The Maxima Workbook

Roland Salz

Version 0.2 October 13, 2017

Contents

Preface

Maxima was developed from 1968-1982 at MIT (Massachusetts Institute of Technology) as the first comprehensive Computer Algebra System. It was improved ever since. Today it is free software, maintained by an energetic group of volunteers called the “Maxima team”. The author wishes to thank its friendly and helpful members!

The intention of the Maxima Workbook is to provide a new documentation of the computer algebra system Maxima. It is aimed at both users and developers. As a users’ manual it contains a description of the Maxima language, here abbreviated MaximaL. User functions written by the author are added wherever he felt that Maxima’s standard functionality is lacking them. As a developers’ manual it describes a possible Lisp development environment. Maxima is written in Common Lisp. So the interrelation between MaximaL and Lisp is highlighted. We are convinced that there is no clear distinction between a Maxima user and a developer. Any sophisticated user tends to become a developer, too, and he can do so either on his own or by joining the Maxima team.

Comments and suggestions for improvement are welcome under mail@roland-salz.de.

This project is work in progress! It is in the beginning phase.

Part I

Historical evolution, comparison, documentation

Chapter 1

Historical evolution

1.1 Overview

Maxima was developed, originally under the name Macsyma, from 1968 until 1982 at the Massachusetts Institute of Technology (MIT) as part of project MAC. Development took place in parallel to the development of MACLisp, the language in which Macsyma was originally written.

In 1982 the project was split. A commercial license was given to a company named Symbolics, Inc., created by Macsyma users and former MIT developers, while at the same time the United States Department of Energy (DOE) obtained a license for the source code of Macsyma to be made available in a semi-public data base. This version is known as DOE Macsyma. When Symbolics got into financial problems the license was forwarded to a company called Macsyma, Inc., which continued development until 1999. This version of Macsyma is still sold today by Symbolics, but the latest Windows version it supports is XP.

From 1982 until his death in 2001, William Shelter from the University of Texas maintained a copy of DOE Macsyma. In 1999 he received permission from the Department of Energy to publish the source code on the net under a GNU public license. In 2000 he initiated the open source software project at Sourceforge, where it has remained until today. In order to avoid legal conflicts with the commercial license, the open source project was renamed into Maxima.

1.2 MAC, MACLisp and MACSyMa: The project at MIT

1.2.1 Initialization and basic design concepts

While William A. Martin (1938-1981) had studied at MIT since 1960 and worked on his doctoral thesis under the computer science pioneer Marvin Minsky (1927–2016) since 1962, Joel Moses (born 1941) entered MIT in 1963 and also took up a doctorate under Marvin Minsky. After both having pursued various other projects in the area of artificial intelligence and symbolic computation, and after having completed their respective theses in 1967 (Joel Moses's thesis was entitled *Symbolic integration*), while staying at MIT they joined their efforts and initialized, together with Carl Engelman, the development of a computer algebra system called *Macsyma*, project MAC's SYmbolic MAnipulator. It was meant to be a combination of all their previous projects, an interactive system for solving symbolic mathematical problems designed for engineers, scientists and mathematicians, with the capabil- [MosesMPH12]

ity of two-dimensional display of formulas on the screen, an interpreter for step-by-step processing, and using the latest and most sophisticated algorithms in symbolic computation available at the time.

Since both liked Lisp for its short and elegant notation and the universal and flexible list structure, and since they had used it in most of their previous projects, Lisp was going to be the language in which Macsyma was to be written.

Another conceptual decision based on previous experiences was to use multiple internal representations for mathematical expressions. Apart from the general representation there would be a rational function representation for manipulating ratios of polynomials in multiple variables, and another representation for power and Taylor series. These different representations can still be found in today's Maxima.

Bill Martin led the project. But Carl Engelman and his staff already left in 1969.

In 1971 the project was presented at a Symposium on Symbolic and Algebraic Manipulation by William Martin and Richard Fateman (born 1946), who had joined the project right from the beginning. Officially he did his doctorate at Harvard, but de facto he worked in the Macsyma project at MIT. His thesis on *Algebraic Simplification* describes various components of Macsyma which he had implemented. The project now comprised a considerable number of doctoral students and post-doc staff members. But soon after this presentation William Martin left the project, too, which was then led by Joel Moses. [MartFate71] [FatemThe72]

1.2.2 Major contributors

Major contributors to the Macsyma software were:

William A. Martin (front end, expression display, polynomial arithmetic) and Joel Moses (simplifier, indefinite integration: heuristic/Risch). Some code came from earlier work, notably Knut Korsvold's simplifier. Later major contributors to the core mathematics engine were:[citation needed] Yannis Avgoustis (special functions), David Barton (solving algebraic systems of equations), Richard Bogen (special functions), Bill Dubuque (indefinite integration, limits, power series, number theory, special functions, functional equations, pattern matching, sign queries, Gröbner, TriangSys), Richard Fateman (rational functions, pattern matching, arbitrary precision floating-point), Michael Genesereth (comparison, knowledge database), Jeff Golden (simplifier, language, system), R. W. Gosper (definite summation, special functions, simplification, number theory), Carl Hoffman (general simplifier, macros, non-commutative simplifier, ports to Multics and LispM, system, visual equation editor), Charles Karney (plotting), John Kulp, Ed Lafferty (ODE solution, special functions), Stavros Macrakis (real/imaginary parts, compiler, system), Richard Pavelle (indicial tensor calculus, general relativity package, ordinary and partial differential equations), David A. Spear (Gröbner), Barry Trager (algebraic integration, factoring, Gröbner), Paul Wang (polynomial factorization and GCD, complex numbers, limits, definite integration, Fortran and LaTeX code generation), David Y. Y. Yun (polynomial GCDs), Gail Zacharias (Gröbner) and Rich Zippel (power series, polynomial factorization, number theory, combinatorics). [wikMacsy17]

1.2.3 The users community

A nationwide Macsyma users community, to which belonged, among others, DOE, NASA and the US Navy, had evolved in parallel to the ongoing development of the system at MIT, and they jointly used computers and system resources provided by ARPA and ARPANET. Significant funds for the project came from this user group, too. The Macsyma software had grown so large that always the newest version of a PDP-10 computer from DEC was needed to host it. Eventually, when DEC took a decision to change to the VAX architecture, the whole Macsyma project had to be turned over to follow it.

1.3 Users conferences and first competition

In 1977 Richard Fateman, who, as a professor for computer science, had gone to the University of California in Berkeley, organized the first one of what would become altogether four Macsyma Users Conferences.

1.3.1 The beginning of Mathematica

Stephen Wolfram, a physicist and former Macsyma user from Cal Tech, designed and presented his own commercial computer algebra system, called SMP, in 1981. This eventually led to the development of Mathematica. [ColeSMP81]

1.3.2 Announcement of Maple

At the Macsyma Users Conference of 1984 another new and commercial CAS project, called Maple, was presented. Although strongly influence by Macsyma, it aimed at increasing the speed of computation and at the same time at reducing system memory size, so that it could operate on smaller and cheaper hardware and eventually on personal computers. [CharMap84]

1.4 Commercial licensing of Macsyma

1.4.1 End of the development at MIT

In 1981 the idea came up among Macsyma developers at MIT to form a company which should take over development of Macsyma and market the product commercially. The intention was to run the CAS on VAX-like machines and possibly smaller computers. Joel Moses, who had led the project since 1971, became increasingly engaged in an administrative career at MIT (he was provost from 1995-1998), leaving him little time to continue heading the Macsyma project. In 1982 the development of Macsyma at MIT had come to an end.

1.4.2 Symbolics, Inc. and Macsyma, Inc.

Symbolics, Inc., a company that had been founded by former MIT developers to produce LISP-based hardware, the so-called lisp machines, received a license for the Macsyma software in 1982. While the product started well on VAX-machines, the development of Macsyma for use on personal computers fell way behind the competitive commercial systems Maple and Mathematica.

