CAS Maxima Workbook

Roland Salz

April 23, 2022

Vers. 1.0.0

This project is work in progress.
Comments and suggestions for improvement are welcome.

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

In some cases the objective is clear — and the results are surprising.

Richard J. Fateman

Preface

Maxima was developed from 1968-1982 at MIT (Massachusetts Institute of Technology) as the first comprehensive computer algebra system (CAS). Allowing not only for numerical, but also symbolical computation it was used by the leading US universities, by US Government institutions like the DOE, by the US Navy, or NASA. Having been enhanced and improved ever since, now Maxima is free (GPL) software and counts about 150.000 users worldwide. It is employed in education and research by mathematicians, physicists, engineers, and economists, coping with the major commercial CAS' of today. Since 2000 the software is maintained by an energetic group of volunteers called the *Maxima team*. The author wishes to thank its kind and helpful members, in particular Dr. Robert Dodier, who is in charge of the project, Gunter Königsmann, in charge of the frontend wxMaxima, as well as Prof. Richard J. Fateman and Dr. Stavros Macrakis, who participated in the original MIT project and have been contributing to Maxima ever since, for almost half a century now.

The intention of the *Maxima Workbook* is to provide a new documentation of the CAS 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 software 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 the [**MosesMPH**]

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*, standing for *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, in particular the simplifier and the pattern matcher. 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. [**MartFate**] [**FatemThe**]

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 [**wikMacsy**]

Golden (simplifier, language, system), R. W. Gosper (definite summation, special functions, simplification, number theory), Carl Hoffman (general simplifier, macros, non-commutative simplifier, ports to Multics and LispM, system, visual equation editor), Charles Karney (plotting), John Kulp, Ed Lafferty (ODE solution, special functions), Stavros Macrakis (real/imaginary parts, compiler, system), Richard Pavelle (indicial tensor calculus, general relativity package, ordinary and partial differential equations), David A. Spear (Gröbner), Barry Trager (algebraic integration, factoring, Gröbner), Paul Wang (polynomial factorization and GCD, complex numbers, limits, definite integration, Fortran and LaTeX code generation), David Y. Y. Yun (polynomial GCDs), Gail Zacharias (Gröbner) and Rich Zippel (power series, polynomial factorization, number theory, combinatorics).

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. [**ColeSMP**]

In May, 1993 Prof. Fateman gave a guest lecture at Stanford's CS50 introductory course in computer science held by Nancy Blachman. It contains a review of the Mathematica system and its underlying language as of 1993 including some illustrations of pitfalls in the design of such systems and Mathematica in particular, as well as comments on the use of Mathematica for introductory programming and system building. This lecture is now on YouTube. [**ytFatemM**]

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 [**CharMap**]

system memory size, so that it could operate on smaller and cheaper hardware and eventually on personal computers.

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. [**GosperHP**]

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 Schelter 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 Dr. Robert Dodier from Portland, Oregon, today maintains Maxima. Among the Lisp developers are Dr. Raymond Toy, Barton Willis (Prof. of Mathematics, 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, Dr. Viktor T. Toth (Canada) is in charge of new releases and maintains the tensor packages. Jaime Villate (Prof. of Physics, 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, Dr. 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 (Prof.

of Physics, California State University, Long Beach) has spent years writing a highly sophisticated and free Maxima tutorial for applications in Physics, called *Maxima by example*. Richard J. Fateman (Prof. of Computer Science, University of California at Berkeley) and Dr. Stavros Macrakis (Cambridge, Ma.), 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: [**FatemanRM**]

"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, here abbreviated MaximaL, 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 have 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 completely.

This obvious lack of documentation motivated us to start the Maxima Workbook project. But before diving into it, let us get an overview about exactly what sources and what extend of information we have available already. As a first reference, the user should consult the bibliography contained in Maxima's official documentation page.

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. [**MaxiManE**]

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. [**MaxiManD**]

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. [**SouzaMaxB**]

2.3.2 Tutorials

The tutorials presented first are those not focused on a specific field of application. The order is according to their date of publication.

2.3.2.1 Michel Talon: Rules and Patterns in Maxima, 2019

This tutorial of some 20 pages facilitates access to understanding how to use Maxima's pattern matching facilities, which remains difficult from reading the section from Maxima's manual alone. It is particularly useful for someone who furthermore wants to understand how the pattern matcher works internally. Hints to example applications from mathematics and physics are given at the end. Altogether, a very substantial work written by someone deeply interested in Maxima. [**TalonRP**]

2.3.2.2 Jorge Alberto Calvo: Scientific Programming, 2018

Scientific Programming. Numeric, Symbolic, and Graphical Computing with Maxima uses Maxima to illustrate some methods of numeric and symbolic computation for application in mathematically oriented sciences, and at the same time the general use of computer programming. [**CalvoSP**]

2.3.2.3 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**] [**HanMC2**]

2.3.2.4 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. [**HaagCAM**]

2.3.2.5 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. [**HaagGM**]

2.3.2.6 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. [**StewenMT**]

2.3.3 Mathematics

2.3.3.1 G. Jay Kerns: Multivariable Calculus with Maxima, 2009

Originating from material the author compiled for a university course in Calculus, this document of some 50 pages grew up to become a real introduction to Maxima. A concise and very illustrative work for the undergraduate level. [**KernsMVC**]

2.3.4 Physics

2.3.4.1 Edwin L. (Ted) Woollett: "Maxima by Example", 2018, and "Computational Physics with Maxima or R"

This tutorial by Edwin L. (Ted) Woollett, Prof. Emeritus of Physics and Astronomy at California State University (CSULB), is free online-material and certainly one of the best and most inspiring tutorials around, and Ted's work is still continuing! Here we find valuable advice and many examples from the viewpoint of a computational physicist, and some impressive, highly sophisticated worked-out applications. [**WoolMbE**]

2.3.4.2 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. [**TimbCMM**]

2.3.4.3 Viktor Toth: Tensor Manipulation in GPL Maxima

Written by Viktor T. Toth, theoretical physicist, member of the Maxima team, and responsible for maintaining the tensor packages, this highly recommended paper published in arxiv gives a comprehensive description of the present abilities of Maxima's tensor packages for applications in physics, in particular general relativity. [**TothTenM**]

2.3.5 Engineering

2.3.5.1 Andreas Baumgart: Toolbox Technische Mechanik, 2018

Andreas Baumgart from Hochschule für Angewandte Wissenschaften, Hamburg, has created an extensive and very well designed internet site for illustrating how problems in engineering mechanics can be solved with Maxima and Matlab. The site is in German. [**BaumgTM**]

2.3.5.2 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. [**HaagCEM**]

2.3.5.3 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. [**FredmCME**]

2.3.5.4 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. [**UrrozMSE**]

2.3.6 Economics

2.3.6.1 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. [**HammMTC**]

2.3.6.2 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. [**LeydoldME**]

2.4 Articles and Papers

A very comprehensive bibliography can be found in [**SouzaMaxB**].

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 pub-

lished 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. [**FredmCME**]

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 days 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*, the input line was executed, creating the output to be display after a simple line-feed. The REPL makes very intelligent use of this initial situation, and many even very experienced CAS users still work with no other interface today. In Maxima this interface is called *command line Maxima*, sect. 33.1.1, or simply the *console*.

Nowadays, however, most people are used to employ the full screen of the computer, and the mouse has become even more important as an input medium than the keyboard. CAS interfaces have been developed that take this evolution into account. *wxMaxima*, sect. 33.1.2, 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 Basic operation

3.2.1 Executing an input line or cell

In *command line Maxima*, sect. 33.1.1, use *enter* to execute an input line. In *wxMaxima*, sect. 33.1.2, use *shift+enter*.

3.3 Basic notation

3.3.1 Output description and numbering conventions

In this manual we use certain conventions to facilitate the description of Maxima's output and the interactive dialogue with Maxima. Note, however, that we never change the input required by Maxima.

We represent output formulas always in the usual mathematical 2D notation. In order to make output better readable, we usually omit the %-character in front of Maxima system constants such as %e, %i, %pi, etc. We write Re and Im instead of *realpart* and *imagpart*. And as wxMaxima does, we write $\overline{z}$ instead of conjugate(z).

Input and output tags, see section 4.1.1, are sometimes represented as they would be in wxMaxima with its cell-based structure. Other frontends therefore might number input and output differently.

3.3.2 Syntax description operators

In order to facilitate describing the MaximaL syntax, we use a number of *syntax description operators*. These do not form part of MaximaL itself and thus cannot been entered in Maxima by the user. In order to distinguish them form the proper MaximaL syntax, throughout this manual they have green color and a slightly bigger size.

$\langle \ldots \rangle$ [syntax description operator]

Optional elements, e.g. optional function parameters, are enclosed in angle brackets. Example: see *genmatrix*.

$(\ldots \mid \ldots)$ [syntax description operator]

Alternatives are separated by $\mid$ and enclosed in $()$. More than two alternatives can be represented by repeating the $\mid$ operator inside of the green parentheses. Exactly one of the alternative has to be selected. Example: see *to_poly_solve*.

3.3.3 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.

3.3.4 Assignment operators

3.3.4.1 Basic :

: [infix operator]

This is the basic *assignment operator*. When the lhs (lhs) is a simple variable (not subscripted), : evaluates its rhs (rhs), unless quoted, and associates that value with the symbol on the lhs.

```
(%i1)   a:3;
(%o1)                            3
(%i2)   b:a;       /* The rhs is evaluated before assigning. */
(%o2)                            3
(%i3)   c:'a;                  /* The rhs is not evaluated. */
```

```
(%o3)                                a
(%i4)   ev(c);                                       /* Evaluation of c. */
(%o4)                                3

(%i1)   b:a;                  /* The rhs evaluates to itself. */
(%o1)                                a
(%i2)   a:c$ c:3;
(%o3)                                3
(%i4)   b;                                  /* Simple evaluation of b. */
(%o4)                                a
(%i5)   ev(b);                              /* Double evaluation of b. */
(%o5)                                c
(%i6)   ev(ev(b));                          /* Triple evaluation of b. */
(%o6)                                3
```

Chain constructions are allowed; in this case all positions but the right-most one are considered lhs.

```
(%i1)   x : y : 3;
(%o1)                                3
(%i2)   x;
(%o2)                                3
(%i3)   y;
(%o3)                                3
```

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

When the lhs is a subscripted element of an undeclared Maxima array, the rhs is assigned to that element, if it already exists, or a new element is allocated, if it does not already exist.

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

```
(%i1)   [a, b, c] : [4, 7, 10];
(%o1)                          [4, 7, 10]
(%i2)   a;
(%o2)                           4
```

3.3.4.2 Indirect ::

:: [infix operator]

This is the *indirect assignment operator*. :: is the same as :, except that :: evaluates its lhs as well as its rhs. Thus, the evaluated rhs is assigned not to the symbol on the lhs, but to the *value* of the variable on the lhs, 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.

3.3.5 Miscellaneous operators

3.3.5.1 Comment

/ ... */* [matchfix operator]

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

3.3.5.2 Documentation reference

? [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 MaximaL 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 MaximaL 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 capital letters to separate words within a symbol, e.g. *ExtractEquations*.

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);
                                  e^log(x)
(%o2)
(%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.

3.4.3 Correpondence of MaximaL and Lisp identifiers

MaximaL	Lisp
var	($VAR, $var, $Var) → $VAR; \|$VAR|
VAR	|$var|
Var	|$Var|
?var	(VAR, var, Var) → VAR
?* *var*\-1*	*VAR-1*

Table 3.1 – Correspondence of MaximaL and Lisp identifiers

MaximaL and Lisp symbols are distinguished by a naming convention. A Lisp symbol which begins with a dollar sign $ corresponds to a MaximaL symbol without the dollar sign. For example, the MaximaL symbol *foo* corresponds to the Lisp symbol *$FOO*. Lisp functions and variables which are to be visible in Maxima as functions and variables with ordinary names (no special punctuation) must have Lisp names beginning with the dollar sign $.

On the other hand, a MaximaL symbol which begins with a question mark ? corresponds to a Lisp symbol without the question mark. For example, the MaximaL 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 the Maxima function *describe("foo")* which can also be written as *? foo*.

Hyphen -, asterisk *, or other special characters in Lisp symbols must be escaped by backslash \ where they appear in MaximaL code. For example, the Lisp identifier *foo-bar* is written ?* *foo*\-*bar** in MaximaL.

While Maxima is case-sensitive, distinguishing between lowercase and uppercase letters in identifiers, Lisp does not make this distinction. $foo, $FOO and $Foo are all converted by the Lisp reader by default to the Lisp symbol $FOO.
This discrepancy requires some rules governing the translation of names between Lisp and Maxima.

1. A Lisp identifier not enclosed in vertical bars || corresponds to a Maxima identifier in lowercase. Whether a Lisp identifier is uppercase, lowercase, or mixed case, is ignored, e.g., Lisp $foo, $FOO, and $Foo all correspond to Maxima foo. This is because $foo, $FOO and $Foo are converted by the Lisp reader to the Lisp symbol $FOO, since Lisp is not case-sensitive.

2. A Lisp identifier enclosed in vertical bars and

2.1. which is all uppercase or all lowercase corresponds to a Maxima identifier with case reversed. That is, uppercase is changed to lowercase and lowercase

to uppercase. E.g., Lisp |$FOO| and |$foo| correspond to Maxima foo and FOO, respectively.

2.2. which is mixed uppercase and lowercase corresponds to a Maxima identifier with the same case. E.g., Lisp |$Foo| corresponds to Maxima Foo.

Chapter 4

Using the Maxima REPL at the interactive prompt

4.1 Input and output

4.1.1 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* or *ctrl-enter* starts evaluation. When an input expression is an assignment, the corresponding output expression displays no numbered output tag, but instead the symbol to the left of the assignment in parentheses. If the input expression is only a symbol, the normal output tag is displayed.

```
(%i1)   temp:-30.5;
(temp)                              -30.5
(%i2)   temp;
(%o2)                               -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.

4.1.2 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.

4.1.3 Special characters

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

4.2 Input

4.2.0.1 One-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.

4.2.1 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.

4.2.2 System variables for backward references

_ (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 not 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.

4.2.3 General option variables

4.3 Output

4.3.0.1 One- and two-dimensional form

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.

4.3.0.2 System variables for backward references

% [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 $(s_1, \ldots, s_n)$, block, or lambda, 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.

4.3.1 Functions for output

print $(p_1, \ldots, p_n)$ [function]
print0 $(p_1, \ldots, p_n)$ [function of *rs_print0*]

4.3.2 General option variables

powerdisp default: *false* [option variable]

When *powerdisp* is true, an expression is displayed in reverse canonical order, see sect. 9.3.

verbose default: *false* [option variable]

This global variable controls the amount of output printed by various function, e.g. *powerseries*.

4.3.3 Variables generated by Maxima

In certain situations Maxima functions may generate there own new variables.

General variables are composed of a small g followed by a number, starting with $g1, g2, \ldots$ Summation indeces beginn with a small i instead and are numbered independently of the g-variables.

For instance, each time *powerseries* returns a power series expansion, it generates a new summation index, starting with $i1, i2, \ldots$

4.3.4 Pretty print for wxMaxima

Package *rs_pretty_print.mac* provides functions for pretty output. When placed at the end of an input form, they will add a comment and a noun form of the input at the beginning of Maxima's return value. These functions are particularly useful when employed within wxMaxima for evaluating large cells. With the additional information provided by functions *Pr* and *Pr0*, the output can be read fluently without constantly having to refer back to the input.

This package uses function *print0* instead of *print*, and thus it requires package *rs_print0.lisp*. This also means that between the parameters of the leading comment, blanks have to be inserted manually.

expr$ Pr(⟨"text"⟩)$ [function of *rs_pretty_print*]
expr$ Pr0(⟨"text"⟩)$ [function of *rs_pretty_print*]
expr$ Pr00(⟨"text"⟩)$ [function of *rs_pretty_print*]

Function *Pr* is used in the following way. Terminate the input expression with $. Then continue on the same line with the function call of *Pr*, also terminated with $. If a parameter string "text" is supplied, Maxima's output will be preceeded by "text" as a comment to what follows, terminated with a colon. Then, if the return value is different from the input, a noun form of the input will preceed Maxima's return value, separated by either = or <=>, depending on whether the expression is an equation or not. In case of the input being an assignment, the variable assigned to will preceed the assigned value (again split into a noun form and the evaluated form, if different), separated by :=.

Mathematical expressions evaluated by Maxima can be included in the leading comment. The comment can comprise a variable number of parameters (including zero), separated by commas.

Pr0 is the same as *Pr*, but the noun form is not displayed. *Pr0* is useful, when a number of consecutive transformations of an expression is performed and the leading comment is to replace the information given by the noun form of a step. *Pr00* is the same as *Pr0*, but the equal or equivalence sign is omitted as well.

```
(%i1)  bg[x]: x[t]=v[ox]*t$ Pr("Bewegungsgleichung in x-Richtung")$
(%i2)  bg[z]: z[t]=-g*t^2/2+v[oz]*t+h$ Pr("Bew.gl. in z.Richtung")$
(%i3)  eliminate([bg[x],bg[z]],[t])[1]$ Pr00("t eliminieren")$
(%i4)  expand(solve(%th(2),z[t]))[1]$ Pr00("Auflösen nach z")$
(%i5)  z[x]: ev(rhs(%th(2)),x[t]=x)$ Pr00("Wurfparabel-Funktion")$
```

(%o1) Bewegungsgleichung in x-Richtung: $bg_x := x_t = v_{ox}\,t$

(%o2) Bew.gl. in z-Richtung: $bg_z := z_t = -(g*t^2)/2 + v_{oz}*t + h$

(%o3) t eliminieren: $v_{ox}^2\,(2z_t - 2h) + g\,x_t^2 - 2v_{ox}\,v_{oz}\,x_t$

(%o4) Auflösen nach z: $$z_t = -\frac{g\,x_t^2}{2v_{ox}^2} + \frac{v_{oz}\,x_t}{v_{ox}} + h$$

(%o5) Wurfparabel-Funktion: $$z_x := -\frac{g\,x^2}{2v_{ox}^2} + \frac{v_{oz}\,x}{v_{ox}} + h$$

Note that when using one of the pretty print functions, %th(2) has to be used instead of % when refering to the last output expression. The next example shows that we can even display the most challenging tensor notations.

```
(%i1)  goijρ:diff(goij,ρ)$ Pr00('g[",ρ"]^ij)$
(%i2)  zeromatrix(3,3)$ Pr00('g[",φ"]^ij," = ",'g[",z"]^ij," ",'g[k]^".l")
   $
```

(%o1) $$g^{ij}_{,\rho}: \quad goij\rho := \begin{pmatrix} 0 & 0 & 0 \\ 0 & -\frac{2}{\rho^3} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

(%o2) $$g^{ij}_{,\varphi} = g^{ij}_{,z} = g^{.l}_{k}: \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Chapter 5

Graphical representation of functions

5.1 Introduction

There are two different Maxima interfaces for plotting, both being based on *GNUplot*: *plot* and *draw*. Both interfaces are able to deliver 2D and 3D representations. Although they cover the same kind of problems, the two interfaces are substantially different with respect to the structure of their commands, so we treat them separately. *Plot* is the older interface, offering less functionality, but being easier at the same time, so we describe it first.

Both *plot* and *draw* come with additional special functions for use with wxMaxima only. These functions start with the prefix *wx* (e.g. *wxplot2d*, *wxplot3d*). They are the same as the ordinary functions *plot2d* and *plot3d*, with the only difference that they do not open a separate window to display the plot, but instead integrate it into the output of the *.wxm* file or into the *.wxmx* file.

5.2 Plot

5.2.1 General

5.2.1.1 Options, (user) standard options, and system standard options

The user can customize any of the plot functions by setting plot options. This can be done individually for each function call. It is also possible to set *(user) standard options* which then apply to any function call unless they are overwritten by it. Certain individual options cannot be set as standard, see details in the description of plot options.

Certain options are set standard by the system already, e.g. the order of colors in a multiple plot, if no colors are specified by the user. They can be viewed with the following function.

set_plot_option (⟨$option_1, \ldots, option_n$⟩) [function]
get_plot_option (*name* ⟨, *index*⟩) [function]
remove_plot_option (name) [function]

Setting *(user) standard options* is done with function *set_plot_option*. Each option is a list in square brackets, as described below. *set_plot_option* returns a list not only of the standard options currently set by the user, but also of all *system standard options*. Giving an empty set of parentheses to this function will only return the currently set (user and system) standard options without adding any to them.

get_plot_option returns as a list in square brackets the current standard setting of the option *name*. If the second argument *index* is present, only the *index*th element of this list will be returned (the first element is the option name).

remove_plot_option removes from the list of standard options the option *name*. Note that this function requires exactly one argument; multiple removals are not possible.

5.2.1.2 Options for both 2D and 3D plots

All options (this also holds for the options specific to either 2D or 3D as described in sections 5.2.2.4 and 5.2.3.3) consist of a list (in square brackets) starting with one of the keywords in this section, followed by one or more values. (This layout is comparable to a function name and its arguments.) The options that accept among their possible values *true* or *false*, can also be set to *true* by simply writing their names. For instance, typing *logx* as an option is equivalent to writing *[logx, true]*.

[box, true | false] default: *true* [plot option]

If set to *true*, a bounding box will be drawn around the plot; if set to *false*, no box will be drawn.

[color, color$_1$, ..., color$_n$] [plot option]

In 2d plots this option defines the color (or colors) for the various curves. In plot3d, it defines the colors used for the mesh lines of the surfaces, when no palette is being used. If there are more curves or surfaces than colors, the colors will be repeated in sequence. The valid colors are red, green, blue, magenta, cyan, yellow, orange, violet, brown, gray, black, white, or a string starting with the character # and followed by six hexadecimal digits: two for the red component, two for green component and two for the blue component. If the name of a given color is unknown color, black will be used instead.

[legend, false | string$_1$, ..., string$_n$] [plot option]

Specifies the labels for the plots when various plots are shown. If there are more plots than the number of labels given, they will be repeated. If given the value *false*, no legends will be shown. By default, the names of the expressions or functions will be used, or the words *discrete$_1$, ..., discrete$_n$* for discrete sets of points.

[logx, true | false] default: *false* [plot option]
[logy, true | false] default: *false* [plot option]

Makes the horizontal or vertical axes to be scaled logarithmically.

[plot_format, format] default: *gnuplot | gnuplot_pipes* [plot option]

Specifies the format for the plot. In Windows the default is *gnuplot*, in all other systems it is *gnuplot_pipes*. The formats *xmaxima* or *openmath* will cause the plot to be displayed in an xMaxima window.

[plot_realpart, true | false] default: *false* [plot option]

If set to *true*, the functions to be plotted will be considered as complex functions whose real part should be plotted; this is equivalent to plotting realpart(function). If set to *false*, nothing will be plotted when the function does not give a purely real value. For instance, when x is negative, log(x) gives a complex value, with the real value equal to log(abs(x)); if *plot_realpart* were true, log(-5) would be plotted as log(5), while nothing would be plotted if *plot_realpart* were *false*.

```
(%i1)   plot2d(realpart(log(x)),[x,-2,2],[y,-4,2]);
(%i2)   plot2d(log(x),[x,-2,2],[y,-4,2],plot_realpart);
```

Both plots will return exactly the same graph.

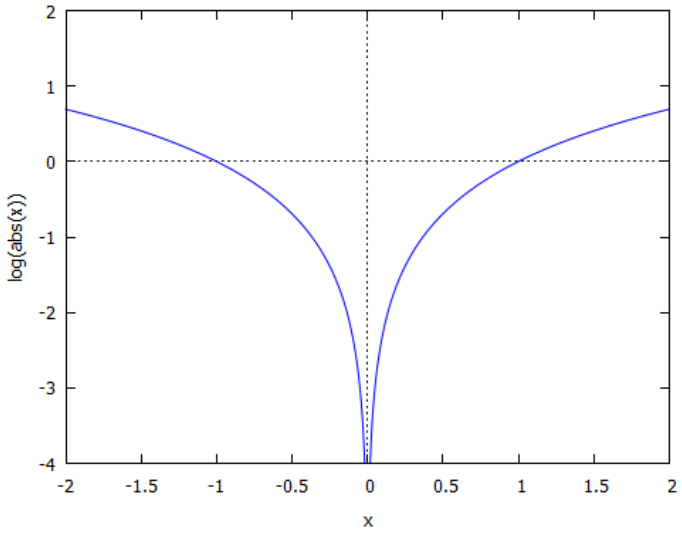

Figure 5.1 – Plotting the real part of the complex logarithm.

[same_xy, true | false] default: *false* [plot option]

If *true*, displays the graph with the same scale for both x and y axes. For a 2D plot, see also yx_ratio.

[xlabel, string] [plot option]
[ylabel, string] [plot option]
[zlabel, string] [plot option]

xlabel and *ylabel* specify the string that will label the first/second axis; if this option is not used, that label will be the name of the independent variable / "y", when plotting functions with *plot2d* or *implicit_plot*, or the name of the first/second variable, when plotting surfaces with *plot3d* or contours with *contour_plot*, or the first/second expression in the case of a parametric plot.

zlabel specifies the string that will label the third axis, when using *plot3d*. If this option is not used, that label will be “z”, when plotting surfaces, or the third expression in the case of a parametric plot. It will be ignored by *plot2d* and *implicit_plot*.

These options cannot be used with *set_plot_option*.

5.2.1.3 Zooming the plot

mails Robert and Laurent, 12.12.2018

5.2.2 2D

There are 5 basic types of 2D plot: explicit plot, parametric plot, discrete plot, implicit plot, and contour plot. The first three are implemented in function *plot2d*, the last two in separate functions.

5.2.2.1 plot2d

plot2d (⟨ *plot* | [$plot_1, \ldots, plot_n$]⟩⟨, *x_range*⟩⟨, *y_range*⟩⟨, *options*⟩) [function]
wxplot2d(...) [function]

These functions plot a two-dimensional graph of
- an expression giving the y-coordinate as a function of one variable being the x-coordinate (explicit plot),
- two expressions, one for the x- and one for the y-coordinate, as being functions of a single common parameter (parametric plot), or
- a number of discrete points in the xy-plane (discrete plot).

Each type can be used in single or multiple form, and different types can be combined to one representation.

plot | [$plot_1, \ldots, plot_n$]

A single plot is given as the first argument to *plot2d* while a multiple plot is given as a list (of plots) being the first argument. Each of the plots is either an expression (for an explicit plot), a parametric plot, or a discrete plot.

5.2.2.1.1 Explicit plot

A single 2D explicit plot displays the graph of an expression as a function of one variable. While the independent variable determines the x-coordinate of a plot point, the function value determines its y-coordinate. A multiple explicit plot displays multiple such graphs. An explicit functional expression in terms of the independent variable is given for each individual plot. The independent variable has to be the same for all plots of a multiple explicit plot.

x_range is of the form: *[x_name, min, max]*.

This is mandatory for explicit plots and specifies the name of the independent variable of the expression(s) to be plotted, and the range of its domain to be displayed on the horizontal axis. In case of a multiple explicit plot, the same *x_range* is used for all expressions. Individual plotting ranges are not possible (in contrast to *plot3d*). Hence, it is not possible to plot a piecewise defined function. In a combination of explicit and parametric plots, the name of the independent variable has to be *x*.

y_range is of the form: *[y, min, max]*.

This is optional and specifies the range of the codomain to be displayed on the vertical axis. If this option is used, the plot will show this exact vertical range, independently of the values reached by the plot. Everything outside of the given range will be clipped off. If the vertical range is not specified, it will be set according to the minimum and maximum values of the second coordinate reached by the plot. For *y_range* the name is always *y*. So it is wise not to use *y* as the name of the independent variable.

The complete syntax for an explicit plot is

plot2d ($\langle$ *expr* | *[*$expr_1, \ldots, expr_n$*]*$\rangle\langle$*, x_range*$\rangle\langle$*, y_range*$\rangle\langle$*, options*$\rangle$*)*

Options are described in sections 5.2.1.2 and 5.2.2.4. In case of a multiple plot, different colors will be used automatically for the different expressions and a legend will be created. Options present in case of a multiple plot apply to all plots; it is not possible to set options individually.

Note that the separate plot window (not when integrated into the wxMaxima file with wxplot2d) can be scrolled both horizontally and vertically to see beyond the selected ranges. The plot can be exported, e.g. as a *.png* file, directly from the separate plot window.

```
(%i1)   plot2d([%e^x, %e^(-x), log(x), 1/x, sqrt(x)],[x,-3,5],[y,-10,10]);
```

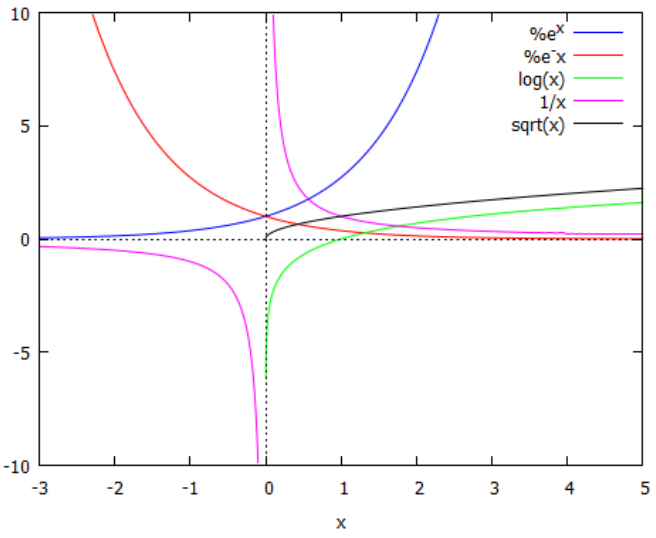

Figure 5.2 – Multiple 2D explicit plot.

5.2.2.1.2 Parametric plot

A single 2D parametric plot displays a graph generated in parallel by two different expressions, one for the x- and one for the y-coordinate, as being functions of a common single parameter. The name of the parameter always has to be t. A multiple parametric plot displays multiple such graphs. The complete syntax for a single parametric plot is

plot3d ([parametric, $expr_x, expr_y$*, [t, min, max]]* $\langle$*, options*$\rangle$*)*.

This creates a curve in in the two-dimensional space $expr_x \times expr_y$ in terms of the parameter t ranging from *min* to *max*.

Neither *x_range* nor *y_range* have to be present. When they are, they will specify the ranges to be displayed in the graph for the horizontal and the vertical axis. When they are not present, ranges will be set according to the minimum and maximum values of the coordinates reached by the plot points.

```
(%i1)   plot2d([[parametric, sin(t), cos(t),[t,0,2*%pi]],[parametric, sin(t
   ), cos(t)/2,[t,0,2*%pi]]],same_xy);
```

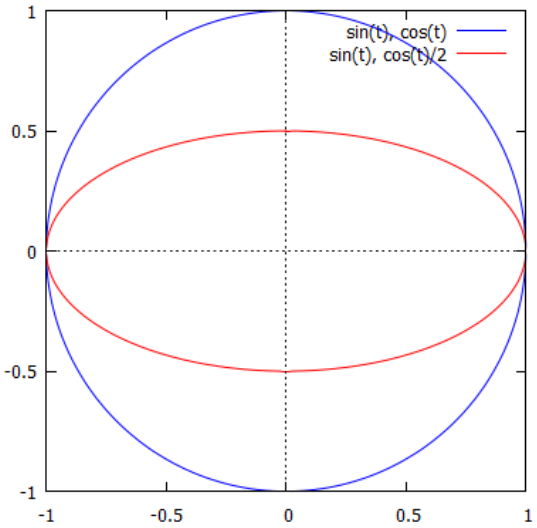

Figure 5.3 – Multiple 2D parametric plot.

5.2.2.1.3 Discrete plot

A single 2D discrete plot displays a graph consisting of a number of discrete points specified explicitly by their x- and y-coordinates. A multiple discrete plot displays multiple such graphs. The syntax for a single discrete plot is

$$[discrete, xlist, ylist] \mid [discrete, [[x_1, y_1], \ldots, [x_n, y_n]]$$

This creates a plot of n discrete points, where *xlist* and *ylist* are lists in square brackets of n elements each, containing in sequence the x- resp. y-coordinates of the points to be plotted. So the coordinates of the points can be enterd either separately for x- and y- valuse, or point by point. If no option *styles* is present, by default *[style, lines]* is assumed, that is, the discrete points are linked by line segments, see section 5.2.2.4.

```
(%i1)   plot2d([[discrete, makelist(i,i,1,10),makelist(sqrt(i),i,1,10],[
   discrete, makelist(i,i,0,10),makelist(sqrt(i)+sin(i),i,0,10)]],[style,
   points],[point_type,plus]);
```

For more examples see the examples to the function *rk* implementing the Runge-Kutta method for numerically solving a first order ODE.

Combining a discrete with an explicit plot, e.g., it is possible to represent the discrete data of an experiment together with a theoretically assumed continuous function to interpret them.

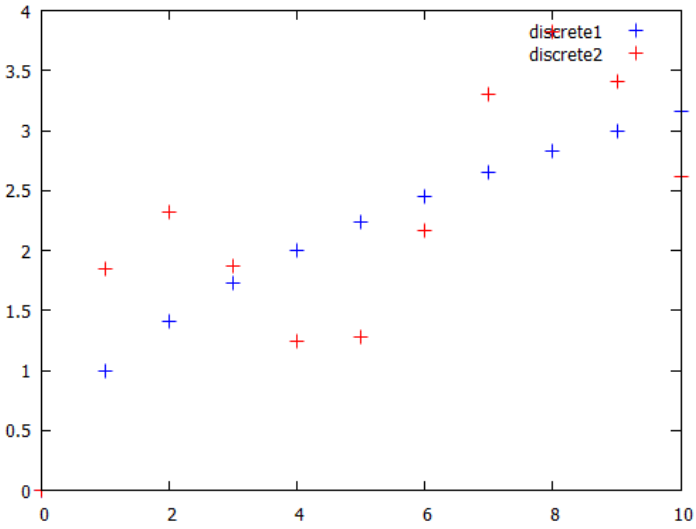

Figure 5.4 – Multiple discrete plot2d. The x- and y-coordinates of the points are generated by function *makelist*.

5.2.2.2 Implicit plot

A single 2D implicit plot displays the graph of a function given implicitly by an equation containing both the independent (x-coordinate) and the dependent (y-coordinate) variable. This equation does not have to be in explicit form.

implicit_plot (*eq* | [$eq_1, \ldots, eq_n$], *x_range*, *y_range* ⟨, *options*⟩) [function]
wximplicit_plot(...) [function]

In the first case this plots a single function defined implicitly by equation *equ*. The syntax is similar to *plot2d*. The domain is defined by *x_range* and *y_range* which are both mandatory. Both variable names can be selected freely. Multiple implicit plots can be combined to a graph by giving a list of equations [$eq_1, \ldots, eq_n$], one for each plot. Before it can be used this function has to be loaded.

```
(%i1)   load(implicit_plot);
(%i1)   implicit_plot([x^2+y^2=1, (x/2)^2+y^2=1/4], [x,-1,1], [y,-1,1],
   same_xy);
```

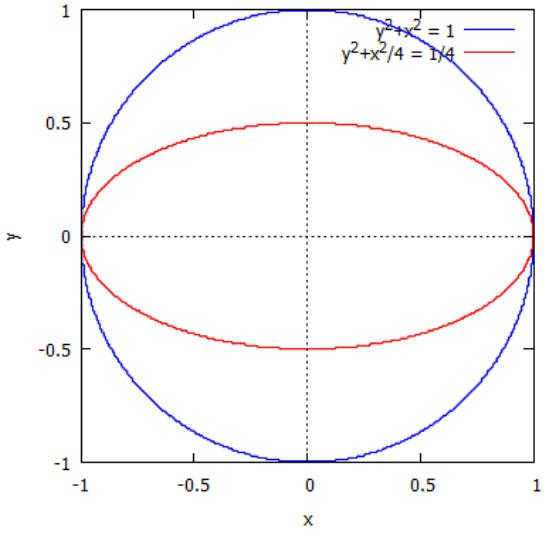

Figure 5.5 – Multiple implicit plot. The resulting curves are the same as in the multiple parametric plot of Fig. 5.3.

5.2.2.3 Contour plot

A 2D contour plot displays contours (curves of equal value) of a scalar-valued function of two arguments over a 2D region defined by the domains of these two arguments. Such a function can be considered a scalar field.

contour_plot (expr, x_range, y_range $\langle$*,* $[opt_1], \ldots, [opt_n]\rangle$*)* [function]
wxcontour_plot(...) [function]

This plots several curves of equal value of *expr* over the region defined by *x_range* and *y_range*. The names of the x- and y-coordinates can be selected freely. *contour_plot* accepts only options which can be used for *plot3d*. Each one of them has to be present as a list, i.e. the abbreviation of giving only the name of an option to indicate its value as true, is not allowed. Some of these options, e.g. *same_xy*, will cause the 2D plot to be displayed in a 3D representation.

