Maxima Workbook

Roland Salz

April 16, 2018
Vers. 0.2.4

This project is work in progress. It is in the beginning phase.
Comments and suggestions for improvement are welcome.

Roland Salz
Braunsberger Str. 26
D-44809 Bochum
mail@roland-salz.de

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 (GPL) software, maintained by an energetic group of volunteers called the *Maxima team*. The author wishes to thank its kind 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.

Contents

Part I

Historical Evolution, Documentation

Chapter 1

Historical evolution

1.1 Overview

The computer algebra system Maxima was developed, originally under the name Macsyma, from 1968 until 1982 at Massachusetts Institute of Technology (MIT) as part of project MAC. Together with Reduce it belongs to the first comprehensive CAS systems and was based on the most modern computational algorithms of the time. Macsyma was written in MacLisp, a pre-Common Lisp which had also been developed by MIT.

In 1982 the project was split. An exclusive 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 (for a considerable fee) to academic and government institutions. This version is known as DOE Macsyma. When Symbolics got into financial problems, enhancement and support for the commercial Macsyma license was separated to a company named Macsyma, Inc., which continued development until 1999. Financial failure of this company has left the enhanced source code unavailable ever since.

From 1982 until his death in 2001, William Schelter, professor of mathematics at the University of Texas, maintained a copy of DOE Macsyma. He ported Macsyma from MacLisp to Common Lisp. In 1999 he requested and received permission from the Department of Energy to publish the source code on the Internet 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 Macsyma trademark, the open source project was named Maxima. Since then, Maxima has been continuously improved.

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 [MosesMPH12]

the area of artificial intelligence and symbolic computation, and after having completed their respective theses in 1967 (Joel Moses' thesis is 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 capability 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. He was a graduate student in the Division of Engineering and Applied Physics of Harvard, (1966-71) but found an opportunity to pursue research down the road in Cambridge, at MIT. He received his Ph.D. from Harvard, but de facto he had contributed to the Macsyma project. His thesis from 1971 on *Algebraic Simplification* describes various components of Macsyma which he had implemented. From 1971-1974 he taught at MIT (in Mathematics), before he left for University of California at Berkeley, in Computer Science. The Macsyma 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 [wikMacsy17]

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).

1.2.3 The users' community

A nationwide Macsyma users community, to which belonged, among others, DOE, NASA and the US Navy, but also companies like Eastman Kodak, 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, meanwhile professor of Computer Science 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. This was possible due to the Bayh-Dole Act having recently passed the Congress. It allowed universities under certain conditions to sell licenses for their developments funded by the government to companies. 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 an exclusive 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 resources 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. The last version of Macsyma is still for sale on the INTERNET (as of 2017) for Microsoft's Windows XP operating system. Later versions of Windows, however, are not supported. Macsyma for Linux is not available at all any more.

Nevertheless, mainly due to the work of a number of former MIT developers, like Jeff and Ellen Golden or Bill Gosper, who had switched to work for Symbolics, the computational capabilities of Macsyma were significantly enhanced during this period of commercial development from 1982-1999. These enhancements are not included in present Maxima, which is based on another branch of Macsyma development, split off in 1982 under the name of DOE Macsyma. If these enhancements from the Symbolics era were ever made available to Maxima in the future and could be integrated into the present system, maybe at least in parts, this could certainly result in a major improvement for the open source project. [GosperHP17]

1.5 Academic and US government licensing

1.5.1 Berkeley Macsyma and DOE Macsyma

Richard Fateman had gone to Berkeley already in 1974. He continued to work on computers at MIT via ARPANET, predecessor of the Internet. He was interested in making Macsyma run on computers with larger virtual memory than the existing PDP-10, and when the VAX-11/780 was available he fought for Berkeley to get

one. This achieved, his idea was to write a Lisp compiler compatible with MacLisp and which would run on Berkley UNIX. *Franz Lisp* was created, the name having been invented spontaneously for its resemblance with *Franz Liszt*; it was still a pre-Common Lisp. With these resources rapidly developed, Fateman had in mind to share usage of Macsyma with other universities around. But MIT resisted these efforts.

UC Berkeley finally reached an agreement with MIT to be allowed to distribute copies of Macsyma running on VAX UNIX. But this agreement could be recalled by MIT when a commercial license was to be sold by them, which eventually was done to Symbolics. About 50 copies of Macsyma were running on VAX systems at the time. But Fateman wanted to go on and ported Franz Lisp to Motorola 68000, so that Macsyma could run on prototypes of workstations by Sun Microsystems.

Around 1981, when the discussion about commercial licensing of Macsyma became more and more intense at MIT, Richard Fateman and a number of other Macsyma users asked the United States Department of Energy (DOE), one of the main and therefore influential sponsors of the Macsyma project, for help to make MIT allow the software to become available for free to everyone. What he had in mind was a kind of Berkeley BSD license, which does not, like a GNU general public license, prevent commercial exploitation of the software. On the contrary, such a license, which can be considered really *free*, would not only allow everyone to use and enhance the software, but also to market their product. This license, for instance, allowed Berkeley students to launch startups with software developed at their school.

Finally, in 1982, at the same time when the commercial license was sold to Symbolics, DOE obtained a copy of the source code from MIT to be kept in their library. It was not made available to the public, its use remained restricted to academic and US government institutions. For a considerable fee these institutions could obtain the source code for there own development and use, but without the right to release it to others. This version of Macsyma is known as *DOE Macsyma*.

The version of the Macsyma source code given to DOE had been recently ported from MacLisp to NIL, *New Implementation of Lisp*, another MIT development. Unfortunately, this porting was not really complete, MIT never finalized it, and the DOE version was substantially broken. All academic and government users fought with these defects. Some revisions were exchanged or even passed back to DOE, but basically everyone was left alone with having to find and fix the bugs.

1.5.2 William Shelter at the University of Texas

From 1982 until his sudden death in 2001 during a journey in Russia, William Schelter, professor of mathematics at the University of Texas in Austin, maintained one of these copies of DOE Macsyma. He ported Macsyma from MacLisp to Common Lisp, the Lisp standard which had been established in the meantime. Schelter, who was a very prolific programmer and a fine person, added major enhancements to DOE Macsyma.

1.6 GNU public licensing

In 1999, in the same year when development of commercial Macsyma terminated, DOE was about to close the NESC (National Energy Software Center), the library which distributed the DOE Macsyma source code. Before it was closed, William Schelter asked if he could distribute DOE Macsyma under GPL. No one else seemed to care for this software anymore and neither did DOE. Schelter 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 Macsyma trademark, the open source project was named *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 and Macsyma Inc. between 1982 and 1999.

1.6.1 Maxima, the open source project since 2001

Judging from the number of downloads, Maxima today has about 150.000 users worldwide. 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, on Linux, Windows or Macintosh.

An enthusiastic group of volunteers, called the *Maxima team* and led by Robert Dodier from Portland, Oregon, today maintains Maxima. Among the Lisp developers are Raymond Toy, Barton Willis (University of Nebraska, Kearney), Kris Katterjohn, 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 J. 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 both developers and users on Maxima's principal communication channel, the mailing list at Sourceforge.

1.7 Further reading

A review of Macsyma is a long article by Richard Fateman in *IEEE Transactions on Knowledge and Data Engineering* from 1989, available as free PDF. Fateman writes in the abstract: [FatemanRM89]

"We review the successes and failures of the Macsyma algebraic manipulation system from the point of view of one of the original contributors. We provide a retrospective examination of some of the controversial ideas that worked, and some that did not. We consider input/output, language semantics, data types, pattern matching, knowledge-adjunction, mathematical semantics, the user community, and software engineering. We also comment on the porting of this system to a variety of computing systems, and possible future directions for algebraic manipulation system-building."

What better inspiration for the following chapters can we wish for?

Chapter 2

Documentation

2.1 Introduction

It is our feeling that Maxima's documentation can be improved. Both as a user and even more as a developer one would like to have much more information at hand than what the official Maxima manual, the other internal documentation that comes with the Maxima installation, and the comments in Maxima's program code provide.

Especially in the beginning, the user will often not understand the information in the manual easily. It contains a concise description of the Maxima language, but primarily as a reference directed to the very experienced user. It takes years to really understand and efficiently use a CAS. The beginner will need further explanation of all the implications of the condensed information from the official manual, more examples and a better understanding of the overall structure of the complex Maxima language (it comprises of thousands (!) of different functions and option variables).

Numerous external tutorials, some of them generally covering Maxima's mathematical capabilities, others restricted to applications of Maxima in the most important fields, such as Physics, Engineering or Economics, have been written and are of immense help for the beginner. Some of them are so comprehensive that they come close to a reference manual. Our intention is not to write a tutorial, but a manual, directed to a broader audience than the existing one, ranging from the still unexperienced user to the Lisp developer.

A considerable number of user interfaces has been developed, and the user will be quite lost about judging which one will best fit his needs.

Users and developers wanting to build Maxima themselves will find little documentation of the build process, especially if they want to work under Windows.

Even for an experienced Lisp developer the structure of Maxima's huge amount of program code will not be easy to understand. There is almost no documentation besides the program code, and this code itself is poorly documented, having been revised by many hands over many years. There are inconsistencies, forgotten sections, relics of ancient Lisp dialects and lots of bugs. The complicated process of Maxima's dynamic scoping and the information flow within the running system are

hard to keep track of. Very few of Maxima's few Lisp developers really overlook it in total.

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

2.2 Official documentation

2.2.1 Manuals

2.2.1.1 English current version

The official Maxima manual in English is updated with each new Maxima release. It is included in HTML format, as PDF and as the online help in each Maxima installer or tarball. It can also be built when building Maxima from source code. Our Maxima Workbook is primarily based on this documentation. [MaxiManE17]

2.2.1.2 German version from 2011

A German version of the manual exists. 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 introductins, 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 External documentation

2.3.1 Manuals

2.3.1.1 Paulo Ney de Souza: The Maxima Book, 2004

Paulo Ney de Souza has written, together with Richard Fateman, Joel Moses and Cliff Yapp, one of the most comprehensive Maxima manuals. Unfortunately, the project has not been finalized and is no longer updated, the last version dating from 2004. In particular, the Maxima Book contains detailed information about different user interfaces, including installation instructions. [SouzaMaxB04]

2.3.2 Tutorials

2.3.2.1 Zachary Hannan: wxMaxima for Calculus I + II, 2015

This tutorial by Zachary Hannan from Solano Community College, Vallejo, Ca., although having wxMaxima in its title, really covers the CAS Maxima, viewed through the wxMaxima user interface. Two volumes of about 160 pages each cover basic methods of using Maxima to solve problems from Calculus. Volumes on other fields of application are to follow. [HanMC1-15] [HanMC2-15]

2.3.2.2 Wilhelm Haager: Computeralgebra mit Maxima: Grundlagen der Anwendung und Programmierung, 2014

Wilhelm Haager's major work on the CAS Maxima was published 2014 in German at Hanser Verlag. This tutorial has over 300 pages and comes close to a comprehensive manual of the Maxima language. For example, rule-based programming is coverd in a separate chapter, data transfer to other programs and the implications of Lisp are treated. A very valuable publications that one would like to see available in English, too. [HaagCAM14]

2.3.2.3 Wilhelm Haager: Grafiken mit Maxima, 2011

A tutorial in German on graphics with Maxima of about 35 pages, in the typical, well-edited Haager style. [HaagGM11]

2.3.2.4 Roland Stewen: Maxima in Beispielen, 2013

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]

2.3.3 Physics

2.3.3.1 Ted Woollett: Maxima by example

2.3.3.2 Timberlake and Mixon: Classical Mechanics with Maxima, 2016

In their series *Undergraduate Lecture Notes in Physics*, Springer in 2016 published *Classical Mechanics with Maxima*, written by Todd Keene Timberlake, Prof. of Physics and Astronomy, and J. Wilson Mixon, Jr., Prof. Emeritus of Economics, both at Berry College, Mount Berry, Georgia. This elegantly written, professionally styled and therefore well readable book contains on some 260 pages applications of Maxima to problems from classical mechanics at the undergraduate level. After opening the view to a wide range of problems for symbolical computation from the field of Newtonian mechanics, the book focuses on the programming facilities inherent in the Maxima language and on the methodology and techniques of how to transform sophisticated algorithms for the symbolical or numerical solution of problems from mathematical physics into Maxima. Graphical representations of the data obtained are always in the center of interest, too, and throughout the book vividly illustrate the results from computations. [TimbCMM16]

