The Maxima Book

Paulo Ney de Souza
Richard J. Fateman
Joel Moses
Cliff Yapp

17th December 2003

Credits

Paulo Ney de Souza
Jay Belanger
Richard Fateman
Joel Moses
Cliff Yapp

Contents

Many hands and minds have contributed to Macsyma, in developing it as a research program, and tuning it for use by others. We have enjoyed constructing Macsyma and we hope that you will enjoy using it. We hope you will consider contributing your carefully polished and documented application programs to libraries at your local installation and other sites.

Examples

Part I

The Maxima Program and Standard Packages

CHAPTER 1

Introduction

1.1 What is Maxima?

Maxima (pronounced mæximə[1]) is a large computer program designed for the manipulation of algebraic expressions. You can use Maxima for manipulation of algebraic expressions involving constants, variables, and functions. It can differentiate, integrate, take limits, solve equations, factor polynomials, expand functions in power series, solve differential equations in closed form, and perform many other operations. It also has a programming language that you can use to extend Maxima's capabilities.

The Dangers of Computer Algebra

With all this marvelous capability, however, you must bear in mind the limitations inherent in any such tool. Those considering the use of computers to do mathematics, particularly students, must be warned that these systems are no substitute for hands on work with equations and struggling with concepts. These systems do not build your mathematical intuition, nor will they strengthen your core skills. This will matter a great deal down the road, especially to those of you who wish to break new ground in theoretical mathematics and science. Do not use a computer as a substitute for your basic education.

By the same token, however, proficiency with computers and computer based mathematics is crucial for attacking the many problems which literally cannot be solved by pencil and paper methods. In many cases problems which would take years by hand can be reduced to seconds by powerful computers. Also, in the course of a long derivation, it is sometimes useful for those who have already mastered the fundamentals to do work in these systems as a guard against careless errors, or a faster means than a table of deriving some particular result. Also, in case of an error, fixing the resulting error can often be much quicker and simpler courtesy of a mathematical notebook, which can be reevaluated with the correct parameters in place.

But just as a computer can guard against human error, the human must not trust the computer unquestioningly. All of these systems have limits, and when those limits are reached it is quite possible for bizarre errors to result, or in some cases answers which are actually wrong, to say nothing of the fact that the people who programmed these systems were human, and make mistakes. To illustrate the limits of computer algebra systems, we take the following example: when given the integral `Integrate 1/sqrt(2-2*cos(x)) from x=-pi/2 to pi/2`, Mathematica 4.1 gives, with no warnings, `\!\(2\ Log[4] - 2\ Log[Cos[\[Pi]\/8]] + 2\ Log[Sin[\[Pi]\/8]]\)` which `N[%]` evalutates numerically to give 1.00984. Maxima 5.6 returns the integral unevaluated, the commercial Macsyma says the integral is divergent, and Maple 7 says infinity. (Cite Maxima Email list here.) Had the person who wished to learn the result

[1]The acronym Maxima is the corruption of the main project name MACSYMA, which stands for Project MAC's SYmbolic MAnipulation System. *MAC* itself is an acronym, usually cited as meaning Man and Computer or Machine Aided Cognition. The Laboratory for Computer Science at the Massachusetts Institute of Technology was known as Project MAC during the initial development of MACSYMA. The name MACSYMA is now trademarked by Macsyma Inc.

blindly trusted most of the systems in question, he might have been misled. So remember to think about the results you are given. The computer is not always necessarily right, and even if it gives a correct answer that answer is not necessarily complete.

1.2 A Brief History of Macsyma

The birthplace of Macsyma, where much of the original coding took place, was Project MAC at MIT in the late 1960s and earlier 1970s. Project MAC was an MIT research unit, which was folded into the current Laboratory for Computer Science. Research support for Macsyma included the Advanced Research Projects Agency(ARPA), Department of Defense, the US Department of Energy, and other government and private sources.

The original idea, first voiced by Marvin Minsky, was to automate the kinds of manipulations done by mathematicians, as a step toward understanding the power of computers to exhibit a kind of intelligent behavior. [?] The undertaking grew out of a previous effort at MITRE Corp called Mathlab, work of Carl Engelman and others, plus the MIT thesis work of Joel Moses on symbolic integration, and the MIT thesis work of William A. Martin. The new effort was dubbed Macsyma - Project MAC's SYmbolic MAnipulator. The original core design was done in July 1968, and coding began in July 1969. This was long before the days of personal computers and cheap memory - initial development was centered around a single computer shared with the Artificial Intelligence laboratory, a DEC PDP-6. This was replaced by newer more powerful machines over the years, and eventually the Mathlab group acquired its own DEC-PDP-10, MIT-ML running the ITS operating system. This machine became a host on the early ARPANET, predecessor to the internet, which helped it gain a wider audience. As the effort grew in scope and ability the general interest it created led to attempts to "port" the code - that is, to take the series of instructions which had been written for one machine and operating system and adapt them to run on another, different system. The earliest such effort was the running of Macsyma in a MacLisp environment on a GE/Honeywell Multics mainframe, another system at MIT. The Multics environment provided essentially unlimited address space, but for various reasons the system was not favored by programmers and the Multics implementation was never popular. The next effort came about when a group at MIT designed and implemented a machine which was based on the notion that hardware support of Lisp would make it possible to overcome problems that inhibited the solution of many interesting problems. The Lisp machine clearly had to support Macsyma, the largest Lisp program of the day, and the effort paid off with probably the best environment for Macsyma to date (although requiring something of an expert perspective). Lisp machines, as well as other special purpose hardware, tended to become slow and expensive compared to off-the-shelf machines built around merchant-semiconductor CPUs, and so the two companies that were spun off from MIT (Symbolics Inc, and LMI) both eventually disappeared. Texas Instruments built a machine called the Explorer bases on the LMI design, but also stopped production.

Around 1980, the idea of porting Macsyma began to be more interesting, and the Unix based vaxima distribution, which ran on a Lisp system built at the University of California at Berkeley for VAX UNIX demonstrated that it was both possible and practical to run the software on less expensive systems. (This system, Franz Lisp, was implemented primarily in Lisp with some parts written in C.) Once the code stabilized, the new version opened up porting possibilities, ultimately producing at least six variations on the theme which included Macsyma, Maxima, Paramax/Paramacs, Punimax, Aljbar, and Vaxima. These have followed somewhat different paths, and most were destined to fade into the sunset. The two which survived obscurity, Maxima and Macsyma, we will discuss below. Punimax was actually an offshoot of Maxima - some time around 1994 Bruno Haible (author of clisp) ported maxima to clisp. Due to the legal concerns of Richard Petti, then the owner of the commercial Macsyma, the name was changed to Punimax. It has not seen much activity since the initial port, and although it is still available the ability of the main Maxima distribution to compile on Clisp makes further development of Punimax unlikely.

There is a certain surprising aspect in this multiplicity of versions and platforms, given how the code seemed tied to the development environment, which included a unique operating system. Fortunately, Berkeley's building a replica of the Maclisp environment on the MIT-ML PDP-10, using tools available in almost any UNIX/C environment, helped solve this problem. Complicating the matter was the eventual demise of the PDP-10 and Maclisp systems as Common Lisp (resembling lisp-machine lisp), influenced by BBN lisp and researchers at Stanford, Carnegie Mellon University, and Xerox, began to take hold. It seemed sensible to re-target the code to make it compatible with what eventually became the ANSI Common Lisp standard. Since almost everything needed for for Macsyma can be done in ANSI CL, the trend toward standardization made many things simpler. There are a few places where the language is not standardized, in particular connecting to modules written in other languages, but much of the power of the system can be expressed within ANSI CL. It is a trend the Maxima project is planning to carry on, to maintain and expand on this flexibility which has emerged.

With all these versions, in recent history there are two which have been major players, due this time more to economics than to code quality. 1982 was a watershed year in many respects for Macsyma - it marks clearly the branching of Macsyma into two distinct products, and ultimately gave rise to the events which have made Maxima both possible and desirable. MIT had decided, with the gradual spread of computers throughout the academic world, to put Macsyma on the market commercially, using as a marketing partner the firm of Arthur D. Little, Inc. This version was sold to the Symbolics Inc., which, depending on your perspective, either turned the project into a significant marketing effort to help sell their high-priced lisp machines, or was a diversionary tactic to deny their competitors (LMI) this program. At the same time MIT forced UC Berkeley (Richard Fateman) to withdraw the copies from about 50 sites of the VAX/UNIX and VAX/VMS versions of Macsyma that he had distributed with MIT's consent, until some agreement could be reached for technology transfer. Symbolics hired some of the MIT staff to work at Symbolics in order to improve the code,which was now proprietary. The MIT-ML PDP-10 also went off the Arpanet in 1983. (Interestingly, the closing of the MIT Lisp and Macsyma efforts was a key reason Richard Stallman decided to form the Free Software Foundation.) Between the high prices, closed source code, and neglecting all platforms in favor of Lisp Machines pressure came to bear on MIT to release another version to accommodate these needs, which they did with some reluctance. The new version was distributed via the National Energy Software Center, and called DOE Macsyma. It had been re-coded in a dialect of lisp written for the VAX at MIT called NIL. There was never a complete implementation. At about the same time a VAX/UNIX version "VAXIMA" was put into the same library by Berkeley. This ran on any of hundreds of machines running the Berkeley version of VAX Unix, and through a UNIX simulator on VMS, on any VAX system. The DOE versions formed the basis of the subsequent non-Symbolics distributions. The code was made available through the National Energy Software Center, which in its attempt to recoup its costs, charged a significant fee `($1-2k?)`. It provided full source, but in a concession to MIT, did not allow redistribution. This prohibition seems to have been disregarded, and especially so since NESC disappeared. Perhaps it didn't recoup its costs! Among all the new activity centered around DOE Macsyma, Prof. William Schelter began maintaining a version of the code at UT Austin, calling his variation Maxima. He refreshed the NESC version with a common-lisp compatible code version.

There were, from the earliest days, other computer algebra systems including Reduce, CAMAL, Mathlab-68, PM, and ALTRAN. More serious competition, however, did not arrive until Maple and Mathematica were released, Maple in 1985 (Cite list of dates) and Mathematica in 1988 (cite wolfram website). These systems were inspired by Macsyma in terms of their capabilities, but they proved to be much better at the challenge of building mind-share. DOE-Macsyma, because of the nature of its users and maintainers, never responded to this challenge. Symbolics' successor Macsyma Inc, having lost market share and unable to meet its expenses, was sold in the summer of 1999 after attempts to find endowment and academic buyers failed. (Cite Richard Petti usenet post.) The purchaser withdrew Macsyma from the market and the developers and maintainers of that system dispersed. Mathematica and Maple appeared to have vanquished Macsyma.

It was at this point Maxima re-entered the game. Although it was not widely known in the general academic public, W. Schelter had been maintaining and extending his copy of the code ever since 1982. He had decided to see what he could do about distributing it more widely. He attempted to contact the NESC to request permission to distribute derivative works. The duties of the NESC had been assumed in 1991 by the Energy Science and Technology Software Center, which granted him virtually unlimited license to make and distribute derivative works, with some minor export related caveats.

It was a significant breakthrough. While Schelter's code had been available for downloading for years, this activity became legal with the release from DOE granted in Oct. 1998, and Maxima began to attract more attention. When the Macsyma company abruptly vanished in 1999, with no warning or explanation, it left their customer base hanging. They began looking for a solution, and some drifted toward Maxima.

Dr. Schelter maintained the Maxima system until his untimely death in July, 2001. It was a hard and unexpected blow, but Schelter's obtaining the go-ahead to release the source code saved the project and possibly even the Macsyma system itself. A group of users and developers who had been brought together by the email list for Maxima decided to try and form a working open source project around the Maxima system, rather than let it fade - which is where we are today.

CHAPTER 2

Installing

2.1 Installing on Linux

Note: This is written for version 5.6. With 5.9 things should change significantly, both for Windows and Linux, so a rewrite will be performed at that time.

2.1.1. The Easy Way

If you are using Debian Linux, there are packages for Maxima in the Debian archives. Redhat users can find packages at rpmfind.net and mirrors. To the best of my knowledge these packages are all compiled against GNU Common Lisp (GCL). GCL does not access the readline library, which makes using Maxima on the terminal somewhat cumbersome. There are various interfaces which address this issue, but in the case of these packages the left arrow key will not take you back to the middle of an expression and let you edit just what you want - you will have to delete everything back to the point where you made your error. See the interfaces section for more about this, or if you really want to use maxima in the terminal I'd recommend compiling against CLISP.

2.1.2. The Hard Way

Compiling Maxima on Linux should go reasonably smoothly, although some distributions seem to have problems with the makefiles. Redhat 7.1 is known to be a bit flaky in this regard.

Compiling with GCL

GCL is the version of Lisp Maxima has been maintained on for many years by William Schelter, and thus the default system on which to build Maxima. Maxima will typically be made to work with GCL, and then be fixed to work with other Lisps. If in doubt, start here, but again be warned that readline support is not present in GCL.

The first step in this process is to compile GCL. The files from the compiling of GCL are needed when building Maxima against it, so first download and decompress the most current version of GCL. Compiling it should be simple:

```
cd $HOME/gcl
./configure
make
```

Once that is complete, download and decompress Maxima. Now the process gets a little more complicated - you will have to hand edit the file `configure`. What follows is the first part of the file, with the parts you must edit in bold:

```
#!/bin/sh
```

```
# edit next few lines
# GCLDIR should be where gcl was built, and the o/*.o lsp/*.o etc must be
# there to link with maxima
GCLDIR=/home/cliff/gcl-2.3.8
# the directory where this file is, and where you will build maxima
# could use
MAXDIR=/home/cliff/maxima-5.5
MAXDIR=‘pwd‘
# where to put some maxima .el files
EMACS_SITE_LISP=/usr/lib/emacs/site-lisp
# determines where to install
# PREFIX_DIR=/usr/local puts things in /usr/local/lib/maxima-x.y
# and /usr/local/bin
PREFIX_DIR=/usr/local
INFO_DIR=${PREFIX_DIR}/info
MAN_DIR=${PREFIX_DIR}/man/man1
##### end editing #########
```

Once this is correctly done, run the configure script: `./configure.` If this succeeds without any problems, run `make`. This could be a long process, depending on your machine. Once it is done, and if no errors result, run `make test`. If any errors appear in either stage, consult the trouble shooting section. Otherwise, you are ready to su into root and run `make install`. Once this process is completed, you should be able to run `maxima` and `xmaxima`. Emacs mode may take a little more work, and the other interfaces are separate programs - see the Interfaces chapter for details.

Compiling with CLISP

Clisp has, among other features, the ability to use the readline library. This means that many of the limitations created in the terminal by GCL are not relevant here, and for the beginner especially this is a good place to start. Unfortunately, compiling on a flavor of Lisp other than GCL is not quite as smooth - here are the steps.

Change into the directory $HOME/maxima/src (or $HOME/maxima-5.6/src - whatever your setup dictates.)

For newer versions of clisp, you need to change a couple lines in the file compile-clisp.lisp:

```
< (lisp:gc) → (ext:gc)
< (lisp:saveinitmem "maxima-clisp.mem" → (ext:saveinitmem "maxima-clisp.mem"
```

Then run the following commands:

```
make clisp-compile CLISP=clisp
make maxima-clisp.mem CLISP=clisp
make test-clisp CLISP=clisp
```

Once that is complete, you can either leave maxima where you built it, or move the directory to a more suitable place. Once that is done, in order to set up a convenient way to run Maxima, you can create some shell scripts in /usr/local/bin (make sure to do chmod +x on both files):

```
===================maxima==============
#!/bin/bash
export MAXIMA_DIRECTORY=/PATH_TO_NEW_DIRECTORY/maxima
exec clisp -M ${MAXIMA_DIRECTORY}/src/maxima-clisp.mem
========================================

==============xmaxima==================
#!/bin/bash
```

export MAXIMA_DIRECTORY=/PATH_TO_NEW_DIRECTORY/maxima
exec ${MAXIMA_DIRECTORY}/bin/xmaxima -lisp clisp
==

Compiling with CMULISP

Other Lisps

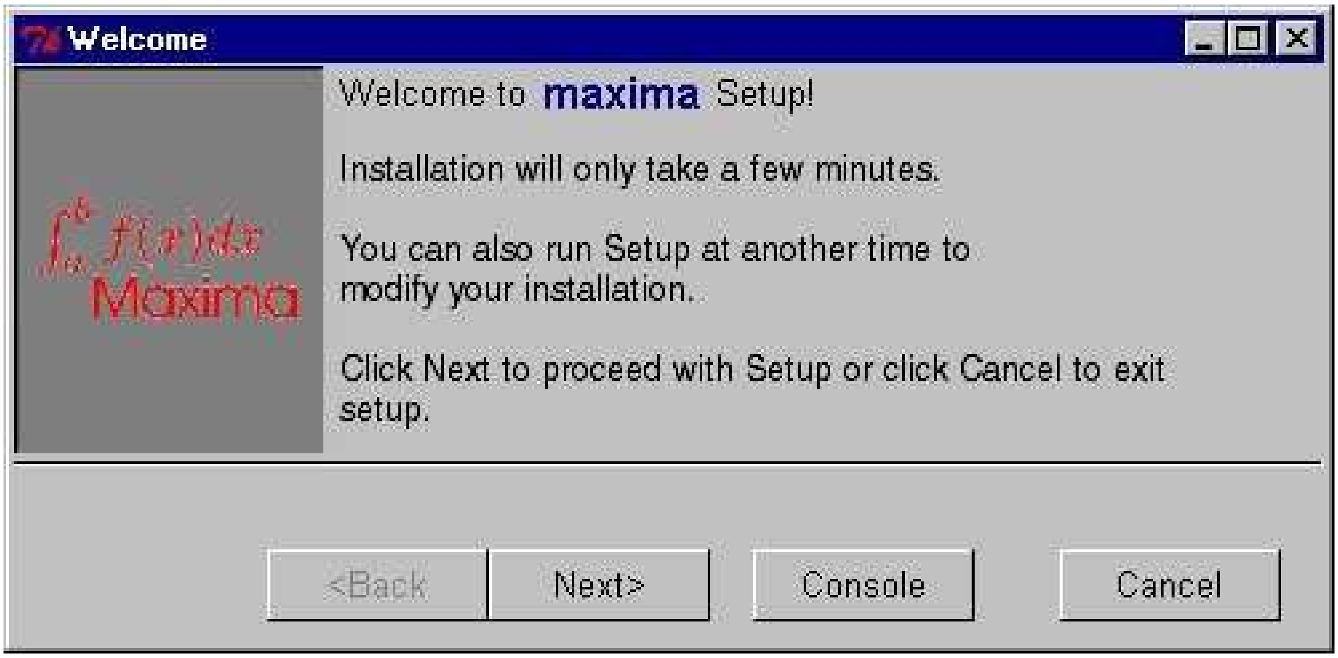
2.2 Installing on Windows

2.2.1. The Easy Way - Installing the Windows Binary

This is the recommended way to use Maxima on Windows. The current binary release is technically a beta, but should serve most needs quite well. Here is how a basic install works:

1. Download the binary from http://www.ma.utexas.edu/maxima.html It should be named maxima55l-setup.exe

2. Once you have downloaded that file, launch the installer. You should see this screen:



3. Simply follow the menus from that point - the install dialog should be very clear. Do read the README and License when they are presented - the information there is good to know.

4. Simply launch the program from it's icon - everything should work out of box. It is known to run on the following versions of Windows:

   Windows 98

2.2.2. The Hard Way

Compiling on Windows (I have no idea how to do this.)

CHAPTER 3

Available Interfaces to Maxima

Maxima is at heart a command line program, and by itself it is not capable of displaying formatted mathematics beyond the ascii text level. However there are other interfaces which may be used. Maxima has the ability to export expressions using the TEX syntax, and several programs use this device to help with output formatting. (None at this time allow formatted input.) All have their strengths and weaknesses - the choice will likely depend on the skill of the user and the task at hand. We will discuss here all of the interfaces currently available, and the user can make the choice him/herself which one to use.

3.1 The Terminal Interface

```
maxima
GCL (GNU Common Lisp)  Version(2.4.0) Wed May  9 12:02:00 CDT 2001
Licensed under GNU Library General Public License
Contains Enhancements by W. Schelter
Maxima 5.6 Wed May 9 12:01:49 CDT 2001 (with enhancements by W. Schelter).
Licensed under the GNU Public License (see file COPYING)
(C1) 'diff(x^2+y^x+t(y),y);

                                 d           x    2
(D1)                             -- (t(y) + y  + x )
                                 dy
(C2) ev(D1,diff);

                                d              x - 1
(D2)                            -- (t(y)) + x y
                                dy
(C3)
```

The terminal interface is the original interface to Maxima. While in some sense all of the interfaces to Maxima could be termed Terminal interfaces, when we refer to it here we mean the command line, no frills interface you would use in an xterm or a nongraphical terminal. It is the least capable of all the alternatives in many respects, but it is also the least demanding.

How comfortable this interface is depends to a large extent on what Lisp you used to compile Maxima originally. If you used the default, which is gcl, (what all binary packages I am aware of use) you do not have readline support on the terminal. This means you will not be able to use the back arrow key to go to the middle of an expression and change it - you must erase everything as you go back. This is a serious limitation, and if this is the case it is recommended by the author that you either compile against a Lisp flavor such as Clisp, which has readline support, or use one of the other interfaces. Any of the others should solve this problem. If you choose to use this interface, you activate it by simply typing `maxima` on the terminal prompt.

The only times I'd really recommend this interface are when none of the other options are viable, such as a telnet connection from a really primitive terminal. The other options, especially the Emacs mode, are far more comfortable working environments.

3.2 The Emacs Interface

A really excellent Emacs mode has been written for Maxima, and this is probably the choice I would recommend instead of the bare terminal, with or without a graphical interface. You get to go back in an expression without having to erase your expression, and in a graphical environment you get syntax highlighting, among many other goodies. There is also an environment for including Maxima input in LaTeX documents.

3.2.1. Installing the Maxima Emacs Mode

The Emaxima package consists of the files `maxima.el`, `emaxima.el`, `maxima-font-lock.el`, `emaxima.sty` and `emaxima.lisp`. To install, place the `.el` files, as well as `emaxima.lisp`[1] somewhere in the load path for Emacs. Finally, if you want to run LaTeX on the resulting document, put `emaxima.sty` somewhere in the TeX inputs path. If you use pdflatex, you'll also need `pdfcolmk.sty`.

Although AucTeX is not strictly necessary, you will most likely find it worth you time to install it, as many of the best features of Emaxima are LaTeX oriented.

Copy the `.el` and `.lisp` files to the site-lisp directory of your Emacs installation. On a Redhat Linux system, for example, this would be /usr/share/emacs/site-lisp, /usr/local/share/emacs/site-lisp, or some variation thereof. Copy the `.sty` files to a directory where LaTeX can see them - I've found on Redhat Linux /usr/share/texmf/tex/latex/emaxima works fairly well. Once you have done this, run the command mktexlsr. You should now be almost ready to roll.

The last step is to edit your .emacs file. In order to use the enhanced terminal mode, insert the following line:

```
(autoload 'maxima "maxima" "Maxima interaction" t)
```

If you wish to associate files ending in `.max` with this particular Emacs mode, add this line:

```
(setq auto-mode-alist (cons '("\\.max" . maxima-mode) auto-mode-alist)
```

This will allow you to start Maxima from within Emacs. You can do this one of two ways - either start Emacs and from within it type `M-x maxima`, or from the command line type `emacs -f maxima` to have the whole thing work in one step. If you wish to create a desktop icon to start the command line Maxima, simply place this line where they ask you what the name of your program or executable is, and it should work quite smoothly.

For Maxima-mode, add the following line to .emacs:

```
(autoload 'maxima-mode "maxima" "Maxima mode" t)
```

The command `M-x maxima-mode` will start you off here.

In the case of Emaxima, the line

```
(autoload 'emaxima-mode "emaxima" "EMaxima" t)
```

should be inserted into your .emacs file. Then typing `M-x emaxima-mode` will start Emaxima mode. The command `M-x emaxima-mark-file-as-emaxima` will put the line

```
%-*-EMaxima-*-
```

at the beginning of the file, if it isn't there already, and will ensure that the next time the file is opened, it will be in `emaxima-mode`. This can be done automatically everytime a file is put in `emaxima-mode` by putting the line

```
(add-hook 'emaxima-mode-hook 'emaxima-mark-file-as-emaxima)
```

somewhere in your `.emacs` file.

