILE_SEARCH` [*Variable*]

this is a list of files naming directories to search by LOAD and a number of other functions. The default value of this is a list of the various SHARE directories used by **Maxima**.

`FILE_SEARCH` (*"filename"*) [*Function*]

searches on those directories and devices specified by the FILE_SEARCH variable, and returns the name of the first file it finds. This function is invoked by the LOAD function, which is why LOAD("FFT") finds and loads '`share/fft.o`'. You may do FILE_SEARCH:CONS("devdir",FILE_SEARCH); to add other directories to the search rules, where *devdir* is the name of a device and a directory.

11.1 Loading Files

`LOAD` (*"filename"*) [*Function*]

takes one argument, a filename represented as a string (i.e. inside quotation marks), or as list (e.g. inside square brackets), and locates and loads in the indicated file. If no directory is specified, it then searches the SHAREi directories and any other directories listed in the FILE_SEARCH variable, and loads the first file that is found. LOAD("EIGEN") will load the eigen package, without the need for the user to be aware of the details of whether the package was compiled, translated, saved, or fassaved, i.e. LOAD will work on both LOADFILEable and BATCHable files. Note that LOAD will use BATCHLOAD if it finds the file is BATCHable, which means that it will BATCH the file in silently without terminal output or labels.

`LOADFILE` (*"filename"*) [*Special Form*]

loads a file as designated by its argument. This function may be used to bring back quantities that were stored from a prior **Maxima** session by use of the SAVE or STORE functions. *filename* must be a file of Lisp functions and expressions, not of **Maxima** command lines, in which case BATCH or DEMO is to be used.

LOADPRINT [*variable, default: TRUE*]

governs the printing of messages accompanying loading of files. The following options are available:

T**RUE** means always print the message;

'LOADFILE means print only when the LOADFILE command is used;

'AUTOLOAD means print only when a file is automatically loaded in;

F**ALSE** means never print the loading message.

PACKAGEFILE [*variable, default: FALSE*]

designers who use SAVE, FASSAVE, or TRANSLATE to create packages (files) for others to use may want to set PACKAGEFILE:TRUE; to prevent information from being added to **Maxima**'s information-lists (e.g. VALUES, FUNCTIONS) except where necessary when the file is loaded in. In this way, the contents of the package will not get in the user's way when he adds his own data. Note that this will not solve the problem of possible name conflicts. Also note that the flag simply affects what is output to the package file. Setting the flag to TRUE is also useful for creating **Maxima** init files.

11.1.1 Autoloading

SETUP_AUTOLOAD (*file, func1, ...*) [*Special Form*]

which takes two or more arguments: a file specification, and one or more function names, *funci*, and which indicates that if a call to *funci* is made and *funci* is not defined, that the file specified by *file* is to be automatically loaded in via LOAD, which file should contain a definition for *funci*. (This is the process by which calling e.g. INTEGRATE in a fresh **Maxima** causes various files to be loaded in.) As with the other file-handling commands in **Maxima**, the arguments to SETUP_AUTOLOAD are not evaluated. Note: SETUP_AUTOLOAD does not work for array functions.

11.2 Batching Files

BATCH (*file-specification*) [*Special Form*]

reads in and evaluates **Maxima** command lines from a file. This is a facility for executing command lines stored on a disk file rather than in the usual on-line mode. This facility has several uses, namely to provide a reservoir for working command lines, for giving error-free demonstrations, or helping in organizing one's thinking in complex problem-solving situations where modifications may be done via a text editor.

A batch file consists of a set of **Maxima** command lines, each with its terminating ; or $, which may be further separated by spaces, carriage-returns, form-feeds, and the like. The BATCH function calls for reading in the command lines from the file one at a time, echoing them on the user console, and executing them in turn. Control is returned to the user console only when serious errors occur, or when the end of the

file is met. Of course, the user may quit out of the file-processing by typing control-G at any point. BATCH files may be created using a text editor or by use of the STRINGOUT command.

BATCHKILL [*variable, default: FALSE*]

if TRUE then the effect of all previous BATCH files is nullified because a KILL(ALL) and a RESET() will be done automatically when the next one is read in. If BATCHKILL is bound to any other atom, then a KILL of the value of BATCHKILL will be done.

BATCHLOAD (*file-specification*) [*Function*]

BATCHes in the file silently without terminal output or labels.

BATCON (*argument*) [*Special Form*]

continues BATCHing in a file which was interrupted.

BATCHCOUNT [*variable, default: 0*]

may be set to the number of the last expression BATCHed in from a file. Thus BATCON(BATCOUNT-1) will resume BATCHing from the expression before the last BATCHed in from before.

11.2.1 Indexed Batch Files

OPEN_INDEX_FILE (*filename*) [*Function*]

returns a symbol which represents the indexed file. It is easiest to use if the user sets a variable to the value returned, or refers to it by its D-line number. E.g. F:OPEN_INDEX_FILE(KAM1,EXPODE); sets the variable F to the symbol which represents the indexed file object.

MAKE_INDEX_FILE (*filename*) [*Function*]

will parse a batch file without evaluating it. This is useful for debugging init files.

READ_NTH_OBJECT (*n, symbol-returned-by-OPEN_INDEX_FILE*) [*Function*]

returns the *n*'th object in an indexed file.

MAP_OVER_INDEX_FILE (*function, fileobject*) [*Function*]

is convenient for generating an index list of properties of the objects in a file vs. their positions in the file.

INDEX_FILE_DIM(*symbol-returned-by-OPEN_INDEX_FILE*) returns an integer indicating the number of expressions in the indexed file.

11.3 Demoing Files

`DEMO` (*file*) [*Special Form*]

This is the same as BATCH but pauses after each command line and continues when a space is typed. In øNIL **Maxima**, DEMO does file searching like BATCH does, and in addition searches for files on the DEMO directory in the **Maxima** directory hierarchy, and also for files with extensions of `.`DEM, `.`DM1, and `.`DM2 in addition to `.`MC.

`PROMPT` [*variable, default: _*]

is the prompt symbol of the DEMO function, PLAYBACK (SLOW) mode, and BREAK.

11.4 Writing to Files

`WRITEFILE` (*"file"*) [*Special Form*]

opens up a file for writing. All interaction between the user and **Maxima** is then recorded in this file, just as it is on the console. Such a file is a transcript of the session, and is not reloadable or BATCHable into **Maxima** again. (See also CLOSEFILE.)

`APPENDFILE` (*filename1, filename2, directory*) [*Special Form*]

is like WRITEFILE(*filename*) but appends to the file whose name is *filename*. A subsequent CLOSEFILE will delete the original file and rename the appended file.

`STRINGOUT` (*args*) [*Special Form*]

will output an expression to a file in a linear format. Such files are then used by the BATCH or DEMO commands.

STRINGOUT(*file, A1, A2, ...*) outputs to a file given by *file* the values given by *A1, A2, ...* in a **Maxima** readable format. The file-specification may be omitted, in which case the default values will be used. The *Ai* are usually C labels or may be

INPUT meaning the value of all C labels.

FUNCTIONS which will cause all of the user's function definitions to be strungout (i.e. all those retrieved by DISPFUN(ALL)).

VALUES all the variables to which the user has assigned values will be strungout.

ai may also be a list [m,n] which means to stringout all labels in the range *m* through *n* inclusive. This function may be used to create a file of FORTRAN statements by doing some simple editing on the strungout expressions. If the GRIND switch is set to TRUE, then STRINGOUT will use GRIND format instead of STRING format. Note: a STRINGOUT may be done while a WRITEFILE is in progress.

`CLOSEFILE` (*file*) [*Special Form*]

closes a file opened by WRITEFILE and gives it the name *file*. (On a Lisp Machine, or under Franz Lisp, one need only say CLOSEFILE();.) Thus to save a file consisting of the display of all input and output during some part of a session with **Maxima** the user issues a WRITEFILE, interacts with **Maxima**, then issues a CLOSEFILE. The user can also issue the PLAYBACK function after a WRITEFILE to save the display of previous transactions. The expression can then be brought back into **Maxima** via the LOADFILE function. To save the expression in a linear form which may then be BATCHed in later, the STRINGOUT function is used.

`DELFILE` (*file*) [*Special Form*]

will delete the file given by the file-specification *file*.

11.5 Operating on Files

`PRINTFILE` (*file*) [*Special Form*]

prints the contents of the file *file* on the user's terminal.

There are some usage notes in the file: '`share/fileop.usg`'.

`FILENAME_MERGE` (*"filename1", "filename2", ...*) [*Function*]

merges together filenames. What this means is that it returns *filename1* except that missing components come from the corresponding components of *filename2*, and if they are missing there, then from *filename3*, etc.

`RENAMEFILE` (*oldfile, newfile*) [*Special Form*]

Gives a new name to a file. *Oldfile* may name any file. *Newfile* must be a filename on the same device and directory. Attempting to change the name of a file that does not exist, or to rename a file to the name of a file that exists already will generate an error; hence, it is not possible to inadvertently destroy a file using this command.

There are some usage notes in the file: '`share/fileop.usg`'.

11.6 Directories

`LISTFILES` (*DSK, username*) [*Special Form*]

To list your directory. If you use a shared directory such as Users or Plasma, only your files–the ones with your login name as first file name–will be shown. The length and date of creation of each file is also shown. There is a shorter list command, QLISTFILES which gives just the file names and no length or date information.

There are some usage notes in the file: '`share/fileop.usg`'.

`QLISTFILES` (*DSK,username*) [*Special Form*]

"Quick LISTFILES" lists the names of your files. There are some usage notes in the file: '`share/fileop.usg`'.

11.7 File Defaults

`FILE_TYPE` (*filename*) [*Function*]

returns `FASL`, `LISP`, or `Maxima`, depending on what kind of file it is. `FASL` means a compiled Lisp file, `LISP` means Lisp source code, and `Maxima` means **Maxima** source code. The type of the file is determined by looking up the filename extension in the variable **FILE_TYPES**.

11.8 Saving and Restoring

`SAVE` (*[optional file spec], arg1, arg2, . . . , argi*) [*Special Form*]

saves quantities described by its arguments on disk and keeps them in core also. If you omit the file spec, **Maxima** will select a file name for you consisting of the first three letters of your login name followed by a digit (the lowest digit which will give a unique file name). The *arg*'s are the expressions to be SAVEd. ALL is the simplest, but note that saving ALL will save the entire contents of your **Maxima**, which in the case of a large computation may result in a file which is too large to be reloaded. VALUES, FUNCTIONS, or any other items on the INFOLISTS may be SAVEd, as may functions and variables by name. `C` and `D` lines may also be saved, but it is better to give them explicit names, which may be done in the command line, e.g. SAVE(RES1=D15);. Files saved with SAVE should be reloaded with LOADFILE.

SAVE returns a list of the form [<name of file>, <size of file in blocks>,. . .] where `...` are the things saved. Warnings are printed out in the case of large files, or if an empty file is accidently generated. It is possible to do a SAVE when a WRITEFILE is in progress.

`FASSAVE` (*args*) [*Special Form*]

is similar to SAVE but produces a FASL file in which the sharing of subexpressions which are shared in core is preserved in the file created. Hence, expressions which have common subexpressions will consume less

space when loaded back from a file created by FASSAVE rather than by SAVE. Files created with FASSAVE are reloaded using LOADFILE, just as files created with SAVE. FASSAVE returns a list of the form [<name of file>, <size of file in blocks>, ...] where ... are the things saved. Warnings are printed out in the case of large files. FASSAVE may be used while a WRITEFILE is in progress.

RESTORE (*file*) [*Special Form*]

reinitializes all quantities filed away by a use of the SAVE or STORE functions, in a prior **Maxima** session, from the file given by '`file`' without bringing them into core.

CHAPTER

TWELVE

Programming Environment

12.1 On-Line Help

For introductory help with **Maxima** you may try the on-line Primer, which is an interactive tutorial. Type PRIMER(); in **Maxima**. For HELP with **Maxima** the DESCRIBE command is useful. DESCRIBE(); will tell you how to use it, or to find out about a specific **Maxima** command, do DESCRIBE("command");. See section 2.6 [Reviewing Options], page 29 for the definition of INFOLISTS.

12.1.1 Apropos

APROPOS (*string*) [*Special Form*]

takes a character string as argument and looks at all the **Maxima** names for ones with that string appearing anywhere within them. Thus, APROPOS(EXP); will return a long list of all the flags and functions which have *EXP* as part of their names, such as EXPAND, EXP, EXPONENTIALIZE. Thus if you can only remember part of the name of something you can use this command to find the rest of the name. Similarly, you could say APROPOS(TR_); to find a list of many of the switches relating to the TRANSLATOR, as most of these begin with TR_.

12.1.2 Describe

DESCRIBE (*arg*) [*Special Form*]

prints out information about *arg*, which may be any **Maxima** command, switch or variable. Certain key words have also been included, where they seem appropriate, thus DESCRIBE(TRIG); will print out a list of the trig functions implemented in **Maxima**. Some function names or operators may require quotation marks around them, e.g. DESCRIBE("DO"); or DESCRIBE(".");. See also APROPOS which allows you to locate command names even if you aren't sure of the full name.

12.1.3 Example

EXAMPLE (*command*) [*Special Form*]

will start up a demonstration of how *command* works on some expressions. After each command line it will

pause and wait for a space to be typed, as in the DEMO command.

12.1.4 Primer

`PRIMER` () [*Special Form*]

starts up the on-line PRIMER in **Maxima**. The first time it is run by a particular user, the script CONSOLEPRIMER is gone through to make sure the user knows how to use a console. Subsequently it will start up offering a list of scripts to choose from. PRIMER(*script*) will start it with a particular script.

12.2 Editing

12.2.1 Line Editing

There is a line editor in **Maxima**, which you may enter by typing an altmode (the `ESC` key on most consoles). Your current command line will be automatically brought into the editor for you to edit. Commands are terminated by two altmodes. An additional two altmodes will return you to **Maxima** with the edited string on your command line. Some useful commands:

nC moves the cursor past n characters.

nR moves the cursor past n characters in the reverse direction (nR = -nC).

nSstring<$> moves the cursor to the right (left if *n* is negative) of the *n*'th occurrence of *string* in the input string.

) or] moves the cursor right from the current position over the next balanced pair of parentheses (or brackets).

(or [similar to) or] but moves left.

nD deletes *n* characters, and saves them in the save-register (see the `G`R command below).

nK deletes all the characters through the *n*'th carriage return (0K kills left), and saves them in the save-register; e.g., `K` deletes the remainder of this line.

Istring<$> inserts the characters *string* at the current cursor position. The cursor is positioned at the right of the inserted text. If no argument is given then the string of the last I command which had one is used.

GR inserts at the current cursor position the characters deleted by the last use of `D` or `K`. Thus `G`R may be used in combination with `D` or `K` to move characters from one place to another in the input string; or to recover from an accidental use of `D` or `K`. There is only one save-register.

`CURSOR` [*variable, default: $*]

is the prompt symbol of the **Maxima** line editor.

12.2.2 Full Screen Editing

12.2.3 Expression Editor

12.2.3.1 Terminology

Expression A **Maxima** expression is the fundamental data object used by the expression editor, just as characters or lines are fundamental units within a text editor. An expression as used by the editor consists of a mathematical expression in the internal **Maxima** format, and a label displayed with it. Several other pieces of information are also associated with the expression. One is its "reveal depth", i.e. the depth to which the body of the expression is displayed within the buffer, as done by the REVEAL command. The position of the expression relative to the others within the buffer is also maintained. Finally, an expression will possess a "region", which is a subexpression to which the next **Maxima** command will be applied.

Region Essentially, the region is that portion of an expression which is of interest. The subexpression designated by the region can be used as input to any **Maxima** command, and the result returned by the command can displace the original subexpression. This capability allows one to selectively apply the **Maxima** simplifiers to a portion of an expression, without modifying the entire form. The region of an expression must exist entirely at one level within the expression, and can contain any number of adjacent branches at that level.

Level A single node within the tree structure representing a mathematical expression. A level is characterized by an operator and a number of operands which are the branches of that node. For example, the operator '+' and the operands a**2, 2ab, and b**2 describe the top level of the expression a**2+2ab+b**2. Similarly, the operator '*' and the operands 2, a, and b describe the middle node of the next level of the expression.

Branch Also a single node within a tree structure. Refers to one of the operands at a given level. For example, a**2 is a branch of the top level of the expression a**2+2ab+b**2.

Region Width The number of branches contained within the region of an expression.

Buffer An object used to group expressions together, for viewing as a whole or for storing into a file as a single unit. A buffer can contain any number of expressions, and is designated by a name. When a collection of expressions in a buffer is being edited, one of the expressions will be considered "current", and the region of the current expression will be outlined in a box.

Window A rectangular region of the screen which is used for viewing a portion of a buffer. Only a certain number of the expressions within a buffer will be displayed in a window at single time, however this window can be adjusted to view any portion of the buffer which will fit within the size of the window.

12.2.3.2 User Interface

The editor itself is structured into several components—expression and buffer manipulating primitives, screen management routines, and a terminal/operating system interface. The expression and buffer manipulation is performed by a set of commands, which are listed below. Each command will have a long name and may optionally be associated with a single key on the terminal keyboard. Typing the key associated with a command will cause the command to be invoked, in much the same way single keystrokes are used by display-oriented text editors for moving a cursor and modifying text. An initial association between commands and keys is provided by the editor, however this association can be easily modified by the user to fit particular needs.

12.2.3.3 Keyboard Commands

The editor is ideally designed to be used from a Knight keyboard, i.e. one of the type used on a Lisp Machine or on the MIT Plasma Physics Lab terminal system, however it can be used from an ASCII keyboard as well. Three types of keystrokes are recognized: The first involves the typing of a single keyboard character without any shift key being depressed. The second is the typing of a character with the '`Control`' key of the keyboard depressed. The keystroke generated by typing the '`A`' key with the '`Control`' key depressed will be written as '`Control-A`' or '`C-A`'. The third is the typing of a character with the '`Meta`' key depressed. Meta keys will be written with a prefix of '`Meta-`' or '`M-`'. Terminals without Meta keys can type these keystrokes by using the '`ESCAPE`' key as a prefix character. Those of you familiar with the øEmacs text editor will recognize that the keystroke/editor command association mechanism and user interface in general are quite similar. In addition, the commands for moving the region within an expression correspond to those used in øEmacs for moving the cursor within a text buffer.