2.3.4 Engineering

2.3.4.1 Wilhelm Haager: Control Engineering with Maxima, 2017

This well-illustrated tutorial of some 35 pages has been written by Wilhelm Haager from HTL St. Pölten, Austria. It shows applications of Maxima in the field of Electrical Engineering. [HaagCEM17]

2.3.4.2 Tom Fredman: Computer Mathematics for the Engineer, 2014

A free tutorial of 135 pages covering both Maxima and Octave has been written by Tom Fredman of Abo Akademi University, Finnland for applications in Engineering. Its bibliography contains a number of other sources for Maxima applied to engineering. [FredmCME14]

2.3.4.3 Gilberto Urroz: Maxima: Science and Engineering Applications, 2012

The extensive tutorial by Gilberto Urroz used to be available online for free, but now comes as a self-published paperback for a very moderate price, considering its size of 438 pages. It contains a large number of applications in engineering. [UrrozMSE12]

2.3.5 Economics

2.3.5.1 Hammock and Mixon: Microeconomic Theory and Computation, 2013

J. Wilson Mixon, Jr., Professor Emeritus of Economics at Berry College, Mount Berry, Georgia, published *Microeconomic Theory and Computation. Applying the Maxima Open-Source Computer Algebra System* together with Michael R. Hammock in 2013 with Springer. This extensive work of about 385 pages shows how Maxima can be applied to solve a wide variety of symbolical and numerical problems that arise in the field of economics and finance, from exploring empirical relationships between variables up to modeling and analyzing microeconomic systems. This is the most comprehensive book written so far which demonstrates the usefulness of Maxima in Economic Sciences. [HammMTC13]

2.3.5.2 Leydold and Petry: Introduction to Maxima for Economics, 2011

A detailed Maxima tutorial of some 120 pages with applications to Economics has been written by Josef Leydold and Martin Petry from Institute for Statistics and Mathematics, WU Wien. It is based on version 5.25 and was last published in 2011. It is available online as PDF. [LeydoldME11]

2.4 Articles and Papers

A very comprehensive bibliography can be found in [SouzaMaxB04].

2.4.1 Publications by Richard Fateman

Richard J. Fateman, Prof. Emeritus of University of California at Berkeley, Department of Computer Science, who has accompanied this CAS for 50 years, has published a large number of articles and other papers on Macsyma/Maxima. Subjects range from specific technical and algorithmic problems to reflections about the history of Macsyma's development and its place in the evolution of CAS in general. Most references can be found on his Berkeley homepage

http://people.eecs.berkeley.edu/ fateman/.

A considerable number of very interesting papers is available for free download at https://people.eecs.berkeley.edu/ fateman/papers/.

2.5 Comparison with other CAS

2.5.1 Tom Fredman: Computer Mathematics for the Engineer, 2014

A free tutorial of 135 pages covering both Maxima and Octave has been published in 2014 by Tom Fredman of Abo Akademi University, Finnland. [FredmCME14]

2.6 Internal and program documentation

2.7 Mailing list archives

Part II

Basic Operation

Chapter 3

Basics

3.1 Introduction

3.1.1 REPL: The read-evaluate-print loop

Maxima is written in the programming language *Lisp*. Originally, before this language was standardized, *MacLisp*, a dialect developed at MIT, was used, later the Maxima source code was translated to *Common Lisp*, the Lisp standard still valid today. One of the key features of Lisp is the so-called *REPL*, the *read-evaluate-print loop*. When launching Lisp, the user sees a *prompt* where he can enter a Lisp *form*. The Lisp system reads the form, evaluates it and displays the result. After having done this, Lisp outputs the prompt again, giving back the initiative to the user to start a new cycle of operation by entering his next form. The Lisp system primarily works as an interpreter. Nevertheless, functions and packages can also be compiled.

The same basic principle of operation has been employed to the Maxima language, which in this book we will abbreviate *MaximaL*. Maxima also works with a REPL, as being the cycle of interpretation of some expression entered by the user. (Later we will see that Maxima program code can be compiled, too.) This design principle for the user interface was easy to implement and therefore the natural choice in the early times. With one exception, all Maxima front ends still use this principle today. It may seem simple and out of date, but it offers a number of significant advantages which the user will quickly learn to appreciate. The successive loops, as they are operated sequentially and recorded chronologically on the screen, provide a natural log which the user can scroll back at any time to see what he has done and what results he has obtained so far. By simply copying and pasting, the user can take both input and output from previous loops and insert it again at the input prompt. Previous commands can be modified and reentered, and intermediate results can be used for further computation.

But the benefits of this way of working reach even further: when programming in MaximaL, the user can test out every bit of code in the REPL first, before integrating it into his program. Bottom up, step by step, he builds the program, from the most detailed routines to the most abstract layers, always basing every new part on the direct experience in the test environment of his Maxima REPL. This way of programming had proved to be very efficient in Lisp, and with good reason the

same could be expected for Maxima.

This basic principle of operation has been adopted by almost all other computer algebra systems as well. By the way: most CAS' are implemented in Lisp or a Lisp-like language.

Thus, with regard to this general procedure of the REPL, MaximaL and Lisp have a certain similarity. The user who takes the effort to learn Lisp will soon find out that similarities reach much further. However, there are also significant differences. While Lisp is a strictly and visibly list based language working with a non-intuitive, but highly efficient prefix notation, MaximaL is much closer to traditional languages of the Algol-type, more intuitive, more *natural* to the human user, with a structure and notation closer to the mathematical one.

3.1.2 Command line oriented vs. graphical user interfaces

User interfaces in the early times were command line oriented, not graphical. They worked in text mode, centered around a specific spot on the screen, called the *prompt*. Input was done with the keyboard. On hitting *enter*, output followed the input, separated by a simple line-feed. The REPL makes very intelligent use of this initial situation, and many even very experienced CAS users still work with nothing else today. In Maxima this interface is simply called the *console*.

Today, however, one is used to employ the full screen of the computer, the mouse for most users being even more important as input medium than the keyboard. CAS interfaces have been developed that take this evolution into account. *wxMaxima* has been designed in a way similar to the Mathematica notebook, and just as the latter one is most important for Mathematica, wxMaxima is now the predominant Maxima front-end. The basic structural element of this interface is the *cell*, which is a kind of a *local* command line interface. Multiple cells can be created in a Maxima session, allowing the user to work with multiple command line interfaces in parallel. This shows that the basic structure of working with the CAS does not significantly change when moving from the console to wxMaxima. However, the output is no longer displayed in one-dimensional text mode, but in two-dimensional graphical mode, allowing mathematical formulas to be represented in a much more readable way.

We should mention here already that wxMaxima, being based on *wxWidgets*, has significant drawbacks if it comes to error handling, sometimes making it less efficient for sophisticated MaximaL programming and debugging compared to the other front-ends. Between the original console and wxMaxima are a number of Maxima user interfaces which keep the singular REPL, but integrate it in some kind of more graphical environment. Examples are *XMaxima* and *iMaxima*.

Since *Gnuplot* has been integrated into Maxima, output of functions can be done in a fully graphical way with 2D- and 3D-plots in separate windows. 2D-plots can be scrolled in four directions, while 3D-plots can even be turned around easily and freely, with surfaces of adaptable transparency, to be viewed from all perspectives, inside and out, like objects in a CAD program.

3.2 Input and output: using the Maxima REPL at the interactive prompt

3.2.1 Input and output tags

In order to make backward references easier, the cycles of operation of the Maxima REPL are numbered consecutively. On launching a Maxima session at the Maxima console, the user sees the first *input tag*.

```
(%i1)
```

Now he can input a MaximaL expression to be evaluated. We call this a *statement* or *form*. *Enter* starts evaluation. The result (the value returned) is shown with an *output tag* having the same number as the input tag. Then a new input tag appears, introducing the next cycle of operation.

```
(%i1)   2+3;
(%o1)                                5
(%i2)
```

wxMaxima shows a slightly different behavior. The input tag appears only at evaluation time. *Enter* will only cause a line-feed, having no other effect on evaluation than a blank, while *shift-enter* starts evaluation. When an output expression represents the value of a symbol, being either a system variable or a user defined variable, wxMaxima displays no output tag, but instead the symbol in parentheses.

```
(%i1)   temp:-30.5;
(temp)                             -30.5
```

linenum [variable]

Maxima keeps the current tag number in the global variable *linenum*. Entering *linenum:0* or *kill(all)* resets the input and output tag number to 1.

```
(%i17)  linenum:0;
(%o0)                                0
(%i1)   a;
(%o1)                                a
```

inchar default: "*%i*" [variable]
outchar default: "*%o*" [variable]

These global variables contain the symbols used in input and output tags. They can be changed by the user.

3.2.2 Statement termination operators

; [postfix operator]
$ [postfix operator]
, [infix operator]

After entering an input expression, either a semicolon or a dollar sign is expected as a *statement termination operator*. In both cases the next output tag is assigned the result from evaluation of the input expression, but in the latter case, output is not displayed on the screen. Multiple expressions can be entered in the same line, but each of them needs a termination character. They are also expected at the end of every input expression to be processed from a file. Inside of a compound statement, however, the individual statements are not separated by a colon or dollar sign, but by a comma.

3.2.3 Format for input and output

In this section we only introduce the basics. For special options see the next chapter.

3.2.3.1 One- and two-dimensional form

Maxima and all of its front-ends allow input of mathematical expressions only in one-dimensional form. Parentheses have to be used to group subexpressions, e.g. the numerator and denominator of a fraction.

display2d default: *true* [variable]

Output will normally be displayed in two-dimensional form, including in the command-line mode of the console. If the option variable *display2d* is set to *false*, output will be displayed in one-dimensional form as in the input.

3.2.3.2 Entering and display of special characters

The standard Maxima console does not allow for entering and display of special characters. iMaxima displays in Latex output form, thus allowing for the display of special characters. Only wxMaxima allows entering special characters from palettes and also displays them.

3.2.3.3 Display of multiplication operator

The * (asterisk) operator for multiplication cannot be omitted in input; a blank does not mean multiplication.

stardisp default: *false* [variable]

In output, * normally is not displayed, here blank means multiplication. When *stardisp* is set to *true*, however, the * is displayed.

3.2.4 Backward references

Certain system variables allow for easy reference of previous (or current) Maxima input and output. We start with the output.

3.2.4.1 System variables for output

% [variable]

This system variable contains the output expression most recently computed by Maxima, whether or not it was displayed, i.e. the expression with output tag (%o*n*), $n \in \mathbb{N}$ being the most recent cycle having been evaluated. When the output was not displayed, this output tag is not visible on the screen either.

% is recognized by batch and load. In a file processed by batch, % has the same meaning as at the interactive prompt. In a file processed by load, % is bound to the output expression most recently computed at the interactive prompt or in a batch file; % is not bound to output expressions in the file being processed.

Note that a :lisp command does not create an output tag and therefore cannot be referenced by %.

%th (n) [function]

This system function returns the n-th previous output expression, $n \in \mathbb{N}$. Its behavior corresponds to %.

%o*n* [variable]

This system variable contains the output expression with output tag (%o*n*), $n \in \mathbb{N}$. Its behavior corresponds exactly to %.

%% [variable]

In compound statements, namely block, lambda, or (s_1, ..., s_n), this system variable contains the value of the previous statement. At the first statement in a compound statement, or outside of a compound statement, %% is undefined. %% is recognized by batch and load, and it has the same meaning as at the interactive prompt. A compound statement may comprise other compound statements. Whether a statement be simple or compound, %% contains the value of the previous statement. Within a compound statement, the value of %% may be inspected at a break prompt, which is opened by executing the break function.

3.2.4.2 System variables for input

_ (underscore) [variable]

This system variable contains the most recently evaluated input expression, i.e. the expression with input tag (%i*n*), $n \in \mathbb{N}$ being the most recent cycle having been evaluated. _ is assigned the input expression before the input is simplified or evaluated. However, the value of _ is simplified (but not evaluated) when it is displayed.

_ is recognized by batch and load. In a file processed by batch, _ has the same meaning as at the interactive prompt. In a file processed by load, _ is bound to the input expression most recently evaluated at the interactive prompt or in a batch file. _ is not bound to the input expressions in the file being processed.

