The Maxima Workbook

Roland Salz

December 10, 2017
Vers. 0.1.4

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, 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 professor of computer science, had gone to the University of California in Berkeley, organized the first one of what would become altogether three 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 3. Macsyma Users' Conference, which took place 1984 in Schenectady, N.Y., home of General Electric Research Labs, 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 Kat-

terjohn, David Billinghurst and Volker van Nek. 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 (Canada) is in charge of new releases and maintains the tensor packages. Jaime Villate (University of Porto, Portugal), contributed to the graphical interface Xmaxima and designed the Maxima homepage. Many more could be mentioned who contribute to Maxima in one way or the other, for instance by writing and providing external software packages. For example, Dimiter Prodanov (Belgium) recently developed a package for Clifford algebras, Serge de Marre, also from Belgium, a package for solving Diophantine equations. 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 chief designers and 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 Introduction

As a general statement, it is our feeling that Maxima's documentation can be improved. Both as a user and even more as a developer one would want to have at hand much more information than what the official manual, the internal documentation and the comments in the program code provide. This deficiency is most visible if it comes to information describing the overall structure of the MaximaL user interface (it comprises of thousands of different functions and option variables), the structure of the huge amount of the program code, or of the complicated information flow within the running system or even the general structure of build process.

This obvious lack of information motivates us to start the Workbook documentation project. But before getting into it, let us find an overview about exactly what sources and what extend of information about Maxima we have available already.

2.2 Manuals

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 each Maxima installer or tarball. It can also be built when building Maxima from source code. The Maxima Workbook is primarily based on this documentation. [MaximManE17]

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, it is currently not being updated. 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]

2.3 Mailing list archives

2.4 Internal and program documentation

2.5 Papers

2.6 External documentation projects

2.7 Tutorials

2.7.1 Ted Woollett: Maxima by example

2.7.2 Roland Stewen: Maxima in Beispielen

Roland Stewen from Rahel Varnhagen Kolleg in Hagen, Germany, has written a Maxima tutorial in German of some 400 pages primarily addressed to highschool students. It is available online in html format and can be downloaded as PDF. The document is clearly written, well structured, contains a detailed table of content, an index, a bibligraphy, and can be highly recommended for the intended purpose. [StewenMT13]

Part II

Basic Operation

Chapter 3

Basics

Chapter 4

Input and output

Chapter 5

Plotting

Chapter 6

Batch Processing

Part III

Concepts of Symbolic 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);
```

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

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
```

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

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

Knowledge database system

In Maxima, variables and user-defined functions can be associated not only with values, but also with *properties* and with *assumptions*. Properties contain information about the *type* of value the respective variable or function is supposed to take, while assumptions limit the *numerical range* of their allowed values. Both categories of information can be used by Maxima or by user-written functions for computation and simplification of expressions comprising these variables.

Maxima's mathematical knowledge database system was written by Michael Genesereth while studying at MIT in the early 1970^s. Today he is professor of computer science at Stanford University.

Before looking closer at the information it contains, namely properties and assumptions, we will focus on general features of this database system. We describe its user interface first, then some aspects of the implementation.

11.1 Facts and contexts: The general system

11.1.1 User interface

11.1.1.1 Introduction

Properties and assumptions associated with a Maxima symbol are called *facts*. There are certain facts already provided by the system, for instance about general and predefined mathematical constants such as e, i or π. In addition, the user may assign one or more of a number of system-defined properties to any of his variables or user functions. He can also define his own new property types, called *features*, and assign them to symbols just like the system-defined properties. Finally, using assumptions, he can impose restrictions on the numerical range of values to be taken by a symbol denoting a variable or function.

Some, but not all Maxima functions recognize facts. For example, *solve* does not consider assumptions (it was written before the knowledge database was introduced into Maxima), whereas *to_poly_solve*, a more recent and sometimes more powerful solver, does. User-written functions, of course, may also take facts into account.

If they need certain information about user variables in order to proceed operating on them, some Maxima functions will ask the user interactively at the time they are called. This is a useful procedure 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 properties or assumptions prior to calling the function in order to avoid these questions.

Maxima's mathematical knowledge database system organizes facts in a hierarchical structure of *contexts*. The context named *global* forms the root of this hierarchy, the parent of all other contexts. It contains information for instance about predefined constants, e.g. %e, %i or %pi, and their respective values. When a Maxima session is started, the user sees a child context of *global* named *initial*. If he does not specify any other context, all facts, that means all properties created by *declare* and all assumptions created by *assume*, will be stored in this context. The context which presently accomodates newly declared assumptions is called the *current context*. Function *facts* may be used to list all facts contained in a certain context, or all facts defined for a particular symbol and kept within the current context.

The user may create child contexts to any existing context, including *global*. The facts that are visible and are used for deductions at any moment are those of the current context *plus all of its parent contexts*. In addition, the user may activate any other context freely at will with function *activate*. This context *plus all of its parent contexts* will then also be visible in addition to the current context and its parents. The user can deactivate any explicitely activated context with *deactivate*. A list of all activated contexts is kept in *activecontexts*.

Function *context* can be used to show the current context or to 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 the user to bind together and name a collection of facts. Once this is done, he can activate or deactivate large numbers of previously defined facts merely by activating or deactivating the 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 context 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. Due to this ambiguity we stay with the parent-child terminology.

Facts and contexts are global in Maxima, even if the corresponding variables are local. However, it is possible to make facts associated with a local variable local, too, by declaring (inside of the local environment) the respective local variable or

function a with the system function *local(a)*.

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.

11.1.1.2 Functions and system variables

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 variable or function *item* in the current context.

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

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 ($context_1, \ldots, context_n$) [function]

Adds the contexts $context_1, \ldots, context_n$ 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 ($context_1, \ldots, context_n$) [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 any one of them 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 explicitely activated with function *activate*. Note that this list does not include the (active) parent contexts of an activated context, nor the current context or any of its parents.

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*.

11.1.2 Implementation

11.1.2.1 Internal data structure

11.1.2.2 Notes on the program code

11.2 Values, properties and assumptions

Values, properties and assumptions are independant of one another. They are not cross-checked.

General statements on values in Lisp and MaximaL.

Predicates sometimes check properties, sometimes values.

Functions on assumptions don't take actual values into consideration.

etc.

11.3 MaximaL Properties

11.3.1 User interface

11.3.1.1 Introduction

In Maxima, variables and user-defined functions can be associated not only with values, but also with *properties*. Properties contain information about the *kind* of variable or function which the respective symbol is to represent, or the *type* of value which the respective variable or function is supposed to take.

The concept of properties is inherent in Lisp. In order to distinguish both types, we will henceforth use the terms *Lisp property* to refer to the properties on the Lisp level, and *MaximaL property* (sometimes also called: *mathematical property*) to refer to the properties on the MaximaL level.

There are three types of MaximaL properties: *system-declared*, *user-declared* (sometimes also called: *system-defined* or *predefined*) and *user-defined* properties. System-declared properties can only be declared for a symbol by the system. User-declared properties are predefined properties which the user can declare for a symbol and remove from it. User-defined properties can be defined by the user and then declared for a symbol and removed.

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) not only unbinds the symbol *a*, but also removes all associated properties.

11.3.1.2 System-declared properties

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

For instance, *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, the user may bind f 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 be unbound again and have no properties at all.

11.3.1.3 User-declared properties

These are pre-defined properties, which the user can assign to a variable or function with *declare*, or delete with *remove*. Properties are recognized 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.

11.3.1.3.1 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, complex or pure imaginary variable.

constant [property]

The declaration of a_j to be *constant* does not prevent the assignment of a nonconstant 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.

11.3.1.3.2 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.

11.3.1.4 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 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*.

11.3.1.5 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.