The editor also has a primitive self-documenting capability. Hopefully, this will be extended to include many of the features available in the current øEmacs system. Currently, one can type '`?`' and then a key, to see a description of the command associated with that key.

12.2.3.4 Usage

The **Maxima** expression editor currently runs on both the Maxima Consortium machine and the Lisp Machines at MIT. The redisplay algorithm currently used is not very sophisticated, and the editor can only effectively be used from terminals operating at 120 characters/sec speed or greater. When the redisplay is improved to take advantage of such terminal capabilities as line insertion and deletion (as provided by the Teleray 1061 or HP 2645 terminals) and variable scroll regions (as proved by the HDS Concept 100 or DEC VT-100), the editor should be quite usable at 30 characters/sec. The editor is invoked by the function DISPLAYEDIT which may be called with no arguments or with a number of expressions to edit.

12.2.3.5 Command Summary

The first set of commands deal with moving the region within an expression and selecting expressions within a buffer. Those commands which end in "branch", "level", or "region" apply to the current expression, i.e. involve movement within or modification of the current expression. These commands are generally associated with the control keys. Those commands ending with "expression" apply to the current buffer and may involve selection or creation of an expression within the buffer. These commands are associated with the meta keys.

C-F Forward Branch

C-B Next Branch

M-F Grow Region

M-B Shrink Region

< Top Level

C-A First Branch

C-E Last Branch

C-P Previous Level

C-N Next Level

M-A First Expression

M-E Last Expression

M-P Previous Expression

M-N Next Expression

This group of commands deal with creating, deleting, and modifying expressions within a buffer, and with performing the same operations on portions of an expression.

C-D Delete Region

C-K Kill Following Branches

C-C Copy Region

C-I Insert Branch

C-S Substitute In Region

C-T Transpose Branch

C-R Replace Region

C-Y Yank Branch

M-D Delete Expression

M-K Kill Following Expressions

M-C Copy Expression

M-I Insert Expression

M-S Substitute In Expression

M-T Transpose Expression

M-R Replace Expression

M-Y Yank Expression

Maxima Commands

Each of the following commands are applied to the subexpression contained in the region and the result displaces the subexpression originally there. These commands were selected as being among the most commonly used **Maxima** commands, but are by no means all that could be integrated into the expression editor. Any **Maxima** command which is frequently used and often applied to only part of an expression can be easily added.

: Assign Expression

+ Add To Region

***** Multiply To Region

^ Exponentiate Region

s Simplify Region

v Evaluate Region

r Ratsimp Region

m Multthru Region

d Differentiate Region

e Expand Region

f Factor Region

i Integrate Region

Auxiliary Commands

? Describe Key

C-G Command Abort

C-^ Control Prefix

ESC Meta Prefix

0-9 Numeric argumen

C-L New Window

C-X C-C Exit

C-X B Select Buffer

C-X C-B List Buffers

C-X C-E Evaluate Maxima Expression

C-X C-R Read File

C-X C-W Write File

12.3 System Functions

12.3.1 System Status

STATUS (*arg*) [*Special Form*]

will return miscellaneous status information about the user's **Maxima** depending upon the *arg* given. Not all of these features are supported on all versions of **Maxima**. Permissible arguments and results are as follows:

TIME the time used so far in the computation.

DAY the day of the week.

DATE a list of the year, month, and day.

DAYTIME a list of the hour, minute, and second.

RUNTIME accumulated cpu time in milliseconds in the current **Maxima**.

REALTIME the real time (in sec) elapsed since the user started up his **Maxima**.

GCTIME the garbage collection time used so far in the current computation.

TOTALGCTIME gives the total garbage collection time used in **Maxima** so far.

FREECORE the number of blocks of core your **Maxima** can expand before it runs out of address space. (A block is 1024 words.)

FEATURE gives you a list of system features. For example, he list for øFranz Unix **Maxima** is LONGFILENAMES, SUN, PORTABLE, 68K, SYSTEMS, STRING, UNIX and FRANZ. Any of these features may be given as a second argument to STATUS(FEATURE,...); If the specified feature exists, TRUE will be returned,, otherwise FALSE. Note: these are system features, and not really user related.

SSTATUS (*feature,package*) [*Special Form*]

meaning "set status." It can be used to SSTATUS(FEATURE,HACK_PACKAGE) so that × STATUS(FEATURE,HACK_PACKAGE) will then return TRUE. This can be useful for package writers, to keep track of what features they have loaded in.

VERSION [*variable, default: 304*]

is the version number of **Maxima**. This could be useful if the user wants to label his output, or to associate bugs with a particular version.

12.3.2 Timing the Evaluation of Expressions

`SHOWTIME` [*variable, default: FALSE*]

if TRUE then the computation time will be printed automatically with each output expression. By setting SHOWTIME:ALL, in addition to the cpu time **Maxima** now also prints out (when not zero) the amount of time spent in garbage collection (gc) in the course of a computation. This time is of course included in the time printed out as `time=` . (It should be noted that since the `time=` time only includes computation time and not any intermediate display time or time it takes to load in out-of-core files, and since it is difficult to ascribe responsibility for gc's, the gctime printed will include all gctime incurred in the course of the computation and hence may in rare cases even be larger than `time=`).

`TIME` (*Di1, Di2, . . .*) [*Special Form*]

gives a list of the times in milliseconds taken to compute the *Di*. (Note: the variable SHOWTIME default: [FALSE], may be set to TRUE to have computation times printed out with each D-line.)

`LASTTIME` [*Variable*]

returns the time to compute the last expression in milliseconds presented as a list of time and gctime.

12.4 Error Handling

`ERROR` (*arg1, arg2, . . .*) [*Function*]

will evaluate and print its arguments and then will cause an error return to top level **Maxima**, or to the nearest enclosing ERRCATCH. This is useful for breaking out of nested functions if an error condition is detected, or wherever one can't type control-^. The variable **ERROR** is set to a list describing the error, the first of it being a string of text, and the rest the objects in question. ERRORMSG(); is the preferred way to see the last error message.

`ERREXP` [*Variable*]

When an error occurs in the course of a computation, **Maxima** prints out an error message and terminates the computation. ERREXP is set to the offending expression and the message ERREXP contains the offending expression is printed. The user can then type ERREXP; to see this and hopefully find the problem.

`ERROR_SYMS` [*Variable*]

In error messages, expressions larger than ERROR_SIZE are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the list ERROR_SYMS, and are initially ERREXP1, ERREXP2, ERREXP3, etc. After an error message is printed, e.g. The function FOO doesn't like ERREXP1 as input., the user can type ERREXP1; to see the expression. ERROR_SYMS may be set by the user to a different set of symbols, if desired.

PARSEWINDOW [*variable, default: 10*]

the maximum number of lexical tokens that are printed out on each side of the error-point when a syntax (parsing) error occurs. This option is especially useful on slow terminals. Setting it to -1 causes the entire input string to be printed out when an error occurs.

ERROR_SIZE [*variable, default: 20*]

controls the size of error messages. For example, let U:(C**D**E+B+A)/(COS(X-1)+1);. U has an error size of 24. So if ERROR_SIZE has value < 24 then

```
(C1) ERROR("The function", FOO,"doesn't like", U,"as input.");
```

prints as: The function FOO doesn't like ERREXP1 as input. If ERROR_SIZE>24 then it prints as:

```
                                  E
                                 D
                                C   + B + A
The function FOO doesn't like -------------- as input.
                              COS(X - 1) + 1
```

Expressions larger than ERROR_SIZE are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the user-settable list ERROR_SYMS default: [ERREXP1, ERREXP2, ERREXP3]. The default value of this switch might change depending on user experience.

ERRORFUN [*variable, default: FALSE*]

if set to the name of a function of no arguments will cause that function to be executed whenever an error occurs. This is useful in BATCH files where the user may want his or her **Maxima** killed or his terminal logged out if an error occurs. In these cases ERRORFUN would be set to QUIT or LOGOUT.

ERRORMSG () [*Function*]

reprints the last error message. This is very helpful if you are using a display console and the message has gone off the screen. The variable ERROR is set to a list describing the error, the first of it being a string of text, and the rest the objects in question. TTYINTFUN:LAMBDA([],ERRORMSG(),PRINT("")); will set up the user-interrupt character to reprint the message.

See section 2.1.3 [Throw and Catch], page 15, for a programming construct useful for catching errors.

12.5 Break Points and Debugging

BREAK (*arg1, ...*) [*Function*]

Will evaluate and print its arguments and will then cause a Maxima-BREAK at which point the user can examine and change his environment. Upon typing EXIT; the computation resumes. Control-A will enter

a Maxima-BREAK from any point interactively. EXIT; will continue the computation. Control-X may be used inside the Maxima-BREAK to quit locally, without quitting the main computation.

DEBUGMODE [*variable, default: FALSE*]

causes **Maxima** to enter a **Maxima** break loop whenever a **Maxima** error occurs if it is TRUE and to terminate that mode if it is FALSE. If it is set to ALL then the user may examine BACKTRACE for the list of functions currently entered.

BACKTRACE [*Variable*]

when DEBUGMODE:ALL has been done, this variable has as value a list of all functions currently entered.

DEBUGPRINTMODE () [*Function*]

returns current printing mode used by the DEBUG function. DEBUGPRINTMODE(LISP) sets it to lisp printing (the default), DEBUGPRINTMODE(x, y) where *x* is FALSE or a positive fixnum, sets the maximum length to which Lisp expressions are printed to *x*, and the maximum depth to *y*. This is used to abbreviate printout. FALSE means INFINITY.

DEBUGPRINTMODE(Maxima) sets it to try printing expressions as they would be displayed at **Maxima** level, except with more information and a slightly different notation.

LISPDEBUGMODE (*TRUE*) [*Function*]

will cause lisp errors to enter normal lisp break points, from which it is normal to call the lisp function (DEBUG) which prompts for single character commands to move up and down the evaluation stack at the point of the error. The following are some of its commands:

U move up the stack.

D move down the stack.

S pretty-print the expression current stack level.

A print indented list of function calls on stack.

? type out list of commands.

LISPDEBUGMODE is useful for debugging translated code. It is possible to do sophisticated error recovery from inside these break loops. Do LISPDEBUGMODE(FALSE) to turn off. LISPDEBUGMODE() returns present debugging state. Note: LISPDEBUGMODE also passes its argument to DEBUGMODE.

In Lisp (DEBUG) calls the lisp stack debugger when at lisp level (e.g. a lisp debugging break).

TOPLEVEL [*Special Form*]

During a break one may type TOPLEVEL;. This will cause top-level **Maxima** to be entered recursively. Labels will now be bound as usual. Everything will be identical to the previous top-level state except that the

computation which was interrupted is saved. The function TOBREAK will cause the break which was left by typing TOPLEVEL; to be re-entered. If TOBREAK is given any argument whatsoever, then the break will be exited, which is equivalent to typing TOBREAK() immediately followed by EXIT;.

TOBREAK () [*Special Form*]

causes the **Maxima** break which was left by typing TOPLEVEL; to be re-entered. If TOBREAK is given any argument whatsoever, then the break will be exited, which is equivalent to typing TOBREAK() immediately followed by EXIT;.

SETCHECKBREAK [*variable, default: FALSE*]

if set to TRUE will cause a Maxima-BREAK to occur whenever the variables on the SETCHECK list are bound. The break occurs before the binding is done. At this point, SETVAL holds the value to which the variable is about to be set. Hence, one may change this value by resetting SETVAL.

SETVAL [*Variable*]

holds the value to which a variable is about to be set when a SETCHECKBREAK occurs. Hence, one may change this value by resetting SETVAL.

12.6 Tracing

TRACE (*name1, name2, ...*) [*Function*]

gives a trace printout whenever the functions mentioned are called. TRACE() prints a list of the functions currently under TRACE. To remove tracing, see UNTRACE. There is a demo in the file: '`demo/trace.dem`'. There are some usage notes in the file: '`macdoc/trace.usg`'.

UNTRACE (*name, ...*) [*Special Form*]

removes tracing invoked by the TRACE function. UNTRACE() removes tracing from all functions.

TRACE_OPTIONS (*fun, option1, option2, ...*) [*Special Form*]

gives the function *fun* the options indicated. An option is either a keyword or an expression. The possible keywords are:

NOPRINT if TRUE do no printing.

BREAK if TRUE give a breakpoint.

LISP_PRINT if TRUE use lisp printing.

INFO Extra info to print.

ERRORCATCH if TRUE errors are caught.

A keyword means that the option is in effect. Using a keyword as an expression, e.g. NOPRINT(predicate_function) means to apply the *predicate_function* (which is user-defined) to some arguments to determine if the option is in effect. The argument list to this predicate function is always [LEVEL, DIRECTION, FUNCTION, ITEM] where LEVEL is the recursion level for the function. DIRECTION is either ENTER or EXIT. FUNCTION is the name of the function. ITEM is either the argument list or the return value. Its usage and motivation is probably best shown by the following example:

```
(C21) DEBUGMODE:TRUE$ /* For best results */
(C22) f(x):=1/x$
(C23) trace(F);
(D23)       [F]
(C24) trace_options(f,errorcatch);
(D24)       [ERRORCATCH]
(C25) f(0);
1 Enter F [0]
Division by 0
ERROR-BREAK  (Type EXIT; to exit.)
_f(x):=if x=0 then und else 1/x$
_exit;
Exited from the break
Error during application of F at level 1
 Do you want to:
 0 -> Signal an error, i.e. PUNT?
 1 -> Retry with same arguments?
 2 -> Retry with new arguments?
 3 -> Exit with user supplied value
1;
Re-applying the function F
1 Enter F [0]
1 Exit  F UND
(D25)          UND
```

What the person did was to realize that the definition of F was wrong, and then simply re-define it while in the break loop, telling **Maxima** to retry the function with the same arguments as before. However, had it been the caller of F who was at fault, he could have fixed the caller, and then used option 2 to manually give F the correct arguments. Option 3 is used when you don't want to fix anything yet, and just want to manually make F return the correct value. Option 0 is good for returning to the next higher dynamic level of control, (hence the not-quite-accurate football analogy PUNT). There is a demo in the file: '`demo/trace.dem`'. There are some usage notes in the file: '`macdoc/trace.rcn`'. ḍone

`TIMER` (*fun*) [*Function*]

will put a timer-wrapper on the function *fun*, within the TRACE package, i.e. it will print out the time spent in computing *fun*.

`TIMER_INFO` (*F*) [*Special Form*]

will print the information on timing which is stored as GET('F,'CALLS); GET('F,'RUNTIME); and GET('F,'GCTIME);. This is a function in the TRACE package.

12.6.1 Tracing Flags

`TRACE_BREAK_ARG` [*Variable*]

Bound to list of argument during BREAK ENTER, and the return value during BREAK EXIT. This lets you change the arguments to a function, or make a function return a different value, which are both useful debugging hacks.

`TRACE_MAX_INDENT` [*Variable*]

The maximum indentation of trace printing.

`TRACE_SAFETY` [*variable, default: TRUE*]

Consider for example

```
F(X):=X;
BREAKP([L]):=(PRINT("Hi!",L), FALSE), TRACE(F,BREAKP);
TRACE\_OPTIONS(F,BREAK(BREAKP));
F(X);
```

Note that even though BREAKP is traced, and it is called, it does not print out as if it were traced. If you set TRACE_SAFETY to FALSE then F(X); will cause a normal trace-printing for BREAKP. However, then consider TRACE_OPTIONS(BREAKP,BREAK(BREAKP)); When TRACE_SAFETY:FALSE; F(X); will give an infinite recursion, which it would not if TRACE_SAFETY were turned on.

12.7 Operating System

`TIMEDATE` () [*Function*]

prints out the current date and time.

CHAPTER

THIRTEEN

Translation and Compiling

13.1 Mode Declarations

`MODE_DECLARE` (*y1, mode1, y2, mode2, ...*) [*Function*]

MODE_DECLARE is used to declare the modes of variables and functions for subsequent translation or compilation of functions. Its arguments are pairs consisting of a variable *yi*, and a mode which is one of BOOLEAN, FIXNUM, NUMBER, RATIONAL, or FLOAT. Each *yi* may also be a list of variables all of which are declared to have *modei*. **MODEDECLARE** is a synonym for this.

If *yi* is an array, and if every element of the array which is referenced has a value, then ARRAY(yi, COMPLETE, dim1, dim2, ...) rather than ARRAY(yi, dim1, dim2, ...) should be used when first declaring the bounds of the array. If all the elements of the array are of mode FIXNUM or FLOAT, use FIXNUM or FLOAT instead of COMPLETE. Also if every element of the array is of the same mode, say `m`, then MODE_DECLARE(COMPLETEARRAY(yi),m)) should be used for efficient translation. Also numeric code using arrays can be made to run faster by declaring the expected size of the array, as in: MODE_DECLARE(COMPLETEARRAY(A[10,10]),FLOAT) for a floating point number array which is 10 x 10.

Additionally one may declare the mode of the result of a function by using FUNCTION(F1, F2, ...) as an argument; here F1,F2,... are the names of functions. For example the expression,

```
MODE_DECLARE([FUNCTION(F1,F2,\dots),X],FIXNUM,Q,COMPLETEARRAY(Q),FLOAT)
```

declares that X and the values returned by F1,F2,... are single-word integers and that Q is an array of floating point numbers.

MODE_DECLARE is used either immediately inside of a function definition or at top-level for global variables. See the file '`maxdoc/mcompi.doc`' for some examples of the use of MODE_DECLARE in translation and compilation.

`MODE_IDENTITY` (*arg1, arg2*) [*Special Form*]

A special form used with MODE_DECLARE and MACROS to declare, e.g., a list of lists of flonums, or other compound data object. The first argument to MODE_IDENTITY is a primitive value mode name as given to MODE_DECLARE (i.e. [FLOAT,FIXNUM,NUMBER, LIST,ANY]), and the second argument is an expression which is evaluated and returned as the value of MODE_IDENTITY. However, if the return

value is not allowed by the mode declared in the first argument, an error or warning is signalled. The important thing is that the MODE of the expression as determined by the **Maxima** to Lisp translator, will be that given as the first argument, independent of anything that goes on in the second argument. E.g. X:3.3; MODE_IDENTITY(FIXNUM,X); is an error. MODE_IDENTITY(FLONUM,X) returns 3.3.