Lisp-machines did not become the commercial success that had been expected, so Symbolics did not have the financial capabilities to continue the development of Macsyma. In 1992 Symbolics sold the license to a company called Macsyma, Inc. which continued to develop Macsyma until 1999. This version of Macsyma is still sold (as of 2017) by Symbolics for Microsoft's Windows XP operating system. Later versions of Windows, however, are not supported.

1.5 Academic and US government licensing

1.5.1 DOE Macsyma

Already in 1981, when the discussion about a commercial licensing of Macsyma began at MIT, a number of Macsyma users and former MIT developers asked DOE, one of the main sponsors of the project, to allow the software to become available for free to everyone. In 1982, at the same time when the commercial license was sold to Symbolics, the United States Department of Energy (DOE) obtained a copy of the source code from MIT to be kept in a semi-public data base. It was not made available to the public for free, its use remained restricted to academics and US government institutions. This version is known as *DOE Macsyma*.

1.5.2 William Shelter from the University of Texas

From 1982 until his death in 2001, William Shelter from the University of Texas in Austin maintained DOE Macsyma.

1.6 GNU public licensing

In 1999, when the development of commercial Macsyma de facto terminated, William Shelter received permission from the Department of Energy to publish the source code of DOE Macsyma under a GNU public license. In 2000 he initiated the open source software project at Sourceforge, where it has remained until today. In order to avoid legal conflicts with the still existing commercial license of Macsyma, the open source project was renamed *Maxima*.

Since 1982, the source code of DOE Macsyma had remained completely separated from the commercial version of Macsyma. So the code of Maxima does not include any of the enhancements, revisions or bug fixings made by Symbolics between 1982 and 1999.

1.6.1 Maxima, the open source project since 2001

Maxima today is free software. Judging from the number of downloads, it has about 150.000 users. New releases come about twice a year. Installers are provided for Linux and Windows (32 and 64 bit versions), but Maxima can also be built by anyone directly from the source code.

A worldwide, energetic group of volunteers, called the "Maxima team" and led by Robert Dodier from Portland, Oregon, today maintains Maxima. Among the Lisp developers are Raymond Toy, Barton Willis (University of Nebraska, Kearney), Kris Katterjohn, and David Billinghurst. Gunter Königsmann (Erlangen, Germany) maintains the most popular user interface, wxMaxima, developed by Andrej Vodopivec

(Slovenia). Wolfgang Dautermann (Graz, Austria) created a cross compiling mechanism for the Windows installers. Yasuaki Honda (Japan) developed the iMaxima interface running under Emacs. Mario Rodriguez (Spain) integrated and maintains the plotting software, Viktor Toth (Kanada) is in charge of new releases and maintains the tensor packages. Jaime Villate (Portugal) among other things designed the homepage. Many others could be mentioned who contribute to Maxima in one way or the other, for instance by writing and providing external software packages. Edwin (Ted) Woollett (San Luis Obispo, California), has spent years writing a highly sophisticated and free Maxima tutorial for applications in physics, called "Maxima by example". Richard Fateman (Berkeley) and Stavros Macrakis (Boston), who already were major contributors to the Macsyma software at MIT, are both still with the Maxima project today, giving valuable advice to developers and users on Maxima’s principal communication channel, the mailing list at Sourceforge.

Chapter 2

Documentation

2.1 Manuals and internal documentation

The official Maxima manual (in English) can be found in html format on Sourceforge under http://maxima.sourceforge.net/docs/manual/maxima.html. It is revised together with each new Maxima release. It is included in html format, in PDF format and as an online help in eache Maxima installer or tarball. It can also be built when building Maxima from source code. The Maxima Workbook is primarily based on this documentation. [MaximMan17]

A German version is available under http://maxima.sourceforge.net/docs/manual/de/maxima.html. It is also distributed with the Maxima installers and tarballs. Note, however, that it reflects the status of release 5.29. Compared to the English version, it contains numerous additional comments and examples and a much more complete index. It was translated/written by Dieter Kaiser in 2011. Many of his amendments and improvements have been incorporated in the Maxima Workbook. The author wishes to express his thanks to Dieter Kaiser for his pioneer work in improving the Maxima documentation. [MaxiManD11]

2.2 External documentation

2.3 Tutorials

2.3.1 Maxima by example

Part II

Basic Operation

Chapter 3

Basics

Chapter 4

Input and output

Chapter 5

Plotting

Chapter 6

Batch Processing

Part III

Concepts of Symbolical Computation

Chapter 7

Data types and structures

7.1 Numbers

7.1.1 Introduction

7.1.1.1 Types

Maxima distinguishes four generic types of numbers: integer, rational number, floating point number and big floating point number. There is no generic type for complex numbers.

7.1.1.2 Predicate functions

numberp (expr) [predicate function]

If *expr* evaluates to an integer, a rational number, a floating point number or a big floating point number, *true* is returned. In all other cases (including a complex number) *false* is returned.

Note. The argument to this and the following predicate functions described in this section concerning numbers must really evaluate to a number in order for the function to be able to return *true*. A symbol that does not evaluate to a number, even if it is declared to be of a numerical type, will always cause the function to return *false.* The special predicate function *featurep (symbol, feature)* can be used to test for such merely declared properties of a symbol.

```
(%i1)   c;
(%o1)                          c
(%i2)   declare(c, even);
(%o2)                        done
(%i3)   featurep(c, integer);
(%o3)                        true
(%i4)   integerp(c);
(%o4)                        false
(%i5)   numberp(c);
(%o5)                        false
```

7.1.2 Integer and rational numbers

7.1.2.1 Representation

7.1.2.1.1 External

Integers are returned without a decimal point. Rational numbers are returned as a fraction of integers. Arithmetic calculations with interger and rational numbers are exact. In principal, integer and rational numbers can have an unlimited number of digits.

```
(%i1)  a:1;
(%o1)                                  1
(%i2)  b:-2/3;
                                       2
(%o2)                                 - -
                                       3
(%i3)  100!;
(%o3)  933262154439441526816992388562667004907159682643816214685929\
       638952175999932299156089414639761565182862536979208272237582\
       511852109168640000000000000000000000
```

7.1.2.1.2 Internal

```
(%i1)  a:1/2;
                                       1
(%o1)                                  -
                                       2
(%i3)  :lisp $a
       ((RAT SIMP) 1 2)
```

7.1.2.1.2.1 Canonical rational expression (CRE)

7.1.2.2 Predicate functions

```
(%i1)  a:1$  b:2$  c:0$  d:3/4;
(%i5)  integerp(a);
(%o5)                                 true
(%i6)  evenp(c);
(%o6)                                 true
(%i7)  oddp(a-b);
(%o7)                                 true
(%i8)  nonnegintegerp(2*c*a);
(%o8)                                 true
(%i8)  ratnump(a+d);
(%o8)                                 true
```

integerp (expr) [Predicate function]

If *expr* evaluates to an integer, *true* is returned. In all other cases *false* is returned.

evenp (expr) [Predicate function]

If *expr* evaluates to an even integer, *true* is returned. In all other cases *false* is returned.

oddp (expr) [Predicate function]

If *expr* evaluates to an odd integer, *true* is returned. In all other cases *false* is returned.

nonnegintegerp (expr) [Predicate function]

If *expr* evaluates to a non-negative integer, *true* is returned. In all other cases *false* is returned.

ratnump (expr) [Predicate function]

If *expr* evaluates to an integer or a rationl number, *true* is returned. In all other cases *false* is returned.

7.1.2.3 Type conversion

7.1.2.3.1 Automatic

If any element of an expression that does not contain floating point numbers evaluates to a rational number, then all integers in this expression are, when evaluated, converted to rational numbers, too, and the value returned is a rational number.

7.1.2.3.2 Manual

rationalize (expr) [Function]

Converts all floating point numbers and bigfloats in *expr* to rational numbers. Maxima knows a lot of identities but applies them only to exactly equivalent expressions. Floats are considered inexact so the identities aren't applied. *rationalize* replaces floats with exactly equivalent rationals, so the identities can be applied.