[1] If Emacs cannot find `emaxima.lisp`, then the TeX output functions will not work, any attempts to get TeX output will only result in standard output.

3.2.2. Maxima-mode

This mode is fairly basic, and is not dependant on LaTeX. It basically amounts to a text editor which allows you to send lines to Maxima.

For moving around in the code, Maxima mode provides the following
Maxima mode has the following completions commands:

Motion

Key	Description
`M-C-a`	Go to the beginning of the form.
`M-C-e`	Go to the end of the form.
`M-C-b`	Go to the beginning of the sexp.
`M-C-f`	Go to the end of the sexp.

Process

Key	Description
`C-cC-p`	Start a Maxima process.
`C-cC-r`	Send the region to the Maxima process.
`C-cC-b`	Send the buffer to the Maxima process.
`C-cC-c`	Send the line to the Maxima process.
`C-cC-e`	Send the form to the Maxima process.
`C-cC-k`	Kill the Maxima process.
`C-cC-p`	Display the Maxima buffer.

Completion

Key	Description
`M-TAB`	Complete the Maxima symbol.

Comments

Key	Description
`C-c ;`	Comment the region.
`C-c :`	Uncomment the region.
`M-;`	Insert a short comment.
`C-c *`	Insert a comment environment.

Indentation

Key	Description
`TAB`	Indent line.
`M-C-q`	Indent form.

Maxima help

Key	Description
`C-c C-d`	Get help on a (prompted for) subject.
`C-c C-m`	Read the manual.
`C-cC-h`	Get help with the symbol under point.
`C-cC-a`	Get apropos with the symbol under point.

Miscellaneous

Key	Description
`M-h`	Mark the form.
`C-c)`	Check the region for balanced parentheses.
`C-c C-)`	Check the form for balanced parentheses.

When something is sent to Maxima, a buffer running an inferior Maxima process will appear. It can also be made to appear by using the command `C-c C-p`. If an argument is given to a command to send information to Maxima, the region (buffer, line, form) will first be checked to make sure the parentheses are balanced. The Maxima process can be killed, after asking for confirmation with `C-cC-k`. To kill without confirmation, give `C-cC-k` an argument.

The behaviour of indent can be changed by the command `M-x maxima-change-indent-style`. The possibilities are:

Standard Standard indentation.

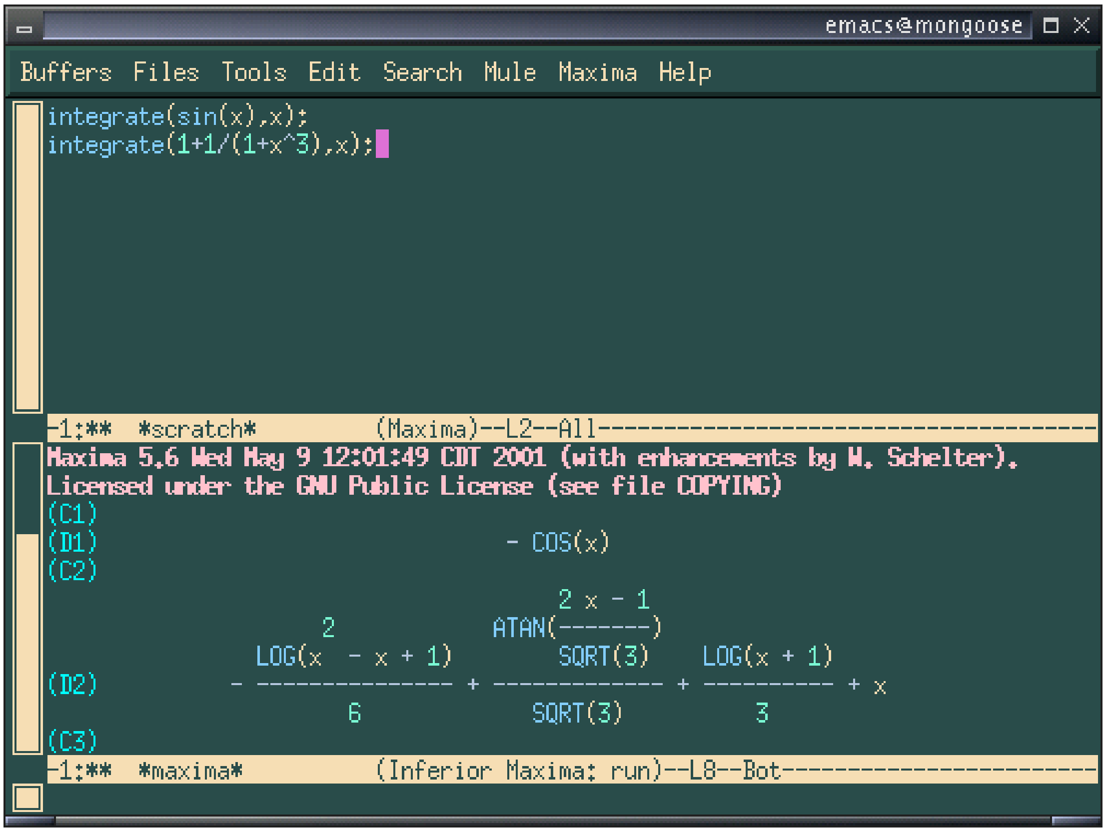


Figure 3.1: Maxima-mode

Perhaps smart Tries to guess an appropriate indentation, based on open parentheses, "do" loops, etc. A newline will re-indent the current line, then indent the new line an appropriate amount.

The default can be set by setting the value of the variable `maxima-newline-style` to either `'standard` or `'perhaps-smart`. In all cases, `M-x maxima-untab` will remove a level of indentation.

3.2.3. Enhanced Terminal Mode

For those just want a better terminal session, you can run a regular terminal style session in Emacs. This gives you everything the terminal interface does, plus syntax highlighting, plus more flexibility when editing your commands. If you already have a copy of Emacs open, you can start up the Maxima buffer by typing `M-x maxima` . If you do not have Emacs running, a shortcut is to start emacs using the following command: `emacs -f maxima` .

In the Maxima process buffer, return will check the line for balanced parentheses, and send line as input. Control return will send the line as input without checking for balanced parentheses. The following commands are also available.

`M-TAB`	Complete the Maxima symbol as much as possible, providing a completion buffer if there is more than one possible completion. (If `maxima-use-dynamic-complete` is non-nil, then instead this will cycle through possible completions.
`C-M-TAB`	Complete the input line, based on previous input lines.
`C-c C-d`	Get help on a Maxima topic.
`C-c C-m`	Bring up the Maxima info manual.
`C-cC-k`	Kill the process and the buffer, after asking for confirmation. To kill without confirmation, give `C-cC-k` an argument.
`M-p`	Bring the previous input to the current prompt.
`M-n`	Bring the next input to the prompt.
`M-r`	Bring the previous input matching a regular expression to the prompt.
`M-s`	Bring the next input matching a regular expression to the prompt.

3.2.4. Emaxima Mode

Emaxima is a major mode for Emacs that allows the user to insert Maxima sessions and code in a LaTeX document. It is based on Dan Dill's TeX/*Mathematica* package[2], and uses a modified version of William Schelter's `maxima.el`. Emaxima is an extension of the LaTeX mode provided by AUCTeX[3], and so has the LaTeX mode commands available. The resulting document can be processed by LaTeX; this requires putting

```
\usepackage{emaxima}
```

in the preamble.

This is in no sense a graphical environment, and the user will not see the benefits of any TeX formatting in real time. This mode is most useful to those who are accustomed to writing documents in LaTeX, and would like to include Maxima sessions in them. This manual itself is a good example of Emaxima in action.

Cells

The basic unit of Maxima code in Emaxima is a **cell**. A cell consists of text between the delimiters

```
\beginmaxima
```

and

```
\endmaxima
```

A cell can be created by typing `C-c C-o`. (The `C-o` in this case stands for **o**pening a cell.) The delimiters will then be placed in the buffer, and the point will be placed between them.

When working with several cells, you can jump between them by using `C-c +` to go to the next cell and `C-c -` to go to the previous cell.

[2] TeX/*Mathematica* is available from `ftp://chem.bu.edu/pub/tex-mathematica-2.0`.

[3] This can be configured so that Emaxima extends the standard TeX mode provided by Emacs, or just text mode.

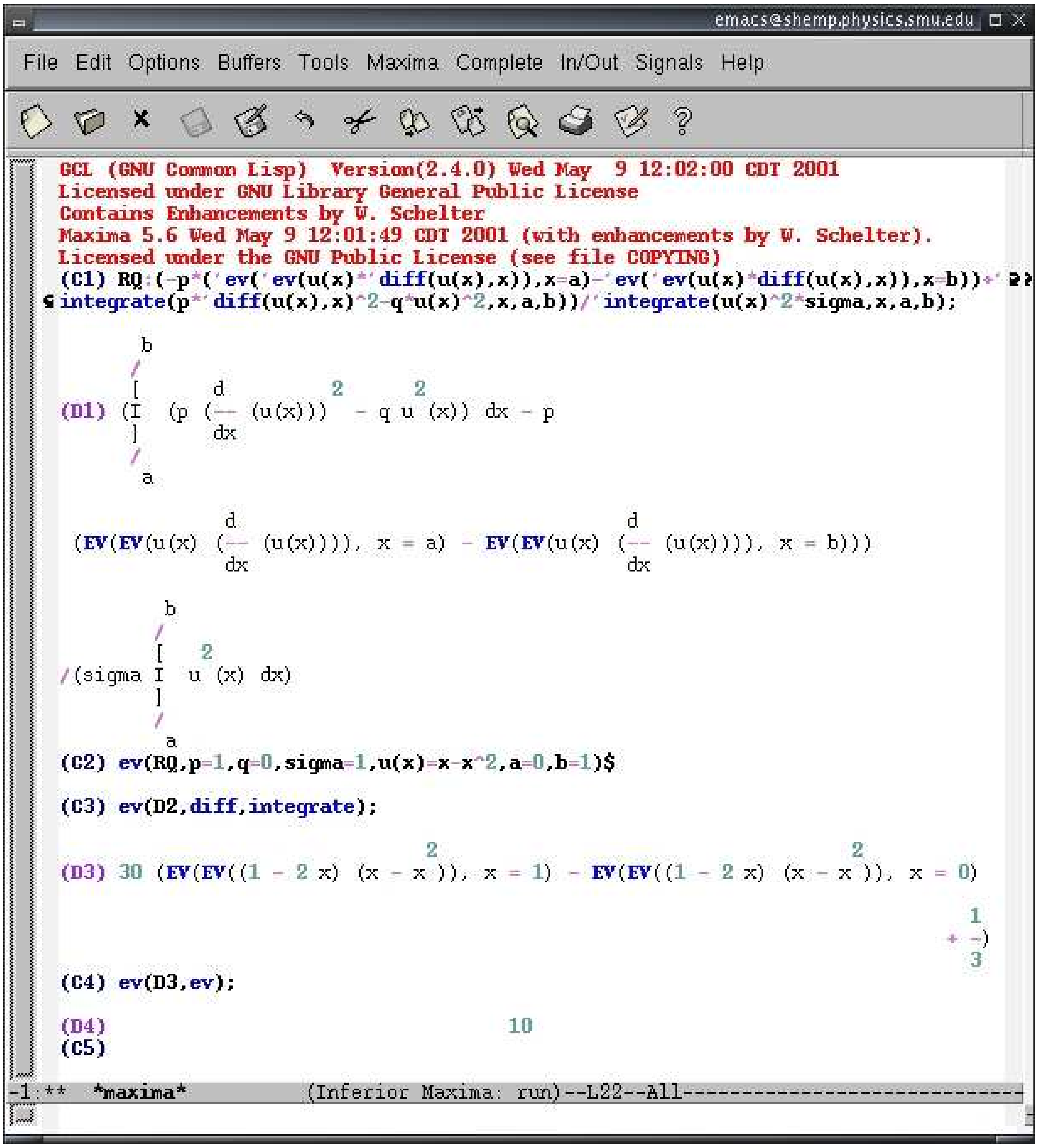


Figure 3.2: Enhanced Terminal Mode

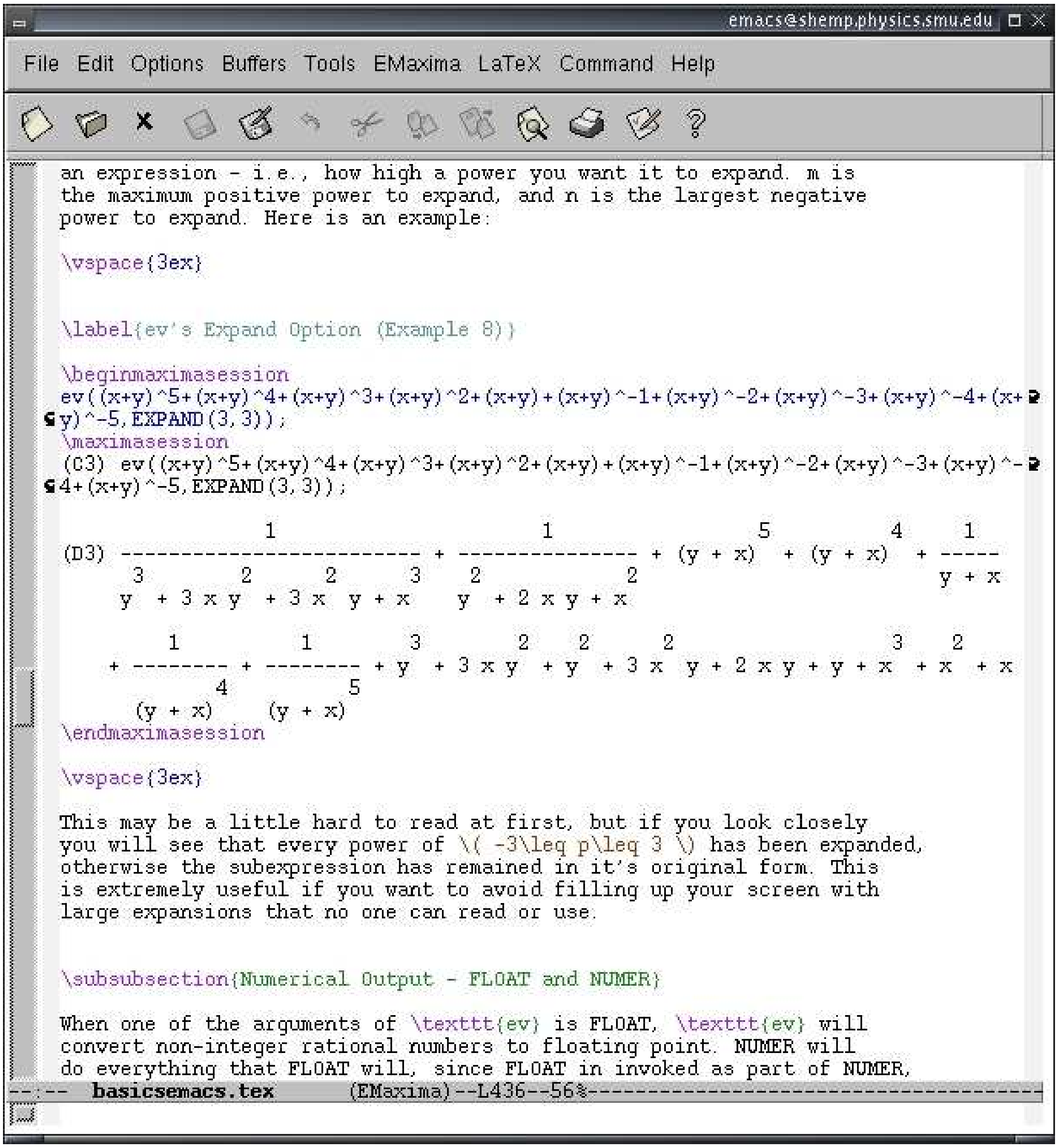


Figure 3.3: An Emaxima Session

Evaluating cells

To evaluate the contents of a cell, the command `C-cC-uc` (`emaxima-update-cell`)[4] will send the contents of the cell to a Maxima process (if there is no Maxima process running, one will be started) and return the results to the cell, separated from the input by the marker

```
\maximaoutput
```

To differentiate $sin(x^2)$, for example, type `diff(sin(x^2),x);` in a cell:

```
\beginmaxima
diff(sin(x^2),x);
\endmaxima
```

After typing `C-c C-u c`, it will look like

```
\beginmaxima
diff(sin(x^2),x);
\maximaoutput
                                  2
                          2 x COS(x )
\endmaxima
```

To delete the output and return the cell to its original form, you can use the command `C-c C-d`. If the document is to be TeXed, the above cell will look like:

```
Maxima
diff(sin(x^2),x);
```

and the cell with output will look like:

```
Maxima
diff(sin(x^2),x);
Output
                                  2
                          2 x COS(x )
```

Emaxima mode can take advantage of the fact that Maxima can give its output in LaTeX form. The command `C-c C-u C` works the same as `C-c C-u c`, except now the output is in LaTeX form, ready to be formatted by LaTeX. In general, if `C-c C-u` *letter* returns Maxima output, then `C-c C-u` *capital letter* will return the output in TeX form. The above cell would become

```
\beginmaxima
diff(sin(x^2),x);
\maximatexoutput
$$   2\*x\*\cos x^{2} $$
\endmaxima
```

which, when LaTeXed, would become

```
Maxima
diff(sin(x^2),x);
TeX Output
```

[4] Sending the cells contents to a Maxima process and returning the results is called **updating** the cell, the prefix `C-c C-u` will be used to update cells in different ways.

$$2x\cos x^2$$

(Note that whenever a cell is updated, any old output is discarded and replaced with new output.) The command `C-c C-u a` will update all of the cells in your document, stopping at each one to ask if you indeed want it updated. Given an argument, `C-u C-c C-u a`, it will update all of the cells in your document without asking. The command `C-c C-u A` behaves similarly, except now all the output is returned in LaTeX form.

Initialization Cells

It is possible that you want certain cells evaluated separate from the others; perhaps, for example, you want certain cells evaluated whenever you open the document. This can be done using initialization cells. An initialization cell is delimited by

```
\beginmaxima[* Initialization Cell *]
```

and

```
\endmaxima
```

The command `C-c C-t` will turn a cell into an initialization cell, applying `C-c C-t` again will turn it back into a regular cell. When LaTeXed, an initialization cell will look like

Maxima Initial

```
diff(sin(x^2),x);
```

Initialization cells behave like regular cells, except that they can be treated as a group. To evaluate all initialization cells (without displaying the output in the document buffer), the command `C-c C-u t` will go to each of the initialization cells and evaluate them. If you want the output of the initialization cells to be brought back to the document buffer, stopping at each one to see it you indeed want it updated, then use the command `C-c C-u i`. With an argument, `C-u C-c C-u i`, the initialization cells will be updated without asking. The command `C-c C-u I` behaves just like `C-c C-u i`, except that the output is returned in TeX form.

Referencing Other Cells

Instead of Maxima code, a cell can contain a reference to another cell, and when the original cell is sent to Maxima, the reference is replaced by the referenced cell's contents (but only in the Maxima process buffer, the cell's content in the document's buffer is not changed). In order to do this, the original cell must be marked by having a label of the form <*filename*:*cell label*>. (The reason for the *filename* will become apparent later, and *cell label* is optional for the referencing cell.) The referenced cell must also be labeled, with the same *filename* but a unique *cell label*. To reference the other cell, the original cell need only contain the marker for the referenced cell. For example, given cell 1:

```
\beginmaxima<filename:optional>
<filename:definef>
diff(f(x),x);
\endmaxima
```

and cell 2:

```
\beginmaxima<filename:definef>
f(x):=sin(x^2);
\endmaxima
```

then the result of updating cell 1 (`C-c C-u c`) will be:

```
\beginmaxima<filename:optional>
<filename:definef>
diff(f(x),x);
\maximaoutput
                                        2
                              f(x) := SIN(x )
                                      2
                                2 x COS(x )
\endmaxima
```

When LaTeXed, the top line will contain a copy of the marker.

```
Maxima                          <filename:optional>

<filename:definef>
diff(f(x),x);
Output
                                        2
                              f(x) := SIN(x )
                                      2
                                2 x COS(x )
```

A cell can contain more than one reference, and referenced cells can themselves contain references.

To aid in labelling the cells, the command `C-c C-x` will prompt for a label name and label the cell. To aid in calling references, the command `C-c C-TAB` can be used for completing the the *filename* and *cell label* parts of a reference, based on the current labels. Another option is to set the Emacs variable `emaxima-abbreviations-allowed` to `t`, say, by putting the line

```
(setq emaxima-abbreviations-allowed t)
```

in your `.emacs` file. This will allow the *filename* and *cell label* parts of a reference to be abbreviated by enough of a prefix to uniquely identify it, followed by ellipses `...` For example, if there are cells labelled

```
<filename:long description>
<filename:lengthy description>
```

Then the reference

```
<...:le...>
```

will suffice to refer to the second label above.

If you want the references in a cell to be replaced by the actual code, the command `C-c @` will expand all the references and put the code into a separate buffer (so it will not affect the original document).

WEB

The reason for the ability to reference other cells is so that you can write what Donald Knuth calls literate programs. The idea is that the program is written in a form natural to the author rather than natural to the computer. (Another aspect of Knuth's system is that the code is carefully documented, hence the name "literate programming", but that is done naturally in Emaxima.) Knuth called his original literate programming tool `WEB`, since, as he puts it, "the structure of a software program may be thought of as a web that is made up of many interconnected pieces." Emaxima's ability in this respect is taken directly from TeX/*Mathematica*, and is ultimately based on `WEB`. To create a program, the "base cell" or "package cell" should contain a label of the form <*filename*:> (no cell label), and can contain references of the form <*filename*:*part*> (same file name as the base cell).

As a simple (and rather silly) example, suppose we want to create a program to sum the first n squares. We could start:

```
\beginmaxima<squaresum.max:>
squaresum(n) := (
  <squaresum.max:makelist>
  <squaresum.max:squarelist>
  <squaresum.max:addlist>
  );
\endmaxima
```

We would then need cells

```
\beginmaxima<squaresum.max:makelist>,
L:makelist(k,k,1,n),
\endmaxima

\beginmaxima<squaresum.max:squarelist>
<squaresum.max:definesquare>
L:map(square,L),
\endmaxima

\beginmaxima<squaresum.max:addlist>
lsum(k,k,L)
\endmaxima
```

and then we would also need:

```
\beginmaxima<squaresum.max:definesquare>
square(k) := k^2,
\endmaxima
```

When TEXed, the header of the cell will say that it determines the file `squaresum.mu`.

Maxima — Definition of package squaresum.max

```
squaresum(n) := (
  <squaresum.max:makelist>
  <squaresum.max:squarelist>
  <squaresum.max:addlist>
  );
```

The command `C-u C-c @` will put all the pieces together in the file it determines. The resulting file, in this case, will be `squaresum.max` and will look like:

```
squaresum(n) := (
  L:makelist(k,k,1,n),
  square(k) := k^2,
  L:map(square,L),
  lsum(k,k,L)
  );
```

(Although the idea is that only the computer need look at this file.)