This has a number of uses, e.g., if you knew that FIRST(L) returned a number then you might write MODE_IDENTITY(NUMBER,FIRST(L)). However, a more efficient way to do it would be to define a new primitive,

```
FIRSTNUMB(X) ::= BUILDQ([X],MODE_IDENTITY(NUMBER,X));
```

and use FIRSTNUMB every time you take the first of a list of numbers.

13.1.1 Mode Declaration Flags

MODE_CHECKP [*variable, default: TRUE*]

If TRUE, MODE_DECLARE checks the modes of bound variables.

MODE_CHECK_ERRORP [*variable, default: FALSE*]

If TRUE, MODE_DECLARE calls ERROR.

MODE_CHECK_WARNP [*variable, default: TRUE*]

If TRUE, mode errors are described.

13.2 Translation

TRANSLATE (*fun1, fun2, . . .*) [*Special Form*]

translates the user defined functions *funi* from the **Maxima** language to Lisp (i.e. it makes them EXPRs). This results in a gain in speed when they are called. There is now a version of **Maxima** with the **Maxima** to lisp translator pre-loaded into it. Functions to be translated should include a call to MODE_DECLARE at the beginning when possible in order to produce more efficient code. For example:

```
F(X1,X2,\dots) := BLOCK([v1,v2,\dots ],
        MODE_DECLARE(v1,mode1,v2,mode2,\dots),\dots)
```

where the *X1,X2,. . .* are the parameters to the function and the *v1,v2,. . .* are the local variables. The names of translated functions are removed from the FUNCTIONS list if SAVEDEF is FALSE (see below), and are added to the PROPS lists. Functions should not be translated unless they are fully debugged. Also, expressions are assumed simplified; if they are not, correct but non-optimal code gets generated. Thus, the user should not set the SIMP switch to FALSE which inhibits simplification of the expressions to be translated.

One can translate functions stored in a file by giving TRANSLATE an argument which is a file specification. The result returned by TRANSLATE is a list of the names of the functions TRANSLATEd. In the case of a file translation, the corresponding element of the list is a list of the first and second new file names containing the Lisp code resulting from the translation. This will be fn1 Lisp on the disk directory dir. The file of Lisp code may be read into **Maxima** by using the LOADFILE function.

`TRANSLATE_FILE` (*file*) [*Function*]

translates a file of **Maxima** code into a file of Lisp code. It takes one or two arguments. The first argument is the name of the **Maxima** file, and the optional second argument is the name of the Lisp file to produce. The second argument defaults to the first argument with second file name the value of TR_OUTPUT_FILE_DEFAULT which defaults to `.LSP`. Also produced is a file of translator warning messages of various degrees of severity. The second file name is usually `.UNL`. This file contains valuable (albeit obscure for some) information for tracking down bugs in translated code. Do APROPOS(TR_) to get a list of TRANSLATE switches. In summary, TRANSLATE_FILE([FOO.BAR]), LOADFILE(FOO.TRLISP) is equal to BATCH(FOO,BAR) modulo certain restrictions (the use of ” and % for example).

13.2.1 Translation Flags

`TRANSLATE` [*variable, default: FALSE*]

If TRUE, causes automatic translation of a user’s function to Lisp. Note that translated functions may not run identically to the way they did before translation as certain incompatpbilities may exist between the Lisp and **Maxima** versions. Principally, the RAT function with more than one argument, and the RATVARS function should not be used if any variables are MODE_DECLAREd CRE form. Also the PREDERROR default: [FALSE] setting will not translate.

`SAVEDEF` [*variable, default: TRUE*]

if TRUE will cause the **Maxima** version of a user function to remain when the function is TRANSLATEd. This permits the definition to be displayed by DISPFUN, and allows the function to be edited.

`TRANSRUN` [*variable, default: TRUE*]

if FALSE will cause the interpreted version of all functions to be run (provided they are still around), rather than the translated version.

`TRANSBIND` [*Variable*]

if TRUE removes global declarations in the local context. This applies to variables which are formal parameters to functions which one is TRANSLATEing from **Maxima** code to Lisp.

TRANSCOMPILE [*variable, default: FALSE*]

if true, TRANSLATE will generate the declarations necessary for possible compilation. The COMPFILE command uses TRANSCOMPILE:TRUE;.

UNDECLAREDWARN [*variable, default: COMPFILE*]

A switch in the Translator. There are four relevant settings:

FALSE never print warning messages.

COMPFILE warn when in COMPFILE

TRANSLATE warn when in TRANSLATE and when TRANSLATE:TRUE

ALL warn in COMPFILE and TRANSLATE

Do MODE_DECLARE(variable,ANY) to declare a variable to be a general **Maxima** variable (i.e. not limited to being FLOAT or FIXNUM). The extra work in declaring all your variables in code to be compiled should pay off.

TR_ARRAY_AS_REF [*variable, default: TRUE*]

if TRUE runtime code uses the value of the variable as the array.

TR_BOUND_FUNCTION_APPLYP [*variable, default: TRUE*]

Gives a warning if a bound variable is found being used as a function.

TR_FILE_TTY_MESSAGESP [*variable, default: FALSE*]

Determines whether messages generated by TRANSLATE_FILE during translation of a file will be sent to the TTY. If FALSE (the default), messages about translation of the file are only inserted into the UNLisp file. If TRUE, the messages are sent to the TTY and are also inserted into the UNLisp file.

TR_FLOAT_CAN_BRANCH_COMPLEX [*variable, default: TRUE*]

States whether the arc functions might return complex results. The arc functions are SQRT, LOG, ACOS, etc. When it is TRUE then ACOS(X) will be of mode ANY even if X is of mode FLOAT. When FALSE then ACOS(X) will be of mode FLOAT if and only if X is of mode FLOAT.

TR_FUNCTION_CALL_DEFAULT [*variable, default: GENERAL*]

FALSE means give up and call MEVAL, EXPR means assume Lisp fixed arg function. GENERAL, the default gives code good for MEXPRS and MLEXPRS but not MACROS. GENERAL assures variable bindings are correct in compiled code. In GENERAL mode, when translating F(X), if F is a bound variable, then it assumes that APPLY(F,[X]) is meant, and translates a such, with appropriate warning. There is no

need to turn this off. With the default settings, no warning messages implies full compatibility of translated and compiled code with the **Maxima** interpreter.

TR_GEN_TAGS [*variable, default: FALSE*]

If TRUE, TRANSLATE_FILE generates a Tags file for use by the text editor.

TR_NUMER [*variable, default: FALSE*]

If TRUE numer properties are used for atoms which have them, e.g. %PI.

TR_OPTIMIZE_MAX_LOOP [*variable, default: 100*]

The maximum number of times the macro-expansion and optimization pass of the translator will loop in considering a form. This is to catch macro expansion errors, and non-terminating optimization properties.

TR_OUTPUT_FILE_DEFAULT [*variable, default: TRLISP*]

This is the second file name to be used for translated lisp output (on øMC).

TR_PREDICATE_BRAIN_DAMAGE [*variable, default: FALSE*]

If TRUE, output possible multiple evaluations in an attempt to interface to the COMPARE package.

TR_SEMICOMPILE [*variable, default: FALSE*]

if TRUE TRANSLATE_FILE and COMPFILE output forms which will be macro expanded, but not compiled into machine code by the lisp compiler.

TR_STATE_VARS [*Variable*]

Default values: [TRANSCOMPILE, TR_SEMICOMPILE, TR_WARN_UNDECLARED, TR_WARN_MEVAL, TR_WARN_FEXPR, TR_WARN_MODE, TR_WARN_UNDEFINED_VARIABLE, × TR_FUNCTION_CALL_DEFAULT, TR_ARRAY_AS_REF, TR_NUMER]. The list of the switches that affect the form of the translated output. This information is useful to system people when trying to debug the translator. By comparing the translated product to what should have been produced for a given state, it is possible to track down bugs.

TR_TRUE_NAME_OF_FILE_BEING_TRANSLATED [*Variable*]

Is bound to the quoted string form of the TRUE name of the file most recently translated by TRANSLATE_FILE.

TR_VERSION [*Variable*]

The version number of the translator.

`TR_WARNINGS_GET` () [*Function*]

Prints a list of warnings which have been given by the translator during the current translation.

`TR_WARN_BAD_FUNCTION_CALLS` [*variable, default: TRUE*]

Gives a warning when when function calls are being made which may not be correct due to improper declarations that were made at translate time.

`TR_WARN_FEXPR` [*variable, default: COMPFILE*]

Gives a warning if any FEXPRs are encountered. FEXPRs should not normally be output in translated code, all legitimate special program forms are translated.

`TR_WARN_MEVAL` [*variable, default: COMPFILE*]

Gives a warning if the function MEVAL gets called. If MEVAL is called that indicates problems in the translation.

`TR_WARN_MODE` [*variable, default: ALL*]

Gives a warning when variables are assigned values inappropriate for their mode.

`TR_WARN_UNDECLARED` [*variable, default: COMPILE*]

Determines when to send warnings about undeclared variables to the TTY.

`TR_WARN_UNDEFINED_VARIABLE` [*variable, default: ALL*]

Gives a warning when undefined global variables are seen.

`TR_WINDY` [*variable, default: TRUE*]

Generate helpful comments and programming hints.

13.2.2 Optimizing

When using TRANSLATE and generating code with **Maxima**, there are a number of techniques which can save time and be helpful. In particular, the function FLOATDEFUNK from '`transl/optimu.mc`' creates a function definition from a math-like expression, but it optimizes it (with OPTIMIZE) and puts in the MODE_DECLAREations needed to COMPILE correctly. (This can be done by hand, of course). There is a demo in the file: '`demo/optimub.dem`'.

`OPTIMIZE` (*exp*) [*Function*]

returns an expression that produces the same value and side effects as *exp* but does so more efficiently by avoiding the recomputation of common subexpressions. OPTIMIZE also has the side effect of collapsing its argument so that all common subexpressions are shared.

`COLLAPSE` (*exp*) [*Function*]

collapses its argument by causing all of its common (i.e. equal) subexpressions to share (i.e. use the same cells), thereby saving space. (COLLAPSE is a subroutine used by the OPTIMIZE command.) Thus, calling COLLAPSE may be useful before using FASSAVE or after loading in a SAVE file. You can collapse several expressions together by using COLLAPSE([expr1,...,exprN]);. Similarly, you can collapse the elements of the array A by doing COLLAPSE(LISTARRAY('A));.

`OPTIMPREFIX` [*variable, default: %*]

The prefix used for generated symbols by the OPTIMIZE command.

13.3 Compiling

`COMPILE` (*fun1, fun2, ...*) [*Special Form*]

The COMPILE command is a convenience feature in **Maxima**. It handles the calling of the function COMPFILE, which translates **Maxima** functions into lisp, the calling of the lisp compiler on the file produced by COMPFILE, and the loading of the output of the compiler, know as a FASL file, into the **Maxima**. It also checks the compiler comment listing output file for certain common errors. All these things can be done manually of course, but I have found that using COMPILE, with its convenient default actions, make some work go much faster, mainly use of PLOT2 and numerical integration. COMPILE*()*; causes **Maxima** to prompt for arguments.

COMPILE*(fun1, fun2, ...)* compiles the functions, it uses the name of *fun1* as the first name of the file to put the lisp output. COMPILE(ALL); or COMPILE('FUNCTIONS); will compile all functions. The arguments to COMPILE are evaluated, therefore COMPILE(FUNCTIONS) will not work. You must do COMPILE('FUNCTIONS); This is so that COMPILE works when called inside a program the same as it does when called on a C-line. See section 13.3.1 [Compiler Declarations], page 192 for the definition of FLOATDEFUNK.

`COMPFILE` (*[file], f1, f2, ..., fn*) [*Special Form*]

Compiles functions *fi* in the file *file*.

`COMPILE_LISP_FILE` (*"input-filename"*) [*Function*]

which takes an optional second argument of *output-filename*, can be used in conjunction with TRANSLATE_FILE("filename"). For convenience you might define

```
COMPILE\_AND\_LOAD(FILENAME):=
LOAD(COMPILE\_LISP\_FILE(TRANSLATE\_FILE(FILENAME)[2]))[2]);
```

These file-oriented commands are to be preferred over the use of COMPILE, COMPFILE, and the TRANSLATE SAVE combination.

`COMPGRIND` [*variable, default: FALSE*]

when `TRUE` function definitions output by COMPFILE are pretty-printed.

13.3.1 Compiler Declarations

In order to get the most out of compilation, declarations about what things are, need to be made. The basic idea of compilation is for the computer to make certain calculations only once, that is, at compile time. The not-so-basic ideas are optimization and variable binding semantics, which we can talk about some other time.

For example: `F`(X,Y):=X*Y; If what you want to do is call `F`(3.2,2.2) then there is some inefficiency, because you could have also done `F`('X,'X) and expected to receive back X**2, and it takes many times longer for **Maxima** to figure out that it needs to call the floating point multiply instruction, than it does to execute this instruction.

`F`(X,Y):=(MODE_DECLARE(X,FLOAT,Y,FLOAT),X*Y); tells **Maxima** that `X` and `Y` are the equivalent of the Fortran REAL, and when you COMPILE or TRANSLATE the function, it can use that information to decide to use the multiply instruction.

In general, the MODE_DECLARE should appear as the first function call where ever new variables are introduced. Variables should be MODE_DECLAREd as soon as they are introduced, not before, or after. By introduced, we mean the lexical contour in which they appear. The following things, and only these things, mark the beginning of a contour:

`:=` starts a contour for all the formal parameters, and for all free variables used in the function.

BLOCK starts a contour for all of the block variables.

DO starts a contour for its FOR variable, presently the FOR variable should be declared in the next outer contour though, this bug will be soon fixed. Then a proper example would be `FOR` X:X1 THRU XN STEP DX DO(MODE_DECLARE(X,FLOAT),SUM:SUM+F(X)*DX)

LAMBDA starts a contour for its lambda variables.

Certain constructs, such as SUM, have an implied contour of limited scope. e.g. `SUM`(J**2,J,1,N). In this case, it is not the responsibility of the user to declare the mode of the variable `J`, which is bound by the construct. The code which handles the SUM can look at those modes of the lower and upper limits and infer the mode of `J`.

```
F(X,Y):=(MODE_DECLARE([X,Y],FLOAT),SIN(4*X)+SQRT(1+X^2)*COS(X));
F(X,Y):=(MODE_DECLARE([X,Y,A,B],FLOAT,N,INTEGER), SQRT(A*B*X+Y^N));
F(X):=(MODE_DECLARE(X,FLOAT),
       BLOCK([P,Q],
             MODE_DECLARE([P,Q],FLOAT),
             P:SIN(X)*X^2,
             Q:4*X^2-X+14,
             IF X>0 THEN SQRT(P^2+ABS(P*Q)+SIN(Q)+1)
                    ELSE P+Q/10));

/* dY/dX=F(X,Y), [X0,Y0] and X_FIN and DX */
```

Declaring that a function will return a floating point number:

```
EULER(F,X0,Y0,X_FIN,DX):=
     (MODE_DECLARE([X0,Y0,X_FIN,DX],FLOAT,FUNCTION(F),FLOAT),
      BLOCK([Y:Y0,X:X0],
            MODE_DECLARE([Y,X],FLOAT),
            LOOP, IF X>=X_FIN THEN RETURN(Y),
            Y:Y+DX*APPLY(F,[X]),   X:X+DX,
            GO(LOOP)));
```

Or, without the GO:

```
EULER(F,X0,Y0,X_FIN,DX):=
     (MODE_DECLARE([X0,Y0,X_FIN,DX],FLOAT,FUNCTION(F),FLOAT),
      BLOCK([Y:Y0],
            MODE_DECLARE([Y,X],FLOAT),
            FOR X:X0 THRU X_FIN STEP DX
               DO Y:Y+DX*APPLY(F,[X]),
            Y);
```

Declaring arrays, for example, A is an array that returns some number:

```
SUM_ARRAY(A):=(MODE_DECLARE(ARRAY(A),NUMBER),
       IF LENGTH(ARRAYDIMS(A))#1 THEN ERROR("Takes only 1 dim array"),
       BLOCK([N:ARRAYDIMS(A)[1],SUM:0],
       MODE_DECLARE([N,J],FIXNUM,SUM,NUMBER),
       FOR J:1 THRU N DO SUM:SUM+A[J],
           SUM));
```

FLOATDEFUNK (*'fun, ['var], exp*) [*Function*]

is a utility for making floating point functions from mathematical expression. It will take the input expression and FLOAT it, then OPTIMIZE it, and then insert MODE_DECLAREations for all the variables. This is *the* way to use ROMBERG, fPLOT2, INTERPOLATE, etc. e.g. EXP:expression; FLOATDEFUNK('F,['X],EXP); will define the function F(X) for you.

CHAPTER

FOURTEEN

Plotting and Graphing

Maxima has a large number of PLOT commands, ranging from simple character plots of specified functions and data points to plotting 3 dimensional surfaces. Tektronix 4010 and 4013, Imlac PDS 1 and PDS 4 (using ARDS Graphics conventions), the XGP and the Gould Line printer (at MIT), as well as printing and display consoles are supported by the plot packages. See PLOT, PLOT2, PLOT3D, GRAPH, GRAPH2, GRAPH3D, MULTIGRAPH, PARAMPLOT, and PARAMPLOT2 for details, as well as the file '`share/plot2.usg`'.

14.1 Character Plotting

`PLOT` (*exp, var, low, high*) [*Special Form*]

produces a character-plot of the expression *exp* as *var* (the independent variable) ranges from *low* to *high*. An optional fifth argument of `I`NTEGER causes PLOT to choose only integer values for *var* in the given domain. For graphics terminals there are routines which do more sophisticated plots; see PLOT2. There is a demo in the file: '`demo/plot.dem`'.

PLOT(*fun(x), x, [x1, x2, x3, . . . , xn]*) Plots the function *fun(x)* for the values *x1, x2, x3, . . . , xn*.

`PARAMPLOT` (*f1(t), f2(t), t, low, high*) [*Special Form*]

plots the plane curve *f(t) = (f1(t), f2(t))* with parameter *t*. The syntax is basically like that of PLOT. For example,

```
PARAMPLOT(COS(T),SIN(T),T,0,2*\%PI)
PARAMPLOT([f1(t), g1(t), \dots, h1(t)],
        [f2(t), g2(t), \dots, h2(t)], t, low, high,
                [list of plotting characters])
```

plots the plane curves *f(t) = (f1,f2), g(t) = (g1,g2),. . . ,h(t) = (h1,h2)* using the specified plotting characters, or the default `*`. For example,

```
PARAMPLOT([COS(T),COS(T)+7],[SIN(T),SIN(T)],
                T,0,2*\%PI,["\"])
```

plots two circles.