It might be surprising that `rationalize` (0.1) does not equal 1/10. This behavior is because the number 1/10 has a repeating, not a terminating binary representation.

```
(%i1)    rationalize(0.1);
```

$$\text{(\%o1)} \qquad \frac{3602879701896397}{36028797018963968}$$

Note. The exact value can be obtained with either function *fullratsimp (expr)* or, if a CRE form is desired, with *rat(expr)*.

```
(%i1)    rat(0.1);
rat: replaced 0.1 by 1/10 = 0.1
```

$$\text{(\%o1)} \qquad /R/ \qquad \frac{1}{10}$$

7.1.3 Floating point numbers

7.1.3.1 Ordinary floating point numbers

Maxima uses floating point numbers (floating points) with double presicion. Internally, all calculations are carried out in floating point.

Floating point numbers are returned with a decimal point, even when they denote an integer. The decimal point thus indicates that the internal format of this number is floating point and not integer.

```
(%i1)    a:1;
(%o1)                                  1
(%i2)    float(a);
(%o2)                                1.0
```

In scientific notation, the exponent of a floating point number can be separated by either "d", "e", or "f". Output is always returned with "e", as it is used in all internal calculations. Up to a certain number of digits, floating points given in scientific notation are returned in normal, non-exponential form.

```
(%i1)    a:2.3e3;
(%o1)                                2300.0
(%i2)    b:3.456789e-47
(%o2)                          3.456789e-47
```

The file `scientific-engineering-format.lisp`[1], if loaded, provides a feature for having all floating points be returned in scientific notation, with one non-zero digit in front of the decimal point and the number of significant digits according to the value of *fpprintprec*. This feature is activated by setting the option variable *scientific_format_floats*.

```
(%i1)    load("scientific-engineering-format.lisp")$
(%i2)    scientific_format_floats:true$
(%i3)    a:2300.0;
(%o3)                                2.3e3
```

Another feature of this file allows for all floating points to be returned in engineering format, that is with an exponent that is a multiple of three, with 1-3 non-zero digits in front of the decimal point and the number of significant digits according to the value of *fpprintprec*. If set, *engineering_format_floats* overrides *scientific_format_floats*.

```
(%i1)    engineering_format_floats:true$
(%i2)    b:0.23
(%o2)                                230.0e-3
```

If any element of an expression that does not contain bigfloats evaluates to a floating point number, then all other numbers in this expression are, when evaluated, transformed to floating point, and the numerical value returned is a floating point number.

```
(%i1) a:1/4;   b:23.4e2;
                                       1
(%o1)                                 - -
                                       4
(%o2)                                2340.0
(%i2) a+b+c;
(%o2)                          2340.25 + c
```

[1] RS only. In standard Maxima the file engineering-format.lisp provides only the engineering format.

7.1.3.2 Big floating point numbers

In principal, big floating point numbers (bigfloats) can have an unlimited presicion.

Bigfloats are always represented in scientific notation, the exponent being separated by "b".

If any element of an expression evaluates to a bigfloat number, then all other numbers in this expression, including ordinary floating point numbers, are, when evaluated, converted to bigfloats, and the numerical value returned is a bigfloat.

bfloatp (expr) [Predicate function]

If *expr* evaluates to a big floating point number, *true* is returned. In all other cases *false* is returned.

bfloat(expr) [Function]

Converts all numbers in *expr* to bigfloats and returns a bigfloat. The number of significant digits in the returned bigfloat is specified by the option variable *fpprec*.

fpprec Default value: 16 [Option variable]

Sets the number of significant digits for output of and for arithmetic operations on bigfloat numbers. This does not affect ordinary floating point numbers.

```
(%i1)    bfloat(%pi);
(%o1)                      3.141592653589793b0
(%i2)    fpprec:32$  bfloat(%pi);
(%o3)              3.1415926535897932384626433832795b0
```

7.1.4 Complex numbers

7.1.4.1 Introduction

7.1.4.1.1 Imaginary unit

In Maxima the imaginary unit i with $i^2 = -1$ is written as *%i*.

```
(%i1)    sqrt(-1);
(%o1)    %i
(%i2)    %i^2;
(%o2)    -1
```

7.1.4.1.2 Internal representation

There is no generic data type for complex numbers. Maxima represents them internally as a sum $a + ib$, *realpart* and *imagpart* each being of one of the four generic types of numbers.

```
(%i1) a: 3+%i*5;
(%o1)                                5 %i + 3
(%i2) :lisp $a
((MPLUS SIMP) 3 ((MTIMES SIMP) 5 $%I))
(%i2) p: polarform(a);
```

(%o2) $\sqrt{34}\, e^{i \arctan \frac{5}{3}}$

```
(%i3) :lisp $p
((MTIMES SIMP) ((MEXPT SIMP) 34 ((RAT SIMP) 1 2))
((MEXPT SIMP) $%E ((MTIMES SIMP) $%I ((%ATAN SIMP) ((RAT SIMP) 5 3)))))
```

7.1.4.1.3 Canonical order

The canonical order in which Maxima returns a complex-valued expression in standard form *a+ib* might not always seem very logical. Symbols are returned in inverse alphabetical order, no matter whether they belong to the real or the imaginary part, that is, whether they are multiplied by %i or not. In imaginary elements, %i preceeds the symbol. Numerical constants follow the symbols, the ones containing %i preceeding the other ones. Within an imaginary element, the number preceeds %i. However, in a sum of two elements, one being positive and the other one negative, the positive element is always situated in front.
If *powerdisp* is set, the order of the sum is turned around, but not the order of the product within imaginary elements.

```
(%i5)  powerdisp:false$  /* default */
       a + b*%i;
       1 + 2*%i;
       1 + b*%i;
       -b*%i +1;
(%o2)                          i*b + a
(%o3)                          2*i + 1
(%o4)                          i*b + 1
(%o5)                          1 - i*b
(%i6)  z+k*%i+b+a*%i+4+3*%i+2-%i;
(%o6)                    z+i*k+b+i*a+2*i+6
```

7.1.4.1.4 Standard form and polar form

Maxima distinguishes standard form and polar form of complex-valued expressions. The standard form is obtained by function *rectform*, the polar form by function *polarform*. We get the real part of an expression in standard form with function *realpart*, the imaginary part with *imagpart*. Function *cabs* returns the complex absolute value, *carg* the complex argument of an expression in polar form.

7.1.4.1.5 Simplification

Complex expressions are, in contrast to real ones, not always simplified as much as possibel automatically. Products of complex expressions can be simplified by expanding them. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using rectform.

7.1.4.1.6 Properties

Properties for complex numbers include real, complex, imaginary.

7.1.4.1.7 Code

Files: conjugate.lisp

7.1.4.1.8 Generic complex data type

There have been atempts in Maxima to introduce a generic data type for complex numbers, see Maxima-discuss "Complex numeric type - almost done in numeric.lisp but not activated - why?" (August 2017).

7.1.4.2 Standard form

rectform (expr) [Function]

Converts a complex expression to standard form $a + ib$ with $a, b \in \mathbb{R}$. While the imaginary part is parenthesized, if it contains more than one element, this is not done for the real part. Maxima's rules for canonical order imply that the real part may appear before or after the imaginary part and even be split.

```
(%i1)    rectform(z+k*%i+b+a*%i+4+3*%i+2-%i);
(%o1)                          z+i*(k+a+2)+b+6
(%i2)    rectform(sqrt(2)*%e^(%i*%pi/4));
(%o2)                              i + 1
```

realpart (expr) [Function]

Returns the real part of *expr*. realpart and imagpart will work on expressions involving trigonometric and hyperbolic functions, as well as square root, logarithm, and exponentiation.

imagpart (expr) [Function]

Returns the imaginary part of *expr*.

7.1.4.3 Polar form

polarform (expr) [Function]

Converts a complex expression to the equivalent polar form $r\,e^{i\varphi}$ with r being the complex absolute value and φ the complex argument.

cabs (expr) [Function]

Returns the complex absolute value of *expr*.

carg (expr) [Function]

Returns the complex argument of *expr*.