Other types of cells

When a cell is TEXed, the input and output are kept separate. To have the results look like a Maxima session, there are, in addition to the standard cells, special cells called *session cells*. A session cell is delimited by

```
\beginmaximasession
```

and

```
\endmaximasession
```

The command `C-c C-p` will create a session cell. When a session cell is updated, the output will be marked off with `\maximasession`, and will contain both the input and the output, with the Maxima prompts. For example, if the session cell

```
\beginmaximasession
diff(sin(x),x);
int(cos(x),x);
\endmaximasession
```

were updated, the result would look like

```
\beginmaximasession
diff(sin(x),x);
integrate(cos(x),x);
\maximasession
(C1)diff(sin(x),x);

(D1)                                  COS(x)
(C2)integrate(cos(x),x);

(D2)                                  SIN(x)
\endmaximasession
```

which, when TEXed, would look like

```
(C1)diff(sin(x),x);
(D1)                                  COS(x)
(C2)integrate(cos(x),x);
(D2)                                  SIN(x)
```

If it is updated in TEX form, it will look like

```
\beginmaximasession
diff(sin(x),x);
integrate(cos(x),x);
\maximatexsession
\C1.  diff(sin(x),x); \\
\D1.   \cos x \\
\C2.  integrate(cos(x),x); \\
\D2.   \sin x \\
\endmaximasession
```

which, when TEXed, will look like

(C1) diff(sin(x),x);
(D1)

$$\cos x$$

(C2) integrate(cos(x),x);
(D2)

$$\sin x$$

For particularly long output lines inside the `\maximatexsession` part of a session cell, the command `\DD` will typeset anything between the command and `\\`. Unfortunately, to take advantage of this, the output has to be broken up by hand. If a session cell has not been updated, or has no output for some other reason, it will not appear when the document is TEXed.

There is one other type of cell, a *noshow cell*, which can be used to send Maxima a command, but won't appear in the TEXed output. A noshow cell can be created with `C-c C-n`, and will be delimited by

```
\beginmaximanoshow
```

and

```
\endmaximanoshow
```

Session cells and noshow cells cannot be initialization cells or part of packages.[5]

If the command to create one type of cell is called while inside another type of cell, the type of cell will be changed. So, for example, the command `C-c C-p` from inside the cell

```
\beginmaxima
diff(x*sin(x),x);
\endmaxima
```

will result in

```
\beginmaximasession
diff(x*sin(x),x);
\endmaximasession
```

If a standard cell is an initialization cell or a package part, its type cannot be changed.

Miscellaneous

Some Maxima commands can be used even outside of cells. The command `C-c C-u l` send the current line to a Maxima process, comment out the current line, and insert the Maxima output in the current buffer. The command `C-c C-u L` will do the same, but return the result in LaTeX form.

The command `C-c C-h` will provide information on a prompted for function (like Maxima's `describe`), and `C-c C-i` will give the Maxima info manual.

Finally, the Maxima process can be killed with `C-c C-k`.

Customizing EMaxima

There are a few things that you can do to customize Emaxima.

By default, Emaxima is an extension of AUCTeX mode. This can be changed by changing the variable `emaxima-use-tex`. The possible values are `'auctex`, `'tex` and `nil`. Setting `emaxima-use-tex` (the default) to `'auctex` will make Emaxima an extension of AUCTeX, setting it to `'tex` will make Emaxima an extension of Emacs's default TeX mode, and setting `emaxima-use-tex` to `nil` will make Emaxima an extension of text-mode. So, for example, putting

```
(setq emaxima-use-tex nil)
```

in your `.emacs` file will make Emaxima default to an extension of text mode.

Whether or not the dots (...) abbreviation is allowed in cell references is controlled by the elisp variable `emaxima-abbreviations-a` which is set to `t` by default. Setting this to `nil` will disallow the abbreviations, but will speed up package assembly.

The LaTeXed output can also be configured in a couple of ways. The lines that appear around cells when the document is TeXed can be turned off with the command (in the LaTeX document)

```
\maximalinesfalse
```

They can be turned back on with the command

```
\maximalinestrue
```

The fonts used to display the Maxima input and output in a cell are by default `cmtt10`. They can be changed, seperately, by changing the TeX values of `\maximainputfont` and `\maximaoutputfont`. So, for example, to use `cmtt12` as the input font, use the command

```
\font\maximainputfont = cmtt12
```

[5]That could be changed, but I don't know why it'd be useful.

The spacing in the cells can be controlled by changing the TEX variables `\maximainputbaselineskip` and `\maximaoutputbaselinesk` and so to increase the space between the lines of the output, the command

```
\maximaoutputbaselineskip = 14pt
```

could be used. The amount of space that appears before a cell can be changed by changing the value of `\premaximaspace` (by default, 0pt), and that after a cell can be changed by changing the value of `\postmaximaspace` (by default, 1.5 ex).

Session cells can be configured similarly. Lines can be placed around a Maxima session with the command

```
\maximasessionlinestrue
```

and they can be turned back off with

```
\maximasessionlinesfalse
```

The font can be changed by changing the value of `\maximasessionfont`. The color of the prompts when the session is in TEX form is controlled by
`\maximapromptcolor`, by default red, the colors of the input lines and output lines are controlled by `\maximainputcolor` and `\maximaoutputcolor`, respectively. So the command

```
\def\maximainputcolor{green}
```

would make the input in a TEXed session green. The session can be TEXed without the colors by using the command `\maximasessionnocolor`. The baselineskip is set by `\maximasessionbaselineskip` for normal session cells, and by `\maixmatexsessionbaselineskip` for TEX sessions. The amount of space that appears before a session cell can be changed by changing the value of `\premaximasessionspace` (by default, 0pt), and that after a cell can be changed by changing the value of `\postmaximasessionspace` (by default, 1.5 ex).

Emaxima mode commands

Key	Description
`C-c C-o`	Create a (standard) cell.
`C-c C-p`	Create a session cell.
`C-c C-n`	Create a noshow cell.
`C-c +`	Go the the next cell.
`C-c -`	Go to the previous cell.
`C-c C-u a`	Update all of the cells. With an argument, don't ask before updating.
`C-c C-u A`	Update all of the cells in TEX form. With an argument don't ask before updating.
`C-c C-u t`	Evaluate all of the initialization cells.
`C-c C-u i`	Update all of the initialization cells. With an argument, don't ask before updating.
`C-c C-u I`	Update all of the initialization cells in TEX form. With an argument, don't ask before updating.
`C-c C-u s`	Update all of the session cells in TEX form. With an argument, don't ask before updating.
`C-c C-k`	Kill the current Maxima core - this will lose all data entered into the maxima system up until this point by other cells.

Commands only available in cells.

Key	Description
`C-c C-v`	Send the current cell to the Maxima process.
`C-c C-u c`	Update the current cell.
`C-c C-u C`	Update the current cell in TEX form.
`C-c C-d`	Delete the output from the current cell.
`C-c C-t`	Toggle whether or not the current cell is an initialization cell.
`C-c C-x`	Insert a heading for the cell indicating that it's part of a package.
`C-c @`	Assemble the references contained in the cell. With an argument, assemble the package that the cell defines.
`C-c C-TAB`	Complete a reference within a cell.

Commands only available outside of cells.

Key	Description
`C-c C-u l`	Send the current line to Maxima, and replace the line with the Maxima output.
`C-c C-u L`	Send the current line to Maxima, and replace the line with the Maxima output in TEX form.

AUCTEX commands

Inserting commands

Key	Description
`C-c C-e`	Insert an environment.
`C-c C-s`	Insert a section.
`C-c ]`	Close an environment.
`C-c C-j`	Insert an item into a list.
`"`	Smart quote.
`$`	Smart dollar sign.
`C-c @`	Insert double brace.
`C-c C-m`	Insert TEX macro.
`M-TAB`	Complete TEX macro.

Formatting

Key	Description
`C-c C-q C-r`	Format region.
`C-c C-q C-s`	Format section.
`C-c C-q C-e`	Format environment.
`C-c .`	Mark an environment.
`C-c *`	Mark a section.

Commenting

Key	Description
`C-c ;`	Comment a region.
`C-u C-c ;`	Uncomment a region.
`C-c %`	Comment a paragraph.
`C-u C-c %`	Uncomment a paragraph.

Font selection

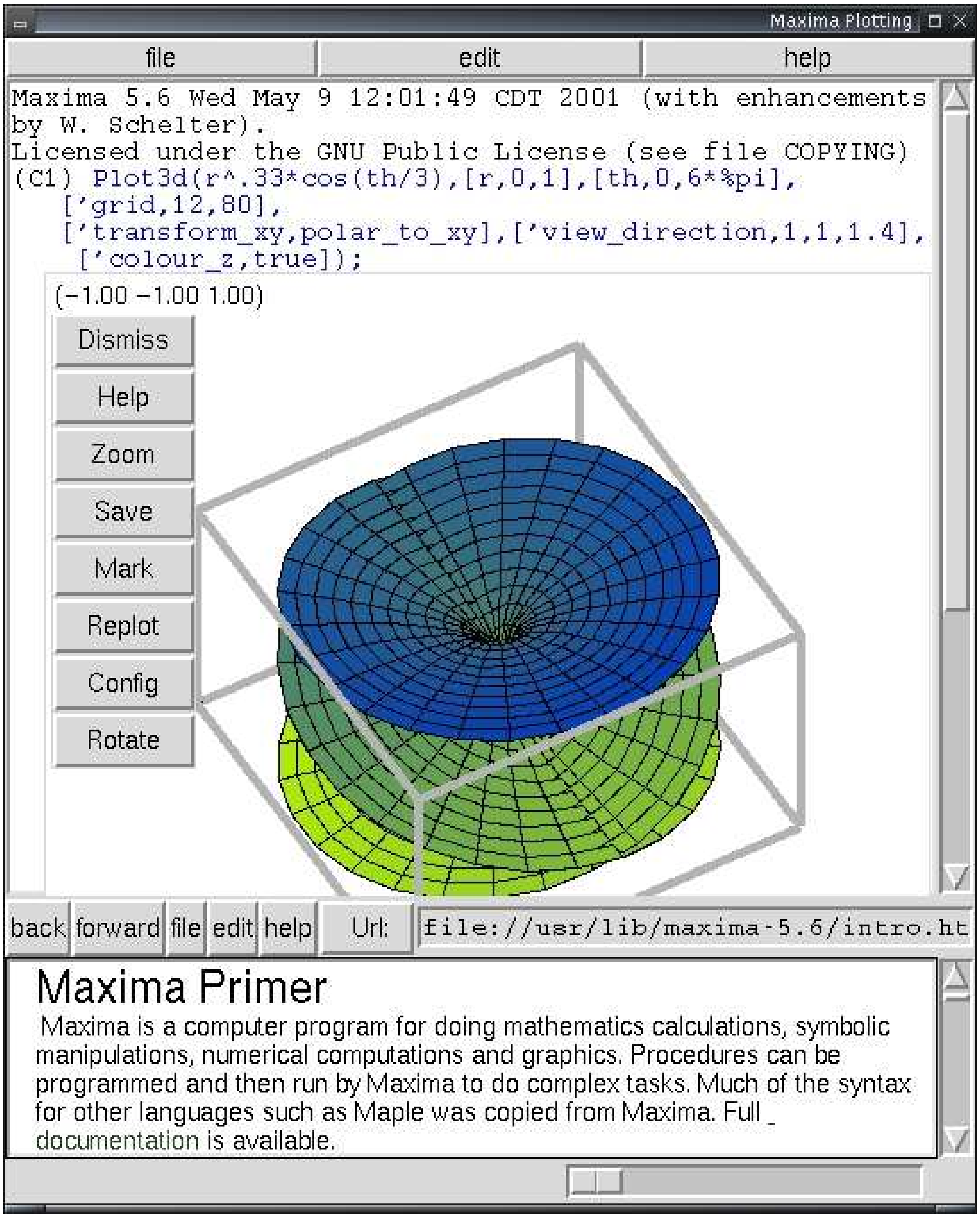
Key	Description
`C-c C-f C-b`	Bold.
`C-c C-f C-i`	Italics.
`C-c C-f C-r`	Roman.
`C-c C-f C-e`	Emphasized.
`C-c C-f C-t`	Typewriter.
`C-c C-f C-s`	Slanted.
`C-c C-f C-d`	Delete font.
`C-u C-c C-f`	Change font.

Running TEX

(Commands: `TeX`, `TeX Interactive`, `LaTeX`, `LaTeX Interactive`, `SliTeX`, `View`, `Print`, `BibTeX`, `Index`, `Check`, `File`, `Spell`.)

Key	Description
`C-c C-c`	Run a command on the master file.
`C-c C-r`	Run a command on the current region.
`C-c C-b`	Run a command on the buffer.
``C-c ` ``	Go to the next error.
`C-c C-k`	Kill the TEX process.
`C-c C-l`	Center the output buffer.
`C-c C-^`	Switch to the master file.
`C-c C-w`	Toggle debug of overful boxes.

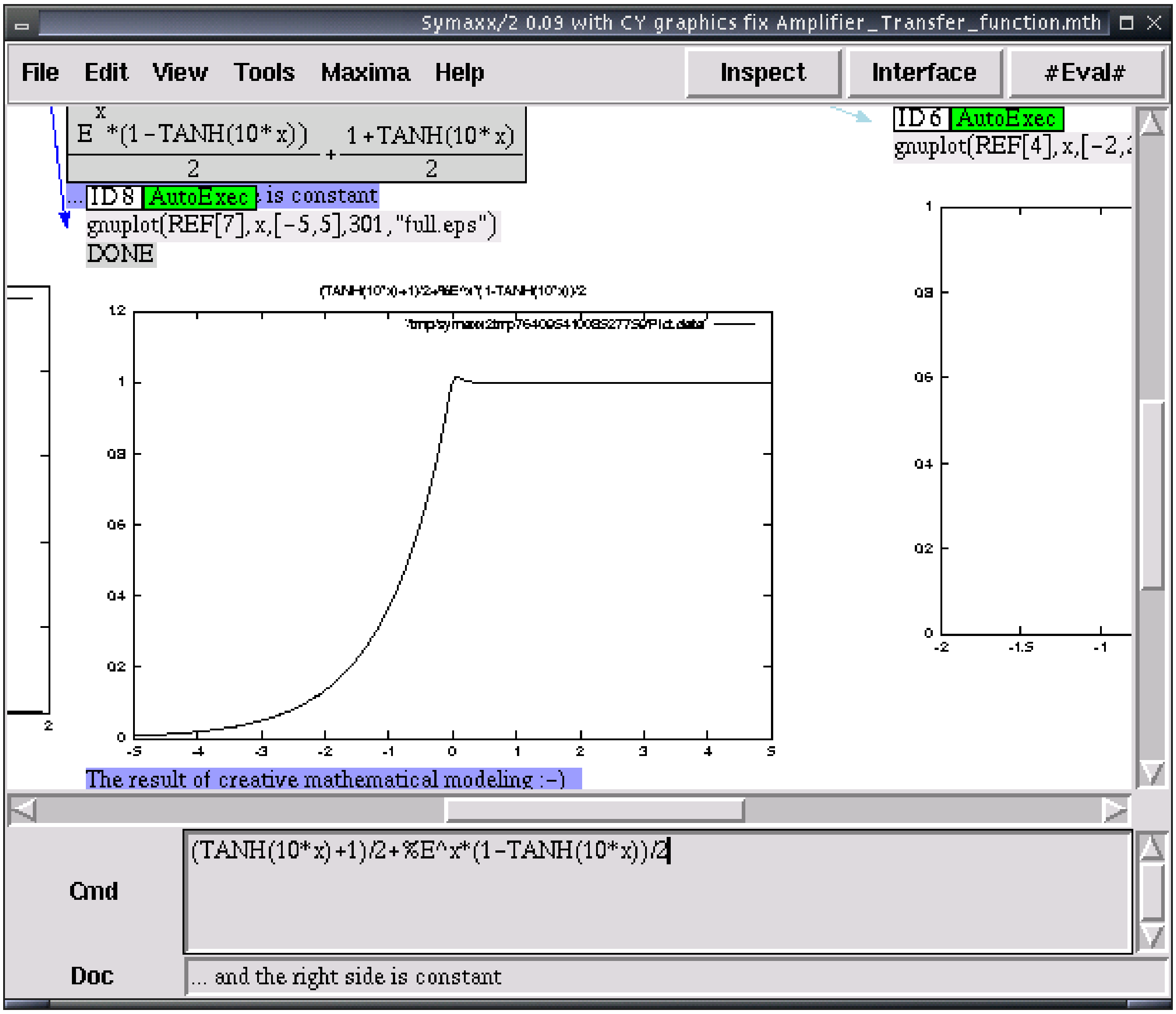
3.3 Xmaxima



Xmaxima is a new development in the lifetime of Maxima. It is based on Tcl/Tk, and has the nifty feature of in-line plotting when using the default plotting routine. No version earlier than ? has this interface, so if you aren't using the new versions you won't have this included in your default package. This is probably where most people will start learning Maxima, and it is not a bad place to start. You avoid the earlier mentioned limitations of the vanilla terminal, and avoid having to master the setup for Emacs. In order to start this interface simply type xmaxima in a terminal. From there, you should get what looks like the terminal interface in a Tk window, with an introductory html document below it. There is also a pull down menu system. For Windows users, this will more than likely be the default choice. Symaxx and TEXmacs are Unix only.

3.4 Symaxx

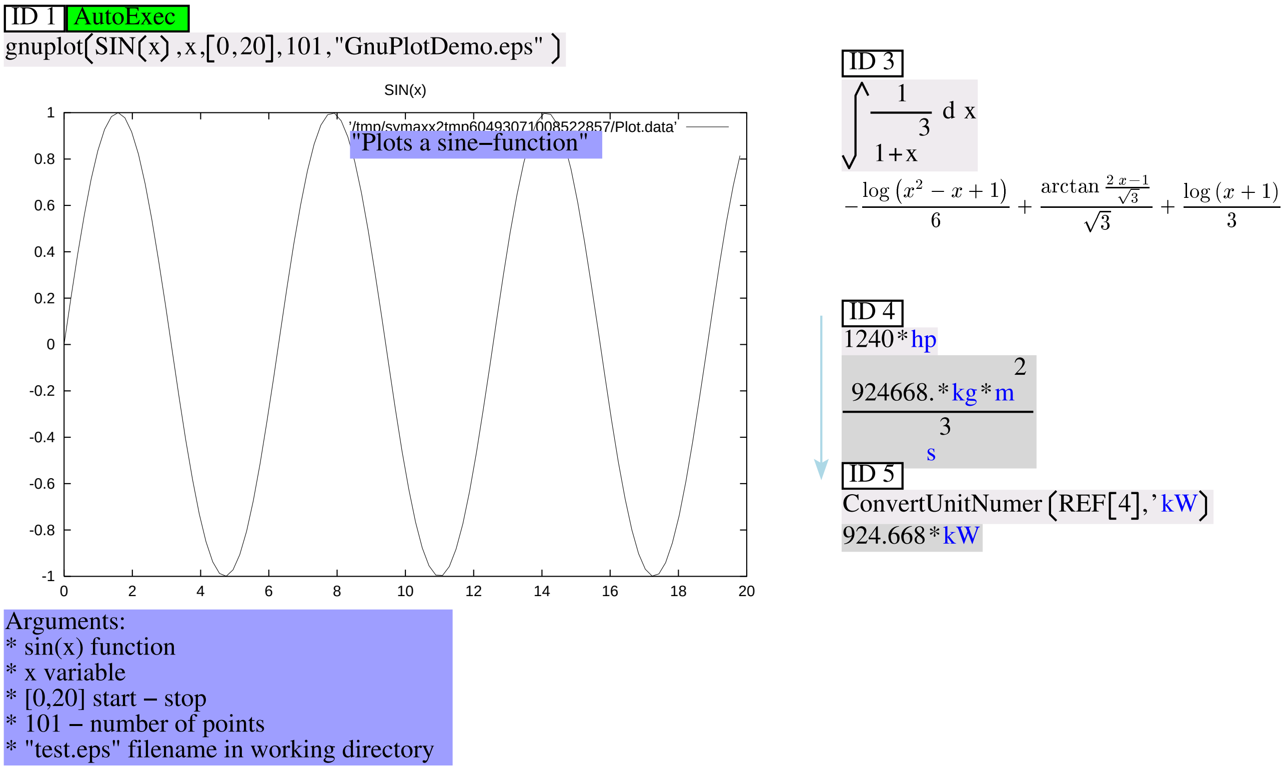
In the words of its creator Markus Nentwig "Symaxx provides those 5% of features, that are needed to do 95% of your work." Perhaps the best description of its display is that of a mathematical flowchart - it indicates relationships between cells with arrows, and allows free placement of expressions on a canvas. It is more graphical than any of the other Maxima interfaces, and although input is still ascii based the output is formatted. It is based on Perl and Tk, both of which are required to run it. This interface tends, at least in the experience of this author, to be a bit resource intensive, but allows some page formatting possibilities which make it a very interesting program. We will show you some of its features here, and if you decide to use this interfacethe Symaxx manual is a highly recommended read.



Symaxx uses a fairly basic display format - each unit of mathematical input and output is labeled with an ID number, which one can use to refer to that particular unit elsewhere in your document. There are three levels to each unit - the input box, which displays the command input into Maxima, the output box, which displays the results, and the documentation box, where you can record information and descriptions which are not intended to be evaluated. This may take a little getting used to - pressing return once you are done creating the input expression doesn't evaluate it, but only sends it to the formatter used by Symaxx to create a graphical representation of the input. To send the command to Maxima, you press the evaluate button.

For graphing purposes, Symaxx is able to embed gnuplot figures. Also included in its abilities are the ability to use

TeX to display output, manipulate units, and export to postscript files. Here is a sample output showing these abilities in action:



(Add to troubleshooting tips: Some versions of Symaxx seem to have trouble interacting with some versions of netpbm - if Symaxx should fail to display graphs, try this solution: get solution from Symaxx discussion forum)

In order to utilize the full abilities of Symaxx, you must have a fair bit of supporting software installed. There is Maxima itself, of course, and then there are the following:

- Perl
- Tk800.022
- netpbm
- TEX for advanced output formatting

Here are some basic install instructions. We will assume here you need to install Tk800.022 and Symaxx, having already installed the others earlier. If you haven't they are readily available on most Linux distributions.

```
user$ tar -xvzf Tk800.022.tar.gz
user$ cd Tk800.022

If you have root access, do the following:

user$ perl Makefile.PL
user$ make
user$ make test (optional)
user$ su
Password:
root$ make install
```

```
if you don't have root access, use

user$ perl Makefile.PL LIB=/home/(place to install) or
user$ perl Makefile.PL PREFIX=/home/(place to install).
user$ make; make install
```

Find the location of the folder 'Tk' in the folder you gave as argument to LIB or PREFIX. Now you'll have to change the first line in 'symaxx' and 'Symaxx2/Watchdog' to include the Tk library. Append a `-I /home/(where you installed Tk)(The folder that contains Tk)` as in `#!/usr/bin/perl -w -I /home/somewhere/`

Then simply decompress Symaxx in the directory of your choice. If perl is not in /usr/bin, you'll have to change the first line of the files 'symaxx' and 'Symaxx2/Watchdog' to the location on your system, as found by 'whereis perl'.

3.5 TeXmacs

TeXmacs is a new WYSIWYG scientific editor, and it has the ability to interface with many computer algebra systems, including Maxima. This program takes advantage of the TeX output Maxima can produce to format it's output. To launch Maxima inside of TeXmacs you go up to the menu and select Insert -> sessions -> maxima. Then things work like they do in xmaxima or Emacs. This is probably the most visually appealing way to run Maxima. You can insert a maxima session in TeXmacs by selecting Insert->session->maxima from the menu. From there the interface will feel pretty much like a regular terminal.