```
(%i1)   contour_plot(x/y,[x,-2,2],[y,-2,2]);
```

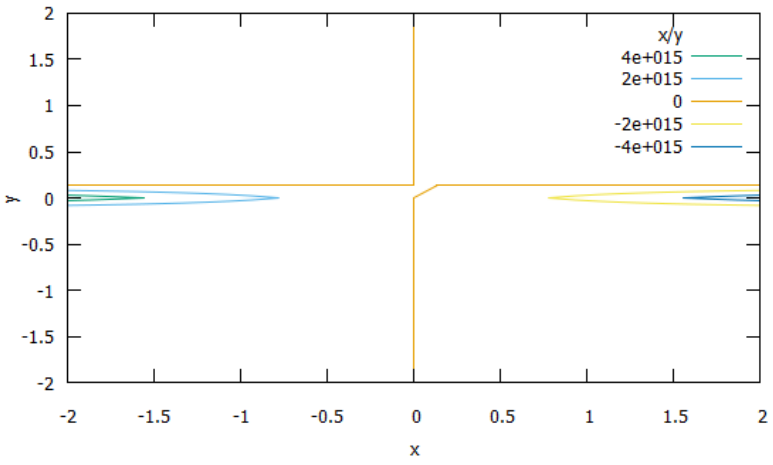

Figure 5.6 – Contour plot.

5.2.2.4 Options for 2D

[axes, $(value \mid false)$*]* default: *true* [plot option]

value can be either *true*, *false*, *x*, *y* or *solid*. If *false*, no axes are shown; if *x* or *y*, only the *x* or *y* axis will be shown; if *true*, both axes will be shown. *solid* will show the two axes with a solid line, rather than the default broken line.

[point_type, $type_1, \ldots, type_n$*]* [plot option]

Each set of points to be plotted with the style *points* or *linespoints* will be represented with objects taken from this list, in sequential order. If there are more sets of points than objects in this list, they will be repeated sequentially. The possible objects that can be used are: bullet, circle, plus, times, asterisk, box, square, triangle, delta, wedge, nabla, diamond, lozenge.

[style, $style_1 \mid [style_1], \ldots, style_n, \mid [style_n]]$ [plot option]

Describes the style(s) of the plot(s). If there are more plots than styles present, the styles will be repeated sequentially. Each style is either given by its name only, or as a list with additional arguments. In the first case, standard values are assumed for the style. In the second case, the first element of the list is the name of the style, followed by the arguments.

Each style can be either *lines* for line segments, *points* for isolated points, *linespoints* for segments and points, or *dots* for small isolated dots. Gnuplot accepts also an *impulses* style. If enclosed in a list, *lines* accepts one or two arguments: the width of the line and an integer that identifies a color. The default color codes are: 1: blue, 2: red, 3: magenta, 4: orange, 5: brown, 6: lime and 7: aqua. If Gnuplot is used with a terminal different than X11, those colors might be different. *points* accepts one to three arguments; the first one is the radius of the points, the second one is an integer that selects the color, using the same code used for lines and the third one is currently used only by Gnuplot and it corresponds to several objects instead of points. The default types of objects are: 1: filled circles, 2: open circles, 3: plus signs, 4: x, 5: *, 6: filled squares, 7: open squares, 8: filled triangles, 9: open triangles, 10: filled inverted triangles, 11: open inverted triangles, 12: filled lozenges and 13: open lozenges. Note that point types can be specified with option *point_type*, see above. *linespoints* accepts up to four arguments: line width, points radius, color and type of object to replace the points.

[yx_ratio, r] [plot option]

r defines the ratio between the vertical and the horizontal sides of the rectangle used to make the plot. See also same_xy.

5.2.3 3D

In 3D only two basic types of plot are possible: explicit plot and parametric plot. They are both implemented in function *plot3d*. Implicit 3D plots are possible only with *draw3d*.

5.2.3.1 plot3d

plot3d (plot ⟨*, options*⟩*)* [function]
wxplot3d(...) [function]

These functions plot a three-dimensional graph of
- an expression giving the z-coordinate as a function of two variables being the x- and the y-coordinates (explicit plot), - three expressions, one for each of the x-, y-, and z-coordinates, as being functions of two common parameters (parametric plot).

Multiple explicit plots can be combined to one representation. In contrast to plot2d, however, only single parametric plots can be displayed, and the combination of explicit and parametric plots is not possible, either.

5.2.3.1.1 Explicit plot

A single 3D explicit plot displays the graph of an expression giving the z-coordinate as a function of two variables being the x- and y-coordinates. A multiple explicit

plot displays multiple such graphs. In this case, an explicit functional expression in terms of the independent variables is given for each individual plot.

For a single plot, the explicit functional expression is given as the first argument to *plot3d*. In this case, *x_range* and *y_range* have to be the second and third argument, possibly followed by options. The complete syntax for a single plot is

plot3d (expr, x_range, y_range $\langle$*, options*$\rangle$*)*.

A multiple explicit plot can have two different forms, depending on whether the individual plots share the same x_range and y_range or not. In both cases, and in contrast to plot2d, x_range and y_range form part of the list of plots. The syntax for a multiple explicit plot using the same x_range and y_range is

plot3d ($[expr_1, \ldots, expr_n, x\_range, y\_range]$ $\langle$*, options*$\rangle$*)*.

The syntax for a multiple plot using a different x_range and y_range for each individual plot is

plot3d ($[[expr_1, x\_range_1, y\_range_1], \ldots, [expr_n, x\_range_n, y\_range_n]]$ $\langle$*, options*$\rangle$*)*.

x_range is of the form: *[x_name, min, max]*,
y_range is of the form: *[y_name, min, max]*.

These are both mandatory within explicit plots and specify the names (which can be chosen freely) of the independent variables of the expression(s) to be plotted, and the ranges of their domains. *x_range* and *y_range*, however, can be repeated as part of the options. In this case, their names have to be x and y, and they specify the ranges to be displayed on the two horizontal axes. Everything outside of the given ranges will be clipped off. If the ranges are not specified within the options, ranges to be displayed will be set according to the minimum and maximum values of the domains specified within the explicit plots.

z_range is of the form: *[z, min, max]*.

This is optional and specifies the range of the codomain to be displayed on the vertical axis. If this option is used, the plot will show that exact vertical range, independently of the values reached by the plot. Everything outside of the given range will be clipped off. If the vertical range is not specified, it will be set according to the minimum and maximum values of the third coordinate of the plot points. For *z_range* the name is always *z*. So it is wise not to use *z* as the name of one of the independent variables.

Options are described in sections 5.2.1.2 and 5.2.3.3. In case of a multiple plot, different colors will be used automatically for the different expressions and a legend will be created. Options present in case of a multiple plot apply to all plots; it is not possible to set options individually.

Note that the separate plot window (not when integrated into the wxMaxima file with wxplot3d) can be scrolled in all three directions to see beyond the selected ranges. Furthermore, by using the mouse, the surface plotted can be turned around and viewed from all sides. The plot can be exported, e.g. as a *.png* file, directly from the separate plot window.

Here is an example of a multiple explicit plot consisting of three individual plots, each having different x- and y-ranges

```
(%i1)   plot3d([[x^2+y^2,[x,-4,4],[y,-4,4]],[x^3+y^3,[x,-3,3],[y,-3,3]],[x
   ^4+y^4,[x,-2,2],[y,-2,2]]]);
```

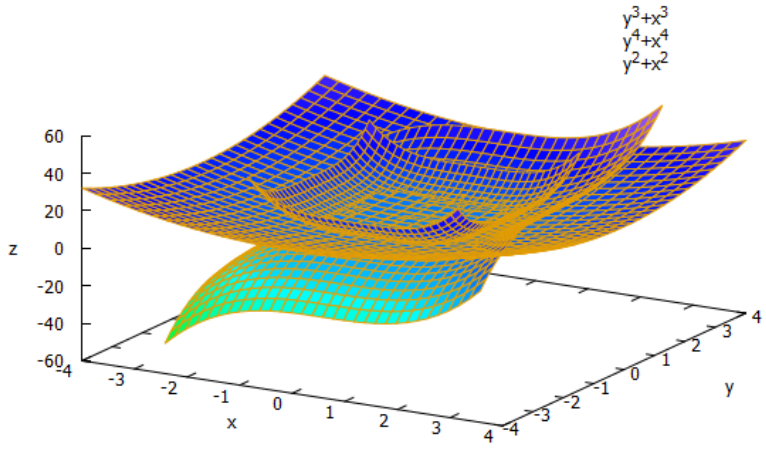

Figure 5.7 – Multiple 3D explicit plot with different x- and y-ranges for each surface.

5.2.3.1.2 Parametric plot

A single 3D parametric plot displays a surface generated in parallel by three different expressions (for the x-, y- and z-coordinates) as functions of two common parameters. The names and the ranges of these parameters don't necessarily have anything to do with the names and ranges of the x-, y- and z-coordinates. A multiple parametric plot displays multiple such surfaces. The complete syntax for a single parametric plot is

plot3d ([expr$_x$, expr$_y$, expr$_z$], [p_name$_1$, min$_1$, max$_1$],[p_name$_2$, min$_2$, max$_2$] ⟨, options⟩).

This creates a surface in the three-dimensional space $expr_x \times expr_y \times expr_z$ in terms of the two common parameters p_name_1 and p_name_2.

Neither *x_range* nor *y_range* nor *z_range* have to be present (in the options section). When they are, their names have to be x, y, and z, and they will specify the ranges to be displayed for the two horizontal and the vertical axes. When they are not present, ranges will be set according to the minimum and maximum values of the coordinates of the plot points.

```
(%i1)   plot3d([t+u,t-u,t*u],[t,0,2],[u,0,2]);
```

5.2.3.2 Coordinate transformations for 3D

plot3d not only supports standard coordinate transformations from cylindrical or spherical to cartesian coordinates, but in addition lets the user define and apply his own special coordinate transformation functions. This not only allows for giving the expressions to be plotted in cylindrical or spherical coordinates, but in any type of coordinates the user wants.

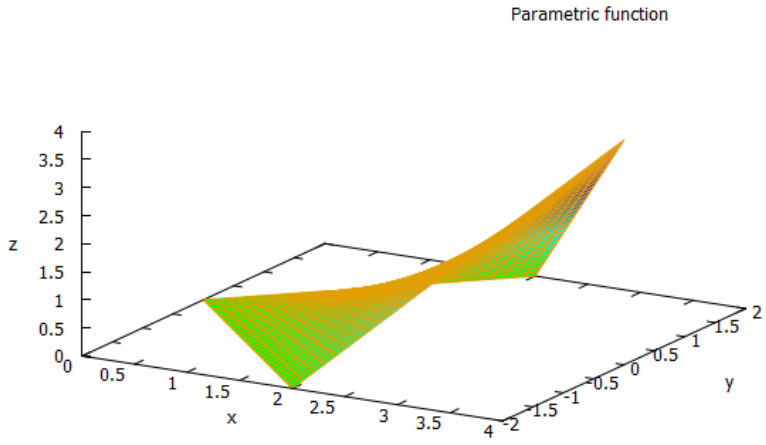

Figure 5.8 – Single 3D parametric plot.

5.2.3.2.1 Standard coordinate transformations

Standard coordinate transformations predefined for *plot3d* are
- cylindrical to cartesian (*polar_to_xy*), and
- spherical to cartesian (*spherical_to_xyz*).

Note that *polar_to_xy* cannot be used with plot2d, it is only a 3D feature, and it should have better been called *cylindrical_to_xyz*. In the next section we will define our own coordinate transformation carrying precisely this name.

A coordinate transformation is invoked in a *plot3d* with option *transform_xy*, see section 5.2.3.3:

```
(%i1)   plot3d ( 5, [theta, 0, %pi], [phi, 0, 2*%pi], same_xyz,
[transform_xy, spherical_to_xyz]);
```

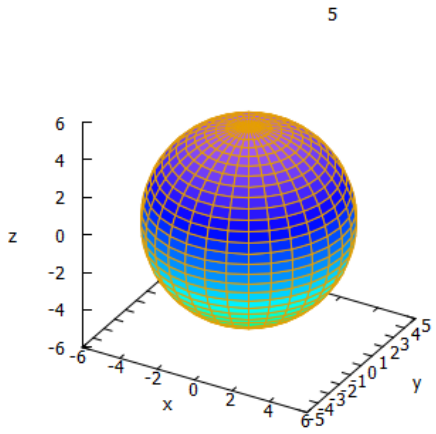

Figure 5.9 – 3D explicit plot in spherical coordinates.

5.2.3.2.2 User-defined coordinate transformations

make_transform ([cname$_1$, cname$_2$, cname$_3$],[expr$_x$, expr$_y$, expr$_z$]) [function]

Returns a function suitable to be used in the option *transform_xy* of *plot3d*. $cname_1, cname_2, cname_3$ specify the names of the three new coordinates, and $expr_x$, $expr_y$, $expr_z$ their functional expressions to build the cartesian x-, y- and z-coordinates.

As an example, we shall define a coordinate transformation called *cylindrical_to_xyz* which is in fact identical to the preconfigured one *polar_to_xy*

```
(%i1)   cylindrical_to_xyz: make_transform([r,phi,z], r*cos(phi), r*sin(phi
   ),z)$
(%i2)   plot3d (-r, [r, 0, 3], [phi, 0, 2*%pi], [transform_xy,
   cylindrical_to_xyz]);
```

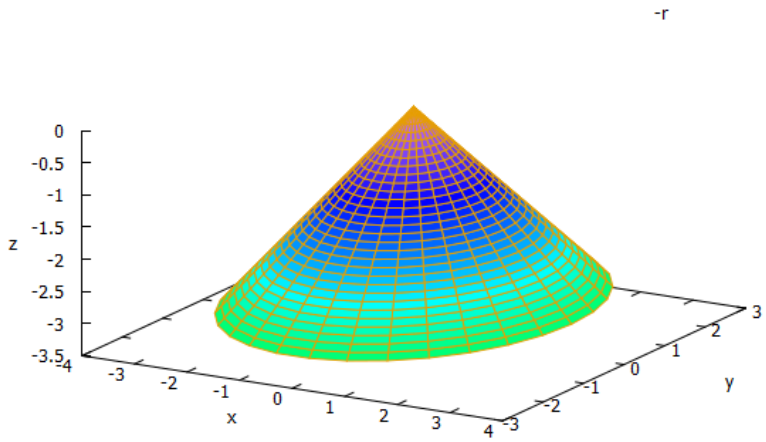

Figure 5.10 – 3D explicit plot in cylindrical coordinates.

5.2.3.3 Options for 3D

[same_xyz, true | false] default: *false* [plot option]

If *true*, the scales of all three axes will be the same.

[transform_xy, false | ct_name] default: *false* [plot option]

This is a 3D option only. It allows for coodinate transformations within *plot3d*. *ct_name* is the name of either a predefined coordinate transformation (*polar_to_xy* or *spherical_to_xyz*), or one defined by the user with *make_transform*. See section 5.2.3.2 for details.

5.3 Draw

5.3.1 Introduction

This package is a Maxima interface to GNUplot. It allows for significantly more functionality compared to Maxima's propriatory *plot* package, but at the price of a far more complicated syntax.

This package was written and is being maintained by Mario Rodriguez Riotorto. Ample examples can be found in ...

5.3.2 General structure

The *draw* package has to be loaded explicitly by the user with *load(draw)* prior to using it.

draw (..., (*gr2d* | *gr3d*), ... ⟨, *options*⟩) [function]

This main function of the package plots a column of *scenes*, each of them being a picture, a graphical diagram, a plot in either 2D or 3D. Each scene is evoked by an appearance of a scene constructor, either *gr2d* or *gr3d*, which can be combined in any order and number. General options for all scenes may follow. Each scene can contain multiple graphical objects, e.g. plots.

5.3.2.1 Using options

5.3.2.1.1 General syntax

The general syntax for options is

option_name = [$value_1, \ldots, value_n$].

Global options may appear anywhere in *draw*, *gr2d* or *gr3d*, *draw2d* or *draw3d*, their position does not matter.

5.3.2.1.2 Setting defaults for multiple scenes

set_draw_defaults ($opt_1, \ldots, opt_m$) [function]
set_draw_defaults ()

The first line sets up user defaults for options to be used for all subsequent scenes. The second line removes all existing user defaults for the subsequent scenes.

5.3.2.1.3 Predefined personal sets of options

In *maxima-init.mac* I have predefined lists of personal default options: *my_general_options*, *my_2d_options* and *my_3d_options*. They can be incorporated as needed in any scene by simply including the respective symbols as global options.

```
(%i1)   draw3d(implicit(x^2+y^2=z^2,x,-1,1,y,-1,1,z,-1,1),
   my_general_options,my_3d_options);
```

Alternatively, they can be permanently assigned by *set_draw_defaults*. This assignment is not yet done in *maxima-init.mac*, because it depends on the dimension of the plot to be created.

```
(%i1)   apply(set_draw_defaults,my_general_options);
(%i2)   draw3d(implicit(x^2+y^2=z^2,x,-1,1,y,-1,1,z,-1,1,my_3d_options));
```

Or in case two lists shall be combined:

```
(%i1)   apply(set_draw_defaults,append(my_general_options,my_3d_options));
(%i2)   draw3d(implicit(x^2+y^2=z^2,x,-1,1,y,-1,1,z,-1,1));
```

5.3.2.1.4 User_preamble

This option allows to specify certain gnuplot settings which cannot be incorporated with the usual syntax for options.

user_preamble = "*set* opt_1;...;*set* opt_n"

Options are specified by using gnuplot's *set* command followed by the option and possible values. Options are separated by a semicolon.

```
user_preamble = "set raxis; set grid polar; set size 1.1,1.1"
```

5.3.2.1.4.1 Predefined personal user_preambles

In *maxima-init.mac* I have a predefined list of options for the user_preamble in *my_user_preamble* which can be easily incorporated into a scene.

```
(%i1)   draw2d(explicit(x,x,0,1),user_preamble=my_user_preamble);
```

The user preamble of a specific scene can contain other options as well.

```
(%i1)   draw2d(polar(1,theta,0,2*%pi),user_preamble=append(my_user_preamble
   ,["set raxis","set grid polar"]);
```

5.3.3 2D

gr2d ($\langle opt_1,...,opt_m,\rangle graph\_obj_1,...,graph\_obj_n$) [scene constructur]

This is the constructor for a single 2D scene to be used as an argument to function *draw*. Multiple graphical objects $gobj_1,...,gobj_n$ can be plotted within the scene under global options $opt_1,...,opt_m$.

draw2d ($\langle opt_1,...,opt_m,\rangle graph\_obj_1,...,graph\_obj_n$) [function]
wxdraw2d (...) [function]

These two functions, see this chapter's introduction for their difference, are a shortcut for *draw(gr2d(*$\langle opt_1,...,opt_m,\rangle graph\_obj_1,...,graph\_obj_n$*))*.

5.3.3.1 Explicit plot

explicit (f, x, min, max) [graphical object]

A graphical object of this type plots function f, given in explicit form, with the independent variable x in the range from x=min to x=max.

5.3.3.1.1 Piecewise defined function

In combination with the global options *xrange* and *yrange* it is possible to plot a piecewise defined function.

```
(%i1)   draw2d(explicit(0.5,x,0,1),explicit(1,x,1,2),explicit(1.5,x,2,3),
   xrange=[0,3],yrange=[0,2]);
```

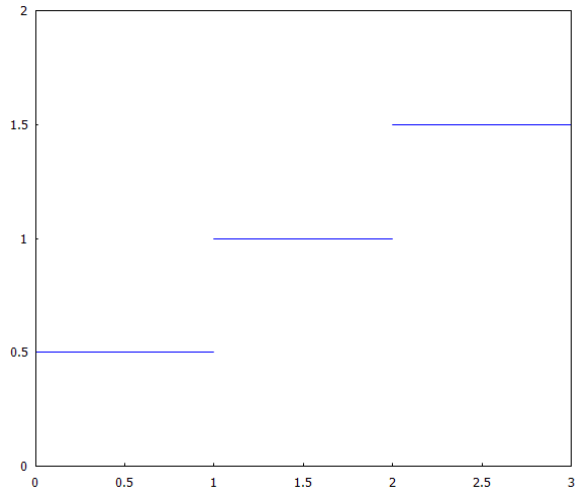

Figure 5.11 – Plotting a piecewise defined function with draw.

5.3.3.2 Implicit plot

explicit (f, x, min, max) [graphical object]

A graphical object of this type plots function f, given in implicit form, with dependent variable x in the range from x=min to x=max. Note that in combination with the global option *xrange* it is possible to plot a piecewise defined function.

5.3.3.3 Polar plot

polar (radius, ang, ang_{min}, ang_{max}) [graphical object]

Plots the radius as a function of the angle in the given range. This object can be plotted with an underlying polar grid, see thread in maxima-discuss from March 2019.

```
(%i1)   draw2d(polar(1-(theta/(2*%pi)),theta,0,2*%pi), xrange=[-1,1],
   yrange=[-1,1], proportional_axes = xy, user_preamble="set raxis; set
   grid polar");
```

Underlying cartesian and polar grids can be combined, too.

```
(%i1)   draw2d(polar(1-(theta/(2*%pi)),theta,0,2*%pi), xrange=[-1,1],
   yrange=[-1,1], proportional_axes = xy, grid=true, user_preamble="set
   raxis; set grid polar");
```

5.3.4 3D

gr3d ($\langle opt_1, \ldots, opt_m, \rangle graph\_obj_1, \ldots, graph\_obj_n$) [scene constructur]

This is the constructor for a single 3D scene to be used as an argument to function *draw*. Multiple graphical objects $gobj_1, \ldots, gobj_n$ can be plotted within the scene under global options $opt_1, \ldots, opt_m$.

draw3d ($\langle opt_1, \ldots, opt_m, \rangle graph\_obj_1, \ldots, graph\_obj_n$) [function]
wxdraw3d(...) [function]

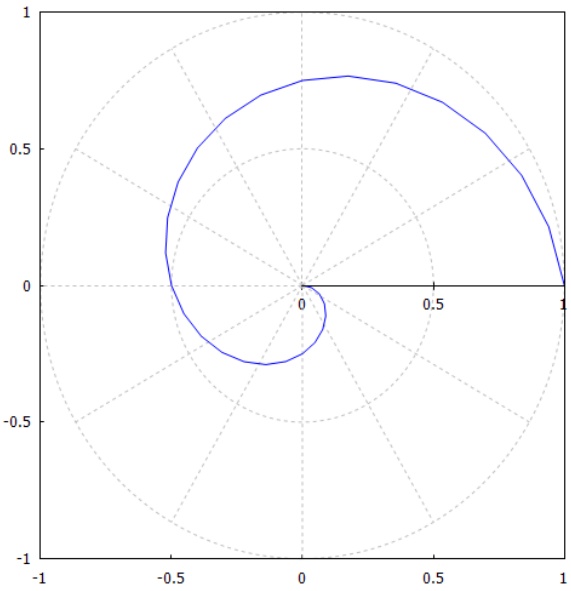

Figure 5.12 – Plotting a function in polar coordinates and with an underlying polar grid with draw.

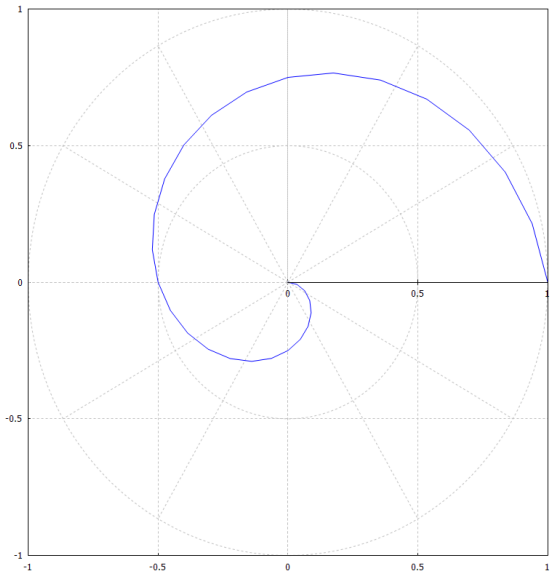

Figure 5.13 – Plotting a function in polar coordinates and with aunderlying cartesian and polar grids with draw.

These two functions, see this chapter's introduction for their difference, are a shortcut for *draw(gr3d(*$\langle opt_1, \ldots, opt_m, \rangle gobj_1, \ldots, gobj_n$*))*.

5.3.4.1 Explicit plot

explicit $(f, x, x_{min}, x_{max}, y, y_{min}, y_{max})$ [graphical object]

A graphical object of this type plots function f, given in explicit form, with the independent variables x and y in the given ranges.

Just like in the 2D case, in combination with the global options *xrange*, *yrange* and *zrange* it is possible to plot a piecewise defined function.

5.3.4.2 Implicit plot

implicit $(f, x, x_{min}, x_{max}, y, y_{min}, y_{max}, z, z_{min}, z_{max})$ [graphical object]

A graphical object of this type plots function f, given in implicit form, with dependent variable x in the range from x=min to x=max. Note that in combination with the global option *xrange* it is possible to plot a piecewise defined function.

5.3.5 List of available options

proportional_axes=(*xy* | *xyz*) default: *none* [plot option]

Displays with the specified axes proportional to their relative lengths. The value *xy* can be used for 3D, too.

xrange=[min,max] default: *auto* [plot option]

Specifies the range of the x-axis for this scene. If this option is missing, the minimal range used by the graphical objects will be shown in the scene.

yrange=[min,max] default: *auto* [plot option]

Specifies the range of the y-axis for this scene. If this option is missing, the minimal range used by the graphical objects will be shown in the scene.

Chapter 6

Batch Processing

Part III

Concepts of Symbolic Computation

Chapter 7

Data types and structures

7.1 Introduction

For the data type *string* see section 30.1.

7.2 Numbers

7.2.1 Introduction

7.2.1.1 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.

7.2.1.2 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
```

7.2.2 Integer and rational numbers

7.2.2.1 Representation

7.2.2.1.1 External

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

```
(%i1)   a:1;
(%o1)                                1
(%i2)   b:-2/3;
                                     2
(%o2)                               - -
                                     3
(%i3)   100!;
(%o3)   933262154439441526816992388562667004907159682643816214685929\
        638952175999932299156089414639761565182862536979208272237582\
        511852109168640000000000000000000000000
```

7.2.2.1.2 Internal

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

7.2.2.1.2.1 Canonical rational expression (CRE)

7.2.2.2 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
(%i9)   ratnump(a+d);
(%o9)                              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.

7.2.2.3 Type conversion

7.2.2.3.1 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.

7.2.2.3.2 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);
                                3602879701896397
(%o1)                           ----------------
                                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
                                      1
(%o1)                          /R/    --
                                      10
```

7.2.3 Floating point numbers

7.2.3.1 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
(%i1)                    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

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

7.2.3.2 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);
(%o2)                3.1415926535897932384626433832795b0
```

7.2.4 Complex numbers

7.2.4.1 Introduction

7.2.4.1.1 Imaginary unit

%i [special variable]

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

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

7.2.4.1.2 Internal representation

There is no generic data type for complex numbers. Maxima represents a complex number in standard form as a sum $a + ib$, *realpart* and *imagpart* each being of one of the four generic types of numbers, see sect. 7.2.1.1. More complicated expressions involving complex numbers are represented just as real valued ones, with the only difference that the special variable %i appears in them. This variable is treated as would be any other variable.

```
(%i1)   r: 3+%i*5;
(%o1)                                5i + 3
(%i2)   :lisp $r
((MPLUS SIMP) 3 ((MTIMES SIMP) 5 $%I))
(%i3)   p: polarform(r);
(%o1)                          sqrt(34) %e^(%i arctan(5/3))

(%i4)   :lisp $p
((MTIMES SIMP) ((MEXPT SIMP) 34 ((RAT SIMP) 1 2))
((MEXPT SIMP) $%E ((MTIMES SIMP) $%I ((%ATAN SIMP) ((RAT SIMP) 5 3)))))
```

7.2.4.1.3 Canonical order

The canonical order in which Maxima returns a complex-valued expression does not differ from the order of an equivalent expression which replaces %i by any other variable. All the rules for determining the canonical order, including the effect of *powerdisp*, therefore are completely unaware of complex numbers.

```
(%i5)   powerdisp:false$  /* default */
        a + b*%i;
        1 + 2*%i;
        1 + b*%i;
        -b*%i +1;
(%o2)                              ib + a
(%o3)                              2i + 1
(%04)                              ib + 1
(%05)                              1 - ib
(%i6)   z+k*%i+b+a*%i+4+3*%i+2-%i;
(%o6)                     z + ik + b + ia + 2i + 6
(%i6)   z+k*%j+b+a*%j+4+3*%j+2-%j;
(%o6)                    z + %jk + b + %ja + 2%j + 6
```

7.2.4.1.4 Simplification

Complex expressions are, in contrast to real ones, not always simplified as much as possible automatically. Simplification of products, quotients, roots, and other functions of complex expressions can usually be accomplished by applying *rectform*.

```
(%i1)   (2+%i)*(3-%i);
(%o1)                              (3 - i)(i + 2)
(%i2)   rectform(%);
(%o2)                                 i + 7
(%i3)   (2+%i)/(3-%i);
```

```
(%o3)                                  (i+2)/(3-i)
(%i4)    rectform(%);
(%o4)                                  i/2 + 1/2
```

7.2.4.1.5 Properties

A variable can be declared the property *real*, *complex*, or *imaginary*. These properties are recognized by the functions of section 7.2.4.2 and 7.2.4.3. Note that these functions consider symbols, unless declared otherwise (*complex*, *imaginary*) or evaluating to a complex expression, as *real*.

```
(%i1)    declare(z,complex,r,real)$
```

7.2.4.1.6 Code

The code of functions and variables for complex numbers is contained in file *conjugate.lisp*.

7.2.4.1.7 Generic complex data type

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

7.2.4.2 Standard (rectangular) and polar form

Maxima distinguishes standard (rectangular) 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.

Note that these functions consider symbols, unless declared otherwise (*complex*, *imaginary*), see section 7.2.4.1.5, or evaluating to a complex expression, as *real*.

7.2.4.2.1 Standard (rectangular) form

rectform (expr) [function]

Converts a complex expression *expr* to standard form $a + i\,b$ with $a, b \in \mathbb{R}$. Note that e.g. for a complex function this decomposition is always possible. While the imaginary part is parenthezised when it contains more than one element, this is not done for the real part. If *expr* is an equation, both sides are decomposed separately. *rectform* recognizes if a variable has been declared any of the properties *real*, *imaginary* or *complex*.

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

```
(%o2)                    z + ik + b + ia + 2i + 6
(%i3)   rectform(expr);
(%o3)                    z + i(k + a + 2) + b + 6
(%i5)   declare(z,complex,b,real)$ rectform(expr);
(%o5)               Re(z) + i(Im(z) + k + a + 2) + b + 6
```

realpart (expr) [function]
imagpart (expr) [function]

Return the real resp. imaginary part of *expr*. These functions work on expressions involving trigonometric and hyperbolic functions, as well as square root, logarithm, and exponentiation.

```
(%i6)   realpart(expr);
(%o6)                                z+b+6
(%i7)   imagpart(expr);
(%o7)                                k+a+2
```

7.2.4.2.2 Polar coordinate form

polarform (expr) [function]

Converts a complex expression to the equivalent polar coordinate form

$$r\,e^{i\varphi} = r\left(\cos\varphi + i\sin\varphi\right)$$

with r being the complex absolute value and φ the complex argument.

cabs (expr) [function]
carg (expr) [function]

Return the complex absolute value resp. the complex argument of *expr*.

```
(%i1)   polarform(3+4*%i);
(%o1)                              5 %e^(%i arctan 4/3)
(%i2)   polarform(a+b*%i);
(%o2)                          sqrt(a^2 + b^2) %e^(%i atan2 (b,a))
(%i3)   cabs(a+b*%i);
(%o3)                              sqrt(a^2 + b^2)
(%i4)   carg(a+b*%i);
(%o4)                              atan2 (b, a)
```

7.2.4.3 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)$ 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
(%i8)   declare([z1,z2],complex)$ conjugate(z1*z2);
(%o9)                              conjugate(z1) conjugate(z2)
(%i10)  f:a+b*%i$ (f+conjugate(f))/2;
(%o10)                               a
```

7.2.4.3.1 Internal representation

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

```
(%i1)   declare(a,complex)$ b:conjugate(a);
(%o1)                                conjugate(a)
(%i2)   :lisp $b
(($CONJUGATE SIMP) $A)
```

7.2.4.4 Predicate function

complexp (expr) [own 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(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
```

7.3 Boolean values

7.4 Constant

7.5 Sharing of data

It is very important to understand a concept employed not only in Lisp and Maxima, but also in many other programming languages and CAS', the concept of *sharing* data instead of *copying* them. Assignment of a list or matrix (or a vector, which is always a list or a matrix) from symbol *a* to symbol *b* will not create a *copy* of this data structure which then belongs to *b*, but will *share* the existing data structure belonging to symbol *a*. This means that symbol *b* will only receive a pointer to the existing data structure, not a new one with the same values as the one of symbol *a*.

Now if symbol *a* is killed or is assigned a completely *new* data structure, the old data structure will remain belonging only to symbol *b*. But if the old data structure of symbol *a* is only *modified* in the value of some element, symbol *b* will evaluate to this modified data structure of symbol *a*. And vice versa, if symbol *b* modifies the value of some element of the shared data structure, symbol *a* will evaluate to this modified structure.

```
(%i1)   a:[1,2,3];
(%o1)                           [1,2,3]
(%i2)   b:a;
(%o2)                           [1,2,3]
(%i3)   a:[4,5,6];
(%o3)                           [4,5,6]
(%i4)   b;
(%o4)                           [1,2,3]

(%i1)   a:[1,2,3];
(%o1)                           [1,2,3]
(%i2)   b:a;
(%o2)                           [1,2,3]
(%i3)   a[2]:x$ a;
(%o4)                           [1,x,3]
(%i5)   b;
(%o5)                           [1,x,3]
(%i6)   b[3]:y$ b;
(%o7)                           [1,x,y]
(%i8)   a;
(%o8)                           [1,x,y]
```

Note: Adding columns or rows to an existing matrix with *addcol* or *addrow* will create a *new* data structure with respect to sharing.

In order to create a real copy of an existing list or matrix, the functions *copylist* and *copymatrix* have to be used.

Chapter 8

List, matrix, structure

8.1 List

8.1.1 makelist

makelist (⟨*expr*⟨*, n*⟩⟩*)* | [function]
makelist (expr, i⟨*, i*$_0$⟩*i*$_{max}$⟨*, step*⟩*)* |
makelist ((*expr* | [*expr*$_1$*, . . . , expr*$_n$]), *x, list)*

This is a very powerful function to create lists from expressions and/or other lists. There are three general forms, each of them with some possible variations.

The first form ...

The second form ...

For an example see the example to the discrete plot of *plot2d*.

The third form returns a list whose elements are evaluations of *expr* or sublists being evaluations of *expr*$_1, \ldots, expr_n$. These expressions are functions of variable x, which takes its values running through *list*. *makelist*'s return value has as many elements as *list* has, i.e. *length(list)* elements. For *j* running from 1 through *length(list)*, the j^{th} element of the list returned is given by *ev(expr,x=list[j])* or *ev(*[*expr*$_1, \ldots, expr_n$]*,x=list[j])*.