7.1.4.4 Complex conjugate

conjugate (expr) [Function]

Returns the complex conjugate of *expr*. Symbols, unless declared otherwise (complex, imaginary) or evaluating to a complex expression, are considered real. *conjugate* knows identities involving complex conjugates and applies them for simplification, if it can determine that the arguments are complex.

```
(%i1)   conjugate (a + b*%i);
(%o1)                              a-ib
(%i2)   conjugate (c);
(%o2)                               c
(%i3)   declare (d, imaginary)$
(%i4)   conjugate (d);
(%o4)                              -d
(%i5)   polarform(1+2*%i);
```

(%o5) $\sqrt{5}\,e^{i \arctan 2}$

```
(%i6)   conjugate(%);
```

(%o6) $\sqrt{5}\,e^{-i \arctan 2}$

```
(%i7)   conjugate(a1*a2);
(%o7)                            a1 a2
(%i7)   declare ([z1,z2], complex)$
(%i8)   conjugate(z1*z2);
```

(%o8) $\overline{z1}\ \overline{z2}$

```
(%i9)   f:a+b*%i$
(%i10)  (f+conjugate(f))/2;
(%o10)                              a
```

7.1.4.4.1 Internal representation

Internally, the complex conjugate is represented in the following way:

```
(%i1)   declare(a,complex)$
(%i2)   b:conjugate(a);
```

(%o8) $\overline{a}$

```
(%o10)  :lisp $b
(($CONJUGATE SIMP) $A)
```

7.1.4.5 Predicate function

complexp [Self-written predicate function]

If *expr* evaluates to a complex number, *true* is returned. In all other cases *false* is returned.

```
complexp(expr):=if numberp(float(realpart(expr)))
and numberp(float(imagpart(expr))) then true;

(%i1)   complexp(2/3);
(%o1)                              true
(%i2)   complexp((2+3*%i)/(5+2*%i));
(%o2)                              true
(%i3)   polarform(2+3*%i);
```

(%o3) $\sqrt{(13)}\,e^{i \arctan \frac{3}{2}}$

```
(%i4)    complexp(%);
(%o4)                                true
(%i5)    complexp(3*cos(%pi/2)+7*%i*sin(0.5));
(%o5)                                true
(%i6)    complexp(a+b*%i);
(%o6)                                false
```

Chapter 8

Expressions, operators

Chapter 9

Evaluation

Chapter 10

Simplification

10.1 Properties for simplification

10.2 Functions for simplification

10.3 Self-written simplfication functions

10.3.1 Pull minus into fraction

pull_minus_into_fraction (expr, num_denom, side) [Self-written funktion]

A minus sign in front of an expression or any side of an equation is pulled into the numerator (1) or denominator (2) of this fraction. Only if *expr* is an equation, the side (lhs=1, rhs=2) is given as a third parameter.

```
pull_minus_into_fraction(expr,num_denom,[side]) := block([],
  if op(expr)="=" then (                          /* Main operator is "=" */
    expr: substpart("+",expr,side[1],0),
    substpart(-part(expr,side[1],num_denom),expr,side[1],num_denom)
  )
  else (                                          /* Main operator is "-" */
    expr: substpart("+",expr,0),
    substpart(-part(expr,num_denom),expr,num_denom)
  )
)$
```

Chapter 11

Properties, contexts and facts

11.1 Maxima's database for properties and facts

In a running session Maxima keeps a database where some of the properties and all facts and contexts are stored.

11.2 Properties

11.2.1 Introduction

A symbol in Maxima can not only have a value, but also properties. In fact, *value* itself is a system-declared property of a symbol, indicating that it has been bound to a value.

If a user defines a function f, the symbol f is declared the property *function* by the system. Nevertheless, it can be bound to a value, too, and thus is declared the property *value* by the system in addition. f will now behave as a variable or as a function, depending on the context. If the user removes the property *function* from f, its function declaration will be lost and it will behave solely as a variable. If the user removes *value*, too, the symbol f will again be unbound and have no properties at all.

There are pre-defined properties, called user-declared properties, which the user can assign to a variable or function with *declare*, or again removed with *remove*. These informations are considered by the simplifier and other Maxima functions. There are general (*featurep*) or specific, e.g. *constantp*, predicate functions which can test a certain symbol for having a specific user-declared or user-defined property. *properties* returns all properties associated with a specific symbol. *propvars* returns a list of all atoms that have a specific user-declared or user-defined property. *props* is a list containing all atoms that have been assigned any user-declared or user-defined property.

A new property p_u can even be defined by the user with *declare(p_u, feature)*.

Maxima stores the properties associated with a symbol in different places. Some of them are stored in Maxima's database, while others are stored in the Lisp feature list associated with this symbol.

Unlike values, properties (except for the property *value*) are global in Maxima. Thus, a property assigned to a local variable inside of a local environment (like a block or a function) will remain associated with this symbol outside of the block or function

(after it has been called). This holds in particular for function definitions: a function defined inside of a block will be global (once the block has been evaluated). In order to prevent properties of a local variable *a* to become global, the variable has to be declared *local (a)* inside of the local environment.

kill (a) will not only unbind the symbol *a*, but also remove all associated properties.

11.2.2 System-declared properties

There are properties declared by Maxima that cannot be declared by the user, e.g. *function*, *macro*, or *mode_declare*. System-declared properties, however, can be removed by the user, see *remove*.

11.2.3 User-declared properties

11.2.3.1 Properties of variables

integer [property]
noninteger [property]

Tells Maxima to recognize a_j as an integer or noninteger variable. Function *askinteger* recognize this property, but *integerp* does not.

even [property]
odd [property]

Tells Maxima to recognize a_j as an even or odd integer variable. The properties even and odd are recognized by function *askinteger*, but not by the predicate functions *evenp*, *oddp*, and *integerp*.

```
(%i1)   declare(n, even);
(%o1)                                done
(%i2)   askinteger(n, even);
(%o2)                                yes
(%i3)   askinteger(n);
(%o3)                                yes
(%i4)   evenp(n);
(%o4)                               false
```

rational [property]
irrational [property]

Tells Maxima to recognize a_j as a rational variable or an irrational real variable.

real [property]
complex [property]
imaginary [property]

Tells Maxima to recognize a_j as a real, pure imaginary, or complex variable.

constant [property]

The declaration of a_j to be *constant* does not prevent the assignment of a non-constant value to a_j. Such an assignment, on the other hand, does not remove the property *constant* from a_j. The following predicate function *constantp* not only tests for a variable declared *constant*, but for a constant expression in general.

constantp (expr) [predicate function]

Returns *true*, if expr is a constant expression, otherwise *false*. An expression is considered a constant expression, if its arguments are numbers (including rational numbers as displayed with /R/), symbolic constants such as %pi, %e, or %i, variables bound to a constant or declared *constant* by declare, or functions whose arguments are constant. *constantp* evaluates its arguments. See the property *constant* which declares a symbol to be constant.

scalar [property]
nonscalar [property]

Tells Maxima to recognize a_j as a scalar or nonscalar variable. The usual application is to declare a variable as a symbolic vector or matrix. Makes a_j behave as does a list or matrix with respect to the dot operator. The following predicate functions *scalarp* and *nonscalarp* not only test variables declared scalar or nonscalar.

scalarp (expr) [predicate function]
nonscalarp (expr) [predicate function]

scalarp returns *true*, if *expr* is a number, a constant, or a variable declared *scalar*, or composed entirely of numbers, constants, and such declared variables, but not containing matrices or lists. *nonscalar* returns *true* if *expr* contains atoms declared *nonscalar*, or lists, or matrices.

nonarray [property]

Tells Maxima to consider a_j not to be an array. This prevents multiple evaluation of a subscripted variable.

