Maxima by Example: Ch. 1, Introduction to Maxima [*]

Edwin L. Woollett

August 11, 2009

Contents

[*] This is a live document. This version uses **Maxima 5.19.0**. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see later in this chapter
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the Xmaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
  Maxima, a Computer Algebra System.
  Some numerical results depend on the Lisp version used.
  This chapter uses Version 5.19.0 (2009) using Lisp GNU
  Common Lisp (GCL) GCL 2.6.8 (aka GCL).
  http://maxima.sourceforge.net/
```

Acknowledgements

Some of the examples used in these notes are from the Maxima html help manual or the Maxima mailing list:
`http://maxima.sourceforge.net/maximalist.html`.

Our discussion begins with some of the "nuts and bolts" of using the software in a Windows XP environment, and continues with information useful to a new user.

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

1.1 What is Maxima?

Maxima is a powerful computer algebra system (CAS) which combines symbolic, numerical, and graphical abilities. See the Maxima sourceforge webpage **`http://maxima.sourceforge.net/`**.

A cousin of the commercial Macsyma CAS (now available but without support), Maxima is a freely available and open source program which is being continuously improved by a team of volunteers. When compared with Mathematica or Maple, Maxima has a more basic interface, but has the advantage in price (!). Students, teachers, and researchers can "own" multiple copies for home, laptop, and desktop without the expense of buying licenses for each copy.

There are known "bugs" in the present version (a new version is available about three times each year), and the volunteer developers and programming experts are dealing with these known bugs as time permits.

Maxima is not only "free" and will always stay that way, but also comes with a copy of the underlying source code (in a dialect of the Lisp language), which a user can modify to suit her own research needs and then share with the Maxima community of users.

A self-installing binary for Windows users is available, making it easy for Windows users to get a fast start.

Here is a more technical description of Maxima, taken from the Unix/Linus man document:

> Maxima is a version of the MIT-developed MACSYMA system, modified to run under CLISP. It is an interactive expert system and programming environment for symbolic and numerical mathematical manipulation. Written in Lisp, it allows differentiation, integration, solution of linear or polynomial equations, factoring of polynomials, expansion of functions in Laurent or Taylor series, computation of Poisson series, matrix and tensor manipulations, and two- and three-dimensional graphics.
>
> Procedures may be written using an ALGOL-like syntax, and both Lisp-like functions and pattern matching facilities are provided. Files containing Maxima objects may be read from and written to disk files. Pre-written Maxima commands may be read from a file and executed, allowing batch-mode use.
>
> Maxima is a complex system. It includes both known and unknown bugs. Use at your own risk. The Maxima bug database is available at
> **`http://sourceforge.net/tracker/?atid=104933&group_id=4933&func=browse`**.
>
> New bug reports are always appreciated. Please include the output of the Maxima function **build_info()** with the report.

Information about the history of Maxima and its relation to Macsyma can be found in the last section of this chapter.

You should first familiarize yourself with the Maxima work environment by downloading and installing Maxima on your computer, and starting up either the wxMaxima or the XMaxima interface.

1.2 Which Maxima Interface Should You Use?

New users generally like to start with **wxMaxima** since there are convenient icons which help locate Maxima functions for common tasks. **wxMaxima** allows the user to construct a combination text, calculation, and plot document which can be saved and used as a homework submission, used as a tutorial for others and/or simply used as a "permanent" record of work on some topic. For new users, the menus and buttons allow a gradual learning of Maxima syntax, by reading what the menus and buttons do with an expression, and the user can pick up a knowledge of most of the "power tools" in Maxima in this way. The reader should be warned, however, that no menu and button system can include every Maxima function which might be either useful or needed for a particular task.

Experienced users tend to split between **wxMaxima** and **Xmaxima**, switching to **Xmaxima** because they already know the names of common Maxima functions which help in getting the job done, and appreciate a simple stable interface without distractions. Experienced users tend to become good "touch typists", able to type most things without looking at the keyboard and using both hands. Such experienced users usually find that it is faster to just type the name than to reach for the mouse and manipulate the cursor to the right button. The **Xmaxima** interface is quite stable between new versions of Maxima, whereas the **wxMaxima** interface is being actively developed and changes to the appearance and behavior occur frequently during this period of rapid development.

Xmaxima is also a faster environment for testing and playing with code ideas, and the session record can be easily copied and pasted into a Latex verbatim environment with zero hassle. The current version of **wxMaxima** does not provide this hassle free transfer to a latex document (although one can save output as an image, but then one must go through the hassle of converting to eps file image format if one is using the conventional latex to dvi to pdf route).

1.3 Using the wxMaxima Interface

From the `wxMaxima` webpage `http://wxmaxima.sourceforge.net/wiki/index.php/Main_Page` which could be accessed via