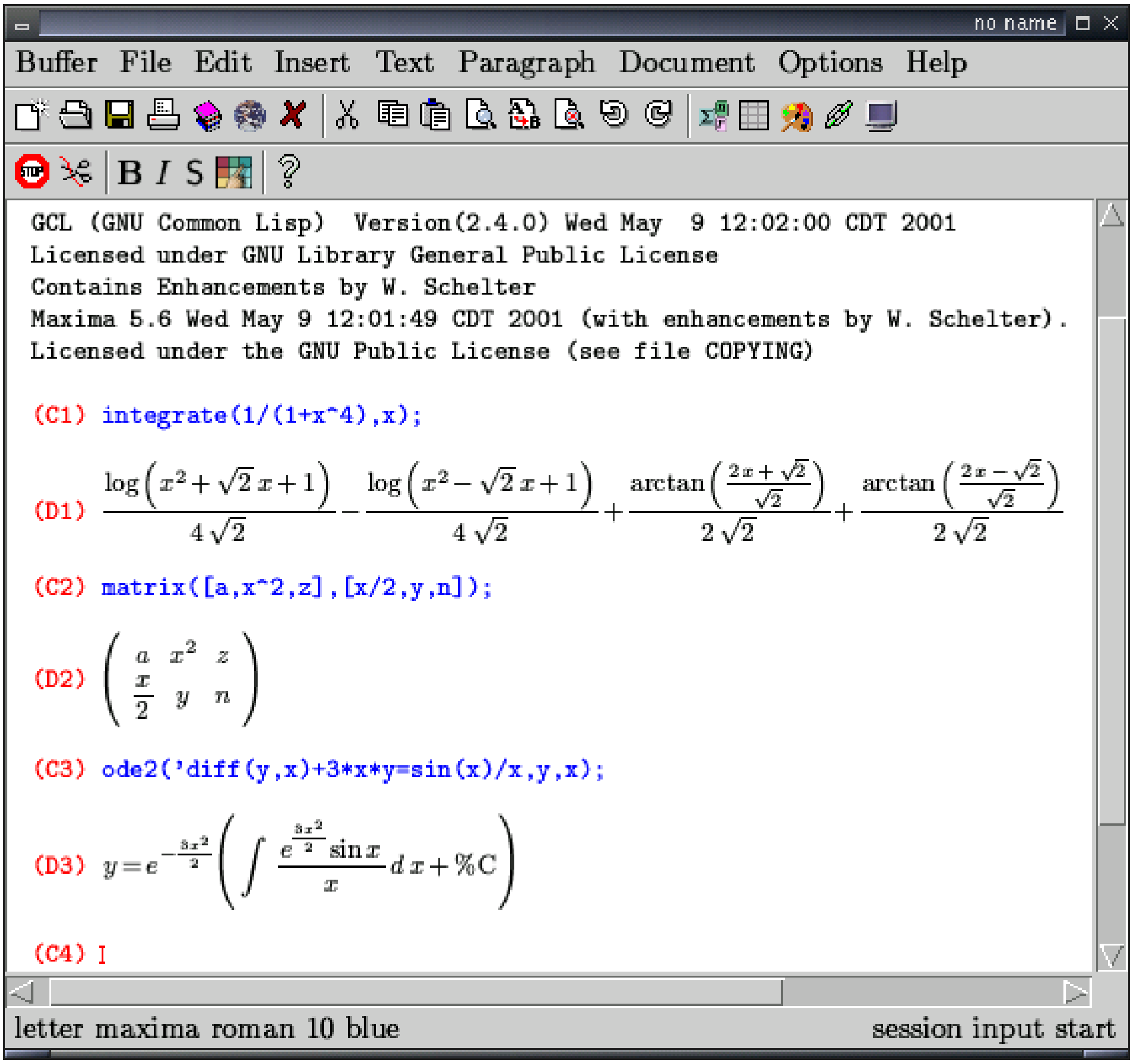


Figure 3.4: TeXmacs with a Maxima Session

CHAPTER 4

The Basics - What you need to know to operate in Maxima

Here we will attempt to address universal concepts which you will need to know when using Maxima for a wide variety of tasks.

4.1 The Very Beginning

All computer algebra systems have syntactical rules, i.e. a structured language by which the user communicates his/her commands to the system. Without being able to communicate in this language, it is impossible to accomplish anything in such as system. So we will attempt to describe herein the basics.

4.1.1. Our first Maxima Session

We will start by demonstrating the ultimate basics: $+$, $-$, $*$, and /. These symbols are virtually universal in any mathematical system, and mean exactly what you think they mean. We will demonstrate this, and at the same time introduce you to your first session in Maxima. In the interface of your choice, try the following:

```
GCL (GNU Common Lisp)  Version(2.3.8) Wed Sep  5 08:00:22 CDT 2001
Licensed under GNU Library General Public License
Contains Enhancements by W. Schelter
Maxima 5.5 Wed Sep 5 07:59:43 CDT 2001 (with enhancements by W. Schelter).
Licensed under the GNU Public License (see file COPYING)
```

(C1) `2+2;`
(D1)

$$4$$

(C2) `3-1;`
(D2)

$$2$$

(C3) `3*4;`
(D3)

$$12$$

(C4) `9/3;`
(D4)

$$3$$

(C5) `9/4;`
(D5)

$$\frac{9}{4}$$

(C6) `quit();`

Above is our first example of a Maxima session. We notice already several characteristics of a Maxima session: The startup message, which gives the version of Maxima being used, the date of compilation, which is the day your executable was created, the labels in front of each line, the semicolon at the end of each line, and the way we exit the session. The startup message is not important to the session, but you should take note of what version of Maxima you are using, especially if there is a known problem in an earlier version which might impact what you are trying to do.

Exiting a Maxima Session

You want to be able to get out of what you get into. So the first command we discuss will be the command that gets you out of Maxima, and while we are at it we will discuss how to get out of debugging mode. Debugging mode is quite useful for some things, and the reader is encouraged to look to later chapters for an in-depth look at the debugging mode, but for now we will stick to basics. As you see above, `quit();` is the command which will exit Maxima. This is a bit confusing for new users, but you must type that full command. Simply typing `quit` or `exit` will not work, nor will pressing `CTRL-C` - if you try the latter you will be dumped into the debugging mode. If that happens, simply type `:q` if you are running GNU Common Lisp, or `:a` if running CLISP. (If in doubt use `:q` - most binary packages use GCL at this time.) Here's an example of what not to do, and how to get out of it if you do:

```
(C1) quit;

(D1)                               QUIT
(C2) exit;

(D2)                               EXIT
(C3)
Correctable error:  Console interrupt.
Signalled by MACSYMA-TOP-LEVEL.
If continued:  Type :r to resume execution, or :q to quit to top level.
Broken at SYSTEM:TERMINAL-INTERRUPT.  Type :H for Help.
MAXIMA>>
Correctable error:  Console interrupt.
Signalled by SYSTEM:UNIVERSAL-ERROR-HANDLER.
If continued:  Type :r to resume execution, or :q to quit to top level.
Broken at SYSTEM:TERMINAL-INTERRUPT.
MAXIMA>>>:q

(C3) quit();
```

The first two lines show what happens if you forget the `()` or type `exit`. Nothing major, but you won't exit Maxima. `CTRL-C` causes a few more problems - if you actually read the message, you will see it tells you how to handle it. In the above example, `CTRL-C` was hit twice - that is not a proper way to exit Maxima either. Just remember to use `:q` to exit debugging and `quit();` to exit Maxima, and you should always be able to escape trouble. If you find youself trying to read a long output going by quickly on a terminal, press `CTRL-S` to temporarily halt the output, and `CTRL-Q` to resume.

The End of Entry Character

All expressions entered into Maxima must end with either the ; character or the $ character. The ; character is the standard character to use for this purpose. The $ symbol, while performing the same job of ending the line, suppresses the output of that line. This example illustrates these properties:

(C1) `x^5+3*x^4+2*x^3+5*x^2+4*x+7;`

(D1)

$$x^5+3x^4+2x^3+5x^2+4x+7$$

(C2) `x^5+3*x^4+2*x^3+5*x^2+4*x+7$`

```
(C3)  D2;
(D3)
```

$$x^5+3x^4+2x^3+5x^2+4x+7$$

```
(C4)  x^5+3*x^4+2*x^3
+5*x^2+4*x+7;
(D4)
```

$$x^5+3x^4+2x^3+5x^2+4x+7$$

```
(C5)  x^5+3*x^4+2*x^3
     +5*x^2+4*x+7;
(D5)
```

$$x^5+3x^4+2x^3+5x^2+4x+7$$

In (C1), we input the expression using the ; character to end the expression, and on the return line we see that D1 now contains that expression. In (C2), we input an identical expression, except that we use the $ to end the line. (D2) is assigned the contents of (C2), but does not visually display those contents. Just to verify that (D2) does in fact contain what we think it contains we ask Maxima to display it's contents on (C3) and we see that the are in fact present. This is extremely useful if you are working on a problem which has many steps, and some of those steps would produce long outputs you don't need to actually see. In (C4), we input part of the expression, press return, finish the expression, and then use the ; character. Notice that the input did not end until we used that character and pressed return - return by itself does nothing. You see (D4) contains the same expression as (D1) shown above. To Maxima, the inputs are the same. This can be useful if you are going to input a long expression and wish to keep it straight visually, to avoid errors. You can also input spaces without adversely affecting the formula, as shown in (C5).

The (C*) and (D*) Labels

These labels are more than just line markers - they are actually the names in memory of the contents of the lines. This is quite useful for a number of tasks. Let's say you wish to apply a routine, say a `solve` routine, to an expression for several different values. Rather than retyping the entire expression, we can use the fact that the line numbers act as markers to shorten our task considerably, as in this example:

```
(C1)  3*x^2+7*x+5;
(D1)
```

$$3x^2+7x+5$$

```
(C2)  solve(D1=3,x);
(D2)
```

$$\left[x=-\frac{1}{3},x=-2\right]$$

```
(C3)  solve(D1=7,x);
(D3)
```

$$\left[x=-\frac{\sqrt{73}+7}{6},x=\frac{\sqrt{73}-7}{6}\right]$$

```
(C4)  solve(D1=a,x);
(D4)
```

$$\left[x=-\frac{\sqrt{12a-11}+7}{6},x=\frac{\sqrt{12a-11}-7}{6}\right]$$

In this example, we desire to solve the expression $3x^2+7x+5$ for x when $3x^2+7x+5=3$, $3x^2+7x+5=7$, and $3x^2+7x+5=a$. (a in this case is an arbitrary constant.) Rather than retype the equation many times, we merely enter it once, and then use that label to set up the similar problems more easily.

The (E*) Labels

In some cases, particularly when a command needs to assign generated values to variable names, the E labels will be used. These may be treated like any other maxima variable. Here is an example of E label use:

Custom Labels

You do not need to settle for this method of labeling - you can define your own expressions if you so choose, by using the : assign operator. Let us say, for example, that we wish to solve the problem above, but would rather call our equation FirstEquation than D1. We will show that process here, with one deliberate error for illustration of a property:

```
(C5) FirstEquation:3*x^2+7*x+5;
(D5)
```

$$3x^2+7x+5$$

```
(C6) solve(FirstEquation=3,x);
(D6)
```

$$\left[x=-\frac{1}{3},x=-2\right]$$

```
(C7) solve(FirstEquation=7,x);
(D7)
```

$$\left[x=-\frac{\sqrt{73}+7}{6},x=\frac{\sqrt{73}-7}{6}\right]$$

```
(C8) solve(FirstEquation=a,x);
(D8)
```

$$\left[x=-\frac{\sqrt{12a-11}+7}{6},x=\frac{\sqrt{12a-11}-7}{6}\right]$$

```
(C9) solve(firstequation=a,x);
(D9)
```

$$[]$$

You see that this process works exactly the same as before. On line (C5), you see we entered the name of FirstEquation as lower case, and the calculation failed. These names are case sensitive. This is true of all variables. Later on you will see cases in Maxima such as sin, where SIN and sin are the same, but it is not safe to assume this is always true and when in doubt, watch your cases. In general we suggest you use lower case for your maxima commands and programs - it will make them easier to read and debug.

4.1.2. To Evaluate or Not to Evaluate

Operators in Maxima, such as diff for derivative, are a common feature in many Maxima expressions. The problem is, while you need to include an operator at a given point in your process, you may not want to deal with the output from it at that point in the problem. Therefore, Maxima provides the ' toggle for operators. See the example below for an example of how this works.

```
(C10) diff(1/sqrt(1+x^3),x);
(D10)
```

$$-\frac{3x^2}{2\left(x^3+1\right)^{\frac{3}{2}}}$$

```
(C11) 'diff(1/sqrt(1+x^3),x);
```

(D11)

$$\frac{d}{dx}\frac{1}{\sqrt{x^3+1}}$$

4.1.3. The Concept of Environment - The ev Command

All mathematical operations in Maxima take place in an environment, which is to say the system is assuming it should do some things and not do other things. There will be many times you will want to change this behavior, without doing so on a global scale. Maxima provides a way to define a local environment on a per command basis, using the `ev` command. `ev` is one of the most powerful commands in Maxima, and the user will benefit greatly if they master this command early on while using Maxima.

From the top

We will begin with a very simple example:

```
(C1) ev(solve(a*x^2+b*x+c=d,x),a=3,b=4,c=5,d=6);
(D1)
```

$$\left[x=-\frac{\sqrt{7}+2}{3},x=\frac{\sqrt{7}-2}{3}\right]$$

```
(C2) a;
(D2)
```

$$a$$

The first line uses the ev command to solve for x without setting variables in the global environment. To make sure that our variables remain undefined, we check that a is still undefined in line (C2), and it is.

Now lets examine some of the more interesting features of ev. The general syntax of the ev command is ev(exp, arg1, ..., argn). exp is an expression, like the one in the example above. You can also use a D* entry name or your own name for an expression. arg* has many possibilities, and we will try to step through them here.

EXPAND(m,n)

Expand is an argument which allows you to limit how Maxima expands an expression - i.e., how high a power you want it to expand. m is the maximum positive power to expand, and n is the largest negative power to expand. Here is an example:

```
   (C3) ev((x+y)^5+(x+y)^4+(x+y)^3+(x+y)^2+(x+y)+(x+y)^-1+(x+y)^-2+(x+y)^-3+(x+y)^-4+(x+y)^-
5,EXPAND(3,3));
                  1                        1                  5          4     1
   (D3) ------------------------- + --------------- + (y + x)  + (y + x)  + -----
         3         2       2      3     2            2                      y + x
        y  + 3 x y  + 3 x  y + x     y  + 2 x y + x
        1           1          3        2     2       2                 3    2
     + -------- + -------- + y  + 3 x y  + y  + 3 x  y + 2 x y + y + x  + x  + x
              4          5
       (y + x)    (y + x)
```

This may be a little hard to read at first, but if you look closely you will see that every power of $-3 \leq p \leq 3$ has been expanded, otherwise the subexpression has remained in it's original form. This is extremely useful if you want to avoid filling up your screen with large expansions that no one can read or use.

Numerical Output - FLOAT and NUMER

When one of the arguments of `ev` is FLOAT, `ev` will convert non-integer rational numbers to floating point. NUMER will do everything that FLOAT will, since FLOAT in invoked as part of NUMER. NUMER also handles variables defined by the user with the NUMERVAL command, which the FLOAT toggle will leave unevaluated. In order to evaluate these expressions, you can also use the `float` command.

```
(C1) a:9/4;
(D1)
```

$$\frac{9}{4}$$

```
(C2) exp(a);
(D2)
```

$$e^{\frac{9}{4}}$$

```
(C3) ev(exp(a),FLOAT);
(D3)
```

$$9.487735836358526$$

```
(C4) ev(exp(a*x),FLOAT);
(D4)
```

$$e^{2.25x}$$

```
(C5) numerval(b, 25);
(D5)
```

$$[b]$$

```
(C6) a*b;
(D6)
```

$$\frac{9b}{4}$$

```
(C7) ev(a*b,FLOAT);
(D7)
```

$$2.25b$$

```
(C8) ev(a*b,NUMER);
(D8)
```

$$56.25$$

```
(C9) float(a);
(D9)
```

$$2.25$$

```
(C10) float(b);
(D10)
```

$$25$$

```
(C11) float(a*b);
(D11)
```

$$56.25$$

Specifying Local Values for Variables, Functions, etc.

One of the best things about the ev command is that for one evaluation you may specify in an arg what values are to be used for the evaluation in place of variables, how to define functions, which functions to evaluate, etc. We will work through a series of examples here, probably this will be the best way to illustrate the various possibilities of this aspect of ev.

```
(C8)  eqn1:'diff(x/(x+y)+y/(y+z)+z/(z+x),x);
(D8)
```

$$\frac{d}{dx}\left(\frac{y}{z+y}+\frac{z}{z+x}+\frac{x}{y+x}\right)$$

```
(C9)  ev(eqn1,diff);
(D9)
```

$$-\frac{z}{(z+x)^2}+\frac{1}{y+x}-\frac{x}{(y+x)^2}$$

```
(C10)  ev(eqn1,y=x+z);
(D10)
```

$$\frac{d}{dx}\left(\frac{z+x}{2z+x}+\frac{x}{z+2x}+\frac{z}{z+x}\right)$$

```
(C11)  ev(eqn1,y=x+z,diff);
(D11)
```

$$\frac{1}{2z+x}-\frac{z+x}{(2z+x)^2}+\frac{1}{z+2x}-\frac{2x}{(z+2x)^2}-\frac{z}{(z+x)^2}$$

In this example, we define eqn1 to be the derivative of a function, but use the ' character in front of the diff operator to notify Maxima that we don't want it to evaluate that derivative at this time. (More on that in the ?? section.) In the next line, we use the ev with the diff argument, which instructs ev to take all derivatives in this expression. Now, let's say we want to define y as a function of z and x, but again avoid evaluating the derivative. We supply our definition of y as an argument to ev, and in (D3) we see that the substitution has been made. Now, let's evaluate the derivative after the substitution has been made. We work as before, except this time we supply both the new definition of y and the diff argument, telling ev to make the substitution and then take the derivative. In this particular case, the order of the arguments does not matter. The case where it will matter is if you are making multiple substitutions - then they are handled in sequence from left to right.

(need example here, one where the difference is noticeable).

We can also locally define functions:

```
(C12)  eqn4:f(x,y)*'diff(g(x,y),x);
(D12)
```

$$f(x,y)\left(\frac{d}{dx}g(x,y)\right)$$

```
(C13)  ev(eqn4,f(x,y)=x+y,g(x,y)=x^2+y^2);
(D13)
```

$$(y+x)\left(\frac{d}{dx}\left(y^2+x^2\right)\right)$$

```
(C14)  ev(eqn4,f(x,y)=x+y,g(x,y)=x^2+y^2,DIFF);
(D14)
```

$$2x(y+x)$$

(At the moment, ev seems to take only the first argument in the following example from solve: the manual seems to indicate it should be taking both as a list??)

```
(C15) eqn1:f(x,y)*'diff(g(x,y),x);
(D15)
```

$$f(x,y)\left(\frac{d}{dx}g(x,y)\right)$$

```
(C16) eqn2:3*y^2+5*y+7;
(D16)
```

$$3y^2+5y+7$$

```
(C17) ev(eqn1,g(x,y)=x^2+y^2,f(x,y)=5*x+y^3,solve(eqn2=5,y));
(D17)
```

$$\left(5x-\frac{8}{27}\right)\left(\frac{d}{dx}\left(x^2+\frac{4}{9}\right)\right)$$

```
(C18) ev(eqn1,g(x,y)=x^2+y^2,f(x,y)=5*x+y^3,solve(eqn2=1,y),diff);
(D18)
```

$$2x\left(5x-\frac{\left(\sqrt{47}i+5\right)^3}{216}\right)$$

```
(C19) ev(eqn1,g(x,y)=x^2+y^2,f(x,y)=5*x+y^3,solve(eqn2=1,y),diff,FLOAT);
(D19)
```

$$2x\left(5x-0.00462962962963\left(\sqrt{47}i+5\right)^3\right)$$

Other arguments for ev

INFEVAL - This option leads to an "infinite evaluation" mode, where ev repeatedly evaluates an expression until it stops changing. To prevent a variable, say X, from being evaluated a way in this mode, simply include X='X as an argument to ev. There are dangers with this command - it is quite possible to generate infinite evaluation loops. For example, ev(X,X=X+1,INFEVAL); will generate such a loop. Here is an example: (need example where this is useful.)

How ev works

The flow of the ev command works like this:

1. The environment is set up by scanning the arguments. During this step, a list is made of 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. Both subscripted variables which do not have associated array functions and non-subscripted variables in the expression exp are replaced by their global values, except for those appearing in the generated list.

2. If any substitutions are indicated, they are carried out.

3. The resulting expression is then re-evaluated, unless one of the arguments was NO-EVAL, and simplified according to the arguments. Note that any function calls in exp will be carried out AFTER the variables in it are evaluated.

4. If one of the arguments was EVAL, the previous two steps are repeated.

4.1.4. Clearing values from the system - the `kill` command

Many times you will define something in Maxima, only to want to remove that definition later in the computation. The way you do this in Maxima is quite simple - using the `kill` command. Here is an example:

```
(C5)  A:7$
(C6)  A;
(D6)
```

$$7$$

```
(C7)  kill(A);
(D7)
```

$$DONE$$

```
(C8)  A;
(D8)
```

$$A$$

`kill` is used in many situations, and has many uses. You will see it appear throughout this manual, in different contexts. There are general arguements you can use, such as `kill(all)`, which will essentially start you out in a new, clean environment. (Add any relevant general kill options here - save kill(rules) for rules section, etc.)

4.2 Common Operators in Maxima

An operator is simply something that signals a specific operation is to be performed. There are many, many possible operators in Maxima. We will address various operators for specific jobs all throughout this manual - this section is not comprehensive.

4.2.1. Assignment Operators

In mathematics, we quite often want to declare functions, assign values to numbers, and do many similarly useful things. Maxima has a variety of operators for this purpose.

: The basic assignment operator. We have already seen this operator in action; it is one of the most common in maxima.

```
(C20)  A:7;
(D20)
```

$$7$$

```
(C21)  A;
(D21)
```

$$7$$

:= This is the operator you would use to define functions. This is a common thing to do in computer algebra, so we will illustrate both how to and how not to do this.

The right way:

```
(C2)  y(x):=x^2;
(D2)
```

$$y(x) := x^2$$

```
(C3)  y(2);
(D3)
```

$$4$$

Several possible wrong ways:

```
(C22)  y:=x^2;
  Improper function definition:
y
 -- an error.  Quitting.  To debug this try DEBUGMODE(TRUE);)
(C23)  y=x^2;
(D23)
```

$$y = x^2$$

```
(C24)  y(2);
(D24)
```

$$y(2)$$

```
(C25)  y(x)=x^2;
(D25)
```

$$y(x) = x^2$$

```
(C26)  y(2);
(D26)
```

$$y(2)$$

```
(C27)  y[x]=x^2;
(D27)
```

$$y_x = x^2$$

```
(C28)  y[2];
(D28)
```

$$y_2$$

```
(C29)  y(x):=x^2;
(D29)
```

$$y(x) := x^2$$

```
(C30)  y(2);
(D30)
```

$$4$$

Look over the above example - it pays to know what doesn't work. If you recognize the error or incorrect result you get, it will make for faster debugging.

:: This operator is related to the : operator, but does not function in quite the same way. This is more what a programmer would refer to as a pointer. The best way to explain is to give you an example of how it behaves:

```
(C9)  A:3$
(C10)  B:5;
(D10)
```

$$5$$

```
(C11)  C:'A;
(D11)
```

$$A$$

```
(C12)  C::B;
(D12)
```

$$5$$

```
(C13)  C;
(D13)
```

$$A$$

```
(C14)  A;
(D14)
```

$$5$$

You see C points to A, and A is thus assigned the value of B.

! This is the factorial operator.

```
(C4)  8!;
(D4)
```

$$40320$$

!! This is the double factorial operator. This is defined in Maxima as the product of all the consecutive odd (or even) integers from 1 (or 2) to the odd (or even) arguement.

```
(C6)  8!!;
(D6)
```

$$384$$

```
(C7)  2*4*6*8;
(D7)
```

$$384$$

sqrt(x) This is your basic square root operator.

```
(C13)  sqrt(x^2);
(D13)
```

$$|x|$$

```
(C14)  sqrt(1/2);
(D14)
```

$$\frac{1}{\sqrt{2}}$$

```
(C15)  sqrt(9);
(D15)
```

$$3$$

Of course, this hardly begins to describe all the operators in the system, but what you see here are some of the more common and useful ones.

CHAPTER 5

Trig through Calculus

This chapter and the next will probably split into many more - Trig, Algebra, Calculus, Programming, etc,etc,etc. I just don't know at this point. These chapters will probably largely be example based. Using things such as ratsimp, trigsimp, etc.

Here we will discuss Maxima's ability to handle integration, differentiation, and other related concepts.