11.2.3.2 Properties of functions

integervalued [property]

Tells Maxima to recognize a_j as an integer-valued function.

increasing [property]
decreasing [property]

Tells Maxima to recognize a_j as an increasing or decreasing function.

```
(%i1)   assume(a > b);
(%o1)                              [a > b]
(%i2)   is(f(a) > f(b));
(%o2)                              unknown
(%i3)   declare(f, increasing);
(%o3)                               done
(%i4)   is(f(a) > f(b));
(%o4)                               true
```

posfun [property]

Tells Maxima to recognize a_j as a positive function.

evenfun [property]

A function with this property is recognized as an even function. $f(-x)$ will be simplified to $f(x)$.

oddfun [property]

A function with this property is recognized as an odd function. $f(-x)$ will be simplified to $-f(x)$.

outative [property]

If a function has this property and it is applied to an argument forming a product, constant factors are pulled out on simplification. Constants in this sense are numbers, standard Maxima constants such as %e, %i or %pi, and variables that have been declared *constant*.

```
(%i1)  declare(f,outative)$
(%i2)  f((r-2+%e^%i)*x);
(%o2)                          f((r+e^i-2)x)
(%i3)  declare(r,constant)$
(%i4)  f((r-2+%e^%i)*x);
(%o4)                          (r+e^i-2)f(x)
```

The standard functions *sum*, *integrate* and *limit* are by default *outative*. However, this property can be removed from them by the user.

additive [property]

If a function has this property and it is applied to an argument forming a sum, the function is distributed over this sum, i.e. f(y+x) will simplify to f(y)+f(x).

linear [property]

Equivalent to declaring a_j both outative and additive.

multiplicative [property]

If a function has this property and it is applied to an argument forming a product, the function is distributed over this product, i.e. f(y*x) will simplify to f(y)*f(x).

commutative [property]
symmetric [property]

These two properties are synonyms. If assigned to a function $f(x,z,y)$, it will be simplified to $f(x,y,z)$.

antisymmetric [property]

If assigned to a function $f(x,y,z)$, it will be simplified to $-f(x,y,z)$. That is, it will give $(-1)^n$ times the result given by *symmetric* or *commutative*, where n is the number of interchanges of wo arguments necessary to convert it to that form.

lassociative [property]
rassociative [property]

A function with this property is recognized as being left-associative or right-associative.

11.2.4 Functions and system variables for properties

declare ($a_1, p_1, a_2, p_2, \ldots, a_n, p_n$) [function]

Assigns property (or list of properties) p_j to atom (or list of atoms) a_j, $j = 1, \ldots, n$. Atoms may be variables, functions, operators, etc. Arguments are not evaluated. *declare* always returns *done*. To remove a property from an atom, see *remove*. To test whether an atom has a specific (user-declared or user-defined) property, see featurep. To show all properties of an atom, see *properties*. To show all atoms with a specific property, see *propvars*. For the use of *declare* to create user-defined properties, see declare (p_u, feature).

```
(%i1)   declare(a,outative,b,additive)$
(%i2)   declare([r,s,t],real)$
(%i3)   declare(c,[constant,complex])$
```

properties (*a*) [function]

Returns a list of all properties associated with atom a.

props [system variable]

This system variable contains a list of all atoms that have been assigned any user-declared or user-defined property. See function *propvars* to show a sublist of *props* containing only the atoms with a specific property.

propvars (*p*) [function]

Returns a sublist of those atoms on the system list *props* which have the property indicated by p.

remove ($a_1, p_1, a_2, p_2, \ldots, a_n, p_n$) [function]

Removes property (list of properties) p_j from atom (list of atoms) a_j, $j = 1, \ldots, n$. *remove (all, p)* removes the property p from all atoms which have it.

The removed properties may be system-declared properties such as *function*, *macro*, or *mode_declare*. Arguments are not evaluated. *remove* always returns *done*. To assign a property to an atom, see *declare*.

11.2.5 User-defined properties

The user may define new properties by *declare* (p_u,*feature*) and assign them to variables or functions with *declare* in the same way it is done for predefined, user-declared properties. The user-defined properties are kept in the system list *features* together with some (but not all) of the predefined, user-declared properties. The predicate function *featurep* may be used to test a variable or function for having a user-defined (or a predefined, user-declared) property.

declare (p_u,*feature*) [function]

Declares p_u to be a new property. This can now be assigned to variables or functions, tested for, view in lists and removed. User-written functions can then consider this information.

```
(%i1)   declare(new_property, feature);
```

```
(%o1)                                done
(%i2)   declare(a, new_property);
(%o2)                                done
(%i3)   featurep(a,new_property);
(%o3)                                true
(%i4)   a:b;
(%o4)                                 b
(%i5)   featurep(a,new_property);
(%o5)                               false
(%i6)   declare(b,new_property);
(%o6)                                done
(%i7)   featurep(a,new_property);
(%o7)                                true
(%i8)   c:new_property;
(%o8)                          new_property
(%i9)   featurep(a,c);
(%o9)                                true
```

featurep (a, p) [predicate function]

Tries to determine whether atom a has property p. Note that featurep returns *false* also in the case where it cannot determine whether atom a has property p or not. Only user-declared and user-defined properties can be tested with *featurep*, but not system-declared properties.

Note that *featurep* evaluates its arguments. Thus, if a has a value that is itself a variable or function, and if p has a value that is itself a property, then it is the variable or function which is the value of a that is tested for the property which is the value of p.

features [system variable]

This list contains some (but not all) of the predefined, user-declared properties plus all user-defined properties.

11.3 Assumptions and their contexts

11.3.1 Introduction

In Maxima certain assumptions, they are called "facts", can be associated with variables or functions. Such assumptions are limited to statements comprising the relational operators "<", "<=", equal, notequal, ">=" und ">" and some combinations of them with the boolean operators AND and NOT (but not OR). Facts are declared by using function *assume*. They are remove with *forget*. The function *facts* can be used to list all assumptions contained in a certain context, or all assumptions defined for a particular variable or function within the current context.

Some, but not all Maxima functions recognize facts. For example, *solve* does not (it was written before facts and contexts were introduced in Maxima), whereas *to_poly_solve*, a more recent and sometimes more powerful solver, does. User written functions, of course, may also take facts into consideration.

If they need certain assumptions about user variables in order to proceed operating on them, some Maxima functions will ask the user for more information at the time

they are called. This is a useful feature in order to reach computational results, since the user may not be aware of any such necessity in advance. He can, however, declare the corresponding assumptions prior to calling the function in order to avoid these questions.

Facts are stored in Maxima's database. Here they are organized in a hierarchical structure of *contexts*. The context named *gloabal* forms the root of this hieachy, the parent of all other contexts. It contains system information needed by some functions. When Maxima is started, the user sees a child context of *global* named *initial*. If he does not specify any other context, all assumptions, that is, all facts created by *assume*, will be stored in this context.

However, the user may create child contexts to any existing context, including *global*. When in any context, all facts within this so-called *current context* and all of its parent contexts are visible and are used in deductions. In addition, the user may activate any other context freely at will with function *activate*. This context and all its parent contexts will then also be available. He can also deactivate contexts with *deactivate*.

Function *context* can be used to show the current context or change it. New contexts are defined by either *newcontext* or *supcontext*. *contexts* gives a list of all contexts presently defined.

The context mechanism makes it possible for a user to bind together and name a collection of facts. Once this is done, the user can have Maxima activate or deactivate large numbers of previously defined facts merely by activating or deactivating their respective context. Facts contained in a context will be retained in storage until destroyed one by one by calling *forget* or as a whole by calling *killcontext* to destroy the context to which they belong.

The terms "subcontext" and "sup(er)context" are used in Maxima, but they have some inherent ambiguity. A child context is always bigger than its parent context as a collection of facts, because the facts a child context contains are added to the facts already active in the line of its parent contexts (it is not possible to deactivate parent contexts to the current or any other explicitly active context). The child context therefore is a superset of the parent context. Thus, function *supcontext* creates a child context to the current context. Parent contexts are called subcontexts. This terminology, however, contradicts the normal description of a tree structure, where one would naturally tend to name a leave a sub-element to its parent. There is another interpretation contradicting the terminology used in Maxima. If a context is bigger because it contains more facts, on the other hand it is smaller, because every additional fact narrows and constrains the possibilities for the corresponding variable or function to take values.