```
start, My Programs, Maxima-5.19.0, wxMaxima on the Web
```

one finds the information:

```
wxMaxima features include:
  * 2D formatted math display: wxMaxima implements its
        own math display engine to nicely display maxima output.
  * Menu system: most Maxima commands are available through menus.
      Most used functions are also available through a button
        panel below the document.
  * Dialogs: commands which require more that one argument can
        be entered through dialogs so that there is no need to
        remember the exact syntax.
  * Create documents: text can be mixed with math
      calculations to create documents. Documents can be
        saved and edited again later.
  * Animations: version 0.7.4 adds support for simple animations.
```

On that page is a link to a set of `wxMaxima` tutorials on the page
`http://wxmaxima.sourceforge.net/wiki/index.php/Tutorials`.

There you can download tutorials in the form of zip files, which, when unzipped, become wxMaxima document format (wxm) files which can then be loaded into your wxMaxima work session.

`10minute.zip` contains a "ten minute" wxMaxima tutorial. `usingwxmaxima.zip` provides general information about the cell structure of wxMaxima.

Starting wxMaxima

One can use the Windows Start menu:

```
start, All Programs, Maxima-5.19.0, wxMaxima
```

During the Windows binary setup process you can select the options which place icons for both **`wxMaxima`** and **`XMaxima`** on your desktop for a convenient start, and you can later copy any shortcut and paste it into your work folder for an alternative start method.

Quitting wxMaxima

The quick way to quit is the two-key command **Ctrl + q**.

The Maxima Manual

To access the Maxima manual from within **`wxMaxima`**, you can use function key **`F1`** or the menu item
`Help, Maxima Help`.

The Online wxMaxima Forum

You can access the **`wxMaxima`** online forum at the web page

```
http://sourceforge.net/forum/forum.php?forum_id=435775
```

which can be accessed using

```
Start, My Programs, Maxima-5.19.0, wxMaxima Online Forum
```

and search for topics of interest.

The Cell Structure of wxMaxima

In the second tutorial, Using wxMaxima, the cell structure of a wxMaxima document is explained (we will use parts of that discussion here and we give only a rough idea of the actual appearance here):

The top of the cell bracket is actually a triangle. The following is a “text cell” which has no Maxima code.

```
-----
|
| Unlike "command-line Maxima"' (such as XMaxima or Console mode)
| which works in a simple input-output manner, wxMaxima introduces
| the concept of a live mathematical document, in which you mix
| text, calculations and plots.
|
| Each wxMaxima document consists of a number of so called "cells".
| The cell is the basic building block of a wxMaxima document. Each cell has a
| bracket on the left border of the document, indicating where the cell
| begins and ends. Cells are of different types. You can have a "title
| cell", a "section cell", and a "text cell" like this one. The most
| important cell type is the "input cell".
---
```

The next cell is an example of an input cell, which has the input prompt **>>** at the top. Note that we can include a text comment within an input cell (which will be ignored by the Maxima computational engine) by putting the comment between the delimiters **/*** and ***/** :