Note that a :lisp command is not associated with an input tag and cannot be referenced by _.

```
(%i1)   13 + 29;
(%o1)                                42
(%i2)   :lisp $_
        ((MPLUS) 13 29)
(%i2)   _;
(%o2)                                42
(%i3)   sin (%pi/2);
(%o3)                                1
(%i4)   :lisp $_
        ((%SIN) ((MQUOTIENT) $%PI 2))
(%i4)   _;
(%o4)                                1
(%i5)   a: 13$
(%i6)   a + a;
(%o6)                                26
(%i7)   :lisp $_
        ((MPLUS) $A $A)
(%i7)   _;
(%o7)                                2 a
(%i8)   a + a;
(%o8)                                26
(%i9)   ev (_);
(%o9)                                26
```

The above example no only illustrates the _ operator, but also nicely demonstrates the difference between *evaluation* and *simplification*. Although in a broader sense we often talk about "evaluation" when we want to indicate that Maxima processes an input expression in order to compute an output, in the strict sense the meaning of evaluation is limited to *dereferencing*. Everything else is simplification. In the example above, only at %o6, %o8 and %o9 we see evaluation, as the symbol a is dereferenced, i.e. replaced by its value. After this replacement, the addition of the values constitutes another simplification.

%i*n* [variable]

This system variable contains the input expression with input tag (%i*n*), $n \in \mathbb{N}$. Its behavior corresponds exactly to _.

__ (double underscore) [variable]

This system variable contains the input expression *currently being evaluated*. Its behavior corresponds exactly to _. In particular, when *load (filename)* is called from the interactive prompt, __ is bound to *load (filename)* while the file is being processed.

3.3 Basic notation

Here we describe Maxima's basic *notation* conventions which form the basis of the MaximaL *syntax*. Note that statement termination operators were already de-

scribed above.

3.3.1 Compound and separation operators

(, , ...) [matchfix operator]

While in Lisp any kind of *list* is enclosed in *parentheses*, in Maxima these are reserved for specific lists, e.g. the list of parameters of an ordinary function definition, the list of arguments of a function call, or a list of statements in a simple sequential compound statement. The elements are separated by commas.

[, , ...] [matchfix operator]

Square bracketes enclose data lists, e.g. the elements of a one-dimensional list, or the the rows of a matrix. They also enclose the subscripts of a variable, array, hash array, or array function. They are also used to enclose the local variable definitions of a block. The elements are separated by commas.

```
(%i1)   x: [a,b,c];
(%o1)                          [a,b,c]
(%i2)   x[3];
(%o2)                            c
(%i3)   array(y,fixnum,3);
(%o3)                            y
(%i4)   y[2]: %pi;
(%o4)                            π
(%i5)   y[2];
(%o5)                            π
(%i6)   z[a]:b;
(%o6)                            b
(%i7)   z[a];
(%o7)                            b
(%i8)   g[k] := 1/(k^2+1);
                                 1
(%o8)                          ------
                                 2
                                k  + 1
(%i9)   g[10];
                                 1
(%o9)                          ----
                               101
```

{ , , ...} [matchfix operator]

Braces enclose sets. The elements are separated by commas. Note that the elements of a set, unlike a list, are not ordered.

, [infix operator]

Separator of elements of a list or set. Note that in Lisp, instead, the separation character of a list is the blank.

3.3.2 Identity and relational operators

= [infix operator]
[infix operator]
is (expr) [function]

= and # are the *equation* and inequation operators. They are binary operators. Chains like a = b = c are not allowed.

When Maxima encounters an equation, its arguments, which means the left-hand side and the right-hand side, are evaluated and simplified separately. The operator = by itself does nothing more. It does not compare the two sides at all and the two sides are not simplified against each other. An expression like a = b represents an *unevaluated equation*, which might or might not hold. Unevaluated equations may be passed as arguments to *solve*, *algsys* or some other functions.

```
(%i1)   c:3$ d:3$
(%i3)   a+a=c+d;
(%o3)                              2a=6
(%i4)   a+b=a+e;
(%o4)                             b+a=e+a;
```

Any desired simplification across the = operator has to be carried out manually. For example, functions *rhs(eq)* and *lhs(eq)* return the right-hand side and left-hand side, respectively, of an equation or inequation. Using them, we can indirectly achieve some basic simplification of an unevaluated equation by subtracting one side from the other, thus, bringing them both to one side. Of course, the user may write his own simplification routines to handle specific situations, as for example to subtract equal terms on both sides, to divide both sides by a common factor, etc.

```
(%i1)   c:3$ d:3$
(%i3)   eq: a+a=c+d;
(%o3)                              2a=6
(%i4)   eq/2;
(%o4)                              a=3

(%i5)   eq: a+b=a+e;
(%o5)                             b+a=e+a;
(%i6)   lhs(eq)-rhs(eq)=0;
(%o6)                              b-e=0;
```

Function *is* evaluates an equation to a Boolean value. *is(a = b)* evaluates *a = b* to *true* if a and b, after each having been evaluated and simplified separately, which includes bringing them into canonical form, are *syntactically equal*. This means, string(a) is identical to string(b). This is the case if a and b are atoms which are identical, or they are not atoms and their operators are all identical and their arguments are all identical. Otherwise, *is(a = b)* evaluates to *false*; *is* never evaluates to *unknown*.

Note that in contrast to function *equal*, *is(a=b)* does not check assumptions in Maxima's database. Thus, Maxima properties of a and b are not considered, only their values. Assumptions of equality cannot be specified with the = operator, only with

function *equal*.

```
(%i1)   assume(a=b);
                                  Error!
(%i2)   assume(equal(a,b))$
(%i3)   is(a=b);
(%o3)                             false
(%i4)   is(equal(a,b));
(%o4)                             true
```

The negation of = is represented by #. *is(a # b)* evaluates *a # b* to *true* or *false*. Note that because of the rules for evaluation of predicate expressions (in particular because *not expr* causes evaluation of expr), *not a = b* is equivalent to *is(a # b)*, and not to *a # b*. Assumptions of inequality cannot be specified with the # operator, only with function *notequal*.

In addition to function *is*, some other operators evaluate = and # to *true* or *false*, namely *if*, *while*, *unless*, *and*, *or*, and *not*.

equal (a,b) [function]
notequal (a,b) [function]

These functions, by themselves, like = and #, do nothing more than evaluate both arguments. Unlike a = b, however, *equal(a,b)* is not an *unevaluated equation* which can be passed as an argument to *solve*, *algsys* or some other functions.

Functions *equal* and *notequal* can be used to specify assumptions with *assume*.

Function *is* tries to evaluate *equal(a,b)* to a Boolean value. *is(equal(a,b))* evaluates *equal(a,b)* to *true*, if a and b are *mathematically equivalent expressions*. This means, they are mathematically equal for all possible values of their arguments. Comparison is carried out and equivalence established by checking the Maxima database for user-postulated assumptions, and by checking whether *ratsimp(a-b)* returns zero.

Assumptions are stored as Maxima properties of the variables concerned. Thus, comparison can be carried out and equivalence established between variables which are unbound, having no (numerical or symbolical) values assigned. But comparison can also be carried out and equivalence established by retrieving the variables' values (process of evaluation, dereferencing) and subsequent simplification. Of course, a combination of both methods is possible, too.

```
(%i1)   c:3$ d:3$
(%i3)   equal(a+a, c+d);
(%o3)                             2a=6
(%i4)   equal(a+b, a+e);
(%o4)                           b+a=e+a;

(%i5)   assume(equal(a,b))$ assume(e<f)$
(%i6)   is(a=b);
(%o6)                             false
(%i7)   is(equal(a,b));
(%o7)                             true
```

```
(%i8)   is(equal(e,f));
(%o8)                                false

(%i9)   is(x^2-1 = (x+1)*(x-1));
(%o9)                                false
(%i10)  is(equal(x^2-1, (x+1)*(x-1)));
(%o10)                               true

(%i11)  is(equal((a-1)*a, b^2-b));
(%o11)                               true

(%i12)  is(equal(sinh(x), (%e^x-%e^-x)/2));
(%o12)                             unknown
(%i13)  exponentialize: true$
(%i14)  is(equal(sinh(x), (%e^x-%e^-x)/2));
(%o14)                               true
```

When *is* fails to reduce *equal* to *true* or *false*, the result is governed by the global flag *prederror*. When *prederror* is *true*, *is* returns an error message. Otherwise (default), it returns *unknown*.

notequal (a,b) represents the negation of *equal(a,b)*. Because *not expr* causes evaluation of *expr*, *not equal(a,b)* is equivalent to *is(notequal(a,b))*.

<	[infix operator]
>	[infix operator]
<=	[infix operator]
>=	[infix operator]

These are the *relational operators*. They are binary operators. Chains like *a<b<c* are not allowed. Just like = and #, relational operators do nothing more than evaluate and simplify their arguments separately. An expression like $a < b$ is an *unevaluated relational expression*, which might or might not hold. Any desired simplification across the relational operator has to be carried out manually. Function *solve* does not accept relational expressions.

Relational operators can be used to specify assumptions with *assume*.

Function *is* tries to evaluate a relational expression like *a<b* to a Boolean value. Comparison is carried out by checking Maxima's database for user-postulated assumptions, and by checking what *ratsimp(a-b)* returns. Thus, as for functions *equal* and *notequal*, both Maxima properties of the variables concerned and their values are considered.

```
(%i1)   assume(-1<x, x<0)$
(%i2)   is(diff((x-t)/(1+t),t)<0);
(%o2)                                 true
(%i3)   factor(diff((x-t)/(1+t),t));
                                      x + 1
(%o3)                               - ------
                                      2
                                     (t + 1)
```

When *is* fails to reduce a relational expression to *true* or *false*, the result is governed by the global flag *prederror*. When *prederror* is *true*, *is* returns an error message. Otherwise (default), it returns *unknown*.

In addition to function *is*, some other operators evaluate relational expressions to *true* or *false*, namely *if*, *while*, *unless*, *and*, *or*, and *not*.

3.3.3 Assignment operators

: [infix operator]

This is the basic *assignment operator*. When the left-hand side is a simple variable (not subscripted), : evaluates its right-hand side and associates that value with the symbol on the left-hand side. Chain constructions are allowed; in this case all positions but the right-most one are considered left hand side.

When the left-hand side is a subscripted element of a list, matrix, declared Maxima array, or Lisp array, the right-hand side is assigned to that element. The subscript must name an existing element; such objects cannot be extended by naming nonexistent elements.

When the left-hand side is a subscripted element of an undeclared Maxima array, the right-hand side is assigned to that element, if it already exists, or a new element is allocated, if it does not already exist.

When the left-hand side is a list of simple and/or subscripted variables, the right-hand side must evaluate to a list, and the elements of the right-hand side are assigned to the elements of the left-hand side, element by element, in parallel (not in serial; thus evaluation of an element may not depend on the evaluation of a preceding one).

```
(%i1)   x : y : 3;
(%o1)                                3
(%i2)   x;
(%o2)                                3
(%i3)   y;
(%o3)                                3
(%i4)   [a, b, c] : [4, 7, 10];
(%o4)                             [4, 7, 10]
(%i5)   a;
(%o5)                                4
```

:: [infix operator]

This is the *indirect assignment operator*. :: is the same as :, except that :: evaluates its left-hand side as well as its right-hand side. Thus, the evaluated right-hand side is assigned not to the symbol on the left-hand side, but to the *value* of the variable on the left-hand side, which itself has to be a symbol.

```
(%i1)   x : 'y;
(%o1)                                y
(%i2)   x :: 123;
(%o2)                               123
```

```
(%i3)   x;
(%o3)                                y
(%i4)   y;
(%o4)                               123
(%i5)   x : '[a, b, c];
(%o5)                            [a, b, c]
(%i6)   x :: [1, 2, 3];
(%o6)                            [1, 2, 3]
(%i7)   a;
(%o7)                                1
(%i8)   b;
(%o8)                                2
(%i9)   c;
(%o9)                                3
```

A value (and other bindings) can be removed from a variable by functions *kill* and *remvalue*. These *unassignment functions* are more important than they might seem. Unbinding variables from values no longer needed should be made a habit by the user, because forgetting about assigned values is a frequent cause of mistakes in following computations which use the same variables in other contexts.

3.3.4 Substitution of symbol by value in an expression

subst (eq_1, expr) [function]
subst ([eq_1,. . . ,eq_k], expr)

This is a special format of the general substitution function *subst*. It allows for the substitution of symbols (or certain subexpressions) in an expression *expr* by numerical values, other symbols, or expressions.