Facts and contexts, like properties, are global in Maxima. But unlike a property, a fact or context *b* cannot even be made local inside of a block or function by using *local(b)*.

Killing a variable or function *a* with *kill(a)* will not delete facts associated with *a*. Only *kill(all)* will delete everything, including the defined facts and contexts.

11.3.2 Assumptions

assume $(pred_1, pred_2, \ldots, pred_n)$ [function]

Adds predicates $pred_1, pred_2, \ldots, pred_n$ to the current context. If a predicate is inconsistent or redundant with the predicates in the current context, it is not added. *assume* returns a list whose elements are the predicates added to the context, or *redundant*, *inconsistent* or *meaningless* where applicable. *assume* evaluates its arguments. The context accumulates predicates from each call to *assume*. *assume* does not except a list of predicates as does *forget*.

The predicates defined may only be expressions with the relational operators $<, \leq$, equal (a, b), notequal (a, b), $\geq$ and $>$. Predicates cannot be literal equality (=) or literal inequality (#) expressions, nor can they be predicate functions such as *integerp*. *assume* does not allow predicates with complex numbers either.

Boolean compound predicates of the form "$pred_1$ AND $\ldots$ AND $pred_n$" are recognized, but not "$pred_1$ OR $\ldots$ OR $pred_n$". "NOT $pred_k$" is recognized, if $pred_k$ is a relational predicate. Expressions of the form "NOT ($pred_1$ AND $pred_2$)" and "NOT ($pred_1$ OR $pred_2$)" are not recognized.

Maxima's deduction mechanism is not very strong; there are many obvious consequences which cannot be determined by *is*. This is a known weakness.

```
(%i1)   assume (x > 0, y < -1, z >= 0);
(%o1)                        [x > 0, y < - 1, z >= 0]
(%i2)   assume (a < b and b < c);
(%o2)                              [b > a, c > b]
(%i3)   assume (2*b < 2*c);
(%o3)                                redundant
(%i4)   assume (c < b);
(%o4)                               inconsistant
(%i5)   facts ();
(%o5)                [x > 0, - 1 > y, z >= 0, b > a, c > b]
(%i6)   is (x > y);
(%o6)                                  true
(%i7)   is (y < -y);
(%o7)                                  true
(%i8)   is (sinh (b - a) > 0);
(%o8)                                  true
(%i9)   forget (b > a);
(%o9)                                 [b > a]
(%i10)  is (sinh (b - a) > 0);
(%o10)                                unknown
(%i11)  is (b^2 < c^2);
(%o11)                                unknown
```

forget $(pred_1, pred_2, \ldots, pred_n)$ [function]
forget (L)

Removes predicates from the current context. Alternatively, the arguments can be passed to *forget* as a list L. *forget* evaluates its arguments. In a very limited way, the predicates may be equivalent (not necessarily identical) expressions to those previously assumed (e.g., b*2>4 eliminates b>2, but 2*a<2*b does not eliminate a<b).

forget does not complain if a predicate to be forgotten does not exist. In any case, $pred_1, pred_2, \ldots, pred_n$ or L is returned.

facts (item) [function]
facts ()

If *item* is the name of a context, which is either the current context, a parent of it, a context on the list *activecontexts*, or a parent of it, *facts (item)* returns a list of the facts in the specified context. In the case of all other contexts, it returns an empty list. If *item* is not the name of a context, *facts (item)* returns a list of the facts known about *item* in the current context.

facts () returns a list of the facts in the current context.

is (expr) [function]

ev(expr, pred), which can be written *expr, pred* at the interactive prompt, is equivalent to *is(expr)*.

Attempts to determine whether the predicate *expr* is provable from the facts in the database. If the predicate is provably *true* or *false*, *is* returns this respectively. Otherwise, the return value is governed by the global flag *prederror*. When it is not set (default), it returns *unknown*. Otherwise, *is* returns an error message.

is also evaluates any predicate, independently of the facts database. Special attention has to be paid for test of equality. *is(a=b)* tests a and b to be literally equal, that is identical. *is(equal(a,b))* tests for equivalence, which does not necessarily imply literal identity:

```
(%i1)   is (%pi > %e);
(%o1)                                true
(%i2)   c: (x - 1) * (x + 1) $
(%i3)   d: x^2 - 1 $
(%i4)   is(c = d);
(%o4)                                false
(%i5)   is(equal(c,d));
(%o5)                                true
(%i6)   d: 5 $
(%i7)   is(equal(d,5));
(%o7)                                true
(%i8)   is(d=5);
(%o8)                                true
(%i9)   is(integerp(d));
(%o9)                                true
```

is attempts to derive predicates from the facts database:

```
(%i1)   assume (a > b);
(%o1)                              [a > b]
(%i2)   assume (b > c);
(%o2)                              [b > c]
(%i3)   is (a + b > b + c);
(%o3)                               true
(%i4)   is (equal (a, c));
(%o4)                               false
(%i5)   is (2*a > 3*c);
(%o5)                              unknown
(%i6)   assume (equal(d,5));
(%o6)                          [equal(d,5)]
(%i7)   is (equal (d, 5));
```

```
(%o7)                                true
(%i8)    is (d=5);
(%o8)                                false   (!)
```

If *is* can neither prove nor disprove a predicate by itself of from the facts database, the global flag *prederror* governs the behavior of *is*:

```
(%i1)    assume (a > b);
(%o1)                               [a > b]
(%i2)    prederror: true$
(%i3)    is (a > 0);
Maxima was unable to evaluate the predicate: a > 0
-- an error. Quitting. To debug this try debugmode(true);
(%i4)    prederror: false$
(%i5)    is (a > 0);
(%o5)                               unknown
```

11.3.3 Contexts

context default: *initial* [system variable and function]

The value of *context* indicates the current context. Binding *context* to a symbol *name* will change the current context to *name*. If a context with this name does not yet exist, it is created as a direct child to *global* (as done with function *newcontext*) and then made to be the current context.

contexts default: [*initial*, *global*] [system variable]

This is a list of all contexts which are currently defined.

newcontext (name) [function]

Creates a new context as a direct child to *global* and makes it the current context. If *name* is not specified, a pair of empty parenthses has to remain. In this case, a new name is created at random by the *gensym* function. *newcontext* evaluates its argument. *newcontext* returns name (if specified) or the newly created context name.

supcontext (name, cont) [function]

Creates a new context *name* as a direct child to *cont* and makes it the current context. If *context* is not specified, the current context will be the parent. If *name* is not specified, a pair of empty parenthses has to remain. In this case, a new name is created at random by the *gensym* function and the current context is used as parent. *supcontext* evaluates its arguments. *supcontext* returns name (if specified) or the newly created context name.

activate ($context_1, ..., context_n$) [function]

Adds the contexts $context_1, ..., context_n$ to the list *activecontexts*. The facts in these contexts are then available to make deductions. *activate* returns *done* if the contexts exist, otherwise an error message.
Note that by activating a context, the facts of all its parent contexts also become available for deductions, although these parent contexts are not added to the list *activecontexts*.

deactivate ($context_1, ..., context_n$) [function]

Removes the contexts $context_1, ..., context_n$ from the list *activecontexts*. The facts in these contexts are then no longer available to make deductions. *deactivate* returns *done* if the contexts exist (even if this context cannot be deactivated), otherwise an error message.

Note that it is only possible to deactivate contexts that have previously been activated by *activate*. Facts within parent contexts of a context removed from the list *activecontexts* are also no longer available for deductions, unless these contexts are the current context or a parent of it, or any other context remaining on the list *activecontexts* or any parent of it.

activecontexts [system variable]

This is a list of all contexts currently activated with function *activate*. Note, however, that this list does not include the (active) parent contexts of an activated context.

killcontext ($context_1, ..., context_n$) [function]

Kills the contexts $context_1, ..., context_n$. *killcontext* evaluates its arguments. *killcontext* returns done. If one of the killed contexts is the current context, its next available direct parent context will become the new current context. If context *initial* is killed, a new, empty *initial* context is created. If a killed context has childs, they will be connected to the next available parent of the killed context. *killcontext*, however, refuses (by returning a corresponding message) to kill a context which is on the list *activecontexts* or to kill context *global*.