```
----
| >> /* this is an input cell - it holds Maxima code and can be
| evaluated by first left-clicking once anywhere in the cell and
| then using the two-key command SHIFT-ENTER. The code entered in this cell
| will be sent to Maxima when you press SHIFT-ENTER. Before
| wxMaxima sends code to Maxima, it checks if the contents
| of each code command in this input cell ends with a ';' or a '$'.
| If it doesn't, wxMaxima adds a ';' at the end. Maxima requires that
| each completed Maxima statement end with either ';' or '$'.
| Note that this does not mean you have to have each statement on
| one line. An input cell could have the contents
|     sin
|     (
|      x
|       );
|
| and this would be accepted as a complete Maxima input, equivalent to
|    sin(x);
|
| Any *output* wxMaxima gets from Maxima will be attached to the end of
| the input cell. Try clicking in this cell and pressing SHIFT-ENTER. */
|
|   /*  example Maxmima code: */
|
|   print("Hello, world!")$
|   integrate(x^2, x);
---
```

If you click once somewhere inside this cell the vertical left line changes to (if you have not changed the default) bright red, indicating that the cell has been selected for some action. If you then press the two-key combination **Shift + Enter**, the prompt **>>** will be replaced by **(%i1)**, and the results of the two separate Maxima commands will be shown at the bottom of the cell as:

```
|  Hello, world!
|            3
|           x
|    (%o2)  ---
|           3
----
```

except that the results are shown using pixel graphics with much better looking results than we show here.

There is no **(%o1)** since the **print** command was followed by the **$** sign, which surpresses the normal default output.

If you have either a blank screen or a horizontal line present and just start typing, an input type cell is automatically created. (You can instead, prior to typing, use the function key **F7** to create a new input cell and can use the function key **F6** to create a new text cell).

In the current version of wxMaxima, if you change windows to some other task unrelated to wxMaxima, a junk input cell may be created in wxMaxima containing something which looks like a percent sign **>> %** which you will see when you come back to your wxMaxima window. If you want to have an input cell at that location, just backspace to get rid of the weird symbol and continue.

If you then want to delete this junk cell, (or any cell you might be working on) just left-click on the bottom of the cell bracket (which will highlight the bracket) and press the **Delete** key (or else select **Edit, Cell, Cut Cell**).

An alternative method is to use the horizontal line as a vertical position cursor which acts on cells. If your cursor is still inside a cell you want to delete, use the DOWN key to get the cursor out of the cell, and the horizontal black line appears. At that point you can press the backspace (or delete) key twice to delete the cell above the black horizontal line. Or you can press SHIFT-UP once to select that cell and then press DELETE. Using SHIFT-UP more than once will select a group of cells for action.

In the author's copy of **wxMaxima** (ie., using Windows XP), trying to use the usual Windows menu key combination **Alt + E**, for example, does not actuate the **wxMaxima** Edit menu icon; one must left-click on that Edit icon at the top of the screen to choose among the options.

The next cell is a "text cell" which does not need Maxima comment delimiters `/*` and `*/`.