14.1.1 Character Plotting Flags

`XAXIS` [*variable, default: FALSE*]

if set to `TRUE` will cause the `Y=0` axis to be displayed in PLOT commands.

`YAXIS` [*variable, default: FALSE*]

If set to `TRUE` will cause the `X=0` axis to be displayed in PLOT commands.

`PLOT` [*Variable*]

the height of the area used for plotting, in terms of characters.

14.2 Character Graphing

`GRAPH` (*xlist, ylist, xlabel, ylabel*) [*Special Form*]

graphs the two lists of data points, and labels the axes as indicated or omits labels if just the first two arguments are given.

GRAPH*([x1, x2, x3, . . . , xn], [y1, y2, y3, . . . , yn])* graphs the two sets of data points.

GRAPH*([[x1, y1], [x2, y2], . . . , [xn, yn]])* graphs the points specified by the list of coordinate pairs.

GRAPH(*xset, [yset1, yset2,. . . , ysetn],optional-args*) allows graphing of one x-domain with several y-ranges; e.g.

```
GRAPH([0,1],[[0,1],[1,2]],["\&"]).
```

There is a demo in the file: '`demo/graph.dem`'.

`MULTIGRAPH` (*[[xset1, yset1], . . . , [xsetn, ysetn]], optional-args*) [*Special Form*]

allows the user to produce a scatter-graph involving several x-domains each with a single y-range; e.g. MULTIGRAPH([[[0,1],[0,1]],[[3,4],[1,2]]],["&"]).

`GRAPH3D` (*x-lists, y-lists, z-lists, optional-args*) [*Special Form*]

takes 3 arguments (GRAPH2 takes 2), and interprets them as lists of `x`, `y`, and `z` points which it uses to draw lines using the 3d transformations. It can be used to add lines (e.g. axes) to your 3D plot. The hidden line routines are not used.

14.3 2D Plotting

The capabilities of the routines described here include plotting of several curves on a single graph, plotting several graphs in different positions on the screen, saving plots, replotting plots with different scales without having to recompute any points, plotting of 3 dimensional surfaces, plotting of user defined dashed lines and symbols.

The devices supported are: The Tektronix 4010, 4013, 4025 and 4662, the Imlac PDS 1 and PDS 4 (using ARDS graphics conventions), the XGP, the Gould line-printer (in 38-246), the Dover, some Versatecs on the MFE-NET and in a preview mode printing and display consoles.

PLOT2 (*y-exprs, variable, var-range, optionals-args*) [*Special Form*]

plots *y-exprs* on the y axis as variable (the x axis) takes on values specified by *var-range*. *y-exprs* can take one of two forms:

1. *exp* plots a curve of *exp* against variable
2. *[expr1,expr2, .., expri, .. ,exprn]* plots *n* curves of *expri* against variable. *expri* gets evaluated in the context FLOAT(EV(expri,variable=value gotten from var-range, NUMER)). It is an error if this doesn't result in a floating point number.

var-range can have the following forms:

1. *low, high* where *low* and *high* evaluate to numbers. *low* may be either greater or less than *high*. variable will take on PLOTNUM values equally spaced between *low* and *high*.

   Note that the first argument will be evaluated at *low* first e.g. PLOT2(1/X, X, -1, -3); calculates 1/(-1.0) before 1/(-3.0). This will only make a difference if the computation of the first argument changes a variable which changes the value returned by subsequent computation. Whether or not *low* < *high*, *min(low, high*) will be on the left of the plot.
2. low,*high*,INTEGER. As in (1), except variable will take on all integer values between *low* and *high* inclusive.
3. *[val1,val2, . . . , valn]* variable takes the values specified by the list.
4. *arrayname* where *arrayname* is the name of a declared floating-point one-dimensional array (i.e. declared by ARRAY(arrayname,FLOAT, max-index);). *variable* takes the values from arrayname[0] thru arrayname[max-index] (max-index is the maximum index of arrayname.

optional-args can be any of the following:

1. X-Label, Y-Label or Title descriptor
2. Line type descriptor
3. FIRST, SAME and LAST
4. POLAR, LOG, LINLOG, LOGLIN

The optional arguments may appear in any order. The rule for evaluation of the optional args is as follows. If the argument is atomic it gets evaluated. The resulting arguments are the ones that get used. If you want to plot more than 3-4 curves on the same plot investigate using the NOT3D option to PLOT3D. Examples:

```
PLOT2(SIN(X),X,-\%PI,\%PI);     plots sin(X) against X as X takes on
                                \vr{PLOTNUM} values between -\%PI to \%PI
PLOT2(X!,X,0,6,INTEGER);        plots X! as X takes integral values
                                between 0 and 6
F(X):=SQRT(X);
PLOT2(F(X),X,[-2,3,100.12]);    plots F(X) as X takes the values in the
                                values in the list
PLOT2([X+1,X^2+1],X,-1,1);      plots 2 curves on top of each other
```

PLOT2*(y-funs, var-range, optionals-args)* is the alternative form for PLOT2. *y-funs* must be a function of one argument or a list of functions of one argument. The functions must be either translated or compiled functions which return a floating point number when it is given floating point arg (or integer arg if the INTEGER arg to PLOT2 is given). This form of PLOT2 acts as though you had given a argument to the *y-funs*, and also specified that argument as the variable in the form above. E.g. PLOT2(F,-2,2); acts like PLOT2(F(X),X,-2,2);. This is supposed to provide a quicker evaluation of the first arg, and for that reason no checking is done on the result. If the wrong kind of number is returned, the resulting plot will not be meaningful.

```
TRANSLATE:TRUE;
F(X):=(MODE_DECLARE(X,FLOAT),EXP(-X*X));
PLOT2(F,-2,2);
PLOT2(F,[-2,-1,0,1,2]);
ARRAY(V,FLOAT,10);
FOR I FROM 0 THRU 10 DO V[I]:FLOAT(I*I);
PLOT2(F,V);
```

There is a demo in the file: '`demo/plot2.dem`'. There is a demo in the file: '`share/plot2.usg`'.

PARAMPLOT2 (*x-exprs, y-exprs, variable, var-range, optional-args*) [*Special Form*]

plots *x-exprs* as the x coordinate against *y-exprs* as the y coordinate. The format for the first two arguments is the same as that for the first argument to PLOT2. Thus if *x-exprs* is *[x-expr1, x-expr2, . . . , x-expri, . . . ,x-exprn]*, and *y-exprs* is *[y-expr1, y-expr2, . . . ,y-expri,. . . ,y-exprk]*, then *max(n,k)* curves will be plotted. They will be (assuming n > k): *x-expr1 vs. y-expr1, . . . , x-exprk vs. y-exprk, x-expr(k+1) vs. y-exprk, . . . , x-exprn vs. y-exprk*. The format for the remaining arguments is the same as for PLOT2.

PARAMPLOT2*(x-funs.y-funs,var-range,optional-args)* efficiently evaluates its first 2 arguments in the same way that the alternative form of PLOT2 works.

Examples:

```
TRANSLATE:TRUE;       /* causes automatic translation */
F(X):=(MODE_DECLARE(X,FLOAT),COS(X));
G(X):=(MODE_DECLARE(X,FLOAT),SIN(X));
PARAMPLOT2(F,G,0,2*\%PI);
```

plots F(x) vs G(x) as x goes from 0 to 2*%PI

```
PARAMPLOT2(SIN(T),COS(T),T,0,2*\%PI);
```

plots Sin(T) for the x-axis and Cos(T) for the y-axis as T takes on PLOTNUM values between 0 and 2*%PI. If EQUALSCALE is TRUE, this draws a circle.

PLOTNUM default: [20] is the number of points PLOT2 and PARAMPLOT2 plot when given the *low*,*high* type of variable *range*. The default when PLOT2 is first called is 20, which is sufficient for trying things out. 100 is a suitable value for a final hard copy.

14.4 2D Graphing

GRAPH2 (*x-lists*,*y-lists*,*optional-args*) [*Special Form*]

plots points specified by the first *x-lists* and *y-lists*.

The format for *x-lists* can be one of

1. *[x-pt1, . . . , x-ptn]* where *x-pti* each evaluates to a number
2. arrayname where *arrayname* is the name of a declared one-dimensional array of floating point numbers
3. 2d-arrayname where *2d-arrayname* is the name of a declared two-dimensional array of floating point numbers (i.e. by ARRAY(2d-arrayname, FLOAT, max-row-index, max-col-index);)
4. [x-list1,x-list2, . . . , x-listi, . . . , x-listk] where x-listi can have the form of either (1) or (2).

The format of *y-lists* is similar. The format of *optional-args* is the same as for PLOT2. Note that GRAPH2 is more similar to the **Maxima** function MULTIGRAPH than to GRAPH.

14.5 3D Plotting

PLOT3D (*z-exprs, x-var, var-range, y-var, var1-range, optional-args*) [*Special Form*]

makes a 3-dimensional plot of *z-exprs* against *x-var* and *y-var*. The plot consists of curves of *y-exprs* against *x-var* (the x coordinate) with *y-var* (the y coordinate) held fixed. Perspective is used and curves further away from the viewer have those parts of them which are hidden by the closer curves removed.

The format of *y-exprs* is the same as for PLOT2. The context of evaluation is *FLOAT(EV(expri,x-var=value gotten from var-range,y-var=value gotten from var1-range,NUMER))*.

The format for *var-range* and *var1-range* is the same as for PLOT2 except that if *var1-range* is of the form *low*,*high* then *y-var* will take on PLOTNUM1 values.

The format of *optional-args* is the same as for PLOT2 except that additional options NOT3D, 3D and CONTOUR are available.

PLOT3D*(z-funs, var-range, var1-range, optional-args)* is analogous to the alternative form for PLOT2. *z-funs* must be a function or list of functions of 2 arguments, which must return a floating point argument when given floating point (integer, if the INTEGER argument is used for either *var-range* or *var1-range*) arguments. The functions must be translated or compiled. If you expect to make several 3D plots this form is recommended.

A simple example is

```
TRANSLATE:TRUE;                    / *causes automatic translation */
G(X,Y):=(MODECLARE(X,FLOAT),
         EXP(-X*X-Y*Y));           /* defines a funtion G */
PLOT3D(G,-2,2,-2,2);               /* plots it */
```

14.5.1 Plotting Flags

`3D` [*Variable*]

in the plotting functions, indicates a 3 Dimensional plot will be made.

`NOT3D` [*Variable*]

The addition of NOT3D as an argument to PLOT3D causes exactly the same points as in the bare PLOT3D to be calculated. Instead of plotting a 3-dimensional representation of the data, the data is plotted in a 2D one. Specifically 1 2D curve of `z` vs. `x` for each `y` value, and so is a convenient way to plot several curves on the same plot.

`CONTOUR` [*Variable*]

An option in the PLOT3D package to allow contour plotting.

`PERSPECTIVE` [*variable, default: TRUE*]

In the plotting functions, if PERSPECTIVE is set to `F`ALSE, it causes a non-perspective view to be taken. This is equivalent to extending the viewing position out to infinity along a line connecting the origin and VIEWPT.

`REVERSE` [*variable, default: FALSE*]

in the plotting functions, if `T`RUE it causes a left-handed coordinate system to be assumed.

`PLOTBELL` [*variable, default: TRUE*]

when `F`ALSE inhibits the dinging of the bell.

PLOTBOTMAR [*variable, default: 320*]

adjusts the bottom margin for XGP plots. This defaults to values (in increments of 1/200 inch) such that the plots will fit comfortably on an 8 1/2 x 11 page.

PLOTLFTMAR [*variable, default: 128*]

adjusts the left margin for XGP plots. This defaults to values (in increments of 1/200 inch) such that the plots will fit comfortably on an 8 1/2 x 11 page.

PLOTNUM [*variable, default: 20*]

is the number of points PLOT2 and PARAMPLOT2 plot when given the *low*, *high* type of variable range. The default when PLOT2 is first called is 20, which is sufficient for trying things out. 100 is a suitable value for a final hard copy.

CENTERPLOT [*Variable*]

VIEWPT and CENTERPLOT determine the perspective view taken for plotting commands. They are defaulted to be unbound. VIEWPT may be set to a list of 3 numbers and gives the point from which the projection should be made. CENTERPLOT may likewise be set to a list of 3 numbers and gives a point on the line of sight. The projection will be made onto a plane perpendicular to a line joining VIEWPT and CENTERPLOT.

If VIEWPT and CENTERPLOT are unbound (the default), then they will be chosen as follows: the extreme values of the coordinates are determined. This gives the two points min: [xmin, ymin, zmin], max:[xmax, ymax, zmax]. CENTERPLOT is chosen as (min+max)/2, and VIEWPT is chosen as max+3*(max-min). The view is then one in which the z axis is vertical, the x axis is increasing towards you to the left and the y axis is increasing towards you to the right. If CENTERPLOT is FALSE then the old type of perspective view will be given (like setting the x and z components of CENTERPLOT to the corresponding components of VIEWPT).

VIEWPT [*Variable*]

VIEWPT and CENTERPLOT determine the perspective view taken for plotting commands. They are defaulted to be unbound. VIEWPT may be set to a list of 3 numbers and gives the point from which the projection should be made. CENTERPLOT may likewise be set to a list of 3 numbers and gives a point on the line of sight. The projection will be made onto a plane perpendicular to a line joining VIEWPT and CENTERPLOT.

If VIEWPT and CENTERPLOT are unbound (the default), then they will be chosen as follows: the extreme values of the coordinates are determined. This gives the two points *min:[xmin, ymin, zmin], max:[xmax, ymax, zmax]*. CENTERPLOT is chosen as (min+max)/2, and VIEWPT is chosen as *max+3*(max-min)*. The view is then one in which the z axis is vertical, the x axis is increasing towards you to the left and the y axis is increasing towards you to the right.

If CENTERPLOT is FALSE then the old type of perspective view will be given (like setting the x and z components of CENTERPLOT to the corresponding components of VIEWPT).

DOVARD_VIEWPORT [*variable, default: [1,7,1,7]*]

determines the perspective for Dover output plots, similar to VIEWPT. It accepts 4 arguments, [XMIN,XMAX,YMIN,YMAX] in inches on the page.

CHAPTER

FIFTEEN

Numeric Interface

15.1 Generating Fortran Code

FORTRAN (*exp*) [*Special Form*]

converts *exp* into a FORTRAN linear expression in legal FORTRAN with 6 spaces inserted at the beginning of each line, continuation lines, and ** for exponentiation. When FORTSPACES default:[FALSE] is TRUE, the FORTRAN command fills out to 80 columns using spaces.

If FORTRAN is called on a bound symbolic atom, e.g. FORTRAN(X); where X:A*B has been done, then X=value of X, e.g. X=A*B will be generated. In particular, if e.g. M:MATRIX(...); has been done, then FORTRAN(M); will generate the appropriate assignment statements of the form name(i,j)=<corresponding matrix element>.

FORTSPACES [*variable, default: FALSE*]

if TRUE, the FORTRAN command fills out to 80 columns using spaces.

FORTINDENT [*variable, default: 0*]

controls the left margin indentation of expressions printed out by the FORTRAN command. 0 gives normal printout (i.e. 6 spaces), and positive values will causes the expressions to be printed farther to the right.

15.2 Numerical Integration

15.2.1 Romberg Integration

ROMBERG (*exp, var, ll, ul*) [*Function*]

There are two ways to use this function. The first is an inefficient way like the definite integral version of INTEGRATE:

```
ROMBERG(<integrand>,<variable of integration>,<lower limit>,
                <upper limit>);

        ROMBERG(SIN(Y),Y,1,\%PI);
                TIME= 39 MSEC.          1.5403023
        F(X):=1/(X^5+X+1);
        ROMBERG(F(X),X,1.5,0);
                TIME= 162 MSEC.         - 0.75293843
```

The second is an efficient way that is used as follows: Example:

```
ROMBERG(<function name>,<lower limit>,<upper limit>);

        F(X):=(MODE_DECLARE([FUNCTION(F),X],FLOAT),1/(X^5+X+1));
        TRANSLATE(F);
        ROMBERG(F,1.5,0);
                TIME= 13 MSEC.          - 0.75293843
```

The first argument must be a TRANSLATEd or compiled function. (If it is compiled, it must be declared to return a FLONUM.) If the first argument is not already TRANSLATEd, ROMBERG will not attempt to TRANSLATE it but will give an error.

The accuracy of the integration is governed by the global variables ROMBERGTOL default: [1.E-4] and ROMBERGIT default: [11]. ROMBERG will return a result if the relative difference in successive approximations is less than ROMBERGTOL. It will try halving the stepsize ROMBERGIT times before it gives up. The number of iterations and function evaluations which ROMBERG will do is governed by ROMBERGABS and ROMBERGMIN.

ROMBERG may be called recursively and thus can do double and triple integrals.

Example:

```
INTEGRATE(INTEGRATE(X*Y/(X+Y),Y,0,X/2),X,1,3);
                        13/3 (2 LOG(2/3) + 1)
\%,NUMER;
                        0.81930233
DEFINE_VARIABLE(X,0.0,FLOAT,"Global variable in function F")$
F(Y):=(MODE_DECLARE(Y,FLOAT), X*Y/(X+Y) )$
G(X):=ROMBERG('F,0,X/2)$
ROMBERG(G,1,3);
                                0.8193023
```

The advantage with this way is that the function F can be used for other purposes, like plotting. The disadvantage is that you have to think up a name for both the function F and its free variable X. Or, without the global:

```
G1(X):=(MODE_DECLARE(X,FLOAT), ROMBERG(X*Y/(X+Y),Y,0,X/2))$
ROMBERG(G1,1,3);
                        0.8193023
```

The advantage here is shortness.

```
Q(A,B):=ROMBERG(ROMBERG(X*Y/(X+Y),Y,0,X/2),X,A,B)$
Q(1,3);
                              0.8193023
```

It is even shorter this way, and the variables do not need to be declared because they are in the context of ROMBERG.