Chapter 12

Rules and patterns

tellsimp (pattern, replacement) [function]

Establishes a user-defined simplification rule that will be applied by the simplifier automatically to any expression before applying the built-in simplification rules. See *tellsimpafter* for user-defined rules that will be applied after the built-in simplification rules.

pattern is an expression comprising pattern variables (declared by *matchdeclare*) and other atoms and operators. *replacement* is substituted for an actual expression which matches *pattern*. Pattern variables in *replacement* are assigned the values matched in the actual expression.

pattern may be any nonatomic expression in which the main operator is not a pattern variable. The newly defined simplification rule is associated with *pattern*'s main operator, as it is done for the built-in simplification rules. *tellsimp* returns the list of all simplification rules for the main operator of *pattern*, including the newly established rule. (Thus, this function can also be used to see what are the built-in simplification rules for a given main operator.)

The rule constructed by *tellsimp* is named after *pattern*'s main operator. Rules for built-in operators and user-defined operators defined by infix, prefix, postfix, matchfix and nofix have names which are Lisp identifiers. Rules for other functions have names which are Maxima identifiers.

Rules defined with tellsimp are applied after evaluation of an expression (if not suppressed through quotation or the flag noeval). They are applied in the order they were defined, and before any built-in rules. Rules are applied bottom-up, that is, applied first to subexpressions before applied to the whole expression. It may be necessary to repeatedly simplify a result (e.g. via the quote-quote operator '' or the flag infeval) to ensure that all rules are applied.

tellsimp does not evaluate its arguments.

tellsimpafter (pattern, replacement) [function]

Establishes a user-defined simplification rule that will be applied by the simplifier automatically to any expression after having applied the built-in simplification rules. See *tellsimp* for rules that will be applied before the built-in simplification rules.

Part IV

Basic mathematical computation

Chapter 13

Root, exponential and logarithmic functions

13.1 Roots

13.1.1 Vereinfachungen

radexpand Default: *true* [Optionsvariable]

13.2 Exponential function

exp (expr) [Funktion]

exp ist die natürliche Exponentialfunktion. Maxima vereinfacht `exp(x)` sofort zu e^x.

13.2.1 Vereinfachungen

radcan (expr) [Funktion]

Die Funktion *radcan* vereinfacht Ausdrücke, die die Exponentialfunktion, den Logarithmus und Wurzeln enthalten.

```
(%i2)  (%e^x-1)/(1+%e^(x/2));
       radcan(%);
```

(%o1)
$$\frac{e^x-1}{e^{\frac{x}{2}+1}}$$

(%o2)
$$e^{\frac{x}{2}-1}$$

logsimp Standardwert: *true* [Optionsvariable]

Ist die Optionsvariable *logsimp* gesetzt, wird eine Exponentialform `%e^(r*log(x))` $\equiv$ $e^{r\,ln(x)}$ zu `x^r` vereinfacht, falls $r \in \mathbb{Z}$.

%e_to_numlog Standardwert: *false* [Optionsvariable]

Ist die Optionsvariable *%e_to_numlog* gesetzt, wird eine Exponentialform der Art `%e^(r*log(x))` $\equiv$ $e^{r\,ln(x)}$ zu `x^r` vereinfacht, falls $r \in \mathbb{Q}$.

demoivre Standardwert: *false* [Optionsvariable]

Ist die Optionsvariable *demoivre* gesetzt, wird eine Exponentialform `%e^(a+%i*b)` $\equiv$ e^{a+ib} mit $a, b \in \mathbb{R}$, also mit komplexem Exponenten in Standardform, mit der Euler'schen Formel zu `%e^a*(cos(b)+%i*sin(b))` $\equiv$ $e^a(\cos b + i \sin b)$, also zu einem äquivalenten Ausdruck mit Kreisfunktionen, umgeformt.

Die Optionsvariable *exponentialize* führt die gegenteilige Umformung durch. Es können also nicht beide Optionsvariablen gleichzeitig gesetzt sein. Beide Umformungen können auch durch Funktionen gleichen Namens bewirkt werden, ohne daß die Optionsvariablen gesetzt sind.

```
(%i4)  %e^(a+ %i*b);
       %e^(a+ %i*b), demoivre:true;
       %, exponentialize:true;
       radcan(%);
```

(%o1) $$e^{a+ib}$$

(%o2) $$e^a(\cos b + i \sin b)$$

(%o3) $$e^a\left(\frac{e^{ib}-e^{-ib}}{2}+\frac{e^{ib}+e^{-ib}}{2}\right)$$

(%o4) $$e^{a+ib}$$

%emode Standardwert: *true* [Optionsvariable]

Ist die Optionsvariable *%emode* gesetzt, wird eine Exponentialform `%e^(%i*%pi*x)` $\equiv$ $e^{i\pi x}$ vereinfacht

- falls x eine ganze Zahl, ein ganzzahliges Vielfaches von 1/2, 1/3, 1/4 oder 1/6 oder eine Gleitkommazahl ist, die einer ganzen oder halbganzzahligen Zahl entspricht: nach der Euler'schen Formel zu einer komplexen Zahl in der Standardform `cos(%pi*x)+%i*sin(%` und dann wenn möglich weiter vereinfacht,

- für andere rationale x zu einer Exponentialform `%e^(%i*%pi*y)`, mit $y = x - 2k$ für ein $k \in \mathbb{N}$, sodaß $|y| < 1$ ist.

Eine Exponentialform `%e^(%i*%pi*(x+y))` $\equiv$ $e^{i\pi(x+y)}$ wird zu $e^{i\pi x}e^{i\pi y}$ umgeformt und dann der erste Faktor entsprechend vereinfacht, wenn y ein Polynom oder etwa eine trigonometrische Funktion ist, nicht jedoch, wenn y eine rationale Funktion ist.

Wenn mit komplexen Zahlen in Polarkoordinatenform gerechnet werden soll, kann es hilfreich sein, *%emode* auf den Wert *false* zu setzen.

%enumer Default: *false* [Option variable]

In an exponential form with floating point exponent, %e is always evaluated to floating point, and therefore the whole form. If both *%enumer* and *numer* are true, %e is evaluated to floating point in any expression.

Chapter 14

Limits

Chapter 15

Sums, products and series

15.1 Sums and products

15.1.1 Sums

15.1.1.1 Introduction

Sums can be created with function *sum*. They can be displayed in sigma notation, simplified and evaluated. Sums can also be differentiated or integrated, and they can be subject to limits.

15.1.1.2 Constructing, simplifying and evaluating sums

sum (expr, i, i_0, i_1) [function]

Builds a summation of *expr* (evaluated) as the summation index i (not evaluated) runs from i_0 to i_1 (both evaluated). Both a noun form and a sum that on simplification and evaluation cannot be resolved are displayed in sigma notation.

```
(%i1) 'sum(1/k!,k,0,4);
```

(%o1)
$$\sum_{k=1}^{4} \frac{1}{k!}$$

```
(%i2) sum(1/k!,k,0,4);
```

(%o2)
$$\frac{65}{24}$$

```
(%i3) sum(1/k!,k,1,n);
```

(%o3)
$$\sum_{k=1}^{n} \frac{1}{k!}$$

```
(%i4) sum (a[i], i, 1, 5);
```

(%o4)
$$a_1 + a_2 + a_3 + a_4 + a_5$$

```
(%i5) sum (a(i), i, 1, 5);
(%o5)                a(5) + a(4) + a(3) + a(2) + a(1)
```

Some basic rules are applied automatically to simplify sums. More rules are activated by setting flag *simpsum*.

simpsum default: *false* [option variable]

When simpsum is set, the result of a sum is simplified. This simplification may sometimes be able to produce a closed form.

```
(%i6) sum (2^k + k^2, k, 0, n);
```