For an example see the second example to the function *rk* implementing the Runge-Kutta method for numerically solving a first order ODE.

8.1.2 create_list

8.2 Matrix

8.3 Structure

Chapter 9

Expression

9.1 General definitions

Any meaningful combination of operators, symbols and numbers is called an *expression*. An expression can be a mathematical expression (numerical, symbolical, or a combination of both), but also a function call, a function definition, any other program statement (in Lisp called a *form*) or even a whole program.

An *expression* is built up of elements, which are the operators, symbols and numbers. An expression consisting of only one element is called an *atom* or *atomic expression*.

Expressions are structured, so that they can be subdivided into *subexpressions*. A subexpression is itself an expression, which can again be subdiveded. Thus, an expression can have different *levels* of subdivision and can be viewed as having a tree structure.

A non-atomic subexpression is called *complete*, if it is an operand together with all of its arguments. For example, $x + y + z$ is a complete subexpression of $2 * (x + y + z)/w$ while $x + y$ is not. Note that *part* can select an incomplete subexpression when given a list as the last argument.

9.2 Forms of representation

There are three different forms of representing expressions in Maxima: the *user visible form* (UVF), the *general internal form* (GIF) and another, specific internal representation called *canonical rational expression* (CRE). There are fundamental differences between these representations which the user has to be aware of.

9.2.1 User visible form (UVF)

The *user visible representation* (UVF) is the way Maxima displays an expression to the user. Most often, the user will also use this representation to enter an expression. For example, a fraction like $1/\sqrt{(x)}$ is entered and displayed as a fraction. If, instead, it is entered in exponention form $x^{-1/2}$, which is possible, it will still be displayed as a fraction, not in exponential form.

Within the UVF we have different appearances of the expression, depending on the

user interface we work with (e.g. wxMaxima, iMaxima or the console), or depending on whether *display2d* is set or not. However, in this chapter we are not concerned about these differences in appearance, they belong into the chapter on input and output or the chapter on different user interfaces. We don't distinguish between these appearances here, because the abstract UVF representation is the same for all of them.

```
(%i1)   1/sqrt(x);
(%o1)                                 1
                                    ──────
                                    √(x)
(%i2)   x^(-1/2);
                                      1
(%o2)                               ──────
                                    √(x)
(%i3)   display2d
(%o3)                                true
(%i4)   display2d:false$
(%i5)   1/sqrt(x);
(%o5)                                1/sqrt(x)
```

9.2.2 General internal form (GIF)

The *general internal representation* (GIF) is the way Maxima stores and handles an expression internally. This is done on the Lisp level. But again, here we are not concerned about how actually expressions are stored internally as lists, but in the abstract representation of the expression. There are fundamental differences in comparison to the UVF. Nevertheless, to make visible these differences of the abstract format, we have to go down into the Lisp level and look at the lists.

```
(%i1)   expr: x^(-1);
                                      1
(%o1)                                 ─
                                      x
(%i2)   :lisp $expr
((MEXPT SIMP) $X -1 )
(%i2)   1/x;
                                      1
(%o2)                                 ─
                                      x
(%i3)   :lisp $%
((MEXPT SIMP) $X -1 )
(%i3)   x^(-1/2);
                                      1
(%o3)                               ──────
                                    √(x)
(%i4)   :lisp $%
((MEXPT SIMP) $X ((RAT SIMP) -1 2))
(%i4)   1/sqrt(x);
                                      1
(%o4)                               ──────
                                    √(x)
(%i5)   :lisp $%
((MEXPT SIMP) $X ((RAT SIMP) -1 2))
(%i5)   -x/2;
```

```
(%o5)                                  -x
                                       --
                                       2

(%i6)    :lisp $%
((MTIMES SIMP) ((RAT SIMP) -1 2) $X)
```

What we see is that internally our top level fraction having a variable in the denominator is always represented as an exponential form of the variable having a negative exponent. Only if the denominator is a numerical value, a fraction will be represented internally as such. This happens for example in the exponent of $x^{-1/2}$. The expression $-x/2$ is internally represented as the product $-1/2 * x$, and $-x$ is represented as the product $-1 * x$.

In these examples we have seen some of the major differences between the UVF and the GIF representation. A more comprehensive explanation of Maxima's internal representation of expressions can be found in [**FatemMGS**].

We also saw in the above examples, that the different levels of subexpressions, that is: the tree structure of the overall expression, are nicely represented in Lisp by nested lists.

9.2.3 Canonical rational expression (CRE)

The *canonical rational expression* (CRE) is an additional, special internal representation of expressions which Maxima uses in certain cases. It is explained in section 7.1 of [**FatemMGS**].

9.3 Canonical order

When an expression is entered, Maxima orders its elements in a specifc way before storing the expression. Maxima uses what is called the *canonical order* in any representation (UVF, GIF, CTE). This helps Maxima (and, by the way, the user, too) to determine, whether two expressions are literally equal or not. For example, the terms of a sum of powers of the same variable are internally stored in the order of increasing powers.

```
(%i1)    powerdisp:true$
(%i2)    x+x^3+x^2;
(%o2)                                  x+x^2+x^3;
(%i2)    :lisp$%
((MPLUS SIMP) $X ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $X 3))
```

Note, however, that the order in which an expression is displayed in UVF depends on whether the flag *powerdisp* is set to *true* or *false* (default). If it is *false*, display is in the reverse canonical order. GIF is not affected by *powerdisp*.

9.4 Noun and verb

9.5 Equation

lhs = rhs [equation]
expr [equation]

An *equation* in Maxima usually has the form *lhs* = *rhs*, where *lhs* and *rhs* are expressions and = is the *equation operator*. However, if a function requires an equation as its argument, one side of it can be omitted if it is zero, and only the other side *expr* be provided. In this case, Maxima assumes the equation $expr = 0$. So for example $a * x^2 + b * x + c = 0$ can be represented simply by $a * x^2 + b * x + c$.

9.6 Reference to subexpression

9.6.1 Identify and pick out subexpression

Internally, a Maxima expression is represented as a tree structure. The root, each node and each leaf of this tree can be identified by a finite sequence of indices, which are natural numbers including zero. In general, the main operator of the expression is its root and carries the number zero, while its main operands are numbered from left to right in the displayed form of the expression using natural numbers starting from one. These indices, assigned to the root and the upmost level of nodes, constitute level one of the identification scheme of the expression. Within each of the main operands, a level two numbering is obtained in the same way: zero is assigned to the main operand's main operator, while its operands are numbered from left to right starting with one. This numbering scheme is repeated from level to level descending into the tree structure of the expression, until finally an atom is reached, which is now uniquely identified by a finite sequence of indices. We demonstrate this scheme with the help of the following function, which can pick any node or leaf (or even a number of them from the same level) of the expression's tree structure.

part (expr, $n_1, \ldots, n_{k-1}, \left(n_k \,\middle|\, [n_{k1}, \ldots, n_{kl}]\right)$*)* [function]

Returns the root (main operator), a node (subexpression), or a leaf (atom) of the displayed form of the expression *expr*. The part obtained from expr corresponds to the finite sequence of indices $n_1, \ldots, n_k$. At first, part n_1 of expr is obtained, then part n_2 of that, etc. The value return is part n_k of part n_{k-1} of $\ldots$ part n_2 of part n_1 of *expr*. If no indices are specified, *expr* is returned.

```
(%i1)   eq:'diff(2*y,x) = a*y + (2+b)/x;
```

$$\text{(\%o1)} \qquad \frac{d}{dx}(2y) = ay + \frac{b+2}{x}$$

```
(%i4)   part(eq,0); part(eq,1); part(eq,2);
(%o2)                                =
```

$$\text{(\%o3)} \qquad \frac{d}{dx}(2y)$$

```
(%o4)                                  ay + (b + 2)/x
(%i9)   part(eq,1,0); part(eq,1,1); part(eq,1,2); part(eq,1,1,0); part(eq
   ,1,1,1);
(%o5)                                 derivative
(%o6)                                     2y
(%o7)                                     x
(%o8)                                     *
(%o9)                                     2
(%i14)   part(eq,2,0); part(eq,2,1); part(eq,2,2); part(eq,2,2,0); part(eq
   ,2,2,1);
(%o10)                                    +
(%o11)                                   ay
                                        b + 2
(%o12)                                  -----
                                          x
(%o13)                                    /
(%o14)                                   b+2
```

If the last index is a list of indices, then a subexpression is returned which is made up of multiple operands (subexpressions or atoms) of the last operator, each index in the list standing for one. These operands are combined by their operator in the expression.

```
(%i1)   part(2*(x+y+z),2,[1,3]);
(%o1)                                    x+z
```

Function *part* can also be used to obtain an element of a list, a row of a matrix, etc.

9.6.2 Substitute subexpression

substpart (repl, expr, $n_1, \ldots, n_{k-1}, (n_k \,|\, [n_{k1}, \ldots, n_{kl}])$*)* [function]

In *expr* the replacement *repl* is substituted for the subexpression

$$part\ (expr,\ n_1, \ldots, n_{k-1}, (n_k \,|\, [n_{k1}, \ldots, n_{kl}])),$$

and the new value of *expr* is returned. *repl* may be some operator to be substituted for an operator of *expr*. In some cases *repl* needs to be enclosed in double-quotes " (e.g. substpart ("+", a*b, 0) yields b + a).

9.7 Manipulate expression

9.7.1 Substitute pattern

9.7.1.1 subst: substitute explicite pattern

subst (new, old, expr) [function]
subst (old=new, expr)
subst ([eq_1,. . . ,eq_k], expr)

This function substitutes *new* for *old* everywhere in *expr*. *old* must be an atom or a complete subexpression of *expr*. *subst(old=new, expr)* is equivalent to *subst(new,*

old, expr). *eq_i* are equations of the form *old=new* indicating multiple substitutions to be done in *expr*, carried out in serial. See example after *sublis*.

As usual, all arguments are evaluated and can be quoted or, if necessary to enforce evaluation, even double quoted, see example. This function allows for substituting values (numbers or symbolic expressions) for symbols or vice versa.

If *old* or *new* are to be single-character operators, they must be enclosed in double-quotes. Note, however, that, due to the differences between UVF and GIF, replacing operators in an expression often does not yield the expected result and should be avoided.

In any case, if the result of *subst* is not what was expected, one should take a look at the GIF of the expression in order to find an explanation. Although *subst* is implemented on the basis of Maxima's rules and patterns mechanism, which itself works strictly on the basis of GIF, some concessions have been made to the UVF in *subst*. In general, any subexpression which can be selected by *part* and which is complete can be replaced. For instance, in $1/\sqrt{(x)}$ the subexpression $\sqrt{(x)}$ can be replaced, although in GIF this subexpression does not exist. See sect. 14.5.1 for an alternative to *subst* working directly with rules and patterns, and therefore strictly on the basis of GIF. You will see that this alternative is more powerful than *subst*. The most powerful tool for substituting mathematical patterns, however, is *ratsubst*.

```
(%i1)   expr:x*a+(x*b)/2;
```

$$\frac{bx}{2}+ax \tag{\%o1}$$

```
(%i2)   e:c+d$
(%i3)   x:'e$
(%i4)   expr;
```

$$\frac{bx}{2}+ax \tag{\%o4}$$

```
(%i5)   ev(expr);
```

$$\frac{be}{2}+ae \tag{\%o5}$$

```
(%i6)   ev(expr,eval);
```

$$\frac{b\,(d+c)}{2}+a\,(d+c) \tag{\%o6}$$

```
(%i8)   subst(''x,'x,expr);
```

$$\frac{b\,(d+c)}{2}+a\,(d+c) \tag{\%o8}$$

```
(%i9)   subst(x,''x,%);
```

$$\frac{be}{2}+ae \tag{\%o9}$$

```
(%i7)   subst('x,x,%);
```

$$\frac{bx}{2}+ax \tag{\%o7}$$

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
```

9.7.1.2 ratsubst: substitute implicit mathematical pattern

ratsubst (new, old, expr) [function]

Just like *subst* this function substitutes *new* for *old* everywhere in *expr*. But in contrast to *subst*, all three arguments have to be mathematical expressions. *old* does not have to be an atom or a complete subexpression of *expr*, it does not even have to be a subexpression explicitely visible in *expr*. In general, *ratsubst* can substitute any subexpression which could be made explicit by any kind of equivalence transformation of *expr*.

For instance, *old* can be a subexpression visible only in *expand(expr)* or, vice versa, only in *factor(expr)*. If *radsubstflag* is *true*, *old* can be a root, which is not explicit in *expr*, but which could be made explicit by an equivalence transformation. To illustrate this, we give an easy alternative to the example given in sect. 14.5.1.

```
(%i1)   radsubstflag:true$
(%i2)   ratsubst(b,sqrt(x),x);
(%o2)                                b^2
(%i3)   ratsubst(b,sqrt(x),x^(-3/2));
                                     1
(%o3)                                --
                                      3
                                     b
```

radsubstflag default: *false* [option variable]

When *true*, this flag allows *ratsubst* to substitute roots, which are not explicit in *expr*, see example above.

9.7.2 Box and rembox

box (expr) [function]
rembox (expr) [function]

If an expression *expr* is the argument of function *box*, it is called a *boxed* expression. A boxed expression does not evaluate to its content, so it is effectively excluded from computations. However, *box* evaluates its argument. The return value is an expression with *box* as the operator and expr as the argument. *rembox* removes all boxes within its argument, so any boxed expressions it contains are evaluated. Thus, the box-rembox mechanism is useful for temporarily excluding parts of an expression from being evaluated. See function PullFactorOut for an example.

```
(%i1)   a:2$  box(a);
(%o1)                                2
(%i2)   box(a)*3;
(%o2)                               3(2)
(%i3)   part(box(a),0);
(%o3)                               box
(%i4)   part(box(a),1);
(%o4)                                2
(%i5)   %*3;
(%o5)                                6
(%i6)   rembox(a*box(a)*box(4));
(%o6)                                16
```

Chapter 10

Operators

10.1 Defining and using operators

10.1.1 Function notation of an operator

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

10.1.2 Miscellaneous

Any prefix operator can be used with or without parentheses: not a <=> not(a) -a <=> -(a)

10.2 System defined operators

10.2.1 Identity operators and functions

Note. : and *::* are the assignment operators.

10.2.1.1 Equation operator

= [infix operator]

This is the *equation operator*. Chains like a = b = c are not allowed. See sect. 9.5 for the exceptional case where the equation operator can be omitted.

When Maxima encounters an equation, its arguments, which means the lhs and the rhs, 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.

Only is*(a = b)* and some other functions, namely *if*, *while*, *unless*, *and*, *or*, and *not*, will evaluate the equation *a = b* to *true* or *false*.

Assumptions of equality cannot be specified with the = operator, only with function *equal*.

```
(%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 rhs and lhs, 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;
```

10.2.1.2 Inequation operator

[infix operator]

The negation of = is represented by #, which is the *inequation operator*. Just like for an equation, only the lhs and rhs will be evaluated separately, the returned expression constitutes an *unevaluated inequation*.

Only *is(a # b)* and the other functions mentioned above will evaluate the inequation *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*.

10.2.1.3 equal, notequal

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

These functions by themselves, like = and #, do nothing more than evaluate both arguments separately. 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. Instead, 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.

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))*.

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
```

10.2.1.4 is, is(a=b), is(equal(a,b))

is (expr) [function]

Function *is* evaluates an (in)equation, a relation, or a function call of equal or nonequal 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
```

10.2.2 Relational operators

< [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));
```

(%o3)
$$-\frac{x+1}{(t+1)^2}$$

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*.

10.2.3 Logical (Boolean) operators

Chapter 11

Evaluation

11.1 Introduction to evaluation

For a general overview of the role and philosophy of the evaluator in a CAS system and for a comparison of its implementation in various existing CAS systems see Richard Fateman's paper from 1996, which can also be found on his homepage in a revised version from 1999. [**FatemEv**] [**FatemEvR**]

Evaluation in Maxima means dereferencing. If a symbol is bound, i.e. if it has a value, or as we say *refers* to a value, then evaluation of a symbol means retrieving this value. Evaluation is not to be confused with *simplification*.

A symbol which is not bound evaluates to itself.

```
(%i1)   a;
(%o1)                                a
```

Consider the following example where a symbol a is bound to another symbol b which itself has a value c. Usually Maxima will evaluate a only once, retrieving b. In order to retrieve the value of b from a, we have to explicitly make Maxima evaluate a again. To achieve such multiple evaluation, a function like *ev* has to be called. Of course, we can also assign to a the value obtained from ev(a). Then a refers directly to c without the detour over b.

```
(%i1)   a:b;
(%o1)                                b
(%i2)   b:c;
(%o2)                                c
(%i3)   a;
(%o3)                                b
(%i4)   ev(a);
(%o4)                                c
(%i5)   a:ev(a);
(%o5)                                c
(%i6)   a;
(%o6)                                c
```

11.1.1 Stavros' warning note about *ev* and quote-quote

Mail from 3.11.2017 to maxima-discuss: NO NO NO NO NO

Double-quote does not "force evaluation" — it substitutes a value at *read time*. It is a handy shortcut in interactive use, but I urge you to avoid it in general. For one thing, you can't prototype a calculation interactively and then package it up as a function if you use ' '(...).

ev is another convenience function with some surprising behavior. In particular, $ev(\ldots, x = 1)$ is *not* equivalent to $block([x : 1], \ldots)$. For example, $ev(diff(x,'x), x = 1)$ gives an error, while $block([x : 1], diff(x,'x))$ gives 0. Since it performs an evaluation, it also risks free-variable capture in programs (again, a problem when you package up an interactive prototype). I always urge people to avoid ev as much as possible. It is always cleaner and clearer to use subst rather than ev.

Trying to get *ev* to do what you want by clever use of '(...) or ' '(...) is a fool's errand: you may get it to work in one case, but you will quickly find cases where it isn't quite right.

To recap:

ev(...) and ' '(...) are handy hacks, but have *peculiar semantics* and are best to avoid.

11.2 Function ev

ev (expr $\langle, arg_1, \ldots, arg_n\rangle$*)* | [function]
expr, $arg_1\langle, arg_2, \ldots, arg_n\rangle$

Function *ev* evaluates the expression *expr* in the environment specified by the arguments $arg_1, \ldots, arg_n$. These arguments are switches (Boolean flags), assignments, equations, and functions from the list given below. *ev* returns the result (another expression) of the evaluation.

An alternate top level syntax has been provided for *ev*, whereby one may just type in the expression and its arguments separated by commas. This is not permitted as part of another expression, e.g., in functions, blocks, etc. For an example see sect. 24.2.1.2.2.

The evaluation is carried out in steps, as follows.

1. First the environment is set up by scanning the arguments which may be any or all of the following.

- *simp* causes *expr* to be simplified regardless of the setting of the switch *simp* which inhibits simplification if *false*.
- *noeval* suppresses the evaluation phase of *ev* (see step (4) below). This is useful in conjunction with the other switches and in causing *expr* to be resimplified without being reevaluated.
- *nouns* causes the evaluation of noun forms (e.g. unevaluated user-defined function calls, functions such as *'integrate* or *'diff*, functions previously used undeclared which now have been declared) in *expr*.

  See the example to *gradef*.

- *expand* causes expansion.
- *expand (m, n)* causes expansion, setting the values of *maxposex* and *maxnegex* to m and n respectively.
- *detout* causes any matrix inverses computed in *expr* to have their determinant kept outside of the inverse rather than dividing through each element.
- *diff* causes all differentiations indicated in *expr* to be performed.
- *derivlist (x, y, z, . . .)* causes only differentiations with respect to the indicated variables.
- *risch* causes integrals in *expr* to be evaluated using the Risch algorithm. The standard integration routine is invoked when using the special symbol *nouns*.
- *float* causes non-integral rational numbers to be converted to floating point.
- *numer* causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in *expr* which have been given *numervals* to be replaced by their values. It also sets the *float* switch on.
- *pred* causes predicates (expressions which evaluate to *true* or *false*) to be evaluated.
- *eval* causes an extra post-evaluation of *expr* to occur. (See step (5) below.) *eval* may occur multiple times. For each instance of *eval*, the expression is evaluated again.
- A where A is an atom declared to be an evaluation flag. *evflag* causes A to be bound to true during the evaluation of *expr*.
- *V: expression*, or alternately *V=expression* causes V to be bound to the value of expression during the evaluation of expr. Note that if V is a Maxima option, then expression is used for its value during the evaluation of *expr*. If more than one argument to *ev* is of this type, then the binding is done in parallel. If V is a non-atomic expression, then a substitution rather than a binding is performed.
- F where F, a function name, has been declared to be an evaluation function. *evfun* causes F to be applied to *expr*.
- Any other function names, e.g. *sum*, cause evaluation of occurrences of those names in *expr* as though they were verbs.

  See example of *gradef*.
- In addition, a function occurring in *expr*, e.g. F(x), may be defined locally for the purpose of this evaluation of *expr* by giving *F(x) := expression* as an argument to *ev*.
- If an atom not mentioned above or a subscripted variable or subscripted expression is given as an argument, it is evaluated and if the result is an equation

or assignment, then the indicated binding or substitution is performed. If the result is a list, then the elements of the list are treated as though they were additional arguments given to *ev*. This permits a list of equations to be given (e.g. [X=1, Y=A**2]), or a list of names of equations (e.g., [%t1, %t2] where %t1 and %t2 are equations) such as returned by solve.

The arguments of *ev* may be given in any order with the exception of substitution equations which are handled in sequence, left to right, and evaluation functions which are composed, e.g., *ev (expr, ratsimp, realpart)* is handled as *realpart (ratsimp (expr))*. The *simp*, *numer*, and *float* switches may also be set locally in a block, or globally in Maxima so that they will remain in effect until being reset. If *expr* is a canonical rational expression (CRE), then the expression returned by *ev* is also a CRE, provided the numer and float switches are not both true.

2. During step (1), a list is made of the non-subscripted variables appearing on the left side of equations in the arguments or in the value of some arguments if the value is an equation. The variables (subscripted variables which do not have associated array functions as well as non-subscripted variables) in the expression expr are replaced by their global values, except for those appearing in this list. Usually, expr is just a label or % (as in %i2 in the example below), so this step simply retrieves the expression named by the label, so that *ev* may work on it.

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

4. The resulting expression is then re-evaluated (unless one of the arguments was *noeval*) and simplified according to the arguments. Note that any function calls in expr will be carried out after the variables in it are evaluated and that *ev(F(x))* thus may behave like *F(ev(x))*.

5. For each instance of *eval* in the arguments, steps (3) and (4) are repeated.

11.3 Quote-quote operator ''

''*expr* [prefix operator]

The quote-quote operator '' (two single quote marks) modifies evaluation in input expressions. Applied to a general expression *expr*, quote-quote causes the value of expr to be substituted for *expr* in the input expression. Applied to the operator of an expression, quote-quote changes the operator from a noun to a verb (if it is not already a verb). The quote-quote operator is applied by the input parser; it is not stored as part of a parsed input expression. The quote-quote operator is always applied as soon as it is parsed, and cannot be quoted. Thus quote-quote causes evaluation when evaluation is otherwise suppressed, such as in function definitions, lambda expressions, and expressions quoted by single quote '.

Quote-quote is recognized by batch and load.

11.4 Substitution

Maxima has three different functions which carry out substitutions: *ev*, *at*, and *subst*.

11.4.1 Substituting values for variables

at (*expr* | [$expr_1, \ldots, expr_n$]), (*eqn* | [$eqn_1, \ldots, eqn_n$]) [function]

Evaluates *expr* or the expressions in the list with variables assuming values as specified in *eqn* or the list of equations. *at* carries out multiple substitutions in parallel. Depending on the first argument, *at* returns a single expression or a list of expressions.

Note that values do not necessarily mean numerical values. Symbols and even expressions can also be substituted for symbols. (Substituting expressions for expressions sometimes is possible, sometimes not. Anyway, this use of *at* is discouraged.)

In particular, *at* allows to indicate that the derivative of an unspecified function is to be evaluated at a certain point.

```
(%i1)   at(x^2,x=x0);
(%o1)                                x0^2
(%i2)   at(f(x),x=x0);
(%o2)                               f(x0)
(%i3)   at(diff(f(x),x),x=x0);
                                    d
(%o3)                               -- f(x)|
                                    dx     |x=x0
```

Chapter 12

Simplification

12.1 Properties for simplification

12.2 General simplification

12.2.1 Conversion between (complex) exponentials and circular/hyperbolic functions

exponentialize (expr) [function]
exponentialize *default: false* [option variable]

The function *exponentialize* converts circular and hyperbolic functions in *expr* to equivalent (complex) exponentials. This is useful for instance to be able to apply *solve* to *expr*, see sect. 18.2.2.

If the option variable *exponentialize* is *true*, all circular and hyperbolic functions encountered in the course of the following computations will be converted to (complex) exponentials; so in this case it is not necessary any more to apply the function to any expression. Flags *exponentialize* and *demoivre* cannot both be true at the same time.

demoivre (expr) [function]
demoivre *default: false* [option variable]

The function *demoivre* converts (complex) exponentials in *expr* to equivalent circular functions. Note that while exponentialize is also capable of converting hyperbolic functions to their exponential equivalents, demoivre is not capable of the inverse.

If the option variable *demoivre* is *true*, Maxima will try to convert all (complex) exponentials encountered in the course of the following computations to circular functions; so in this case it is not necessary any more to apply the function to any expression. Flags *demoivre* and *exponentialize* cannot both be true at the same time.

```
(%i1)  exponentialize(2*cos(s)+sin(s/2));
```

$$(\%o1) \qquad \%e^{\%is} - \frac{\%i\left(\%e^{\frac{\%is}{2}} - \%e^{-\frac{\%is}{2}}\right)}{2} + \%e^{-\%is}$$

```
(%i2)    demoivre(%);
```

$$(\%o2) \qquad 2\cos(s) + \sin\left(\frac{s}{2}\right)$$

```
(%i3)    exponentialize(tanh(s));
```

$$(\%o3) \qquad \frac{\%e^{s} - \%e^{-s}}{\%e^{s} + \%e^{-s}}$$

```
(%i4)    demoivre(%);
```

$$(\%o4) \qquad \frac{\%e^{s} - \%e^{-s}}{\%e^{s} + \%e^{-s}}$$

See sect. 18.2.2 for an application of *exponentialize*.

12.3 Trigonometric simplification

trigsimp kann nicht mit Komma nachgestellt werden.

12.4 Own simplification functions

12.4.1 Apply2Part

Apply2Part ((*'function()* | *'λ-expr()*), *expr* ⟨, $i_1, \ldots, i_n$⟨, ([$j_1, \ldots, j_l$] | *allbut*($j_1, \ldots, j_l$))⟩⟩)
.
[function of *rs_simplification*]

Selectively applies *function*, which must be quoted[1] and followed by parentheses (either empty or with additional arguments to *function*[2]), to the part of *expr*, which is specified by the following arguments in the way of the arguments of function *part*. As in *part*, a list of selected terms can be specified as the last argument, or the construction with *allbut* can be used. The complete *expr* with the substitution accomplished is returned.

A lamba expression can be specified instead of a symbol for *function*. Again it must be quoted and followed by parentheses (either empty or with additional arguments).

Apply2Part can be used e.g. with functions *factor*, *expand*, *ratexpand* (which brings terms to a common denominator), or *trigsimp*. Note that *PullFactorOut* has this functionality already built in, so its combination with Apply2Part is unnecessary.

As an example, *function* being set to *factor* allows to selectively factor a part (or even specific terms from a sum) anywhere in an expression. This constructs and evaluates an expression of the form *substpart(factor(part(expr, indices)),expr, indices)*, where the last index can be a list of the terms of a sum to be factored. The factor itself is specified only implicitely by this selection.

[1] *ratexpand* for instance is also a flag which by default evaluates to *false*.
[2] E.g. *partfrac* needs a second argument.

```
(%i1)   expr: f(t)=a*x+b*y+c*x*y+d*y;
(%o1)                          f(t)=cxy+dy+by+ax
(%i2)   Apply2Part('factor(),expr,2,[1,2,3]);
(%o2)                          f(t)=(cx+d+b)y+ax
(%i3)   Apply2Part('factor(),expr,2,[1,4]);
(%o3)                          f(t)=x(cy+a)+dy+by
(%i4)   Apply2Part('factor(),%,2,[2,3]);
(%o4)                          f(t)=x(cy+a)+(d+b)y
(%i5)   Apply2Part('lambda([x],x^2)(),%,2,1,1);
(%o5)                          f(t)=x^2(cy+a)+(d+b)y
```

12.4.2 ChangeSign

ChangeSign (expr⟨, $i_1, \ldots, i_n$⟩) [function of *rs_simplification*]

Changes the sign of *expr* or the subexpression *expr*, $i_1, \ldots, i_n$ specified as in function *part*. By applying this function to an expression twice, the minus sign can be switched between two places, e.g. two factors, between numerator and denominator of a fraction, or between a product as a whole and one of its factors. The inner function call should be tested alone before being wrapped by the second call, because the canonical order may be changed by the inner call. Note that the operator of the sum $a-b$ is "+", with the operator of $-b$ being "−" (b can be a subexpression).

```
(%i1)   expr:-((a-b)/(c-d));
```

$$(\%o1) \qquad -\frac{a-b}{c-d}$$

```
(%i2)   ChangeSign(ChangeSign(expr,1,1),1,2);
```

$$(\%o2) \qquad -\frac{b-a}{d-c}$$

```
(%i3)   expr:f=('diff(a,x)+((a+b)/(c-d)))/(h+j+log(d));
```

$$(\%o3) \qquad f=\frac{\frac{b+a}{c-d}+\frac{d}{dx}a}{j+h+\log(d)}$$

```
(%i4)   ChangeSign(ChangeSign(expr,2,1,1,1),2,1,1);
```

$$(\%o4) \qquad f=\frac{\frac{d}{dx}a-\frac{-b-a}{c-d}}{j+h+\log(d)}$$

```
(%i5)   expr:-s*(-a+b)*(-c+d);
(%o5)                          -(-a+b)*(-c+d)*s
(%i6)   ChangeSign(ChangeSign(expr),2);
(%o6)                          (-a+b)*(c-d)*s
```

12.4.3 FactorTerms

FactorTerms ([$fac_1, \ldots, fac_m$], expr ⟨, $i_1, \ldots, i_n$⟩) [function of *rs_simplification*]

Factors out the factors $fac_1, \ldots, fac_n$ from the terms where they appear in the subexpression specified by *expr*, $i_1, \ldots, i_n$. This function uses *ratcoeff*. Not every term of the specified subexpression needs to contain one of the given factors. But *FactorTerms* can't work properly, if a term of the subexpression contains more than

one of the given factors. In this case, the result will equal the given subexpression, which can be shown by expanding it again, but the desired factoring is impossible. So in this case *Apply2Part(factor)* has to be used instead, which allows selecting terms individually; here the specific factor to be factored out of the multi-factor term is specified implicitely.

The complete *expr* with the substitution accomplished is returned.

```
(%i1)   expr: y=a*3+b*4+b*c+d*a+e$
(%i2)   FactorTerms([a,b],expr,2);
(%o2)                          y=e+a*(d+3)+b*(c+4)
```

12.4.4 PullFactorOut

PullFactorOut ((*expr* | *'part (expr, i*$_1$*, ..., i*$_n$⟨*,* ([*j*$_1$*, ..., j*$_l$] | *allbut(j*$_1$*, ..., j*$_l$)))⟩)⟨*, factor*⟩*)*
.
[function of *rs_simplification*]

PullFactorOut2 (expr ⟨*, factor*⟩*)*
[function of *rs_simplification*]

Pulls *factor* out of *expr* and wraps it in a box which normally preceeds the remainder. *expr* can be a product, fraction, sum, list, vector or matrix. If the remainder is a fraction, a list or a matrix, it is placed in a second box in order for the first box not to be pulled into the numerator or each element of the list or matrix. If no factor is specified, the gcd is determined and pulled out. However, this does not make sense and does not work for products or fractions.

If part of an *expr* is specified as in function *part* (note that this has to be quoted here), only this part will be factored, but the whole *expr* will be returned. Thus, combination of *PullFactorOut* with *Apply2Part* is not necessary. Giving the last parameter as a list of selected terms or using *allbut* to exclude selected terms as in function *part* is also possible, see "*'Kugel rollte im Hohlkegel.wxm*"'.

PullFactorOut2 is an experimental version with the same functionality, but not requiring to put a box around the pulled out factor, unless -1 is pulled out of a matrix.

```
(%i1)   v:r*CVect(cos(t),sin(t));
```

$$\begin{pmatrix} r\cos(t) \\ r\sin(t) \end{pmatrix}$$

```
(%i2)   PullFactorOut(v,r);
```

$$(r)\begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix}$$

```
(%i1)   g1:-3*a*b/2;
```

$$-\frac{3ab}{2}$$

```
(%i2)   PullFactorOut('part(g1,2),3/2);
```

$$-\left(\frac{3}{2}\right)ab$$

```
(%i3)   g2:Fn=p+(a-(3*a*b/(2*c*d)))/(d+g);
```

(%o3) $$Fn = p + \frac{a - \frac{3ab}{2cd}}{g + d}$$

```
(%i4)  PullFactorOut('part(g2,2,2,1,2),-3*a/2);
```

(%o4) $$Fn = p + \frac{\left(-\frac{3a}{2}\right)\left(\frac{b}{cd}\right) + a}{g + d}$$

ElimCommon (equ) [function of *rs_simplification*]

On a recursive basis this function eliminates common factors and/or common terms from both sides of the equation *equ*.

This function was written by Stavros Macrakis, 2016. It employs the global function ElimCommonTerms.

Chapter 13

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.

13.1 Facts and contexts: The general system

13.1.1 User interface

13.1.1.1 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.

13.1.1.2 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*.

13.1.2 Implementation

13.1.2.1 Internal data structure

13.1.2.2 Notes on the program code

13.2 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.

13.3 MaximaL Properties

13.3.1 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* properties can be declared for a symbol only by the system (but they can be removed by the user),
- *User-declared* (sometimes also called: *system-defined* or *predefined*) properties are predefined properties which the user can declare for a symbol or remove from it,
- *User-defined* properties can be defined by the user and then be declared for a symbol or removed from it.

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.

13.3.2 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.

13.3.3 User-declared properties

These are pre-defined properties, which the user can assign to a variable or user-defined function or remove from it. Properties are recognized by the simplifier and other Maxima functions. There are general (*featurep*) and specific (e.g. *constantp*) predicate functions which can test a certain symbol for having a specific user-declared or user-defined property or not.

13.3.3.1 Declaration, information, removal

declare $(([a_1 \mid [a_{11}, \ldots, a_{1k}]), ([p_1 \mid [a_{11}, \ldots, a_{1l}]), \ldots, ([a_n \mid [\ldots]), ([p_n \mid [\ldots]))$ [function]

Assigns property (or list of properties) p_j to symbol (or list of symbols) a_j, $j = 1, \ldots, n$. Symbols 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. 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 symbol *a*. This includes system properties and properties having been previously defined by the user.

props [system variable]

The system variable contains a list of all symbols that have been assigned any user-declared or user-defined property.

propvars (p) [function]

Returns a list of all symbols on the system list *props* which have property *p*.

remove $(([a_1 \mid [a_{11}, \ldots, a_{1k}]), ([p_1 \mid [a_{11}, \ldots, a_{1l}]), \ldots, ([a_n \mid [\ldots]), ([p_n \mid [\ldots]))$ |
remove (all,p) [function]

Removes property (or list of properties) p_j from symbol (or list of symbols) a_j, $j = 1, \ldots, n$. *remove (all, p)* removes 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*.

13.3.3.2 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 rational numbers as displayed with /R/), symbolic constants such as %pi, %e, or %i, variables bound to a constant or declared *constant* by declare, or functions whose arguments are constant. *constantp* evaluates its arguments. See the property *constant* which declares a symbol to be constant.

scalar [property]
nonscalar [property]

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

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

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

nonarray [property]

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

13.3.3.3 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.