5.1 Trigonometric Functions

These operate more or less as you would expect. The following functions are defined by default:

sin	Sine	asin	Arc Sine
cos	Cosine	acos	Arc Cosine
tan	Tangent	atan	Arc Tangent
csc	Cosecant	acsc	Arc Cosecant
sec	Secant	asec	Arc Secant
cot	Cotangent	acot	Arc Cotangent
sinh	Hyperbolic Sine	asinh	Hyperbolic Arc Sine
cosh	Hyperbolic Cosine	acosh	Hyperbolic Arc Cosine
tanh	Hyperbolic Tangent	atanh	Hyperbolic Arc Tangent
csch	Hyperbolic Cosecant	acsch	Hyperbolic Arc Cosecant
sech	Hyperbolic Secant	asech	Hyperbolic Arc Secant
coth	Hyperbolic Cotangent	acoth	Hyperbolic Arc Cotangent

There are a couple wrinkles worth noting - by default, Maxima will not simplify expressions which numerically are nice fractions of π, so there exists a package which may be loaded to allow this called atrig1. Here is an example:

```
(C1)  acos(1/sqrt(2));
(D1)
```

$$\arccos\left(\frac{1}{\sqrt{2}}\right)$$

```
(C2)  load(atrig1)$
(C3)  acos(1/sqrt(2));
(D3)
```

$$\frac{\pi}{4}$$

Maxima is aware of the Half Angle relations, but by default will not use them. There is a variable which can be set called `halfangles`, and when that is set to true the Half Angle definitions will be used.

```
(C1) sin(a/2);
(D1)
```

$$\sin\left(\frac{a}{2}\right)$$

```
(C2) halfangles:true;
(D2)
```

$$\mathbf{true}$$

```
(C3) sin(a/2);
(D3)
```

$$\frac{\sqrt{1-\cos a}}{\sqrt{2}}$$

You should be aware that when solving expressions involving trig functions, not all solutions will be presented. This is inevitable, since in many cases there are an infinite number - typically one will be displayed. Usually you are warned when this is happens.

```
(C4) solve(sin(x)=%PI/2,x);
SOLVE is using arc-trig functions to get a solution.
Some solutions will be lost.
(D4)
```

$$\left[x = \arcsin\left(\frac{\pi}{2}\right)\right]$$

There are a few global variables you can set which will change how Maxima handles trig expressions:

- TRIGINVERSES
  This can be set to one of three values: ALL, TRUE, or FALSE. The default is ALL
  - ALL When set to ALL, both arctfun(tfun(x)) and fun(arctfun(x)) are evaluated to x.
  - TRUE When set to TRUE, the arctfun(tfun(x)) simplification is turned off.
  - FALSE When set to FALSE, both simplifications are turned off.
- TRIGSIGN
  Can be set to TRUE or FALSE. The default is TRUE. If TRUE, for example, sin(-x) simplifies to -sin(x).

5.2 Differentiation

To differentiate an expression, use the `diff` command. `diff(expr,var)` differentiates an expression with respect to the variable `var`.

```
(C1) sin(x)*cos(x);
(D1)
```

$$\cos x \sin x$$

```
(C2) diff(%,x);
(D2)
```

$$\cos^2 x - \sin^2 x$$

To take a second order derivative, use `diff(expr,var,2)`.

```
(C4) diff(sin(x)*cos(x),x,2);
(D4)
```

$$-4\cos x \sin x$$

Differentiation, unlike integration, can be handled in a fairly general way by computer algebra. As a result, you will be able to take derivatives in most cases. We will show some examples here:

Basic Algebraic Examples:

```
(C1) diff(3*x^5+x^4+7*x^3-x^2+17,x);
(D1)
```

$$15x^4 + 4x^3 + 21x^2 - 2x$$

```
(C2) diff((x^2+1)/(x^2-1),x);
(D2)
```

$$\frac{2x}{x^2-1} - \frac{2x\left(x^2+1\right)}{\left(x^2-1\right)^2}$$

```
(C3) diff((x^2+1)^5*(x^7-5*x-2)^19,x);
(D3)
```

$$10x\left(x^2+1\right)^4\left(x^7-5x-2\right)^{19} + 19\left(x^2+1\right)^5\left(7x^6-5\right)\left(x^7-5x-2\right)^{18}$$

```
(C4) diff(x^(2/3)+x^(5/7),x);
(D4)
```

$$\frac{5}{7x^{\frac{2}{7}}} + \frac{2}{3x^{\frac{1}{3}}}$$

Chain Rule Example:

In order to handle the problem of a function which depends in an unknown way upon some variable, Maxima provides the `depends` command. Using it, you can derive general chain rule formulas. It should be noted these relations are understood only by the diff command - for operations such as integration you must give their dependencies explicitly in the command.

```
(C1) DEPENDS([U],[r,theta],[r,theta],[x,y]);
(D1)
```

$$[U(r,\vartheta), r(x,y), \vartheta(x,y)]$$

```
(C2) diff(U,x)+diff(U,y);
(D2)
```

$$\frac{d}{dy}\vartheta\left(\frac{d}{d\vartheta}U\right) + \frac{d}{dx}\vartheta\left(\frac{d}{d\vartheta}U\right) + \frac{d}{dy}r\left(\frac{d}{dr}U\right) + \frac{d}{dx}r\left(\frac{d}{dr}U\right)$$

If we wish to take derivatives with respect to multiple variables, for example $\frac{d^2}{dxdy}$, the syntax for derivatives is quite general and we can perform the operation as follows:

```
(C7) diff(U,x,1,y,1);
(D7)
```

$$\frac{d}{dx}\vartheta\left(\frac{d}{dy}\vartheta\left(\frac{d^2}{d\vartheta^2}U\right) + \frac{d}{dy}r\left(\frac{d^2}{drd\vartheta}U\right)\right) + \frac{d^2}{dxdy}\vartheta\left(\frac{d}{d\vartheta}U\right) + \frac{d}{dx}r\left(\frac{d}{dy}r\left(\frac{d^2}{dr^2}U\right) + \frac{d}{dy}\vartheta\left(\frac{d^2}{drd\vartheta}U\right)\right) + \frac{d^2}{dxdy}r\left(\frac{d}{dr}U\right)$$

This is the same thing as doing

```
(C6) diff(diff(U,x),y);
(D6)
```

$$\frac{d}{dx}\vartheta\left(\frac{d}{dy}\vartheta\left(\frac{d^2}{d\vartheta^2}U\right) + \frac{d}{dy}r\left(\frac{d^2}{drd\vartheta}U\right)\right) + \frac{d^2}{dxdy}\vartheta\left(\frac{d}{d\vartheta}U\right) + \frac{d}{dx}r\left(\frac{d}{dy}r\left(\frac{d^2}{dr^2}U\right) + \frac{d}{dy}\vartheta\left(\frac{d^2}{drd\vartheta}U\right)\right) + \frac{d^2}{dxdy}r\left(\frac{d}{dr}U\right)$$

Trigonometric Derivatives:

```
(C3) diff(cos(x),x);
(D3)
```

$$-\sin x$$

```
(C4) diff(acos(x),x);
```

(D4)

$$-\frac{1}{\sqrt{1-x^2}}$$

```
(C5)  diff(tan(x),x);
(D5)
```

$$\sec^2 x$$

```
(C6)  diff(atan(x),x);
(D6)
```

$$\frac{1}{x^2+1}$$

```
(C7)  diff(sinh(x),x);
(D7)
```

$$\cosh x$$

```
(C8)  diff(asinh(x),x);
(D8)
```

$$\frac{1}{\sqrt{x^2+1}}$$

5.3 Integration

Unlike differentiation, integration cannot be readily expressed in a general way. Maxima is quite capable when it comes to to such problems, although like all computer algebra systems it has its limits.

(Figure out what and how to discuss here. Will need to discuss the assume options.)

Since integration is such a major feature, we will include here a fairly extensive collection of examples of integrals.

```
(C59)  integrate(x,x);
(D59)
```

$$\frac{x^2}{2}$$

```
(C60)  assume(a>0)$
(C61)  assume(n>0)$
(C62)  integrate(a*x^n,x);
(D62)
```

$$\frac{ax^{n+1}}{n+1}$$

```
(C63)  assume(b>0)$
(C64)  integrate(a*exp(x*b),x);
(D64)
```

$$\frac{ae^{bx}}{b}$$

```
(C65)  assume(c>0)$
(C66)  integrate(a*b^(x*c),x);
(D66)
```

$$\frac{ab^{cx}}{\log bc}$$

```
(C67)  integrate(log(x),x);
(D67)
```

$$x\log x - x$$

```
(C68)  integrate(a/(b^2+x^2),x);
(D68)
```

$$\frac{a\arctan\left(\frac{x}{b}\right)}{b}$$

```
(C69)  assume(m>0)$
(C70)  integrate(x^m*(a+b*x)^5,x);
```

(D70)
$$\frac{b^5x^{m+6}}{m+6}+\frac{5ab^4x^{m+5}}{m+5}+\frac{10a^2b^3x^{m+4}}{m+4}+\frac{10a^3b^2x^{m+3}}{m+3}+\frac{5a^4bx^{m+2}}{m+2}+\frac{a^5x^{m+1}}{m+1}$$

(C71) `integrate(x/(a+b*x)^n,x);`
(D71)
$$-\frac{\left(b^2\,(n-1)\,x^2+abnx+a^2\right)e^{-n\log(bx+a)}}{b^2\,(n^2-3n+2)}$$

(C72) `integrate(x^2/(a+b*x)^n,x);`
(D72)
$$-\frac{\left(b^3\left(n^2-3n+2\right)x^3+ab^2\left(n^2-n\right)x^2+2a^2bnx+2a^3\right)e^{-n\log(bx+a)}}{b^3\,(n^3-6n^2+11n-6)}$$

(C73) `integrate(1/(x^2-c^2)^5,x);`
(D73)
$$-\frac{35\log(x+c)}{256c^9}+\frac{35\log(x-c)}{256c^9}+\frac{105x^7-385c^2x^5+511c^4x^3-279c^6x}{384c^8x^8-1536c^{10}x^6+2304c^{12}x^4-1536c^{14}x^2+384c^{16}}$$

(C74) `integrate(1/(a+b*x^2)^4,x);`
(D74)
$$\frac{5\arctan\left(\frac{\sqrt{b}x}{\sqrt{a}}\right)}{16a^{\frac{7}{2}}\sqrt{b}}+\frac{15b^2x^5+40abx^3+33a^2x}{48a^3b^3x^6+144a^4b^2x^4+144a^5bx^2+48a^6}$$

(C75) `integrate(sqrt(a+b*x)/(x^5),x);`
(D75)
$$-\frac{5b^4\log\left(\frac{2\sqrt{bx+a}-2\sqrt{a}}{2\sqrt{bx+a}+2\sqrt{a}}\right)}{128a^{\frac{7}{2}}}-\frac{15b^4\,(bx+a)^{\frac{7}{2}}-55ab^4\,(bx+a)^{\frac{5}{2}}+73a^2b^4\,(bx+a)^{\frac{3}{2}}+15a^3b^4\sqrt{bx+a}}{192a^3\,(bx+a)^4-768a^4\,(bx+a)^3+1152a^5\,(bx+a)^2-768a^6\,(bx+a)+192a^7}$$

CHAPTER 6

Advanced Mathematics - ODEs and Beyond

6.1 Ordinary Differential Equations

6.1.1. Defining Ordinary Differential Equations

There are three standard ways to represent an ordinary differential equation, such as

$$x^2y' + 3xy = \sin(x)/x,$$

in Maxima. The simplest way is to represent the derivatives by `'diff(y,x)`, `'diff(y,x,2)`, etc. The above ordinary differential equation would then be entered as

```
(C1)  x^2*'diff(y,x) + 3*x*y = sin(x)/x;
(D1)
```

$$x^2\left(\frac{d}{dx}y\right) + 3xy = \frac{\sin x}{x}$$

Note that the derivative `'diff(y,x)` is quoted, to prevent it from being evaluated (to `0`). The second way is to use the `depends` command to tell Maxima that `y` is a functions of `x`, making the quotes unnecessary. The above equation would then be entered as

```
(C2)  depends(y,x);
(D2)
```

$$[y(x)]$$

```
(C3)  x^2*diff(y,x) + 3*x*y = sin(x)/x;
(D3)
```

$$x^2\left(\frac{d}{dx}y\right) + 3xy = \frac{\sin x}{x}$$

The third way would be to write `y(x)` explicitly as a function of `x`. The above equation would then be entered as

```
(C4)  x^2*diff(y(x),x) + 3*x*y(x) = sin(x)/x;
(D4)
```

$$x^2\left(\frac{d}{dx}y(x)\right) + 3xy(x) = \frac{\sin x}{x}$$

Different commands for working with differential equations require different representations of the equations. For the command `ode2` (see subsection 6.1.2), it is often more useful to use one of the first two representations, while for the command `desolve` (see subsection 6.1.3) it is required to use the third representation.

6.1.2. Solving Ordinary Differential Equations: ode2

Using ode2

Maxima can solve first and second order differential equations using the `ode2` command. The command `ode2(`*eqn*`,`*depvar*`,`*indvar*`)` will solve the differential equation given by *eqn*, assuming that *depvar* and *indvar* are the dependent and independent variables, respectively. (If an expression *expr* is given instead of an equation, it is assumed that the expression represents the equation *expr*`=0`.)

```
(C5) ode2(x^2*diff(y,x) + 3*x*y = sin(x)/x, y, x);
(D5)
```

$$y = \frac{\%C - \cos x}{x^3}$$

If `ode2` cannot solve a given equation, it returns the value `FALSE`.

Initial and Boundary Conditions

After a differential equation is solved by `ode2`, initial values or boundary conditions can be given to the solution. The commands for giving the conditions to the solution, however, require that the differential equation **not** be given explicitly as a function of the variable; i.e., `diff(y,x)` would have to be used rather than `diff(y(x),x)` to denote the derivative.

For a first order differential equation, the initial condition can be given using `ic1`. If `ode2` returns the general solution *soln* to a first order differential equation, the command `ic1(`*soln*`, `*indvar*`=`*a*`, `*depvar*`=`*b*`)` will return the particular solution which equals *b* when the variable equals *a*.

```
(C6) soln1:ode2(x^2*diff(y,x) + 3*x*y = sin(x)/x, y, x);
(D6)
```

$$y = \frac{\%C - \cos x}{x^3}$$

```
(C7) ic1(soln1, x=1, y=1);
(D7)
```

$$y = -\frac{\cos x - \cos 1 - 1}{x^3}$$

For a second order differential equation, conditions can be given as initial conditions, using `ic2`, or as boundary conditions, using `bc2`. If `ode2` returns the general solution *soln* to a second order differential equation, the command `ic2(`*soln*`, `*indvar*`=`*a*`, `*depvar*`=`*b*`, diff(`*depvar*`, `*indvar*`)=`*c*`)` will return the particular solution which equals *b* and whose derivative equals *c* when the variable equals *a*.

```
(C8) eqn2: diff(y,x,2) + y = 4*x;
(D8)
```

$$\frac{d^2}{dx^2}y + y = 4x$$

```
(C9) soln2: ode2(eqn2, y, x);
(D9)
```

$$y = \%K1 \sin x + \%K2 \cos x + 4x$$

```
(C10) ic2(soln2, x=0, y=1, diff(y,x)=3);
(D10)
```

$$y = -\sin x + \cos x + 4x$$

Similarly, if `ode2` returns the general solution *soln* to a second order differential equation, the command `bc2(`*soln*`, `*indvar*`=`*a*`, `*depvar*`=`*b*`, `*indvar*`=`*c*`, `*depvar*`=`*d*`)` will return the particular solution which equals *b* when the variable equals *a* and which equals *d* when the variable equals *c*.

```
(C11) bc2(soln2, x=0, y=3, x=2, y=1);
```

(D11)

$$y = -\frac{(3\cos 2+7)\sin x}{\sin 2} + 3\cos x + 4x$$

ode2 Methods

To solve a given differential equation, `ode2` will attempt a series of standard methods for solving differential equations. These methods will be described below, more in-depth discussions of these techniques can be found in any standard introductory text on ordinary differential equations (such as *Elementary Differential Equations and Boundary Value Problems* by Boyce and DiPrima, from which most of these routines were taken).

The first thing `ode2` will do with a differential equation is determine whether it is first order or second order. For first order differential equations, `ode2` will check to see if the equation falls into one of the following categories, in which case the equation will be solved appropriately.

Linear. A first order differential equation is *linear* if it can be written in the form $y' + p(x)y = q(x)$. In this case, the solution is given by $y = (I(x)+c)/\mu(x)$, where $\mu(x)$ is $e^{P(x)}$ for some antiderivative $P(x)$ of $p(x)$, $I(x)$ is an antiderivative of $\mu(x)q(x)$, and c is an arbitrary constant.

```
(C12) linode:diff(y,x) + x*y = x^2;
(D12)
```

$$\frac{d}{dx}y + xy = x^2$$

```
(C13) ode2(linode,y,x);
(D13)
```

$$y = e^{-\frac{x^2}{2}} \left(\frac{\sqrt{2}\sqrt{\pi}i\mathrm{erf}\left(\frac{ix}{\sqrt{2}}\right)}{2} + xe^{\frac{x^2}{2}} + \%C \right)$$

Separable. A first order differential equation is *separable* if it can be put in the form $M(x) = N(y)y'$. In this case, an implicit solution is obtained by integrating both sides of $M(x)dx = N(y)dy$. (It may or may not be possible to solve for y explicitly.)

```
(C14) separableode:(3*x^2+4*x+2)=(2*y-1)*diff(y,x);
(D14)
```

$$3x^2 + 4x + 2 = (2y-1)\left(\frac{d}{dx}y\right)$$

```
(C15) ode2(separableode, y, x);
(D15)
```

$$y^2 - y = x^3 + 2x^2 + 2x + \%C$$

Exact. A first order differential equation is *exact* if it can be put in the form $p(x,y)y' + q(x,y) = 0$, where $p(x,y) = \partial M(x,y)/\partial y$ and $q(x,y) = \partial M(x,y)/\partial x$ for some $M(x,y)$. In this case, the solution will be given implicitly by $M(x,y) = 0$. (It may or may not be possible to solve for y explicitly.)

```
(C16) exactode:x^2*cos(x*y)*diff(y,x) + (sin(x*y) + x*y*cos(x*y))=0;
(D16)
```

$$\sin(xy) + x^2\left(\frac{d}{dx}y\right)\cos(xy) + xy\cos(xy) = 0$$

```
(C17) ode2(exactode,y,x);
(D17)
```

$$x\sin(xy) = \%C$$

If the given differential equation can be put in the form $p(x,y)y' + q(x,y) = 0$ but is not exact, `ode2` checks to see if there is an integrating factor $\mu(x,y)$ which will make $\mu(x,y)p(x,y)y' + \mu(x,y)q(x,y) = 0$ exact, in which case this new equation will be solved as above.

```
(C18) intfactorode:(2*x*y - exp(-2*y))*diff(y,x) + y =0;
(D18)
```

$$\left(2xy - e^{-2y}\right)\left(\frac{d}{dx}y\right) + y = 0$$

```
(C19) ode2(intfactorode,y,x);
(D19)
```

$$xe^{2y} - \log y = \%C$$

Homogeneous. A first order differential equation is *homogeneous* if it can be put in the form $y' = F(y/x)$. In this case, the substitution $v = y/x$ will transform the equation into the separable equation $xv' + v = F(v)$, which can be solved as above.

```
(C20) homode:diff(y,x) = (y/x)^2 + 2*(y/x);
(D20)
```

$$\frac{d}{dx}y = \frac{y^2}{x^2} + \frac{2y}{x}$$

```
(C21) ode2(homode,y,x);
(D21)
```

$$-\frac{xy + x^2}{y} = \%C$$

Bernoulli. The equation $y' + p(x)y = q(x)y^n$, $n \neq 0, 1$, is called *Bernoulli's equation* with index n. The transformation $v = y^{1-n}$ will transform Bernoulli's equation into the linear equation $v' + (1-n)p(x)v = (1-n)q(x)$, which can be solved as above.

```
(C22) berode:diff(y,x) + (2/x)*y = (1/x^2)* y^3;
(D22)
```

$$\frac{d}{dx}y + \frac{2y}{x} = \frac{y^3}{x^2}$$

```
(C23) ode2(berode, y, x);
(D23)
```

$$y = \frac{1}{\sqrt{\frac{2}{5x^5} + \%Cx^2}}$$

General Homogeneous. A first order differential equation is said to be *general homogeneous* of index n if it can be written in the form $y' = (y/x)G(yx^n)$. In this case, a solution is given implicitly by $x = ce^{I(yx^n)}$, where $I(u)$ is an antiderivative of $1/(u(n+G(u)))$ and c is an arbitrary constant. (It may or may not be possible to solve for y explicitly.)

If the differential equation is second order, then `ode2` will determine if the equation is linear or not. In the linear case, when the equation can be written $y'' + p(x)y' + q(x)y = r(x)$, `ode2` will try to solve the equation by first solving the homogeneous part, $y'' + p(x)y' + q(x)y = 0$. The general solution of the homogeneous part will be of the form $y = k_1y_1 + k_2y_2$ for arbitrary constants k_1 and k_2. If $r(x) \neq 0$, `ode2` will then use variation of parameters to find a particular solution y_p of the original equation. The general solution of the full equation will then be $y = k_1y_1 + k_2y_2 + y_p$. To solve the homogeneous part, `ode2` will check to see if the equation falls into one of the following categories, in which case the equation will be solved appropriately.

Constant Coefficients. If the differential equation has constant coefficients, and so is of the form $y'' + ay' + by = 0$, then the solution is $y = k_1e^{r_1x} + k_2e^{r_2x}$, where r_1 and r_2 are the solutions of $r^2 + ar + b = 0$. In case $r^2 + ar + b = 0$ has a double root, the solution of the differential equation is $y = k_1e^{rx} + k_2xe^{rx}$. In some cases where the equation doesn't have constant coefficients, ode2 will attempt to use a simple transformation to reduce it to constant coefficents.

```
(C24) ccode1: diff(y,x,2) - 3*diff(y,x) + 2*y=0;
(D24)
```

$$\frac{d^2}{dx^2}y - 3\left(\frac{d}{dx}y\right) + 2y = 0$$

```
(C25) ccode2: diff(y,x,2) - 4*diff(y,x) + 4*y=0;
(D25)
```

$$\frac{d^2}{dx^2}y - 4\left(\frac{d}{dx}y\right) + 4y = 0$$

```
(C26) ode2(ccode1, y, x);
(D26)
```

$$y = \%\mathrm{K}1e^{2x} + \%\mathrm{K}2e^{x}$$

```
(C27) ode2(ccode2, y, x);
(D27)
```

$$y = (\%\mathrm{K}2x + \%\mathrm{K}1)\, e^{2x}$$

Exact. A second order differential equation is *exact* if it can be written in the form $[f(x)y']' + [g(x)y]' = 0$. Integrating this equation will reduce it to a first order differential equation, which can be solved as above.

```
(C28) exactode2: x^2*diff(y,x,2) + x*diff(y,x) - y =0;
(D28)
```

$$x^2\left(\frac{d^2}{dx^2}y\right) + x\left(\frac{d}{dx}y\right) - y = 0$$

```
(C29) ode2(exactode2, y,x);
(D29)
```

$$y = \%\mathrm{K}2x - \frac{\%\mathrm{K}1}{2x}$$

Euler. The equation $x^2y'' + axy' + by = 0$ is *Euler's equation*. The solution is given by $y = k_1x^{r_1} + k_2x^{r_2}$, where r_1 and r_2 are solutions of $r(r-1) + ar + b = 0$. In case $r(r-1) + ar + b = 0$ has a double root, the solution is given by $y = k_1x^r + k_2\ln(x)x^r$.

```
(C30) eulerode1: x^2*diff(y,x,2) + 4*x*diff(y,x) + 2*y = 0;
(D30)
```

$$x^2\left(\frac{d^2}{dx^2}y\right) + 4x\left(\frac{d}{dx}y\right) + 2y = 0$$

```
(C31) eulerode2: x^2*diff(y,x,2) + 5*x*diff(y,x) + 4*y = 0;
(D31)
```

$$x^2\left(\frac{d^2}{dx^2}y\right) + 5x\left(\frac{d}{dx}y\right) + 4y = 0$$

```
(C32) ode2(eulerode1, y, x);
(D32)
```

$$y = \frac{\%\mathrm{K}1}{x} + \frac{\%\mathrm{K}2}{x^2}$$

```
(C33) ode2(eulerode2, y, x);
(D33)
```

$$y = \frac{\%\mathrm{K}2\log x + \%\mathrm{K}1}{x^2}$$

Bessel's Equation. The equation $x^2y'' + xy' + (x^2 - \nu^2)y = 0$ is called *Bessel's equation* of order ν. For $\nu = 1/2$, the solution is $y = k_1 \sin(x)/\sqrt{x} + k_2 \cos(x)/\sqrt{x}$; for integer ν, the answer will be $y = k_1 Y_\nu(x) + k_2 J_\nu(x)$, where J_ν and Y_ν are the Bessel functions of the first and second kind.

```
(C34) besselode1:x^2*diff(y,x,2) + x*diff(y,x) + (x^2 - 1/4)*y=0;
(D34)
```

$$x^2 \left(\frac{d^2}{dx^2} y\right) + x \left(\frac{d}{dx} y\right) + \left(x^2 - \frac{1}{4}\right) y = 0$$

```
(C35) besselode2:x^2*diff(y,x,2) + x*diff(y,x) + (x^2 - 4)*y=0;
(D35)
```

$$x^2 \left(\frac{d^2}{dx^2} y\right) + x \left(\frac{d}{dx} y\right) + \left(x^2 - 4\right) y = 0$$

```
(C36) ode2(besselode1, y, x);
(D36)
```

$$y = \frac{\%K1 \sin x + \%K2 \cos x}{\sqrt{x}}$$

```
(C37) ode2(besselode2, y, x);
(D37)
```

$$y = \%K2 \% Y_2(x) + \%K1 \% J_2(x)$$

`ode2` can also handle translates of Bessel's equation; i.e., differential equations of the form $(x-a)^2y'' + (x-a)y' + ((x-a)^2 - \nu^2)y = 0$

```
(C38) besselode3:(x-1)^2*diff(y,x,2) + (x-1)*diff(y,x) + ((x-1)^2 - 4)*y=0;
(D38)
```

$$(x-1)^2 \left(\frac{d^2}{dx^2} y\right) + (x-1) \left(\frac{d}{dx} y\right) + \left((x-1)^2 - 4\right) y = 0$$

```
(C39) ode2(besselode3, y, x);
(D39)
```

$$y = \%K2 \% Y_2(x-1) + \%K1 \% J_2(x-1)$$

If `ode2` successfully solves the homogeneous part of an inhomogeneous equation, it then tries to find a particular solution using variation of parameters.