11.3.2 Implementation

11.4 Assumptions

11.4.1 User interface

11.4.1.1 Introduction

In Maxima, variables and user-defined functions can be associated with so-called assumptions. Assumptions limit the range of values these variables or functions are supposed to take. It is sometimes useful or even necessary to impose such restrictions in order to obtain usable results from symbolic computation. Assumptions can be 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*. See there for details on the assumptions that can be made. Assumptions are remove with *forget*.

11.4.1.2 Functions and system variables for 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 redundant or inconsistent 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 accept a Maxima 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 Maxima 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.

is (expr) [function]

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

is 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*. If it is not set (default), it returns *unknown*. Otherwise, *is* returns an error message.

Note that *is* can evaluate any other predicate, too,independently of the assumptions in the database. Special attention has to be paid for tests 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. Different symbolic expressions, that can be simplified by Maxima to the same (canonical) expression, are considered equivalent.

```
(%i1)   is (%pi > %e);
(%o1)                                true
(%i2)   is(integerp(d));
(%o2)                                true
```

```
(%i3)   c: (x − 1) ∗ (x + 1) $
(%i4)   d: x^2 − 1 $
(%i5)   is(c = d);
(%o5)                                false
(%i6)   is(equal(c,d));
(%o6)                                true
```

is attempts to derive predicates from the facts database. Note that assumptions cannot be tested for literal equality or inequality.

```
(%i1)   assume (a > b, b > c);
(%o1)                          [a > b, 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
```

11.4.2 Implementation

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

21.1 Vectors

21.1.1 Create a vector

covect (L) [function of *eigen*]
columnvector (L) [function of *eigen*]

Returns a column vector which is a matrix of one column and *length (L)* rows, containing the elements of the list L. *covect* is a synonym for *columnvector*.

```
(%i1) covect([x,y,z]);
```

(%o1)

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

cvect $(x_1, x_2, \ldots, x_n)$ [function of *rs_algebra*]
rvect $(x_1, x_2, \ldots, x_n)$ [function of *rs_algebra*]

cvect returns a column vector which is a matrix of one column and n rows, containing the arguments. *rvect* returns a row vector which is a matrix of one row and n columns, containing the arguments.

```
(%i1) cvect(x,y,z);
```

(%o1)

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

```
(%i1) rvect(x,y,z);
(%o1)
```

$$\begin{pmatrix} x & y & z \end{pmatrix}$$

Of course a list can be used as a kind of row vector, too. However, it is displayed with commas. A list can even be *transpose*d into a column vector, but not vice versa.

```
(%i1) [x,y,z];
(%o1)                          [x, y, z]
```

make_cvect (x, n) [function of *rs_algebra*]
make_rvect (x, n) [function of *rs_algebra*]

These functions create the respective vectors with the components being the elements $1, \ldots, n$ of an undeclared array named x.

21.1.2 Transpose a vector or list

transpose (v) [function]

Transposes a column vector to a row vector and vice versa. A list can be transposed to a column vector, too, but not vice versa. In general, any → matrix can be *transpose*d.

```
(%i1) v:cvect(x,y,z)$
(%i2) r:transpose(v);
```

(%o2)
$$\begin{pmatrix} x & y & z \end{pmatrix}$$

```
(%i3) transpose(r);
```

(%o3)
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

21.2 Matrices

Part V

Advanced Mathematical Computation

Chapter 22

Tensors

Chapter 23

Numerical Computation

Part VI

Maxima Programming

Chapter 24

Functions

Chapter 25

Program Flow

Part VII

User interfaces, Package libraries

Chapter 26

User interfaces

26.1 Internal interfaces

26.1.1 Command line Maxima

26.1.2 iMaxima

26.1.3 XMaxima

26.1.4 wxMaxima

26.1.5 GNUplot

26.2 External interfaces

26.2.1 Sage

26.2.2 Python, Jupyter, Java, etc.

Chapter 27

Package libraries

Part VIII

Developer’s environment

Chapter 28

Emacs-based Maxima Lisp developer's environment

28.1 Introduction

It has to be mentioned first that I owe large parts of the information provided in this chapter to the kind help of Michel Talon and Serge de Marre. Michel could answer me almost any question about how to set up the environment under Windows, although he himself does not have a Windows machine at all. Serge was maybe the first one who had figured out how to fully set it up under Windows. With videos on Youtube he showed me how it works. Both helped me for weeks with this non-trivial matter. Thanks a lot to both of you.

Hopefully, what took me months to find out and set up can be accomplished by the reader of the following instructions in a couple of days.

28.2 Overview of the software used

28.2.1 Operating systems and shells

We are going to use the Emacs-based Maxima developer's environment on an x64 machine primarily running under Windows 10. But we will also set up a complete Linux environment inside of *VirtualBox* under Windows and in addition use Linux-like environments directly under Windows, namely *MinGW* and *Cygwin*.

28.2.1.1 Windows

We are using 64 bit Windows 10 with the latest updates.

28.2.1.2 Linux in VirtualBox under Windows

28.2.1.3 Cygwin

28.2.1.4 MinGW

28.2.2 Maxima

28.2.2.1 Installer

The easiest way to install Maxima on Windows is to use the *Maxima installer* which can be downloaded from Sourceforge and which is available for every new release.

It comes with 64 bit *SBCL* and *Clisp*. Although it is preset to Clisp, it is recommended to set the standard Lisp to SBCL, because it is much faster and much more powerful. We will only use SBCL. Note that Clisp does not support threading and does not work properly under Emacs in combination with Slime, especially if it comes to the *slime-connect* facility, see below.

28.2.2.2 Build, Tarball, Repository

Using Maxima from an installer does have some drawbacks, though. Due to the fact that it was not compiled on the same system where it is used, Emacs cannot find the source code interactively within a running Maxima session under Slime. Finding the source code automatically for a given MaximaL function, however, is a very useful feature, as we will see later.

In order to allow for this feature to work, we will have to build Maxima ourselves. This can be done from a *Maxima tarball* which is provided for every new release and can be downloaded from Sourceforge. Or it can be done from a local copy of the *Maxima repository* which also resides on Sourceforge. In this case, the build process is a little bit longer, but we can use the latest snapshot available.

We build Maxima directly under Windows with the so-called *Lisp only build* process. Alternatively, Maxima can be built for Windows under Cygwin.

28.2.3 External program editor

28.2.3.1 Notepad++

If we are not really familiar with the Emacs editor yet, it is worthwhile to use *Notepad++* in addition. It is widely used and has parenthesis highlighting which is most important for programming in Lisp and very useful for Maxima, too. We will install a highlighting profile for MaximaL, too.

28.2.4 Lisp

28.2.4.1 SBCL: Steel Bank Common Lisp

SBCL already comes with the Maxima installer. This installation of SBCL can be used for Emacs' *inferior Lisp*, too. However, we can also install SBCL separately in addition, for instance if we want to use a different (newer) version or if we want to be independent of what happens to come with the consecutive installers.

28.2.4.2 Quicklisp

Quicklisp is a Lisp library and installation system. It runs under Lisp, so we will use it from SBCL. A good introduction and instruction how to use it can be found at https://www.quicklisp.org/beta/. We use quicklisp to install Slime.

28.2.4.3 Asdf/Uiop

Asdf (Another system definition facility) is a Lisp build system. See https://common-lisp.net/project/asdf/ for a description. *Uiop* is an extension of asdf which significantly enhances Common Lisp's functionality. For instance, it emulates file handling procedures for Windows.

28.2.5 Latex

We need to have a Latex installation on our system if we want to use the iMaxima interface, which runs under Emacs and gives Latex output.

28.2.5.1 MikTeX

MikTeX provides the Latex environment needed for iMaxima.

28.2.5.2 Ghostscript

Ghostscript is needed for iMaxima.