13.3.4 User-defined properties

The user may define new properties and assign them to variables or user-defined functions with *declare* in the same way it is done for predefined, user-declared properties. 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 or not.

declare (p_u,feature) [function]

Declares p_u to be a new property. It can then be assigned to variables or user-defined functions, tested for, view in lists, or removed. User-written functions can consider this property.

```
(%i1)   declare(new_property, feature)$
(%i2)   declare(a, new_property)%
(%i3)   properties(a);
(%o3)                [database info,kind(a,new_property)]
(%i4)   featurep(a,new_property);
(%o4)                                true
(%i5)   a:b;
(%o5)                                 b
(%i6)   featurep(a,new_property);
(%o6)                                false
(%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 both 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.

13.3.5 Implementation

13.4 Assumptions

13.4.1 User interface

13.4.1.1 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*.

13.4.1.2 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
```

13.4.2 Implementation

Chapter 14

Patterns and rules

14.1 Introduction

This chapter describes *pattern matching* and *user-defined simplification rules*. Maxima's pattern matcher was written by Richard J. Fateman. His dissertation from 1971, entitled *Algebraic Simplification*, describes it together with other components of Macsyma which he had implemented. We recommend reading chapter 2, "The User-Level Semantic Matching Capability In MACSYMA", of this thesis, because it motivates why we want to use pattern matching in a CAS, and on what theoretical background Maxima's pattern matcher was designed. Repeatedly, when related question arose on maxima-discuss in the past, Richard took the time to explain the principles of Maxima's pattern matcher. So the archives of maxima-discuss constitute another valuable source of information in this respect. [**FatemThe**]

The very concise chapter on rules and patterns of the Maxima manual was written by Robert Dodier. Michel Talon recently contributed an introductory tutorial which focuses on special issues and potential problems in application and includes references to how the pattern matcher works on the Lisp level. [**TalonRP**]

There are two groups of functions which implement different pattern matching schemes. The first group comprises *defmatch*, *defrule*, *tellsimp*, *tellsimpafter*, *apply1, applyb1*, and *apply2*. To the second group belong *let* and *letsimp*. Both schemes define patterns in terms of *pattern variables* declared by *matchdeclare*. Pattern-matching rules defined by *tellsimp* and *tellsimpafter* are applied automatically by the Maxima simplifier, while rules defined by *defmatch*, *defrule*, and *let* are applied by an explicit function call.

There are additional mechanisms for rules applied to polynomials by *tellrat*, and for commutative and noncommutative algebra in the *affine* package.

14.1.1 What pattern matching is and how it works in Maxima

Pattern matching means taking an arbitrary expression as input and comparing it (as a whole or in parts) with a *pattern* previously defined. Some pattern matching functions (e.g. *defmatch*) just inform the user about whether a given expression or subexpression *matches* the pattern or not, and in case of a positive match, how the *pattern variables* used to define the pattern are matched by parts of the given expression. Other pattern matching functions (e.g. *defrule*, *tellsimp*, *tellsimpafter*)

will, in case of a positive match, also replace the matching expression or subexpression with some *replacement* expression, that is, they will modify the original expression.

In combination with functions which can decompose a given expression into all of its subexpressions (*apply1, applyb1, apply2*), pattern matching functions can compare a defined pattern with all subexpressions on all levels of a given expression. The replacement can then be done to all subexpressions which match the pattern. This constitutes a very powerful mechanism which allows to modify or simplify the given expression according to certain *rules*. Such rules are nothing more than a combination of a pattern and a corresponding replacement.

Pattern matching is done in several steps. First we have to define a pattern. This is done with the help of *pattern variables*. So actually, defining the pattern variables is the very first step. This is done with *matchdeclare*. The definition of the actual pattern is done in the next step, when we create a function which can test a given expression for whether it matches the pattern (and how) or not. Creating this function and defining the pattern is done with *defmatch*. *defmatch* not only uses pattern variables, but also *pattern parameters* to define the pattern. Alternatively, we can define a function which substitutes an expression matching the pattern with a *replacement* expression. This is done with *defrule*. Thus, *defrule* not only defines a pattern, but a complete *rule* consisting of a pattern and the corresponding replacement to be carried out in case of a positive match.

Both *defmatch* and *defrule* create a *match function* which can be called explicitely by the user. Calling this function with an *actual expression* (an expression to be tested for whether it matches the pattern or not) as an argument forms the third step in pattern matching. If we want to apply our match function to all subexpressions of the actual expression, we have to wrap it in *apply1, applyb1*, or *apply2* before calling it. However, it is also possible to make the simplifier use our newly defined rule automatically for any expression (and any of its subexpressions) which is being simplified by the system. Depending on whether our new rule is to be used before or after the system simplification rules, its definition is done with *tellsimp* or *tellsimpafter*.

14.1.1.1 Pattern, pattern variable, pattern parameter, match

A *pattern* is a kind of *template expression* comprising both fixed elements, which have to match exactly with the corresponding parts of an actual expression (although, if they are symbols introduced by the user, they might be evaluated), and variable elements, which can be *pattern variables*, each having a specific variability determined by a condition associated with it, the so-called *match predicate*, or *pattern parameters*. An atom or subexpression of the actual expression *matches* a pattern variable, if it satisfies its respective match predicate. It matches a pattern parameter, if it is identical with the corresponding pattern argument.

Pattern variables, also called *match variables*, are the most important element used for the definition of patterns. Each pattern variable is associated with a condition which allows to determine whether a subexpression (an atom is also a subexpression) of the actual expression is able to match this variable or not. This condition is

defined in the form of a *predicate*, a Boolean function returning either *true* or *false*. More than one pattern variable can occur in a pattern, and a pattern variable can occor more than once.

A pattern is an expression in which pattern variables occur together with pattern parameters, other symbols introduced by the user, numbers, operators, or function calls. If a pattern is to match an actual expression, all pattern variables occuring in the pattern and all of their occurrences have to match a subexpression of the actual expression; all pattern parameters, if any, have to match a symbol identical with the pattern argument; and all other elements of the pattern have to literally match a counterpart of the actual expression (with the exception of a possible evaluation of any symbol introduced by the user). Only if all of this is fulfilled and nothing of the actual expression is left over, the pattern matches as a whole. In this case, every pattern variable will be bound (i.e. assigned as its value) to the corresponding subexpression of the actual expression.

14.1.1.2 No backtracking

Maxima's pattern matcher works *without backtracking*. This seemingly harmless little statement has to be considered with the utmost care in order to be able to successfully use pattern matching with Maxima.

To put it another way, Maxima's pattern matcher does not work according to the principle of *trial and error*. It does not try one way, and if it doesn't get to the end, try another way, and so on. If, during an attempted match, it does not succeed with matching a particular pattern variable, it won't go back and try it again under different considerations, by shuffling around matching variables and potential corresponding subexpressions of the actual expression. This means, Maxima's pattern matcher has a certain strategy and order in which to proceed with trying to match each element of the pattern, one after the other, each pattern variable and each of its possibly multiple occurrences, and the other elements of the pattern, and if this fails for any element of the pattern, this was it. So the user has to have some precise knowledge about this strategy and order when he wants to set up his pattern in a way so that it matches exactly with what it is supposed to match, nothing more and nothing less.

There are other CAS systems whose pattern matcher in fact do work according to the principle of backtracking. Such a pattern matcher runs through a large number of potential combinations of partial matches, trying again and again to achieve the global match. So it might preliminarily assign a subexpression to the first pattern variable, and then see whether the second pattern variable finds something adequate from what is left of the actual expression. If not, it will undo the assignment to the first pattern variable and try it with a different subexpression, hoping that under this new condition the second pattern variable will also find what it needs, and so on. While such a pattern matching scheme is probably easier for the user to handle, it can lead to exponential time cost. Maxima's pattern matcher, on the contrary, was designed to be efficient. But successfully working with it is much more challenging for the user. Nevertheless, we will show how it can be elegantly employed in solving problems.

14.1.1.3 The matching strategy in detail

Maxima's pattern matcher is more than just a literal matcher. It considers algebraic properties of expressions, for instance the commutativity of addition and multiplication.

The usual strategy of the matcher is to compare a given pattern variable, according to its match predicate, with all subexpressions of the actual expression, one after the other. The first subexpression it finds which satisfies the match predicate, it will *take*. Then the matcher goes to the next pattern variable and repeats the process with what is left from the actual expression. If at any point a pattern variable cannot find a matching counterpart, the global match fails. This implies that the order in which the pattern variables are compared against the subexpressions is important. Pattern variables are tested against subexpressions in the inverse order in which they appear in the pattern. If a subexpression of the actual expression satisfies the match predicate of more than one pattern variable, it will be assigned to the first pattern variable which *finds* it. If a pattern variable occurs more than once in a pattern, then of course what it takes must be identical for all occurrences (this is an extra condition in addition to the match predicate).

14.1.1.3.1 Peculiarities of addition and multiplication

Addition and multiplication are treated differently from other operands. In case of the subexpression actually under consideration being a sum or a product, a pattern variable may not only take one term or factor which fits, but it may take multiple terms or factors fitting. In fact, if all terms or all factors agree with its match predicate, a pattern variable will take the whole sum or product. This means, from a sum or a product a match variable will always take as much as it can, it is said to be *greedy* with respect to addition and multiplication.

This immediately leads to another important point. A pattern variable is also allowed to take "0" in case of a sum, or "1" in case of a product, if "0" resp. "1" agree with its specific match predicate. Together with what we said in the previous paragraph, this means:

If, for example, we have $a * b$ being part of a pattern, with pattern variables a and b, and this part of the pattern is compared with a subexpression $x * y$ of the actual expression, x and y being symbols, it does not necessarily mean that a will take x and b will take y, or vice versa. If both x and y fulfill the match predicates of both a and b, the first pattern variable to be compared against this subexpression, say it is b, will take $x * y$ and the second one, say it is a, will be left with "1". If the user wants a to take x and b to take y, he has to specify the corresponding match predicates in a way that x matches a, but not b, and y to match b, but not a. Only with such a specification he will be on the save side that the matcher will do what he wants. We see here already that match predicates always should be as specific as possible. An intended match will most probably not work correctly, if all match predicates are *true* or *all* (see *matchdeclare*).

14.1.1.3.2 The anchor principle

If a (part of a) pattern of the form $a*b+c*d$ with pattern variables a, b, c, d, that is a sum of subexpression being products, is to be compared with a subexpression $x*y+u*w$ of the actual expression, x, y, u, w possibly themselves being subexpressions, and we expect say a to match with x, we will most likely not be able to set up a correct matching scheme unless we employ the *anchor principle*.

In the above pattern, Maxima's pattern matcher can correctly determine whether a matches x only, if b and y are identical, or if at least the matcher can determine what y is, possibly with the help of some pattern parameter. The matcher needs to have an *anchor* for being able to match a with x, and this anchor is y. In fact, what the matcher simply does in this case, is to use *ratcoeff(x*y+u*w,y)*. This way it can determine the coefficient of y, which is x. x might well be a complicated expression consisting of multiple factors. But if the matcher does not know what y is, if y is unknown in the same way that x is unknown, it cannot apply *ratcoeff*, because it does not know what coefficient (the coefficient of what) to look for. In this case most likely the matcher (e.g. function *defmatch*) will issue a warning, saying that it cannot safely match the pattern specified by the user under the given conditions, i.e. the specifiactions of the pattern variables it contains.

We will give an example of the anchor principle in sect. 14.3.1, when we discuss *defmatch* and *defrule*.

14.2 Matchdeclare

matchdeclare $(\left(var_1 \middle| [var_{1_1}, \ldots, var_{1_{k_1}}]\right), pred_1, \ldots, var_n, pred_n)$ [function]

The arguments of *matchdeclare* are pairs $i = 1, \ldots, n$ consisting of a *pattern variable* var_i or a list $[var_{i_1}, \ldots, var_{i_{k_i}}]$ of pattern variables, and a *match predicate* $pred_i$. *matchdeclare* associates var_i or the corresponding list of pattern variables i with $pred_i$. See the introduction for the meaning of *pattern variable* and *match predicate*.

The functions *defmatch, defrule, tellsimp, tellsimpafter*, and *let* use pattern variables to construct patterns.

A *match predicate* is an unfinished function call or lambda call, in the sense that it lacks its last argument, or has no argument at all, if the function or lambda expression requires only one. In the first case, the list of arguments given in parentheses to the function call or lambda call lacks the last element. In the latter case, only the name of the function or only the lambda expression itself is given, with no argument (and no empty parentheses). A match predicate, however, can also be *true* or *all*. Here are some examples of valid match predicates.

```
(%i1)   matchdeclare (a, integerp)$
(%i2)   matchdeclare (b, lambda ([x], x > 0))$
(%i3)   matchdeclare (c, freeof (%e, %pi, %i))$
(%i4)   matchdeclare (d, lambda ([x, y], gcd (x, y) = 1) (1728))$
(%i5)   matchdeclare (e, true)$
(%i6)   matchdeclare (f, all)$
```

The missing argument will be supplied later, when the match predicate is evaluated. This will not be done before the *match function*, which will be defined by e.g. *defmatch* or *defrule*, is called to test an actual expression against the defined pattern containing the pattern variables we have just defined.

When a pattern containing a pattern variable is tested against an actual expression, the matcher will compare subexpressions of the actual expression with the predicate of the pattern variable, in order to find out whether this subexpression *matches* the pattern variable or not. If the predicate returns anything other than *false*, this particular subexpression is said to match the pattern variable and will be assigned to it as its value. If a replacement expression (e.g. in *defrule*) contains this pattern variable, it will be evaluated to this subexpression bound to it. See the introduction for how multiple pattern variables are matched and at what point the pattern matches as a whole.

When a pattern containing a pattern variable is tested against an actual expression, the subexpression to be tested against the particular pattern variable is appended to the list of arguments of the function call or lambda call of its match predicate, or, if it has no arguments yet, it is supplied as its sole argument. In any case, the tested subexpression completes the required number of arguments of the match predicate.

At this point it should be clear that a match predicate cannot simply be a relational or Boolean expression. Instead, it has to be wrapped in a function or lambda expression waiting for the particular subexpression to be its (last) argument. It is not necessary to call *is* to evaluate relational expressions within the match predicate. This will be done automatically when the match is attempted.

Any subexpression matches a match predicates which is defined as *true* or *all*. If the match predicate is a function, it need not be defined yet when *matchdeclare* is called, since the predicate is not evaluated until a match is attempted. *matchdeclare* quotes its arguments and always returns *done*.

If an subexpression satisfies a match predicate, the match variable is assigned this subexpression and nothing more. However, addition and multiplication are treated differently; other *nary* operators (both built-in and user-defined) are treated like ordinary functions. In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions. Such multiple-term matching is *greedy*, which means: predicates are evaluated in the order in which their associated variables appear in the pattern, and a term which satisfies more than one predicate is taken by the first predicate which it satisfies. A pattern variable’s predicate is tested against all operands of the sum or product before the next pattern variable’s predicate is evaluated. Furthermore, if "0" or "1" (respectively) satisfy a match predicate and there are no other terms which satisfy the predicate, "o" or "1" is assigned to the match variable associated with the predicate.

The algorithm for processing addition and multiplication patterns makes some match results (for example, a pattern in which a "match anything" variable appears) dependent on the ordering of terms in the match pattern and in the expression to be

matched. However, if all match predicates are mutually exclusive, the match result is insensitive to ordering, as one match predicate cannot accept terms matched by another. See the introduction for more explications.

Calling *matchdeclare* with a variable *var* as an argument changes the *matchdeclare property* of *var*, if one was already declared; only the most recent matchdeclare is in effect when a rule for *var* is defined. Later changes to the matchdeclare property of *var* (via matchdeclare or remove) do not affect already existing rules.

propvars (matchdeclare) returns the list of all variables for which there is a *matchdeclare* property. *printprops (var, matchdeclare)* returns the predicate for variable var. *printprops (all, matchdeclare)* returns the list of predicates for all match variables. *remove (var, matchdeclare)* removes the *matchdeclare* property from *var*.

14.3 Defmatch and defrule

defmatch (matchfunc, pattern$\langle, x_1, \ldots, x_n \rangle$*)* [function]

Defines a *match function* named *matchfunc(expr,* $x_1, \ldots, x_n$*)* which tests expr to see if it matches *pattern* while providing arguments $x_1, \ldots, x_n$ for the respective pattern parameters defined in *defmatch*. *pattern* is an expression containing the *pattern parameters* $x_1, \ldots, x_n$ (if any) and also pattern variables (if any), having been declared with *matchdeclare*. Any other symbol neither declared as a pattern variable in *matchdeclare* nor as a pattern parameter in *defmatch* only matches itself. However, it it is a symbol introduced by the user, it will be evaluated at the time the match is attempted.

The first argument to the created function *matchfunc* is an expression to be matched against the pattern. The other arguments of matchfunc are assigned to the pattern parameters of defmatch, which occur in *pattern*. Maxima evaluates and simplifies the argument of *matchfunc*.

If *matchfunc* is applied to an expression *expr* and the match is successful, *matchfunc* returns a list of equations whose left hand sides are the pattern parameters and pattern variables, and whose right hand sides are the subexpressions of *expr* which matched the pattern parameters and pattern variables. The pattern variables, but not the pattern parameters, are assigned the subexpressions they match. If the match fails, *matchfunc* returns *false*. A match function with a *literal pattern* (that is, a pattern which contains neither pattern parameters nor pattern variables) returns *true* if the match succeeds.

defmatch returns its first argument, which is the name of the newly defined match function.

In the following example we define a match function *linearp(expr)* which tests *expr* to see if it is of the form *a*x + b*, such that a and b do not contain x, and a is nonzero. Thus, this function matches expressions linear in x.

```
(%i1)   matchdeclare (a, lambda ([k], k#0 and freeof(x, k)), b, freeof(x))$
(%i2)   defmatch (linearp, a*x + b)$
(%i3)   linearp (3*x + (y + 1)*x + y^2);
```

```
(%o3)                          [b = y^2, a = y + 4]
(%i4)   linearp (3*z + (y + 1)*z + y^2);
(%o4)                                false
```

The first expression $3x+(y+1)x+y^2=(y+4)x+y^2$ is linear in x. The second one is linear in z, but not in x, so the match does not succeed. Note that *k#0* means $k \neq 0$, and that *freeof* is a function requiring two arguments; see *matchdeclare* for how the last and missing parameter is appended to a function call being a predicate. If we want to see whether an expression is linear in any variable, we have to introduce a pattern parameter in *defmatch*.

```
(%i5)   defmatch (linearp, a*x + b, x)$
(%i6)   linearp (3*z + (y + 1)*z + y^2, z);
(%o6)                     [b = y^2, a = y + 4, x = z]
(%i7)   a;  b;  x;
(%o7)                                y+4
(%o8)                                 y^2
(%o9)                                 x
```

Note that both defmatch and linearp now have two arguments. We specifically ask *linearp* for linearity in z. This is the case. The global pattern variables a and b have been assigned the matching subexpressions, while the pattern parameter x has not.

defrule (repfunc, pattern, replacement) [function]

Defines a *replacement function repfunc(expr)* which returns *replacement*, if *expr* matches *pattern*. Otherwise, if the match fails, *repfunc(expr)* returns *false*.

While a match function defined by *defmatch* only determines whether a given expression matches a pattern or not, and returns the values which have been assigned to the pattern variables, a replacement function, also called a *replacement rule*, *rule function* or simply a *rule*, determines whether the expression matches *pattern*, and in case of a match constructs the *replacement* with the actual values of the pattern variables. When all pattern variables occuring in the replacement have been assigned their actual values, the resulting expression is simplified.

defrule does not, in addition to pattern variables, support pattern parameters, as does *defmatch*.

If *repfunc* is applied to an expression by *apply1*, *applyb1*, or *apply2*, every subexpression matching the pattern will be replaced by *replacement*.

defrule returns the names of its parameters in the following form:
repfunc: *pattern -> replacement*.

14.3.1 Example: Rewriting an oscillation function

In sect. 24.2.1.1.1 and 24.2.1.2.2 we solved a linear second order ODE representing a free harmonic oscillator without damp, Satz 5.10. The result of the IVP was the time function in the general form

$$\varphi(t) = C_1 \sin(\omega t) + C_2 \cos(\omega t) \tag{14.1}$$

with the angular frequency ω depending on the particular ODE and the constants C_1, C_2 depending on the initial conditions. Note that the arguments of the sin and the cos are identical. Any expression like this, representing the superposition of two oscillations with the same frequency, but with different amplitudes and a phase shift between them, can be brought into the form of a single oscillation

$$\varphi = A\sin(\omega t + \alpha), \tag{14.2}$$

with the amplitude A and the phase constant α. The formulas are

$$A = \sqrt{C_1^2 + C_2^2} \qquad \text{and} \qquad \alpha = atan2(C_1, C_2). \tag{14.3}$$

Thus, we need the constants C_1, C_2 in order to compute this representation. How can we isolate these factors from the solution of the IVP as returned by *ic2*, if we take (%o5) from sect. 24.2.1.2.2 as an example? Of course, for a specific example, we can extract them manually. But suppose we want to perform this in an automated way, for instance as part of a bigger program dealing with a large number of such equations and IVPs. We can do this with Maxima's pattern matching. In demonstrating how, we will deliberately start with what is a rather intuitive approach, but turning out not to work properly.

```
(%i1)   matchdeclare([a,b,c],all)$
(%i2)   defmatch(m1,a*sin(c)+b*cos(c));
defmatch: a*sin(c) will be matched uniquely since sub-parts would otherwise
    be ambigious.
defmatch: cos(c)*b will be matched uniquely since sub-parts would otherwise
    be ambigious.
(%o2)                                m1
(%i3)   m1(sin(sqrt(g/l)*t)*sin(x)*cos(y)*s+cos(sqrt(g/l)*t)*cos(x));
```

$$(\%o3) \qquad [b = \cos\left(\sqrt{\frac{g}{l}}t\right), a = s\sin\left(\sqrt{\frac{g}{l}}t\right)\cos(y), c = x]$$

We got warnings from *defmatch*, which we ignored, because we did not understand them yet. We chose to test our match function *m1* with an expression, which contains other sines and cosines as part of the factors we want to extract. But *m1* messed it up: instead of selecting the sin and cos with the argument $\sqrt{g/l}\,t$ as the anchors, m1 took the sin and cos whose arguments are x. We want to improve our match predicates, knowing that our anchors must both contain in their arguments the factor t, and that no other sin or cos occuring in the expression can contain t.

```
(%i1)   matchdeclare([a,b],all,c,lambda([i],not(freeof(t,i))))$
(%i2)   defmatch(m1,a*sin(c)+b*cos(c));
defmatch: a*sin(c) will be matched uniquely since sub-parts would otherwise
    be ambigious.
defmatch: cos(c)*b will be matched uniquely since sub-parts would otherwise
    be ambigious.
(%o2)                                m1
(%i2)   m1(sin(sqrt(g/l)*t)*cos(x)*s+cos(sqrt(g/l)*t)*sin(x));
```

$$(\%o2) \qquad [b = \sin(x), a = s\cos(x), c = \sqrt{\frac{g}{l}}t]$$

```
(%i3)   m1(sin(sqrt(g/l)*t)*sin(x)*cos(y)*s+cos(sqrt(g/l)*t)*cos(x));
(%o2)                                false
```

The warnings are still there. But the first try of *m1* with a slightly less complicated expression than before looks promising: *m1* has computed the factors properly. However, the second try fails: *m1* does not find any match with the expression from above.

What went wrong? We have to read the introduction carefully again, in particular the section about the *anchor principle*. Then we realize, that what we are trying to do cannot succeed. It is impossible to match all three pattern variables in this one step, because for a and b we have no unambiguous anchor available: sin(c) and cos(c) cannot be identified in the actual expression by the matcher, because c also is unknown. And vice versa: the matcher cannot identify the anchor for finding c either, because a and b are unknown. In this situation the result from the matcher is unpredictable: it might by coincidence return a correct match in one situation, and it may just as well return an incorrect match in another one, but most likely it will not find any match, returning *false*. We just shouldn't have ignored the warnings.

So we have to start all over again and use an approach in two steps. First we need to find the anchor, and then with its help we determine the factors. We give the first step an intuitive try with a little function called *anchor*, which makes use of function *gatherargs* of the *opsubst* package. We collect and test all arguments from sin function calls appearing in the expression. If we do not find any one containing t, we do the same with the cos function calls. *anchor* will return the complete argument to what will be our sin and cos anchor, or 0 in case we did not find any, meaning that our oszillation function is the zero function.

```
(%i1)   load("opsubst")$
(%i2)   anchor(expr):=block([erg,g:0], local(expr,erg,g),
          erg: gatherargs(expr,sin),
          for i:1 thru length(erg) do
            if not(freeof(t,erg[i][1])) then g:erg[i][1],
          if g=0 then (
            erg: gatherargs(expr,cos),
            for i:1 thru length(erg) do
              if not(freeof(t,erg[i][1])) then g:erg[i][1]
          ),
          g
        )$
```

With the value returned from *anchor* we now go into the second step. Instead of declaring the argument of sin and cos as a match variable, we make it a match parameter. This way, defmatch issues no warning any more, and our match function properly isolates the factors a and b, even for complicated expressions.

```
(%i3)   matchdeclare([a,b],all)$
(%i4)   defmatch(m1,a*sin(anc)+b*cos(anc),anc);
(%o4)                                 m1
(%i5)   expr:sqrt(l/g)*sin(sqrt(g/l)*t)+cos(sqrt(g/l)*t);
```

```
(%o5)
```
$$\sqrt{\frac{l}{g}}\sin\left(\sqrt{\frac{g}{l}}t\right)+\cos\left(\sqrt{\frac{g}{l}}t\right)$$

```
(%i6)  an:anchor(%);
```

```
(%o6)
```
$$\sqrt{\frac{g}{l}}t$$

```
(%i7)  if an#0 then m1(expr,an) else (a:b:0,['a=0,'b=0]);
```

```
(%o7)
```
$$[b=1,a=\sqrt{\frac{l}{g}},anc=\sqrt{\frac{g}{l}}t]$$

Although we have solved the problem, we are not quite happy yet with the solution of step 1. There is a vage feeling that our Pascal-like loops are not the most elegant way of doing things in the Lisp world. Fortunately, Robert Dodier shows us how we can program it in a Lisp-like fashion employing pattern matching once more, this time with *defrule*.

```
(%i1)  matchdeclare(a,lambda([i],i=t),f,lambda([i],freeof(t,i)))$
(%i2)  defrule(r1,sin(a*f),(anc:a*f,sin(a*f)))$
(%i3)  defrule(r2,cos(a*f),(anc:a*f,cos(a*f)))$
(%i4)  anchor(expr):= block([anc:0],local(a,c,expr)),
         apply1(expr,r1,r2),
         anc
       )$
(%i5)  expr:sqrt(l/g)*sin(sqrt(g/l)*t)+cos(sqrt(g/l)*t);
```

```
(%o5)
```
$$\sqrt{\frac{l}{g}}\sin\left(\sqrt{\frac{g}{l}}t\right)+\cos\left(\sqrt{\frac{g}{l}}t\right)$$

```
(%i6)  an:anchor(%);
```

```
(%o6)
```
$$\sqrt{\frac{g}{l}}t$$

We do not really change anything when applying the rules r1, r2 to every subexpression of *expr* with *apply1*. We simply use a side-effect to make an assignment to *anc* when we have found the right subexpression. It does not matter, which one of the two rules does the assignment, it will be the first one (or the only one) finding the argument containing t. If *expr* contains both terms, the second rule, when having found the argument containing t, too, will overwrite *anc*; but this does not matter, since the arguments are always identical for both sin and cos. Note, that in the way we do the assignment ot *anc* we use *dynamic scoping*, not *lexical scoping*.

14.4 Tellsimp and tellsimpafter

tellsimp (pattern, replacement) [function]
tellsimpafter (pattern, replacement) [function]

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

pattern is an expression comprising *pattern variables*, declared by *matchdeclare*, as well as other atoms and operators, considered literals for the purpose of pattern matching. *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 nor "+" nor "*". The newly defined simplification rule is associated with *pattern*'s main operator, as it is done for the built-in simplification rules.

tellsimp/tellsimpafter does not evaluate its arguments, and it 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 names of functions (with one exception, described below), lists, and arrays may appear in pattern as the main operator only as literals, but not pattern variables. This excludes expressions like a(x) or b[y] as patterns, if a and b are pattern variables. Names of functions, lists, and arrays which are pattern variables may appear as operators other than the main operator in pattern. There is one exception to the above rule concerning names of functions. The name of a subscripted function in an expression such as a[x](y) may be a pattern variable, because the main operator is not a, but rather the Lisp atom *mqapply*. This is a consequence of the representation of expressions involving subscripted functions.

The rule constructed by *tellsimp/tellsimpafter* 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 MaximaL identifiers.

Rules defined with *tellsimp/tellsimpafter* 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/after 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.

Pattern variables are treated as local variables in simplification rules. Once a rule is defined, the value of a pattern variable does not affect the rule, and is not affected by the rule. An assignment to a pattern variable which results from a successful rule match does not affect the current assignment (or lack of it) of the pattern variable. However, as with all atoms in Maxima, the properties of pattern variables (as declared by *put* and related functions) are global.

The treatment of noun and verb forms is slightly confused. If a rule is defined for a noun (or verb) form and a rule for the corresponding verb (or noun) form already exists, the newly-defined rule applies to both forms (noun and verb). If a rule for the corresponding verb (or noun) form does not exist, the newly-defined rule applies only to the noun (or verb) form.

The rule constructed by tellsimpafter is an ordinary Lisp function. If the name of the rule is *$foorule1*, the construct : *lisp*(*trace* $*foorule*1) traces the function, and

: *lisp*(*symbol−function* '$*foorule*1) displays its definition.

remrule (op, rulename | all) [function]

Removes rules defined by tellsimp or tellsimpafter. *remrule (op, rulename)* removes the rule *rulename* from the operator *op*. When *op* is a built-in or user-defined operator (as defined by *infix*, *prefix*, etc.), *op* and *rulename* must be enclosed in double quotes. *remrule (op, all)* removes all rules from the operator *op*.

14.5 Apply1, applyb1, apply2

apply1 (expr, rule$_1$, . . . , rule$_n$) [function]

Repeatedly applies *rule*$_1$ to expr until it fails, then repeatedly applies the same rule to all subexpressions of expr, left to right, until *rule*$_1$ has failed on all subexpressions. Call the result of transforming expr in this manner *expr*$_2$. Then *rule*$_2$ is applied in the same fashion starting at the top of *expr*$_2$. When rule n fails on the final subexpression, the result is returned.

applyb1 (expr, rule$_1$, . . . , rule$_n$) [function]

This function is similar to *apply1* but works from the bottom up instead of from the top down. *applyb1* repeatedly applies *rule*$_1$ to the deepest subexpression of expr until it fails, then repeatedly applies the same rule one level higher (i.e., larger subexpressions), until *rule*$_1$ has failed on the top-level expression. Then *rule*$_2$ is applied in the same fashion to the result of *rule*$_1$. After *rule*$_n$ has been applied to the top-level expression, the result is returned.

apply2 (expr, rule$_1$, . . . , rule$_n$) [function]

If *rule*$_1$ fails on a given subexpression, then *rule*$_2$ is repeatedly applied, etc. Only if all rules fail on a given subexpression is the whole set of rules repeatedly applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with *rule*$_1$.

14.5.1 Example: substituting in an expression

The following example presents an alternative to *subst*, the substitution of a pattern whereever it occurs in an expression, which works strictly on the basis of the GIR representation of the expression. We will see that it is more powerful than using *subst*.

We want to substitute *sqrt*(*x*), which is $x^{1/2}$, by b in expressions containing subexpressions of type *sqrt*(*x*)^n, which is $x^{n/2}$, where n is an integer. This cannot easily be accomplished with *subst*, because the UVR of this subexpression depends on whether we have a simple square root, being represented as such, or a power of a square root, being represented in exponential notation. In GIR, however, the representation is always exponential, and therefore the substitution can be done in a uniform way. Only for the subexpression being x we need an additional, particular rule, because x is not represented in exponential form internally, but simply as x.

```
(%i1)   halfintegerp(r):=is(integerp(2*r))$
(%i2)   matchdeclare(half,halfintegerp)$
(%i3)   defrule(r1,x^half,b^(2*half))$
(%i4)   defrule(r2,x,b^2)$
(%i5)   apply1(sqrt(x)^3+x+sqrt(x)+1/sqrt(x)+1/x+x^(-3/2),r1,r2);
```

$$(\%o5) \qquad b^3+b^2+b+\frac{1}{b}+\frac{1}{b^2}+\frac{1}{b^3}$$

See, however, that the above example could have easily been done with *ratsubst*, too.

14.6 Rules, disprule, printprops, propvars

rules [system variable]

The system variable *rules* is the list of all match and replacement functions or rules defined by *defmatch, defrule, tellsimp*, and *tellsimpafter*. This list can be displayed by *disprule(all)*.

disprule ($func_1, \ldots, func_n$ | *all*) [function]

Displays the match and replacement functions or rules with the names $func_1, \ldots, func_n$ as declared by *defmatch*, *defrule*, *tellsimp*, or *tellsimpafter*. Each function is displayed with an intermediate expression label (%t<n>). *disprule (all)* displays all rules and patterns as contained in the system variable *rules*. *disprule* quotes its arguments and returns the list of intermediate expression labels corresponding to the displayed functions.

14.7 Killing and removing rules

remrule (op, rulename | all) [function]

Removes rules defined by tellsimp or tellsimpafter. *remrule (op, rulename)* removes the rule *rulename* from the operator *op*. When *op* is a built-in or user-defined operator (as defined by *infix*, *prefix*, etc.), *op* and *rulename* must be enclosed in double quotes. *remrule (op, all)* removes all rules from the operator *op*.

kill (rules) [function]

Removes all rules.

clear_rules() [function]

Calls *kill (rules)* and then resets the next rule number to 1 for addition +, multiplication *, and exponentiation ^.

Part IV

Basic Mathematical Computation

Chapter 15

Basic mathematical functions

15.1 Algebraic functions

15.1.1 Division with remainder, modulo

mod (a, b) [function]

Returns the remainder r of the division of integers a, b. This division is not defined in the same way as Satz M-5.17 for negative arguments, because it can return a negative remainder. *mod* can also be used for non-integers. [**MaxiManE**]

divide, when used for division with remainder of integers, does't either deliver the results defined in Satz M-5.17 when a or b or both are negative, nor does it deliver the same result for the remainder as *mod*.

15.2 Combinatorial functions

15.2.1 Factorials

15.2.1.1 Functions and operators

factorial(expr) [function]
expr ! [operator]

Represents the factorial function. Maxima treats *x!* the same as *factorial(x)*.

For a complex number x, except for negative integers, x! is defined as $\Gamma(x+1)$, where Γ is the gamma function.

For an integer x, x! simplifies to the product of the integers from 1 to x inclusive. 0! simplifies to 1. For a real or complex number x in float or bigfloat precision, x! simplifies to the value of $\Gamma(x+1)$. For x equal to n/2 where n is an odd integer, x! simplifies to a rational factor times $\sqrt{\pi}$, since $\Gamma(\frac{1}{2})$ is equal to $\sqrt{\pi}$.

The factorial of an integer is simplified to an exact number unless the operand is greater than factlim. The factorial for real and complex numbers is evaluated in float or bigfloat precision.

double_factorial(expr) [function]
expr !! [operator]

Represents the double factorial function, generally defined for an argument z as

$$\left(\frac{2}{\pi}\right)^{\frac{1}{4}(1-\cos(z\pi))} 2^{\frac{z}{2}}\Gamma\left(\frac{z}{2}+1\right).$$

double_factorial computes the double factorial, if its argument is a non-negative or an odd negative integer, a float, a bigfloat, or a complex float. The double factorial is not defined for even negative integers. For rationals, *double_factorial* returns a noun form. Maxima knows the derivative of the double factorial.