Variation of Parameters. If $y = k_1y_1 + k_2y_2$ is the general solution of $y'' + p(x)y' + q(x)y = 0$, then a particular solution of $y'' + p(x)y' + q(x)y = r(x)$ can be found by replacing the arbitrary constants k_1 and k_2 with arbitrary functions u_1 and u_2, and looking for a solution of the form $y = u_1y_1 + u_2y_2$. One such solution is given if u_1 and u_2 are antiderivatives of $-y_2r/(y_1y_2' - y_2y_1')$ and $y_1r/(y_1y_2' - y_2y_1')$, respectively.

```
(C40) varparode:diff(y,x,2) + 2*diff(y,x) + y = exp(x);
(D40)
```

$$\frac{d^2}{dx^2} y + 2 \left(\frac{d}{dx} y\right) + y = e^x$$

```
(C41) ode2(varparode,y,x);
(D41)
```

$$y = \frac{e^x}{4} + (\%K2 x + \%K1)\, e^{-x}$$

In case the second order differential equation is not linear, `ode2` will check to see if either the dependent variable or the independent variable is missing, in which case one of the following two methods will be used.

Missing dependent variable. If the undifferentiated dependent variable y is not present in a second order differential equation, the substitution $v = y'$, $v' = y''$ will reduce the differential equation to first order. This can be solved as above. Once v is obtained, y can be obtained by integrating v.

```
(C42) noyode:x*diff(y,x,2) + (diff(y,x))^2=0;
(D42)
```

$$x\left(\frac{d^2}{dx^2}y\right) + \%\mathrm{DERIVATIVE}^2\left((y,x,1)\right) = 0$$

```
(C43) ode2(noyode,y,x);
(D43)
```

$$y = \int \frac{1}{\log x + \%\mathrm{K1}}\, dx + \%\mathrm{K2}$$

Missing independent variable. If the independent variable x is not present in a second order differential equation, then making the dependent variable y a temporary independent variable and using the substitution $v = y'$, $v' = y''$, the equation can again be reduced to first order. Since the derivatives are taken with respect to x, however, the new equation will involve three variables. This can be resolved by noting that $v' = \mathrm{d}v/\mathrm{d}x = (\mathrm{d}v/\mathrm{d}y)(\mathrm{d}y/\mathrm{d}x) = (\mathrm{d}v/\mathrm{d}y)v$, and so $v' = \mathrm{d}v/\mathrm{d}x$ can be replaced by $v\mathrm{d}v/textrmdy$. This will result in a first order differential equation with dependent variable v and independent variable y. This can be solved as above. Once v is obtained (in terms of y), y is a solution of the differential equation $y' = v(y)$, which can be solved as above.

```
(C44) noxode: y*diff(y,x,2) + (diff(y,x))^2 = 0;
(D44)
```

$$y\left(\frac{d^2}{dx^2}y\right) + \%\mathrm{DERIVATIVE}^2\left((y,x,1)\right) = 0$$

```
(C45) ode2(noxode,y,x);
(D45)
```

$$\frac{y^2}{2\%\mathrm{K1}} = x + \%\mathrm{K2}$$

Information on the Method

The `ode2` routine will store information about the method used to solve the differential equations in various variables. The variable `method` will keep track of the method used to solve the differential equations.

```
(C46) ode2(separableode, y, x);
(D46)
```

$$y^2 - y = x^3 + 2x^2 + 2x + \%\mathrm{C}$$

```
(C47) method;
(D47)
```

SEPARABLE

In the case where an integrating factor was used to make a differential equation exact, the variable `intfactor` will be the integrating factor used.

```
(C48) ode2(intfactorode, y, x);
(D48)
```

$$xe^{2y} - \log y = \%\mathrm{C}$$

```
(C49) method;
(D49)
```

EXACT

```
(C50) intfactor;
(D50)
```

$$\frac{e^{2y}}{y}$$

When Bernoulli's equation or a generalized homogeneous equation is solved, the variable `odeindex` will be the index of the equation.

```
(C51) ode2(berode, y, x);
(D51)
```

$$y = \frac{1}{\sqrt{\frac{2}{5x^5} + \%Cx^2}}$$

```
(C52) method;
(D52)
```

BERNOULLI

```
(C53) odeindex;
(D53)
```

3

When an inhomogeneous second degree linear differential equation is solved, the variable `yp` will be the particular solution arrived at by variation of parameters.

```
(C54) ode2(varparode, y, x);
(D54)
```

$$y = \frac{e^x}{4} + (\%K2x + \%K1)\,e^{-x}$$

```
(C55) method;
(D55)
```

VARIATIONOFPARAMETERS

```
(C56) yp;
(D56)
```

$$\frac{e^x}{4}$$

6.1.3. *Solving Ordinary Differential Equations:* `desolve`

Using `desolve`

Maxima can solve systems of linear ordinary differential equation with constant coefficients using the `desolve` command. The differential equations must be given using functional notation, rather than with dependent variables; i.e., `diff(y(x),x)` would have to be used rather than `diff(y,x)` to denote the derivative. The command `desolve(`*delist*`,` *fnlist*`)`, will solve the system of differential equations given by the list *delist*, where *fnlist* is a list of the functions to be solved for.

```
(C57) de1:diff(f(x),x)=diff(g(x),x)+sin(x);
(D57)
```

$$\frac{d}{dx}f(x) = \frac{d}{dx}g(x) + \sin x$$

```
(C58) de2:diff(g(x),x,2)=diff(f(x),x) - cos(x);
(D58)
```

$$\frac{d^2}{dx^2}g(x) = \frac{d}{dx}f(x) - \cos x$$

```
(C59) desolve([de1,de2],[f(x),g(x)]);
(D59)
```

$$\left[f(x) = e^x\left(\left.\frac{d}{dx}g(x)\right|_{x=0}\right) - \left.\frac{d}{dx}g(x)\right|_{x=0} + f(0), g(x) = e^x\left(\left.\frac{d}{dx}g(x)\right|_{x=0}\right) - \left.\frac{d}{dx}g(x)\right|_{x=0} + \cos x + g(0) - 1\right]$$

A single differential equation of a single unknown function can also be solved by `desolve`; in this case, it isn't necessary to enter them as lists.

```
(C60) de3:'diff(f(x),x,2)+ f(x) = 2*x;
(D60)
```

$$\frac{d^2}{dx^2} f(x) + f(x) = 2x$$

```
(C61) desolve(de3, f(x));
(D61)
```

$$f(x) = \sin x \left(\left. \frac{d}{dx} f(x) \right|_{x=0} - 2 \right) + f(0) \cos x + 2x$$

Initial Conditions

Initial conditions can be specified for solutions given by `desolve` using the `atvalue` command. The conditions, however, can only be given at `0`, and must be given before the equations are solved.

```
(C62) atvalue(f(x),x=0,1);
(D62)
```

$$1$$

```
(C63) atvalue(g(x),x=0,2);
(D63)
```

$$2$$

```
(C64) atvalue(diff(g(x),x),x=0,3);
(D64)
```

$$3$$

```
(C65) desolve([de1,de2],[f(x),g(x)]);
(D65)
```

$$[f(x) = 3e^x - 2, g(x) = \cos x + 3e^x - 2]$$

`desolve` Method

The `desolve` routine uses the LaPlace transform to solve the systems of differential equations. If $f(t)$ is defined for all $t \geq 0$, then the LaPlace transform of f is given by $F(s) = \mathcal{L}\{f(t)\} = \int_0^\infty e^{-st} f(t) \mathrm{d}t$. The LaPlace transform has the useful property that a derivative is transformed into multiplication by the variable; if $\mathcal{L}\{f(t)\} = F(s)$, then $\mathcal{L}\{f'(t)\} = sF(s) - f(0)$. The LaPlace transform can thus transform a system of linear differential equations into a system of ordinary equations. (Note, however, that the LaPlace transform will transform multiplication by a variable into differentiation; if $\mathcal{L}\{f(t)\} = F(s)$, then $\mathcal{L}\{tf(t)\} = -F'(s)$. The original differential equations need to have constant coefficients to prevent this.) If this new system can be solved, the LaPlace tranform can be inverted to give solutions of the original system of differential equations.

CHAPTER 7

Matrix Operations and Vectors

Maxima has many matrix capabilities. Unfortunately it's vector handling at the time of this writing is not terribly robust, and the reader is cautioned to use it with care.

CHAPTER 8

Introduction to Maxima's Programming Language

This chapter assumes that readers are familiar with the basic ideas of algebraic manipulation from Chapter **??**, and know at least one programming language, and wish to use Maxima for more ambitious tasks than can be handled in a few sequential commands. If you are familiar with Pascal or Algol 60, you will probably find this adequate as a programming background. Familiarity with Fortran or Basic is less useful.

Maxima's user-programming language[1] is designed to allow you to define program modules or *functions* for algebraic manipulation. Each module uses zero or more arguments, and returns an algebraic expression. Since numbers are special cases of algebraic expressions, Maxima's user-language can be used as a numeric language too. Because the language is implemented as an interpreter it is usually more general than compiler-based languages, and also tends to be rather slow in tight inner loops of simple operations, by comparison. It has novel linguistic features, some of which are illustrated below.

8.1 Some Examples

`f1` and `f2` defined below, are versions of the *factorial* function. Observe the punctuation carefully. Assignment is `:`, function definition is `:=`, statements are separated from one another by `,` (not *terminated* by commas). The label *loop* is set off by a comma as though it were a statement too. We have added indentation to make the programs conform to what you might expect, but extra spaces and tabs are optional. There is a conditional execution statement (the `if`) which has an optional `else` clause.

```
(C1) f1(x):=if x<1 then 1 else x*f1(x-1)$
(C2) f1(5);
(D2)                                120
(C3) f2(x) := block ([temp],
                temp:1,
                while x > 1 do (
                        temp:x*temp,
                        x:x-1),   /*end while*/
                temp)$
(C4) f2(5);
(D4)                                120
```

Observe the simplicity of using Maxima compared to, say, Pascal. There is no need to write a *driver* or main program with input or output commands. The input is provided through function application on an argument, and the output is the displayed value.

[1]The system-programming language used for implementing Maxima namely LISP, is quite different. If you wish to see how Maxima was constructed, you need to know LISP to understand the source code.

Every command or function in Maxima has a value, and may, in addition, have some side-effects, such as the setting of variables, or the printing of messages.

The `block` construction illustrated above is analogous to a procedure declaration. The first part of it is a list of local variables, and following that, expressions which are evaluated in order. Certain expressions or commands make sense only within a block, not at Maxima's command level: these are labels, `return` s and `go` s. The semantics of each of these commands conforms to the usual intuitive meaning. If the last statement in a `block` does not cause a transfer of control, and *execution falls through the bottom*, the value returned from the `block` is the value of the last expression evaluated.

8.2 Unconventional Conditionals

The next example shows a function with a side-effect, but the major point is to illustrate some subtleties which you may not have thought about in conditional statements (`if-then-else`).

If you were asked, *Is A greater than B*, it would seem you could respond either *yes* or *no*. In your conventional programming language, certainly, that would be a reasonable assumption. But really, wouldn't it be appropriate in some circumstances for you to answer, *How should I know?*?

That option is preserved in Maxima. If the flag `prederror` is set to `true`, the default, and if Maxima is unable to evaluate a predicate, it signals an error, and unless directed otherwise, returns control to the top-level Maxima command monitor. However, if the `prederror` flag is `false`, execution continues to the next statement, ignoring both `then` and `else` clauses!

This is illustrated below:

```
(C5) test(x,y):=block([],
        if x > y then print(x, "is greater than", y)
                 else print(x, "is not greater than", y),
        return(alldone))$
(C6) test(4,3);
4 is greater than 3
(D6)                            alldone
(C7) test(3,4);
3 is not greater than 4
(D7)                            alldone
(C8) test(y^2+1,-y);
MACSYMA was unable to evaluate the predicate:
ERREXP1
#0: test(x=y^2+1,y=-y)
 -- an error.  Quitting.  To debug this try DEBUGMODE(TRUE);)
(C9) prederror:false$
(C10) test(y^2+1,-y);
(D10)                           alldone
```

Note that no message was printed for line `d10,` but the return value, *alldone* was displayed.

8.3 Assumptions

What do YOU think? Is $y^2+1 > -y$? For this question to make sense, both sides of the inequality must be in the same ordered domain. We do not know, offhand, whether y can assume values which are matrices, complex numbers, sets, or even programs!

If y were known to be real, or more specifically, positive real, a program could try some deduction. Maxima has some features of this nature, as illustrated below.

```
(C11) assume(y>0);
(D11)                           [y > 0]
(C12) test(y^2+1,-y);
```

```
 2
y  + 1 is greater than - y
(D12)                            alldone
```

If we wish Maxima to forget that assumption,

```
(C13) forget(y>0);
(D13)                            [y > 0]
```

8.4 Arbitrary Numbers of Parameters

Ambitious packages of programs have been written by many Maxima users. Sometimes the requirement that a command has a fixed number of arguments causes discomfort[2]. It is possible to write a Maxima program which counts the number of arguments it is given, sets default values for others, and does any number of clever things. A simple example is shown below. Note the way the *left-hand-side* of the *:=* is set up.

```
(C14) prog3([l]) := block( [],
              print ("l is bound to", l, "and l[1] is" ,l[1]),
              return(length(l)))$
(C15) prog3(a,b,c,d);
l is bound to [a, b, c, d] and l[1] is a
(D15)                               4
(C16) prog3(a,b);
l is bound to [a, b] and l[1] is a
(D16)                               2
```

8.5 Arrays

Arrays are a useful data structure, and are provided in most programming languages. Maxima provides arrays, but does not require that they be declared, or that they have numeric (integer) index-sets. Rather than writing a program to fill up an array and then iterating through all elements, sometimes it is easier to describe a program to generate elements as they are called for. Such array-associated functions are often quite convenient. The usual way you set them up is to provide specific values for certain index values, and then let others be assigned as needed. Note carefully the use of `:=` and `:`.

```
(C17) a[4]:4*u;
(D17)                              4 u
(C18) a[22/7]:%pi;
(D18)                              %PI
(C19) a[x]:mystery;
(D19)                            mystery
(C20) a[h]:=cos(h);
(D20)                         a  := COS(h)
                               h
(C21) a[3];
(D21)                            COS(3)
(C22) a[x+1];
(D22)                          COS(x + 1)
```

You might wonder what the value of *a* is, after all this. Most disappointing:

```
(C23) a;
(D23)                               a
```

[2]The language Pascal does not allow you to define a function `f` which can be used with a variable number of actual parameters, although the Pascal design includes built-in procedures with variable numbers of arguments (`write` for example).

The information that you are after is available this way:

```
(C24) arrayinfo(a);
                                        22
(D24)              [HASHED, 1, [3], [--], [4], [x], [x + 1]]
                                        7
```

The list of information supplied indicates several aspects of the array. It is *hashed*: uses the data-structure of a hash-table for storage (this is a common encoding trick discussed in data structure texts). The number of subscripts is 1. The specific indexes for which it has recorded values are listed. The array-associated function defined on line `c20` can be displayed by `dispfun`.

8.6 Iteration

`For` loops provide the major iteration facility in Maxima. Three examples which illustrate variants of this are the factorial functions below:

```
(C25) f3(n) := block([temp],
                temp:1,
                for i:1 thru n do temp : temp*i,
                return(temp))$
(C26) f4(n) := block([temp],
                temp:1,
                for i:n step -1 thru 1 do temp : temp*i,
                return(temp))$
(C27) f5(n) := block([temp],
                temp:1,
                for i:n unless i <= 1 do (temp : temp*i , i:i-2),
                return(temp))$
```

Decrementing *i* by 2 in the previous program was needed because the default step size of 1 is added to *i* each time through. `F5` is certainly a perverse program. Incidentally, `return` s from within a `for` exit from the loop, and not from the enclosing `block.` It is important to note that one can group a collection of statement to be *done* together with parentheses, as illustrated in `f5.`

8.7 Serious Business

Most serious users of Maxima find that they are repeatedly using the same programs, and need to save them for another day. Some users also find they rarely get the programs or the data quite right the first time, and would rather type these things in to a text editor, and have Maxima gobble the text up from a file rather than the keyboard. Publication quality programs require comments and other features you are unlikely to want to type into Maxima interactively.

Some people perfect a function or get an algebraic expression correct by typing definitions and commands into a file, say, `newstuff`, using an editor such as `vi` or `emacs` and then within Maxima type `batch(newstuff);`. If your filename has funny characters like periods or slashes, you must use quotes. For example, `batch("/usr/local/maxima/demo.begin")`.

Maxima then reads the statements the file, assigning labels etc., and if there are syntax or other errors, prompts for help from the keyboard. If you want it to continue on to the next line after an error, type `batcon(true);`.

Reading very large text files of programs and data can be slow using `batch`, and if you are not changing the text, you might prefer saving your environment in another way. You can use `save(savedstuff,all);` to save every named or labelled object in a Maxima system, on the file `savedstuff` in your working directory. (You better have write-permission or you'll get an error message.) Another time you can start up from where you left off by typing `loadfile(savedstuff);` into a fresh Maxima. You can load several saved files into a single system. Naturally if the files contain items with identical names, there is a potential for conflict. These will be resolved in favor of the last item read in.

If you want to save only some material you have produced, say only the functions defined and the values of variables x, y, and z, you can type

```
save(savfunxyz,functions,x,y,z);
```

A neat way to save a useful section of your environment is to (carefully) use `kill` to remove useless items first, and then save `all` that is left. Doing `kill(labels,...)` after first making sure that any useful result also has a name other than a c or d-label is sometimes a good start. You generally should not save every computation, since disk space is not infinite.

8.8 Hardcopy

If you print out the files produced by `save` to show to your colleagues who are as yet unconvinced of the merits of Maxima you will be disappointed. While such files are *ASCII* character text, they do not make easy reading for persons uninitiated in various arcane matters.

What you might want to do, then, is store human-readable versions of your output on a file. This is especially useful if you have a display terminal, or a slow hardcopy printer. Maxima provides a way of opening a file for echoing both input and output of the system. The command `writefile(fn);` where fn is a filename, starts the echoing, and `closefile();` stops the echoing. If you want the output to be human-readable after being run through a typesetting program (TeX, for example), you can experiment with the output produced by setting `typeset : true`. (Suggestion: `gcprint : false` is a good idea.)

8.9 Return to Arrays and Functions

Maxima provides a fascinating trick: arrays of functions. Imagine an array, each of whose elements is a function. For example, Legendre polynomials have an index, and an argument. Thus we can refer to $P_4(z^2)$. Just as arrays can have an associated function, function-arrays can have such an associated function.

Because we like to show off occasionally, and typesetting can be done by the Maxima system, we illustrate this feature here. This was produced by saving output in a file and running it through TeX.

```
(C31) p[n](x):=ratsimp(1/(2^n*n!)*diff((x^2-1)^n,x,n));
(D31)
```

$$p_n(x) := \mathrm{RATSIMP}\left(\frac{1}{2^n n!}\mathrm{DIFF}\left(\left(x^2-1\right)^n, x, n\right)\right)$$

```
(C32) p[4];
(D32)
```

$$\mathrm{LAMBDA}\left([x], \frac{35x^4-30x^2+3}{8}\right)$$

```
(C33) p[4](y+1);
(D33)
```

$$\frac{35\,(y+1)^4-30\,(y+1)^2+3}{8}$$

The *lambda* (λ) notation for a raw function of one variable appears on line `c29`. This notation comes from the λ-calculus used to describe the programming language LISP, and illustrates the fact that Maxima can talk about such structures. You can generally ignore this fact, however.

8.10 More Useful Examples

As has been indicated earlier, procedures in Maxima can be used for almost any classical programming purposes (e.g. numerical techniques, combinatorial search). We have already indicated some differences from a purely numerical language, such as FORTRAN. We have seen that in Maxima there are no required type declarations, and floating-point numbers, while *contagious* in the sense that mixed-mode (floating-point + integer) is converted to floating-point, do not necessarily arise from certain calculations. For example, `sin(3)` normally results in `sin(3)` in Maxima rather than its floating-point equivalent. `sin(3.0)` will, however, return a floating-point number. The numerical subroutine below, a Newton's method zero-finder sets the flag `numer` to `true` to force Maxima to convert most expressions free of indeterminates to numbers.

This program uses Newton's method to find a zero of the expression `exp` which depends on the variable `var.` The iteration starts with `var=x0` and terminates when the expression, evaluated at the trial-point, has absolute value less than `eps.` The derivative of `exp` is computed algebraically, and its value is computed at various points as the search is being conducted:

```
(C1) newton(exp,var,x0,eps):=                   /* 1 */
               block([xn,s,numer],              /* 2 */
               numer:true,                      /* 3 */
               s:diff(exp,var),                 /* 4 */
               xn:x0,                           /* 5 */
          while abs(subst(xn,var,exp)) >  eps do
               xn:xn-subst(xn,var,exp)/subst(xn,var,s),
          return(xn) )                          /* 8 */$
```

This procedure for Newton's method uses an explicit expression for the first argument `exp` (e.g. `sin(x)*erf(2*x)-%e^x`). It is not a function name, e.g. `f` as we used before. The use of such an expression is straightforward and probably the best way for a beginner. The resulting program is somewhat verbose, because, as illustrated in lines 6 and 7 above, it is necessary to substitute values for variables in the expression and its derivative, `s`, to get numbers. Note the setting of `numer` on line 3 to assure that the `if` statement on line 6 would get numerical values for its test. The rest of the procedure is the classical Newton iteration. The advantage of this procedure over a purely numerical one is that it takes advantage of the ability to compute the derivative of `exp` algebraically and automatically, once, before evaluating it at any point.