28.2.5.3 TeXstudio

TeXstudio is not needed for iMaxima, but it is a nice Latex editor which runs on top of MikTeX. This documentation was written with TeXstudio. The author wishes to thank the TeXstudio team for the kind help and support.

28.2.6 Emacs

Emacs is an integrated, Lisp based development environment and much more than that. [EmacsMan12]

28.2.6.1 Editor

It's not without any reason that one generally defines [EmacsTut]

Emacs = Escape, Meta, Alt, Control, Shift.

Although the Emacs editor and in particular its embeding in the overal IDE structure has very powerful features, it will take some time to get used to it. Before starting to work with Emacs, the *Emacs Tutorial*, an introductory tutorial about the editor and the basic environment should be studied in detail. It comes with the Emacs installation and is a plain text file of some 20 pages linked to the Emacs opening screen. The German version of Emacs comes with a German translation.

28.2.6.2 eLisp under Emacs

Emacs is written in *eLisp*, a dialect of Common Lisp. eLisp must be used to program the *.emacs* init file and any file to be loaded from it. But of course eLisp can also be used under Emacs for any other purpose. [eLispMan13]

28.2.6.3 "Inferior Lisp" under Emacs

Any other Common Lisp variant installed on the computer can be set up to be used as *inferior Lisp* under Emacs. This setup is done in the *.emacs* init-file. We use SBCL. Note that inferior Lisp is independent of the Lisp used by Maxima and of eLisp. All can be different.

28.2.6.4 Maxima under Emacs

There are various Maxima interfaces that work under Emacs. We use the Maxima console and iMaxima which is based on Latex.

28.2.6.4.1 Maxima command line interface

28.2.6.4.2 iMaxima interface

The *iMaxima* interface and how to set it up under Emacs is described in detail on Yasuaki Honda's iMaxima and iMath website. [iMaximaHP17]

28.2.6.5 Slime: Superior Interaction Mode for Emacs

Slime is an enhancement for Emacs. It contains much more elaborate debugging facilities and with *slime-connect*, see below, allows to create a parallel session of MaximaL and Maxima Lisp. [SlimeMan15]

28.3 Software installation and update

28.3.1 Maxima

28.3.1.1 Installer

Download the latest Maxima installer and install it in C:/Maxima/. Select SBCL as Lisp, either with the special routine that can be found in the program menu or from wxMaxima. Copy shortcuts for wxMaxima, console Maxima and XMaxima to the desktop.

28.3.1.2 Tarball

28.3.1.3 Repository

28.3.2 Notepad++

Install the latest version of Notepad++ 64 bit in the default directory C:/Program Files/Notepad++. We will soon need it. Make it the default program to open files of type .lisp, .mac, .txt, .sbclrc, .emacs, etc., whenever you read any of these file types later.

David Scherfgen from the Maxima team has written a *highlighting profile for Maxima*. It is available at http://www.roland-salz.de/html/maxima.html. To download it, rightclick on *Maxima_Notepad++.xml* and "Save as" *Maxima_Notepad++.xml*. (The file name on the server is Maxima_Notepad__.xml.) To install it in Notepad++, select Language/Select your language/Import. After restarting Notepad++, *Maxima* will appear in the menu and automatically be applied to *.mac* files.

28.3.3 Latex setup for iMaxima

28.3.3.1 Ghostscript

Install Ghostscript in the default directory C:/Program Files/gs.

28.3.3.2 MikTeX

Download the latest version from miktex.org. Execute the program as administrator (Rightclick). Install MikTeX in the default directory C:/Program Files/MikTeX 2.9. Load packages on the fly: "yes". If during installation the antivirus program complains, ignore it and continue the installation.

For maintenance always use the subdirectory Maintenance(Admin). After the installation, open the MikTex packet manager from the MikTeX 2.9/Maintenance(Admin) directory in the program menu. Install packages mhequ, breqn, mathtools, l3kernel, unicode-data. These files are needed for iMaxima. Immediately run Update from Maintenance(Admin), too, and install all available updates.

28.3.3.3 TeXstudio

Install TeXstudio in the default directory C:/Program Files (x86)/TeXstudio.

28.3.4 Emacs

Download the preconfigured installer version emacs-w64-25.3-O2-with-modules.7z from Sourceforge. This will set up Emacs properly with all the necessary dll files installed in the bin directory. Unzip it with 7zip (download and install this, if you don't have it yet). Unzip it to C:/ first. Then move the folder to C:/Program Files/Emacs/emacs-25.3-02-with-modules (this does not work directly, because it needs administrator approval which cannot be given during the unzip process).

Alternatively, a version with almost no dll files is emacs-25.3-x86_64.zip from the GNU mirror.

Numerous lib*.dll files can be added to the bin directory in order to bring Emacs to its full power (read the readme file that comes with Emacs). A large number of them and many other dependencies (.exe files) are included in emacs-25-x86_64-deps.zip, which also gives a complete Emacs installation.

In particular we need zlib1.dll and libpng16-16.dll, which gives support for png files, required for the iMaxima Latex interface to work.

Run bin/runemacs.exe to start Emacs and create a shortcut for it on the desktop.

28.3.5 SBCL

We install the latest version of SBCL in the default directory, that is in *C:/Program Files/Steel Bank Common Lisp/<version>*. The Windows path and the environment variable SBCL_HOME will be created automatically for our Windows user, if they don't exist yet. However, a Windows restart is necessary to activate them.
We should check that they are properly set. We should see in the path variable of our Windows user the path

```
C:\Program Files\Steel Bank Common Lisp\1.3.18\
```

In addition, we should see the environment variable SBCL_HOME with the value

```
C:\Program Files\Steel Bank Common Lisp\1.3.18\
```

If we alternately use the separately installed SBCL and the one from the Maxima installer later under Emacs, we do not need to change the Windows environment variables any more. Instead, the local copies of them can easily be adjusted in the .emacs init file, see section 28.4.2.2.

SBCL uses this environment variable to locate the folder where to search for its core file. If the folder does not match the SBCL version that was invoked with the .exe file, a severe error situation will arise and it will not be able to start SBCL.

To update the SBCL version, just execute the new SBCL installer. We do not need to deinstall the old one first. A subfolder with the new version will be created and the Windows environment variables adjusted automatically. We only need to adapt our personal setup and initialization files (e.g. *.emacs*, see below).

28.3.6 Quicklisp

Quicklisp will be installed via our Lisp system, which is SBCL. Download the file *quicklisp.lisp* from the Quicklisp homepage. Start SBCL from the Windows console by typing "SBCL" at the DOS prompt. Then, at the SBCL prompt, enter the following Lisp commands one by one. This will install Quicklisp in "C:/Quicklisp". Don't install it in the program files subdirectory, because Quicklisp does not like blanks in the filename. Then Quicklisp is loaded and some code is added to our .sbclrc init-file, see section 28.4.1.2, in order for Quicklisp to be loaded automatically whenever we start SBCL.

```
(load "C:/Users/<user>/Downloads/quicklisp.lisp")
(quicklisp-quickstart:install :path "C:/Quicklisp/")
(load "C:/Quicklisp/setup.lisp")
(ql:add-to-init-file)
```

If in the future we want to update our quicklisp installation, all we have to do is (from SBCL)

```
(ql:update-client)
(ql:update-dist "quicklisp")
```

Now that we have installed Quicklisp, we stay in SBCL to continue with installing Slime.

28.3.7 Slime

If we install Slime via Quicklisp (alternatively it can be installed from Melpa), it will be stored inside of C:/Quicklisp. Under SBCL, execute the following Lisp forms one by one. This will install Slime, including the Swank facilities. The last form will install slime-helper.el and add some code to our .emacs init file, see section 28.4.2.2, in order to load it and facilitate working with Slime. See *http://quickdocs.org/quicklisp-slime-helper/*.

```
(ql:update-client)
(ql:update-dist "quicklisp")
(ql:system-apropos "slime")
(ql:quickload "swank")
(ql:quickload "quicklisp-slime-helper")
```

We can check which version we have installed by looking at *C:/Quicklisp/dists/quicklisp/software*. We should find a folder here named *slime-v2.20*.