The *eq_i* are equations of the form b=a indicating substitutions to be made in *expr*. For each equation, the right side will be substituted for the left in *expr*. *subst(b=a, c)* is equivalent to *subst(a, b, c)*. Like in the general form of *subst*, b must be an atom or a complete subexpression of *expr*. The equations are substituted in serial from left to right in *expr*.

sublis ([eq_1,. . . ,eq_k], expr) [function]

This is the same as the corresponding form of *subst*, but the substitutions are done in parallel. As opposed to *subst*, the left side of the equation must be an atom; a complete subexpression of *expr* is not allowed. The form with a single equation as the first argument is not allowed, either.

```
(%i1)   subst([a=b, b=c], a+b);
(%o1)                               2 c
(%i2)   sublis([a=b, b=c], a+b);
(%o2)                               c+b
```

While an assignment has a global effect, substitution affects only the expression *expr*. Substitution should therefore be preferred whenever possible.

3.3.5 Function and macro definition operators

3.3.5.1 Function definition operator

:= [infix operator]

This is the *function definition operator*. A *user function* has to be defined before it can be used, i.e. *called*. See section 26.2 for details.

3.3.5.2 Macro function definition operator

::= [infix operator]

This is the *macro function definition operator*. A MaximaL macro function is very similar to a Lisp macro. The difference to an ordinary MaximaL function is the following:

— a macro function when being called does not evaluate its arguments (before the macro function itself is evaluated). We say that a macro function *quotes* its arguments, because a Maxima *quote operator* inhibits evaluation of its arguments,

— the macro call returns a *form* which is itself evaluated in the context from which the macro was called. Thus, what the macro function returns is itself a Maxima *statement* which will subsequently be evaluated. The expression returned by a macro call is called *macro expansion*, a term which again refers to the macro concept in Lisp.

A macro function is otherwise the same as an ordinary function.

3.3.6 Miscellaneous operators

/ ... */* [matchfix operator]

This is the *comment operator*. Any input in-between will be ignored.

? [prefix operator]
? [prefix operator]

These are the *documentation operators*. ? placed before a system function name f (and separated from it by a blank) is a synonym for *describe (f)*. This will cause the online documentation about system function f to be displayed on the screen.

?? placed before a system function name f (and separated from it by a blank) is a synonym for *describe (f, inexact)*. This will cause the online documentation about function f and all other system functions having a name which starts with "f" to be displayed on the screen.

3.4 Naming of identifiers

3.4.1 Naming specifications

3.4.1.1 Case sensitivity

Symbols (identifiers) in Maxima are *case-sensitive*, i.e. Maxima distinguishes between upper-case (capital) and lower-case letters. Thus, *NAME*, *Name* and *name*

are all different symbols and may denote different variables.

3.4.1.2 ASCII standard

Maxima identifiers may comprise *alphabetic characters*, the *digits* 0 through 9, the underscore _, the percent sign %, and any *special character* preceded by the backslash character. A digit may be the first character of an identifier, if it is preceded by a backslash. Digits which are the second or later characters need not be preceded by a backslash.

alphabetic [property]

Special characters may be declared *alphabetic* using the *declare* function. If so declared, they need not be preceded by a backslash in an identifier. The special characters declared *alphabetic* are initially %, and _. The list of all characters presently declared *alphabetic* can be seen as the Lisp variable **alphabet**.

Since almost all special characters from the ASCII code set are in use for other purposes in Maxima, often as operators for which the parser pays special attention, it makes little sense to declare them alphabetic. Thus, we have taken an example with non-ASCII characters (which does not make much more sense, as we will soon see).

```
(%i1)   declare("äöüÄÖÜß",alphabetic);
(%o1)                                done
(%i2)   Größe : 123;
(%o2)                                123
(%i3)   :lisp *alphabet*
(_ % ä ö ü Ä Ö Ü ß)
(%i4)   featurep("ä",alphabetic);
(%o4)                                true
```

All characters in the string passed to *declare* as the first argument are declared to be *alphabetic*. Function *featurep* returns true, if all characters in the string passed to it as the first argument have been declared *alphabetic* by the user or are the _ or % characters.

3.4.1.3 Unicode support

Recently, efforts have been made to include Unicode support in Maxima. It has to be stated, however, that Unicode support is not a universal feature of Maxima, but depends to some extend on the operating system, on the Lisp and on the front-end used. Given that our actual system supports it, almost any Unicode character can nowadays be used within a Maxima identifier, including in the first position. Thus, we do not need to declare German Umlaute as *alphabetic*, we can just use them. We can use Greek letters, too, or even Chinese.

Special attention has to be payed, though, when using non-ASCII characters. If things work well on one system, this does not guarantee it will work without problems on another one. Besides, there might still be issues in some situations and circumstances that have not been solved in a satisfactory way yet.

As a general statement we can say that Linux gives better and more consistent Unicode support than Windows. Concerning the Lisp, we find that SBCL is always a good choice, combining most efficient behavior with least problems. From the point of view of the front-ends, wxMaxima takes most efforts to provide comprehensive Unicode support.

3.4.1.3.1 Implementation notes

Maxima uses Lisp function *alphabetp* to determine whether a character is allowed as an alphabetic character in an identifier. This function refers to CL system function *alpha-char-p*. In a working UTF8 environment, this will allow almost any Unicode character except for punctuation and digits. In addition, *alphabetp* checks the global variable **alphabet** for characters declared *alphabetic* by the user.

3.4.2 Basic naming conventions

3.4.2.1 System functions and variables

In general, Maxima's system functions and variables use lower-case letters only and use the underscore character to separate words within a symbol, e.g. *cartesian_product*.

In order to clearly distinguish them from system functions, our own additional functions and variables start with capital letters and use capitol letters to separate words within a symbol, e.g. *ExtractCequations*.

3.4.2.2 System constants

System constants like the imaginary unit i, the Euler's number e, or the constants π and γ are preceded by % in Maxima (i.e. %i, %e, %pi, %gamma) to make them better distinguishable from ordinary letters or identifiers. One has to keep this in mind in order not to be confused. Note in the following example that *log* denotes the natural logarithm with base e. Maxima and its system functions return the input expression, if they cannot evaluate it.

```
(%i1)   %e^log(x);
(%o1)                                x
(%i2)   e^log(x);
                                  log(x)
(%o2)                            e
(%i3)   %pi;
(%o3)                               %pi
(%i4)   float(%pi);
(%o4)                        2.128231705849268
(%i5)   float(pi);
(%o6)                               pi
```

wxMaxima will return π both in number 3 and 6. In 3 it denotes the constant, in 6 the lower-case Greek letter.

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;
```

$$(\%o2) \qquad -\frac{2}{3}$$

```
(%i3)   100!;
(%o3)   933262154439441526816992388562667004907159682643816214685929\
        638952175999932299156089414639761565182862536979208272237582\
        51185210916864000000000000000000000000
```

7.1.2.1.2 Internal

```
(%i1)   a:1/2;
```

$$(\%o1) \qquad \frac{1}{2}$$

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

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

```
(%o1)                                  1/4
(%o2)                               2340.0
(%i2) a+b+c;
(%o2)                           2340.25 + c
```

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 $a + i\,b$, *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 $a + i\,b$ 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
```

7.2 Constants

7.3 Strings

7.4 Lists

7.5 Structures

Chapter 8

Expressions, operators

8.1 Operators

"="(a,b) is a functional notation equivalent to a=b.

Chapter 9

Evaluation

Chapter 10

Simplification

10.1 Properties for simplification

10.2 Functions for simplification

PullFactorOut(expr, factor) [function of *rs_simplification*]

Pulls *factor* out of *expr*. If *expr* is a list, a vector or a matrix, *factor* is pulled out of every component.

PullMinusIntoFraction (expr, num_denom, side) [funktion of *rs_simplification*]

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 ($context_1, ..., context_n$) [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 non-constant value to a_j. Such an assignment, on the other hand, does not remove the property *constant* from a_j. The following predicate function *constantp* not only tests for a variable declared *constant*, but for a constant expression in general.

constantp (expr) [predicate function]

Returns *true*, if expr is a constant expression, otherwise *false*. An expression is considered a constant expression, if its arguments are numbers (including ratio-

nal 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 ... AND $pred_n$" are recognized, but not "$pred_1$ OR ... 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]
(%i2)   is (a + b > b + c);
(%o2)                                true
(%i3)   is (equal (a, c));
(%o3)                                false
(%i4)   is (2*a > 3*c);
(%o4)                               unknown
(%i5)   assume (equal(d,5));
(%o5)                           [equal(d,5)]
(%i6)   is (equal (d, 5));
(%o6)                                true
(%i7)   is (d=5);
(%o7)                                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);
(%i1)                              [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);
(%i1)                              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 Representations and their internal data structures

Maxima does not have a specific data structure for vectors. A vector can be represented as a list or as a matrix of either one column or one row. These possible representations and their internal Lisp data structure equivalents are demonstrated:

```
(%i1)   u:[x,y,z];                              /* This is a MaximaL list.
   */
(%o1)                           [x, y, z]
(%i2)   :lisp $u
                          ((MLIST SIMP) x y z)
(%i3)   v:covect(u);                      /* This creates a column vector.
   */
```

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

```
(%i4)   :lisp $v
        (($MATRIX SIMP) ((MLIST SIMP) x) ((MLIST SIMP) y) ((MLIST SIMP) z))
(%i5)   w:transpose(v);                       /* This creates a row vector.
   */
```

$$(\%o5) \qquad \begin{pmatrix} x & y & z \end{pmatrix}$$

```
(%i6)   :lisp $w
                  (($MATRIX SIMP) ((MLIST SIMP) x y z))
```

Note that a matrix internally is a list of MaximaL lists, each of them representing one row.

21.1.2 Create, enter, transform and transpose vectors

A list can simply be entered by typing the elements inside of square brackets, separated by commas. Special functions for creating lists (e.g. *makelist* and *create_list*) are described in section 7.4 on lists.

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

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

```
(%i2) Rvect(x,y,z);
```

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

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. The first argument of this function must not be bound and must not have any properties.

```
(%i1) x:Make_cvect(x,3);
```

(%o1)
$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

```
(%i2) y:Make_rvect(y,3);
```

(%o2)
$$\begin{pmatrix} y_1 & y_2 & y_3 \end{pmatrix}$$

Note that system function *genmatrix* can be used to construct a column vector, too, but with symbolic elements having two indices instead of one.

```
(%i1) x:genmatrix(x,3,1);
```

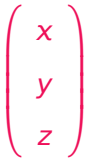
(%o1)
$$\begin{pmatrix} x_{1,1} \\ x_{2,1} \\ x_{3,1} \end{pmatrix}$$

The following functions achieve transformations between the different representations.

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)


transpose (v) [function]

Transposes a list or a row vector into a column vector, and a column vector into a row vector.

Transpose (v) [function of *rs_algebra*]

Transposes a list or a row vector into a column vector, and a column vector into a list.

Vlist (v) [function of *rs_algebra*]

Transforms a vector of any kind into a list. If v is already a list, it will be returned.

Note that *transpose(Vlist(v))* and *transpose(transpose(Vlist(v)))* will transform a vector of any kind into a column vector and a row vector respectively.

21.1.3 Dimension of a vector

System function *length(v)* can be used to determine the dimension of a column vector or list. We should not talk about the *length* of a vector here, because this term is used for the *norm* of a vector, too.

Vdim(v) [function of *rs_algebra*]

Returns the dimension of a vector, independent of its representation.

21.1.4 Addressing the elements of a vector

While elements of a list are addressed simply by providing the number of the element in square brackets, elements of a column vector or a row vector (as being matrices) are addressed by two arguments in square brackets, separated by a comma, where the first argument specifies the row and the second one the column.

21.1.5 Arithmetic operations between vectors, scalar multiplication, other MaximaL functions applicable to vectors

listarith [option variable]

Scalar multiplication of a vector and arithmetic operations between vectors work component-wise, if the flag *listarith* is *true*, which is the default. They are only possible between vectors of the same type, with the exception that lists and column vectors can be combined. In this case, the result will always be a column vector.

doallmxops [option variable]
distribute_over [option variable]

Many other computational or simplifying/manipulating MaximaL functions can be applied to vectors, which means that they operate component-wise. The flags

doallmxops and *distribute_over* must be *true* (default). Examples are diff, factor, expand.