```
----
|
| Again, there is a triangle at the top of both text and
|  input type cells which we don't show. An open triangle is the
|  default, but if you click on the triangle, it will turn solid
|  black and a) for a text cell, the text content is hidden, and
|  b) for an input type cell, the Maxima ouput of the cell is hidden.
|  Clicking that solid black triangle again will restore the hidden
|  portions to view.
|
|  Editing cells is easy. To modify the contents of a cell,
|  click into it (ie., left-click anywhere in the cell once).
|  A cursor should appear and the left cell bracket should turn red,
|  (in default style mode) indicating that the cell is ready to be edited.
|
|  The usual Windows methods can be used to select parts of a cell.
|  One method is to hold the left mouse button down while you drag
|  over your selection. A second method combines the SHIFT key with
|  other keys to make the selection. For example, left-click at
|  a starting location. Then use the two-key command SHIFT-END to
|  select to the end of the line (say) or SHIFT-RIGHTARROW to
|  select part of the line. You can then use CTRL-C  to copy your
|  selection to the clipboard. You can then left-click somewhere
|  else and use CTRL-V to paste the clipboard contents at that location.
|  (This is the usual behvior: if you experience lack of correct pasting,
|  you can temporarily revert to the longer process of using the
|  Edit, Copy, and Edit, Paste menu items to get wxMaxima in the
|   right spirit).
|  The DOWN arrow key will step down one line at a time through the
|  cell lines and then make a horizontal line under the cell. You can
|  also simply left-click in the space between cells to create the
|  active horizontal line, which you can think of as a sort of
|  cursor for vertical position. With the horizontal line present,
|  simply start typing to automatically create a new input type cell.
|  Or press the function key F6 to create a new text type cell
|  at that vertical location.
--
```

When you're satisfied with the document you've created, save it using the two-key **Ctrl + s** command or the **File, Save** menu command. Note that the output parts of input cells will not be saved. When you load your document later, you can evaluate all cells by using the **Edit, Cell, Evaluate all cells** menu command or the shortcut two-key command **Ctrl + r**. If the shortcut two-key command doesn't work the first time, just repeat once and it should work. The present version of

wxMaxima is a little cranky still. This will evaluate all the cells in the order they were originally created.

Some common Maxima commands are available through the seven menu icons: **Maxima, Equations, Algebra, Calculus, Simplify, Plot and Numeric**. All of the menu choices which end with `.....` will open a dialog, to help you formulate your desired command. The resulting command will be inserted at the current horizontal cursor's position or below the currently active cell. The chosen command will also be evaluated.

Configuring the Buttons and Fonts

The bottom button panel can be configured through **Edit, Configure, Options Tab, Button Panel, Full or Basic**, and the font type and size can be configured through **Edit, Configure, Style Tab, Default Font, Choose Font, Times New Roman, Regular, 12** , for 12 pt generic roman for example. This is smaller than the startup default font, and may be too small for some users, but will allow more information to be seen per screen.

The default text cell background color is a six variable custom green that is nice. In the following, the italic box is left unchecked unless mentioned otherwise. To change a color, click the color bar once. The author combines the 12 pt roman choice with input labels in bold red, Maxima input in blue bold, text cell in bold black, strings in black bold italic, Maxima questions in blue bold, and all of the following in bold black: output labels, function names, variables, numbers, special constants, greek constants.

When you want to save your choices, select the Save button, which will write a file **style.ini** to the folder in which wx-Maxima was started.

(The author uses a shortcut to wxMaxima placed in his **`c:\work2`** windows xp folder, since that folder is where the author expects saved wxm and xml files to be saved. By starting with the contents of that folder in view, the author can then simply click on the wxMaxima shortcut link to begin work with wxMaxima in that folder.)

The author chose the **Full** bottom button option (the default is **Basic**), which draws twenty buttons in two rows at the bottom of the screen. The top row contains the **Simplify, Simplify(r), Factor, Expand, Simplify(tr), Expand(tr), Reduce(tr), Rectform, Sum..., Product...** buttons.
The bottom row contains the **Solve..., Solve ODE..., Diff..., Integrate..., Limit..., Series..., Subst..., Map..., Plot2D..., Plot2D...** buttons.
Note that if you have made a selection before you left-click a button panel command or a menu command, that selection will be used as the main (or only) argument to the command. Selection of a previous output or part of an input will work in a similar manner. The selected button function will act on the input cell contents or selection and immediately proceed with evaluation unless there is an argument dialog which must be completed.

Multiple Maxima Requests in One Input Cell

Here is a calculation of the cross sectional area **`a`** and volume **`v`** of a right circular cylinder in terms of the cross section radius **`r`** and the height **`h`** (ignoring units). We first create an input cell as described above, by just starting to type the following lines, ending each line with the ENTER key to allow entry of the following line.

```
|---
| >>  (r : 10, h : 100)$
|      a : %pi * r^2;
|      v : a * h;
---
```

Note that we put two Maxima assignment statements on one line, using the syntax **(job_1, job_2)$**. Naturally, you can put more than two statements between the beginning and ending parentheses. Thus far, the Maxima computational engine has not been invoked. The left cell bracket will be bright red (in the default style), and you will have a blinking cursor in the cell somewhere.

If you now use the two-key command SHIFT-ENTER, all Maxima commands will be carried out and the ouputs will show up (in the same order as the inputs) at the bottom of that input cell, looking like

```
----
| (%i1)  (r : 10, h : 100)$
|         a : %pi * r^2;
|         v : a * h;
| (%o2)   100*%pi
| (%o3)   10000*%pi
---
```

with the horizontal (vertical location) line present underneath. The input prompt `>>` was replaced with the input number `(%i1)`. Output `(%o1)` was not printed to the screen because the first input ended with `$`.

Now just start typing to start the next input cell:

```
----
| >>   [ a, v ], numer ;
---
```

With the cell bracket highlighted in red and the blinking cursor inside the cell, press SHIFT-ENTER to carry out the operation, which returns the numerical value of the cross-sectional area `a` and the cylinder volume `v` as a list.