If we want to update an existing Slime installation, we follow exactly the same procedure as described above. A subfolder with the new version will be installed. It is not necessary to uninstall the old one. We only have to adapt the paths in our personal setup and initialization files (e.g. in *startswank.lisp*, see below).

28.3.8 Asdf/Uiop

Our Quicklisp installation comes with a Lisp source file *asdf.lisp* in the main folder. But Asdf/Uiop is already included in our SBCL installation, too. Here, in the contrib folder, we find the compiled files *asdf.fasl* and *uiop.fasl*. These are the files used by SBCL. It is important to have the latest possible version of Asdf/Uiop installed. To find out which version we have in our SBCL installation we can do from SBCL

```
(require 'asdf)
asdf::*asdf-version*
       "3.1.5"
```

The version of the asdf.lisp in our Quicklisp installation can be found in the source code itself. Just open the file with Notepad++. It turns out to be much older, in our case it is 2.29. We continue our investigations from SBCL:

```
(ql:update-client)
(ql:update-dist "quicklisp")
(ql:system-apropos "asdf")
```

tells us that the Quicklisp library has version 3.2.1 available. Finally, we take a look at the Asdf homepage and find out that the latest released version is 3.3.1. So we download the corresponding asdf.tar.gz and unpack it with 7zip (This goes in two steps: first we unzip the .tar.gz, then the resulting .tar). In addition, we download the *asdf.lisp* file from the Asdf archive. Oops, if we just click on the file, we get one very long string without any line breaks. But what we want can be done in the following way: rightclick on the file in the archive, select "save as" and set the file name to *asdf.lisp*. Then we open the file with Notepad++. Now we have the correct Windows line endings (CR/LF instead of Unix LF only)! What we want to do now is compile this file ourselves to create the asdf.fasl (which should include Uiop as well and) which we will insert into our SBCL/contrib folder to replace the existing version. We always save the existing versions, of course, by renaming them. Let's assume the asdf.lisp is in the downloads folder. Then we continue with SBCL

```
(compile-file "C:/Users/<user>/Downloads/asdf.lisp")
```

and wait patiently until the compilation process is finished. At the end, the asdf.fasl file should be in the download folder, too. We copy it into the folder *Steel Bank Common Lisp/1.3.18/contrib*. Then we leave SBCL by entering (quit), start it again from the Windows DOS prompt and continue with checking

```
(require 'asdf)
asdf::*asdf-version*
       "3.3.1"
```

It is obvious how we have to install a possible update later.

28.3.9 Linux and Linux-like environments

28.3.9.1 Linux in VirtualBox

28.3.9.2 MinGW

Install MinGW in C:/Program Files/MinGW.

28.3.9.3 Cygwin

Install Cygwin in C:/Program Files/cygwin64.

28.4 Setup

28.4.1 SBCL

28.4.1.1 Set start directory

The directory from which SBCL is started is called the SBCL start directory. The SBCL system variable *default-pathname-defaults* will be set to this directory and make it the so-called current directory. This will be the default path for file loads from within SBCL. Note that relative paths can be used on the basis of the current directory, and the standard file extension .lisp can be omitted. This also works under Maxima, if a Lisp load command is executed, e.g.

```
:lisp (load "System/Emacs/startswank")
```

However, if we load with the Maxima command, we can use relative paths, too, but we have to include the file extension *.lisp*

```
load ("System/Emacs/startswank.lisp")
```

28.4.1.2 Init file ".sbclrc"

A Lisp init file named ".sbclrc" can be created. It will be loaded and executed every time SBCL starts. Unfortunately, this file has to be placed in two different locations:

C:/Users/<user>
for wxMaxima, xMaxima, the Maxima console under Windows and the SBCL console (64 bit) under Windows.

C:/Users/<user>/AppData/Roaming
for all applications under Emacs and for the SBCL console (32 bit) under Windows.

In order to find out where the init-file is supposed to be for a specific SBCL application, use one of the following commands from within the particular application:

```
(sb-impl::userinit-pathname)
(funcall sb-ext:*userinit-pathname-function*)
```

If it is a Maxima application, simply preceed each Lisp command by ":lisp " at the Maxima prompt:

```
:lisp (sb-impl::userinit-pathname)
:lisp (funcall sb-ext:*userinit-pathname-function*)
```

The copies from both directories can be loaded into Notepad++ simultaneously under identical file names; as you will soon see, we will introduce a tiny difference between the two copies.

For our Maxima Lisp developer's environment this file should contain the following forms. The complete model file can be found in Annex A.

1. The following lines are inserted automatically by (ql:add-to-init-file). They will cause Quicklisp to be loaded on each start of SBCL.

```
#-quicklisp
(let ((quicklisp-init (merge-pathnames "C:/quicklisp/setup.lisp" (
    user-homedir-pathname))))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))
(format t "~%~a" "Quicklisp loaded.")
```

2. Set compiler option for maximum debug support:

```
(declaim (optimize (debug 3)))
(format t "~%~a" "(declaim␣(optimize␣(debug␣3)))␣set.")
```

3. Set external format to UTF-8:

```
(setf sb-impl::*default-external-format* :utf-8)
(format t "~%~a" "External␣format␣set␣to␣UTF-8.")
```

4. Display final messages:

```
(format t "~%~a" "Init-File␣C:/Users/<user>(/AppData/Roaming)/.sbclrc␣
    completed.")
(format t "~%~a~a" "Current␣directory␣(also␣from␣Maxima)␣is␣" *
    default-pathname-defaults*)
(format t "~%~a" "To␣change␣the␣current␣directory␣use␣(setq␣*
    default-pathnames-default*␣#P\"D:/Maxima/Builds/\").")
(format t "~%~a" "Relative␣paths␣can␣be␣used␣and␣standard␣file␣extension␣.
    lisp␣omitted,␣e.g.:␣(load␣\"subdir/subdir/filename\").")
(format t "~%~a" "␣")
```

In the first command adjust the Windows user and include or omit the parenthesized part, according to where the init file is placed. This way the init file will itself show where it is located for each SBCL application. The second line will show the current directory to the user on start of SBCL.

28.4.1.3 Starting sessions from the Windows console

We can start an SBCL session from the Windows console. Open the Windows shell (DOS prompt), cd to what you want to have as start directory and type SBCL.

28.4.2 Emacs

28.4.2.1 Set start directory

We can set the start directory for Emacs in the desktop shortcut (right click / properties / start directory). We use the path

```
D:\Programme\Lisp
```

This will be the default path for file loads from within Emacs (by typing C-x C-f in the mini buffer). This will also be the default for the start directory and therefore the current directory for SBCL, to which the variable *default-pathname-defaults* will be set. To show or change it from within SBCL use

```
*default-pathname-defaults*
(setf *default-pathname-defaults* #P"C:/maxima/repos/")
```

If we want a different SBCL start directory than the one for Emacs, we can cd to a different directory in start-sbcl.bat (see below) prior to invoking SBCL.

28.4.2.2 Init file ".emacs"

An eLisp init file named ".emacs" can be placed in C:/Users/<user>/AppData/Roaming. It will be loaded and executed every time Emacs starts. [EmacsMan12]

Under Windows it is sometimes difficult to copy/rename a file with a leading dot. However, it can always be done with "save as" from Notepad++.

For our Maxima Lisp developer's environment this file should contain the following lines. The complete model file can be found in Annex B.