Use of ROMBERG for multiple integrals can have great disadvantages, though. The amount of extra calculation needed because of the geometric information thrown away by expressing multiple integrals this way can be incredible. The user should be sure to understand and use the ROMBERGTOL and ROMBERGIT switches.

ROMBERGABS [*variable, default: 0.0*]

Assuming that successive estimates produced by ROMBERG are Y[0], Y[1], Y[2] etc., then ROMBERG will return after N iterations if (roughly speaking)

```
(ABS(Y[N]-Y[N-1]) <= ROMBERGABS OR
ABS(Y[N]-Y[N-1])/(IF Y[N]=0.0 THEN 1.0 ELSE Y[N]) <= ROMBERGTOL)
```

is TRUE. The condition on the number of iterations given by ROMBERGMIN must also be satisfied. Thus if ROMBERGABS is 0.0 (the default), you just get the relative error test. The usefulness of the additional variable comes when you want to perform an integral, where the dominant contribution comes from a small region. Then you can do the integral over the small dominant region first, using the relative accuracy check, followed by the integral over the rest of the region using the absolute accuracy check. Suppose you want to compute Integral(exp(-x),x,0,50) (numerically) with a relative accuracy of 1 part in 10000000.

```
/* Define the function.  N is a counter, so we can see how many
   function evaluations were needed. */
F(X):=(MODE_DECLARE(N,INTEGER,X,FLOAT),N:N+1,EXP(-X))$
TRANSLATE(F)$
/* First of all try doing the whole integral at once */
BLOCK([ROMBERGTOL:1.E-6,ROMBERABS:0.],N:0,ROMBERG(F,0,50));
   ==> 1.00000003
N; ==> 257  /* Number of function evaluations*/
/* Now do the integral intelligently, by first doing
     Integral(exp(-x),x,0,10) and then setting ROMBERGABS to 1.E-6*(this
     partial integral).  */
BLOCK([ROMBERGTOL:1.E-6,ROMBERGABS:0.,SUM:0.],
      N:0,SUM:ROMBERG(F,0,10),ROMBERGABS:SUM*ROMBERGTOL,ROMBERGTOL:0.,
      SUM+ROMBERG(F,10,50));  ==> 1.00000001  /* Same as before */
N;  ==> 130
```

So if F(X) were a function that took a long time to compute, the second method would be about 2 times quicker.

ROMBERGIT [*variable, default: 11*]

The accuracy of the ROMBERG integration command is governed by the global variables ROMBERGTOL default: [1.E-4] and ROMBERGIT default: [11]. ROMBERG will return a result if the relative difference

in successive approximations is less than ROMBERGTOL. It will try halving the stepsize ROMBERGIT times before it gives up.

ROMBERGMIN [*variable, default: 0*]

governs the minimum number of function evaluations that ROMBERG will make. ROMBERG will evaluate its first argument at least 2**(ROMBERGMIN+2)+1 times. This is useful for integrating oscillatory functions, when the normal converge test might sometimes wrongly pass.

ROMBERGTOL [*variable, default: 1.E-4*]

The accuracy of the ROMBERG integration command is governed by the global variables ROMBERGTOL default: [1.E-4] and ROMBERGIT default: [11]. ROMBERG will return a result if the relative difference in successive approximations is less than ROMBERGTOL. It will try halving the stepsize ROMBERGIT times before it gives up.

15.2.2 Newton-Coates Integration

QUANC8 (*'fun, lo, hi*) [*Function*]

The file '`share1/qq.mc`' (which may be loaded with LOAD("QQ");) contains a function QUANC8 which can take either 3 or 4 arguments. The 3 argument version computes the integral of the function specified as the first argument over the interval from *lo* to *hi* as in QUANC8('fun, lo, hi);. The function name should be quoted. The 4 arg version will compute the integral of the function or expression (first arg) with respect to the variable (second arg) over the interval from *lo* to *hi* as in QUANC8*(f(x), x, lo, hi)*.

The method used is the Newton-Cotes 8th order polynomial quadrature, and the routine is adaptive. It will thus spend time dividing the interval only when necessary to achieve the error conditions specified by the global variables QUANC8_RELERR default: 1.0e-4 and QUANC8_ABSERR default : 1.0e-8 which give the relative error test: |integral(function)-computed value| < quanc8_relerr*|integral(function)| and the absolute error test: |integral(function)-computed value|<quanc8_abserr.

The error from each subinterval is estimated and the contribution from a subinterval is accepted only when the integral over the subinterval satisfies the error test over the subinterval. The total estimated error of the integral is contained in the global variable QUANC8_ERREST default: [0.0].

QUANC8_FLAG [*variable, default: 0.0*]

will contain valuable information if the computation fails to satisfy the error conditions. The integer part will tell you how many subintervals failed to converge and the fractional part will tell you where the singular behavior is, as follows: singular point=lo+(1.-frac part)*(hi-lo). Thus quanc8(tan(x),x,1.57,1.6); gives frac=.97 so trouble is at 1.57+.03*.03=1.5709 (=pi/2). If QUANC8_FLAG is not 0.0, you should be cautious in using the return value, and should try ROMBERG or a Simpson method and see if the result checks. Analysis of possible singular behavior might be advisable. You may get QUANC8_FLAG=<integer>.0 and an error message (such as division by 0) when a singular point is hit in the interval. You will have to find the singularity and eliminate it before QUANC8 will get an answer. Functions which have very large derivatives may throw the error estimate way off and cause the wrong points to be used, and a wrong answer

returned. Try romberg(exp(-.002*x**2)*cos(x)**2,x,0.,100.); with the default tolerance, and quanc8(exp(-.002*x**2)*cos(x)**2,x,0.,100.); with quanc8_relerr=1.e-7 and 1.e-8. The last result is consistent with ROMBERG, while the previous one is off by a factor of 2 ! This is due to the bad behavior of the derivatives near x=10.0 which cause the adaptive routine to have trouble. If you use quanc8('f,a,c)+quanc8('f,c,b) where a<c<b, you will do better in such cases.

You can do DEMO("QQ"); for some comparisons with the ROMBERG numerical integrator (which is not adaptive). Note that ROMBERG usually gives more accurate answers for comparable tolerances, while QUANC8 will get the same answer faster even with a smaller tolerance, because ROMBERG subdivides the whole interval if the total result is not within error tolerance, while QUANC8 improves only where needed, thus saving many function calls. ROMBERG will also fail to converge when oscillatory behavior is overwhelming, while QUANC8 will adapt in the regions as it sees fit. (The global variable ROMBERGMIN is designed to allow you a minimum number of function calls in such cases, so that exp(-x)*sin(12*x) can be integrated from 0 to 4*%pi without erroneously giving 0. from the first few function calls.)

To make your **Maxima** user function callable in the fastest way, you must use MODE_DECLARE and then translate and compile the function. The speed of the computation may be increased by well over an order of magnitude when compilation is used. If you do multiple integrals, it is really necessary to compile the function in order to avoid the time spent on function calls. A sample use of QUANC8 for a double integral is in the demo file, and compilation is nearly a hundred times faster in doing the work! There is a demo in the file: '`share1/qq.dem`'. There are some usage notes in the file: '`share1/qq.usg`'.

15.3 IMSL Routines

IMSL-based routines currently return [SUCCESS, ...] or [ERROR, ...]. They will soon be changed to return [SUCCESS, ...], [WARNING, ...] and [ERROR, ...].

This is a numerical integration package using cautious, adaptive Romberg extrapolation. The DCADRE package is written to call the IMSL fortran library routine **DCADRE**. To load this package, do LOAD(IMSL);. The IMSL version of Romberg integration is not available in **Maxima**. The worker function takes the following syntax:

IMSL_ROMBERG(*fun, low, hi*) where *fun* is a function of 1 argument and *low* and *hi* should be the lower and upper bounds of integration. *fun* must return floating point values.

IMSL_ROMBERG*(exp, var, low, hi)* where *exp* should be integrated over the range *var=low* to *hi*. The result of evaluating *exp* must always be a floating point number.

FAST_IMSL_ROMBERG(*fun, low, hi*) This function does no error checking, but may achieve a speed gain over the IMSL_ROMBERG function. It expects that *fun* is a Lisp function (or translated **Maxima** function) which accepts a floating point argument, and that it always returns a floating point value.

Returns either [SUCCESS, answer, error] where answer is the result of the integration and error is the estimated bound on the absolute error of the output, DCADRE, as described in PURPOSE below. or [WARNING, n, answer, error] where *n* is a warning code, answer is the answer, and error is the estimated bound on the absolute error of the output, DCADRE, as described in PURPOSE below.

The following warnings may occur:

65 One or more singularities were successfully handled.

66 In some subinterval(s), the estimate of the integral was accepted merely because the estimated error was

small, even though no regular behavior was recognized.

or [ERROR, errorcode] where error code is the IMSL-generated error code. The following error codes may occur:

132 Failure. This may be due to too much noise in function (relative to the given error requirements) or due to an ill-behaved integrand.

133 RERR is greater than 0.1 or less than 0.0 or is too small for the precision of the machine.

The following flags have an influence upon the operation of IMSL_ROMBERG:

ROMBERG_AERR [*variable, default: 1.0E-5*]

Desired absolute error in answer.

ROMBERG_RERR [*variable, default: 0.0*]

Desired relative error in the answer.

If IMSL signals an error, a message will be printed on the user's console stating the nature of the error. (This error message may be suppressed by setting IMSLVERBOSE to FALSE.)

Because this uses a translated Fortran routine, it may not be recursively invoked.

IMSL ROMBERG Purpose

DCADRE attempts to solve the following problem: Given a real-valued function F of one argument, two real numbers A and B, find a number DCADRE such that:

```
/ B                        [                                  | / B       | ]
[                          [                                  | [         | ]
I F(x)dx - DCADRE <= max [ ROMBERG_AERR, ROMBERG_RERR * | I F(x)dx | ]
]                          [                                  | ]         | ]
/ A                        [                                  | / A       | ]
```

Algorithm (modified version of the IMSL documentation):

This routine uses a scheme whereby DCADRE is computed as the sum of estimates for the integral of F(x) over suitably chosen subintervals of the given interval of integration. Starting with the interval of integration itself as the first such subinterval, cautious Romberg extrapolation is used to find an acceptable estimate on a given subinterval. If this attempt fails, the subinterval is divided into two subintervals of equal length, each of which is considered separately.

Programming Notes (modified version of the IMSL documentation):

1. DCADRE (the translated-Fortran base for IMSL_ROMBERG) can, in many cases, handle jump discontinuities and certain algebraic discontinuities. See reference for full details.

2. The relative error parameter ROMBERG_RERR must be in the interval [0.0,0.1]. For example, ROMBERG_RERR=0.1 indicates that the estimate of the intergral is to be correct to one digit, where

as ROMBERG_RERR=1.0E-4 calls for four digits of accuracy. If DCADRE determines that the relative accuracy requirement cannot be satisfied, IER is set to 133 (ROMBERG_RERR should be large enough that, when added to 100.0, the result is a number greater than 100.0 (this will not be TRUE of very tiny floating point numbers due to the nature of machine arithmetic)).

3. The absolute error parameter, ROMBERG_AERR, should be nonnegative. In order to give a reasonable value for ROMBERG_AERR, the user must know the approximate magnitude of the integral being computed. In many cases, it is satisfactory to use AERR=0.0. In this case, only the relative error requirement is satisfied in the computation. We quote from the reference, "A very cautious man would accept DCADRE only if IER [the warning or error code] is 0 or 65. The merely reasonable man would keep the faith even if IER is 66. The adventurous man is quite often right in accepting DCADRE even if the IER is 131 or 132." Even when IER is not 0, DCADRE returns the best estimate that has been computed.

For references on this technique, see de Boor, Calr, "CADRE: An Algorithm for Numerical Quadrature," Mathematical Software (John R. Rice, Ed.), New York, Academic Press, 1971, Chapter 7.

NDIFFQ is a package residing on the SHARE directory for numerical solutions of differential equations. Do LOAD(NDIFFQ) to load in for use. An example of its use would be: Not available **Maxima**.

```
DEFINE_VARIABLE(N,0.3,FLOAT);
DEFINE_VARIABLE(H,0.175,FLOAT);
F(X,E):=(MODE_DECLARE([X,E],FLOAT),N*EXP(X)/(E+X^(2*H)*EXP(H*X)));
COMPILE(F);
ARRAY([X,E],FLOAT,35);
INIT_FLOAT_ARRAY(X,1.0E-3,6.85); /* Fills X with the interval */
E[0]:5.0;                        /* Initial condition */
RUNGE_KUTTA(F,X,E);              /* Solve it */
GRAPH2(X,E);                     /* Graph the solution */
```

RUNGE_KUTTA(F,X,E,E_Prime) would be the call for a second-order equation. Not available **Maxima**. There is a demo in the file: '`share2/ndiffq.dem`'. There are some usage notes in the file: '`share2/ndiffq.usg`'.

The IMSL ZRPOLY routine for finding the zeros of simple polynomials (single variable, real coefficients, non-negative integer exponents), using the Jenkins-Traub technique. The command is POLYROOTS(polynomial);. Not available in **Maxima**.

For those who can make use of approximate numerical solutions to problems, there is a package which calls a routine which has been translated from the IMSL fortran library to solve N simultaneous non-linear equations in N unknowns. It uses black-box techniques that probably aren't desirable if an exact solution can be obtained from one of the smarter solvers (LINSOLVE, ALGSYS, etc). But for things that the other solvers don't attempt to handle, this can probably give some very useful results. Not available in **Maxima**.

CHAPTER

SIXTEEN

Advanced Packages

The ‘share’ directories contains programs, information files, etc. which are considered to be of interest to the **Maxima** community. Most of these files are not part of the **Maxima** system per se and must be loaded individually by the user, e.g. LOAD(ARRAY);. Many of the files were contributed by **Maxima** users.

16.1 Fast Fourier Transforms

FFT (*real-array, imag-array*) [*Function*]

Fast Fourier Transform. This package may be loaded by doing LOAD(FFT); There is also an IFT command, for Inverse Fourier Transform. These functions perform a (complex) fast fourier transform on either 1 or 2 dimensional floating-point arrays, obtained by: ARRAY(<ary>,FLOAT,<dim1>); or ARRAY(<ary>,FLOAT,<dim1>,<dim2>);. For 1D arrays <dim1> must equal 2**n-1, and for 2D arrays <dim1>=<dim2>=2**n-1 (i.e. the array is square). (Recall that **Maxima** arrays are indexed from a 0 origin so that there will be 2**n and (2**n)**2 arrays elements in the above two cases.) This package also contains two other functions, POLARTORECT and RECTTOPOLAR.

The real and imaginary arrays must of course be the same size. The transforms are done in place so that *real-array* and *imag-array* will contain the real and imaginary parts of the transform. (If you want to keep the transformed and un-transformed arrays separate copy the arrays before calling FFT or IFT using the FILLARRAY function.

The definitions of the Fast Fourier Transform and its inverse are given here. Here *A* is the array to be transformed and *AT* is its transform. Both A and AT are complex arrays, although as noted above FFT and IFT can only deal with separate real arrays for the real and imaginary parts of *A* and *AT*. *N* (or *N**2*) is the number of elements in A in the 1D (or 2D) case. (In fact these definitions are not of the FFTs but of the discrete Fourier transforms. The FFT and IFT functions merely provided efficient algorithms for the implementation of these definitions.)

1D case:

```
       N - 1
       ====                      - 1
       \           2 \%I \%PI I K N
AT  =   >     A   \%E
  K    /       I
       ====
       I = 0

           N - 1
           ====                          - 1
       - 1 \            - 2 \%I \%PI I K N
A   = N     >     AT   \%E
 I         /        K
           ====
           K = 0
```

2D case:

```
           N - 1 N - 1
           ====  ====                                  - 1
           \     \              2 \%I \%PI (I K + J L) N
AT      =   >     >     A      \%E
  K, L     /     /       I, J
           ====  ====
           I = 0 J = 0

               N - 1 N - 1
               ====  ====                                      - 1
           - 2 \     \               - 2 \%I \%PI (I K + J L) N
A      = N     >     >     AT      \%E
 I, J         /     /        K, L
               ====  ====
               K = 0 L = 0
```

Not available in **Maxima** because it depends on a machine-coded FFT routine on the MIT-MC machine. There is a demo in the file: ‘`share/fft.dmo`’. There are some usage notes in the file: ‘`share/fft.usg`’.

`IFT` (*real-array, imag-array*) [*Function*]

Inverse Fourier Transform. Do LOAD(FFT); to load in this package. These functions (FFT and IFT) perform a (complex) fast fourier transform on either 1 or 2 dimensional FLOATING-POINT arrays, obtained by: ARRAY(<ary>,FLOAT,<dim1>); or ARRAY(<ary>,FLOAT,<dim1>,<dim2>);. For 1D arrays <dim1> must equal 2**n-1, and for 2D arrays <dim1>=<dim2>=2**n-1 (i.e. the array is square). (Recall that **Maxima** arrays are indexed from a 0 origin so that there will be 2**n and (2**n)**2 arrays elements in the above two cases.) There is a demo in the file: ‘`share/fft.dmo`’. There are some usage notes in the file: ‘`share/fft.usg`’.

16.2 Tensors

There are two tensor packages in **Maxima**, CTENSR and ITENSR. CTENSR is Component Tensor Manipulation, and may be accessed with LOAD(CTENSR);. ITENSR is Indicial Tensor Manipulation, and is loaded by doing LOAD(ITENSR); A manual for CTENSR AND ITENSR is available from the LCS Publications Office. Request MIT/LCS/TM-167. See the file 'tensor/tensor.doc'. In addition, demos exist in the TENSOR directory under the filenames 'tensor/ctensr.dmX' where X is from 1 to 4, or 'tensor/itensr.dmX' where X is from 1 to 7. Do DEMO("tensor/ctensr.dmX"); where X is from 1 to 4, or DEMO("tensor/itensr.dmX"); where X is from 1 to 7, for a demonstration.

16.2.1 Component Tensor Manipulation

To use the CTENSR package, type TSETUP(); which automatically loads it from within **Maxima** (if it is not already loaded) and then prompts the user to input his coordinate system. The user is first asked to specify the dimension of the manifold. If the dimension is 2, 3 or 4 then the list of coordinates defaults to [X,Y], [X,Y,Z] or [X,Y,Z,T] respectively. These names may be changed by assigning a new list of coordinates to the variable OMEGA (described below) and the user is queried about this. Care must be taken to avoid the coordinate names conflicting with other object definitions.