(%o6) $$\sum_{k=0}^{n} \left(2^k + k^2\right)$$

```
(%i7) sum (2^k + k^2, k, 0, n), simpsum;
```

(%o7) $$2^{n+1} + \frac{2\,n^3 + 3\,n^2 + n}{6} - 1$$

Package simplify_sum contains function *simplify_sum* which is even more powerful in finding closed forms.

simplify_sum (expr) [function in: simplify_sum]

<Text>

```
(%i8) load(simplify_sum);
(%o8) "C:/maxima-5.40.0/../share/maxima/5.40.0/share/solve_rec/simplify_sum.mac"
(%i9) simplify_sum(sum(2^k+k^2,k,0,n));
```

(%o9) $$2^{n+1} + \frac{2\,n^3 + 3\,n^2 + n}{6} - 1$$

15.1.1.3 Differentiation and integration of sums

Sums can be differentiated and integrated.

```
(%i1) s:sum((x-x0)^k,k,1,n);
```

(%o1) $$\sum_{k=1}^{n} (x - x_0)^k$$

```
(%i2) 'diff(s,x) = diff(s,x);
```

(%o2) $$\frac{d}{dx} \sum_{k=1}^{n} (x - x_0)^k = \sum_{k=1}^{n} k\,(x - x_0)^{k-1}$$

```
(%i3) 'integrate(s,x) = integrate(s,x);
```

(%o3) $$\int \sum_{k=1}^{n} (x - x_0)^k \, dx = \sum_{k=1}^{n} \frac{(x - x_0)^{k+1}}{k + 1}$$

15.1.1.4 Limits of sums

Sums can be subject to limits.

15.2 Series

In Maxima a series is represented by function *sum* with the upper bound set to *inf*.

15.2.1 Power series

15.2.2 Taylor series

Chapter 16

Differentiation

Chapter 17

Integration

Chapter 18

Solving Equations

Chapter 19

Differential Equations

Chapter 20

Polynomials

Chapter 21

Linear Algebra

Part V

Advanced Mathematical Computation

Chapter 22

Tensors

Chapter 23

Numerical Computation

Part VI

Maxima Programming

Chapter 24

Functions

Chapter 25

Program Flow

Chapter 26

MaximaL Programming and Debugging

26.1 Debugging MaximaL

26.1.1 Break commands

Break commands are special MaximaL commands which are not interpreted as Maxima expressions. A break command can be entered at the Maxima prompt or the debugger prompt (but not at the break prompt). Break commands start with a colon, ":".

For example, to evaluate a Lisp form you may type :lisp followed by the form to be evaluated. (Chapter 38: Debugging 635 5 The number of arguments taken depends on the particular command. Also, you need not type the whole command, just enough to be unique among the break keywords. Thus :br would suffice for :break. The keyword commands are listed below. :break F n Set a breakpoint in function F at line offset n from the beginning of the function. If F is given as a string, then it is assumed to be a file, and n is the offset from the beginning of the file. The offset is optional. If not given, it is assumed to be zero (first line of the function or file). :bt Print a backtrace of the stack frames :continue Continue the computation :delete Delete the specified breakpoints, or all if none are specified :disable Disable the specified breakpoints, or all if none are specified :enable Enable the specified breakpoints, or all if none are specified :frame n Print stack frame n, or the current frame if none is specified :help Print help on a debugger command, or all commands if none is specified :info Print information about item :lisp some-form Evaluate some-form as a Lisp form :lisp-quiet some-form Evaluate Lisp form some-form without any output :next Like :step, except :next steps over function calls :quit Quit the current debugger level without completing the computation :resume Continue the computation :step Continue the computation until it reaches a new source line :top Return to the Maxima prompt (from any debugger level) without completing the computation

Part VII

Basic Software Structure

Chapter 27

User Interfaces

Chapter 28

The Build System

Chapter 29

Repository and Tarball Structure

Chapter 30

External Packages

Part VIII

Lisp development

Chapter 31

MaximaL and Lisp interaction

31.1 Maxima and Lisp

Maxima is written in Lisp. Much of the terminology used within Maxima is based on the terminology used in Common Lisp. Since Maxima was, especially in the early phase of the 1960s and 1970s, as part of MIT's project MAC, developed in parallel to Lisp, Maxima's basic and overall design decisions were based on the state of the art of the contemporary Lisp available. The early part of Maxima is written in MACLisp, which was developed as part of MIT's project MAC, too. After the definition of Common Lisp had been established, this has been used for all further developments within Maxima instead, but many parts already written in MACLisp remained in this dialect until today. Common Lisp itself has been refined and enhanced over the years up to todays ANSI standard. While new Lisp developments within Maxima can make use of the entire functionality of this advanced Lisp standard, which most of today's Lisp compilers understand, the major part of Maxima is written using only the language elements of the earlier states of Common Lisp.

31.2 MaximaL and Lisp identifiers

, and it is easy to access Lisp functions and variables from Maxima and vice versa. Lisp and Maxima symbols are distinguished by a naming convention. A Lisp symbol which begins with a dollar sign corresponds to a Maxima symbol without the dollar sign. A Maxima symbol which begins with a question mark ? corresponds to a Lisp symbol without the question mark. For example, the Maxima symbol foo corresponds to the Lisp symbol $FOO, while the Maxima symbol ?foo corresponds to the Lisp symbol FOO. Note that ?foo is written without a space between ? and foo; otherwise it might be mistaken for describe ("foo"). Hyphen -, asterisk *, or other special characters in Lisp symbols must be escaped by backslash where they appear in Maxima code. For example, the Lisp identifier *foo-bar* is written ?foobar in Maxima.

31.3 Executing Lisp code from within MaximaL

31.3.1 Break command ":lisp"

The break command *:lisp* can be used to execute a single Lisp form from the Maxima prompt or the debugger prompt.

Use primitive (i.e. standard CL function) "+" to add the values of MaximaL variables x and y:

```
(%i1)   x:10$ y:5$
(%i3)   :lisp (+ $x $y)
15
```

Use Maxima Lisp function *add* to symbolically add MaximaL variables a and b, and assign the result to c:

```
(%i1)   :lisp (setq $c (add '$a '$b))
((MPLUS SIMP) $A $B)
(%i1)   c;
(%o1)                          b + a
```

Show the Lisp properties of MaximaL variable d:

```
(%i1)   context;
(%o1)                         initial
(%i2)   supcontext(d);
(%o2)                           d
(%i3)   :lisp (symbol-plist '$d)
(subc ($initial))
```

Chapter 32

Lisp Programming and Debugging

Chapter 33

Lisp Program Structure

Bibliography

[CharMap84] B. Char. “On the design and performance of the Maple system.” In: *Proc. of the Macsyma Users Conference* (1984), pp. 199–219.

[ColeSMP81] C.A. Cole and Stephen Wolfram. “SMP: A Symbolic Manipulation Program.” In: (1981).

[FatemThe72] Richard J. Fateman. “Essais on Algebraic Simplification.” MAC TR-95. Thesis. Harvard University, 1972. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.648.2190&rep=rep1&type=pdf.

[MaxiManD11] Dieter Kaiser. *Maxima Manual 5.29 dt.* 2011. URL: http://maxima.sourceforge.net/docs/manual/de/maxima.html.

[MartFate71] William Martin and Richard Fateman. “The MACSYMA system.” In: *Proc. of the 2nd Symposium on Symbolic and Algebraic Manipulation* (1971), pp. 59–75.

[MaximMan17] Maxima. *Maxima Manual 5.41.0 engl.* 2017. URL: http://maxima.sourceforge.net/docs/manual/maxima.html.

[MosesMPH12] Joel Moses. “Macsyma: A personal history.” In: *Journal of Symbolic Computation* 47 (2012), pp. 123–130.

[wikMacsy17] Wikipedia. *Macsyma*. [Online; Stand 26. September 2017]. 2017. URL: https://de.wikipedia.org/w/index.php?title=Macsyma&oldid=781784197.

Index