1. *Load Quicklisp Slime Helper*:

```
(load "C:/quicklisp/slime-helper.el")
```

2. *Set inferior Lisp to SBCL*. We write a short Windows batch-file *start-sbcl.bat* which we place in D:/Programme/Lisp/System/SBCL and which we use to start SBCL. It allows us (by means of the Windows cd command) to preselect the start directory for SBCL. It will be SBCL's current directory. If we do not set the start directory in this file, the Emacs start directory will be used as default. The batch file is

```
"C:/Program Files/Steel Bank Common Lisp/1.3.18/sbcl.exe"
rem "C:/Maxima-5.41.0/bin/sbcl.exe"

rem Prior to calling SBCL we can set the SBCL start directory.
rem If we don't, the Emacs start directory will be the default.
rem Example:
rem D:
rem cd /Programme/Lisp
```

The above assumes that we use a separately installed SBCL. If instead we want to use the SBCL from the Maxima installer, we have to activate the out-commented path instead. In the init-file we write

```
(setq inferior-lisp-program "D:/Programme/Lisp/System/SBCL/start-sbcl.bat")
```

3. *Set up Maxima*. We need to load the system eLisp file *setup-imaxima-imath.el* which comes with Maxima. Best is to create a local copy in a fixed place on our computer, so we do not always have to adapt the path to the file if we use different Maxima installations. This file sets up Emacs to support Maxima and the Latex-based interface iMaxima. We do not need to customize this file. But before loading the file we set two system variables. *maxima-build-type* specifies whether we use Maxima from an installer or whether we have built Maxima from a tarball or a local copy of the repository. *maxima-build-dir* specifies the path to the root directory of the Maxima we want to use. If we do not specify these two system variables, the first Maxima installer found in "C:/" will be used. (Note that this is the oldest one installed.) So in the init-file we write [iMaximaHP17]

```
; *maxima-build-type* can be "repo-tarball" or "installer"
(defvar *maxima-build-type* "installer")

; *maxima-build-dir* contains the root directory of the build,
terminated by a slash.
(defvar *maxima-build-dir* "C:/Maxima/maxima-5.41.0/")
; (defvar *maxima-build-dir* "D:/Maxima/builds/lob-2017-04-04-lb/")

(load "D:/Programme/Lisp/System/Emacs/setup-imaxima-imath.el")
```

4. *Key reassignments for Slime*. In order to ease our work under Slime we change the keys for a number of its system functions. [SlimeMan15]

```
(eval-after-load 'slime
  '(progn
    (global-set-key (kbd "C-c_a") 'slime-eval-last-expression)
```

```
    (global-set-key (kbd "C-c c") 'slime-compile-defun)
    (global-set-key (kbd "C-c d") 'slime-eval-defun)
    (global-set-key (kbd "C-c e") 'slime-eval-last-expression-in-repl)
    (global-set-key (kbd "C-c f") 'slime-compile-file)
    (global-set-key (kbd "C-c g") 'slime-compile-and-load-file)
    (global-set-key (kbd "C-c i") 'slime-inspect)
    (global-set-key (kbd "C-c l") 'slime-load-file)
    (global-set-key (kbd "C-c m") 'slime-macroexpand-1)
    (global-set-key (kbd "C-c n") 'slime-macroexpand-all)
    (global-set-key (kbd "C-c p") 'slime-eval-print-last-expression)
    (global-set-key (kbd "C-c r") 'slime-compile-region)
    (global-set-key (kbd "C-c s") 'slime-eval-region)
  ))
```

5. *Customizing Emacs*. Emacs can be extensively customized. The changes made are stored automatically at the end of ".emacs". For example, the following code will be inserted when we do
M-x customize, Editor, Basic settings, Tab width, default 8 -> 2, Save. [EmacsMan12]

```
(custom-set-variables
;; custom-set-variables was added by Custom.
;; If you edit it by hand, you could mess it up, so be careful.
;; Your init file should contain only one such instance.
;; If there is more than one, they won't work right.
'(safe-local-variable-values (quote ((Base . 10) (Syntax . Common-Lisp) (
   Package . Maxima))))
'(tab-width 2))
(custom-set-faces
;; custom-set-faces was added by Custom.
;; If you edit it by hand, you could mess it up, so be careful.
;; Your init file should contain only one such instance.
;; If there is more than one, they won't work right.
)
```

28.4.2.3 Customization

In Emacs *Options/Set Default Font* set Courier New size to 12. Store this, so I don't have to set it on every start of Emacs.

28.4.2.4 Slime and Swank setup

A special setup is necessary for running Maxima or iMaxima under Emacs with Slime. We have to write a short Lisp program named *startswank.lisp* and place it in

```
D:/Programme/Lisp/System/Emacs
```

This is the code

```
(require 'asdf)
(pushnew "C:/quicklisp/dists/quicklisp/software/slime-v2.20/" asdf:*
   central-registry*)
(require :swank)
(swank:create-server :port 4005 :dont-close t)
```

28.4.2.5 Starting sessions under Emacs

To start a Lisp session under Emacs *without* Slime, type Alt-X and then in the minibuffer "run-lisp" or "inferior-lisp".

The error message "spawning child process" is a typical sign of SBCL searching in the wrong directory for its core file. Check that the path specified in start-sbcl.bat is correct. Check that the Windows environment variables of the current user (PATH and SBCL_HOME) are properly set.

To invoke the command history under SBCL, type Ctrl-<uparrow>.

To start a Lisp session under Emacs *with* Slime, type Alt-X and then in the minibuffer "slime". The screen will split and the Slime prompt will show up.

To start a console Maxima session under Emacs *without* Slime, type Alt-X and then in the minibuffer "maxima".

To start an iMaxima session under Emacs *without* Slime, type Alt-X and then in the minibuffer "imaxima".

To start a console Maxima or iMaxima session under Emacs *with* Slime, proceed as follows

1. Start Maxima or iMaxima under Emacs as described above.
2. At the Maxima prompt, enter

```
:load ("System/Emacs/startswank.lisp")
```

3. If the load succeeded, type Alt-X and then in the minibuffer "slime-connect".
4. At the message *Host: 127.0.0.1* hit return in the minibuffer.
5. At the message *Port: 4005* again hit return in the minibuffer.

Now the Emacs screen splits and a new window is opened with a prompt *Maxima>*. This is a Lisp session under Slime inside of the running Maxima session. All Maxima variables and functions can be addressed from it. This Emacs buffer can be used to debug or make modifications to the Maxima source code while Maxima is running. We can switch back and forth between the Maxima-Lisp and the Maxima-MaximaL windows by "Ctrl-x o" and enter input in both. The first time we switch back to the MaximaL window, there will be no Maxima prompt visible. Nevertheless, we can enter something followed by a semicolon, e.g. "a;" and the input prompt will reappear. Note that MaximaL variables have slightly different names under Lisp: they have to be preceeded by a "$" character, so e.g. the variable "a" has to be addressed as "$a" from the Lisp window. And as always in Lisp, commands are not terminated by a semicolon as they are in MaximaL.

It should be noted here that we won't have Slime's full functionality unless we use a Maxima built by ourselves. See section **??** for how this is done. Then, if the build succeeded, set up Emacs to use this build. Only this will allow Slime to interactively find the source code of Maxima functions while Maxima is running under Emacs.

28.4.3 Linux and Linux-like environments

28.4.3.1 Linux in VirtualBox

28.4.3.2 MinGW

28.4.3.3 Cygwin

28.5 Operation

28.5.1 SBCL

To invoke the command history, type C-<uparrow>.