Next, the user enters the metric either directly or from a file by specifying its ordinal position. As an example of a file of common metrics, see the file 'tensor/metric.fil'. The metric is stored in the matrix LG. Finally, the metric inverse is computed and stored in the matrix UG. One has the option of carrying out all calculations in a power series. A sample protocol is begun below for the static, spherically symmetric metric (standard coordinates) which will be applied to the problem of deriving Einstein's vacuum equations (which lead to the Schwarzschild solution) as an example. Many of the functions in CTENSR will be displayed for the standard metric as examples.

```
(C2) TSETUP();
Enter the dimension of the coordinate system:
4;
Do you wish to change the coordinate names?
N;
Do you want to
1. Enter a new metric?
2. Enter a metric from a file?
3. Approximate a metric with a Taylor series?
Enter 1, 2 or 3
1;
```

```
Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4
1;
Row 1 Column 1:  A;
Row 2 Column 2:  X^2;
Row 3 Column 3:  X^2*SIN(Y)^2;
Row 4 Column 4:  -D;
Matrix entered.
Enter functional dependencies with the DEPENDS function or 'N' if none
DEPENDS([A,D],X);
Do you wish to see the metric?
Y;
                          [ A  0       0        0  ]
                          [                        ]
                          [     2                  ]
                          [ 0  X       0        0  ]
                          [                        ]
                          [          2    2        ]
                          [ 0  0   X  SIN (Y)   0  ]
                          [                        ]
                          [ 0  0       0       - D ]
Do you wish to see the metric inverse?
N;
```

TSETUP () [*Function*]

A function in the CTENSR (Component Tensor Manipulation) package which automatically loads the CTENSR package from within **Maxima** (if it is not already loaded) and then prompts the user to make use of it.

CHRISTOF (*arg*) [*Function*]

A function in the CTENSR (Component Tensor Manipulation) package. It computes the Christoffel symbols of both kinds. The *arg* determines which results are to be immediately displayed. The Christoffel symbols of the first and second kinds are stored in the arrays LCS[i,j,k] and MCS[i,j,k] respectively and defined to be symmetric in the first two indices. If the argument to CHRISTOF is LCS or MCS then the unique non-zero values of LCS[i,j,k] or MCS[i,j,k], respectively, will be displayed. If the argument is ALL then the unique non-zero values of LCS[i,j,k] and MCS[i,j,k] will be displayed. If the argument is FALSE then the display of the elements will not occur. The array elements MCS[i,j,k] are defined in such a manner that the final index is contravariant.

DIAGMETRIC [*Variable*]

If DIAGMETRIC is TRUE special routines compute all geometrical objects (which contain the metric tensor explicitly) by taking into consideration the diagonality of the metric. Reduced run times will, of course, result. Note: this option is set automatically by TSETUP if a diagonal metric is specified.

DIM [*variable, default: 4*]

DIM is the dimension of the manifold with the default 4. The command DIM:N; will reset the dimension to any other integral value.

EINSTEIN (*dis*) [*Function*]

EINSTEIN computes the mixed Einstein tensor after the Christoffel symbols and Ricci tensor have been obtained (with the functions CHRISTOF and RICCICOM). If the argument dis is TRUE, then the non-zero values of the mixed Einstein tensor G[i,j] will be displayed where j is the contravariant index. RATEINSTEIN default: [TRUE] if TRUE will cause the rational simplification on these components. If RATFAC[FALSE] is TRUE then the components will also be factored.

RICCICOM (*dis*) [*Function*]

This function first computes the covariant components LR[i,j] of the Ricci tensor (LR is a mnemonic for lower Ricci). Then the mixed Ricci tensor is computed using the contravariant metric tensor. If the value of the argument to RICCICOM is TRUE, then these mixed components, RICCI[i,j] (the index i is covariant and the index j is contravariant), will be displayed directly. Otherwise, RICCICOM(FALSE) will simply compute the entries of the array RICCI[i,j] without displaying the results.

DSCALAR (*function*) [*Function*]

applies the scalar d'Alembertian to the scalar function.

```
(C41) DEPENDENCIES(FIELD(R));
(D41)                          [FIELD(R)]
(C42) DSCALAR(FIELD);
(D43)
    -M
  \%E  ((FIELD  N - FIELD  M + 2 FIELD   ) R + 4 FIELD )
             R  R       R  R         R R             R
- ------------------------------------------------------
                            2 R
```

LRICCICOM (*dis*) [*Function*]

LRICCICOM computes the covariant (symmetric) components LR[i,j] of the Ricci tensor. If the argument *dis* is TRUE, then the non-zero components are displayed.

MOTION (*dis*) [*Function*]

MOTION computes the geodesic equations of motion for a given metric. They are stored in the array EM[i]. If the argument *dis* is TRUE then these equations are displayed.

OMEGA [*Variable*]

OMEGA assigns a list of coordinates to the variable. While normally defined when the function TSETUP is called, one may redefine the coordinates with the assignment OMEGA:[j1, j2, ...] where the *ji* are the new coordinate names. A call to OMEGA will return the coordinate name list. Also see TSETUP.

RIEMANN (*dis*) [*Function*]

RIEMANN computes the Riemann curvature tensor from the given metric and the corresponding Christoffel symbols. If dis is TRUE, the non-zero components R[i,j,k,l] will be displayed. All the indicated indices are covariant. As with the Einstein tensor, various switches set by the user control the simplification of the components of the Riemann tensor. If RATRIEMAN, default TRUE is TRUE then rational simplification will be done. If RATFAC[FALSE] is TRUE then each of the components will also be factored.

RATRIEMANN [*Variable*]

one of the switches which controls simplification of Riemann tensors; if TRUE, then rational simplification will be done; if FACRAT:TRUE then each of the components will also be factored.

RAISERIEMANN (*dis*) [*Function*]

returns the contravariant components of the Riemann curvature tensor as array elements UR[I,J,K,L]. These are displayed if *dis* is TRUE.

RINVARIANT () [*Function*]

forms the invariant obtained by contracting the tensors R[i,j,k,l]*UR[i,j,k,l]. This object is not automatically simplified since it can be very large.

SCURVATURE () [*Function*]

returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.

TTRANSFORM (*matrix*) [*Function*]

A function in the CTENSR (Component Tensor Manipulation) package which will perform a coordinate transformation upon an arbitrary square symmetric matrix. The user must input the functions which define the transformation. (Formerly called TRANSFORM.)

WEYL (*dis*) [*Function*]

computes the Weyl conformal tensor. If the argument *dis* is TRUE, the non-zero components W[I,J,K,L] will be displayed to the user. Otherwise, these components will simply be computed and stored. If the switch RATWEYL is set to TRUE, then the components will be rationally simplified; if FACRAT is TRUE then the results will be factored as well.

RATWEYL [*Variable*]

one of the switches controlling the simplification of components of the Weyl conformal tensor; if TRUE, then the components will be rationally simplified; if FACRAT is TRUE then the results will be factored as well.

RATEINSTEIN [*Variable*]

if TRUE rational simplification will be performed on the non-zero components of Einstein tensors; if FACRAT:TRUE then the components will also be factored.

There is a demo in the file: '`tensor/ctenso.dm1`'. There is a demo in the file: '`tensor/ctenso.dm2`'. There is a demo in the file: '`tensor/ctenso.dm3`'. There is a demo in the file: '`tensor/ctenso.dm4`'. There are some usage notes in the file: '`tensor/tensor.doc`'.

16.2.2 Indicial Tensor Manipulation

The Indicial Tensor Manipulation package may be loaded by LOAD(ITENSR);, A manual for the Tensor packages is available in '`tensor/tensor.doc`'.

CHR1 (*[i,j,k]*) [*Function*]

yields the Christoffel symbol of the first kind via the definition

```
(g      + g      - g     )/2 .
  ik,j     jk,i     ij,k
```

To evaluate the Christoffel symbols for a particular metric, the variable METRIC must be assigned a name as in the example under CHR2.

CHR2 (*[i,j],[k]*) [*Function*]

yields the Christoffel symbol of the second kind defined by the relation

```
                        ks
CHR2([i,j],[k]) = g    (g      + g      - g     )/2
                          is,j     js,i     ij,s
```

ALLSYM [*variable, default: TRUE*]

if TRUE then all indexed objects are assumed symmetric in all of their covariant and contravariant indices. If FALSE then no symmetries of any kind are assumed in these indices. Derivative indices are always taken to be symmetric.

CANFORM (*exp*) [*Function*]

Simplifies *exp* by renaming dummy indices and reordering all indices as dictated by symmetry conditions imposed on them. If ALLSYM is TRUE then all indices are assumed symmetric, otherwise symmetry information provided by DECSYM declarations will be used. The dummy indices are renamed in the same manner as in the RENAME function. When CANFORM is applied to a large expression the calculation may take a considerable amount of time. This time can be shortened by calling RENAME on the expression first. Note: CANFORM may not be able to reduce an expression completely to its simplest form although it will always return a mathematically correct result.

CANTEN (*exp*) [*Function*]

Simplifies *exp* by renaming (see RENAME) and permuting dummy indices. CANTEN is restricted to sums of tensor products in which no derivatives are present. As such it is limited and should only be used if CANFORM is not capable of carrying out the required simplification.

CURVATURE (*[i,j,k],[h]*) [*Function*]

yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (CHR2). The following notation is used:

```
           h             h            h         \%1         h
CURVATURE     = - CHR2      - CHR2     CHR2    + CHR2
          i j k         i k,j        \%1 j     i k        i j,k
                          h          \%1
                    + CHR2       CHR2
                          \%1 k       i j
```

CONTRACT (*exp*) [*Function*]

carries out all possible contractions in *exp*, which may be any well-formed combination of sums and products. This function uses the information given to the DEFCON function. Since all tensors are considered to be symmetric in all indices, the indices are sorted into alphabetical order. Also all dummy indices are renamed using the symbols !1,!2,... to permit the expression to be simplified as much as possible by reducing equivalent terms to a canonical form. For best results *exp* should be fully expanded. RATEXPAND is the fastest way to expand products and powers of sums if there are no variables in the denominators of the terms. The GCD switch should be FALSE if gcd cancellations are unnecessary.

COVDIFF (*exp, v1, v2, ...*) [*Function*]

yields the covariant derivative of *exp* with respect to the variables vi in terms of the Christoffel symbols of the second kind (CHR2). In order to evaluate these, one should use EV(exp,CHR2).

DEFCON (*tensor1, tensor2, tensor3*) [*Function*]

gives *tensor1* the property that the contraction of a product of *tensor1* and *tensor2* results in *tensor3* with the appropriate indices. If only one argument, *tensor1*, is given, then the contraction of the product of

tensor1 with any indexed object having the appropriate indices (say tensor) will yield an indexed object with that name, i.e.tensor, and with a new set of indices reflecting the contractions performed. For example, if METRIC:G, then DEFCON(G) will implement the raising and lowering of indices through contraction with the metric tensor. More than one DEFCON can be given for the same indexed object; the latest one given which applies in a particular contraction will be used.

`CONTRACTIONS` [*Variable*]

is a list of those indexed objects which have been given contraction properties with DEFCON.

`DISPCON` (*tensor1, tensor2, ...*) [*Special Form*]

displays the contraction properties of the *tensori* as were given to DEFCON. DISPCON(ALL) displays all the contraction properties which were defined.

`FLUSH` (*exp, tensor1, tensor2, ...*) [*Function*]

will set to zero, in *exp*, all occurrences of the *tensori* that have no derivative indices.

`FLUSHD` (*exp, tensor1, tensor2, ...*) [*Function*]

will set to zero, in *exp*, all occurrences of the *tensori* that have derivative indices.

`FLUSHND` (*exp, tensor, n*) [*Function*]

will set to zero, in *exp*, all occurrences of the differentiated object *tensor* that have *n* or more derivative indices as the following example demonstrates.

```
(C1) SHOW(A([I],[J,R],K,R)+A([I],[J,R,S],K,R,S));
                                J R S      J R
(D1)                           A        + A
                                I,K R S    I,K R
(C2) SHOW(FLUSHND(D1,A,3));
                                      J R
(D2)                                 A
                                      I,K R
```

`INDICES` (*exp*) [*Function*]

returns a list of two elements. The first is a list of the free indices in *exp* (those that occur only once); the second is the list of dummy indices in *exp* (those that occur exactly twice).

`DUMMY` (*i1, i2, ...*) [*Function*]

will set each index *i1, i2, ...* to name of the form !n where *n* is a positive integer. This guarantees that dummy indices which are needed in forming expressions will not conflict with indices already in use.

COUNTER default: [1] determines the numerical suffix to be used in generating the next dummy index. The prefix is determined by the option *DUMMYX* default: [!].

DUMMYX [*variable, default: #*]

The prefix which will be used by the DUMMY function in generating dummy indices in the Tensor package.

COUNTER [*variable, default: 1*]

determines the numerical suffix to be used in generating the next dummy index in the tensor package. The prefix is determined by the option DUMMYX.

KDELTA (*L1, L2*) [*Function*]

is the generalized Kronecker delta function defined in the Tensor package with *L1* the list of covariant indices and *L2* the list of contravariant indices. KDELTA([i],[j]) returns the ordinary Kronecker delta. The command EV(EXP,KDELTA) causes the evaluation of an expression containing KDELTA([],[]) to the dimension of the manifold.

LC (*list*) [*Function*]

is the permutation (or Levi-Civita) tensor which yields 1 if the list *list* consists of an even permutation of integers, -1 if it consists of an odd permutation, and 0 if some indices in *list* are repeated.

LORENTZ (*exp*) [*Function*]

imposes the Lorentz condition by substituting 0 for all indexed objects in *exp* that have a derivative index identical to a contravariant index.

METRIC (*G*) [*Function*]

specifies the metric by assigning the variable METRIC:G; in addition, the contraction properties of the metric *G* are set up by executing the commands DEFCON(G), DEFCON(G,G,KDELTA).

METRIC [*Variable*]

is bound to the metric, assigned by the METRIC command.

REMCON (*tensor1, tensor2, ...*) [*Special Form*]

removes all the contraction properties from the *tensori*. REMCON(ALL) removes all contraction properties from all indexed objects.

NTERMSG () [*Function*]

gives the user a quick picture of the size of the Einstein tensor. It returns a list of pairs whose second elements give the number of terms in the components specified by the first elements.

NTERMSRCI () [*Function*]

returns a list of pairs, whose second elements give the number of terms in the RICCI component specified by the first elements. In this way, it is possible to quickly find the non-zero expressions and attempt simplification.

MAKEBOX (*exp*) [*Function*]

will display *exp* in the same manner as SHOW; however, any tensor d'Alembertian occurring in *exp* will be indicated using the symbol `[]`. For example, `[]`P([M],[N]) represents G([],[I,J])*P([M],[N],I,J).

SHOW (*exp*) [*Function*]

will display *exp* with the indexed objects in it shown having covariant indices as subscripts, contravariant indices as superscripts. The derivative indices will be displayed as subscripts, separated from the covariant indices by a comma.

UNDIFF (*exp*) [*Function*]

returns an expression equivalent to *exp* but with all derivatives of indexed objects replaced by the noun form of the DIFF function with arguments which would yield that indexed object if the differentiation were carried out. This is useful when it is desired to replace a differentiated indexed object with some function definition and then carry out the differentiation by saying EV(...,DIFF).

There is a demo in the file: '`tensor/ctenso.dm1`'. There is a demo in the file: '`tensor/itenso.dm2`'. There is a demo in the file: '`tensor/itenso.dm3`'. There is a demo in the file: '`tensor/itenso.dm4`'. There is a demo in the file: '`tensor/itenso.dm5`'. There is a demo in the file: '`tensor/itenso.dm6`'. There is a demo in the file: '`tensor/itenso.dm7`'. There are some usage notes in the file: '`tensor/tensor.doc`'.

16.3 Exterior Calculus

The exterior calculus of differential forms is a basic tool of differential geometry developed by Elie Cartan and has important applications in the theory of partial differential equations. The present implementation is due to F.B. Estabrook and H.D. Wahlquist. The program is self-explanatory and can be accessed by doing LOAD(CARTAN);. There is a demo in the file: '`share2/cartan.dem`'.

16.4 Dirac Gamma Matrices

GAMALG is a Dirac gamma matrix algebra program which takes traces of and does manipulations on gamma matrices in *n* dimensions. To load GAMALG into a **Maxima**, it is *necessary* to do:

```
LOAD(share/gamalg.l);
BATCHLOAD(share/gamalg.aut);
```

in order to properly set up the environment.

Some examples of the use of GAMALG may be executed by doing BATCH(share/gam.dem); in **Maxima**, which also loads up the above GAMALG files.

16.4.1 Capabilities

1. Takes traces of products of Dirac gamma matrices in n dimensions. In 4 dimensions, it also takes traces of products involving gamma[5] (G5). The results may have free indices.
2. Squares sums of amplitudes, involving polarized or unpolarized spinors.
3. Contracts free indices.
4. Simplifies products of gamma matrices in n dimensions.

For all manipulations, GAMALG uses the conventions of Bjorken and Drell [BD64], and takes Tr(1)=4 (generalization of the spinor dimensionality is unnecessary).

Further information, especially on the algorithms used by GAMALG, may be found in [Wol79], and Caltech preprint CALT-68-720 (June 1979). These references give some discussion of other programs available for high energy physics calculations (including Feynman parametrization etc.).

16.4.2 Summary of GAMALG Functions

(Note: in all functions taking a string of arguments (e.g. TR), list brackets ([,]) may be included or omitted as desired.)

BTR(list) takes the trace of the gamma matrices represented by its argument in a way that is more efficient than TR for long traces invloving many sums of momenta [1].

CIND(mu1,...,muk) adds mu1 through muk to the list of contracted indices [1].

CGT(exp) converts G's to TR's and does them [3].

COMPDEF(vec1=list1,vec2=list2,ind1=val1,ind2=val2,vec3=...) defines lists as the components of vectors and values for indices, for use by NONCOV

CON(exp) contracts all free indices in exp (including epsilon symbols) [3].

CONJ(amp) returns the conjugate of the amplitude amp [2].

COTR(exp) reduces (in n=4) products of traces with contracted indices to single traces [3].

CRUNCH(exp) simplifies untraced products of gamma matrices in exp [3].

DFIX(exp) expands all dot products in exp [3].