The operator *x!!* is only defined for non-negative integers. For an even (or odd) non-negative integer n, the double factorial evaluates to the product of all the consecutive even (or odd) integers from 2 (or 1) through n inclusive. 0!! simplifies to 1. For all other arguments, !! returns a noun form in terms of function *genfact* or an error.

```
(%i1)   double_factorial(x);
(%o1)                        double_factorial(x)
(%i1)   diff(double_factorial(x),x,1);
```

(%o1) $$\frac{\mathrm{double\_factorial}(x)\left(\frac{\pi\log\left(\frac{2}{\pi}\right)\sin(\pi x)}{2}+\Psi_0\left(\frac{x}{2}+1\right)+\log(2)\right)}{2}$$

genfact(x,y,z) [function]

Returns the generalized factorial, defined as $x(x-z)(x-2z)\ldots(x-(y-1)z)$. Thus, when x is an integer, *genfact* $(x,x,1)\equiv x!$ and *genfact* $(x,x/2,2)\equiv x!!$.

15.2.1.2 Simplification

15.2.2 Binomials

binomial(x,y) [function]

The binomial coefficient is defined as

$$\binom{x}{y}=\frac{x!}{(x-y)!y!}.$$

It can be used for numerical or symbolic computation. If x and y are integers, then the numerical value of the binomial coefficient is simplified to an integer. If x and y are real or complex float numbers, the binomial coefficient is computed according to the generalized factorial. If x is a symbol and y an integer, the binomial coefficient is expressed as a polynomial.

Chapter 16

Roots, exponential and logarithmic functions

16.1 Roots

sqrt(expr) [function]

Returns the square root of *expr*.

16.1.1 Internal representation

Square roots, as well as n-th roots in general, are represented internally as expressions with the exponentiation operator to a rational exponent. So *sqrt(x)* is represented by $x^{(1/2)}$.

16.1.2 Simplification

radexpand default: *true* [option variable]

domain default: *real* [option variable]

When *radexpand* is set to its default value *true* and *domain* to its default value *real*, sqrt(xˆ 2) is simplified to abs(x).

When *radexpand* is *all* or assume(x>0) has been executed, nth roots of factors which are powers of n are pulled outside of the root. E.g. $sqrt(16 * x^2)$ is simplified to $4x$, and $sqrt(x^2)$ to x.

When *radexpand* is *false* or (*radexpand* is *true* and *domain* is set to *complex*), $sqrt(x^2)$ will not be simplified.

rootscontract (expr) [function]

rootsconmode default: *true* [option variable]

rootscontract converts products (or quotients) of roots into roots of products (or quotients). For example, *rootscontract* $(sqrt(x) * y^{(3/2)})$ yields $sqrt(x\,y^3)$.

rootsconmode controls, how *rootscontract* is applied.

16.1.3 Roots of negative real or of complex numbers

Maxima allows negative real numbers, and more generally, complex numbers as arguments of any n-th root. If a real root exists, it is returned, otherwise Maxima computes the principal complex root. This, however, in certain cases will not be accomplished by a single command. Maxima does not automatically simplify complex numbers, so it may be that the expression is simplified only partially and will be returned containing a noun form. Further simplification can be achieved with rectform.

```
(%i1)   (-8)^(1/3)
(%o1)                                -2
(%i2)   sqrt(-4)
(%o2)                                2i
(%i3)   (-8)^(1/4);
                                  1/4 1/4
(%o3)                            (-1)   8
(%i4)   float(%);
                                  1/4
(%o4)              1.681792830507429 (-1)
(%i5)   (-1)^(1/4)
                                  1/4
(%o5)                            (-1)
(%i6)   rectform(%);
                                  %i     1
(%o6)                             --  + ---
                                  sqrt(2) sqrt(2)
(%i7)   float(%);
(%o7)          0.7071067811865475 %i + 0.7071067811865475
```

16.1.3.1 Computing all n complex roots

In the preceeding section we saw that Maxima computes only the principal root with the exponentiation operator to a rational exponent. If we want to compute all n (pairwise different) complex roots, we can use function solve. The principal root will usually be the last one in the list returned. As an example, we want to compute all three cubic roots of $110 + 74i$.

```
(%i1)   float(rectform(solve(z^3=110+74*%i,z)));
(%o1)     [z=3.8301270189221931-3.366025403784439, z=-4.830127018922193i
   -1.633974596215562, z=0.99999999999999i+5.000000000000001]
(%i2)   expand((5+%i)^3);
(%o2)                            74i+110
```

Comparing this with

```
(%i1)   float(rectform((110+74*%i)^(1/3)));
(%o2)                  0.9999999999999997i+5.0
```

we notice that the expressions for the principal root differ slightly, apparantly due to the internal use of different algorithms.

16.2 Exponential function

rexp (expr) [function]

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

16.2.1 Simplification

radcan (expr) [function]

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 default: *true* [option variable]

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 default: *false* [option variable]

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 default: *false* [option variable]

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 default: *true* [option variable]

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 17

Polynomials

17.1 Polynomial division

divide (p, q⟨, (x | x₁, ..., xₙ)⟩) [function]

In its simple form, *divide* computes the quotient and remainder of the polynomial p divided by the polynomial q, in the main polynomial variable x (which does not have to be specified as the third argument, if the first two arguments contain only one variable). *divide* returns a list of two elements, the first of which is the quotient and second the remainder.

If more than one polynomial variable is specified, the last one (x_n) is the main variable, if it is present. All variables specified are declared as potential main variables of the rational expression. If x_n is not present, x_{n-1} is the main variable, and so on; see *ratvars*.

quotient (p, q⟨, (x | x₁, ..., xₙ)⟩) [function]

This function does the same as *divide*, but only the quotient is returned.

remainder (p, q⟨, (x | x₁, ..., xₙ)⟩) [function]

This function does the same as *divide*, but only the remainder is returned.

17.2 Partial fraction decomposition

partfrac (r, x) [function]

Does a complete partial fraction decomposition of the rational function *r*, which is of the form

$$r(x) = \frac{p(x)}{q(x)}$$

with polynomials p, q, with respect to the main variable x. This means, *partfrac* expands *r* into a sum of terms comprising zero or more monomials and zero or more partial fractions, each having a simpler denominator than *r*.

The first step of what *partfrac* does is a polynomial division *p/q* as accomplished by *divide*. The quotient polynomial of this division constitutes the first part (zero

or more terms) of the sum returned by *partfrac*. In the second step, the rational function *rem/q*, with rem being the remainder polynomial of the division, is decomposed into partial fractions. The resulting terms constitute the second part (zero or more terms) of the sum returned by *partfrac*.

The importance of partial fraction decomposition primarily lies in the fact that the resulting terms are much easier to integrate than the original rational function (method of *integration by partial fractions*).

Chapter 18

Solving Equations

18.1 The different solvers

18.1.1 Linsolve

linsolve ([$eq_1, \ldots, eq_n$], [$x_1, \ldots, x_k$]) [function]

Solves the system of linear equations for the list of variables. Both sides of the equations (if present) must be polynomials in the variables.

When *globalsolve* is *true*, each solved-for variable is bound to its value in the solution of the equations. When *backsubst* is *false*, *linsolve* does not carry out back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions.

When *linsolve_params* is *true*, *linsolve* also generates *%r* symbols used to represent arbitrary parameters as described under *algsys*. Otherwise, *linsolve* solves an under-determined system of equations with some variables expressed in terms of others.

18.1.2 Algsys

Solves a system of polynomial equations.

18.1.3 Solve

Maxima's primary solver *solve* was written by Richard J. Fateman. It calls *algsys*. It was written before the knowledge database was introduced, so *solve* does not consider assumptions declared with *assume*.

fractional exponents should be eliminated from any equation befor using it with *solve*. See the thread *Pandora's box*, Jan. 28, 2021.

18.1.4 To_poly_solve, %solve

This solver, written by Barton Willis, is an alternative to using *solve* and is sometimes more powerful, for example with respect to trigonometric equations. But like *solve*, it does not recognize assumptions specified with *assume* or *declare*.

Like *solve*, *to_poly_solve* uses *algsys*. Unlike *solve*, fractional exponents are eliminated from equation automatically by *to_poly_solve*. Sometimes this is not successful, though, and it is better to do this manually as for *solve*.

to_poly_solve (*eq* | [$eq_1, \ldots, eq_n$] | {$eq_1, \ldots, eq_n$}, *x* | [$x_1, \ldots, x_k$] | {$x_1, \ldots, x_k$} ⟨, *options*⟩) [function of *to_poly_solve*]

%solve(...) [alias of *to_poly_solve*]

Tries to solve the equation or list of equations given in the first argument for the variable or list of variables given in the second argument, possibly followed by options. When *to_poly_solve* is able to determine the solution set, each element of the solution set is a list of one element in a *%union* object.

```
(%i1)   load(to_poly_solve)$
(%i2)   to_poly_solve(x*(x-1), x);
(%o2)                        %union([x = 0], [x = 1])
```

When to_poly_solve is unable to determine the solution set, a *%solve* nounform is returned. In this case, a warning is printed which tries to point to the specific cause of the failure.

```
(%i3)   to_poly_solve(x^k + 2* x + 1, x);

Nonalgebraic argument given to 'to_poly'
unable to solve
(%o3)                      %solve([x^k + 2x + 1 = 0], [x])
```

Especially for trigonometric equations, the solver sometimes needs to introduce one or more variables which can take an arbitrary integer value. These variables have the form *%zXXX*, where *XXX* is an index.

```
(%i4)   to_poly_solve(sin(x) = 0, x);
(%o4)           %union([x = 2 %pi %z33 + %pi], [x = 2 %pi %z35])
```

To re-index these variables starting from zero, use *nicedummies*.

```
(%i5)   nicedummies(%);
(%o5)            %union([x = 2 %pi %z0 + %pi], [x = 2 %pi %z1])
```

18.2 Special tasks and techniques

18.2.1 Eliminate variables from a system of equations

eliminate ([$eq_1, \ldots, eq_n$], [$x_1, \ldots, x_k$]) [function]

Eliminates variables from equations (or expressions assumed equal to zero) by taking successive resultants. This returns a list of $n-k$ equations with the k variables $x_1, \ldots, x_k$ eliminated. First x_1 is eliminated yielding $n-1$ equations, then x_2 is eliminated, etc. If $k = n$, a single expression in a list is returned free of the variables $x_1, \ldots, x_k$. In this case *solve* is called to solve the last resultant for the last variable.

```
(%i1)   eq1: 2*x^2 +y*x +z;
(%o1)                                z+xy+2x^2
(%i2)   eq2: 3*x +5*y -z -1;
(%o2)                              -z+5y+3x-1
(%i3)   eq3: z^2 +x -y^2 +5;
(%o3)                              z^2-y^2+x+5
(%i4)   eliminate([eq1, eq2, eq3], [y,z]);
(%o4)                  [x^2(45x^4+3x^3+11x^2+81x+124)]
```

The *to_poly_solve* package contains equivalent functions *elim* and *elim_allbut*. See sect. 18.2.2 for an application of both *eliminate* and *elim_allbut*.

18.2.2 Solving trigonometric or hyperbolic expressions

Equations containing trigonometric or hyperbolic expressions often cannot be solved directly, neither with *solve* nor with *to_poly_solve*. Such equations, however, can often be solved by either *exponentializing* them before employing *solve*, or by *polynomializing* them befor using *to_poly_solve*.

We will use an example to demonstrate both methods. Suppose that we want to obtain a functional expression for the coordinate lines of polar coordinates. For this we assume $r = const.$ and $r \neq 0$, and we eliminate φ from the two equations

$$x = r\,cos(\varphi) \qquad \text{and} \qquad y = r\,sin(\varphi).$$

18.2.2.1 Exponentialize and solve or eliminate

```
(%i1)   e1: x= r*cos(%phi)$
(%i2)   e2: y= r*sin(%phi)$
(%i3)   e3: exponentialize([e1,e2]);
```

$$(\%o3) \qquad [x = \frac{r\left(\%e^{\%i\varphi} + \%e^{-\%i\varphi}\right)}{2}, y = -\frac{\%i\,r\left(\%e^{\%i\varphi} - \%e^{-\%i\varphi}\right)}{2}]$$

```
(%i4)   eliminate(e3,[exp(%i*%phi)]);
```

$$(\%o4) \qquad [4r^2\left(-r^2+y^2+x^2\right)]$$

With $r \neq 0$ we find the solution $y^2 = r^2 - x^2$.

18.2.2.2 To_poly and to_poly_solve or elim(_allbut)

```
(%i3)   load(to_poly_solve)$
(%i4)   e3: to_poly([e1,e2],[%phi]);
```

$$(\%o4) \qquad [[\%i\left(\%g25^2-1\right)r+2\,\%g25y, \left(-\%g25^2-1\right)r+2\,\%g25x],$$
$$[2\,\%g25 \neq 0, 2\,\%g25 \neq 0], [\%g25 = \%e^{\%i\varphi}]]$$

```
(%i5)   elim_allbut(first(e3),[x,y,r]);
```

$$(\%o5) \qquad [[r\left(r^2-y^2-x^2\right)], [\%g25^2 r + r - 2\,\%g25x]]$$

Chapter 19

Linear Algebra

19.1 Introduction

19.1.1 Operation in total or element by element

A clear conceptional distinction should be made between operations which apply to a structure (vector, matrix, etc.) as a whole, and operations which apply to all the elements of a structure individually, i.e. element by element, joining the results to a structure of the original kind to be returned. Examples of operations in total are scalar product or matrix inversion, while examples of operations element by element are scalar multiplication of a vector or matrix, or integration of a vector or matrix, if their elements are functions.

19.2 Dot operator: general non-commutative product

a . b [infix operator]

Maxima's *dot operator* "." represents the general *non-commutative product*, here also called *dot product*. It can be used e.g. for the matrix product, section 19.4.9.1, the scalar product, section 19.3.7, or the tensor product of vectors, section19.3.8. But the dot operator is applicable as a non-commutative product to any other kind of object, too.

In order to clearly distinguish the dot operator from the decimal point of a floating point number, it is advisable to always leave a blank before and after the dot.

19.2.1 Exponentiation

a^^2 [infix operator]

The ^^ operator is the exponentiation of the non-commutative product ".", just as ^ is the exponentiation of the *commutative product* "*". In 2D display mode, the exponent is enclosed in angle brackets.

```
(%i1)   a.a;
(%o1)                          a<2>
(%i2)   b*b;
(%o2)                           2
                               b
```

19.2.2 Option variables for the dot operator

The dot operator is controlled by a large number of flags. They influence the rules which govern its simplification.

dot0nscsimp default: *true* [option variable]

When *dot0nscsimp* is *true*, a non-commutative product of zero and a nonscalar term is simplified to a commutative product.

dot0simp default: *true* [option variable]

When *dot0simp* is *true*, a non-commutative product of zero and a scalar term is simplified to a commutative product.

dot1simp default: *true* [option variable]

When *dot1simp* is *true*, a non-commutative product of one and another term is simplified to a commutative product.

dotassoc default: *true* [option variable]

When dotassoc is *true*, an expression *(A.B).C* simplifies to *A.(B.C)*.

dotconstrules default: *true* [option variable]

When dotconstrules is *true*, a non-commutative product of a constant and another term is simplified to a commutative product. Turning on this flag effectively turns on *dot0simp*, *dot0nscsimp*, and *dot1simp* as well.

dotdistrib default: *true* [option variable]

When *dotdistrib* is *true*, an expression *A.(B+C)* simplifies to *A.B + A.C*.

dotexptsimp default: *true* [option variable]

When *dotexptsimp* is *true*, an expression *A.A* simplifies to $A^{<2>}$, which is *A^^2*.

dotident default: *1* [option variable]

dotident is the value returned by $X^{<0>}$, which is X^^0.

dotscrules default: *false* [option variable]

When *dotscrules* is *true*, an expression *A.SC* or *SC.A* simplifies to *SC*A*, and *A.(SC*B)* simplifies to *SC*(A.B)*.

19.3 Vector

19.3.1 Representations and their internal data structure

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. The following shows the internal data structure of these representations. Note that a matrix internally

is a special list of MaximaL lists, each of them representing one row, see section 19.4.1.

```
(%i1)   u:[x,y,z];
(%o1)                          [x, y, z]
(%i2)   :lisp $U
        ((MLIST SIMP) x y z)
(%i3)   v:covect(u);
                                 ⎛ x ⎞
(%o3)                            ⎜ y ⎟
                                 ⎝ z ⎠
(%i4)   :lisp $V
        (($MATRIX SIMP) ((MLIST SIMP) x) ((MLIST SIMP) y) ((MLIST SIMP) z))
(%i5)   w:transpose(u);
(%o5)                          ⎛ x  y  z ⎞
(%i6)   :lisp $W
        (($MATRIX SIMP) ((MLIST SIMP) x y z))
```

19.3.2 Option variables for vectors

There are only a few specific option variables for vectors. Most option variables relate to either matrices or lists. See section 19.4.3 for option variables applicable to matrices, and section 8.1 for those on lists. Thus, behavior of vector operations may depend on the vector representations, see section 19.3.1. Row and column vectors are matrices.

vect_cross default: *false* [option variable]

When *vect_cross* is *true*, the vector product defined as the operator ~in share package *vect* may be differentiated as in diff(x~y,t). Note that loading *vect* will set *vect_cross* to *true*.

19.3.3 Construct, transform and transpose a vector

A list can be constructed by entering the elements inside of square brackets, separated by commas.

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

Special functions for creating lists (e.g. *makelist* and *create_list*) are described in section 8.1.

CVect $(x_1, x_2, \ldots, x_n)$ [function of *rs_vector*]
vect $(x_1, x_2, \ldots, x_n)$ [function of *rs_vector*]
RVect $(x_1, x_2, \ldots, x_n)$ [function of *rs_vector*]

CVect and vect (synonym, kept for backward compatibility) construct a column vector which is a matrix of one column and n rows, containing the arguments. RVect

constructs 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)

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

MakeList (x,n) [function of *rs_vector*]
MakeCVect (x,n) [function of *rs_vector*]
MakeRVect (x,n) [function of *rs_vector*]

These functions create a vector in the respective representation 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. Note that MakeList is not identical with system function makelist, but it makes use of it.

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

(%o1)

$$[x_1, x_2, x_3]$$

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

(%o2)

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

```
(%i3)    z:MakeRVect(z,3);
```

(%o3)

$$\begin{pmatrix} z_1 & z_2 & z_3 \end{pmatrix}$$

System function *genmatrix* can be used to construct a column or row vector from an *undeclared array*, too, but with symbolic elements having two indices instead of one, as for matrices.

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

(%o1)

$$\begin{pmatrix} x_{1,1} \\ x_{2,1} \\ x_{3,1} \end{pmatrix}$$

```
(%i2)    x:genmatrix(x,1,3);
```

(%o2)

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \end{pmatrix}$$

The following functions achieve transformation between different representations.

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

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

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

(%o1)
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

transpose (v) [function]

Transposes a list or a row vector into a column vector, and a column vector into a row vector. For the more general transposition of a matrix, see *transpose (M)*.

Transpose (v) [function of *rs_vector*]

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

VtoList (v) [function of *rs_vector*]

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

VtoCVect (v) [function of *rs_vector*]

Transforms a vector of any kind into a column vector. If v is already a column vector, it will be returned. Note that *transpose(VtoList(v))* will also transform a vector of any kind into a column vector.

VtoRVect (v) [function of *rs_vector*]

Transforms a vector of any kind into a row vector. If v is already a row vector, it will be returned. Note that *transpose(transpose(VtoList(v)))* will also transform a vector of any kind into a row vector.

19.3.4 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.

VDim(v) [function of *rs_vector*]

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

19.3.5 Indexing: refering to 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.

19.3.6 Arithmetic operations and other MaximaL functions applicable to vectors

listarith [option variable]

Scalar multiplication of a vector and arithmetic operations between vectors work element by element, 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 be a column vector.

distribute_over [option variable]

Many other computational or simplifying/manipulating MaximaL functions can be applied to vectors, which means that they operate element by element. The flags *doallmxops* and *distribute_over* must be *true* (default). Examples are diff, factor, expand.

19.3.7 Scalar product

19.3.7.1 Dot operator

The *scalar product*, *dot product*, or *inner product* $v \cdot w$ of two real valued vectors v and w, which, in case of a list representation of the vectors, is equal to *sum (v[i]*w[i], i, 1, length(v))* can be built with the *dot operator* for the non-commutative matrix product, see sect. 19.4.9.1. The arguments need to have the same dimension, but can be of any representation, 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 two vectors, see sect. 19.3.8. Hence, this operator is non-commutative with respect to the combination of vector representations. For a commutative way (with respect to the combination of vector representations) of computing the scalar product see the operator *SP*. The non-commutative scalar product of complex valued vectors can be computed with *SP*, too, of with functions *inprod* or *Inprod*.

```
(%i1)   powerdisp:true$
(%i2)   v:MakeCvect(v,3)$  w:MakeCvect(w,3)$
(%i3)   v . w;
(%o3)                          v1 w1 + v2 w2 + v3 w3
```

The dot operator is controlled by a number of flags which are described in section 19.2.

19.3.7.2 innerproduct, inprod, Inprod

innerproduct (v,w) [function of *eigen*]
inprod (v,w) [function of *eigen*]

Returns the inner product (also called the scalar product or dot product) of two vectors v and w, which can be lists of equal length, both column or both row vectors of equal dimension. The return value is

$$conjugate(v)\ .\ w,$$

where "." is the dot operator. This function can be applied to complex and real valued vectors.

Inprod (v,w) [function of *rs_vector*]

Returns, under the same conditions as inprod,

$$v \,.\, conjugate(w),$$

which is equal to $-inprod(v, w)$.

19.3.7.3 SP

v SP w [infix operator of *rs_vector*]

The infix operator *SP* computes the scalar product of two complex or real valued vectors of equal dimension, independently of their representations. It is a commutative version (with respect to the combination of vector representations) of the *dot operator* for real valued vectors and of *Inprod* for complex valued vectors. Internally, both vectors are transformed to column vectors first, then the dot operator or Inprod is employed. By this procedure all flags which control the dot operator stay valid.

```
(%i1)   v:MakeCvect(v,3)$  w:MakeRvect(w,3)$
(%i2)   c SP r;
(%o2)                          v1 w1 + v2 w2 + v3 w3
```

19.3.8 Tensor product

The non-commutative *tensor product* $v \otimes w$ can be computed with the *dot operator*, see section 19.3.7, 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. For a description of the flags that control the dot operator, see section 19.2. For a way to compute the tensor product independently of the vector representations see the operator TP.

```
(%i1)   v:MakeCvect(v,3)$  w:MakeRvect(w,3)$
(%i2)   v . w;
                                [ v1 w1  v2 w2  v1 w3 ]
                                [                     ]
(%o2)                           [ v2 w1  v2 w2  v2 w3 ]
                                [                     ]
                                [ v3 w1  v3 w2  v3 w3 ]
```

v TP w [infix operator of *rs_vector*]

The infix operator *TP* computes the tensor product of two vectors of any representation. The arguments need not have the same dimension. TP returns an $m \times n$ matrix. 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:MakeCvect(v,3)$  w:MakeCvect(w,3)$
(%i2)   v TP w;
```

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

19.3.9 Norm and normalization

VNorm(v⟨, ip⟩) [function of *rs_vector*]

Computes the *separation sqrt(abs(v.v))* of a vector v supplied as the first argument. The separation is a generalization of the norm, applicable even for an inner product which is not positive definite. If no second argument is present, for the inner product function SP is used, which computes the norm independently of the representation of v. If the second argument is present, it denotes the function to be used instead. *ip* has to be a prefix function of two arguments. If an infix function is to be used instead, it must be enclosed in double quotes, e.g. "." for the dot operator.

```
(%i1)  v:MakeCvect(v,3)$
(%i2)  VNorm(v,".");
```
(%o2)
$$\sqrt{v_1^2 + v_2^2 + v_3^2}$$

Normalize (v ⟨,ip ⟩) [function of *rs_vector*]
NormalizeColumns default: *true* [option variable]

Function *Normalize* Normalizes a column vector, row vector, list or matrix (column-wise, if the global flag *NormalizeColumns* is *true*, row-wise otherwise) by dividing each vector by its separation (e.g. norm) using function *VNorm*. If a function different from *SP* shall be used by VNorm for the inner product, it has to be supplied as the second argument to *Normalize*. If it is an infix operator, it has to be enclosed in double quotes. The return value will be of the same type as obj and have a separation equal to 1 (matrix: column-wise resp. row-wise).

```
(%i1)  X:matrix([2,1,1],[0,3,0],[-1,0,4]);
```
(%o1)
$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & 3 & 0 \\ -1 & 0 & 4 \end{pmatrix}$$
```
(%i2)  Normalize(X);
```
(%o2)
$$\begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{2}\sqrt{5}} & \frac{1}{\sqrt{17}} \\ 0 & \frac{3}{\sqrt{2}\sqrt{5}} & 0 \\ -\frac{1}{\sqrt{5}} & 0 & \frac{4}{\sqrt{17}} \end{pmatrix}$$
```
(%i3)  Normalize(X), NormalizeColumns:false;
```
(%o3)
$$\begin{pmatrix} \frac{\sqrt{2}}{\sqrt{3}} & \frac{1}{\sqrt{2}\sqrt{3}} & \frac{1}{\sqrt{2}\sqrt{3}} \\ 0 & 1 & 0 \\ -\frac{1}{\sqrt{17}} & 0 & \frac{4}{\sqrt{17}} \end{pmatrix}$$

unitvector (v) [function of *eigen*]
uvect (v) [function of *eigen*]

Returns the normalized vector *v/norm*(*v*), probably using eigen's function *inner-product* for the inner product. This means that it can be used for the standard Euclidean or complex (positive definite) scalar product only.

19.3.10 Vector equations

19.3.10.1 Extract component equations from a vector equation

ExtractCequations (arg) [function of *rs_vector*]

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 list and column vector 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, so that this is guaranteed. ExtractCEquations returns a list of VDim(arg) component equations which e.g. can be forwarded to function *solve*.

```
(%i1)   u:MakeCvect(u,3)$  v:MakeCvect(v,3)$
(%i2)   w:makelist(w[i],i,1,3)$
(%i3)   ExtractCEquations(u+v=w);
(%o3)                [v_1 + u_1 = w_1, v_2 + u_2 = w_2, v_3 + u_3 = w_3]
```

19.3.11 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_vector*]

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

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

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

19.3.12 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:MakeCvect(v,3)$  w:MakeCvect(w,3)$  u:MakeCvect(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));
```

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

19.3.13 Basis

In order to avoid problems arising from the way Maxima implements indexed data objects, i.e. by using undeclared arrays, it is advisable instead to define a basis by using a matrix and to implement any operation on the basis as a whole as an operation on this matrix. Although a matrix in Maxima is structured by rows, it is preferable to consider the individual vectors as columns, as it is usually done in mathematics, e.g. for a basis transformation matrix. In this case a vector cannot be addressed by simply indexing the matrix, its representation being a list. But this representation of a vector is seldomly used and does not balance the drawback of mentally having to transpose a row-wise representation of the basis. It is easy to transform the matrix into a list of column vectors. In the following example, the individual column vectors can be addressed either as ei, $i = 1, 2, 3$, or as $e[i]$. In the last line, the metric tensor (Gram's matrix, positive definite representation matrix) of the Euclidean scalar product ist generated from this particular basis.

```
(%i1)   E:matrix([1,0,0],[0,1,0],[0,0,1]);
```

$$(\%o1)\quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
(%i2)   e:makelist(concat(e,i)::col(E,i),i,1,3);
```

$$(\%o2)\quad \left[\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\right]$$

```
(%i3)   e1;
```

$$(\%o3)\quad \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

```
(%i4)   e[1];
```

$$(\%o4)\quad \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

```
(%i5)   genmatrix(lambda([x,y],e[x] . e[y]),3,3);
```

$$(\%o5)\quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

19.4 Matrix

19.4.1 Internal data structure

A matrix internally is a list of MaximaL lists, each of them representing one row. Nevertheless, a matrix has its own special data type in Maxima. Thereby Maxima can distinguish between a matrix and any other 2-dim. list structure.

```
(%i1)   M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1)
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
(%i2)   :lisp |$m|
        (($MATRIX SIMP) ((MLIST SIMP) 1 2 3) ((MLIST SIMP) 4 5 6) ((MLIST
           SIMP) 7 8 9))
```

19.4.1.1 matrixp

matrixp (expr) [function]

Returns *true* if *expr* is a matrix, otherwise *false*.

```
(%i3)   matrixp(M);
(%o3)                                true
```

19.4.2 Indexing: Refering to the elements of a matrix

Square brackets are used for indexing matrices, that is to refer to its elements. Indices start with 1. The first argument ist the row, the second the column. See the example below.

```
(%i4)   M[2,1];
(%o4)                                 4
```

19.4.3 Option variables for matrices

A number of option variables enable, disable and control different kinds of matrix operations. See section 8.1 for option variables on lists, and section 19.3.2 for those on vectors.

doallmxops default: *true* [option variable]

When *doallmxops* is *true*, all operations relating to matrices are carried out. When it is *false*, the settings of the individual dot switches govern which operations are performed.

domxmxops default: *true* [option variable]

When *domxmxops* is *true*, all matrix-matrix or matrix-list operations are carried out, but not scalar-matrix operations; if this switch is false, such operations are not carried out.

domxnctimes default: *false* [option variable]

When *domxnctimes* is *true*, non-commutative products of matrices are carried out.

doscmxops default: *false* [option variable]

When *doscmxops* is *true*, scalar-matrix operations are carried out.

doscmxplus default: *false* [option variable]

When *doscmxplus* is *true*, scalar-matrix operations yield a matrix result. This switch is not subsumed under *doallmxops*.

matrix_element_add default: + [option variable]

matrix_element_mult default: * [option variable]

matrix_element_transpose default: *false* [option variable]

ratmx default: *false* [option variable]

When *ratmx* is *false*, matrix addition, subtraction, and multiplication as well as function *determinant* are performed in the representation of the matrix elements and cause the result of matrix inversion to be returned in general representation.

When *ratmx* is *true*, the operations mentioned above are performed in CRE form and the result of matrix inverse is returned in CRE form. Note that this may cause the elements to be expanded (depending on the setting of *ratfac*) which might not always be desirable.

scalarmatrixp default: *true* [option variable]

When *scalarmatrixp* is *true*, then whenever a 1 x 1 matrix is produced as a result of computing the dot product of matrices, it is simplified to a scalar, being the sole element of the matrix. When *scalarmatrixp* is *all*, then all 1 x 1 matrices are simplified to scalars. When *scalarmatrixp* is *false*, 1 x 1 matrices are never simplified to scalars.

Known bug: The value returned by computing the dot product v.v of a column or row vector or a list v with v^^2 is a 1 x 1 matrix, even if *scalarmatrixp* is *true*. In case of v being a list, it is even a 1 x 1 matrix when *scalarmatrixp* is *all*.

19.4.4 Build a matrix

There are several ways to build a matrix. It can be entered as a whole or it can be constructed column by column, row by row, or even by joining submatrices. A matrix can also be extracted from a bigger matrix. Special types of matrices like identity or diagonal matrices can easily be built with the respective specialized Maxima functions. *Genmatrix* allows for creating a matrix with a lambda function.

19.4.4.1 Enter a matrix

matrix $(L_{r_1}, \ldots, L_{r_m})$ [function]

This function can be used to enter an $m \times n$ matrix. Each row is given as a MaximaL list and must contain the same number n of elements. In wxMaxima the menu *Algebra / Enter matrix* can be used to facilitate the input.

```
(%i1)   M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

$$\text{(\%o1)} \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

19.4.4.2 Append colums, rows or whole matrices

A matrix can be constructed by starting with a column or row vector and appending columns at the right or rows at the bottom one by one. In the same way, columns or rows can be appended to any existing matrix, too. The following functions can even be used to append whole matrices at the right or bottom of an existing matrix.

addcol $(M, L_{c_1} | M_1, \ldots, L_{c_k} | M_k)$ [function]
addrow $(M, L_{r_1} | M_1, \ldots, L_{r_k} | M_k)$ [function]

addcol $(M, L_{c_1}, \ldots, L_{c_k})$ appends at the right of the $m \times n$ matrix M the k columns containing the elements from lists L_{c_i}, $i = 1, \ldots, k$, each having m elements.

addrow $(M, L_{r_1}, \ldots, L_{r_k})$ appends at the bottom of the $m \times n$ matrix M the k rows containing the elements from lists L_{r_i}, $i = 1, \ldots, k$, each having n elements.

addcol $(M, M_1, \ldots, M_k)$ / *addcol* $(M, M_1, \ldots, M_k)$ append at the right / bottom of the $m \times n$ matrix M the k matrices M_i, $i = 1, \ldots, k$, each having m rows / n columns.

Appending matrices and columns with *addcol* can even be arbitrarily combined. Analogously, this holds for *addrow*.

```
(%i1)   M:Cvect(a,b,c);
```

$$\text{(\%o1)} \quad \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

```
(%i2)   N:addcol(M,[d,e,f]);
```

$$\text{(\%o2)} \quad \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

```
(%i3)   addcol(N,N);
```

$$\text{(\%o3)} \quad \begin{pmatrix} a & d & a & d \\ b & e & b & e \\ c & f & c & f \end{pmatrix}$$

```
(%i4)   addcol(N,[1,2,3],N,[4,5,6]);
```

(%o4)
$$\begin{pmatrix} a & d & 1 & a & d & 4 \\ b & e & 2 & b & e & 5 \\ c & f & 3 & c & f & 6 \end{pmatrix}$$

19.4.4.3 Extract a submatrix, column or row

submatrix ($\langle r_1, \ldots, r_k, \rangle M \langle, c_1, \ldots, c_l \rangle$) [function]

Returns a new matrix constructed from the $m \times n$ matrix M, with rows $r_1, \ldots, r_k$ and/or columns $c_1, \ldots, c_l$ deleted, row indices being $1 \leq r_i \leq k$ and column indices $1 \leq c_j \leq n$. Note that indices preceeding M are interpreted as rows, while those following M are interpreted as columns. The respective indices don't have to be in numerical order.

row (M,i) [function]

Returns the i-th row of the matrix M. The return value is a row vector (which is a matrix).

col (M,j) [function]

Returns the j-th column of the matrix M. The return value is a column vector (which is a matrix).

19.4.4.4 Build special matrices

19.4.4.4.1 Identity matrix

ident (n) [function]

Returns an $n \times n$ identity matrix.

19.4.4.4.2 Zero matrix

zeromatrix (m,n) [function]

Returns an $m \times n$ zero matrix.

19.4.4.4.3 Diagonal matrix

diagmatrix (n,x) [function]

Returns an $n \times n$ diagonal matrix, each element of the diagonal containing x, which can be any kind of expression. If x is a matrix, it is not copied; all diagonal elements refer to the same instance of x.

19.4.4.5 Genmatrix

genmatrix ($a, i_2, j_2 \langle, i_1 \langle, j_1 \rangle \rangle$) [function]

This function creates a matrix

$$\begin{pmatrix} a_{i_1 j_1} & \cdots & a_{i_1 j_2} \\ \vdots & & \vdots \\ a_{i_2 j_1} & \cdots & a_{i_2 j_2} \end{pmatrix}$$

from argument a, which must be either a *declared array* (created by *array*, but not by *make_array*), an *undeclared array*, an *array function* or a *lambda function* of two arguments, taking $a[i_1, j_1]$ as the first and $a[i_2, j_2]$ as the last element of the matrix. If j_1 is omitted, it is assumed to be equal to i_1. If both j_1 and i_1 are omitted, both are assumed to be equal to 1.

An example with an *undeclared array* is given in section 19.3.3, with a lambda function in section 19.3.13.

19.4.5 Transform between representations

Transformation between different representations of a matrix can be achieved with the help of Maxima functions *apply*, *makelist*, and *map* or *maplist*. We give three examples.

19.4.5.1 List of sublists -> matrix

A list of sublists can be transformed into a corresponding matrix in the following way. Note that a sublist corresponds to a row.

```
(%i1)   L:[[1,0,0],[0,1,0],[1,2,3]];
(%o1)                    [[1,0,0],[0,1,0],[1,2,3]]
(%i2)   M:apply(matrix,L);
```

$$\text{(\%o2)} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 2 & 3 \end{pmatrix}$$

19.4.5.2 Matrix -> list of column vectors

A matrix can be transformed into a list of column vectors, see example in sect. 19.3.13. In the following example we use the transpose of matrix M generated above.

```
(%i3)   N:makelist(col(transpose(M),i),i,1,3);
```

$$\text{(\%o3)} \qquad \left[\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}\right]$$

19.4.5.3 List of column vectors -> list of sublists

A list of column vectors can be transformed into a list of lists.

```
(%i4)   map(VtoList,N);
(%o4)                    [[1,0,0],[0,1,0],[1,2,3]]
```

19.4.6 Functions applied element by element

19.4.6.1 Arithmetic operations and other MaximaL functions applicable to matrices

The operations + (addition), - (subtraction), * (multiplication), and / (division), are carried out element by element when the operands are two matrices, a scalar and a matrix, or a matrix and a scalar.