28.5.2 Emacs

28.5.3 Slime

28.5.4 Linux and Linux-like environments

28.5.4.1 Linux in VirtualBox

28.5.4.2 MinGW

28.5.4.3 Cygwin

Chapter 29

Repository management: Git and GitHub

29.1 Introduction

29.1.1 Git and our local repository copy

The repository on Sourceforge works under the version control system *Git*. In order to create a local copy and to facilitate successive downloading of the latest snapshots, we need to install Git on our system, too. [ChProGit14] [GitRef17]

If we have access rights to the Sourceforge repository, we also use Git to send our commits.

A good introduction to Git is [ChProGit14]. All the details can be found in [GitRef17].

29.1.1.1 KDiff3

We will use *KDiff3* to help us resolve merge conflicts under Git.

29.1.2 GitHub and our public repository copy

We can work with a local repository only on our computer. If in addition we want to make public the work we have done or want to cooperate with others outside of Sourecforge, we can create a public copy of our local repository (which started from a copy of the Sourceforge repository). This can be done for instance on *GitHub*. We will explain how a copy (it is called a *mirror*) of the Maxima repository can be created on GitHub and how we can then synchronize it with our local repository.

Eventually we can also use our GitHub repository to communicate with the Maxima external packet manager system.

29.1.3 General intention: two ways to incorporate our changes

29.1.3.1 Modifying, rebasing, rebuilding

Let us briefly preview why we use Git and GitHub and what we want to do with them. We will create a local Maxima repository in order to be able to look at the Maxima source code files and to make changes to them. But we will not only make our own changes, we will also continuously update our local mirror by downloading all changes done to the Maxima repository at Sourceforge. It is only with the help

of Git that we will be able to *merge* (or, as we will se, *rebase*) our changes with the ones being parallelly done at Sourceforge. This will allow us to modify the Maxima code according to our needs without losing the bug fixes, modifications and enhancements done by the Maxima team at the same time.

On GitHub we will create a mirror from Sourceforge, too, but then we will not update it directly from Sourceforge, but instead from our local repository. So it will be a mirror of our local repository. It will make public the changes that we have done here and which are always based on the latest updated done at Sourceforge.

The changes we do to our repository will be incorporated in our Maxima builds.

29.1.3.2 Modifying and loading individual files

There is another and much simpler way to incorporate changes into the Maxima we use. Most people will want to try out this way first and see whether it might be sufficient for their needs. At the start or at any later point within a Maxima session the user can load indivudual Source files, written either in MaximaL or Lisp. He can even load individual functions, and furthermore he can simply use them to overwrite existing functions of the same name. So any Maxima system function can easily be changed by just reloading a modified version of it. It is not necessary to reload the whole system file which contains it, and it is not necessary for the file that contains the modified function to have the same name as the original system file. Only the name of the function has to be identical. Depending on the Lisp used, Maxima will give a warning that an existing function is being redefined, but it will not decline to do so.

This function substitution or function adding can be automized by doing it from the maxima-init files at Maxima startup. And even after a new Maxima release, the procedure does not have to be changed. So up to a certain point, we can apply our changes on top of the latest Maxima release.

This method, however, as nice as it might be in the beginning, will be more and more complicated with a growing number of modifications we make and files that are affected. Furthermore, we cannot easily incorporate modifications that the Maxima team might make at precisely the same files that we have changed ourselves. So at a certain point the user will have no other chance than to use Git to manage his local repository and merge his modifications with the ones from Sourceforge.

29.2 Installation and Setup

29.2.1 Git

29.2.1.1 Installing Git

Download the latest Windows installer from *git-scm.com*. Install it as administrator in the default directory C:/Program Files (x86)/Git with all the default settings. In particular, we want to be sure to use the recommended option to check out files in Windows style (with CR/LF ending) and commit files in Unix style (with LF ending). Also, as the default says, install the TTY console.

Create shortcuts on the desktop from the program menu. We will primarily use the CMD interface which resembles the Windows console. In order to set our start

directory to D:/Maxima/Repos do the following. Rightclick on the CMD interface shortcut. Select properties. Change "Execute in" to the above path. In "Destination" delete the option -cd-to-home.

RS only: When started, rightclick on the margin of the window and in properties set font size to 20.

29.2.1.2 Installing KDiff3

29.2.1.3 Configuring Git

Git allows configuration at various levels: system, user, project. Configuration files are therefore created in various locations. In C:/Users/<user>/ we place the file *.gitconfig* given in Annex C.

Most important is to adjust your name and email. We have also specified the text editor to be used for commit messages and the merge tool. The autocrlf command allows for the correct transformation of line endings from Unix to Windows and vice versa. The whitespace command causes git-diff to ignore "exponentialize-M" characters. In addition we have defined some shortcuts for the most frequent commands (st, ch, br, logol). With

```
git config --global --edit
```

from the Git prompt (Note: two dashes before each option here) Notepad++ should open and display the file *.gitconfig*.

29.2.2 GitHub

29.2.2.1 Creating a GitHub account

On GitHub, presently (Dec. 2017), it is free of charge to open a personal account and create public repositories within it. *Public* here means that we cannot hide the source code of our repositories. Everyone else can see it and clone it. This is independent of whether we use the repository alone or together with others. In the latter case we can give explicit permission to individual other GitHub users to have write access to our repository.

So the first step is to sign up in GitHub. We create a personal account by assigning a user name and password and providing an email address for communication. All other settings we can do later. It is always possible to change any settings at any time. Even the user name can be changed, but it is not advisable to do so, because this change can never be done to 100 percent. It is easily possible to delete the account, too.

On the next screen we select the option *Unlimited public repositories for free*. On the following screen, let us *Skip this step*. Next, instead of *Read the guide* or *Start a project*, we move directly to our profile and use it as a starting point for creating our Maxima repository. So in the upper right corner we click on the little triangle to the right of the avatar symbol and select *Your profile*. We create a browser favorite which leads us to this page, because everything else will start from here. Just to give you a glimpse at how we will continue: click on the little triangle to the right of the "+" sign in the upper right corner and you will see the options *New repository* and *Import repository* which we will soon make use of.

We will use only plain command line Git to communicate with our GitHub repositories. There are special programs from GitHub to do so, too, e.g. the GitHub desktop, but in our opinion it is a waste of time and effort to learn them. Git is the underlying software in any case and in order to have full control of what we want to do, we better stay at this ground level. Every other program on top of it will hide information from us that at one point or another we will urgently need in order to make Git do exactly what we want. This can be complicated at times, we need to learn a number of Git commands, but there is no way around it.

29.3 Cloning the Maxima repository

29.3.1 Creating a mirror on the local computer

This process is called *cloning*. Let's assume we are in our directory D:/Maxima/Repos and want to place the copy of the repository in a subfolder named *Maxima*. We look at the Maxima domain at Sourceforge https://sourceforge.net/p/maxima/code/ci/master/tree/ to find out what the download URL of the git repository is. We select the *https* access rather than the *git://* access. Then we enter at our Git prompt

```
git clone https://git.code.sf.net/p/maxima/code rMaxima
```

where *rMaxima* ist our destination subfolder. And now we wait patiently until the latest snapshot (meaning: the actual status) of the Maxima repository from Sourceforge has been completely copied.

29.3.2 Creating a mirror on GitHub

We will clone the Maxima repository from Sourceforge to our account on GitHub in a similar way as we cloned it to our local computer. But once we have done that, we will update our GitHub repository only via our local repository. This includes all changes made to the Maxima repository on Sourceforge. We will download them periodically to the local repository and upload them from our local repository to the GitHub repository. So in effect, our GitHub repository is only going to be a direct mirror of Sourceforge in the beginning. After this initialization, the GitHub repository will rather be a mirror of the repository on our local computer. It will reflect the work that we have done on our local repository and at the same time incorporate the changes done at Sourceforge.