EPSFIX(exp) expands all epsilon symbols in exp [3].

FLAGS() displays the values of flags and information lists.

GFIX(exp) expands sums of vectors appearing in untraced products of gamma matrices in exp [3].

GLUE3(l1,l2,l3) gives the tensor corresponding to the three-gluon vertex represented by its arguments [3].

KINDEF(dotp1=rep1, dotp2=rep2, . . .) defines kinematics substitutions dotp1=rep1,. . . , [3].

NONCOV(exp) substitutes the non-covariant components specified by COMPDEF for vectors and indices in dot products in exp [3]

NSET(dim) sets the dimensionality of spacetime to dim [1].

SCALS(x1, . . . , xk) adds x1 through xk to the list of scalars [1].

SQ(spn1,amp,spn2) squares the amplitude amp sandwiched between the spinors spn1 and spn2 [2].

SQAM(spn1,amp1,spn2,amp2) sums over spins the amplitude squared amp1*conj(amp2) sandwiched between the spinors spn1 and spn2 [2].

TR(a1,a2,. . .) takes the trace of gamma matrices represented by its argument [1].

UNCIND(mu1,. . . ,muk) removes mu1 through muk from the list of contracted indices [1].

UNCOMPDEF(vec1,ind1,vec2,vec3,. . .) removes the components defined for its arguments [3].

UNKINDEF(dotp1, . . . , dotpk) removes simplifications defined for dot products dotp1 through dotpk [3].

UNSCALS(x1, . . . , xk) removes x1 through xk from the list of scalars [2].

Pseudofunctions

D(p,q) dot product of p and q

D(p,mu) mu component of the vector p

D(mu,nu) mu, nu component of metric tensor

EPS(p,q,r,s) totally antisymmetric product of p,q,r and s

G(a1, . . . , ak) product of gamma matrices represented by a1, . . . , ak (list brackets around sets of arguments optional)

UV(p,m) a fermion spinor with momentum p and mass m

UVS(p,m,s) a polarized fermion spinor with spin s

ZN(p,m) the numerator of a massive fermion propagator (p(slash)+m)

ZD(p,m) or ZA(p,m) the full propagator for a massive fermion ((p(slash)+m)/(d(p,p)-m*m)). [If ZERM:TRUE then m=0 will be assumed, and ZD(p) or ZA(p) may be used.]

ZDEN(p,m) the denominator of a massive propagator (d(p,p)-m**2) suitable for VIPER generated when VIRED:TRUE.

Flags (Default values in brackets)

BORED[FALSE] if TRUE generates interest (TR0 entries and exits).

COF[FALSE] if TRUE alphabetizes CRUNCH outputs by anticommutation.

DEF[TRUE] if FALSE will prevent the expansion of dot products as they are generated

DOF[TRUE] if FALSE will prevent alphabetization of dot products as they are generated

DSIM[TRUE] if FALSE prevents dot product simplifications defined by KINDEF from being applied.

EPSOF[TRUE] if FALSE will prevent alphabetization of epsilon symbols (antisymmetric products) as they are generated.

EPSEF[TRUE] if FALSE will prevent expansion of epsilon symbols as they are generated.

KAHAF[FALSE] if TRUE will cause the Kahane algorithm to be used on traces with many contracted indices in n=4.

MTRICK[TRUE] if TRUE invokes a more efficient algorithm for treating traces with large numbers of massive spinor propagators in 4 dimensions.

NOP[FALSE] if TRUE causes SQ to generate no primed indices (does Feynman gauge polarization sums).

NTR[FALSE] if TRUE causes SQ to generate G's rather than TR's

PLATU[FALSE] if TRUE uses the inefficient methods of templates for large traces.

VIRED[FALSE] if TRUE generates VIPER-compatible output.

ZERM[FALSE] if TRUE assumes all particle masses to be zero.

BORELEN determines the amount of interest generated when BORED:TRUE

METSIG is the signature of the metric used by NONCOV (default [+,-,-,-])

Information lists (initially empty)

COMPS is the list of components defined by COMPDEF

IND is the list of contracted indices (which will be uncontracted if unpaired)

KINS is the list of kinematic substitutions defined by KINDEF

NPIND is the list of indices automatically summed over by SQ (or SQAM)

SCALARS is the list of scalars

16.4.3 Doing Traces

TR (*a1, a2, . . . , ak*) [*Function*]

is the basic trace function in GAMALG. The *ai* can be:

1. Slashed Lorentz vectors. For example, p(slash)=gamma.p is represented by p, and p(slash)+k(slash)/2 by p+k/2. If a combination of vectors, such as x*p+q (where x is a scalar) is required then x must be declared as a scalar using SCALS (see below).
2. Uncontracted gamma matrices, represented by their indices. These are treated just as slashed vectors.
3. Contracted pairs of gamma matrices, also represented by their indices. These indices are specified by doing CIND(mu1,mu2, . . . , .muj); where the mui are the indices to be contracted.
4. Gamma[5], denoted by G5 (only for n=4).

5. LHP or RHP, the left and right-handed projection operators (also only for n=4). (LHP=(1-G5)/2, RHP=(1+G5)/2)

6. The numerator of a massive spinor propagator, denoted by ZN(p,m). This will be expanded as p(slash)+m.

7. The complete fermion propagator, including denominator, denoted by ZD(p,m). This will be expanded as (p(slash)+m)/(d(p,p)-m*m). If all particles required are massless, ZERM may be set to true, and the m in ZN and ZD omitted.

Note that any list brackets ([,]) in the argument list of TR will be ignored. Any lists to be used in the argument of TR may of course be given names and manipulated outside of TR (e.g REVERSEd, APPENDed to, etc.).

The output from TR is in the form of products of D(p,q) (representing the dot product of the vectors p and q), D(p,mu) (representing the mu component of the vector p, where mu is an unpaired index), D(mu,nu) (representing the metric tensor), EPS(p,q,r,s) (representing the totally antisymmetric product of p,q,r and s), and scalars. Note that n is the dimensionality of spacetime. NSET(4); declares that all traces are to be done in 4 dimensions (n=4) (see below). All D(p,q) and EPS(p,q,r,s) are generated with their arguments in alphabetical order, to aid simplification of expressions. (This may be prevented by setting the flags DOF:FALSE and EPSOF:FALSE, respectively.) Also, unless DEF:FALSE and EPSEF:FALSE, all dot products and epsilon symbols of combinations of vectors will be expanded out as they are generated (e.g. D(p+k/2,q) –> D(p,q)+D(k,q)/2).

BTR(a1,a2, . . . , ak) takes the trace of the gamma matrices represented by its argument in a way that is more efficient than TR if many of the ai involve sums of vectors. (Essentially BTR computes the trace without expanding dot products, and only after simplifying the answer does it expand the dot products.)

CIND(mu1,mu2,. . . ,muj) declares the indices mu1 through muj (which may be any atoms) to be contracted indices. They will be contracted by TR, CRUNCH and CON. The list of contracted indices is IND. Even if an index has been declared to be contracted by CIND, it cannot of course be contracted unless it appears twice. If a contracted index appears more than twice, an error will be printed.

UNCIND(mu1,mu2,. . . ,muj) removes the contraction property from the indices mu1 through muj.

NSET(dim) declares that all the operations should be performed in spacetime of dimension dim. (Note that dim must be close to 4 - GAMALG cannot deal with gamma matrices in 3 or 5 dimensions.) If NSET is not performed, the dimensionality will be taken as n. NSET(4) will slightly restructure GAMALG so as to be more efficient at doing traces in 4 dimensions, and to treat gamma[5] correctly. It can only be repealed by reloading GAMALG. If n=4, then the Kahane algorithm may be invoked by setting KAHAF:TRUE. This will only be more efficient than the algorithm usually used by GAMALG for traces with very large numbers of contracted indices.

SCALS(x1, . . . , xk) declares x1 through xk to be scalars (rather than vectors). SCALARS is the list of declared scalars. A variable need only be declared a scalar if it will appear in dot products, where it might be mistaken for a vector.

UNSCALS(*x1*, . . . , *xk*) removes *x1* through *xk* from the list of scalars.

16.4.4 Squaring Amplitudes

SQ(uv(pi,mi),wt1*g(a1,a2,a3, . . . , ak)+wt2*g(b1,b2, . . . , bj)+...,uv(pf,mf)) squares the amplitude

```
  {ubar(pi,mi)*a1*a2*...*ak*u(pf,mf)}*wt1
+ {ubar(pi,mi)*b1*b2*...*bj*u(pf,mf)}*wt2
+ \dots ,
```

where wt1, wt2, ... are the scalar weights of the various pieces of the amplitude (e.g. denominators of propagators etc.). The ai, bi, ... can be slashed vectors, indices, numerators of propagators (ZN(q,m)) as for TR If the m are omitted from UV,then they will be assumed to be zero. Note that setting the flag ZERM:TRUE will take all masses in uv, zn and zd as zero, and improve the efficiency of the program.

Any of the ai, bi, ... which were declared as contracted indices by CIND, but which appear only once (i.e. uncontracted) in the amplitude (e.g. from external photons) will be replaced by the concatenation of their name with 'prime' in the conjugate of the amplitude, unless they appear in the list NPIND, in which case the indices in the conjugate amplitude will be the same as those in the amplitude, and will be contracted with them (for an external photon this corresponds to a Feynman gauge polarization sum). By setting the flag NOP:TRUE no primes will be generated, and all indices will be contracted.

In squaring amplitudes, SQ takes UV(p,m)*conjugate(UV(p,m)) = ZN(p,m) = p(slash)+m. If UV(p,m) represents an antifermion spinor, then the second argument to UV should be minus the mass of the antifermion, instead of simply the mass as for a fermion. With some conventions, extra overall minus signs must be associated with antifermion propagators.

For polarized fermions, the spinors UV(p,m) are replaced by UVS(p,m,s), where s is the spin vector of the fermion. The spin projection operator used is (1+G5*s(slash))/2.

If the flag NTR:TRUE then SQ will generate G's (untraced products of gamma matrices) instead of TR's. The G's may be evaluated as TR's by using CGT, or may be manipulated using CRUNCH or COTR. When indices are to be contracted between different traces (as from a Feynman gauge photon propagator between two fermion lines), it is much more efficient to COTR G's and then use CGT than to perform the traces and then apply CON.

SQAM(uv(pi,mi),wt1*g(a1, . . . , ak),uv(pf,mf),wt2*g(b1,b2, . . . , bj) sums over spins the product of amplitudes:

```
{ubar(pi,mi)*a1*...*ak*u(pf,mf)}*
conjugate{ubar(pi,mi)*b1*...*bj*u(pf,mf)} *wt1*wt2.
```

16.4.5 Contracting Indices

CON(exp) contracts all pairs of indices in exp which have been declared by CIND. All contractions are performed in n dimensions so that D(mu,mu)=n. In GAMALG, the metric tensor with indices mu and nu is represented by D(mu,nu) while the mu component of the four-vector p is represented by D(p,mu) or D(mu,p). The dot product of two vectors p and q is represented by D(p,q). CON also performs contractions involving epsilon symbols (totally antisymmetric products of four vectors in n=4) represented by eps(a1,a2,a3,a4). CON may be used on expressions containing untraced gamma matrix strings G(p1, . . . , pk). If SQRED:TRUE then these will be output from CON in a form suitable for input to SQ or SQAM.

COTR(exp) takes products of G's in exp, assumes them to be traces, and then combines them into a single trace by contracting indices between them, if this is possible. COTR will only work in n=4. It returns GT's rather than G's, representing undone traces. These can be done using CGT.

DFIX(exp) expands out all dot products in exp. For example, it replaces D(p+k/2,q) by D(p,q)+D(k,q)/2, D(mu,p+q) by D(mu,p)+D(mu,q), and D(x*p+q,k) by x*D(k,p)+D(k,q), where x is a scalar declared with SCALS. In all cases GAMALG alphabetizes the arguments to any D(a,b), mimicing the commutativity of the dot product. This alphabetization will be prevented if DOF:FALSE. Dot products will be expanded out automatically (rendering DFIX unnecessary) unless DEF:FALSE. Setting DEF:FALSE may well speed up some calculations, and prevent explosion of the number of terms in intermediate expressions. In evaluating traces, this will be done automatically if BTR, rather than TR, is used. The simplification of dot products is controlled by the flag DSIM[TRUE]. Unless DSIM:FALSE, the substitutions defined by KINDEFs will be applied to all dot products as they are generated.

EPSFIX(exp) expands out all epsilon symbols in exp. Any epsilon symbol with two equal arguments will automatically be set to zero. Unless EPSOF:FALSE, the arguments to all epsilon symbols will be alphabetized (with suitable signs inserted for the signature of the permutation required) as they are generated. When EPSEF:TRUE, epsilon symbols will be expanded as they are generated.

GLUE3([p1,mu1],[p2,mu2],[p3,mu3]) returns a tensor corresponding to the momentum space part of a three-boson vertex in Yang-Mills theories. All momenta are taken to be ingoing. The complete amplitude for the vertex is -g*f[a,b,c] times the tensor returned, where the order of the group indices is the same as that of the mui. The sign convention is such that a fermion-fermion-boson vertex is -i*g*T[a,[i,j]]*gamma[mu].

16.4.6 Simplifying Products of Gamma Matrices

CRUNCH(exp) simplifies all untraced products of slashed vectors and gamma matrices (represented by G's) in exp. It performs any contractions etc. possible in the product, and returns an expression in terms of D(p,q)... and G(a1, ..., aj)... where the latter represents a product of slashed vectors and gamma matrices. If the flag COF is set to TRUE (then CRUNCH anticommutes gamma matrices until all the arguments of the G(a1, ..., aj) that it returns are in alphabetical order. If COF is FALSE, then the arguments of the G(a1, ..., aj) will be in an order that should give the shortest result if no cancellations occurred. In n=4, the Kahane algorithm will be used. List brackets in the arguments of any G's will be ignored.

CGT(exp) will replace any G in exp with TR and evaluate the resulting traces.

GFIX(exp) expands sums of vectors appearing in G's in exp.

16.4.7 Kinematic Substitutions

KINDEF(dotp1=rep1, ..., dotpj=repj) sets up substitutions for the dot products dotp1, ..., dotpj (e.g. D(p,q)=s, d(p1,p1)=m**2, ...). These will be performed when the dot products are generated unless DSIM:FALSE. The list of substitutions is contained in KINS. New substitutions for a particular dot product will overwrite old ones.

UNKINDEF(dotp1, ..., dotpj) removes the substitutions for the dot products dotp1, ..., .dotpj.

COMPDEF, UNCOMPDEF and NONCOV perform non-covariant kinematics: descriptions not yet written.

16.4.8 Technical Information

GAMALG sets the **Maxima** flags DSKGC:TRUE, PARTSWITCH:TRUE, LISTARITH:FALSE. It will not work if the last two flags are altered. The following atoms must not be bound: LHP, RHP, G5. D, G,

TR must not be used at all (if they are, FREEOF will be got wrong.) Several of the functions above use alphabetization. The "alphabetized" order may be changed by giving aliases. The arrays CONTAB and CONTAB4 must not be altered - they contain tables used by TR and CRUNCH. Scalars declared by SCALS have the "constant" property. GAMALG autoloads the disk files [SHARE]GAM5, GAMKAH, GAMMTR, GAMCON and GAMSQ when the GAMALG.AUT file is loaded in.

GAMALG was originally written very quickly (in a few days) for a specific calculation. Since then it has been revised considerably. While we have tried to test it completely, it may still have bugs.

Addendum by Leo Harten

Owing to the lack of source code documentation, it is difficult if not impossible to make significant fixes/alterations in the codes. I have endeavored to make the codes work well enough to run the '`gam.dem`' file, and have been successful, but this does not mean that the results are correct or reliable; please use extreme caution with this program. Since the source code came in the form of Lisp-save code, it was necessary to generate the **Maxima** code definitions, and then start patching the code. It now works

There is a demo in the file: '`share/gam.dem`'. There are some usage notes in the file: '`share/gam.usg`'.

16.5 Linear Programming

`HACH` (*a, b, m, n, l*) [*Function*]

An implementation of Hacijan's linear programming is available by doing LOAD(KACH);. There is a demo in the file: '`share1/kach.dem`'.

16.6 Dimensional Analysis

`NONDIMENSIONALIZE` (*list of physical quantities*) [*Function*]

The file '`share1/dimen.mc`' contains functions for automatic dimensional analysis. LOAD(DIMEN); will load it up for you. Usage is of the form **NONDIMENSIONALIZE**(*list of physical quantities*).

The returned value is a sufficient list of nondimensional products of powers of the physical quantities. A physical relation between only the given physical quantities must be expressible as a relation between the nondimensional quantities. There are usually fewer nondimensional than physical quantities, which reduces the number of experiments or numerical computations necessary to establish the physical relation to a specified resolution, in comparison with the number if all but one dependent physical variable were independently varied. Also, the absence of any given physical quantity in the output reveals that either the quantity is irrelevant or others are necessary to describe the relation.

The program already knows an extensive number of relations between physical quantities, such as VELOCITY=LENGTH/TIME. (CPUTIME plays the role of the customary **Maxima** global variable TIME.) The user may over-ride or supplement the prespecified relations by typing

```
DIMENSION(equation or list of equations);
```

where each equation is of the form indeterminate=expression, where expression is a product or quotient of powers of none or more of the indeterminates CHARGE, TEMPERATURE, LENGTH, TIME, or MASS. To see if a relation is already established type

```
GET(indeterminate, 'DIMENSION);
```

The result of NONDIMENSIONALIZE usually depends upon the value of the global variable %PURE, which is set to a list of none or more of the indeterminates ELECTRICPERMITTIVITYOFAVACUUM, BOLTZMANNSCONSTANT, SPEEDOFLIGHT, PLANCKSCONSTANT, GRAVITYCONSTANT, corresponding to the relation between charge and force, temperature and energy, length and time, length and momentum, and the inverse-square law of gravitation respectively. Each included relation is used to eliminate one of CHARGE, TEMPERATURE, LENGTH, TIME, or MASS from the dimensional basis. To avoid omission of a possibly relevant nondimensional grouping, either include the relevant constant in %PURE or in the argument of NONDIMENSIONALIZE if the corresponding physical effect is thought to be relevant to the problem. However, the inclusion of unnecessary constants, especially the latter three, tends to produce irrelevant or misleading dimensionless groupings, defeating the purpose of dimensional analysis. As an extreme example, if all five constants are included in %PURE, all physical quantities are already dimensionless. %PURE is initially set to '[ELECTRICPERMITTIVITYOFVACUUM, BOLTZMANNSCONSTANT], which is best for most engineering work. %PURE must not include any of the other 3 constants without also including these 2. There is a demo in the file: '`share1/dimen.dem`'. There are some usage notes in the file: '`share1/dimen.usg`'.