Another item to observe here is the use of comments in the text of programs which are `batch` ed in to the system. `/* This is a comment */ .`

There are often many different ways of expressing the same computation, and some of them may be substantially more convenient or efficient than others. While it may not make much of a difference in efficiency in this case, the following revision of the `newton` procedure illustrates some techniques you may find useful in Maxima.

```
(C2) newton(exp,var,x0,eps):=                   /* 1 */
block([ xn:x0, s:diff(exp,var), numer:true],    /* 2 */
        define(f(var), exp),                    /* 3 */
        define(fprime(var),s),                  /* 4 */
  loop, if abs(f(xn)) < eps then return(xn),    /* 5 */
        xn: xn - f(xn)/fprime(xn),              /* 6 */
        go (loop) )                             /* 7 */ $
```

Observe the list of local names at the beginning of the `block,` which are initialized at the same time they are declared on line 2. Lines 3 and 4 are interesting because they define two new functions, `f` and `fprime` each to have as their bodies, the values of `exp` and `s`. The Newton iteration is more easily observed in functional rather than *substitution* notation. An even smaller version of `newton` could be defined by using a single function for `exp/diff(exp,var).`

Let us try this last function:

```
(C3) h:expand((x-1)*(x-3)*(x-5));
                              3      2
(D3)                         x  - 9 x  + 23 x - 15
(C4) newton(h,x,3.5,1.0e-10);
(D4)                           2.999999999994028
```

You might wonder how many iterations that took. One way is to use the very convenient debugging feature provided by Maxima which allows you to watch the assignment of values to variables. You set the variable `setcheck` to a list of the variables you wish to watch. (or `all` to watch all the variables.)

```
(C5) setcheck:[xn]$
(C6) newton(h,x,3.5,1.0e-10);
xn set to 3.5
```

```
xn set to 2.923076923076923
xn set to 3.000228597554008
xn set to 2.999999999994028
(D6)                           2.999999999994028
```

(That message `[*flonum: ...]` is a message from the garbage collector, mentioned in Section **??**.)

That tells us that only three iterations were needed in this case.

We claimed that two functions were defined in running this program. To display a function using the same syntax that you would use to type it in, you use the `grind`[3] command.

```
(C7) grind(fprime);
Error: cursorpos doesn't know position
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by MACSYMA-TOP-LEVEL.
Broken at ERROR.  Type :H for Help.
```

8.11 Part Hacking

An important tool for applications programs in Maxima the ability to extract and test parts of expressions. These are used for the definition or extension of algorithms which do various conditional manipulation of formulas. For example, you can write a symbolic differentiation algorithm for expressions by applying the rules below:

$$\frac{d}{dx}\,x \;=\; 1 \qquad \frac{d}{dx}\,y \;=\; 0 \;\; (y \neq x) \tag{1}$$

$$\frac{d}{dx}\,(u+v) \;=\; \frac{d}{dx}\,u \;+\; \frac{d}{dx}\,v \tag{2}$$

$$\frac{d}{dx}\,(u\cdot v) \;=\; v\frac{d}{dx}\,u \;+\; u\frac{d}{dx}\,v \tag{3}$$

The technique we shall consider for implementing a version of the differentiation program is to take the expression apart using the `part` command, and use the rules above to guide the manipulation.

First, we check to see if the expression is an atom (e.g. number, variable). Thus we begin our differentiation program as follows:

```
newdiff(expr,var):=
        if atom(expr)=true then
                (if expr=var then 1
                             else 0)
```

This fragment implements both parts of rule 1. If the `if` statement falls through to the `else` clause, then we have a composite expression. We then check what its leading operator is by selecting its zeroth *part* via `part(expr,0)`. Based on its value we apply the appropriate rule.

```
        else if part(expr,0)=}+} then
             newdiff(part(expr,1),var)
                    + newdiff(part(expr,2),var)
        else if part(expr,0)=}*} then
             part(expr,2)*newdiff(part(expr,1),var)
                    + part(expr,1)*newdiff(part(expr,2),var)
        else 'newdiff(expr,var)$
```

[3]When computers were slow and programs were large, reformatting programs took a long time. The name of a program at MIT to *grind out* LISP function definitions was `grindef`. The name was carried over to Maxima. Now, most LISP systems call this reformatting *prettyprinting*.

Note the last clause which returns a 'newdiff form, that is, quoted, as a result when the program doesn't know how to handle the leading operator. With some thought, you should now be able to extend newdiff to accommodate expressions including `sin` and `cos` as well as sums and products with more than two terms. (The Maxima language does not at the moment have a `case` statement, which would make this particular program look better.) Is this stiil tru ????

8.12 User Representation of Data

There is no *record* or *structure* data-type *constructor* in Maxima but there is a way of doing something similar. If you have used a language like Pascal which has such a feature, you may appreciate its usefulness. Modern dialects of LISP use such structures, but when Maxima was first designed and initially implemented (in 1967), this was not in widespread use. It also influenced the user-level programming language.

It is natural to have to deal with collections of information in writing programs. For example, you might wish to provide as input or output data, a pair of expressions representing a lower and an upper bound on a formula. One way of handling this is by making up a new *function* name, say `bounds` and using it as though it were a record designator or constructor, but never associating it with an algorithm. Thus `bounds(1,x)` could be an expression representing the ordered pair $< 1, x >$. Another example would be the use of `integer_vector(1,3,5,7)` to designate a sequence (presumably of arbitrary length, in general), of integers. There is a built-in form in Maxima namely a *list*, which can be used to deal with such collections. It is part of the LISP heritage indicated earlier that there was initially only one type of structure: lists of any length or element type in Maxima. A list, when printed, looks like square-brackets enclosing a sequence of items. Thus you could express the bounds above as [`1,x`]. However, if you used lists for all collections, you would not know, on the face of it, whether [1,2] was an instance of `bounds` or `integer_vector`. Maxima allows you to designate functions you intend to treat as lists, although you can use a different *header* like `bounds` with each different type. Maxima makes available certain built-in functions which can then be used on such list-like constructions such as `integer_vector`. These are declared by, for example, `declare(integer_vector,list)`. The built-in operations include `cons, append, member, endcons, map, rest.`
`list2:cons(element,list1)` returns a (new) `list2` which has the given element inserted at the beginning of `list1; List1` and `list2` have a common *shared* tail.
`list3:append(list1,list2)` returns a (new) `list3` which is a combination of the two lists, sharing a common tail with list2. `Member(element,list)` returns true or false depending on a test for membership.
`Endcons(element,list)` returns an (unshared) list where the given element has been attached to the end of given list
`Map(fn,list)` returns a new list where each element has had the function fn applied to it in turn.
`Rest(list,n)` returns the part of the list beginning n items from the front.

These functions are parts of the fundamental repertoire of the programming language LISP.

The use of list-like objects should be considered whenever you are dealing with a collection of elements: a set of coordinates, a series of coefficients, a system of equations, a set of solutions, etc.

Independent of the *whole list* operations above, Maxima has some selection and alteration operations which are available on the same collections of data by the use of a numeric index. If you wish to use these indexing facilities, as we will illustrate for the notion of `complex` below, you `declare(complex,list)`. Then, if you define a complex number by `x:complex(3,4)` meaning that `x` has real part 3, and imaginary part 4, the notation `x[1]:10;` is supported, and changes the value of `x` to `complex(10,4)`. The declaration explains to the Maxima system that the data structure for `complex` will be implemented (in effect) as a list of items, and should be decomposable using the semantics of the Maxima list-handling commands. In fact, both selection and alteration is supported, and if you set the notation up by `(real:1,imag:2)$` you can use the following command: `x[imag]:-x[imag]` to change `x` to its complex conjugate.

An important caution must be observed. When Maxima deals with compound structures, they are usually not recopied, and if there are two names for the same object and the object is changed, then both names refer to the changed object. If `x` and `y` refer to the same `complex` number, then changes to `x[real]` are also made to the corresponding component of `y`. If these items are to be kept separate, `x:copylist(y);` will give `x` a different representation, but whose value is the same as `y` 's. You can then change their components separately.

Maxima's built-in lists mentioned earlier, which use square-brackets, can be altered and selected by indexing.

There are two other compound structures in Maxima which we mention here, which may be useful for collections of data: arrays and matrices. The matrix form is useful for two-dimensional rectangular tables of data. The matrix data format is supported by a number of special commands oriented toward the conventional interpretation of matrices in mathematics: inverses, determinants, etc. Matrices can be used for other kinds of data, but you should be cautious about using them in a manner that is too far distant from the conventional: arithmetic operations in Maxima in particular, have already been defined.

Arrays are unusual in their flexibility in Maxima. They are usually treated as global tables, although they can be declared to be local to a function; they cannot be passed around as parameters as is the case with matrices. The names of arrays may be passed from program to program, but the data itself is not recopied, nor is it convenient to even make a copy of the data into another array. *Hashed* arrays are particularly ingenious, and have been illustrated earlier. Functions can be associated with arrays to provide new values for entries as they are required.

At this point you should be able to make use of the Maxima manual to learn more details for representation of new data types as you develop your application.

The true programming buff may also be interested in the macro-expansion capabilities of Maxima and its extensible syntax. At this point we would discourage the use of these facilities by novices, but encourage their use by persons willing to experiment in providing the most versatile user-oriented packages within the Maxima framework.

There is a *compilation* facility which allows users to translate Maxima code into potentially faster running code. Since most of the time in most programs is used by calls to LISP programs, this is usually ineffective. In general, this should be avoided by novices and most experienced users, since time spent on this is more wisely spent on mathematical restructuring of the solution method, or (in the case of primarily numerical computation), using a numerical library routine written in a suitable language.

CHAPTER 9

Graphics and Forms of Output

In addition to some options with respect to the mathematical output from Maxima on the command line, the system is also capable of interfacing with external programs to provide graphing capabilities. You can also save Maxima sessions to reload later.

9.1 Options on the Command Line

While for normal use the default settings are likely to be preferable, Maxima allows you to set some options with regards to how output is returned.

9.1.1. 1D vs. 2D

The default output Maxima gives you when you evaluate an expression is 2D output, which basically means you have some visual feedback on the structure of things like fractions and integrals as ascii art, and the output expression is not one string. Normally this is a good thing, but if for any reason you wish to turn that feature off, it is quite possible to do so. Maxima has an internal variable regulating this feature called `DISPLAY2D`. Ordinarily it is set to TRUE, providing the typical ascii Maxima output you would get in a terminal session. Setting it to FALSE will cause returns to be a single string. Here is an example:

```
(C1) 2*y*3*x/(4*x+y+3/4);
                                    6 x y
(D1)                             -----------
                                           3
                                 y + 4 x + -
                                           4
(C2) DISPLAY2D:FALSE;
(D2) FALSE
(C3) 2*y*3*x/(4*x+y+3/4);
(D3) 6*x*y/(y+4*x+3/4)
```

One important note here is that if you only wish to change the dimension of your output for one entry, you must set this variable by hand both before and after the command - i.e., `ev(2/3,DISPLAY2D:FALSE)` will not result in `2/3` unless the DISPLAY2D:FALSE command has already been entered - ev will not use it in its evaluation.

9.1.2. TeX Strings as Output

As you probably already know, Maxima is able to output expressions in TeX format. This ability is used transparently in several places, but you can also request this output manually, using the `tex(expression)` command.

```
(C5) A1:integrate(2*x+x^3+3*x^2,x);
                                 4
                                x      3    2
(D5)                            -- + x  + x
                                4
(C6) tex('integrate(sin(x)/(2*x+x^3+3*x^2),x));
$$\int {{{\sin x}\over{x^{3}+3\>x^{2}+2\>x}}}{\>dx}$$
(D6)                                FALSE
(C7) tex(A1);
$${{x^{4}}\over{4}}+x^{3}+x^{2}$$
(D7)                                FALSE
```

This is useful if you are writing a paper without the benefit of Emaxima and wish to include a Maxima result - the above can be pasted directly into a LaTeX environment.

9.1.3. Writing a Session to a File

This is actually a bit tricky, especially if you decide halfway through a session you wish to make a record of it. Fortunately, however, there are some tools you can use.

Note: Need to figure out the subtlties of save()

Writefile

The basic command you will want to start running when you want to record a session. The syntax works like this: `writefile("/home/user/maxima sessionoutput.txt")$`

This will record the entire session into a text file called `maximasessionoutput.txt` in the `/home/user` directory. If you wish close this file, you simply use the command `closefile("/home/user/maxima sessionoutput.txt")$` These files are not loadable as commands - it is merely a transcript of a session as it occurred. This is useful for basic text documentation preperation, or just basic saving of a session for printing.

If you wish to record lines that were entered before you began writing the file to disk, you can use a command called `PLAYBACK` to get them into the record. (Discuss playback options here.)

Creating BATCHable files - Stringout

If the user wishes to create a file containing session information which may be loaded again into the system using the BATCH, then `STRINGOUT` is the command you want to use. There are several choices here - you can save specific input (listing C labels), include all C input lines by supplying the arguement INPUT, all of the functions you have defined by supplying the argument FUNCTIONS, and also all the values you have defined by supplying the arguement VALUES. Since this is potentially quite important in the creation of packages and specialized invironments, we will go into this with some detailed examples.

Let's imagine we want to create a package to calculate the volume of a cube, with certain default values in place. We begin by entering these commands in Maxima:

```
(C1) defaultlength:35$
(C2) defaultheight:45$
(C3) defaultwidth:65$
(C4) volume(length,width,height):=length*width*height;
(D4)        volume(LENGTH, width, height) := LENGTH width height
(C5) defaultvolume:volume(defaultlength,defaultwidth,defaultheight)$
(C6) defaultvolume;
(D6)                               102375
```

Of course, the last line is not strictly necessary, but it will serve to illustrate the difference between various options. Now, as a first example we will save all the lines of input to the file cube.mac:

```
(C7) stringout("cube.mac",INPUT);
```

```
(D7)       /home/user/cube.mac
```

Notice the return from this command is the directory where the file is located. If we look at the contents of that file:

```
defaultlength:35$
defaultheight:45$
defaultwidth:65$
volume(LENGTH,width,height):=LENGTH*width*height;
defaultvolume:volume(defaultlength,defaultwidth,defaultheight)$
defaultvolume;
```

We see that this is the series of commands we entered. If we were to load this file with the `batch` command (reference this somewhere) we would get precisely the same thing we got before. Now let's try the FUNCTIONS and VALUES arguements:

```
(C8) stringout("cubefunctions.mac",FUNCTIONS);
(D8)   /home/user/cubefunctions.mac
(C9) stringout("cubevalues.mac",VALUES);
(D9)    /home/user/cubevalues.mac
```

And we see the contents of those files are:

```
cubefunctions.mac

volume(LENGTH,width,height):=LENGTH*width*height;
```

```
cubevalues.mac

defaultlength:35;
defaultheight:45;
defaultwidth:65;
defaultvolume:102375;
```

So those are the basics of how output works. Experiment a little before you begin creating batch files, so you know in more detail what is saved and what isn't by various options. In the case of FUNCTIONS, the following will tell you what you have defined:

```
(C10) DISPFUN(ALL);
(E10)        volume(LENGTH, width, height) := LENGTH width height
(D10)                                DONE
```

You will also want to examine the `VALUES` variable and the use of the `PACKAGEFILE` variable.

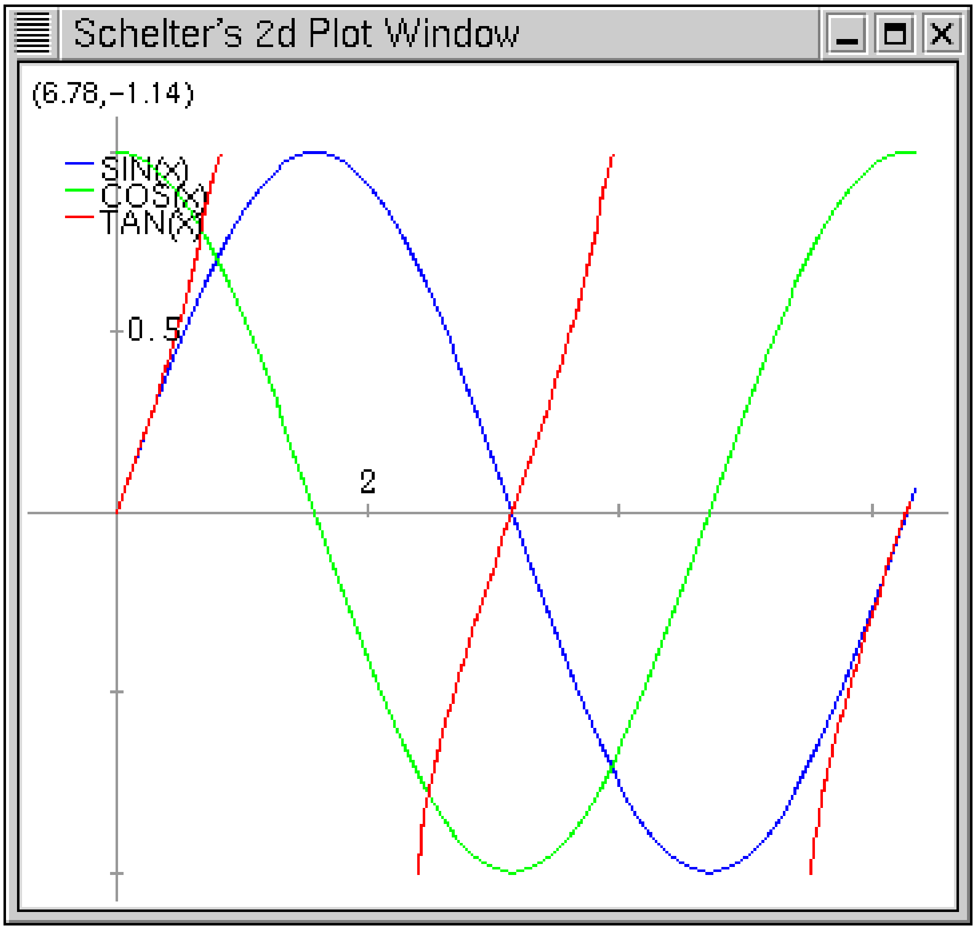
9.2 Graphics

Maxima has, via external programs, the ability to produce 2D and 3D graphs. The information will also be recorded in a file called maxout.openmath. The information in that file will always be the raw openmath format data of the last plot command.

9.2.1. 2D function plotting

2D graphs are produced with the `plot2d` command. Perhaps the simplest way to introduce this command is to show it in action.

```
plot2d([sin(x),cos(x),tan(x)],[x,0,2*%Pi],[y,-1,1]);
```

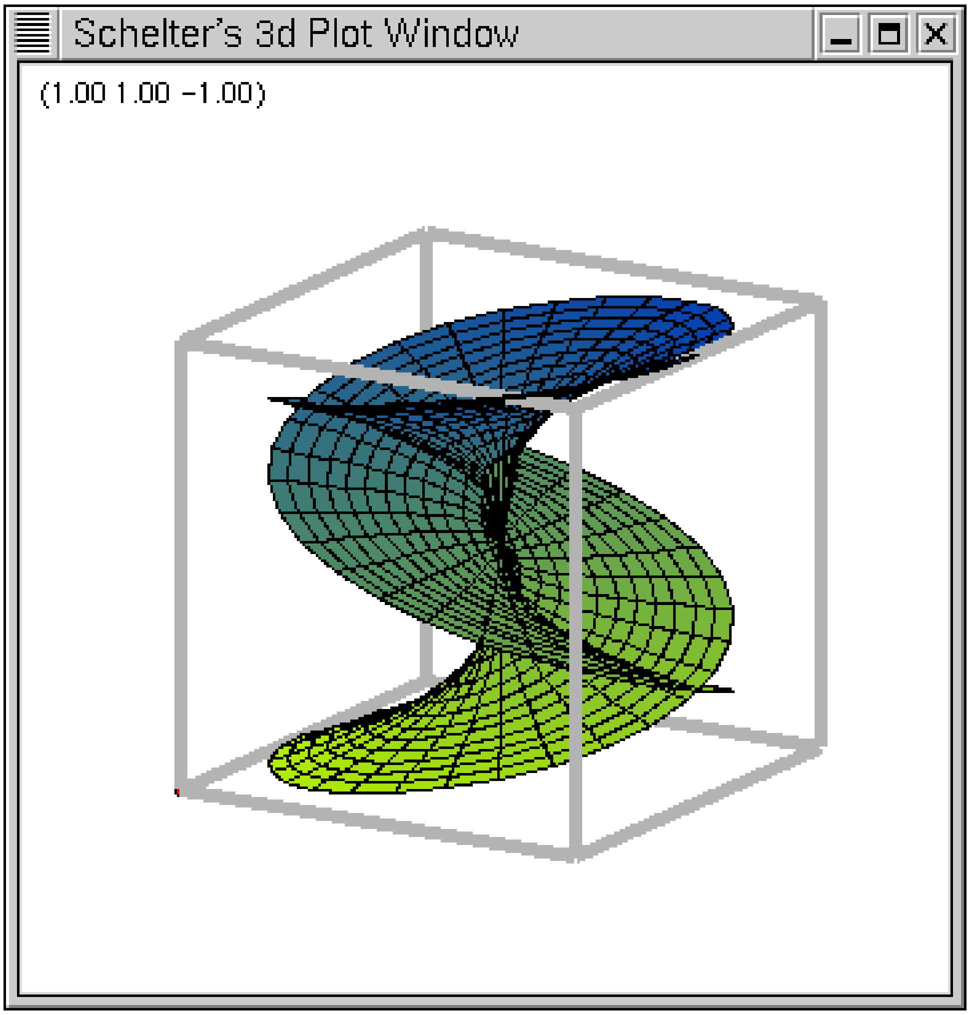


These are just the basics - there are many options which can be set, but most of them are part of an overall plot options system which we will discuss later. In the examples section at the end of this chapter we will show some more 2D plots with various options set.

9.2.2. 3D Function Plotting

This works just about like the 2D plotting, only you need to supply the proper parameters. Here again an example is the best teacher.

```
Plot3d(r^.33*cos(th/3),[r,0,1],[th,0,6*%pi],['grid,12,80],
['transform_xy,polar_to_xy],['view_direction,1,1,1.4],['colour_z,true]);
```



Notice in the latter shot the control menu is visible - this appears if you move the mouse to the upper left hand corner of the plot window, and will allow you to configure things like printing options.

9.3 Plot Options

Maxima defines a list, called `PLOT_OPTIONS`, which controls most of the behavior of Maxima's plotting options. There are two ways of setting these options - you can set an option globally, using the `SET_PLOT_OPTION` command [1], or supply the new options as arguements to a plot command. If you want to check what your global options are currently set at, you can just type `PLOT_OPTIONS;` and the current state of the system will be listed.

```
(C1) PLOT_OPTIONS;
(D1) [[x, - 3, 3], [y, - 3, 3], [GRID, 30, 30], [VIEW_DIRECTION, 1, 1, 1],
[COLOUR_Z, FALSE], [TRANSFORM_XY, FALSE], [RUN_VIEWER, TRUE],
[PLOT_FORMAT, OPENMATH], [NTICKS, 100]]
```

OK, let's look at each of these options.

- `[x, - 3, 3]` - This defines the X range. In order to change this range globally, use a command of the form `SET_PLOT_OPTION([x,-5,5]);`

[1] Setting an option globally only changes the option for the current Maxima session - upon restart, the original defaults will be restored.