We click on the little triangle to the right of the "+" sign in the upper right corner of our GitHub user profile, then select *Import repository*. We have to specify the URL of the source repository at Sourceforge (called the *old repository* on the GitHub screen) which is still

```
https://git.code.sf.net/p/maxima/code
```

and then a name for the mirror on our GitHub account, let's say "rMaxima", too. Then we click on *Begin import*. The import from Sourceforge to GitHub can take a couple of minutes.

Once we have receivd the email notification about our mirror having been successfully installed on GitHub, we go to our account profile again and *Customize our pinned repositories* by selecting our new repository *Maxima*. Now it will be visible on our account profile and we can always find it and move to it easily. On selecting

our new repository, a short description of it can be given which will be displayed on the acount profile together with its name.

29.4 Updating our repository

29.4.1 Setting up the synchronization

Soon there will be new commits submitted at the Sourceforge repository and we will want to download them. Together with the changes we make ourselves we will want to push them to our GitHub mirror. So what we want to do now is prepare for updating our local repository from Sourceforge and our GitHub repository from our local repository.

29.4.2 Pulling to the local computer from Sourceforge

Let's first look into our local repository. We start *Git CMD* and *cd* to *D:/Maxima/Repos /rMaxima*. Then we enter

```
git remote show origin
```

In Git, *origin* is the shortname of our source repository, which is Maxima at Sourceforge. The above command gives us an overview of what exactly we've just cloned from there.

The most interesting one of the remote branches we see is *master*. It is the official, the decisive, the relevant branch with the actual status of the Maxima repository at Sourceforge. Our local branch *master* corresponds to it. Our local *master* shall always be a true copy of the present status at Sourceforge. So we never commit changes to it, we only use it for pulling from Sourceforge and for pushing the changes which come from Sourceforge to our *Maxima* repository at GitHub. Instead, we do our work on other branches which we create from our local *master*.

Updating our local *master* branch from Sourceforge is done by

```
git ch master
git pull
```

Note that we use the shortnames defined in *.gitconfig*, see. Annex C. With the option *pull –all* all tracked branches will be pulled from origin.

New branches on Sourceforge will be shown in the list by the *remote show origin* command, marked as *new*. On the next *git pull* they will automatically be tracked. Branches deleted on Sourceforge will be marked in the list as *stale*. They will not be deleted automatically by *pull*, instead we have to remove them manually with

```
git ch master
git remote prune origin
```

29.4.3 Pushing to the public repository at GitHub

First we create a shortname *github* for our *rMaxima* repository at GitHub by associating it with the URL of our GitHub repository:

```
git remote add github https://github.com/<username>/rMaxima.git
```

Then we take a look at our GitHub repository by entering

```
git remote show github
```

Just as our local *master* shall always be a true copy of *master* at Sourceforge, our *master* at GitHub shall always be a true copy of our local *master*. Updating *master* on GitHub from our local *master* is done by

```
git ch master
git push github
```

With the option *push github –all*, all local branches configured for push (see list *remote show github*) will be pushed to GitHub. In order to configure a branch for push to GitHub or to forward a new branch from Sourceforge to GitHub, we have to track the branch first in our local repository, done with the checkout command, and then push it to GitHub

```
git ch <name of new branch>
git push github <name of new branch>
```

In the *push* command the name of the branch is not necessary, if we are on this branch already. If we want to delete a branch from GitHub, for instance because it has been deleted from Sourceforge, we do

```
git push github –d <name of branch to be deleted>
```

To update the repository completely with all branches from Sourceforge after a year or more, it is easiest to delete the GitHub repository, clone it newly and push all my own branches again.

29.5 Working with the Repository

29.5.1 Preamble

Git is a very intelligent program. It is most important for the user to know that under Git what we see in the Windows directories is not what is physically there, but what Git virtually shows us. The contents of what we see of the repository in Windows explorer depends on what *Git branch* we are currently in. Branches do not correspond to Windows explorer directories! What branch we are in, can only be seen in Git itself, not in the explorer. Changes to files in one branch, even addition and deletion of files, will not be visible *in the same Windows folder* any more, if we switch to another branch where these changes have not been incorporated. Be sure to have understood that very clearly before working with Git. This will prevent you from some severe headaches (you will probably get others with Git at some point or another anyways).

29.5.2 Basic operation

We get a list of all our local branches with

```
git br
```

To see which branch we are presently on, type

```
git st
```

We can create a new branch from an existing one by doing

```
git ch <name of the branch we want to branch from>
git ch —b <name of the new branch>
```

29.5.3 Committing, merging and rebasing our changes

Chapter 30

Building Maxima under Windows

30.1 Introduction

In this section we show how Maxima can be built on the local computer under the Windows operating system. Maxima is primarily designed for Unix-based operating systems, especially Linux. Sophisticated system definition and build tools are employed to automate as much as possible the complicated build process. Since these tools (in particular *GNU autotools*) are not available under Windows, there are two ways how Maxima can be built here. The first one makes use of the Unix-based tools and thus needs an environment which supports them. Such an environment is Cygwin, a Unix-like shell running under Windows and in which Windows executables can be produced. The second one does not use the Unix-based build tools at all, but an (almost) purely Lisp-based method. It can be accomplished under the plain Windows command line shell. All we need is a Lisp system installed. Since this is the simpler and easier method, we demonstrate it first. Note however, that not all Maxima user interfaces and features are supported with this build.

30.2 Lisp-only build

30.2.1 Limitations of the official and enhanced version

The official Lisp-only build process is described in the text file *INSTALL.lisp* which can be found in the main folder of any release tarball or the repository. This procedure has the following limitations:
- XMaxima cannot be built.
- wxMaxima is not included.
- GNUplot is not included.
- the documentation cannot be built.

We have made some enhancements to this procedure. In the following we give a complete description of the revised procedure. Now the documentation can be built with the exception of the PDF version.

We can build Maxima from a release source code tarball or from the latest repository snapshot. The following recipe comprises both alternatives.

30.2.2 Recipe

1. Install the Windows installer of the latest release in *C:/Maxima/maxima-5.41.0*. Download the source code file *maxima-5.41.0.tar.gz* of the latest Maxima release from https://sourceforge.net/projects/maxima/files/Maxima-source/5.41.0-source/ and extract the tarball with 7zip in the folder *D:/Maxima/Tarballs/*.

2. Create the directory of the new build and name it appropriately, e.g. *D:/Maxima/Builds/<lob-2017-12-09-lb>*, now called the *build directory*.

3. Depending on what to build from,
3a. either copy the extracted source code from the release tarball into the build directory; or
3b. select the branch of the local repository *D:Maxima/Repos/rMaxima* from which to build. Pull master and rebase this branch on master first in order to have our changes rebased on the latest Git snapshot from Sourceforge. Copy the selected branch into the build directory.
3c. In both cases, copy the PDF version of the documentation, the file *maxima.pdf*, from the subfolder *share/doc* of the Windows installer into the subfolder *doc/info* of the build directory.

4. The tarball contains the complete documentation of the latest release with the exception of the PDF version. In case the documentation shall not be built (also if we build from a repository snapshot), it can be simply be copied from the tarball into the build directory:
4a. For the online help system: From *doc/info* take *maxima-index.lisp* and all files **.info** and copy them into *doc/info* of the build directory.
4b. For the html version: From *doc/info* take all files **.html* and copy them into *doc/info* of the build directory.

5. Now we use Lisp. The following steps can be executed either using SBCL form a Windows command line shell or under Emacs/Slime (Note, however, that dumping can be done only from the Windows command line!): 5a. Open a Windows command shell and cd to the top-level of the build directory (i.e., the directory which contains src/, tests/, share/, and other directories). Then launch SBCL. Alternatively, 5b.

30.3 Building Maxima with Cygwin

Part IX

Maxima’s file structure, build system

Chapter 31

Maxima’s file structure: repository, tarball, installer

Chapter 32

Maxima’s build system

Part X

Maxima development

Chapter 33

MaximaL development

33.1 Debugging MaximaL

33.1.1 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

Chapter 34

Lisp Development

34.1 MaximaL and Lisp interaction

34.1.1 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.