16.7 Asymptotic Analysis

A preliminary version of a program to find the asymptotic behavior of Feynman diagrams has been installed on the SHARE1 directory. For Asymptotic Analysis functions, see ASYMPA. The source code for this function is not yet available in **Maxima**.

ASYMP is a package for determining the asymptotic behavior of Feynman integrals. Given a topological description of a Feynman diagram as a set of lines and vertices, together with information about the mass of the virtual particle corresponding to each line and the momentum entering at each external leg, it will tell one the leading asymptotic behavior of that graph as some sets of masses get much larger than others.

As this package is very unlikely to be of use to people who are not familiar with Feynman diagrams and other basic aspects of perturbative quantum field theory, we will refrain from describing the basics here and refer the interested reader to any of the standard textbooks on the subject instead.

Perhaps this is also the appropriate place to mention the limitations of the package. These are of two kinds, those which are fundamental limitations of the formalism and methods used in the package itself, and those which are just features which could be added easily if they are ever needed. In the first category we stress that the bounds are obtained for individual Feynman graphs, and not for sums of them; in other words the asymptotic behavior of a green's function might be quite different from that of the graphs which contribute to it, because there may be "miraculous" cancellations. Such cancellations occur in many interesting theories, in particular gauge theories, but they are best dealt with by means of Ward identities rather than explicit calculation. Another mathematical limitation is that the actual behavior of a graph is only bounded by the result given – in reality the graph might have a smaller asymptotic growth: the bounds obtained are usually fairly good, however. In the second class of limitations we should mention that (1) the package currently deals only with boson fields, (2) allows only a trivial dependence of the vertices upon momenta and masses,

(3) tries to compute 1/0 for IR divergent graphs [which is honest, in a way], and (4) returns INF for a UV divergent graph [which is correct]. All of these are simple to generalize in the program, and if one need to get around these limitations, please contact the authors. A slightly harder problem to circumvent is related to point (4) above, namely (5) one cannot currently specify that a UV divergent graph is to be subtracted in a certain way: part of the problem is that there are many different subtraction schemes (minimal, zero momentum Taylor series, etc.) and how to specify which method one wants is not clear, but it would also require a fair amount of thought to make the program renormalize automatically.

16.7.1 Simple Example

The easiest way to see how ASYMP works is to look at the simplest example, the one-loop three-point function in (phi)**3 theory. First of all we must load the ASYMP package into a **Maxima**:

```
(C1) load("asymp")$
ASYMP: version of 11:54pm  Saturday, 4 July 1981

(C2) graph1:diagram(line(a,b,1,m),line(b,c,2,m),line(c,a,3,mm),
        extline(a,4,p),extline(b,5,q),extline(c,6,-p-q))$
1 Loop Diagram


(C3) bound(graph1,[[m,p,q],mm,inf]);

                                        MM
                                    LOG(--)
                                        M
(D3)                                -------
                                        2
                                      MM
```

First of all, in line (C1) we have loaded up the FASL (compiled) version of the ASYMP package. It identifies itself by telling us the date on which it was born. We then define the desired Feynman diagram as GRAPH1 using the DIAGRAM function. DIAGRAM takes an arbitrary number of arguments, each of which is a pseudo-function describing a part of the graph. Currently, there are two such pseudo-functions, LINE and EXTLINE. Logically enough LINE describes an internal line; if we type LINE(LONDON, PARIS, RHUBARB, 5*M[PLANCK]) we are defining a line from a vertex called LONDON to a vertex called PARIS corresponding to a particle of mass 5*M[PLANCK]. A couple of points are to be noted, (1) the vertices can be names, numbers, or anything one want as long as it is a valid argument to a hashed array, (2) the factor of 5 in the mass is pointless, as numerical factors are ignored in asymptotic bounds. The third argument, RHUBARB, is a name for the line, which is solely there for debugging purposes: internally ASYMP will invent its own name for the line. This argument is, like rhubarb, best left by the side of the plate and ignored. EXTLINE describes an external leg to our Feynman diagram. EXTLINE(ROME, CELERY,-P+2*Q) says that there is an external leg attached to our graph at vertex ROME carrying momentum 2*Q-P into the graph. It is one's own responsibility to ensure over all momentum conservation. The second argument, CELERY, has great similarities to RHUBARB and is also best forgotten (well, it has to be there, but it seems to serve no other useful role in life).

OK, so we have now defined our graph. DIAGRAM sets up tables of lines containing their masses etc., assigns internal loop-momenta, routes all momenta through the graph, and tells one the number of loops in

the diagram. If we had been nosey and typed a ; rather than a $ at DIAGRAM, it would have returned a list of the form [G000002653,G000005532,G000007771]. The Goo's are internal line-names of no interest to one, other than that they are used by later programs to index the tables set up by DIAGRAM and its cohorts. The only point of interest is that GRAPH1 is now a list of variable names, in other words it behaves just like any other **Maxima** variable, which is not too surprising because it IS just like any other **Maxima** variable.

In line (C2) we get down to the real business of the day. We use the function BOUND to find the asymptotic behavior of GRAPH1 when the Euclidean momenta p and q and the mass m are much smaller than the mass mm, and both are much smaller than INF (of course: the need to put in INF by hand is just a foible of the program, so don't forget it!). To put it another way, we set up three mass scales, which we shall call m, mm, and INF, such that any mass of order m is asymptotically bounded by (or, in everyday terms, much less than) any mass of order mm, and in turn mm « INF. The second argument to BOUND, therefore, is a list of mass-scales, each of which is either a mass/momentum or a list of masses and/or momenta of the same scale. The result of BOUND is that GRAPH1 is bounded by (an implicit constant) times log(mm/m)/mm**2, at least for mm/m large enough. There is a demo in the file: '`share1/asymp.dm1`'. There is a demo in the file: '`share1/asymp.dm2`'. There is a demo in the file: '`share1/asymp.dm3`'. There is a demo in the file: '`share1/asymp.dmo`'. There are some usage notes in the file: '`share1/asymp.usg`'.

ASYMPA - Asymptotic Analysis. The file '`share1/asympa.mc`' contains simplification functions for asymptotic analysis, including the big-O and little-o functions that are widely used in complexity analysis and numerical analysis. Do LOAD(ASYMPA);. For asymptotic behavior of Feynman diagrams, see ASYMP.

There are some usage notes in the file: '`share1/asympa.usg`'.

16.8 Set Packages

Maxima has two packages for Set Theory. One uses **Maxima**'s list representation, meaning you can convert easily between lists and sets. It contains commands such as UNION, INTERSECTION, COMPLEMENT, etc. It may be accessed by LOAD(SET);. The other is the SETS package, which also contains commands for the usual sorts of operations performed on sets, uses a different representation and is computationally faster. It is available by doing LOAD(SETS);.

16.8.1 Set

The file '`share2/set.lsp`' provides **Maxima** with the basic primitives of set theory. For our purposes, we define a set to be a sorted irredundant list. Thus [2,3,x,1] is not a set since it isn't sorted and [2,3,x,x,y] is not a set since it has a redundancy; however [1,2,3,x] and [2,3,x,y] are sets.

Here is a list of the functions provided: INTERSECT, UNION, COMPLEMENT, SETDIFFERENCE, SYMMDIFFERENCE, POWERSET, SETIFY, SUBSET, SETP, SUBSETP, DISJOINTP. The functions above are mostly self-explanatory, so the following brief descriptions should be sufficient:

INTERSECT(*A*, *B*) returns the intersection of *A* and *B*.

UNION(*A*,*B*) returns the union of *A* and *B*.

COMPLEMENT(*A*,*B*) returns the relative complement of *A* in B, That is, the set of elements of B which are not in *A*.

SETDIFFERENCE(*A*,*B*) returns the set of elements of *A* which are not in *B*. Note that × SETDIFFERENCE(A,B) is just COMPLEMENT(B,A).

SYMMDIFFERENCE(*A*, *B*) returns the symmetric difference of *A* and *B*, which is equal to UNION(SETDIFFERENCE(A, B),SETDIFFERENCE(B, A)).

POWERSET(*S*) returns the set of all subsets of *S*.

SETIFY(*L*) converts the list *L* to a set. For example SETIFY([2,x,2,3,8,x]) yields [2,3,8,x].

SUBSET(*A*,*F*) returns the set of elements of *A* which satisfy the condition F. For example × SUBSET([1,2,x,x+y,z,x+y+z],atom) yields [1,2,z]. The argument F should be a function of one argument which returns TRUE or FALSE. Another example: SUBSET([1,2,7,8,9,14],evenp) yields [2,8,14].

SETP(*L*) returns TRUE if L is a set, FALSE otherwise.

SUBSETP(*A*,*B*) decides whether or not *A* is a subset of *B*, and returns TRUE or FALSE accordingly.

DISJOINTP(*A*,*B*) decides whether *A* and *B* are disjoint (have no common elements) and returns TRUE or FALSE accordingly.

Remarks:

1. We reemphasize that a set is a list. Thus the notation for both input and output is [...] rather than { ... }. We feel that this ambiguity is desirable. It allows list processing on sets and set processing on lists without conversion.

2. However, not every list is a set. A set must satisfy two additional conditions. It must be (1) sorted and (2) irredundant. Implied is a notion of order and a notion of equivalence. We choose order as the fundamental notion.

CANONLT [*variable, default: ORDERLESSP*]

The user can specify an order by resetting the value of CANONLT. Initially, CANONLT has the value ORDERLESSP which is **Maxima**'s canonical order. The value of CANONLT must be a strict total preorder (a function of 2 arguments which returns true or false and such that the associated relation is transitive and irreflexive). It need not be universally well-defined so long as it is well-defined for the arguments it receives. The notion of equivalence is derived from the order notion by the familiar law of trichotomy: given objects a,b we require that exactly one of the following holds:

1. a is less than b

2. b is less than a

3. a and b are equivalent

Note that equivalence is, in general, weaker than equality.

Consider the following example: Suppose we define a function H by: H(N):=FOR I DO IF NOT INTEGERP(N/(2**I)) THEN RETURN(I-1); Thus H returns the largest positive integer *E* such that *n* is divisible by 2**e (*n* is assumed to be a positive integer).

Next define an order function F by: F(A,B):=IS(H(A) < H(B)); For example, with this order, 9 is less than 2 and 6 is equivalent to 10.

Next we set CANONLT by: CANONLT:f;. Then all functions in the set package will return sort and eliminate redundancies (equivalences) using the function f. For example, SETIFY([6, 20, 2, 9, 8); returns [9, 2, 8].

As mentioned above the default value of CANONLT is ORDERLESSP. It follows that the default notion of equivalence is equality.

The functions UNION and INTERSECT can take any number of arguments.

The arguments to functions in the SET package need not be sets - arbitrary lists are OK (i.e. possibly unsorted and/or redundant). Thus, functions such as INTERSECT accept arbitrary lists but always return a set. For example, INTERSECT([2,3,2,y,x],[2,z,y,y]) yields [2,y].

There are some usage notes in the file: 'share2/set.usg'.

16.8.2 Sets

There is a fast sets package available by doing LOAD(SETS);. The set constructor is the {" and the }. So X:{A,B,C,D,E}; creates a set. The usual primitives UNION, **INTERSECTION**, **SETDIFF**, **SYMDIFF** are defined.

```
SYMDIFF(A,B) = UNION(SETDIFF(A,B),
SETDIFF(B,A)) = SETDIFF(UNION(A,B),INTERSECTION(A,B))
```

Predicates are: **ELEMENTP**(*x*, *set*), **EMPTYP**(*set*), and **SUBSETP**(*set1*, *set2*). **CARDINAL**(*SET*) returns the cardinality.

There are two mapping-like functions which are provided for sets:

PREDSET(*predicate*, *set*) returns the set of all elements of *set* such that the *predicate* returns TRUE:

```
X:{1,2,3,4,5,6,7,8,9,10,11}
PREDSET(LAMBDA([U],IS(ABS(U-6)<3)),X); -> {5,6,7}
```

MAPSET(*function*, *set*) creates a set from the results of applying the *function* to the elements of the *set*.

ELEMENTS(*SET*) returns a list of the elements.

The sets are not represented as lists. The set-algebraic functions (UNION, INTERSECTION, SETDIFF, SYMDIFF, PREDSET, CARDINAL), all operate on the internal representation of sets and as such are fast. Things which have to be converted from the set representation to non-set are a bit slower, the things which make sets from raw elements are slower still, however, they are somewhat faster than CONS on the average.

The SETS package is not available in **Maxima**. There is a demo in the file: 'share1/sets.dmo'. There are some usage notes in the file: 'share1/sets.usg'.

16.9 Vectors

The file 'share/vect.mc' contain a vector analysis package. 'share/vect.dem' contains a corresponding demonstration, and 'share/vect.orth' contains definitions of various orthogonal curvilinear coordinate systems. LOAD(VECT); will load this package for you. The vector analysis package can

combine and simplify symbolic expressions including dot products and cross products, together with the gradient, divergence, curl, and Laplacian operators. The distribution of these operators over sums or products is under user control, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There is also a capability for deriving the scalar or vector potential of a field. The package contains the following commands: VECTORSIMP, SCALEFACTORS, EXPRESS, POTENTIAL, and VECTORPOTENTIAL.

VECTORSIMP (*vectorexpression*) [*Function*]

This function employs additional non-controversial simplifications, together with various optional expansions according to the settings of the following global flags: EXPANDALL, EXPANDDOT, EXPANDDOTPLUS, EXPANDCROSS, EXPANDCROSSPLUS, EXPANDCROSSCROSS, EXPANDGRAD, EXPANDGRADPLUS, EXPANDGRADPROD, EXPANDDIV, EXPANDDIVPLUS, EXPANDDIVPROD, EXPANDCURL, EXPANDCURLPLUS, EXPANDCURLCURL, EXPANDLAPLACIAN, EXPANDLAPLACIANPLUS, EXPANDLAPLACIANPROD. All these flags have default value FALSE. The PLUS suffix refers to employing additivity or distributivity. The PROD suffix refers to the expansion for an operand that is any kind of product. EXPANDCROSSCROSS refers to replacing p (q r) with (p.r)*q-(p.q)*r, and EXPANDCURLCURL refers to replacing CURL CURL p with GRAD DIV p + DIV GRAD p. EXPANDCROSS:TRUE has the same effect as EXPANDCROSSPLUS:EXPANDCROSSCROSS:TRUE, etc. Two other flags, EXPANDPLUS and EXPANDPROD, have the same effect as setting all similarly suffixed flags true. When TRUE, another flag named EXPANDLAPLACIANTODIVGRAD, replaces the LAPLACIAN operator with the composition DIV GRAD. All of these flags are initially FALSE. For convenience, all of these flags have been declared EVFLAG.There is a demo in the file: '`share/vect.dem`'. There are some usage notes in the file: '`share/vect.usg`'.

VECTORPOTENTIAL (*givencurl*) [*Function*]

Returns the vector potential of a given curl vector, in the current coordinate system. POTENTIALZEROLOC has a similar role as for POTENTIAL, but the order of the left-hand sides of the equations must be a cyclic permutation of the coordinate variables.

VECT_CROSS [*variable, default: FALSE*]

if TRUE allows DIFF(X Y,T) to work where is defined in '`share/vect`' (where VECT_CROSS is set to TRUE, anyway).

SCALEFACTORS (*coordinatetransform*) [*Function*]

For orthogonal curvilinear coordinates, the global variables COORDINATES[[X,Y,Z]], DIMENSION[3], SF[[1,1,1]], and SFPROD[1] are set by this function invocation. Here *coordinatetransform* evaluates to the form *[[expression1, expression2, ...], indeterminate1, indeterminat2, ...]*, where *indeterminate1*, *indeterminate2*, etc. are the curvilinear coordinate variables and where a set of rectangular Cartesian components is given in terms of the curvilinear coordinates by *expression1, expression2, ...]*. COORDINATES is set to the vector *[indeterminate1, indeterminate2,...]*, and DIMENSION is set to the length of this vector. SF[1], SF[2], ..., SF[DIMENSION] are set to the coordinate scale factors, and SFPROD is set to the product of these scale factors. Initially, COORDINATES is [X, Y, Z], DIMENSION is 3, and SF[1] = SF[2]= SF[3]=

SFPROD= 1, corresponding to 3-dimensional rectangular Cartesian coordinates. There are some usage notes in the file: ‘`share/vect.usg`’.

`POTENTIAL` (*givengradient*) [*Function*]

Returns the scalar potential of a given gradient vector, in the current coordinate system. The calculation makes use of the global variable POTENTIALZEROLOC default: [0], which must be NONLIST, or of the form [indeterminatej = expressionj, indeterminatek = expressionk, ...], the former being equivalent to the nonlist expression for all right-hand sides in the latter. The indicated right-hand sides are used as the lower limit of integration. The success of the integrations may depend upon their values and order. POTENTIALZEROLOC is initially set to 0.

`EXPRESS` (*expression*) [*Function*]

Used to expand an expression into physical components in the current coordinate system. The result uses the noun form of any derivatives arising from expansion of the vector differential operators. To force evaluation of these derivatives, the built-in EV function can be used together with the DIFF evflag, after using the built-in DEPENDS function to establish any new implicit dependencies.

BIBLIOGRAPHY

[AS64] Milton Abramowitz and Irene Stegun, editors. *Handbook of Mathematical Functions*. National Bureau of Standards, Wahington, D.C., 1964.

[BD64] J. D. Bjorken and S. D. Drell. *Relativistic Quantum Mechanics*. McGraw-Hill, New York, 1964.

[Lew79] V. Lewis, editor. *Proceedings of the 1979 MACSYMA User's Conference*, Washington, 1979.

[Wan71] P. S. Wang. Evaluation of definite integrals by symbolic manipulation. Technical report, MIT, 1971. MAC TR-92.

[Wol79] Steve Wolfram. Macsyma tools for feynman diagram calculations. In V. Lewis, editor, *MACSYMA User's Conference*, Washington, 1979.

INDEX