- `[y, - 3, 3]` - This defines the Y range. In order to change this range globally, use a command of the form `SET_PLOT_OPTION([y,-5,5]);`
- `[GRID, 30, 30]` - This controls, in 3D plotting, the number of points used to draw the figure. The function is only calculated at a certain number of points - after that, linear approximations are drawn. Globally, usee a command of the form `SET_PLOT_OPTION([GRID,40,35]);`
- `[VIEW_DIRECTION, 1, 1, 1]` - This option is specific to the case when the plot command outputs directly to postscript in 3D(See `PLOT_FORMAT`.) It determines the direction from which the 'camera' looks at the function, which is along a line parallel to the line from `VIEW_DIRECTION` to the origin. It only needs to be set in the case of postscript output; it is ignored otherwise. Globally, use a command of the form `SET_PLOT_OPTION([VIEW_DIRECTION,1.4,1.4,1.4]);`
- `[COLOUR_Z, FALSE]` - This refers also to the postscript output - if set to TRUE it provides a little color shading in the output. Form is `SET_PLOT_OPTION([COLOUR_Z, TRUE]);`
- `[TRANSFORM_XY, FALSE]` - This appears to provide the ability to produce plots with different coordinate systems, but I am unsure of how to make it work. Need to get help here.
- `[RUN_VIEWER, TRUE]` - If you only wish Maxima to output the openmath file and not launch the graphical viewer, set this option to false. Remember, however, that if you wish to run multiple commands to generate data you will have to recover the information from the maxout.openmath file each time, because each new plot command will overwrite it. Form is `SET_PLOT_OPTION([RUN_VIEWER, FALSE]);`
- `[NTICKS, 100]` - Controls the number of points used to draw the 2D plots - the program will calculate the value of the function at N points, and then draw lines between them. Form is `SET_PLOT_OPTION([NTICKS, 200]);`
- `[PLOT_FORMAT, OPENMATH]` - This controls which program gets the output from Maxima for display. There are currently four viable options - OPENMATH, GNUPLOT, GEOMVIEW, and PS. PS is simply direct output to a postscript file, maxout.ps. All of these programs are freely available. Geomview is currently Unix only, and is available at http://geomview.sourceforge.net as both source and binary. Openmath is distributed as part of Maxima. Gnuplot is widely available, with a homepage at http://www.gnuplot.org and the most current development version at http://sourceforge.net/projects/gnuplot. Gnuplot can run on both Windows and Linux.

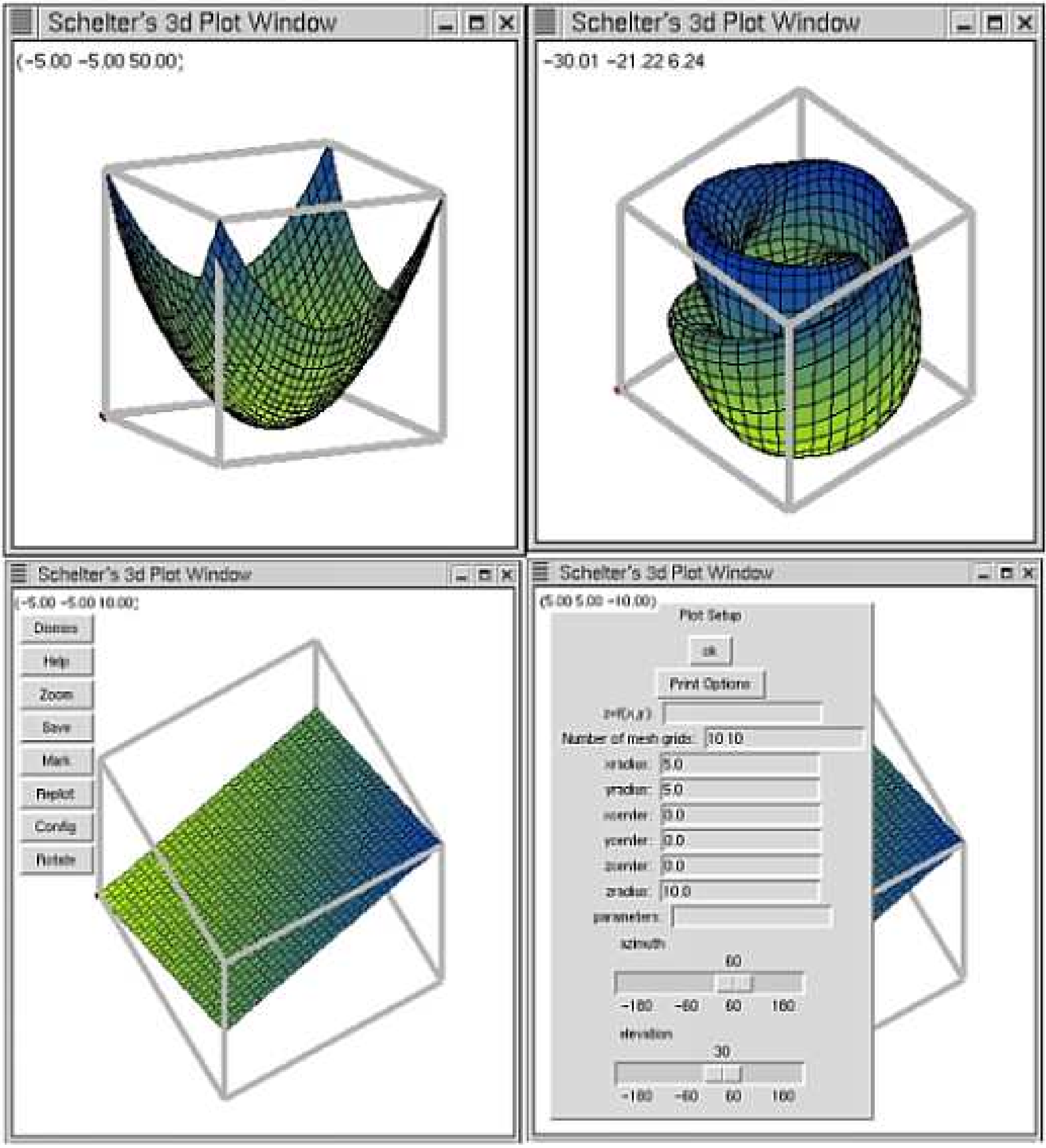


Figure 9.1: Graphing with Openmath, the default Maxima plotting tool

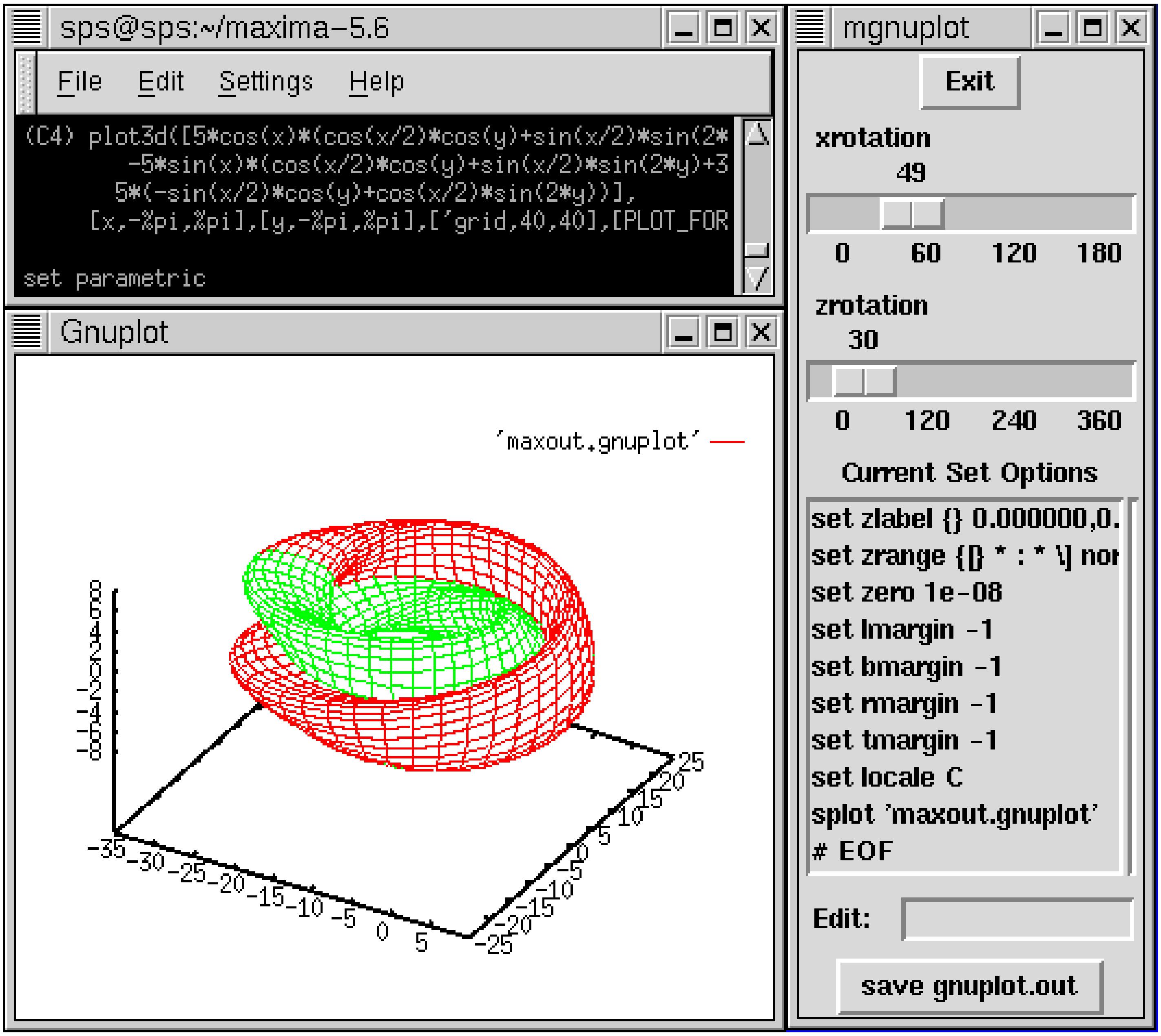


Figure 9.2: Graphing with gnuplot, using the mgnuplot utility

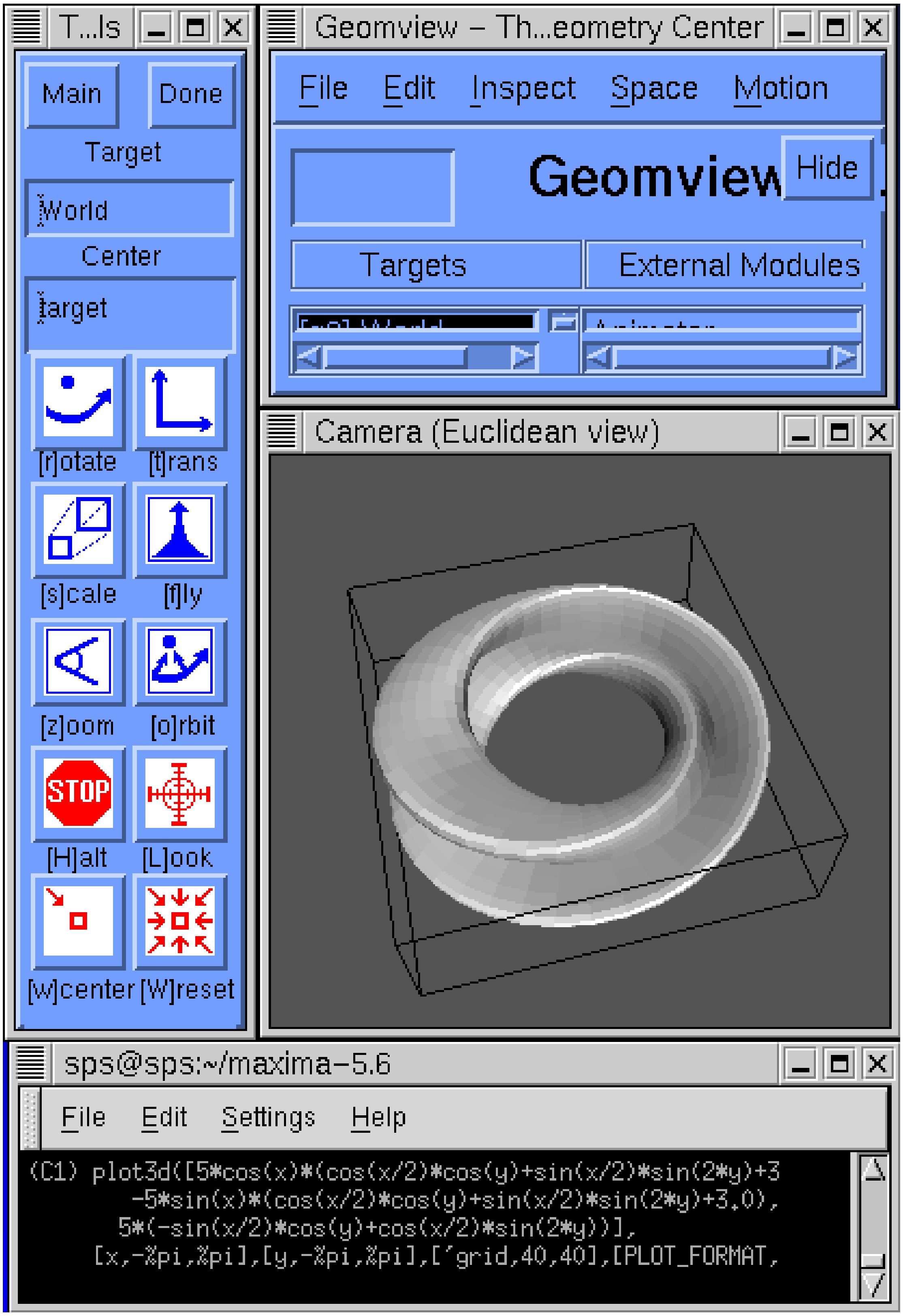


Figure 9.3: Graphing with GeomView.

CHAPTER 10

Maxims for the Maxima User

Some beginning users of algebraic manipulation systems find that their previous experiences with traditional programming systems do not translate easily into algebraic programming; others find Maxima descriptions inadequate because the emphasis is on the mixture of mathematical notations and algorithms, and not on *efficient* use of machine or human resources (no one likes to wait longer than necessary for an answer!). While we cannot provide a complete education in efficient and effective programming, we have collected a few *maxims* in an attempt to help you with some of these *start-up* problems.

> *Algebraic manipulation is a new and different computational approach for which prior experience with computational methods may be misleading. You should attempt to learn the ways in which algebraic manipulation differs from other approaches.*

For example, consider the problem of inverting an $n \times n$ matrix. In numerical analysis we learn that the problem requires on the order of n^3 operations. Theoreticians will point out that for n sufficiently large, one can reduce the number of multiplications below n^3 to $n^{2.8}$. This analysis is unfortunately not relevant in dealing with matrices with symbolic entries. Consider the number of terms in the determinant of the general $n \times n$ matrix whose elements are the symbols $a_{i,j}$. When the inverse is written out in a fully expanded form, just the determinant (a necessary part of representing the inverse) has $n!$ terms. It is impossible to compute this determinant in less than *time proportional to* $n!$ In fact, for large n, it is just not feasible to compute this form explicitly on existing computers. The combinatorial or exponential character that some algebraic manipulation problems have when they are approached with an inefficient algorithm, makes for a vastly different game from, say, numerical computation, where the size of objects is generally known at the onset of the calculation, and does not increase.

> *Needlessly generalizing a problem usually results in unnecessary expense.*

For example, if you wish to obtain determinants for a collection of matrices whose general pattern of entries is represented by parametric formulas, you might consider obtaining the determinant of the general matrix and substituting various values for the parameters into the result. This may work for matrices of low order, but is probably a poor plan for dealing with the exponential growth inherent in computing symbolic determinants. It would probably be better to substitute the parameters first, since this would drastically reduce the cost of the determinant calculation.

Sometimes, when humans are dealing with formulas, it is preferable to use an indeterminate, say G in a formula which is really known to be, say, 3/5. On the other hand, it is likely (although not certain!) that the calculation using 3/5 will take less time than the calculation with G. Since the cost inherent in some computations is usually a function of the number of variables in the expression, *it pays to reduce the number of variables in a problem as much as possible.*

> *You should be aware of the types of calculations which in the general case have exponential growth* (e.g. many matrix calculations with symbolic entries, repeated differentiation of products or quotients, solution of systems of polynomial equations).

> *Your should anticipate a certain amount of trial-and-error in calculations.*

Just as in other problem-solving activities, often the first technique that comes to mind is not the best. While it occasionally happens that brute force carries the day, cleverness in computing can be as important as cleverness in hand calculations. It is natural, during hand calculations, to apply critical simplificiations or substitutions in computations. These simplifications include collecting terms selectively or striking out terms which do not ultimately contribute to the final answer because of physical interpretations. Computer algorithms which do not incorporate these same tricks may bog down surprisingly soon. Thinking about these shortcuts may be important. In fact, it is one of the more rewarding aspects of computer algebra systems that they give the problem solver an opportunity to organize, encapsulate and distribute a particularly clever piece of mathematical manipulation.

> *Try to reduce your problem so that it can be performed in a simpler domain.*

For example, if your problem appears to involve trigonometric functions, logs, exponentials, etc. see if you can reduce it to a rational function (ratio of polynomials) problem. If it appears to be a rational function, see if you can, by substitutions, make it into a polynomial problem, or a truncated power-series problem. If it appears to be a problem involving rational numbers, consider the use of floating-point numbers as an alternative, if the growth in the size of numbers presents difficulties.

There are other special forms that are especially efficient. In a number of areas of investigation *it pays to convert all expressions to the internal rational form (using* `rat`*) or into Poisson-series form (using* `intopois`*) to avoid the overhead the the general representation*. The price you may pay here is thtat the structure of the formulas may be significantly different from those you began with: The canonical transformations used by these representations drastically re-order, expand, and modify the original expressions.

> *Sometimes someone else has already started on your problem.*

You should look through the *share* directory programs available to see if there are contributed packages that might be of use either as subroutines or as models for programming. You should also consider writing programs that you develop in solving your problems in a form suitable for sharing with others.

> *Pattern matching allows you to tune the system simplifier to your application, and develop rule-replacement programs.*

Learning to use the pattern-matching facilities effectively is a nontrivial task. Nevertheless if you have a fairly complex problem involvilng the recognition and application of identities, you should consider making an effort to use these facilities. In recent years, advocates of *rule-based expert systems* have claimed that this type of formalism can or should be used to incorporate varied types of knowledge. Algebraic manipulation programs have depended on pattern matching since at least 1961, for some of their power.

Finally, we would like to point out that algebraic manipulation systems, in spite of their pitfalls, can be of major assistance in solving difficult problems. If you are willing to invest some time in learning, there may be enormous benefits in using such a system. We think it is unfortunate that some users reserve Maxima for difficult problems. Those of us who have grown up with Maxima near at hand find it of great use in routine computations as well.

CHAPTER 11

Help Systems and Debugging

Using the debugger and the online help system.

CHAPTER 12

Troubleshooting

Common errors, their cause and solution.

CHAPTER 13

Advanced Examples

These examples try to draw on everything in the manual to show you what can be done with Maxima in advanced usages. This is NOT a good starting point for new users.

Establishing a Minimum for the Rayleigh Quotient

We begin by defining the Rayleigh Quotient in general. From basic Regular Sturm-Liouville Eigenvalue principles, we know that the Rayleigh Quotient is defined as

$$\lambda = \frac{-p\phi \left. \frac{d\phi}{dx} \right|_a^b + \int_a^b \left[p \left(\frac{d\phi}{dx} \right)^2 - q\phi^2 \right] dx}{\int_a^b \phi^2 \sigma \, dx}$$

given the Sturm-Liouville differential equation

$$\frac{d}{dx}\left(p(x) \frac{d\phi}{dx} \right) + q(x)\phi + \lambda\sigma(x)\phi = 0$$

where $a < x < b$.

Maxima

```
RQ:(-p*('ev('ev(u(x)*'diff(u(x),x)),x=a)-'ev('ev(u(x)*diff(u(x),x)),x=b))+
'integrate(p*'diff(u(x),x)^2-q*u(x)^2,x,a,b))/'integrate(u(x)^2*sigma,x,a,b);
```

TeX Output

$$\frac{\int_a^b p\%\mathrm{DERIVATIVE}^2\left((u(x), x, 1)\right) - qu^2\left((x)\right)\, dx - p\left(\mathrm{EV}\left(\mathrm{EV}\left(u(x)\left(\frac{d}{dx}u(x)\right)\right), x = a\right) - \mathrm{EV}\left(\mathrm{EV}\left(u(x)\left(\frac{d}{dx}u(x)\right)\right), x = b\right)\right)}{\sigma \int_a^b u^2\left((x)\right)\, dx}$$

Now we evaluate it. This must be done in stages, otherwise the ev command will not understand its arguements.

Maxima

```
ev(RQ,p=1,q=0,sigma=1,u(x)=x-x^2,a=0,b=1);
ev(%,diff,integrate);
ev(%,ev);
```

TeX Output

$$\frac{\mathrm{EV}\left(\mathrm{EV}\left(\left(x-x^2\right)\left(\frac{d}{dx}\left(x-x^2\right)\right)\right),x=1\right)-\mathrm{EV}\left(\mathrm{EV}\left(\left(x-x^2\right)\left(\frac{d}{dx}\left(x-x^2\right)\right)\right),x=0\right)+\int_0^1 \%\mathrm{DERIVATIVE}^2\left(\left(x-x^2,x,1\right)\right)\,dx}{\int_0^1 \left(x-x^2\right)^2\,dx}$$

$$\frac{30\left(\mathrm{EV}\left(\mathrm{EV}\left(\left(1-2x\right)\left(x-x^2\right)\right),x=1\right)-\mathrm{EV}\left(\mathrm{EV}\left(\left(1-2x\right)\left(x-x^2\right)\right),x=0\right)+\frac{1}{3}\right)}{10}$$

This can be checked by hand. Seeing that it is correct, we now can use it to search for the minimum eigenvalue on a more difficult problem:

Maxima

```
ev(RQ,p=1,q:-(x^2),sigma=1,u(x)=x-1,a=0,b=1)$
ev(%,diff,integrate)$
EV(%,EV,NUMER);
```

TEX Output

$$6.1$$

Maxima

```
ev(RQ,p=1,q:-(x^2),sigma=1,u(x)=-2*x^2+2,a=0,b=1)$
ev(%,diff,integrate)$
ev(%,ev,NUMER);
```

TEX Output

$$2.642857142857143$$

Maxima

```
ev(RQ,p=1,q:-(x^2),sigma=1,u(x)=x^3+x^2-2,a=0,b=1)$
ev(%,diff,integrate)$
ev(%,ev,NUMER);
```

TEX Output

$$2.776084010840108$$

The smallest eigenvalue must therefore be less than or equal to 2.642857...

Laplacian in Different Coordinate Systems

This will probably go in the main documentation somewhere, but for now I'll stick it here.

It is possible to express the Laplacian in different coordinate systems, provided you know how to define the coordinate system. We will use Spherical Coordinates for our first example:

Maxima

```
load(vect)$
scalefactors([[rho*cos(theta)*sin(phi),rho*sin(theta)*sin(phi),rho*cos(phi)],rho,theta,phi]);
depends(f,[rho,theta,phi]);
express(laplacian(f));
ev(%,diff)$
ratexpand(%);
```

TEX Output

```
  Warning - you are redefining the MACSYMA function TRIGSIMP
Warning - you are redefining the MACSYMA function IMPROVE
Warning - you are redefining the MACSYMA function UPDATE
```

$$\mathrm{DONE}$$

$$[f(\rho, \vartheta, \varphi)]$$

$$\frac{\frac{d}{d\rho}\left(\frac{d}{d\rho} f\, |\sin\varphi|\, \rho^2\right) + \frac{d}{d\vartheta}\frac{\frac{d}{d\vartheta} f\, |\sin\varphi|}{\sin^2\varphi} + \frac{d}{d\varphi}\left(\frac{d}{d\varphi} f\, |\sin\varphi|\right)}{|\sin\varphi|\, \rho^2}$$

$$\frac{2\left(\frac{d}{d\rho} f\right)}{\rho} + \frac{\frac{d}{d\varphi} f \cos\varphi}{\sin\varphi\rho^2} + \frac{\frac{d^2}{d\vartheta^2} f}{\sin^2\varphi\rho^2} + \frac{\frac{d^2}{d\varphi^2} f}{\rho^2} + \frac{d^2}{d\rho^2} f$$

Part II

External, Additional, and Contributed Packages

CHAPTER 14

The Concept of Packages - Expanding Maxima's Abilities

This chapter will try to give some in depth information on the why and how of Maxima packages. It is true that many "standard" abilities of Maxima, such as vectors, are due to packages, but this part of the book will deal with more specialized contributed packages, such as elliptical integrals, special scientific packages, etc. That way the first part of the book can be rewritten less frequently, but we can still have all the information here.

CHAPTER 15

Available Packages for Maxima

Maybe we can do this in one chapter, maybe each package will warrant a chapter. Not sure. For now, I'll assume one Section per package, not one Chapter.

List of Figures