21.1.6 Scalar product

The scalar product of two real valued vectors, which, in case of a list representation, is equal to *sum (a[i]*b[i], i, 1, length(a))* can be built with the dot operator. The arguments need to have the same dimension, but can be of any kind, except for the combination *c.r*, where c is a column vector and r is a row vector or a list. This combination, instead, will return the tensor product of the two vectors. Hence, this operator is non-commutative with respect to the combination of vector representations.

```
(%i1) powerdisp:true$
(%i2) v:Make_cvect(v,3)$  w:Make_cvect(w,3)$
(%i3) v.w;
(%o3)                          v_1 w_1 + v_2 w_2 + v_3 w_3
```

The dot operator, more generally applied to matrices, computes the (also non-commutative) matrix product. It is controlled by a number of flags which will be described there.

v SP w [infix operator of *rs_algebra*]

The infix operator *SP* computes the scalar product of two real valued vectors of equal dimension, independent of the representations and their combination. Hence, it is a commutative version of the dot operator. Both vectors are internally transformed to column vectors first, then the dot operator is employed. By this procedure all flags which control the dot operator stay valid.

```
(%i1) v:Make_cvect(v,3)$  w:Make_rvect(w,3)$
(%i2) c SP r;
(%o2)                          v_1 w_1 + v_2 w_2 + v_3 w_3
```

21.1.7 Tensor product

The tensor product $v \otimes w$ can be created with the dot operator, see scalar product, if the first argument is a column vector of dimension m and the second argument is either a row vector or a list of dimension n. The arguments need not have the same dimension. The result will be an $m \times n$ matrix. Here again the dot operator is non-commutative with respect to the combination of vector representations. For a description of the flags that control the dot operator, see the matrix product.

v TP w [infix operator of *rs_algebra*]

The infix operator *TP* computes the tensor product of two vectors of any representation. The arguments need not have the same dimension. It will return an $m \times n$ matrix. This is a commutative version of the dot operator. Internally, the first argument is transformed to a column vectors, the second one to a row vector, then the dot operator is employed. By this procedure all flags which control the dot operator stay valid.

```
(%i1) v:Make_cvect(v,3)$  w:Make_cvect(w,3)$
(%i2) v TP w;
```

(%o2)
$$\begin{pmatrix} v_1 w_1 & v_2 w_2 & v_1 w_3 \\ v_2 w_1 & v_2 w_2 & v_2 w_3 \\ v_3 w_1 & v_3 w_2 & v_3 w_3 \end{pmatrix}$$

21.1.8 Vector norm and normalization of vectors

Vnorm(v) [function of *rs_algebra*]

Computes the Euclidean norm (2-norm) of vector v according to the formula $\sqrt{v \cdot v}$.

```
(%i1) v:Make_cvect(v,3)$
(%i2) Vnorm(v);
```

(%o2)
$$\sqrt{v_1^2 + v_2^2 + v_3^2}$$

Normalize (v[,"r"]) [function of *rs_algebra*]

This function normalizes a vector v of any representation. It can also be used to normalize the columns or rows of a matrix v. This is useful for matrices which represent a group of column or row vectors, e.g. as being the basis of a vector space. In case a matrix shall be normalized row-wise, a second argument "r" has to be passed (in quotes).

```
(%i1) X:[1,2,3] TP [1,2,3];
```

(%o1)
$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

```
(%i2) Normalize(X);
```

(%o2)
$$\begin{pmatrix} \frac{1}{\sqrt{14}} & \frac{1}{\sqrt{14}} & \frac{1}{\sqrt{14}} \\ \frac{2}{\sqrt{14}} & \frac{2}{\sqrt{14}} & \frac{2}{\sqrt{14}} \\ \frac{3}{\sqrt{14}} & \frac{3}{\sqrt{14}} & \frac{3}{\sqrt{14}} \end{pmatrix}$$

```
(%i3) Normalize(X, "r");
```

(%o3)
$$\begin{pmatrix} \frac{1}{\sqrt{14}} & \frac{2}{\sqrt{14}} & \frac{3}{\sqrt{14}} \\ \frac{1}{\sqrt{14}} & \frac{2}{\sqrt{14}} & \frac{3}{\sqrt{14}} \\ \frac{1}{\sqrt{14}} & \frac{2}{\sqrt{14}} & \frac{3}{\sqrt{14}} \end{pmatrix}$$

21.1.9 Vector equations

ExtractCequations (arg) [function of *rs_algebra*]

Extracts the component equations from a vector equation *arg*. The vectors on the right and on the left side of the equation may be of any, but must be of identical representation, with the exception that a combination of lists and column vectors is possible, too. After the simplfications done at evaluation time of *arg*, this vector

equation has to be condensed to only one vector on each side. Use all kinds of simplification functions first, if this is not guaranteed. ExtractCequations returns a list of Vdim(arg) component equations which e.g. can be forwarded to function *solve*.

```
(%i1) u:Make_cvect(u,3)$  v:Make_cvect(v,3)$
(%i2) w:makelist(w[i],i,1,3)$
(%i3) Extract_cequations(u+v=w);
```

(%o3) $[v_1 + u_1 = w_1, v_2 + u_2 = w_2, v_3 + u_3 = w_3]$

21.1.10 Vector product

Standard Maxima has no operator to compute the vector or cross product between two 3-dimensional vectors.

v VP w [infix operator of *rs_algebra*]

The infix operator *VP* computes the vector product of two vectors of any representation, but dimension three, returning a column vector.

```
(%i1) v:Make_cvect(v,3)$  w:Make_cvect(w,3)$
(%i3) v VP w;
```

(%o3)
$$\begin{pmatrix} v_2 w_3 - w_2 v_3 \\ v_1 w_3 - w_1 v_3 \\ v_1 w_2 - w_1 v_2 \end{pmatrix}$$

21.1.11 Mixed product and double vector product

These products, of course, can be computed by combining the operations of scalar and vector product. The mixed product is

```
(%i1) v:Make_cvect(v,3)$  w:Make_cvect(w,3)$  u:Make_cvect(u,3)$
(%i4) expand(u SP (v VP w));
```

(%o4) $u_1 v_2 w_3 - v_1 u_2 w_3 - u_1 w_2 v_3 + w_1 u_2 v_3 + v_1 w_2 u_3 - w_1 v_2 u_3$

```
(%i5) expand((u VP v) SP w));
```

(%o5) $u_1 v_2 w_3 - v_1 u_2 w_3 - u_1 w_2 v_3 + w_1 u_2 v_3 + v_1 w_2 u_3 - w_1 v_2 u_3$

And for the double vector product we get

```
(%i7) expand(u VP (v VP w));
```

(%o7)
$$\begin{pmatrix} v_1 u_3 w_3 - w_1 u_3 v_3 + v_1 u_2 w_2 - w_1 u_2 v_2 \\ v_2 u_3 w_3 - w_2 u_3 v_3 - u_1 v_1 w_2 + u_1 w_1 v_2 \\ -u_2 v_2 w_3 - u_1 v_1 w_3 + u_2 w_2 v_3 + u_1 w_1 v_3 \end{pmatrix}$$

21.2 Matrix

21.2.1 Create a matrix

21.2.2 Transpose a matrix

21.2.3 Addition and scalar multiplication

21.2.4 Invert a matrix

21.2.5 Matrix product

21.3 Determinant

Chapter 22

Analytic geometry

22.1 Representation and transformation of angles

22.1.1 Degrees ⇆ radiant

Rad2deg(angle) [function of *rs_angles*]
Rad2degf(angle) [function of *rs_angles*]
Deg2rad(angle) [function of *rs_angles*]
Deg2radf(angle) [function of *rs_angles*]

These functions transform *angle* from radiant to degrees and vice versa. While functions not ending with an f return ratios (and multiples of π) wherever possible, the functions ending with an f return floats rounded to 3 (Rad2degf) resp. 5 (Deg2Radf) digits after the dot.

22.1.2 Degrees decimal ⇆ min/sec

Degdec2min(angle,n) [function of *rs_angles*]
Degmin2dec(angle,n) [function of *rs_angles*]
Concminsec(angle) [function of *rs_angles*]

Degdec2min(angle,n) converts *angle* given in degrees decimally into a list of 3 elements containing degrees, minutes and seconds. Seconds are rounded to a maximum of $n \geq 0$ digits after the dot.

Degmin2dec(angle,n) converts *angle* given in a list of 3 elements containing degrees, minutes and seconds into degrees decimally, rounded to a maximum of $n \geq 0$ digits after the dot.

Concminsec(angle) converts *angle* given as a list with 3 elements containing degree, min, sec into a string with the elements separated by "°", "’" and "”" respectively.

22.1.3 $(-\pi, \pi) \leftrightarrows (0, 2\pi)$

The range of angles in radiant for a full circle can be defined to be either $(-\pi, \pi]$ or $[0, 2\pi)$. The following functions transform an angle from one of these ranges to the other.

Pos2pi(angle) [function of *rs_angles*]
Negpospi(angle) [function of *rs_angles*]

Pos2pi transforms *angle*, given in radiant, from range $(-\pi, \pi]$ to range $[0, 2\pi)$. *Negpospi* transforms *angle*, given in radiant, from range $[0, 2\pi)$ to range $(-\pi, \pi]$.

Angles given in degrees can be likewise transformed between the two corresponding ranges with *Rad2deg(Pos2pi(Deg2rad(angle)))* or *Rad2deg(Negpospi(Deg2rad(angle)))*.

Chapter 23

Coordinate systems

Part V

Advanced Mathematical Computation

Chapter 24

Tensors

Chapter 25

Numerical Computation

Part VI

Maxima Programming

Chapter 26

Compound statements

26.1 Sequential and block

26.1.1 Sequential

(expr$_1$, . . . , expr$_n$) [matchfix operator]

A number of statements can be enclosed in parentheses and separated by commas. Such a list of *sub-statements* is the most simple form of a compound statement We call it a *sequential*. Maxima evaluates the sub-statements in sequence and only returns the value of the last one.

26.1.2 Block

block ([v$_1$, . . . , v$_n$], expr$_1$, . . . , expr$_m$) [function]
block ([v$_1$, . . . , v$_n$], local (v$_1$, . . . , v$_n$), expr$_1$, . . . , expr$_m$)

A *block* allows to make variables $v_1, \ldots, v_m$ local to the sequence of sub-statements $expr_1, \ldots, expr_m$. If these variables (symbols) are already bound, block saves their current values upon entry to the block and then unbinds the symbols so that they evaluate to themselves. The local variables may then be bound to arbitrary values within the block. When the block is exited, the saved values are restored, and the values assigned within the block are lost.

Note that the *block* declaration of the first line will make variables $v_1, \ldots, v_m$ local only with respect to their values. However, in Maxima, just like in Lisp, a large number of qualities can be attributed to symbols by means of *properties*. Properties of $v_1, \ldots, v_m$ are not made local by a plain block declaration! They stay global, which means that properties already assigned to these symbols on entry to the block will remain inside of the block, and properties assigned to these symbols inside of the block will not be removed on exiting the block. In order to make symbols $v_1, \ldots, v_m$ local to the block with respect to their properties, too, they have to be declared with function *local* inside of the block. For example, some declarations of a symbol are implemented as properties of that symbol, including *:=, array, dependencies, atvalue, matchdeclare, atomgrad, constant, nonscalar, assume*. *local* saves and removes such declarations, if they exist, and makes declarations done within the block effective only inside of the block; otherwise such declarations done within a block are actually global declarations.

A block may appear within another block. Local variables are established each time a new block is evaluated. Local variables appear to be global to any *enclosed blocks*. If a variable is non-local in a block, its value is the value most recently assigned by an *enclosing block*, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of *dynamic scope*.

The value of the block is the value of its last sub-statement, or the value of the argument to the function *return*, which may be used to exit explicitly from the block at any point.

The function *go* may be used to transfer control to the statement of the block that is tagged with the argument to *go*. To tag a statement, precede it by an atomic argument as another sub-statement in the block. For example:
block ([x], x:1, loop, x: x+1, ..., go (loop), ...).
The argument to *go* must be the name of a tag appearing within the block; one cannot use *go* to transfer to a tag in a block other than the one containing the *go*. Using labels and *go* to transfer control, however, is unfashionable and not recommended.

local $(v_1, \ldots, v_n)$ [function]

The declaration *local* $(v_1, \ldots, v_m)$ within a block saves the properties associated with the symbols $v_1, \ldots, v_m$, removes them from the symbols, and restores any saved properties on exit from the block. This statement should best be placed directly after the list of the local variables at the beginning of the *block*.