The operation ^ (exponentiation, equivalently **) is carried out element by element, if the operands are a scalar and a matrix or vice versa, but not if the operands are two matrices.

Differentiation and integration of a matrix is also performed element by element, each element being considered as a function.

19.4.6.2 Mapping arbitrary functions and operators

matrixmap $(f, M_1, \ldots, M_n)$ [function]

Applies an arbitrary function or operator f of n arguments to matrices $M_1, \ldots, M_n$ element by element, returning a matrix with element [i,j] equal to $f(M_1[i,j], \ldots, M_1[i,j])$. The number of matrices has to correspond to the number of arguments required by f. *matrixmap* is a version of function *map* being applicable to matrices (which *map* is not). See there for more explanations and examples.

In the following example, f is unbound at first and as such can have an arbitrary number of arguments, always returning a noun expression.

```
(%i1)   M:matrix([1,2,3],[4,5,6],[7,8,9])$  matrixmap(f,M);
```

$$\text{(\%o2)} \quad \begin{pmatrix} f(1) & f(2) & f(3) \\ f(4) & f(5) & f(6) \\ f(7) & f(8) & f(9) \end{pmatrix}$$

```
(%i3)   N:matrix([a,b,c],[d,e,f],[g,h,i])$  matrixmap(f,M,N);
```

$$\text{(\%o4)} \quad \begin{pmatrix} f(1,a) & f(2,b) & f(3,c) \\ f(4,d) & f(5,e) & f(6,f) \\ f(7,g) & f(8,h) & f(9,i) \end{pmatrix}$$

```
(%i5)   f(x):=2*x$  matrixmap(f,M);
```

$$\text{(\%o6)} \quad \begin{pmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{pmatrix}$$

```
(%i7)   matrixmap("=",N,M);
```

(%o7)
$$\begin{pmatrix} a=1 & b=2 & c=3 \\ d=4 & e=5 & f=6 \\ g=7 & h=8 & i=9 \end{pmatrix}$$

fullmapl (f, $M_1, \ldots, M_n$) [function]

fullmapl is a version of function *fullmap* being applicable to lists and matrices. See section 8.1 for explanations and examples.

19.4.7 Transposition

transpose (M) [function]

Transposes matrix M. *transpose* can also be used to transform a list into a column vector. For the general transposition of vectors, see *transpose (v)* and *Transpose*.

19.4.8 Inversion

Matrix inversion can be carried out with function *invert*, or directly by matrix exponentiation with -1. Both methods are equivalent.

invert (M) [function]

invert(M) is equivalent to *M^^−1* , that is $M^{<-1>}$. The inverse of the matrix M is returned. The inverse is computed via the LU decomposition.

When *ratmx* is true, elements of M are converted to *canonical rational expressions (CRE)*, and the elements of the return value are also CRE. When *ratmx* is false, elements of M are not converted to a common representation. In particular, *float* and *bigfloat* elements are not converted to rationals.

When *detout* is true, the determinant is factored out of the inverse. The global flags *doallmxops* and *doscmxops* must be false to prevent the determinant from being absorbed into the inverse. *xthru* can multiply the determinant into the inverse.

invert does not apply any simplifications to the elements of the inverse apart from the default arithmetic simplifications. *ratsimp* and *expand* can apply additional simplifications. In particular, when M has polynomial elements, *expand(invert(M))* might be preferable.

19.4.9 Product

19.4.9.1 Non-commutative matrix product

The the non-commutative matrix product can be built with the dot operator, see section 19.2. The number of rows of argument *a* has to equal the number of columns of *b*. The dot operator is controlled by a number of flags which are described in section 19.2.

19.4.10 Rank

rank (M) [function]

Computes the rank of the matrix M. That is, the order of the largest non-singular subdeterminant of M.

rank may return the wrong answer, if it cannot determine that a matrix element equivalent to zero is indeed so.

19.4.11 Gram-Schmidt procedure

19.4.11.1 Orthogonalize

gramschmidt (M⟨, ip⟩) [function of *eigen*]

Carries out the Gram-Schmidt orthogonalization procedure on a set of vectors, given either as the rows (!) of a matrix *M* or a list of lists, the sublists each having the same number of elements. *M* is not modified by gramschmidt.

The second argument *ip*, if present, denotes the function employed by gramschmidt for the inner product; otherwise the function innerproduct will be used. *ip* has to be a prefix function of two arguments. If an infix function is to be used instead, it must be enclosed in double quotes, e.g. "." for the dot operator.

The return value is a list of lists, the sublists of which are orthogonal and span the same space as x. If the dimension of the span of x is less than the number of rows or sublists, some sublists of the return value are zero. factor is called at each stage of the algorithm to simplify intermediate results. As a consequence, the return value may contain factored integers.

19.4.11.2 Orthonormalize

GramSchmidt (M⟨, ip⟩) [function of *rs_vector*]

Carries out the Gram-Schmidt orthonormalization procedure on a set of vectors, given either as the columns of a matrix M, a list of column vectors or a list of lists, the sublists each having the same number of elements. M is not modified by GramSchmidt.

GramSchmidt calls function gramschmidt.

The return value is a matrix, the vectors being its columns. The vectors are not only orthogonal and span the same space as x, but they are also normalized.

19.4.12 Triangularize

triangularize (M) [function]

Returns the upper triangular form of the matrix M, as produced by Gaussian elimination.[1] The return value is the same as from *echelon*, except that the leading

[1] This has nothing to do with triangularization of endomorphisms.

nonzero coefficient in each row is not normalized to 1.

The matrix M is *positive definite*, iff all diagonal elements of *triangularize(M)* are positive. No statement on other forms of definiteness can be made. See math sect. 47.9.3.6. [**wikDefin**]

19.4.13 Eigenvalue, eigenvector, diagonalize

eigenvalues (M) [function of *eigen*]
eivals (M) [function of *eigen*]

eivals is a synonym for *eigenvalues*. This function from the *eigen* package returns a list of two sublists. The first sublist gives the eigenvalues of the matrix M, while the second one gives the algebraic multiplicities of the eigenvalues in the corresponding order. The package *eigen.mac* is loaded automatically when *eigenvalues* or *eigenvectors* is called. This can also be done manually by *load (eigen)*.

eigenvalues calls the Maxima function *solve* to find the roots of the characteristic polynomial of the matrix. Sometimes *solve* may not be able to find the roots of the polynomial; in this case some other functions in this package (except *innerproduct*, *unitvector*, *columnvector* and *gramschmidt*) will not work. Sometimes *solve* may find only a subset of the roots of the polynomial. This may happen when the factoring of the polynomial contains polynomials of degree 5 or more. In such cases a warning message is displayed and only the roots found and their corresponding multiplicities are returned.

In some cases the eigenvalues found by solve may be complicated expressions. In *casus irreducibilis* [2] the return value may contain complex terms which are not obvious to be zero. However, it may be possible to simplify the result using other functions. For example, the following real symmetric matrix should have real eigenvalues only.

```
(%i1)   M: matrix([5/4,1/2,1/2],[1/2,5,-1],[1/2,-1,2]);
```

$$\text{(\%o1)} \qquad \begin{pmatrix} \frac{5}{4} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 5 & -1 \\ \frac{1}{2} & -1 & 2 \end{pmatrix}$$

```
(%i2)   float(ratsimp(rectform(eigenvalues(M))));
(%o2)
   [[2.124542032328667,0.7946677527382047,5.330790214933128],[1.0,1.0,1.0]]
```

charpoly (M,x) [function]

Returns the characteristic polynomial for the matrix M with respect to variable x, i.e. *determinant (M − diagmatrix (length (M), x))*.

[2] See thread maxima-discuss from Sept. 8, 2020.

19.5 Determinant

determinant (M) [function]

Computes the determinant of matrix M. The form of the result depends upon the setting of the flag *ratmx*. There is a special routine for computing the determinant of sparse matrices which is called when both *ratmx* and *sparse* are *true*.

19.5.1 Option variables for *determinant*

Some option variables for matrices, see section 19.4.3, apply to *determinant*, too.

sparse default: *false* [option variable]

When *sparse* and *ratmx* are *true*, *determinant* will use special routines for computing sparse determinants.

Chapter 20

Limits

Chapter 21

Sums, products and series

21.1 Sums and products

21.1.1 Sums

21.1.1.1 Introduction

A consecutive sum, with the index running over a range of consecutive integers, can be created with function *sum*. I can be displayed in sigma notation, simplified and evaluated. Sums can also be differentiated or integrated, and they can be subject to limits.

A selective sum, with the index only taking selected indices from a list, is created with function *lsum*.

21.1.1.2 Sum: consecutive indices

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

Builds a "continuous" 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 by simplification and evaluation cannot be resolved are displayed in sigma notation. By setting i_1 to *inf* for infinity we obtain a series, see section 21.2.2.

```
(%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)
```

21.1.1.2.1 Simplification

21.1.1.2.1.1 Simpsum

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

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. See section 21.2.2 for the application to series.

```
(%i1)   sum (2^k + k^2, k, 0, n);
```

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

```
(%i2)   sum (2^k + k^2, k, 0, n), simpsum;
```

$$2^{n+1} + \frac{2\,n^3 + 3\,n^2 + n}{6} - 1$$

21.1.1.2.1.2 Simplify_sum

Package *simplify_sum* contains function *simplify_sum* which is more powerful in finding closed forms than setting flag *simpsum*.

simplify_sum (expr) [function of *simplify_sum*]

Tries to simplify all sums appearing in *expr* to a closed form.

```
(%i1)   load(simplify_sum);
(%o1)   C:/maxima-5.40.0/../share/maxima/5.40.0/share/solve_rec/simplify_sum.mac
(%i2)   sum(2^k+k^2,k,0,n);
```

$$\sum_{k=0}^{n} 2^k + k^2$$

```
(%i3)   simplify_sum(%);
```

$$2^{n+1} + \frac{2\,n^3 + 3\,n^2 + n}{6} - 1$$

21.1.1.3 Lsum: selected indices

lsum (expr, i, L) [function]

Represents the sum of *expr* for each index contained in the list L. A noun form is returned, if the L does not evaluate to a list. All arguments except for i are evaluated.

```
(%i1)   'lsum (x^i, i, [1, 2, 7]);
```

$$\sum_{i \text{ in } [1,2,7]} x^i$$

```
(%i2)   lsum (x^i, i, [1, 2, 7]);
```

$$x^7 + x^2 + x$$

21.1.1.4 Nusum

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

This is a *new* sum function, more powerful than *sum*, capable of simplification and of finding more closed forms, of series as well.

The noun form of *nusum* is not displayed in sigma notation. However, sigma notation is used for the return value, when *nusum* cannot simplify *expr*.

Let's construct an example, where *sum* throws the towel. (For the last computation, which redoes the summation, see *unsum*).

```
(%i1)   sum (i^4*4^i/binomial(2*i,i), i, 0, n), simpsum;
```

(%o1)
$$\sum_{i=0}^{n} \frac{i^4\, 4^i}{\binom{2i}{i}}$$

```
(%i2)   nusum (i^4*4^i/binomial(2*i,i), i, 0, n);
```

(%o2)
$$\frac{2\,(n+1)\left(63n^4+112n^3+18n^2-22n+3\right)4^n}{693\binom{2n}{n}}-\frac{2}{231}$$

```
(%i3)   unsum(%,n);
```

(%o3)
$$\frac{n^4\, 4^n}{\binom{2n}{n}}$$

21.1.1.5 Differentiating and integrating 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)
$$\int\sum_{k=1}^{n}(x-x_0)^k\,dx=\sum_{k=1}^{n}\frac{(x-x_0)^{k+1}}{k+1}$$

21.1.1.6 Limits of sums

Sums can be subject to limits.

21.1.1.7 Unsum: undoing a sum

unsum (expr, n) [function]

This function is kind of magic. It undoes a definite (i.e. finite) summation done with *sum* or *nusum* and having a symbol as the (finite) upper bound. The undo is done by taking the closed form and returning the expression under the sigma sign. The argument *expr* must be the closed form of a definite summation and n the symbol designating its (finite) upper bound. The lower bound of the original sum does not matter, it can be an integer or a symbol. (This implies, that unsum cannot find it out, either.) See *nusum* for a more sophisticated example, if you still don't believe it.

```
(%i1)   nusum(i^2,i,0,n);
```

$$(\%o1) \qquad \frac{n\,(n+1)\,(2n+1)}{6}$$

```
(%i2)   unsum(%,n);
```

$$(\%o2) \qquad n^2$$

```
(%i3)   nusum(i^2,i,m,n);
```

$$(\%o3) \qquad \frac{(n-m+1)\left(2n^2+2mn+n+2m^2-m\right)}{6}$$

```
(%i2)   unsum(%,n);
```

$$(\%o2) \qquad n^2$$

21.1.2 Products

21.2 Series

21.2.1 Introduction

Maxima contains functions *powerseries* and *taylor* for finding the series of differentiable functions. It also has tools such as *nusum* capable of finding the closed form of some series. Operations such as addition and multiplication work as usual on series. This section presents the global variables which control the expansion.

Series, including power series and truncated taylor expansions, can be differentiated and integrated.

21.2.2 Sum or nusum with infinite upper bound

In Maxima a series is constructed using functions *sum* or *nusum* with the upper bound set to *inf* for infinity. In case of *sum*, all simplification procedures described for finite sums can be used for series as well. Function *nusum* is more powerful concerning simplification, and it achieves closed forms more often than *sum*. We give *sum* a try with a simple geometric series.

```
(%i1)   'sum(x^i,i,0,inf);
```

```
(%o1)
```
$$\sum_{i=0}^{\infty} x^i$$

```
(%i2)   simpsum:true$  sum(x^i,i,0,inf);
       Is |x|-1 positive, negative or zero? pos;
       sum:  sum is divergent.  -- an error.
(%i3)   sum(x^i,i,0,inf);
       Is |x|-1 positive, negative or zero? neg;
(%o1)
```
$$\frac{1}{1-x}$$

Function *nusum* has a different strategy of informing the user about the conditions for convergence and divergence of the series.

```
(%i1)   nusum(x^i,i,0,inf)$
(%i2)   ChangeSign(ChangeSign(%,2),2,2);
(%o2)
```
$$\frac{x^{\infty+1}}{x-1}+\frac{1}{1-x}$$

21.2.3 Power series

powerseries (expr, x, a) [function]

Returns the general form of the power series expansion for *expr* in the variable x about the point a (which may be *inf*). Each time Maxima returns a power series expansion, it creates a new summation index, starting with $i1, i2, \ldots$

If *powerseries* is unable to expand *expr*, *taylor* may be used to give the first several terms of the series.

```
(%i1)   powerseries(sin(x),x,0);
(%o1)
```
$$\sum_{i1=0}^{\infty} \frac{(-1)^{i1}\, x^{2\,i1+1}}{(2\,i1+1)!}$$

When *verbose* is *true*, *powerseries* prints progress messages before returning the result.

```
(%i2)   verbose:true$  powerseries(log(sin(x)/x),x,0);
trigreduce: failed to expand.
```
$$\log\left(\frac{\sin(x)}{x}\right)$$

```
trigreduce: try again after applying rule:
```
$$\log\left(\frac{\sin(x)}{x}\right)=\int \frac{\frac{d}{dx}\frac{\sin(x)}{x}}{\frac{\sin(x)}{x}}\,dx$$

```
powerseries: first simplification returned
```
$$-\int_0^x \frac{\csc(g494)\sin(g494)-g494\cos(g494)\csc(g494)}{g494}\,dg494$$

```
powerseries: first simplification returned
```

$$-\frac{g494\,\cot(g494)-1}{g494}$$

```
powerseries: attempt rational function expansion of
```

$$\frac{1}{g494}$$

(%o3)
$$\sum_{i2=1}^{\infty}\frac{(-1)^{i2}\,2^{2\,i2-1}\,\mathrm{bern}(2\,i2)\,x^{2\,i2}}{i2\,(2\,i2)!}$$

The advanced running index of the g-variable generated by Maxima indicates that during computation the preceding ones have already been used internally.

21.2.4 Taylor and Laurent series expansion

taylor (expr, x, a, p ⟨, 'asymp ⟩) [function]

taylor (expr, [x₁, ..., xₙ], (a | [a₁, ..., aₙ]), (p | [p₁, ..., pₙ]))
taylor (expr, [[x₁, ..., xₙ], (a | [a₁, ..., aₙ]), p ⟨, 'asymp⟩])
taylor (expr, [[x₁, ..., xₙ], a, (p | [p₁, ..., pₙ]) ⟨, 'asymp⟩])

taylor (expr, [x₁, a₁, p₁], ..., [xₙ, aₙ, pₙ])
taylor (expr, x₁, a₁, p₁, ..., xₙ, aₙ, pₙ)

This is the general form of function *taylor*. We will explain the single-variable and the multi-variable forms separately and then the *'asymp* option.

21.2.4.1 Single-variable form

taylor (expr, x, a, p) [function]

This basic form of *taylor* expands the expression *expr* in a truncated Taylor or Laurent series in the variable x around the point a, containing terms through $(x-a)^p$. Maxima precedes the output of a Taylor expansion by a tag /T/ directly after the output tag. (In wxMaxima this is not done, if *taylor* appears on the right side of an assignment.) This indicates that Maxima uses a special internal representation for this type of data. (The CRE form is yet another special internal data format, tagged with /R/ in Maxima output.)

```
(%i1)   taylor(sqrt(x+1),x,0,3);
```

(%o3) /T/
$$1+\frac{x}{2}-\frac{x^2}{8}+\frac{x^3}{16}+\ldots$$

We can evaluate both the original function and Taylor expansions of various orders at a point near a with function *at* to see how the approximation proceeds.

```
(%i1)   t1:taylor(sqrt(x+1),x,0,1);
        t2:taylor(sqrt(x+1),x,0,2);
        t3:taylor(sqrt(x+1),x,0,3);
        t5:taylor(sqrt(x+1),x,0,5);
        at([t1,t2,t3,t5,sqrt(x+1)],x=0.3);
```

(%o1) /T/ $1+\frac{x}{2}+\ldots$

(%o2) /T/ $1+\frac{x}{2}-\frac{x^2}{8}+\ldots$

(%o3) /T/ $1+\frac{x}{2}-\frac{x^2}{8}+\frac{x^3}{16}+\ldots$

(%o4) /T/ $1+\frac{x}{2}-\frac{x^2}{8}+\frac{x^3}{16}-\frac{5x^4}{128}+\frac{7x^5}{256}+\ldots$

(%o5) `[1.15, 1.13875, 1.1404375, 1.1401875390625, 1.140175425099138]`

21.2.4.2 Multi-variable form

taylor (expr, $[x_1,\ldots,x_n]$, $(a \mid [a_1,\ldots,a_n])$, $(p \mid [p_1,\ldots,p_n])$*)*

This basic multi-variable form of *taylor* expands *expr* in the variables $x_1,\ldots,x_n$ about the point $(a_1,\ldots,a_n)$, up to combined powers of p or up to combined powers of max(p_i) for $i=1,\ldots,n$. (Note that here p is not equal to the number of terms as it is in the single-variable form.) If a is identical for all i, it can be given as a single simple variable instead of a list. Thus, a means the point $\underbrace{(a,\ldots,a)}_{n \text{ times}}$.

taylor (expr, $[x_1,a_1,p_1],\ldots,[x_n,a_n,p_n]$*)*

(The square brackets can be omitted.) This form is not only syntactically different from the preceding one, but it also gives a different result, because *expr* is expanded up to the power p_i for variable x_i, $i=1,\ldots,n$. Furthermore, terms are not factored according to combined powers as in the preceding form, but according to powers of the first, second, third, ... variable.

```
(%i1)  taylor(sin(x+y),[x,y],0,5);
       expand(%);
```

(%o1) /T/ $y+x-\frac{x^3+3yx^2+3y^2x+y^3}{6}+\ldots$

(%o2) $-\frac{y^3}{6}-\frac{xy^2}{2}-\frac{x^2y}{2}+y-\frac{x^3}{6}+x$

```
(%o3)  taylor(sin(x+y),[x,0,2],[y,0,3]);
       expand(%);
```

(%o3) /T/ $y-\frac{y^3}{6}+\ldots+\left(1-\frac{y^2}{2}+\ldots\right)x+\left(-\frac{y}{2}+\frac{y^3}{12}+\ldots\right)x^2+\ldots$

(%o4) $\frac{x^2y^3}{12}-\frac{y^3}{6}-\frac{xy^2}{2}-\frac{x^2y}{2}+y+x$

21.2.4.3 Option 'asymp

The option *'asymp* can be applied to both the single- and the multi-variable form of *taylor*. It returns an expansion of *expr* in negative powers of $x-a$. The highest order term is $(x-a)^{-n}$.

21.2.4.4 Option variables

taylordepth default value: 3 [option variable]

If in *taylor (expr, x, a, p)* the expression *expr* is of the form f(x)/g(x) and g(x) has no terms up to degree p, *taylor* attempts to expand g(x) up to degree $2\,p$. If there are still no non-zero terms, *taylor* doubles the degree of the expansion of g(x) so long as the degree of the expansion is less than or equal to $2^{taylordepth}p$.

Chapter 22

Differentiation

22.1 Differentiation operator *diff*

diff (f, x $\langle, p\rangle$*)* |
diff (f, $x_1, p_1, \ldots, x_n, p_n$*)* [function]

This is the general form of function *diff*, the differentiation operator. We will explain the single-variable and the multi-variable forms separately.

As can be done with many other Maxima functions, *diff* can be applied collectively to a list, vector or matrix. If f is a list, vector or matrix, differentiation will be carried out for every individual component and the return value is a structure equivalent to the structure of f.

22.1.1 Single-variable form

diff (f, x $\langle, p\rangle$*)*

When applied to a function f [1] in one variabe, this form returns $D^p f$, the p-th derivative of f with respect to the variable x.[2] If p is omitted, the first derivative is returned.

When f is a multi-variable function, this form returns $D^p_x f$, the p-th partial derivative of f with respect to the variable x. See section 22.1.2.1.

```
(%i1)   diff(3*x^4*sin(x),x);
(%o1)                        12x^3 sin(x) + 3x^4 cos(x)
```

22.1.1.1 Evaluating $D^p f$ at a point

So far we have only computed $D^p f$. We might want to evaluate it at a point x, i.e. compute the continuous linear map $D^p f(x)$, which in case of a single-variable function is a number, to be multiplied with a given number v: $D^p f(x)v$.

Continuing the above example we obtain with function *at*:

[1] Here we use the term *function* in the mathematical sense, not in the sense of MaximaL. In MaximaL this would be called an *expression*.

[2] See sect. 22.5.4.2 for using a derivative noun form for x.

```
(%i2)   at(%,x=%pi/2);
```

(%o2)
$$\frac{3\pi^3}{2}$$

22.1.2 Multi-variable form

22.1.2.1 Partial derivatives

diff (f, $x_1, p_1, \ldots, x_n, p_n$)

This form returns the mixed partial derivative of function f according to the formula

$$\frac{\partial^{p_1+\cdots+p_n} f}{\partial x_1^{p_1} \ldots \partial x_n^{p_n}}$$

where differention is carried out from right to left with respect to the variables listed in the denominator, starting with variable x_n to the order of p_n. Thus, the above form is equivalent to the nested form *diff (. . . (diff(f, x_n, p_n), . . .), x_1, p_1)*.

Note, however, that according to Schwarz's theorem the order of taking multiple partial derivatives does not matter, if all partial derivatives of f up to the desired degree are continuous. This regularity of f can be assumed in most cases.

22.1.2.1.1 Hessian

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

Function *hessian* can be used to compute the (symmetric) Hessian matrix of the second partial derivatives of function f with respect to the list of variables $[x_1, \ldots, x_n]$ according to the scheme

$$H_f = \begin{pmatrix} \partial_1\partial_1 f & \cdots & \partial_1\partial_n f \\ \vdots & \ddots & \vdots \\ \partial_n\partial_1 f & \cdots & \partial_n\partial_n f \end{pmatrix}.$$

We give a rather abstract example using function *depends* to define dependencies of the *undeclared function* f with respect to variables x_1, x_2, x_3.

```
(%i1)   depends (f,[x_1,x_2,x_3]);
```

(%o1) $[f(x_1, x_2, x_3)]$

```
(%i2)   hessian(f,[x_1,x_2,x_3]);
```

(%o2)
$$\begin{pmatrix} \frac{d^2}{dx_1{}^2}f & \frac{d^2}{dx_1\,dx_2}f & \frac{d^2}{dx_1\,dx_3}f \\ \frac{d^2}{dx_1\,dx_2}f & \frac{d^2}{dx_2{}^2}f & \frac{d^2}{dx_2\,dx_3}f \\ \frac{d^2}{dx_1\,dx_3}f & \frac{d^2}{dx_2\,dx_3}f & \frac{d^2}{dx_3{}^2}f \end{pmatrix}$$

22.1.2.2 Total derivative

Function *diff* cannot be used to compute total derivatives of multi-variable functions. However, Maxima can compute the gradient and the Jacobian matrix with other functions. Taylor expansions can be computed, even in the multi-variable case, with function *taylor*.

22.1.2.2.1 Gradient

22.1.2.2.2 Jacobian

jacobian ([$func_1, \ldots, func_n$], [$var_1, \ldots, var_m$]) [function]

Returns the jacobian (Funktionalmatrix) of the list of functions with respect to the list of variables. Dependencies of undeclared functions can be declared with declare.

Application example in *ArensGM s.936-937 Bsp. zur Transformationsformel.wxm*.

22.2 Evaluate expr at a point x with *at*

at ($\left(expr \mid [expr_1, \ldots, expr_n]\right), \left(x = a \mid [x_1 = a_1, \ldots, x_n = a_n]\right)$) [function]

expr or the expressions in the list are evaluated with its variables assuming the values as specified in *eqn* or the list of equations. *eqn* has the form variable=value. *at* carries out multiple substitutions in parallel. If *atvalues* have been defined previously, they are recognized. For an example see sect. 24.2.1.2.2.

22.3 Define value c of expr at a point x with *atvalue*

atvalue (*expr*, $\left(x = a \mid [x_1 = a_1, \ldots, x_n = a_n]\right)$, *c*) [function]

In the single-variable case, *atvalue* assigns the value c to *expr* at the point x = a. In the multi-variable case, the value c is assigned to *expr* at the point specified by the corresponding list of equations. *expr* is a function evaluation, $f(x_1, \ldots, x_m)$, or a derivative given in the form $diff(f(x_1, \ldots, x_m), x_1, p_1, \ldots, x_n, p_m)$, where the function arguments explicitly appear. p_i is the order of differentiation with respect to variable x_i. *atvalue* evaluates its arguments. *atvalue* returns c, which is called the *atvalue*.

The symbols @1, ..., @*n* represent the variables $x_1, \ldots, x_n$ when atvalues are displayed.

```
(%i1)   atvalue(f(x,y),[x=0,y=1],a^2);
```

$$(\%o1) \qquad a^2$$

```
(%i2)   atvalue('diff(f(x,y),x),x=0,1+y);
```

$$(\%o2) \qquad @2 + 1$$

```
(%i3)   at('diff(f(x,y),x)=a,[x=0,y=1]);
```

$$(\%o3) \qquad 2 = a$$

```
(%i4)   at('diff(f(x,y),x)=a,[x=1,y=1]);
```

$$(\%o4) \qquad \frac{d}{dx}\,\mathrm{f}(x, 1)\Big|_{x=1} = a$$

Typically, initial values (IVP) or boundary values (BVP) for solving differential equations are established by this mechanism. For an example see sect. 24.2.2.

printprops (*f* | [$f_1, \ldots, f_n$] | *all*), *atvalue*) [function]

This function displays the atvalues of either function f, the functions defined in the list, or all functions which have atvalues defined by function *atvalue*.

22.4 Evaluation flag *diff*

diff [option variable]

When *diff* is present as an evflag in a call to *ev*, all differentiations indicated in the expression are carried out. For an example see sect. 24.2.1.2.2.

22.5 Noun form differentiation and calculus

We call differentiation of variables or MaximaL functions, for which only functional dependencies are known, *noun form differentiation*. Normal symbolic differentiation and noun form differentiation can be combined in the same function call of *diff*.

If MaximaL functions are used to represent mathematical functions, see below, noun form integration can be accomplished, too. We then generally speak of *noun form calculus*.

22.5.1 Two ways to represent mathematical functions

In Maxima, mathematical functions and functional dependencies can be represented in two fundamentally different ways. As an example comparing both methods see "*Auf Schiene beweglich schwingende Hantel.wxm*".

22.5.1.1 Variables and *depends*

In the first way, MaximaL *variables*, which may be either *unbound* or *bound* (with the *:* operator) to a specific expression being the *value* of this variable, are used to represent mathematical functions, and their dependencies on other variables (which themselves can have dependencies, therefore representing mathematical functions) are explicitly declared with *depends*. When using these variables, their functional dependencies are not immediately visible to the user, since they are not following the variable name in parentheses like they do in the other way described below, when MaximaL functions are used instead. Declared dependencies of variables can only be made visible by using system variable *dependencies*.

This way to implement mathematical functional dependencies in MaximaL is often easier, mathematical expressions are visually shorter and clearer. However, dependencies established with *depends* are recognized only by *diff*, not by *integrate* or other MaximaL functions.

22.5.1.2 MaximaL functions

In the second way, MaximaL *functions* are used to represent mathematical functions. These MaximaL functions can be either *undeclared* or *declared* (with the *:=*

operator) to be a specific expression (here we don't speak of the *value* of a function, but of its *definition*). Dependencies of the functions are now established implicitly by their arguments, which have to be provided both in the *function definition* and in the *function call*.

Note that one and the same symbol can be used in parallel both as a variable and a function; it can have a double life. This is possible because MaximaL functions are implemented as properties, not as values. The user's possibility to make deliberate use of this double life certainly is one of Maxima's highlights.

This way to represent mathematical functions and functional dependencies is generally preferable. It enables the user to easily use noun form differentiation *and* noun form (indefinite or definite) integration, thus freely employing the fundamental theorem of calculus.

The drawback of this method is its more complicated handling. For instance, if (part of) the return value of some computation is to become the function block of a new function, this has to be done according to the following example

```
(%i1)   solve(x(t)^2+y(t)^2=a^2,x(t));
```

$$(\%o1) \qquad [x(t) = -\sqrt{a^2 - y(t)^2}, x(t) = \sqrt{a^2 - y(t)^2}]$$

```
(%i2)   rhs(%[2])$     /* An extra line is necessary here. */
(%i3)   x(t):='%;      /* Quote-quote operator needed for evaluation. */
```

$$(\%o3) \qquad x(t) := \sqrt{a^2 - y(t)^2}$$

Furthermore, careful attention has to be paid for symbols that need to be quoted in the function block, e.g. the second parameter in functions *diff* or *integrate*.

22.5.2 Functional dependency with *depends*

depends $((y_1 \mid [y_{1a}, \ldots, y_{1k}]), (x_1 \mid [x_{1a}, \ldots, x_{1m}]), \ldots, y_n, x_n)$ | [function]
dependencies $(y_1(x_{1a}, \ldots, x_{1k}), \ldots, y_n(x_n))$ [function]

System function *depends* declares functional dependencies among variables for the purpose of computing derivatives. A variable y can be declared to depend on variable x. Although this functional dependency is returned by *depends* as $y(x)$, y is not an *undeclared function*, but remains a variable. If a function y(x) has been defined (or even if it is an undeclared function), *depends* does not refer to the function y(x), but the variable y (variable and function of the same name can coexist).

In the absence of a dependency declared with *depends*, *diff (y, x)* yields zero. If *depends (y, x)* has been performed, *diff (y, x)* yields a symbolic derivative, that is, a noun form.

The arguments to function *depends* are pairs consisting of a variable y and a variable x on which y shall depend. Any of these elements can be a list (of functions or variables respectively). *depends (y, x)* declares f to depend on x. When using a list for the arguments, either several functions can be declared to depend on a

common variable, or a function can be declared to depend on multiple variables, or these features are even combined.

dependencies $(y_1(x_1,\ldots,x_k),y_2(z))$ is equivalent to *depends* $(y_1,[x_1,\ldots,x_k],y_2,z)$.

depends evaluates its arguments and returns a list of the dependencies established. The dependencies are appended to the global variable *dependencies*.

depends (y, x) returns an error, if y is bound. However, y can be bound after *depends (y, x)* has been executed. Alternatively, a bound variable y can be declared to depend on x by using a noun form of y: *depends ('y,x)*.

diff is the only MaximaL system function which recognizes functional dependencies established with *depends*; *integrate* or other functions don't recognize dependencies established for variables. *diff* uses the chain rule when it encounters indirect functional dependencies. Note that in the last line the differentiation is not carried out, because here x is not regarded as a variable, although it has been evaluated as such, but as an *undeclared function*.

```
(%i1)   depends([x,y,r,θ],t);
(%o1)                        [x(t),y(t),r(t),θ(t)]
(%i2)   x:r*cos(θ)$
(%i3)   y:r*sin(θ)$
(%i4)   diff(x,t);
```

$$\left(\frac{d}{dt}r\right)\cos(\theta)-r\sin(\theta)\left(\frac{d}{dt}\theta\right)$$

```
(%i5)   diff(y,t);
```

$$r\cos(\theta)\left(\frac{d}{dt}\theta\right)+\left(\frac{d}{dt}r\right)\sin(\theta)$$

```
(%i6)   diff(x(t),t);
```

$$\frac{d}{dt}(r\cos(\theta))(t)$$

dependencies [system variable]

The system variable *dependencies* contains the list of symbols which have functional dependencies assigned by depends, the function *dependencies*, or *gradef*. The *dependencies* list is cumulative: each call to *depends*, function *dependencies*, or *gradef* (of a variable) appends additional items. The default value of *dependencies* is [].

remove (f, dependency) [function]

Removes all dependencies declared for f.

*Kill*ing a symbol removes it and its dependencies.

22.5.3 Using MaximaL functions

22.5.3.1 Distinction between function and variable

A variable f which has been declared a functional dependency with *depends* remains, although this dependency is returned by Maxima as *f(x)*, a variable and is

not an *undeclared function f(x)*. A MaximaL function is denoted as a symbol followed by its arguments in parentheses, both in the function definition and in the function call. Whether declared or not, a function is treated in a completely different way by Maxima than a variable. In fact, a symbol f can mean a variable f and a function f(x) at the same time, the variable being bound or not, and the function being defined or undefined.

22.5.3.2 Declared function

A *function definition* alone, section 31.2, like *f(x):=3*x* does not yet create a mathematical functional dependency *f(x)*. In the function definition, x is just a formal parameter, which could be replaced by any other. The actual functional dependency is only established by the *function call f(x)*. For instance, *diff (f(x), x)* returns 3, because f(x) evaluates to 3*x. *diff (f(y), y)* also returns 3, because f(y) evaluates to 3*y. But *diff (f(y), x)* returns zero. *diff (f, x)* also returns zero, because f refers to the variable f (which here we assume unbound), not the defined function f(x).

22.5.3.3 Undeclared function

An *undeclared function*, as the name implies, is not defined. The *call f(x)* of an undeclared function, however, establishes a functional dependency which is recognized by *diff* and any other MaximaL function, e.g. *integrate*. In this case, *diff* always returns a noun form; the Leibniz quotient is never evaluated. However, it may be simplified.

```
(%i1)   diff(f(t),t);
```

$$\frac{d}{dt}\,\mathrm{f}(t)$$

```
(%i1)   diff(a*f(t),t,2)$ Pr()$
```

$$\frac{d^2}{dt^2}\,(a\,\mathrm{f}(t)) = a\left(\frac{d^2}{dt^2}\,\mathrm{f}(t)\right)$$

Undeclared functions can be used in an expression assigned to a variable or used in a function definition. If this variable or function is differentiated, *diff* recognizes the indirect functional dependencies and uses the chain rule, see example in the following section.