```
----
| (%i4)    [ a, v ], numer ;
| (%o4)   [314.1592653589793,31415.92653589793]
```

1.3.1 Rational Simplification with ratsimp and fullratsimp

A rational expression is a ratio of polynomials or a sum of such. A particular case of a rational expression is a polynomial (whether expanded out or factored). **ratsimp** and **fullratsimp** can be useful tools for expressions, part of whose structure is of this form. Here is a simple example. If you are using wxMaxima, just start typing the expression you see entered below:

```
----
| >>  (x+2)*(x-2)
---
```

When you type a leading parenthesis `'('`, the trailing parenthesis `')'` also appears automatically. However, if you type an ending parenthesis anyway, wxMaxima will not use it, but rather will properly end with one parenthesis. However, you should pay attention to this feature at first so you understand how it works. Un-needed parentheses will result in a Maxima error message.

Now press HOME and then SHIFT-END to select the whole expression, **(x+2)*(x-2)**, then click on the **Simplify** button at the bottom of the screen. Maxima will apply a default type of simplification using the function **ratsimp**, and the cell contents evaluation will show

```
|(%i5) ratsimp((x+2)*(x-2));
|         2
|(%o5)  x  - 4
```

Thus the **Simplify** button uses **ratsimp** to simplify an expression. Instead of using the **Simplify** button, you could use the menu selection: **Simplify, Simplify Expression**. The word "simplify" is being used here in a loose fashion, since Maxima has no way of knowing what a particular user will consider a useful simplification.

A nice feature of wxMaxima is that you can left-click on a Maxima function name (in an input cell) like **ratsimp** and then press the function key **F1**, and the Maxima manual will open up to the entry for that function.

Note that if you had started with the input cell contents

```
----
| >> log( (x+2)*(x-2) ) + log(x)
---
```

and then highlighted just the `(x+2)*(x-2)` portion, followed by clicking on the **Simplify** button, you would get the same input and output we see above. Thus Maxima will ignore the rest of the expression. On the other hand, if you highlighted the `whole expression` and then clicked on **Simplify**, you would get

```
----
| (%i5)    ratsimp(log((x+2)*(x-2))+log(x));
|                                 2
| (%o5)                     log(x  - 4) + log(x)
---
```

fullratsimp

According to the Maxima manual

> **fullratsimp** repeatedly applies **ratsimp** followed by non-rational simplification to an expression until no further change occurs, and returns the result. When non-rational expressions are involved, one call to **ratsimp** followed as is usual by non-rational (”general”) simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. **fullratsimp** makes this process convenient.

Here is the Maxima manual example for **fullratsimp**, making use of the **Xmaxima** interface.

```
(%i1) expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);

                              a/2     2   a/2     2
                            (x    - 1)  (x    + 1)
(%o1)                       -----------------------
                                     a
                                    x  - 1
(%i2) expr, ratsimp;
                                 2 a      a
                                x    - 2 x  + 1
(%o2)                           ---------------
                                     a
                                    x  - 1
(%i3) expr, fullratsimp;
                                     a
(%o3)                               x  - 1
(%i4) rat (expr);

                              a/2 4       a/2 2
                            (x   )  - 2 (x   )  + 1
(%o4)/R/                    -----------------------
                                     a
                                    x  - 1
```

We see that **fullratsimp** returned a completely simplified result. **rat(expr)** converts **expr** to a **canonical rational expression** (CRE) form by expanding and combining all terms over a common denominator and cancelling out the greatest common divisor of the numerator and denominator, as well as converting floating point numbers to rational numbers within a tolerance of **ratepsilon** (see the Manual for more information about **rat**).

1.4 Using the Xmaxima Interface

To start up **Xmaxima**, you can use the Windows Start Menu route