26.2 Function

26.2.1 Function definition

:= [infix operator]

$f(x_1, \ldots, x_n)$ *:=* *expr*
":="$(f(x_1, \ldots, x_n), expr)$

define $(f(x_1, \ldots, x_n), expr)$ [function]

A *function* can be defined either with the *function definition operator :=* or with function *define*. Both ways are similar, but not identical. The similarity can be seen more clearly if the *:=* operator is written as an *operator function*. The difference between *:=* and *define* is that *:=* never evaluates the parameters or the function body unless explicitly forced by quote-quote ' ', whereas *define* always evaluates the parameters and the function body unless explicitly prevented by quote. The function name is not evaluated in either case. If the function name is to be evaluated, one of the following expressions can be used

define (funmake (f, $[x_1, \ldots, x_n]$*), expr)*
define (funmake (f $[x_1, \ldots, x_n]$*,* $[y_1, \ldots, y_n]$*), expr)*
define (arraymake (f, $[x_1, \ldots, x_n]$*), expr)*
define (ev ($expr_1$*),* $expr_2$*)*.

The first expression using *funmake* returns an *ordinary function* with parameters in parentheses, see section 26.2.2. The expression using *arraymake* returns an *array function* with parameters in square brackets, see section 26.2.3. The second expression using *funmake* returns a *subscripted function*, see section 26.2.4. The expression with *ev* can be used in any case.

```
(%i1)   f:g $  u:x $
(%i3)   define (funmake (f, [u]), cos(u) + 1);
(%o3)                        g(x) := cos(x) + 1
(%i4)   define (arraymake (f, [u]), cos(u) + 1);
(%o4)                          g_x := cos(x) + 1
(%i5)   define (f(x,y), g (y,x));
(%o5)                        f(x,y) := g(y,x)
(%i6)   define (ev(f(x,y)), sin(x) - cos(y));
(%o6)                      g(y,x) := sin(x) - cos(y)
```

26.2.2 Ordinary function

f($x_1, \ldots, x_n$) := *expr*
f($x_1, \ldots, x_n$) := *block* ([$v_1, \ldots, v_p$], $expr_1, \ldots, expr_m$)
f($x_1, \ldots, x_n$) := *block* ([$v_1, \ldots, v_p$], *local* ($x_1, \ldots, x_n, v_1, \ldots, v_p$), $expr_1, \ldots, expr_m$)

The first line defines a function named f with *parameters* $x_1, \ldots, x_n$ and *function body expr*.

An *ordinary function* is a function which encloses its parameters (at function definition) and arguments (at function call) with parentheses (). The function body of an ordinary function is evaluated every time the function is called. Before the function body is evaluated, the function call's *arguments* (after having been evaluated themselves) are assigned to the function's parameters.

Usually the function body will be a *block*, allowing for the declaration of local variables, as demonstrated in the second and third line. Inside of a function body, *local* can - and should - be applied both to local variables and function parameters. If they are not declared *local*, parameters, just like local variables, are local only with respect to their values, but not with respect to their properties!

```
(%i1)   properties(x);
(%o1)                              []
(%i2)   f(x):=block([a], local(a), a:1, declare (x,odd), x:a)$
(%i3)   properties(x);
(%o3)                              []
(%i4)   f(3);
(%o4)                              1
(%i5)   properties(x);
(%o5)                [database info, kind(x,odd)]
(%i6)   kill(all)$
(%i7)   f(x):=block([a], local(x,a), a:1, declare (x,odd), x:a)$
(%i8)   f(3);
(%o8)                              1
(%i9)   properties(x);
(%o9)                              []
```

If some parameter x_k is a quoted symbol (for *define*: after evaluation), the function defined does not evaluate the corresponding argument when it is called. Otherwise all arguments are evaluated.

```
(%i1)   f(x):=x^2;
(%o1)                                f(x) := x^2
(%i2)   a:b$  f(a);
(%o2)                                   b^2

(%i3)   f('x):=x^2;
(%o3)                               f('x) := x^2
(%i4)   a:b$  f(a);
(%o4)                                   a^2

(%i5)   define(f('x),x^2);
(%o5)                                f(x) := x^2
(%i6)   a:b$  f(a);
(%o6)                                   b^2

(%i7)   define(f('('x)),x^2);
(%o7)                               f('x) := x^2
(%i8)   a:b$  f(a);
(%o8)                                   a^2
```

f($x_1, \ldots, x_{n-1}$*, [*L*]) := expr*

If the last or only parameter x_n is a list of one element, the function defined accepts a variable number of arguments. Arguments are assigned one-to-one to parameters $x_1, \ldots, x_{(n-1)}$, and any further arguments, if present, are assigned to x_n as a list. In this case, arguments $x_1, \ldots, x_{(n-1)}$ are called *required arguments*, while all further arguments, if present, are called *optional arguments*.

All functions defined appear in the same global namespace. Thus, defining a function f within another function g does not automatically limit the scope of f to g. However, an additional statement *local (f)* inside of the *block* of g makes the definition of function f effective only within the block of function g.

functions default: *[]* [system variable]

functions is the list of ordinary Maxima functions having been defined by the user in the current session.

26.2.3 Array function, memoizing function

f[$x_1, \ldots, x_n$*] := expr*

define (f[$x_1, \ldots, x_n$*], expr)*
define (funmake (f, [$x_1, \ldots, x_n$*]), expr)*
define (arraymake (f, [$x_1, \ldots, x_n$*]), expr)*
define (ev (expr_1), expr_2)

f[$x_1, \ldots, x_n$*] := expr* defines an *array function*. Its function body is evaluated just once for each distinct value of its arguments, and that value is returned, without evaluating the function body, whenever the arguments have those values again. Such a function is known as a *memoizing function*.

26.2.4 Subscripted function

f[$x_1,\ldots,x_n$]($y_1,\ldots,y_m$) := expr

define (f[$x_1,\ldots,x_n$]($y_1,\ldots,y_n$), expr)

An *subscripted function* *f[$x_1,\ldots,x_n$]($y_1,\ldots,y_m$) := expr* is a special case of an array function *f[$x_1,\ldots,x_n$]* which returns a *lambda expression* with parameters $y_1,\ldots,y_m$. The function body of the subscripted function is evaluated only once for each distinct value of its parameters (subscripts) $x_1,\ldots,x_n$, and the corresponding lambda expression is that value returned. If the subscripted function is called not only with subscripts $x_1,\ldots,x_n$ in square brackets, but also with arguments $y_1,\ldots,y_n$ in parentheses, the corresponding lambda expression is evaluated and only its result is returned.

Note that a normal array function, see section 26.2.3, is also represented by Maxima with its parameters as subscripts, because they appear in square brackets. This is somewhat misleading, since they don't constitute real indices, but plain variables. Therefore we don't call such a function a subscripted function.

In the following example, the function body is a simple sequential compound statement, a list of expressions in parentheses, which are evaluated consecutively. Only the value of the last of them is returned.

```
(%i1)   f[n](x):= (print("Evaluating f for n=", n), diff (sin(x)^2, x, n));
```

$$(\%o1)\qquad f_n(x) := \left(\text{print ("Evaluating f for n=", n)}, \frac{d^n}{dx^n}\sin^2(x)\right)$$

```
(%i2)   f[1];
Evaluating f for n=1
(%o2)                    lambda([x], 2 cos(x) sin(x))
(%i3)   f[1];
(%o3)                    lambda([x], 2 cos(x) sin(x))
(%i3)   f[1](%pi/3);
```

$$(\%o3)\qquad \frac{\sqrt{3}}{2}$$

```
(%i4)   f[2];
Evaluating f for n=2
```

$$(\%o4)\qquad \text{lambda}\left([x], 2\cos^2(x) - 2\sin^2(x)\right)$$

```
(%i5)   f[2](%pi/3);
(%o5)                                -1
(%i6)   f[3](%pi/3);
Evaluating f for n=3
```

$$(\%o3)\qquad -2\sqrt{3}$$

26.2.5 Function call

funmake (f,[$v_1,\ldots,v_n$]) [function]

funmake (f,[$v_1,\ldots,v_n$]) evaluates its arguments and returns an expression f($v_1,\ldots,v_n$) which is a function call of function f with arguments $v_1,\ldots,v_n$ in parentheses. The return value is simplified, but not evaluated. So f is not called, even if it exists. To evaluate the return value, either ev() or quote-quote can be used, but only in a second statement.

f can be an ordinary function, a subscripted function or a macro function. In case f is an already defined array function, *funmake* will nevertheless return an expression with the arguments in parentheses. If an array function call with the arguments in square brackets is to be returned, use *arraymake* instead.

```
(%i1)   f(x,y):= y^2-x^2;
(%o1)                          f(x, y) := y^2 - x^2
(%i2)   funmake(f,[a+1,b+1]);
(%o2)                             f(a+1,b+1)
(%i3)   ev(%);
(%o3)                          (b + 1)^2 - (a + 1)^2

(%i4)   g[a](x) := (x - 1)^a;
(%o4)                          g_a(x) := (x - 1)^a
(%i5)   funmake (g[n],[b]);
(%o5)                      lambda ([x], (x - 1)^n)(b)
(%i6)   ev(%);
(%o6)                                (b - 1)^n
(%i7)   funmake ('g[n],[b]);
(%o7)                                g_n(b)
(%i8)   ev(%);
(%o8)                                (b - 1)^n

(%i9)   h(x) ::= (x - 1)/2;
(%o9)                             h(x) ::= (x - 1)/2

(%i10)  funmake(h,[u]);
(%o10)                               h(u)
(%i11)  ev(%);
(%o11)                               (u - 1)/2
```

funmake can be used in a function definition with *define* to evaluate the function name.

26.3 Operator (function)

Infix operator function definition, example tensor product:
infix("tp");
a tp b := transpose(vlist(a)).transpose(transpose(vlist(b)));
This can alternatively be defined by
"TP"(a,b) := transpose(vlist(a)).transpose(transpose(vlist(b)));

26.4 Lambda function, anonymous function

lambda ([$x_1, \ldots, x_m$], $expr_1, \ldots, expr_n$) [function]

This is called a *lambda function* or *anonymous function*. It defines and returns what is called a *lambda expression*, but does not evaluate it.

```
(%i1)   lambda([x],x+1);
(%o1)                          lambda([x],x+1)
```

A lambda expression can be evaluated like an ordinary function by calling it with *arguments* in parentheses corresponding to the lambda function's *parameters*.

```
(%i1)   lambda([x],x+1)(3);
(%o1)                                  4
```

When a lambda expression is evaluated, unbound local variables $x_1, \ldots, x_m$ are created. Then the arguments (after having been evaluated themselves) are assigned to the parameters. $expr_1, \ldots, expr_n$ are evaluated in turn, and the value of $expr_n$ is returned.

lambda ([$x_1, \ldots, x_m$, [L]], $expr_1, \ldots, expr_n$)

If the last or only parameter x_n is a list of one element, the function defined accepts a variable number of arguments. Arguments are assigned one-to-one to parameters $x_1, \ldots, x_{(n-1)}$, and any further arguments, if present, are assigned to x_n as a list.

lambda may appear within a *block* or another *lambda*; local variables are established each time another block or lambda expression is evaluated. Local variables appear to be global to any enclosed block or lambda. If a variable is not local, its value is the value most recently assigned in an enclosing block or lambda expression, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of *dynamic scope*.