34.1.2 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.

34.1.3 Executing Lisp code from within MaximaL

34.1.3.1 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))
```

Part XI

Lisp program structure (model), control and data flow

Chapter 35

Lisp program structure

35.1 Supported Lisps

Part XII

Appendices

Appendix A

SBCL init file ".sbclrc"

The following is a model of the complete SBCL init file ".sclrc" to be placed both in C:/Users/<user> and C/:Users/<user>/AppData/Roaming. See section 28.4.1.2 for explanations.

```
; initialize Quicklisp
#-quicklisp
(let ((quicklisp-init (merge-pathnames "C:/quicklisp/setup.lisp" (
    user-homedir-pathname))))
(when (probe-file quicklisp-init)
(load quicklisp-init)))
(format t "~%~a" "Quicklisp loaded.")

; Set compiler option for maximum debug support
(declaim (optimize (debug 3)))
(format t "~%~a" "(declaim (optimize (debug 3))) set.")

; Set external format to UTF-8
(setf sb-impl::*default-external-format* :utf-8)
(format t "~%~a" "External format set to UTF-8.")

; display final messages
(format t "~%~a" "Init-File C:/Users/<user>(/AppData/Roaming)/.sbclrc
    completed.")
(format t "~%~a~a" "Current directory (also from Maxima) is " *
    default-pathname-defaults*)
(format t "~%~a" "To change the current directory use (setq *
    default-pathnames-default* #P\"D:/Maxima/Builds/\").")
(format t "~%~a" "Relative paths can be used and standard file extension .
    lisp omitted, e.g.: (load \"subdir/subdir/filename\").")
(format t "~%~a" " ")
```

Appendix B

Emacs init file ".emacs"

The following is a model of the complete Emacs init file ".emacs" to be places in C:/Users/<user>/AppData/Roaming. See section 28.4.2.2 for explanations.

```
; load Quicklisp Slime helper
(load "C:/Quicklisp/slime-helper.el")

; set inferior Lisp to SBCL
(setq inferior-lisp-program "C:Users/<user>/start-sbcl.bat")

; Manually set temporary copy of Windows environment variable SBCL_HOME
; This is here only for debugging. Normally we don't have to do this. The
   Windows environment variable is set to our separately installed inferior
    Lisp, and Maxima will set the temporary copy of the variable itself.
; (setenv "SBCL_HOME" "C:/maxima-5.41.0/bin")
; (setenv "SBCL_HOME" "C:/Program Files/Steel Bank Common Lisp/1.3.18/")

; set up Maxima
; *maxima-build-type* can be "repo-tarball" or "installer"
(defvar *maxima-build-type* "installer")
; *maxima-build-dir* contains the root directory of the build, terminated
   by a slash.
(defvar *maxima-build-dir* "C:/Maxima/maxima-5.41.0/")
; (defvar *maxima-build-dir* "D:/Maxima/builds/lob-2017-04-04-lb/")
(load "D:/Programme/Maxima/System/Emacs and Slime setup for Maxima/
   setup-imaxima-imath.el")

; Key reassignments for Slime
(eval-after-load 'slime
  '(progn
    (global-set-key (kbd "C-c a") 'slime-eval-last-expression)
    (global-set-key (kbd "C-c c") 'slime-compile-defun)
    (global-set-key (kbd "C-c d") 'slime-eval-defun)
    (global-set-key (kbd "C-c e") 'slime-eval-last-expression-in-repl)
    (global-set-key (kbd "C-c f") 'slime-compile-file)
    (global-set-key (kbd "C-c g") 'slime-compile-and-load-file)
    (global-set-key (kbd "C-c i") 'slime-inspect)
    (global-set-key (kbd "C-c l") 'slime-load-file)
    (global-set-key (kbd "C-c m") 'slime-macroexpand-1)
    (global-set-key (kbd "C-c n") 'slime-macroexpand-all)
    (global-set-key (kbd "C-c p") 'slime-eval-print-last-expression)
    (global-set-key (kbd "C-c r") 'slime-compile-region)
```

```
    (global-set-key (kbd "C-c␣s") 'slime-eval-region)
  ))

; The following is placed here automatically by
; M-x customize, Editor, Basic settings, Tab width, default 8 -> 2, Save
(custom-set-variables
;; custom-set-variables was added by Custom.
;; If you edit it by hand, you could mess it up, so be careful.
;; Your init file should contain only one such instance.
;; If there is more than one, they won't work right.
'(safe-local-variable-values (quote ((Base . 10) (Syntax . Common-Lisp) (
   Package . Maxima))))
'(tab-width 2))
(custom-set-faces
;; custom-set-faces was added by Custom.
;; If you edit it by hand, you could mess it up, so be careful.
;; Your init file should contain only one such instance.
;; If there is more than one, they won't work right.
)
```

This is the file *start-sbcl.bat*:

```
"C:/Program Files/Steel Bank Common Lisp/1.3.18/sbcl.exe"
rem "C:/Maxima-5.41.0/bin/sbcl.exe"

rem Prior to calling SBCL we can set the SBCL start directory.
rem If we don't, the Emacs start directory will be the default.
rem Example:
rem D:
rem cd /Programme/Lisp
```

Appendix C

Git configuration file ".gitconfig"

The following is a model of the complete Git configuration file ".gitconfig" to be placed in C:/Users/<user>. See section 29.2.1.3 for explanations.

```
[filter "lfs"]
clean = git-lfs clean -- %f
smudge = git-lfs smudge -- %f
required = true
[user]
name = Roland Salz
[user]
email = maxima@roland-salz.de
[core]
editor = 'c:/Program Files/Notepad++/Notepad++.exe' -multiInst -nosession
autocrlf = true
whitespace = cr-at-eol
[alias]
st = 'status'
ch = 'checkout'
br = 'branch'
logol = log --pretty=format:'%h %cn %cd %s'
[merge]
tool = kdiff3
[mergetool "kdiff3"]
path = c:/Program Files/kdiff3/kdiff3.exe
[diff]
tool = kdiff3
guitool = kdiff3
[difftool "kdiff3"]
path = c:/Program Files/kdiff3/kdiff3.exe
```

Bibliography

[ChProGit14] Scott Chacon and Ben Straub. *Pro Git*. 2. ed. 2014. URL: https://github.com/progit/progit2/releases/download/2.1.15/progit.pdf.

[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).

[EmacsMan12] GNU Emacs. *GNU Emacs Manual 2.14 engl.* 2012. URL: https://www.gnu.org/software/emacs/manual/pdf/emacs.pdf.

[eLispMan13] GNU Emacs. *GNU Emacs Lisp Reference Manual 2.14 engl.* 2013. URL: https://www.gnu.org/software/emacs/manual/pdf/elisp.pdf.

[EmacsTut] GNU Emacs. *Einführung in Emacs*.

[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.

[GitRef17] Git. *Git Online Reference*. [Online; Stand 28. November 2017]. 2017. URL: https://git-scm.com/docs.

[iMaximaHP17] Yasuaki Honda. *iMaxima and iMath Homepage*. [Online; Stand 18. November 2017]. 2017. URL: https://sites.google.com/site/imaximaimath/.

[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.

[MaximManE17] 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.

[SlimeMan15] Slime. *Slime Manual 2.14 engl.* 2015. URL: https://common-lisp.net/project/slime/doc/slime.pdf.

[StewenMT13] Roland Stewen. *Standardaufgaben der Sekundarstufe I und II mit Maxima lösen*. 2013. URL: http://www.rvk-hagen.de/~stewen/maxima_in_beispielen.pdf.

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

Index