22.5.3.4 Function call as the independent variable in *diff*

In a call of *diff* we can even use a function call as the variable with respect to which is to be differentiated.

```
(%i1)   g(x):=f(x)^2$
(%i2)   diff(g(x),x);
```

$$2\,\mathrm{f}(x)\left(\frac{d}{dx}\,\mathrm{f}(x)\right)$$

```
(%i3)   diff(g(x),f(x));
```

$$2\,\mathrm{f}(x)$$

If this function has been declared, it has to be quoted in case it shall not be evaluated immediately. For quoting function calls see sect. 31.2.2.1. The following example illustrates the chain rule.

```
(%i4)   f(x):=3*x$
(%i5)   g(x):=x^2$
(%i6)   diff(g(f(x)),x);
```

(%o6) $$18x$$

```
(%i7)   diff(g('f(x)),'f(x)) * diff(f(x),x);
```

(%o7) $$6\,\mathrm{f}(x)$$

```
(%i8)   ev(%, nouns);
```

(%o8) $$18x$$

22.5.4 Using derivative noun forms in *diff*

Derivative noun forms can be used in *diff* both in the expression to be differentiated and as the variable with respect to which is to be differentiated.

22.5.4.1 Differentiating derivative noun forms

Derivative noun forms can themselves be differentiated, that is, they can appear in the expression to be differentiated. Both the derivative noun form used in the expression and the outer function call of *diff* have to be quoted, unless its respective dependencies have been declared with *depends*.

```
(%i1)   'diff('diff(r,t),s);
```

(%o1) $$\frac{d^2}{ds\,dt}r$$

22.5.4.2 Differentiation with respect to derivative noun form

We can use *diff* to differentiate an expression with respect to a derivative noun form. The derivative noun form has to be quoted, unless its respective dependency has been declared with *depends*.

```
(%i1)   T:  (m*r^2*('diff(θ,t,1))^2+m*('diff(r,t,1))^2)/2
```

(%o1) $$\frac{m\,r^2\left(\frac{d}{dt}\theta\right)^2+m\left(\frac{d}{dt}r\right)^2}{2}$$

```
(%i2)   diff(T,'diff(r,t));
```

(%o2) $$m\left(\frac{d}{dt}r\right)$$

22.5.5 Quoting and evaluating noun calculus forms

In general, when using noun calculus forms, special attention has to be paid to whether these noun forms or their arguments, e.g. the second parameter in functions *diff* or *integrate*, are quoted or not, or whether they need to be quoted or not.

Noun calculus forms returned by Maxima are always quoted. If they are to be evaluated or simplified, *nouns* has to be added as an argument to *ev*.

```
(%i1)   eq1:l*(diff(q(t),t,2))+g*q(t)+(diff(x_1(t),t,2))=0;
```

```
(%o1)
```
$$l\left(\frac{d^2}{dt^2}q(t)\right)+\frac{d^2}{dt^2}x_1(t)+gq(t)=0$$

```
(%i2)  x_1(t):=-(l*m_2*q(t))/(m_2+m_1);
```

```
(%o2)
```
$$x_1(t):=\frac{-lm_2q(t)}{m_2+m_1}$$

```
(%i3)  ev(eq1);
```

```
(%o3)
```
$$\frac{d^2}{dt^2}\left(-\frac{lm_2q(t)}{m_2+m_1}\right)+l\left(\frac{d^2}{dt^2}q(t)\right)+gq(t)=0$$

```
(%i4)  ev(eq1,nouns);
```

```
(%o4)
```
$$-\frac{lm_2\left(\frac{d^2}{dt^2}q(t)\right)}{m_2+m_1}+l\left(\frac{d^2}{dt^2}q(t)\right)+gq(t)=0$$

22.6 Defining (partial) derivatives with gradef

gradef (f(x₁, ..., xₙ), g₁, ..., gₙ) |
gradef (v, x, g) [function]

Defines the partial derivatives (i.e., the components of the gradient) of function f or variable v. Such definitions are needed when a function is not known explicitly but its first derivatives are and, for example, we want to have Maxima apply the chain rule or obtain higher order derivatives.

The first form defines df/dx_i as g_i, where g_i is an expression; g_i may be a function call, but not the name of a function (this means: the function has to be given with arguments). The number of partial derivatives m may be less than the number of arguments n, in which case derivatives are defined with respect to x_1 through x_m only.

Partial derivatives cannot be defined for a function already defined, and the definition of a function for which a derivative is already defined with *gradef* overwrites this definition of the derivative. However, partial derivatives can be defined for the noun form of a function already defined, see example. A derivative can be defined for a variable which is bound, see example. Trying to define a derivative of the noun form of a variable (whether bound or not) causes an error.

The second form defines the derivative of variable v with respect to variable x as expr. This also establishes the dependency of v on x as *depends(v, x)*. The variable may already be bound.

The first argument $f(x_1, \ldots, x_n)$ or v is quoted, but the remaining arguments $g_1, \ldots, g_m$ or x, g are evaluated. *gradef* returns the function or variable for which the partial derivatives are defined.

gradef can define gradients for only one function or one variable at a time.

gradef can redefine the derivatives of Maxima's built-in functions.

gradef cannot define partial derivatives for a subscripted function.

In the following example we use *gradef* to make Maxima apply the chain rule when differentiating the undeclared function g(x,y) with respect to x.

```
(%i1)   diff(g(3*x-2*y),x);
```

$$(\%o1) \qquad \frac{d}{dx}\,g(3x-2y)$$

```
(%i2)   gradef(g(z),G(z));
(%o2)                                g(z)
(%i3)   diff(g(3*x-2*y),x);
(%o1)                           3*G(3*x-2*y)
```

gradef cannot define partial derivatives for a function already defined. (Defining the derivatives before defining the function does not help, because the latter overwrites the derivatives defined by *gradef*.) However, partial derivatives can be defind for a noun form of a function already defined. In this case, the defined function and its noun form will behave differently under *diff*, the noun form not resulting in the usual symbolic Leibniz notation, but in the expression defined by *gradef*.

```
(%i1)   f(x):=x^2;
```

$$(\%o1) \qquad f(x) := x^2$$

```
(%i2)   diff(f(x),x);
(%o2)                                2x
(%i3)   diff(f(log(x)),x);
```

$$(\%o3) \qquad \frac{2\log(x)}{x}$$

```
(%i4)   gradef(f(x),y(x));  /* This returns no error but has no effect. */
(%o4)                               f(x)
(%i5)   diff(f(x),x);
(%o5)                                2x
(%i6)   diff('f(x),x);
```

$$(\%o6) \qquad \frac{d}{dx}\,f(x)$$

```
(%i7)   gradef('f(x),y(x));
(%o7)                               f(x)
(%i8)   diff('f(x),x);  /* The result differs from (%o2) and (%o6). */
(%o8)                               y(x)
(%i9)   diff('f(log(x)),x);
```

$$(\%o9) \qquad \frac{y(\log(x))}{x}$$

```
(%i10)  ev(diff('f(log(x)),x),f);
```

$$(\%o10) \qquad \frac{2\log(x)}{x}$$

```
(%i11)  ev(diff('f(log(x)),x),nouns);
```

$$(\%o11) \qquad \frac{2\log(x)}{x}$$

In case of a variable, however, *gradef* can be defined for a bound variable. Then, again, the bound variable and its noun form will behave differently under *diff*.

```
(%i1)   depends ([r,v,e_r,e_v],t)$
(%i2)   r_: r*e_r$
(%i3)   gradef(r_,t,e_v*r*(diff(v,t,1))+e_r*(diff(r,t,1)))$
(%i4)   diff(r_,t);
```

(%o4) $$e\_r\left(\frac{d}{dt}r\right)+\left(\frac{d}{dt}e\_r\right)r$$

```
(%i5)   diff('r_,t);
```

(%o5) $$e\_v\,r\left(\frac{d}{dt}v\right)+e\_r\left(\frac{d}{dt}r\right)$$

22.6.1 Show existing definitions

printprops ([$f_1,\ldots,f_n$], ,gradef) |
printprops ([$v_1,\ldots,v_n$], atomgrad) [function]

The first form displays the partial derivatives of the functions $f_1,\ldots,f_n$ as defined by gradef. The second form displays the partial derivatives of the variables $v_1,\ldots,v_n$ as defined by gradef.

gradefs [system variable]

gradefs is the list of the *functions* for which partial derivatives have been defined with *gradef*. This list does not include any *variables* for which partial derivatives have been defined with *gradef*.

22.6.2 Remove definitions

remove (f, [dependency,gradef]) |
remove (v, [dependency,atomgrad]) [function]

The first form removes the definition of function f, the second one removes the definition of variable v.

kill (gradefs) [function]

Removes all entries from the list of the system variable *gradefs*, i.e. removes all *gradef* definitions present.

22.7 Gradient

Grad (f, n ,$\left([v_1,\ldots,v_n]\,\middle|\,x\right)$*)* [function of cartesian_coordinates]

Computes the gradient of the n-dim. scalar field f. The last parameter gives a list of the variable names or a singe variable name, if the variables of f are elements of an undeclared array of that name. *Grad* returns an n-dim column vector.

Chapter 23

Integration

integrate(1/x,x); => log(x)
integrate(1/x,x), logabs; => log(|x|)

Chapter 24

Differential Equations

24.1 Introduction

24.1.1 Overview

Only a small portion of the ODEs encounterd in research and engineering have known exact solutions and can be solved by analytical methods. Even if they can, sometimes their solutions involve complicated expressions with special functions and are of no real help. In this case, the user might prefer to look for an approximate numerical solution instead.

Maxima cannot solve PDEs.

24.1.1.1 Analytical methods

Maxima provides function *ode2* to analytically find the general solution of elementary (not necessarily linear) ODEs of first or second order. On the basis of this solution, the initial value problem can be solved by *ic1* or *ic2*, depending on the order of the ODE. Function *bc2* solves the boundary value problem.

In addition, David Billinghurst has developed a new package *contrib_ode* which employs some more methods for solving first order ODEs and linear homogeneous second order ODEs.

A linear ODE of order n or a system of such ODEs can be solved by *desolve* which uses Laplace transformation.

Maxima cannot solve nonlinear ODEs of higher order by analytical means. Note that any (system of) higher order ODE(s) can be transformed into a system of first order ODEs. If this resulting system is linear, it can possibly be solved with *desolve*.

24.1.1.2 Numerical methods

24.1.1.3 Graphical methods

24.2 Analytical solution

24.2.1 Ordinary differential equation (ODE) of 1. or 2. order

24.2.1.1 Find general solution

24.2.1.1.1 ode2

ode2 (eq, depvar(indepvar), indepvar) [function]

Solves an ordinary differential equation (ODE) of first or second order. *ode2* takes three arguments: the ODE (not necessarily in explicit form) given by *eq*, the dependent variable *depvar*, and the independent variable *indepvar*. Note that indepvar has to be specified as the argument of depvar again. When successful, *ode2* returns either an explicit or implicit solution for the dependent variable. *%c* is used to represent the integration constant in the case of first-order equations, *%k1* and *%k2* are the constants for second-order equations.

Derivatives have to be specified in *eq* without quoting function *diff*. The dependence of the dependent variable and its derivative(s) on the independent variable always has to be indicated in *eq*, as in the case of function *desolve*. E.g. in *eq* the dependent variable is written as *y(x)* and not as *y*, and its derivative is written as *diff(y(x),x)* and not as *diff(y,x)*.

If ode2 cannot obtain a solution for whatever reason, it returns *false*, after perhaps printing out an error message.

The methods implemented for *first order ODEs*, in the order in which they are tested, are: linear, separable, exact (perhaps requiring an integrating factor), homogeneous, Bernoulli's equation, and a generalized homogeneous method.

The types of *second-order ODEs* which can be solved are: constant coefficients, exact, linear homogeneous with non-constant coefficients which can be transformed to constant coefficients, the Euler or equi-dimensional equation, equations solvable by the method of variation of parameters, and equations which are free of either the independent or of the dependent variable so that they can be reduced to two first order linear equations to be solved sequentially.

In the course of solving ODE's, several variables are set purely for informational purposes: *method* denotes the method of solution used (e.g., linear), *intfactor* denotes any integrating factor used, *odeindex* denotes the index for Bernoulli's method or for the generalized homogeneous method, and *yp* denotes the particular solution for the variation of parameters technique.

Note that Maxima does not take into consideration the domain of the independent variable, or whether it is simply connected. Neither does Maxima return the precise domain of a solution or any singularities. All his has to be worked out manually, if necessary.

As an example, we wish to solve the following ODE describing a free harmonic

oscillator without damp, Satz 5.10, given by

$$\frac{d^2\varphi}{dt^2} + \frac{g}{l}\varphi(t) = 0 \tag{24.1}$$

under the assumptions that $g, l > 0$. We will continue this example for an IWP to be solved by *ic2*, and later we will solve equation and IWP again with the help of *desolve* to show the differences.

```
(%i1)   assume(g>0,l>0)$
(%i2)   eq:diff(φ(t),t,2)+g/l*φ(t);
```

$$(\%o2) \qquad \frac{d^2}{dt^2}\varphi(t) + \frac{g}{l}\varphi(t)$$

```
(%i3)   gensol: ode2(eq,φ(t),t),rootscontract;
```

$$(\%o3) \qquad \varphi(t) = \%k1 \sin\left(\sqrt{\frac{g}{l}}t\right) + \%k2 \cos\left(\sqrt{\frac{g}{l}}t\right)$$

Note that one side of the equation can be omitted, because it is zero, see sect. 9.5. Note also that the solution returned by ode2 is not an assignment, it does not bind the variable φ. The constants *%k1*, *%k2* inserted by Maxima can be specified by solving an initial value problem, see function *ic2*, or a boundary value problem, see function *bc2*.

24.2.1.1.2 contrib_ode

contrib_ode (eq, depvar, indepvar) [function of contrib_ode]

This function makes uses of more methods than *ode2* for solving linear and non-linear first order ODEs as well as linear homogeneous second order ODEs.

load('contrib_ode)

24.2.1.2 Solve initial value problem (IVP)

24.2.1.2.1 1. order ODE: ic1

ic1 (gensol, $x = x_0$, $y = y_0$) [function]

Solves an initial value problem (IVP) for a first order ordinary differential equation. The first argument *gensol* is a general solution of the ODE as returned by *ode2*. The second argument specified the name and initial value of the independent variable.[1]The last argument gives the name and the initial value of the dependent variable, where $y_0 = y(x_0)$.

24.2.1.2.2 2. order ODE: ic2

ic2 (gensol, $x = x_0$, $y = y_0$, 'diff$(y,x) = y_1$) [function]

Solves an initial value problem for a second order ordinary differential equation. The first argument *gensol* is a general solution of the ODE as found by *ode2*. The second argument specifies the name and initial value of the independent variable.[1] Then

[1]The value of x_0 does not have to be zero. Any point in the domain of y can be selected.

follows the name and the initial value of the dependent variable, where $y_0 = y(x_0)$. The last argument gives the initial value of the first derivative of the dependent variable with respect to the independent variable, evaluated at x_0.[2]

As an example, we want to solve the IVP for the general solution of the oscillator equation found in the example to function *ode2*, first with initial values $t = 0,\ \varphi(t) = 1,\ \varphi(t)' = 0$, then with $t = 0,\ \varphi(t) = 1,\ \varphi(t)' = 1$.

```
(%i4)   ivpsol: ic2(gensol,t=0,φ=1,'diff(φ,t)=0),rootscontract;
```

(%o4) $$\varphi = \cos\left(\sqrt{\frac{g}{l}}t\right)$$

```
(%i5)   ivpsol: ic2(gensol,t=0,φ=1,'diff(φ,t)=1),rootscontract;
```

(%o5) $$\varphi = \sqrt{\frac{l}{g}}\sin\left(\sqrt{\frac{g}{l}}t\right) + \cos\left(\sqrt{\frac{g}{l}}t\right)$$

As with any result obtained for a differential equation, it should be checked to see whether it is really a solution. We show this for the second case. First we proof that it satisfies the conditions given to *ic2*. In order to avoid error messages, we use *at* rather than *ev* for the derivative.

```
(%i6)   ivpsol,t=0;
(%o6)                                  φ=1
(%i7)   at(diff(rhs(ivpsol),t),t=0),ratsimp;
(%o7)                                   1
```

Then we insert the result in our equation *eq*. The following input is equivalent to *ev*(eq, ivpsol, diff, ratsimp) and causes *rhs(ivpsol)* to be substituted for φ in *eq*, then the differentiations to be carried out and finally a simplification. Note that *diff* here is not the function but an evaluation flag.

```
(%i8)   eq,ivpsol,diff,ratsimp;
(%o8)                                   0
```

The result is the missing *rhs* of *eq*. Therefore, *ivpsol* is a valid solution of *eq*. Since the solution of the IVP is unique, it is the only solution.

Finally we want to solve the IVP at points other than zero. This time, first we select $t = \pi/(2 * sqrt(g/l)),\ \varphi(t) = 1,\ \varphi(t)' = 0$, then $t = \pi/(3 * sqrt(g/l)),\ \varphi(t) = 1,\ \varphi(t)' = 0$. We see that this works just as well.

```
(%i9)   ic2(gensol,t=π/(2*sqrt(g/l)),φ=1,'diff(φ,t)=0),rootscontract;
```

(%o9) $$\varphi = \sin\left(\sqrt{\frac{g}{l}}t\right)$$

```
(%i10)  ic2(gensol,t=π/(3*sqrt(g/l)),φ=1,'diff(φ,t)=0)$
(%i11)  PullFactorOut([%,2,1],sqrt(3)/2)$
(%i12)  PullFactorOut([%,2,2],1/2),rootscontract;
```

[2] Although quoting diff is not absolutely necessary, it is recommended, because sometimes this will prevent unexpected errors from occurring.

(%o12)
$$\varphi = \left(\frac{\sqrt{3}}{2}\right)\sin\left(\sqrt{\frac{g}{l}}t\right) + \left(\frac{1}{2}\right)\cos\left(\sqrt{\frac{g}{l}}t\right)$$

The result can be checked in the same way as we demonstrated above.

In sect. 24.2.2 we will demonstrate this same example using function *desolve*, and in sect. 27.1.2 we redo it with *laplace*.

We see that the oscillation function φ returned can be a cos or a sin, but generally is a combination of both, with an angular frequency being identical for the sin and the cos, and factors depending on the specific IVP. In sect. 14.3.1 we will continue this example using pattern matching to transform the oscillation function as returned by *ic2* into the form

$$\varphi = A\sin(\omega t + \alpha)$$

with the amplitude A, the angular frequency ω and the phase shift α.

24.2.1.3 Solve boundary value problem (BVP): bc2

For a first order ODE, the boundary value problem (BVP) is equivalent to the initial value problem.

bc2 (gensol, ival1, depval1, ival2, depval2) [function]

Solves a boundary value problem for a second order differential equation. The first argument *gensol* is a general solution to the equation as found by *ode2*; *ival1* specifies the value of the independent variable in a first point, in the form $x = x_1$, and *depval1* gives the value of the dependent variable in that point, in the form $y = y_1$. The expressions *ival2* and *depval2* give the values for these variables at a second point, using the same form.

24.2.2 System of linear ODEs: desolve

desolve (eq, y(x)) | [function]
desolve ([eq$_1$, ..., eq$_m$],[y$_1$(x), ..., y$_m$(x)])

Solves one or a system of linear ordinary differential equations of order n using Laplace transform. The first argument gives one or a list of differential equations being in the dependent variables y or $y_1, \ldots, y_m$, each of them depending on the independent variable x. Derivatives are given in the equation by quoting function *diff*. The independent variable is not given explicitely as a third argument, as in the case of function *ode2*, but instead, the functional dependence of the dependent variable(s) and its derivative(s) on the independent variable must be indicated both in the equations and in *desolve*'s second argument. E.g. the dependent variable is written as *y(x)* and not as *y*, and its derivative is written as *'diff(y(x),x)* and not as *'diff(y,x)*. If *desolve* cannot obtain a solution, it returns *false*.

desolve returns a general solution specifying integration constants in terms of symbolic initial values of the dependent variables and their derivatives at t=0, with t being the independent variable. If an initial or boundary value problem is to be solved, these values can be defined with *atvalue* prior to calling *desolve*. Note that

because *desolve* uses Laplace transform, this is only possible for initial or boundary conditions specified at t=0, with t being the independent variable. If one has initial or boundary conditions imposed elsewhere, one can impose these on the *general* solution returned by *desolve* and eliminate the constants by solving the general solution for them and substituting their values back. Let's demonstrate all this with the same example we used for *ode2* and *ic2*. First we use initial values $t = 0,\ \varphi(t) = 1,\ \varphi(t)' = 1$.

```
(%i1)   assume(g>0,l>0)$
(%i2)   eq: 'diff(φ(t),t,2)+g/l*φ(t);
```

$$(\%o2)\qquad \frac{d^2}{dt^2}\,\varphi(t)+\frac{g\,\varphi(t)}{l}$$

```
(%i3)   gensol: desolve(eq,φ(t)),expand,rootscontract;
```

$$(\%o3)\qquad \varphi(t)=\sqrt{\frac{l}{g}}\,\sin\left(\sqrt{\frac{g}{l}}\,t\right)\left(\frac{d}{dt}\,\varphi(t)\Big|_{t=0}\right)+\varphi(0)\cos\left(\sqrt{\frac{g}{l}}\,t\right)$$

```
(%i4)   atvalue(φ(t),t=0^3,1)$  atvalue('diff(φ(t),t),t=0^3,1)$
(%i5)   desolve(eq,φ(t)),expand,rootscontract;
```

$$(\%o5)\qquad \varphi(t)=\sqrt{\frac{l}{g}}\,\sin\left(\sqrt{\frac{g}{l}}\,t\right)+\cos\left(\sqrt{\frac{g}{l}}\,t\right)$$

The result can be checked in the same way as it was demonstrated for *ic2*.

Now we will show how an IVP with initial values at a point other than zero can be solved, selecting $t = \pi/(3 * sqrt(g/l)),\ \varphi(t) = 1,\ \varphi(t)' = 0$, like in the example of *ic2*. Suppose that in the above example we have come untill (%o3), which is the general solution, but with two unknown variables $\varphi(0)$ and $\varphi'(0)$. We proceed as follows: First we specify our atvalues $\varphi(t)$ and $\varphi'(t)$. With the first of them we go into gensol, evaluated at t. Then we differentiate gensol and go into the result, evaluated at t again, with $\varphi'(t)$. This provides us two equations for the two unknown variables, which we can now solve and substitute back into the general solution. For this last step we use *subst*; *at* doesn't work properly in this case, since one of the variables is itself a noun *at* form. Note also, that we can't use *ev* instead of *at* for the evaluations of *gensol* and *dgensol* at t, also due to the noun *at* form. In an expression like *gensol, t=π/(3*sqrt(g/l));* the rhs of the equation would be substituted for t everywhere in *gensol*, including in the noun *at* form. This destroys it and makes it evaluate to zero, giving an incorrect overall result. Here we see that the three seemingly equivalent methods of evaluation at a point (ev, at, subst) have subtle differences that want to be considered carefully.

```
(%i4)   atvalue(φ(t),t=t=π/(3*sqrt(g/l)),1)$
(%i5)   atvalue('diff(φ(t),t),t=t=π/(3*sqrt(g/l)),0)$
(%i6)   at(gensol,t=π/(3*sqrt(g/l))),ratsimp$  expand(%)$
(%i7)   PullFactorOut([%,2,1],sqrt(3)/2*sqrt(l/g))$
(%i8)   at1sol: PullFactorOut([%,2,2],1/2);
```

$$(\%o8)\qquad 1=\left(\frac{\sqrt{3}\,\sqrt{l}}{2\sqrt{g}}\right)\left(\frac{d}{dt}\,\varphi(t)\Big|_{t=0}\right)+\varphi(0)\left(\frac{1}{2}\right)$$

[3]The value of t here has to be zero, because *desolve* uses Laplace transform.

```
(%i9)   dgensol:diff(gensol,t);
```

(%o9) $$\frac{d}{dt}\varphi(t)=\sqrt{\frac{g}{l}}\sqrt{\frac{l}{g}}\cos\left(\sqrt{\frac{g}{l}}t\right)\left(\frac{d}{dt}\varphi(t)\Big|_{t=0}\right)-\varphi(0)\sqrt{\frac{g}{l}}\sin\left(\sqrt{\frac{g}{l}}t\right)$$

```
(%i10)  at(dgensol,t=π/(3*sqrt(g/l))),ratsimp$
(%i11)  at2sol:2*sqrt(l)*%;
```

(%o11) $$0=\sqrt{l}\left(\frac{d}{dt}\varphi(t)\Big|_{t=0}\right)-\sqrt{3}\varphi(0)\sqrt{g}$$

```
(%i12)  linsolve([rembox(at1sol),at2sol],[φ(0),at('diff(φ(t),t,1),t=0)]);
```

(%o12) $$[\varphi(0)=\frac{1}{2},\ \frac{d}{dt}\varphi(t)\Big|_{t=0}=\frac{\sqrt{3}\sqrt{g}}{2\sqrt{l}}]$$

```
(%i13) subst((sqrt(3)*sqrt(g))/(2*sqrt(l)),at('diff(φ(t),t,1),t=0),gensol)$
(%i14)  subst(1/2,φ(0),%),ratsimp$  expand(%)$
(%i15)  PullFactorOut([%,2,1],sqrt(3)/2)$
(%i16)  PullFactorOut([%,2,2],1/2),rootscontract;
```

(%o16) $$\varphi(t)=\left(\frac{\sqrt{3}}{2}\right)\sin\left(\sqrt{\frac{g}{l}}t\right)+\left(\frac{1}{2}\right)\cos\left(\sqrt{\frac{g}{l}}t\right)$$

In sect. 27.1.2 we solve the same ODE explicitely with *laplace*, thus demonstrating what *desolve* does internally.

24.3 Numerical solution

24.3.1 Runge-Kutta: rk

Package *dynamics* contains function *rk* for numerically solving a (system of) 1. order ODE(s) given in explicit form with the classical forth order *Runge-Kutta method*. This package is loaded automatically when a Maxima session begins.

rk (*eq* | [$eq_1,\ldots,eq_n$]), (*y* | [$y_1,\ldots,y_n$]), (*y*$_0$ | [$y_{01},\ldots,y_{0n}$]), [*x*, *x*$_0$, *x*$_e$, *inc*])
[function]

Numerically solves an initial value problem (IVP) of either a single or a system of 1. order ODE(s) given in explicit form by eq or the list [$eq_1,\ldots,eq_n$], with the dependent variable(s) being y or [$y_1,\ldots,y_n$] and having initial value(s) $y_0=y(x_0)$ or [$y_{01},\ldots,y_{0n}$]. The independent variable x is evaluated in the interval [x_0,x_e] with constant increment *inc*. Any of the dependent variables $y_k, k=1,\ldots,n$, can appear in any of the equations eq_k. The return value of *rk* can be plotted immediately with *plot2d*.

As an example we want to solve the ODE

$$\frac{dy}{dx}=x-y^2$$

in the range of $x\in[0,8]$ with a constant increment of 0.1 for an initial value of $y_0=y(x_0)=y(0)=1$.

```
(%i1)   rk(t-x^2,x,1,[t,0,8,0.1])$
(%i2)   plot2d ([discrete, %])$
```

Figure 24.1 shows the resulting plot. In the next example we solve the system

$$\frac{dy_1}{dx} = 4 - y_1^2 - 4y_2^2 \qquad \text{and} \qquad \frac{dy_2}{dx} = y_2^2 - y_1^2 + 1$$

in the range of $x \in [0, 4]$ with a constant increment of 0.02 for initial values of $y_{01} = -1.25$, $y_{02} = 0.75$ at x=0.

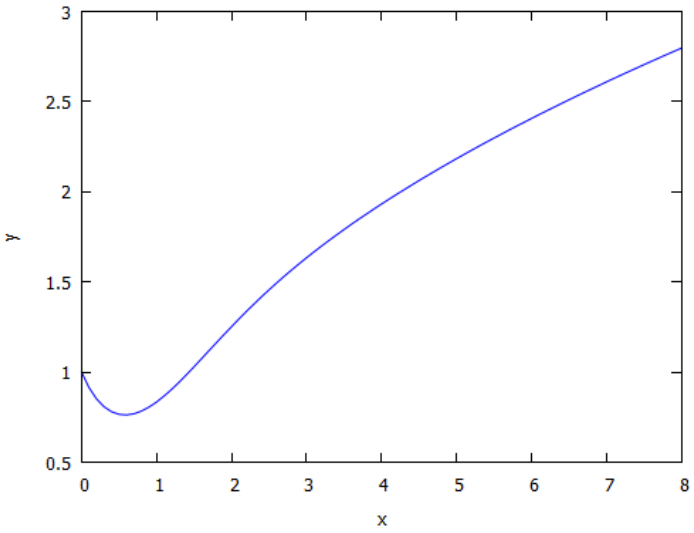

Figure 24.1 – Plot of the return value of function rk having solved one first-order ODE with the Runge-Kutta method.

```
(%i1)   res: rk([4-y1^2-4*y2^2,y2^2-y1^2+1],[y1,y2],[-1.25,0.75],[x
   ,0,4,0.02])$
(%i2)   plot2d ([[discrete, makelist([p[1],p[2]],p,res)],[discrete,
   makelist([p[1],p[3]],p,res)]],[legend,"y_1","y_2"])$
```

Figure 24.2 shows the resulting plot.

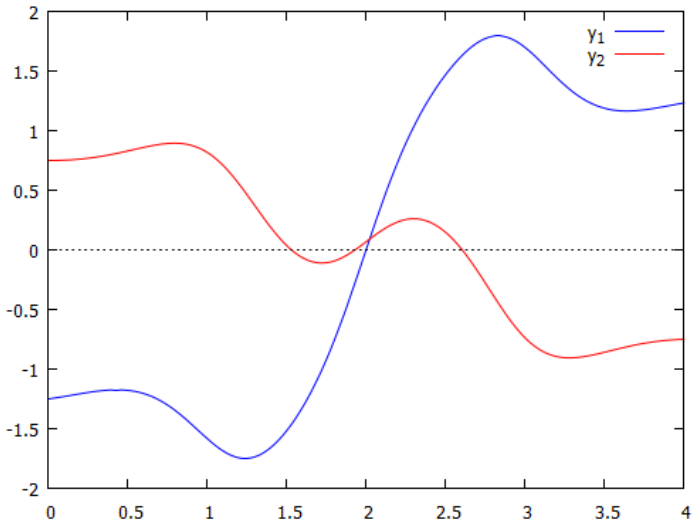

Figure 24.2 – Plot of the return value of function rk having solved a system of two first-order ODEs with the Runge-Kutta method.

24.4 Graphical estimate

24.4.1 Direction field

Direction fields can be plotted either with function *plotdf*, which uses xMaxima, or with function *drawdf*, which uses Gnuplot's *draw*.

24.4.1.1 plotdf

plotdf is a function to plot the direction field of one or two first-order ODEs, possibly together with the specific solution of an initial value problem (IVP). *plotdf* uses xMaxima and depends on it being installed; but it can be used also from other interfaces, e.g. wxMaxima or the console. *plotdf* can export plots only in postscript format (.ps). External programs can be used to transform such files into .jpg or .png format; we use the downloadable freeware *XnConvert*.

plotdf $\left(\left(eq \,\middle|\, [eq_1, eq_2]\right)\langle, [x, y]\rangle\langle, [opt_1], \ldots, [opt_n]\rangle\right)$ [function]

Creates a plot of the direction field of either one first-order ODE or a system of two of them, given in explicit form. In the first case, *eq* is the right-hand side of the ODE

$$y'(x) = F(x, y),$$

while in the second case, the list $[eq_1, eq_2]$ contains the right-hand sides of the ODEs

$$x'(t) = F_1(t, x) \qquad \text{and} \qquad y'(t) = F_1(t, y).$$

In the first case, the second argument provides the names of the independent and the dependent variable; in the second case, it provides the two dependent variables, the independent variable always being t. The second argument can be omitted in either case, if the names are "x" and "y".

The following options, each of them enclosed in a list and separated by commas, can be used:

[trajectory_at,x_0, y_0]: Initial value problem (IVP) with initial values x_0, y_0.

As a first example, we plot the direction field of the first-order ODE

$$y'(x) = e^{-x} y$$

together with an IVP given by $x_0 = 2$, $y_0 = y(x_0) = -0.1$.

```
(%i1)   rk(t-x^2,x,1,[t,0,8,0.1])$
(%i2)   plot2d ([discrete, %])$
```

Figure 24.3 shows the resulting plot.

24.4.1.2 drawdf

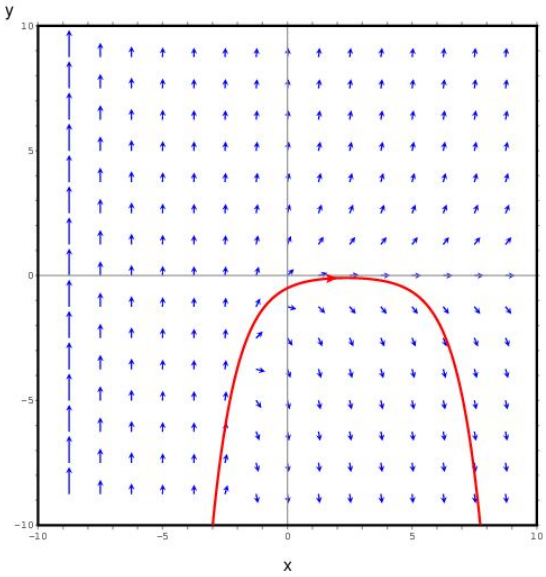

Figure 24.3 – Plot of the direction field of a first-order ODE together with an IVP.

Part V

Special applications

Chapter 25

Analytic geometry

25.1 Representation and transformation of angles

25.1.1 Bring angle into range

RadRange0to2(angle) [function of *rs_angles*]
RadRange1to1(angle) [function of *rs_angles*]
DegRange0to2(angle) [function of *rs_angles*]
DegRange1to1(angle) [function of *rs_angles*]

These functions bring an angle given in radiant into either the range $[0, 2\pi)$ or $(-\pi, \pi]$, and an angle given in degrees into either the range $[0, 360)$ or $(-180, 180]$.

25.1.2 Degrees ⇆ radians

Deg2Rad(degrees, ⟨, n⟨, f⟩⟩) [function of *rs_angles*]
Rad2Deg(radians, ⟨, n⟩) [function of *rs_angles*]

These functions transform an angle from degrees (decimal) to radians and vice versa.

Deg2Rad transforms an angle given in decimal degrees to radians. The result is a term consisting of pi and a factor. *float(Deg2Rad(degrees))* returns a float. If a second argument n>0 is present, the factor of pi is rounded to n digits after the dot. If any third argument is present, Deg2Rad will return a float rounded to n digits after the dot.

Rad2Deg transforms an angle given in radians to decimal degrees. If a second argument $n \geq 0$ is present, in case of n=0 the result is rounded to integer, and in case of n>0 the float result is rounded to n digits after the dot. Note that a float result with maximum precision can be obtained by *float(Rad2Deg(radians, ⟨, n⟩))*.

25.1.3 Degrees decimal ⇆ min/sec

Dec2Min(degrees ⟨, n⟩) [function of *rs_angles*]
Min2Dec([deg,min,sec] ⟨, n⟩) [function of *rs_angles*]
ConcMinSec([deg,min,sec]) [function of *rs_angles*]

Dec2Min converts an angle given in decimal degrees into a list of 3 elements containing degrees, minutes and seconds. The first two elements are integers. If a second argument $n \geq 0$ is present, seconds are rounded to n digits after the dot.

Min2Dec converts an angle given as a list of 3 elements containing degrees, minutes and seconds into decimal degrees. If a second argument $n \geq 0$ is present, the value returned is rounded to n digits after the dot.