A lambda function definition does not evaluate any of its arguments, neither the expressions nor the parameters given as a list in square brackets. Evaluation at definition time can, however, be forced individually with quote-quote. In this respect the lambda function definition behaves like the definition of an ordinary function with *:=*. The difference is, that a lambda function has no individual name; the lambda expression itself substitutes the function name.

```
(%i1)   x:a$
(%i2)   lambda([x],x+1);
(%o2)                          lambda([x],x+1)
(%i3)   lambda([x],x+1)(3);
(%o3)                                  4

(%i1)   x:a$
(%i2)   lambda([''x],x+1);
(%o2)                          lambda([a],x+1)
(%i3)   lambda([''x],x+1)(3);
(%o3)                                 a+1
```

```
(%i1)   x:a$
(%i2)   lambda([''x],''x+1);
(%o2)                          lambda([a],a+1)
(%i3)   lambda([''x],''x+1)(3);
(%o3)                                  4
```

A lambda expression can be assigned to a variable v. Evaluating this variable with arguments in parentheses corresponding to the parameters of the lambda expression looks like a function call of an ordinary function named v. However, *properties* shows that v is not a function.

```
(%i1)   v:lambda([x],x+1);
(%o1)                          lambda([x],x+1)
(%i2)   v(3);
(%o2)                                  4
(%i3)   properties(v);
(%o3)                               [value]
(%i4)   u(x):=x+1;
(%o4)                              u(x):=x+1
(%i5)   u(3);
(%o5)                                  4
(%i6)   properties(u);
(%o6)                              [function]
```

A lambda expression may appear in contexts in which a function name is expected. If a function definition is needed only for one specific context of calling this function, a lambda expression can efficiently substitute such a function definition and function call. It combines both steps, and the definition of a function name becomes unnecessary. In such a situation the definition of a lambda expression and its evaluation fall together.

```
(%i1)   f(x):=2*x$
(%i1)   map(f,[1,2,3,4,5]);
(%o1)                            [2,4,6,8,10]

(%i2)   map(lambda([x],2*x),[1,2,3,4,5]);
(%o2)                            [2,4,6,8,10]
```

26.5 Macro function

Chapter 27

Program Flow

Part VII

User interfaces, Package libraries

Chapter 28

User interfaces

28.1 Internal interfaces

28.1.1 Command line Maxima

28.1.2 wxMaxima

28.1.3 iMaxima

28.1.4 XMaxima

28.1.5 TeXmacs

28.1.6 GNUplot

28.2 External interfaces

28.2.1 Sage

28.2.2 Python, Jupyter, Java, etc.

Chapter 29

Package libraries

29.1 Internal share packages

29.2 External user packages

29.3 The Maxima exernal package manager

Part VIII

Maxima development

Chapter 30

MaximaL development

30.1 Introduction

This chapter describes from the practical viewpoint how larger programs to be written in MaximaL can be developed and how they are made available to be used for the practical work with Maxima. The next chapter will describe the same for developments done in Lisp.

In general, we will want to use MaximaL whenever possible for solving mathematical problems. This language is much easier to learn and to use than Lisp. MaximaL is Maxima's primary user interface. This language has some limitations, though. Since it is not lexically but dynamically scoped, there might be problems with name spaces for variables and functions, if large user packages are to be used. We will focus on these problems later and show what can be do to limit them as much as possible when programming the package and when using it.

Lisp has to be used whenever system features of Maxima shall be changed or amended. In addition, it might be considerable to use Lisp instead of MaximaL if scoping is an issue. Contrary to MaximaL, Lisp comprises strong concepts of lexical scoping.

It is also possible to call Lisp functions from MaximaL and to call MaximaL functions from Lisp. So we can combine both languages in order to find the most efficient programming solution for our problem.

Both MaximaL and Lisp programs can be compiled instead of just interpreted (as Maxima and Lisp usually do). This may be useful for reasons of speed. We will show when this is advisable and how it is done.

Let's start with MaximaL now. To summarize, there are two major issues. The first one is how to support programming packages in the Maxima language. There is no particular IDE available for MaximaL programming, so we have to invent our own development environment.

The second issue is how MaximaL packages we have written can be made available efficiently for our practical computational work with Maxima and possibly for other Maxima users, too.

The source code for MaximaL programs is generally stored in .mac files and can be loaded into a running Maxima session from the command line or from within other

programs. This is possible with all Maxima interfaces. Another option when working with wxMaxima is to store work in .wxm or .wxmx files. But these file types can only be read by this interface. However, a feature to export them to the .mac format is available in wxMaxima, too.

Due to its concept of input cells instead of the purely linear input and output stream of the usual Maxima REPL (read evaluate print loop) that all other interfaces provide, we feel that wxMaxima is most apt as a MaximaL development platform. However, a major drawback is that it suppresses most of MaximaL's debugging facilities and that it has almost no error handling.

30.2 Development with wxMaxima

30.2.1 File management

30.3 Error handling and debugging facilities in MaximaL

30.3.1 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

30.3.2 Tracing

30.3.3 Analyzing data structures

30.4 MaximaL compilaton

30.5 Providing and loading MaximaL packages

Chapter 31

Lisp Development

31.1 MaximaL and Lisp interaction

31.1.1 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.

31.1.2 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.

31.1.3 Lisp modes under MaximaL

31.1.3.1 Pure :lisp mode

31.1.3.2 Maxima-like Lisp mode

My mail to the list from 22.7.2017. Keywords: documentation, print, ?print

31.1.4 Executing Lisp code from within MaximaL

31.1.4.1 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))
```

31.1.5 Calling MaximaL function from within Lisp

31.2 Using the Emacs IDE

31.3 Debugging

31.3.1 Breaks

31.3.2 Tracing

31.3.3 Analyzing data structures

31.4 Lisp compilation

31.5 Providing and loading Lisp code

There are basically two ways how to incorporate changes and amendments to the Lisp code of Maxima. The easy way is to just load it into a Maxima session. Often this method will be sufficient, in particular if we want to load whole new packages written in Lisp. But this method has drawbacks when modifying system code. To overcome them, the new or modified Lisp code has to be committed with Git, and then Maxima has to be rebuilt from the modified source code base.

31.5.1 Loading Lisp code

31.5.1.1 Loading whole Lisp packages

31.5.1.2 Modifying and loading individual system functions or files

The user can, at the start or at any later point within a running Maxima session, modify the code of Maxima itself. This is done by reloading files containing Maxima system or application Lisp code, or even by reloading only individual functions from them. All function definitions, system variables, etc., of a reloaded file or only the individually reloaded functions will overwrite the existing system function definitions and variables of the same name. This is independent of whether the existing file or function was compiled or not. Depending on the Lisp used and on the setting of Lisp system variables, the system may issue a warning concerning the redefinition of each function or variable, but it will not decline to do so. From the moment on where it has been successfully loaded, the new function definition will be used whenever the function is called. So any Maxima system function can easily be changed by just reloading a modified version of its definition. 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. Of course, new functions can be added this way, too.

This method is so easy that most people will want to try it out and see whether it is sufficient for their needs.

The substitution or adding of function definitions can be automated by incorporating the reload procedure in the *maxima-init.lisp* or *maxima-init.mac* files to be executed

at Maxima startup time. Even after a new Maxima release, the procedure does not have to be changed. So in some kind, we can apply our changes on top of the latest Maxima release.

31.5.2 Committing Lisp code and rebuilding Maxima

The method described above, however, as nice as it might seem 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 issue in the meantime at precisely the same files or functions that we have changed ourselves. To prevent such conflicts, at a certain point the user will have no other choice but to use *Git* to manage his local repository, commit and merge his modifications with the ones from Sourceforge, or rebase them on top. This method will be described in detail in chapter 33.

Part IX

Developer’s environment

Chapter 32

Emacs-based Maxima Lisp IDE

It should 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 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 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.

32.1 Operating systems and shells

We are going to set up and use the Emacs-based Maxima Lisp IDE primarily 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*.

32.2 Maxima

As a basis we need to have Maxima installed. There are two basic options.

32.2.1 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.

Download the latest Maxima installer and install it in C:/Maxima/, disregarding the default. Copy shortcuts for wxMaxima, console Maxima and XMaxima to the desktop. Special icons for the latter two can be found in the directory tree.

The installer 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.

Use the *Configure default Lisp for Maxima* feature from the Windows program menu to set Lisp to SBCL.

32.2.2 Building Maxima from tarball or 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, see chapter 34. Alternatively, Maxima can be built for Windows under Cygwin, see section **??**.

32.3 External program editor

32.3.1 Notepad++

If we are not really familiar with the Emacs editor yet, it is worthwhile to use *Notepad++* in addition. See https://notepad-plus-plus.org/ for reference. It is widely used, supported by Git, and has parentheses highlighting which is most important for programming in Lisp and very useful for MaximaL, too. In addition, we will install a special highlighting profile for MaximaL.

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 open any of these file types later.

A *highlighting profile for Maxima*, which recognizes our amended functions, 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*. To install it from Notepad++, select Language/Select your language/Import. After restarting Notepad++, *Maxima* will appear in the language menu and automatically be applied to *.mac* files.

32.4 7zip

Install 7zip, because you will need to unzip .tar.gz files soon.

32.5 SBCL: Steel Bank Common Lisp

A considerable number of Lisp compilers is available. Maxima supports many of them. The Windows installer comes with SBCL and Clisp. Independently of this, we use SBCL for a number of reasons. It is fast, provides a wide range of facilities, usually creates no problems for Maxima and has become a kind of de facto standard for Common Lisp use. See the *SBCL User Manual* for reference. [SbclMan17]

SBCL already comes with the Maxima installer. In principle, this installation of SBCL can be used as *inferior Lisp* under Emacs, too. However, we can 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. We prefer the latter option.

32.5.1 Installation

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. 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 32.6.3.2.

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).

32.5.2 Setup

32.5.2.1 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")
```

32.5.2.2 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 B.

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.

32.5.2.3 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.

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

32.6 Emacs

32.6.1 Overview

Emacs is a Lisp based IDE and much more. The *Emacs Manual* provides an impressive description. [EmacsMan12]

32.6.1.1 Editor

It's not without reason that one generally defines

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 introduction to the editor and the basic Emacs 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. [EmacsTut]

32.6.1.2 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. Emacs supplies is with special debugging facilities. See the extensive *eLisp Manual* for details. [eLispMan13]

32.6.1.3 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 will use SBCL. Note that inferior Lisp is independent of the Lisp used by Maxima and of eLisp. All can be different.

The Emacs IDE can thus be used for any other Lisp development independent of Maxima.

32.6.1.4 Maxima under Emacs

There are various Maxima interfaces that work under Emacs. We use the Maxima console and *iMaxima* which provides output created with LateX.

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

32.6.1.5 Slime: Superior Interaction Mode for Emacs

Slime is an enhancement for Emacs. It provides much more elaborate debugging facilities and with *slime-connect*, see below, it allows for setting up a parallel session of MaximaL and Maxima Lisp. See the *Slime Manual* for details. [SlimeMan15]

32.6.2 Installation and update

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. 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.

Slime has to be installed separately. We will do this with the help of Quicklisp soon.

32.6.3 Setup

32.6.3.1 Set start directory

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

32.6.3.2 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 C.

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 [iMaximaHP17]

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

```
; *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 [EmacsMan12]
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.
)
```

32.6.3.3 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.

32.6.3.4 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)
```

32.6.3.5 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 chapter 34 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 in parallel with a Lisp session under Emacs.

32.7 Quicklisp

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

32.7.1 Installation

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 32.5.2.2, 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.

32.8 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 32.6.3.2, 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).

32.9 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.

32.9.1 Installation

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 here. 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.

32.10 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.

32.10.1 MikTeX

MikTeX provides the Latex environment needed for iMaxima. This is a very complicated system, and it is important to follow the installation instruction carefully.

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 your antivirus program complains, ignore it this time 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 the available updates proposed.

32.10.2 Ghostscript

Ghostscript is needed for iMaxima, too.

Install Ghostscript in the default directory C:/Program Files/gs. An overview about the software is to be found under C:/Program Files/gs/gs9.21/doc/Readme.htm.

32.10.3 TeXstudio, JabRef, etc.

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.

Note that the wxMaxima interface provides nice LateX output via the context menu.

Install TeXstudio in the default directory C:/Program Files (x86)/TeXstudio. Set biber to be the standard bibliography program.

JabRef is a nice program to maintain a larger bibliography. Personally, we prefer to edit the .bib file with Notepad++, however, and use JabRef only to display the result and do searches in it.

32.11 Linux and Linux-like environments

32.11.1 Cygwin

Install Cygwin in C:/Program Files/cygwin64.

32.11.2 MinGW

Install MinGW in C:/Program Files/MinGW.

32.11.3 Linux in VirtualBox under Windows

32.11.3.1 VirtualBox

32.11.3.2 Linux

Chapter 33

Repository management: Git and GitHub

33.1 Introduction

This chapter follows up on the discussion of section 31.5.

33.1.1 General intention

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 modify or enhance them. But we will not only make our own changes, we will also continuously update our local mirror by downloading all modifications done to the Maxima code base at Sourceforge. It is only with the help of Git that we will be able to *merge* (or, as we will see, *rebase*) our code modifications with the ones being done in parallel 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 publish the changes that we have done to the code and which are, as we saw, always based on the latest updates done at Sourceforge.

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

33.1.2 Git and our local repository

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.

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

A good introduction to Git is the book *ProGit* by Scott Chacon which is available as PDF for free. All the details can be found in the *Git Online Reference*. [ChProGit14] [GitRef17]

33.1.2.1 KDiff3

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

33.1.3 GitHub and our public repository

We can work with a local repository on our computer only. If in addition we want to make public our work or 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 work coming from the local repository.

Eventually we can also use our GitHub repository to communicate with the Maxima external packet manager system, if we want to make our packages directly accessible to Maxima users.

33.2 Installation and Setup

33.2.1 Git

33.2.1.1 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 the default settings. But for the default editor select Notepad++. 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 can use the CMD interface which resembles the Windows console. But we prefer Git bash which has the advantage of always displaying the branch we are on. In order to set our start directory to *D:/Maxima/Repos* do the following. Rightclick on the desktop shortcut. Select properties. Change *Execute in* to the above path. In *Destination* delete the option *-cd-to-home*.[1]

33.2.1.2 Installing KDiff3

Install the 64bit version of KDiff3 with all defaults and in the default location.

33.2.1.3 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 D, after having done some personal adjustments to it.

[1] RS only: When CMD is started, rightclick on the margin of the window and in properties set font size to 20. For Git bash, set options/text/font to Courier new, size 14.

Most important is to substitute 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 the double dashes before each option) Notepad++ should open and display the file *.gitconfig*.

There is a known problem with Git not handling UTF-8 characters correctly, for instance when displaying committ messages which contain German umlauts in the name of the committer, see stackexchange. We want to apply the proposed solution and create a Windows environment variable *LC_ALL* which we assign the value *C.UTF-8*. This will solve the problem permanently for both Git CMD and Git bash.

33.2.2 GitHub

33.2.2.1 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.

33.3 Cloning the Maxima repository

33.3.1 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.

33.3.2 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.

33.4 Updating our repository

33.4.1 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.

33.4.2 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 D. 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
```