ConcMinSec converts an angle given as a list of 3 elements containing degrees, minutes, and seconds into a string with the elements followed by °, ’ and ” respectively.

Chapter 26

Coordinate systems and transformations

26.1 Cartesian coordinates

26.1.1 Extended coordinates

Leng ((*vector* | *squarematrix*), ⟨1 | 0⟩*)* [function of *rs_object_transformation*]
Short ((*vector* | *squarematrix*)*)* [function of *rs_object_transformation*]

Function *Leng* transforms a column vector or square matrix of dimension 2 or 3 to extended coordinates. If the first argument is a column vector, add a new component at the end containing 1 for a location vector and 0 for a direction vector, depending on the optional second argument (1 or 0). If the second argument is omitted, use 1, since a location vector is assumed. A column vector is returned. In case of a square matrix, 0 is appended to any column and row, only the last element of the matrix is set to 1.

The inverse function *Short* transforms a column vector or square matrix of dimension 2 or 3 being in extended coordinates (dimension 3 or 4) back to the non-extended format.

26.1.2 Object transformation

26.1.2.1 Rotation

RotMatrix ((*2D* | *axis*)*, phi,* ⟨*extend*⟩*)* [function of *rs_object_transformation*]

Returns the rotation matrix (in extended coordinates, if 3. argument is present, e.g. as *e*) for an active (object) rotation in 2D, if first argument is *2*, else in 3D around axis *x*, *y*, or *z*, with rotation angle *phi* (given in rad) in the mathematically positive direction (counterclockwise). By giving a negative angle −*phi*, the transformation matrix for a passive (coordinate) rotation in the mathematically positive direction (counterclockwise) by angle *phi* can be created.

26.2 Polar coordinates

26.3 Cylindrical coordinates

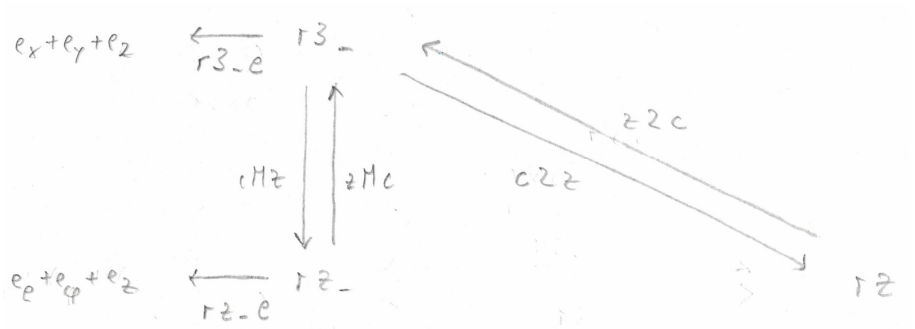

Figure 26.1 – Diagram of representations and transformation functions for cylindrical coordinates from package cylindrical_coordinates.mac.

26.4 Spherical coordinates

26.5 General orthogonal coordinates

Chapter 27

Integral transformation

27.1 Laplace transformation

For historical reasons Maxima has two functions which can compute the Laplace transform, *laplace* and *specint*. While Laplace knows more of the general rules for Laplace transforms and can handle equations, *specint* recognizes some special functions which Laplace doesn't. *laplace* automatically calls *specint*, though, if it cannot compute the Laplace transform, and therefore the user should preferably employ *laplace*.

laplace (expr, t, s) [function]

Computes the Laplace transform of *expr* with respect to the integration variable *t* and the transform parameter *s*. For instance, *laplace(f(t),t,s)* is equivalent to

$$\mathcal{L}f(s) = \int_0^\infty f(t)\, e^{-st}\, dt. \tag{27.1}$$

Note that if *expr* is a function, it has to be expressed in terms of the integration variable t, in the way it appears under the integral sign. However, *expr* can also be an equation, e.g. a differential or integral equation. In this case *laplace* computes the Laplace transform of both sides separately and returns an equation.

Like in desolve and unlike in ode2, functional dependencies must be explicitly specified in *expr*; implicit relations, established by *depends*, are not recognized. For example, if f depends on x and y, f must be written as f(x,y) in *expr*.

laplace recognizes the functions *delta, exp, log, sin, cos, sinh, cosh*, and *erf*, as well as *diff, integrate, sum*, and *ilt*. If *laplace* fails to find a transform, it calls function *specint*. *specint* can find the laplace transform for expressions with special functions like the bessel functions $bessel_j, bessel_i, \ldots$ and can handle the unit_step function. If *specint* cannot find a solution either, a noun laplace is returned.

expr may also be a linear, constant coefficient differential equation. Initial (IVP) or boundary (BVP) values of the dependent variable can be specified with *atvalue* prior to calling laplace. Note that initial conditions for the Laplace transform have to be specified at t=0, with t being the independent variable. If one has initial or boundary conditions imposed elsewhere, one can impose these on the *general* solution returned by *laplace* and eliminate the constants by solving the general solution and possibly its derivative for them and substituting their values back.

laplace recognizes convolution integrals of the form

$$\int_0^t f(t-\tau)\,g(\tau)\,d\tau.$$

Other kinds of convolutions are not recognized.

In the following we give a number of small examples. Worked out examples for solving differential or convolution integral equations are given in sect. 27.1.2.

```
(%i1)   %e^(2*t+a)*sin(t)*t;
(%o1)                              t %e^(2t+a) sin(t)
(%i2)   laplace(%,t,s);
                                        %e^a (2s-4)
(%o2)                                 ------------
                                                 2
                                      (s^2-4s+5)
(%i3)   laplace('diff(f(t),t),t,s);
(%o3)                        s*laplace(f(t),t,s)-f(0)
(%i4)   assume(n>0)$
(%i5)   laplace(t^n,t,s);
                                          gamma(n+1)
(%o5)                                     ----------
                                            s^(n+1)
```

*specint (f(t) * exp(-s * t), t)* [function]

Compute the Laplace transform of function *f(t)* with respect to the variable t. Note that the factor e^{-st} has to be explicitely specified as part of the first argument. The integrand *f(t)* may contain special functions. The following special functions are recognized by *specint*: *incomplete gamma function, error functions* (but not the error function *erfi*; it is easy to transform *erfi* e.g. to the error function *erf*), *exponential integrals, bessel functions* (including products of bessel functions), *hankel functions*, *hermite* and *laguerre polynomials*. Furthermore, *specint* can handle the *hypergeometric function* %f[p,q]([],[],z), the *whittaker function* of the first kind %m[u,k](z) and of the second kind %w[u,k](z). The value returned may be in terms of special functions and can include unsimplified hypergeometric functions.

demo(hypgeo) displays several examples of Laplace transforms computed by specint.

27.1.1 Inverse Laplace transform

ilt (f(s), s, t) [function]

Computes the inverse Laplace transform of function f with respect to variable s and parameter t. t will be the independent variable of the function returned. If f(s) is a rational function, its denominator may only have linear and quadratic factors.

```
(%i1)   laplace(b(t),t,s);
                                  laplace(b(t), t, s)
(%o1)                             -------------------
                                          Lb(s)
(%i2)   ilt(%,s,t);
(%o2)                                     b(t)
```

27.1.2 Solving differential or convolution integral equations

The method of Laplace transformation is a powerful tool in science and engineering. By using *laplace* and the inverse transformation with function *ilt* together with *solve* or *linsolve*, Maxima can solve a single or a system of linear, constant coefficient differential or of convolution integral equation(s). We demonstrate the procedure with the second order linear ODE we already solved with *ode2/ic2* and *desolve/atvalue*.

```
(%i1)  assume(g>0,l>0)$
(%i2)  eq: 'diff(φ(t),t,2)+g/l*φ(t)$
```

$$(\%o2)\qquad \frac{d^2}{dt^2}\varphi(t)+\frac{g\,\varphi(t)}{l}$$

```
(%i3)  laplace(eq,t,s);
```

$$(\%o3)\qquad -\left.\frac{d}{dt}\varphi(t)\right|_{t=0}+s^2\,\mathrm{laplace}(\varphi(t),t,s)+\frac{g\,\mathrm{laplace}(\varphi(t),t,s)}{l}-\phi(0)s$$

```
(%i4)  first(linsolve([%],[laplace(φ(t),t,s)]));
```

$$(\%o4)\qquad \mathrm{laplace}(\varphi(t),t,s)=\frac{l\left(\left.\frac{d}{dt}\varphi(t)\right|_{t=0}\right)+\varphi(0)ls}{ls^2+g}$$

```
(%i5)  φ(t)=ilt(rhs(Leq1),s,t)$
(%i6)  expand(%),rootscontract;
```

$$(\%o6)\qquad \varphi(t)=\sqrt{\frac{l}{g}}\sin\left(\sqrt{\frac{g}{l}}t\right)\left(\left.\frac{d}{dt}\varphi(t)\right|_{t=0}\right)+\varphi(0)\cos\left(\sqrt{\frac{g}{l}}t\right)$$

Here we have achieved exactly the result from *desolve*,[1] the general solution from (%o3) of sect. 24.2.2. For an IVP with initial values at zero or at a point other than zero we proceed exactly as we did there. Next we show, how a single convolution integral equation can be solved with *laplace*.

```
(%i1)  'integrate(sinh(a*x)*f(t-x),x,0,t)+b*f(t) = t**2;
```

$$(\%o1)\qquad \int_0^t \mathrm{f}(t-x)\underbrace{\sinh(ax)}_{g(x)}dx+b\,\mathrm{f}(t)=t^2$$

```
(%i2)  laplace(%,t,s);
```

$$(\%o2)\qquad \frac{a\,\mathrm{laplace}(\mathrm{f}(t),t,s)}{s^2-a^2}+b\,\mathrm{laplace}(\mathrm{f}(t),t,s)=\frac{2}{s^3}$$

```
(%i3)  linsolve([%],['laplace(f(t),t,s)]);
```

$$(\%o3)\qquad [\underbrace{\mathrm{laplace}(\mathrm{f}(t),t,s)}_{\mathcal{L}f(s)}=\frac{2s^2-2a^2}{b\,s^5+\left(a-a^2b\right)s^3}]$$

```
(%i4)  f(t)=ilt(rhs(first(%)),s,t);
       Is a*b*(a*b-1) positive, negative or zero? pos;
```

$$(\%o4)\qquad f(t)=-\frac{2\cosh\left(\frac{\sqrt{ab\,(ab-1)}t}{b}\right)}{a^3b^2-2a^2b+a}+\frac{a\,t^2}{ab-1}+\frac{2}{a^3b^2-2a^2b+a}$$

27.2 Fourier transformation

[1] Note that *desolve* in fact uses *laplace*, so it does exactly what we have just done.

Part VI

Advanced Mathematical Computation

Chapter 28

Tensors

28.1 Kronecker delta

kron_delta ($(i, j \mid i_1, j_1, \ldots, i_p, j_p)$) [function]

Computes according to Def. M-55.6 the Kronecker delta of arguments i and j, which can be arbitrary expressions and are evaluated in the process of being compared for identity by *is(equal(i,j))*. In the second option, p pairs i_a, j_a will be compared, and only in case that all of them match, 1 will be returned.

28.1.1 Generalized Kronecker delta

kdelta ($[j_1, \ldots, j_p], [i_1, \ldots, i_p]$) [function of *itensor*]

Computes according to Def. M-55.7 and Satz M-55.8 the generalized Kronecker delta

$$\delta^{i_1 \ldots i_p}_{j_1 \ldots j_p}.$$

The first list in the contains the covariant and the second one the contravariant indices. The number of elements in both lists has to be identical.

28.1.2 Levi-Civita symbol

28.2 Elementary second order tensor decomposition

ElemTensorDecomp (M) [function of *rs_tensor*]

Decomposes according to Satz M-56.8 a second order tensor, given as an *nxn*-matrix M, into a sum of r elementary tensors, i.e. of tensor (or outer) products of *nxn*-vectors a^i, b_i, where r is the rank of the original tensor, i.e. the matrix M

$$M = \sum_{i=1}^{r} a^i \otimes b_i.$$

The contravariant vectors a^i are derived from the columns of M, while the covariant vectors b_i are computed accordingly. The function returns a matrix A containing the a^i als columns and a matrix B with the respective b_i as rows. The function then checks, whether the sum of the products equals M.

Chapter 29

Numerical Computation

Chapter 30

Strings and string processing

30.1 Data type string

"*string*" [matchfix operator]

A *string* is a character sequence which is not evaluated as an expression. There is a specific data type *string* in Maxima. A string is entered by enclosing it in double quote marks ". Maxima will display a string without double quote marks, unless the option variable *stringdisp* has been set to *true*. There is no data type for a character in Maxima; a single character is represented as a one-character string.

Strings may contain any characters, including embedded tab, newline, and carriage return characters. The sequence \" is recognized as a literal double quote, and \\ as a literal backslash. When backslash appears at the end of a line, the backslash and the line termination (either newline or carriage return and newline) are ignored, so that the string continues with the next line. No other special combinations of backslash with another character are recognized; when backslash appears before any character other than ", or a line termination, the backslash is ignored. There is no way to represent a special character (such as tab, newline, or carriage return) except by embedding the literal character in the string.

30.2 Transformation between expression and string

An expression and a string may look alike,[1] but they have to be distinguished. While an expression can be evaluated by Maxima and used for computation, a string cannot. When the user types in an input expression, basically it is nothing but a string at first. On parsing it, Maxima will transform it into an expression that can be evaluated. On the other hand, a Maxima program may build up a string by concatenation to form an expression which is to be parsed and evaluated. In this case, however, it will remain a string until we explicitly transform it into an expression. We can also transform expressions into strings, for instance in order to use them as the elements for building up a new string by concatenation, to be transformed to a new expression.

[1] In particular, when *stringdisp* is *false*, as by default, and strings are not enclosed in double quotation marks.

30.2.1 Expression → string

string (expr) [function]

Converts the expression *expr* to Maxima's linear notation just as if it had been typed in. The return value is a string.

Functions *concat* and *sconcat* also convert their arguments which are expressions into strings (or symbols).

30.2.2 String → expression

parse_string (str) [function of *stringproc*]

Parses the string *str* as a Maxima expression, but does not evaluate it. The string str may or may not have a terminator (dollar sign $ or semicolon ;). Only the first expression is parsed, if there is more than one.

eval_string (expr) [function of *stringproc*]

Parses the string *str* as a Maxima expression and then evaluates it. The string str may or may not have a terminator (dollar sign $ or semicolon ;). Only the first expression is parsed and evaluated, if there is more than one.

30.3 Display of strings

string (expr) default: *false* [option variable]

When *stringdisp* is *true*, strings are displayed enclosed in double quote marks. Otherwise, quote marks are not displayed. See *concat* for an example. *stringdisp* is always *true* when displaying a function definition.

30.4 Manipulating strings

concat $(arg_1, \ldots, arg_n)$ [function]
sconcat $(arg_1, \ldots, arg_n)$ [function]

concat concatenates its arguments, which can be expressions, symbols or strings. Arguments are evaluated and must evaluate to atoms.[2] The return value is a symbol if the first argument is a symbol, and a string otherwise. The single quote ' preceeding an argument prevents its evaluation.

```
(%i1)   a:5$
(%i1)   str:concat(1+1,a,string(b+c),"we(");
(%o2)                              25c+bwe(
(%i3)   stringdisp:true$
(%i4)   str;
(%o4)                             "25c+bwe("
```

[2] Function *string* can be used to transform an argument evaluating to a non-atomic expression into a string.

sconcat does the same as *concat* with the only differences, that arguments need not evaluate to atoms and that the return value is always a string.

```
(%i1)   i:3$
(%i2)   sconcat("x[",i,"]:",expand((x+y)^2));
(%o2)                          x[3]:y^2+2*x*y+x^2
```

A symbol concatenated by *concat* can be assigned a value and used in computation. The :: (double-colon) assignment operator can be used to evaluate not only the right hand side, but also the left hand side of the assignment.

```
(%i1)   concat(c,6)::7+1$
(%i2)   c6;
(%o2)                                  8
```

30.5 Package *stringproc*

The package *stringproc* contains a large number of sophisticated functions for string processing. It is loaded automatically by Maxima on using one of its functions.

Part VII

Maxima Programming

Chapter 31

Compound statements

31.1 Sequential and block

31.1.1 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.

31.1.2 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*.

31.2 Function

31.2.1 Function definition

31.2.1.1 Defining the function

:= [infix operator]
$f(x_1, \ldots, x_n)$ *:=* *expr*
":=" $(f(x_1, \ldots, x_n), expr)$
define $(f(x_1, \ldots, x_n), 'expr)$

define $(f(x_1, \ldots, x_n), expr)$ [function]
$f(x_1, \ldots, x_n)$ *:=* *''expr*

A *user function* has to be defined before it can be used, i.e. *called*. 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 function body unless explicitly forced by quote-quote ' ', whereas *define* always evaluates the function body unless explicitly prevented by single 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 31.2.3. The expression using *arraymake* returns an *array function* with parameters in square brackets, see section 31.2.4. The second expression using *funmake* returns a *subscripted function*, see section 31.2.5. 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)
```

31.2.1.2 Showing the function definition

fundef(f) [function]

Returns the definition of function f. *fundef* quotes its argument; the quote-quote operator ’’ defeats quotation. The argument may be the name of a macro (defined with ::=), an ordinary function (defined with := or define), an array function (defined with := or define, but enclosing arguments in square brackets []), a subscripted function (defined with := or define, but enclosing some arguments in square brackets and others in parentheses ( )), one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

dispfun($\left(f_1,\ldots,f_n \mid all\right)$*)* [function]

Displays the definition of the user-defined functions $f_1,\ldots,f_n$. Each argument may be the name of a macro (defined with ::=), an ordinary function (defined with := or an array function (defined with := or define, but enclosing arguments in square brackets []), a subscripted function (defined with := or define, but enclosing some arguments in square brackets and others in parentheses ( )), one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript. *dispfun (all)* displays all user-defined functions as given by the functions, arrays, and macros lists, omitting subscripted functions defined with constant subscripts. dispfun creates an *intermediate expression label* (%t1, %t2, etc.) for each displayed function, and assigns the function definition to the label. *dispfun* quotes its arguments; the quote-quote operator ’’ defeats quotation. dispfun returns the list of intermediate expression labels corresponding to the displayed functions. For an example and the use of these expression labels see [**MaxiManE**].

31.2.2 Function call

31.2.2.1 Quoting a function call

A function call can be quoted in two different ways: 'f(x) is the noun form of the function call and has to be evaluated with the *nouns* flag set in *ev*. '(f(x)) quotes the whole expression and can be evaluated without the noun flag set. In order to see whether a function call is a noun form or not, the flag *noundisp* can be set, see Stavros' mail from Oct. 26, 2020 in Maxima discuss .

31.2.3 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. (Note that the function parameters may not be repeated here.) Inside of a function body, *local* can - and should - be applied both to the local variables and the 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 *required arguments*, while all further arguments, if present, are *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.

31.2.4 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*.

31.2.5 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 31.2.4, 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
```

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

31.2.6 Constructing (and calling) a function

31.2.6.1 Apply: construct and call

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

apply (f,[$v_1,\ldots,v_n$]) evaluates its arguments and constructs 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 expression is simplified and evaluated, which means that f is called. The return value of *apply* is the return value of this function call of f.

31.2.6.2 Funmake: construct only

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 '% can be used, but only in a separate, 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;
                                     x - 1
(%o9)                          h(x) ::= -----
                                       2
(%i10)  funmake(h,[u]);
(%o10)                               h(u)
(%i11)  ev(%);
                                     u - 1
(%o11)                               -----
                                       2
```

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

31.3 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]
```

31.4 Macro function

A MaximaL macro function, sometimes simply called a *macro*, 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 body itself is evaluated. We say that a macro function *quotes* its arguments, because a Maxima *quote operator* inhibits evaluation of its arguments. The call of a macro function is executed in two steps. First the so-called *macro expansion* is created,a term again refering to the macro concept in Lisp. The macro expansion is a form, which is immediately afterwards evaluated in the context from which the macro was called. With function *macroexpand* only the first step is done.

In many cases the effect of a macro function call is equivalent to an ordinary function call plus one additional evaluation with either quote-quote or *ev*. Note that additional and even multiple evaluations with either quote-quote or *ev* can follow both an ordinary and a macro function call. See "*Macro function demonstration.wxm*" for an illustrative comparison of two macro functions with their corresponding ordinary functions.

31.4.1 Macro function definition

::= [infix operator]

This is the *macro function definition operator*. It is used with a syntax similar to the := operator.

macros [system variable]

The value of this system variable is a list of all user-defined macro functions. The macro function definition operator ::= puts a new macro function on this list. *kill*, *remove*, and *remfunction* remove macro functions from the list.

31.4.2 Macro function expansion

macroexpand (f(x)) [infix operator]

Expands the macro function call f(x) without evaluating it. If the argument of *macroexpand* is not a macro function call, it is returned. If the expansion of expr yields another macro function call, that macro function call is also expanded. *macroexpand* quotes its argument just as a function call of the macro function does. However, if the expansion of a macro function call has side effects (e.g. printing out something), these side effects are executed.

31.4.3 Macro function call

See introductory section.

Chapter 32

Program Flow

Part VIII

User interfaces, Package libraries

Chapter 33

User interfaces

33.1 Internal interfaces

33.1.1 Command line Maxima

33.1.2 wxMaxima

33.1.3 iMaxima

33.1.4 XMaxima

33.1.5 TeXmacs

33.1.6 GNUplot

33.2 External interfaces

33.2.1 Sage

33.2.2 Python, Jupyter, Java, etc.

Chapter 34

Package libraries

Part IX

Maxima development

Chapter 35

MaximaL development

35.1 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.

35.2 Development with wxMaxima

35.2.1 File management

35.3 Error handling and debugging facilities in MaximaL

35.3.1 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

Chapter 36

Lisp Development

36.1 MaximaL and Lisp interaction

36.1.1 History of Maxima and Lisp

Maxima is written in Lisp, and much of the terminology used within MaximaL is based on the terminology used within Lisp. Since Maxima was, in the early phase of the 1960s and 1970s, as part of MIT's project MAC, developed in parallel to Lisp, Maxima's basic design decisions were based on the state of the art of the contemporary Lisp's available. The early parts of Maxima were written in MACLisp, which was also developed as part of MIT's project MAC. When Common Lisp had been established as a standard, most of Maxima's source code was translated to it. However, some parts remained in MACLisp, having been wrapped into Common Lisp user functions, so that they could be understood by the Common Lisp interpreter or compiler. While these relics have stayed in Maxima until today, Common Lisp itself has been refined and enhanced over the years up to the present ANSI standard. While new Lisp development within Maxima can make use of the entire functionality of this advanced standard, which most of today's Comon Lisp systems understand, the major part of Maxima makes use only of the basic language elements from the early days of Common Lisp.

36.1.2 Accessing Maxima and Lisp functions and variables

Maxima is written in Lisp, and it is easy to access Lisp functions and variables from MaximaL and vice versa.
The correspondence between MaximaL and Lisp identifiers is described in section 3.4.3.
Then we show how the user can use Lisp forms within his Maxima session or inside of his MaximaL code in order to visualize and interact with data and program structures on Maxima's Lisp level.
FInally it is decribed how MaximaL expressions can be used from within Lisp code.

36.1.2.1 Executing Lisp code under MaximaL

36.1.2.1.1 Switch to an interactive Lisp session temporarily

The MaximaL function *to_lisp* opens an interactive Lisp session. Entering the Lisp form *(to-maxima)* closes the Lisp session and returns to MaximaL.

36.1.2.1.2 Single-line Lisp mode

Lisp code may be executed from within a MaximaL session. A single line of Lisp (containing one or more forms) may be executed with the MaximaL break command *:lisp*. E.g.

```
(%i1)   :lisp (foo $x $y)
```

calls the Lisp function *foo* with MaximaL variables x and y as arguments. The *:lisp* construct can appear at the interactive Maxima or debugger prompt or in a file processed by *batch* or *demo*, but not in a file processed by *load*, *batchload*, *translate_file*, or *compile_file*.

Further examples: 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))
```

36.1.2.1.3 Using Lisp forms directly in MaximaL

There is yet another way to execute Lisp code from within a MaximaL session. Lisp forms can - with some syntactical adaptation - be included directly into MaximaL code. This mechanism works with the help of the ? escape to access Lisp identifiers from MaximaL described in section 3.4.3.

In order to synchronize the different syntax of MaximaL and Lisp, Lisp forms here are notated in a way which resembles MaximaL: instead of the Lisp function being the first element of a list and the arguments the remaining elements, the function name is set in front of the list which then includes only the arguments, separated by commas. E.g. the Lisp form

```
(foo a b c)
```

with some Lisp function foo is written when called from MaximaL as

```
?foo (a, b, c);
```

For example, if the internal structure of some MaximaL variable a is to be displayed, we can make use of the Lisp *print* function by

```
?print ($a);
```

Note that this mechanism does not work for all Lisp functions.

In particular, some Lisp functions are shadowed in Maxima, namely the following: *complement, continue, "//", float, functionp, array, exp, listen, signum, atan, asin, acos, asinh, acosh, atanh, tanh, cosh, sinh, tan, break, gcd*.

36.1.2.2 Using MaximaL expressions within Lisp code

36.1.2.2.1 Reading MaximaL expressions into Lisp

The *#$* Lisp macro allows the use of Maxima expressions in Lisp code. *#$expr$* expands to a Lisp expression equivalent to the Maxima expression *expr*. E.g.

```
(msetq $foo #$[x, y]$)
```

in Lisp has the same effect as has in MaximaL

```
(%i1)   foo: [x, y];
```

36.1.2.2.2 Printing MaximaL expressions from Lisp

The Lisp function *displa* prints an expression in Maxima format.

```
(%i1)   :lisp #$[x, y, z]$
((MLIST SIMP) $X $Y $Z)
(%i1)   :lisp (displa '((MLIST SIMP) $X $Y $Z))
[x, y, z]
NIL
```

36.1.2.2.3 Calling MaximaL functions from within Lisp

Functions defined in Maxima are not ordinary Lisp functions. The Lisp function *mfuncall* calls a Maxima function. For example:

```
(%i1)   foo(x,y) := x*y$
(%i1)   :lisp (mfuncall '$foo 'a 'b)
((MTIMES SIMP) A B)
```

36.2 Using the Emacs IDE

36.3 Debugging

36.3.1 Breaks

36.3.2 Tracing

36.3.3 Analyzing data structures

36.4 Lisp compilation

36.5 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.

36.5.1 Loading Lisp code

36.5.1.1 Loading whole Lisp packages

36.5.1.2 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.

36.5.2 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 38.

Part X

Developer’s environment

Chapter 37

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.

37.1 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*.

37.2 Maxima

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

37.2.1 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.

37.2.2 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 39. Alternatively, Maxima can be built for Windows under Cygwin, see section **??**.

37.3 External program editor

37.3.1 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.

37.4 7zip

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

37.5 SBCL: Steel Bank Common Lisp

A considerable number of Lisp compilers is available, and 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 with Maxima and has become a kind of de facto standard for Common Lisp use. See the *SBCL User Manual* for reference. [**SbclMan**]

In principle we can use the SBCL installation coming with the Maxima installer as *inferior Lisp* under Emacs, too. However, we can also install SBCL separately in addition, for instance if we want to use a different (newer) version or if we want to be independent of what happens to come with the consecutive installers. We prefer the latter option.

37.5.1 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 active Windows user, if they don't exist yet. However, a Windows restart is necessary to activate them. Check that they are properly set with left click on *Dieser PC*, *properties*, *Erweiterte Systemeinstellungen*, *environment variables*, looking at the lower field for our active Windows user. We should see appended at the end of the *path* variable the path

```
C:\Program Files\Steel Bank Common Lisp\1.4.2\.
```

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

```
C:\Program Files\Steel Bank Common Lisp\1.4.2\.
```

If, later under Emacs, we want to use the separately installed SBCL and the one from the Maxima installer alternately, we do not need to change the Windows environment variables any more. Instead, the local copies of them, which Emacs actually uses, can be adjusted easily in the *.emacs* init file, see section 37.6.3.2.

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 will be adjusted automatically. We only need to adapt our personal setup and initialization files (e.g. *.emacs*, see below).

37.5.2 Setup

37.5.2.1 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")
```

37.5.2.2 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 the standard file
    extension .lisp can be 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.

37.5.2.3 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 the start directory and type SBCL.

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

37.6 Emacs

37.6.1 Overview

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

37.6.1.1 Editor

It's not without reason that one generally defines

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

Although the Emacs editor and in particular its embedding in the overall 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**]

37.6.1.2 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. [**eLispMan**]

37.6.1.3 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.

37.6.1.4 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. [**iMaximaHP**]

37.6.1.5 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. [**SlimeMan**]

37.6.2 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. First unzip it to C:/. Then move the folder to C:/Program Files/Emacs (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.

37.6.3 Setup

37.6.3.1 Set start directory

We can set the Emacs start directory in its 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 (in case we invoke it from within Emacs), 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"D:/Maxima/Repos/")
```

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

37.6.3.2 Init file ".emacs"

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

Note: 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 e.g. 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/1.4.2/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. *Setup Maxima*. We need to load the system eLisp file *setup-imaxima-imath.el* [**iMaximaHP**]
which comes with Maxima. Best is to create a local copy in a fixed place on our computer, so we do not always have to adapt the path to the file if we use different Maxima installations. This file sets up Emacs to support Maxima and the Latex-based interface iMaxima. We do not need to customize this file. But before loading the file we set two system variables. *maxima-build-type* specifies whether we use

Maxima from an installer or whether we have built Maxima from a tarball or a local copy of the repository. *maxima-build-dir* specifies the path to the root directory of the Maxima we want to use. If we do not specify these two system variables, the first Maxima installer found in "C:/" will be used. (Note that this is the oldest one installed.) So in the init-file we write

```
; *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. [**SlimeMan**]

```
(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 [**EmacsMan**]
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.
)
```

37.6.3.3 Customization

In Emacs *Options/Set Default Font* set Courier New size to 12. Store this with *Save Options*, so I don't have to set it again on every start of Emacs. This will be written automatically into the .emacs file.

37.6.3.4 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 e.g. 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)
```

37.6.3.5 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, see above.

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 39 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.

37.7 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.

37.7.1 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. See that you are connected to the internet. 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 37.5.2.2, 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.

37.8 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 37.6.3.2, 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).

37.9 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.

37.9.1 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*
"3.3.1"
```

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 2.26. 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 3.3.1 available. Finally, we take a look at the Asdf homepage and find out that the latest released version is 3.3.2. 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 latest *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. Check that there were no error conditions. We got three, so we fall back to asdf 3.3.1. With this version, compilation was successful. Now the asdf.fasl file should be in the download folder, too. We copy it into the folder *Program Files/Steel Bank Common Lisp/1.4.2/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.

Note: We experienced that loading startswank.lisp from a (i)Maxima session under Emacs does not work with our sbcl 1.4.2 using its original asdf 3.3.1 nor with our self-compiled asdf 3.3.1. With our sbcl 1.3.18 it works with its original asdf 3.1.5, but not with asdf 3.3.1.

37.10 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.

37.10.1 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.

37.10.2 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.

37.10.3 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.

37.11 Linux and Linux-like environments

37.11.1 Cygwin

Install Cygwin in C:/Program Files/cygwin64.

37.11.2 MinGW

Install MinGW in C:/Program Files/MinGW.

37.11.3 Linux in VirtualBox under Windows

37.11.3.1 VirtualBox

37.11.3.2 Linux

Chapter 38

Repository management: Git and GitHub

38.1 Introduction

This chapter follows up on the discussion of section 36.5.

38.1.1 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 on our computer 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/onto 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 once, too, but then we will not update it directly from Sourceforge, but instead from our local repository. So it will mirror both the branches from Sourceforge and our own ones. 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 make in our local repository can be incorporated in our own Maxima builds.

38.1.2 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 write 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 [**ChProGit**]

PDF in the net for free. All the details you ever want to know can be found in the *Git Online Reference*. It should also be mentioned that almost any special question around Git has already been asked on Stackoverflow. [**GitRef**]

38.1.2.1 KDiff3

We will use *KDiff3* to help us resolve merge conflicts arising under Git when we rebase our own changes onto the original branches from the Sourceforge repository.

38.1.3 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.

38.2 Installation and Setup

38.2.1 Git

38.2.1.1 Installing Git

Download the latest Windows installer from *git-scm.com*. Install it as administrator in the default directory *C:/Program Files/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 shortcutof CMD or Git bash. Select properties. Change *Execute in* to the above path. In *Destination* delete the option *-cd-to-home*. It might be necessary to restart the computer for the changes to take effect. [1]

38.2.1.2 Installing KDiff3

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

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

38.2.1.3 Configuring Git

Git allows configuration at various levels: system, user, project. Configuration files are therefore created in various locations. In C:/Users/<username>/ we place the file *.gitconfig* given in Annex D, after having done our personal adjustments to it.

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 blank and 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 stackoverflow. We want to apply the proposed solution and create a Windows environment variable *LC_ALL* which we assign the value *C.UTF-8*. Don't define it under "'Admin"', but under "'System variables"'. This definition will solve the problem permanently for both Git CMD and Git bash.

38.2.1.4 Using Git

Under Git bash, directory paths are written like e.g. */d/maxima/repos*. Changing the current directory is done with e.g. *cd /c/users/<username>*.

38.2.2 GitHub

38.2.2.1 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.

38.3 Cloning the Maxima repository

38.3.1 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.

38.3.2 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.

38.4 Updating our repository

38.4.1 Setting up the synchronization

Soon there will be new commits submitted at the Sourceforge repository by members of the Maxima team, 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.

38.4.2 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 branches exactly we've just cloned from there.

The most interesting 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. Our own work we will do on other branches which we create from our local branch *master*.

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

```
git ch master
git pull
```

Note that we use the shortnames defined in *.gitconfig*, see. Annex D. With the option *pull ––all*, not only master, but all *tracked* branches will be pulled (i.e. updated) from origin into their respective local branches. When using these commands for the first time or after a long time of not having used them, they can take a while, because Git does a lot of checking in the background, so be patient.

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
```

38.4.3 Pushing to the public repository at GitHub

First we create the shortname *github* on our local machine 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, as it is mirrored on our local machine, 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
```

In this proces we might be asked to enter our GitHub username and password. 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 (e.g. release) 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.

38.5 Working with the Repository

38.5.1 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).

38.5.2 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
```

38.5.3 Committing, merging and rebasing our changes

Chapter 39

Building Maxima under Windows

39.1 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.

39.2 Lisp-only build

39.2.1 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.

39.2.2 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.

39.3 Building Maxima with Cygwin

Part XI

Maxima’s file structure, build system

Chapter 40

Maxima’s file structure: repository, tarball, installer

Chapter 41

Maxima’s build system

Part XII

Lisp program structure (model), control and data flow

Chapter 42

Lisp program structure

42.1 Supported Lisps

Part XIII

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 31.2.3.

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 ...

Canonical rational expression (CRE)

A *canonical rational expression* is a special internal representation of a Maxima expression. See section 9.2.3.

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 hand side, the function name and its parameters are specified, while on the right hand 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 31.2.3.

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 31.2.4.

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 31.2.5.

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 31.3.

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

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* ::=.

An *undeclared function* is just a symbol which stands for a function, possibly followed by one or more arguments in parentheses. It has not been declared with a function definition. It is not bound. On calling it, it evaluates to itself. However, for the purpose of differentiation, dependencies of the function on certain variables can be declared with *depends*.

Lambda expression

The return value of a *lambda function* is called a *lambda expression*. See section 31.3.

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

For the definition see section 14.1.1.

Predicate

A predicate is an expression returning a Boolean value. This may be a function or a lambda expression with a Boolean return value, a relational expression evaluated by *is*, or the Boolean constants *true* and *false*. For a *match predicate* see matchdeclare.

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 31.

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*. For a *match variable* see matchdeclare.

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 37.5.2.2 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 37.6.3.2 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/<username>. See section 38.2.1.3 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
```

Appendix E

blanco

```
(%i1)
(%i2)
(%i3)

(%o1)
(%o2)
(%o3)
(%o1)

(%i2)
(%o2)

(%i3)
(%o3)

(%i4)
(%o4)
```

Index