33.4.3 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.

33.5 Working with the Repository

33.5.1 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).

33.5.2 Basic operations

We get a list of all our local branches with

```
git br
```

To see the status of the current branch, type

```
git st
```

We can create a new branch from an existing one and switch to it by doing

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

In order to obtain a compact log output of the last n commits we can type

```
git logol –n
```

33.5.3 Committing, merging and rebasing our changes

Chapter 34

Building Maxima under Windows

34.1 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.

34.2 Lisp-only build

34.2.1 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.

34.2.2 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.

34.3 Building Maxima with Cygwin

Part X

Maxima’s file structure, build system

Chapter 35

Maxima’s file structure: repository, tarball, installer

Chapter 36

Maxima’s build system

Part XI

Lisp program structure (model), control and data flow

Chapter 37

Lisp program structure

37.1 Supported Lisps

Part XII

Appendices

Appendix A

Glossary

A.1 MaximaL terminology

In this section we define the terminology needed to describe MaximaL. Sometimes this terminology is semantically close to the terminology used in Lisp, which will be given in the next section.

Argument

If a *function* f has been defined with *parameters*, a function call of f has to be supplied with corresponding *arguments*. When f is evaluated, arguments are assigned to their corresponding parameters. For the distinction of *required* and *optional arguments*, see section 26.2.2.

Array

An *array* is a data structure ...

Assignment

Binding a value to a variable. This is done explicitly with the *assignment operator*. The value can be a number, but also a symbol or an expression. In an *indirect assignment*, done with the *indirect assignment operator*, not a symbol is bound with a value, but the value of the symbol, which must again be a symbol, is bound.

Atom

An atom is an *expression* consisting of only one element (symbol or number).

Binding

A binding ...

Constant

There are *numerical constants* and *symbolical constants*. A number is a numerical constant. Maxima also recognizes certain symbolical constants such as *%pi*, *%e* and *%i* which stand for π, Euler's number e and the imaginary unit i, respectively. For Maxima's naming conventions of *system constants* see section 3.4.2.2. Of course the user may assign his own symbolical constants.

Equation

An *equation* is an *expression* comprising an equal sign =, one of the *identity operators*, as its major operator. An *unequation* is an expression with the *unequation operator* # as its major operator.

Expression

Any meaningful combination of operators, symbols and numbers is called an *expression*. An expression can be a mathematical expression, but also a function call, a function definition or any other statement. An expression can have *subexpressions* and is build up of *elements*. An *atom* or *atomic expression* contains only one element. A *complete subexpression* ... See *subst (eq_1, expr)* for an example.

See also *lambda expression*.

Function

A *function* is a special *compound statement* which is assigned a *(function) name*, has *parameters* and in addition can have *local variables*. Maxima comprises a large number of *system functions*, as for instance *diff* and *integrate*. Furthermore, the user can define his own *user functions*. A special operator, the *function definition operator* :=, is used for this purpose. On the left side, the function name and its parameters are specified, and on the right side, the *function body*. Alternatively, function *define* can be used.

On *calling* a function, *arguments* [1]are passed to it which are assigned to the function's parameters at evaluation time. The result of the function's subsequent computations, i.e. the evaluation of the function, is *returned*. We speak of the *return value* of a *function call*. A function call can be incorporated in an expression just like a variable. An *ordinary function* is evaluated on every call, see section 26.2.2.

An *array function* stores the function value the first time it is called with a given argument, and returns the stored value, without recomputing it, when that same argument is given. Such a function is known as a *memoizing function*, see section 26.2.3.

A *subscripted function* is a special kind of array function which returns a *lambda expression*. It can be used to create a whole family of functions with a single definition, see section 26.2.4.

In addition there are functions without name, so-called *lambda functions* or *anonymous functions*, which can be defined and called at the same time. Their return value is called a *lambda expression*. See section 26.4.

A *macro function* is similar to an ordinary function, but has a slightly different behavior. It does not evaluate its arguments and it returns what is known as a *macro expansion*. This means, the *return value* is itself a Maxima *statement* which is immediately evaluated. Macros are defined with the *macro function definition operator* ::=.

[1]Instead of *parameter* and *argument*, the terminology *formal argument* and *actual argument* is used in the Maxima Manual.

Lambda expression

The return value of a *lambda function* is called a *lambda expression*. See section 26.4.

Macro expansion

Macro expansion is part of the mechanism of a *macro function*.

Operator

A Maxima *operator* can be view in a way similar to a mathematical operator. The arithmetic operators +, -, *, /, for example, are employed in an infix notation just as in mathematics.

The equal sign =, the assignment : or the function definition := are examples of other Maxima *system operators*.

Maxima even allows the user to define his own operators, be they used in *prefix*, *infix*, *postfix*, *matchfix* or other notations.

Parameter

A *parameter* is a special local variable defined for a function, which is assigned the value of a corresponding *argument* at function call.

Pattern matching

Pattern matching ...

Property

A *MaximaL property* ... A *Lisp property* ...

Quote-quote ' ' is twice the quote character, not the *doubel-quote "* character.

Rule

A rule ...

Scope

We distinguish *dynamic scope* from *lexical scope*...

Symbol, identifier

Maxima allows for symbolical computation. Its basic element is the *symbol*, also called *identifier*. A symbol is a name that stands for something else. It can stand for a constant (as we have seen already), a variable, an operator, an expression, a function and so on.

Statement

An input expression terminated by ; or $ which is to be evaluated is called a *statement*. In Lisp it would be called a *form*.

If a number of statements are combined, e.g. as a list enclosed in parentheses and

separated by commas, called a *sequential*, we speak of a *compound statement*. The statements forming a compound statement are called its *sub-statements*. *Block* and *function* are other special forms of a compound statement. A block is a compound statement which can have local variables, a function is assigned a name and can have parameters, see chapter 26.

Value

A *symbol* (i.e. a variable, a constant, a function, a parameter, etc.) can be unbound; then it has not been assigned a value. When a value has been assigned to the symbol, it is bound. Binding a value to a symbol is called *assignment*. Retrieving the value of a symbol is called *referencing* or *evaluation*.

The *return value* is what a *function* returns when it is called and evaluated.

Variable

A *variable* has a name (which is represented by a *symbol*) and possibly a *value*. *Assignment* of a value to a variable is called *binding*. We say: the variable is bound to a value. When a variable has been bound, it is *referencing* this particular value. *Evaluation* in the strict sense means *dereferencing*, which is: obtaining from a variable the value which was bound to it previously.

In general, Maxima does not require a variable to be defined explicitly by the user before using it. In particular, Maxima does not require a variable to have a specific type (of value). Just as when doing mathematics on a sheet of paper, we can start using a variable at any time. It will be defined (allocated) at use time by Maxima automatically. We can start using a variable without binding it to a value. Maxima recognizes the symbol, but it remains *unbound*. But we can also bind it at any time, even right at the beginning of its use. The type of value of a specific variable may change at any time, whenever the value itself changes.

The value of a variable does not need to be a numerical constant. It can be another variable or any combination of variables and operators, that is, an *expression*. It can even be much more than this. The variety of types (of values) of a variable is so broad that in Lisp and in Maxima we generally use the term *symbol* to denote not only the name of variable, but the variable as a whole.

One of the specific features of Lisp is that a symbol not only can have a value, but also *properties*. A Maxima symbol can have properties, too, as we will see later. It can even have two types of properties, Lisp properties and Maxima properties.

There are *user variables*, which the user defines, and *system variables*. System variables which can be set by the user to select certain options of operation are called *option variables*. With respect to the name space where the variable appears we distinguish between *global variables* and *local variables*.

A.2 Lisp terminology

Form

A Lisp form ...

Appendix B

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 32.5.2.2 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 C

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 32.6.3.2 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 D

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 33.2.1.3 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).

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

[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.

[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.

[FatemanRM89] Richard J. Fateman. “A review of Macsyma.” In: *IEEE Transactions on Knowledge and Data Engineering* 1 (1 1989), pp. 133–145. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.6365&rank=1.

[FredmCME14] Tom Fredman. *Computer Mathematics for the Engineer: Efficient Computation and Symbolic Manipulation*. 2014, p. 147. URL: http://users.abo.fi/tfredman/comp_math_2014.pdf.

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

[GosperHP17] R. William Gosper. *Homepage vita*. [Online; Stand 18. Dezember 2017]. 2017. URL: http://gosper.org/bill.html.

[HaagGM11] Wilhelm Haager. *Grafiken mit Maxima*. 2011, p. 35. URL: http://www.austromath.at/daten/maxima/zusatz/Grafiken_mit_Maxima.pdf.

[HaagCAM14] Wilhelm Haager. *Computeralgebra mit Maxima: Grundlagen der Anwendung und Programmierung*. Hanser, München, 2014, p. 317.

[HaagCEM17] Wilhelm Haager. *Control Engineering with Maxima*. 2017, p. 36. URL: http://www.austromath.at/daten/maxima/zusatz/Control_Engineering_with_Maxima.pdf.

[HammMTC13] Michael R. Hammock and J. Wilson Mixon. *Microeconomic Theory and Computation. Applying the Maxima Open-Source Computer Algebra System*. 1. Aufl. Springer, New York, 2013, p. 385.

[HanMC1-15] Zachary Hannan. *wxMaxima for Calculus I*. 2015, p. 158. URL: https://wxmaximafor.files.wordpress.com/2015/06/wxmaxima_for_calculus_i_cq.pdf.

[HanMC2-15] Zachary Hannan. *wxMaxima for Calculus II*. 2015, p. 176. URL: https://wxmaximafor.files.wordpress.com/2015/06/wxmaxima_for_calculus_ii_cq.pdf.

[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.

[LeydoldME11] Josef Leydold and Martin Petry. *Introduction to Maxima for Economics*. 2011, p. 119. URL: http://statmath.wu.ac.at/~leydold/maxima/MaximaSkript.pdf.

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

[MaxiManE17] 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.

[SbclMan17] SBCL. *SBCL User Manual 1.4.2 engl.* 2017. URL: http://www.sbcl.org/manual/sbcl.pdf.

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

[SouzaMaxB04] Paulo Ney de Souza. *The Maxima Book*. 2004, p. 155. URL: http://maxima.sourceforge.net/docs/maximabook/maximabook-19-Sept-2004.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.

[TimbCMM16] Todd Keene Timberlake and J. Wilson Mixon. *Classical Mechanics with Maxima*. 1. Aufl. Springer, New York, 2016, p. 258.

[UrrozMSE12] Gilberto Urroz. *Maxima: Science and Engineering Applications*. self-published, 2012, p. 438.

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

Index