```
Start, All Programs, Maxima 5.19.0, xmaxima
```

or click on the desktop icon for Xmaxima. You can copy the desktop icon to the Clipboard and paste it into any work folder for quick access.

If you are new to Maxima, go through the quick start tutorial in the bottom window of Xmaxima. You can either click on the expression to have it evaluated, or you can enter the expression, followed by a semi-colon (`;`) in the upper window and press enter to have Maxima carry out the operation. In the top Xmaxima window, note that the two key command **Alt+p** will type in the previous entry, and you can keep entering **Alt+p** until you get the entry you want (to edit or simply rerun).

Once you know what you are doing, you can close the bottom window of Xmaxima by using the Menu bar. On that menu bar, choose **Options, Toggle Browser Visibility**. To use only keyboard commands to close the bottom window, use the succession: **Alt + e**, **RightArrow**, **Enter**.

To quit Xmaxima, choose **File, Exit** on the menu bar (or **Alt+f**, **x**).

The second short introduction can be found on the **Start,All Programs, Maxima 5.19.0, Introduction** link. Written by Cornell University (Dept. of Theoretical and Applied Mechanics) Professor Richard Rand (`http://tam.cornell.edu/`), this is an excellent short introduction to many basic features of Maxima. The advice in Rand's introduction to exit Maxima by typing `quit();` is relevant if you are using the "command line maxima" version, aka "maxima console".

One important thing to remember when using Xmaxima is to never press **Enter** (to execute code) with a space between the semicolon (or the dollar sign) and the postion of the cursor. At least on the author's Windows XP machine, Xmaxima will "hang" and refuse to issue the next input prompt, and you will have to click on **File, Restart** on the Xmaxima menu bar to start a new session. This type of error can creep in easily if you are copying code you have previously saved to a text file, and have an extra space in the file. If you then select, copy, and paste that code fragment into Xmaxima, with the space at the end intact, you should carefully backspace to either the semicolon or the dollar sign before pressing **Enter**. The safest path is to make sure your original text selection for copy does not include a space beyond the dollar sign.

The Maxima Help Manual

The most important continuous source of information about Maxima syntax and reserved words is the Maxima Manual, which you should leave open in a separate window. To open a separate Maxima Manual window from inside the XMaxima interface, click on the XMaxima menu item: **Help, Maxima Manual** (you can use the shortcut **Alt+h** to open the Help menu).

Move around this reference manual via either **Contents** or **Index**. For example, left-click **Index** and start typing `integrate`. By the time you have typed in `inte`, you are close to the `integrate` entry, and you can either continue to type the whole word, or use the down arrow key to select that entry. Then press the **Enter** key. On the right side will be the Maxima Manual entry for `integrate`.

To scroll the Maxima Manual page, double-click on the page, and then use the **PageDown** and **PageUp** keys and the **UpArrow** and **DownArrow** keys. To return to the index entry panel, left click that panel, and type `diff`, which will take you to the section of the Maxima Manual which explains how to evaluate a derivative.

The Xmaxima Manual

If you look at the Xmaxima manual via **Help, Xmaxima Manual (Web Browser)** , your default browser will come up with a somewhat out of date manual with the sections: 1. Command-line options, 2. Xmaxima Window, 3. Entering commands, 4. Session control, 5. Openmath plots, 6. The browser, 7. Getting Help, and Concept Index.

The first section "1. Command-line options" is not relevant for Windows XP Xmaxima.

Xmaxima Font Choices

In Sec 2, Xmaxima Window, you will find the statement:

> You can also choose different types and sizes for the fonts, in the section 'Preferences' of the Options menu; those settings will be saved for future sessions.

The defaults are Times New Roman with size adjustment 3 for proportional fonts and Courier New with size adjustment 2 for fixed fonts. Using the menu with **Options, Preferences**, you can click on the typeface buttons to select a different font type, and can click on the size number button to select another font size. You then should click the **Apply and Quit** button.

Entering Your Expression

In Xmaxima, every input prompt, like **`(%i1)`**, is waiting for an input which conforms to Maxima's syntax expectations. There is no such thing as a "text cell", although you can include text comments anywhere as long as they are delimited by the standard comment delimiters **`/*`** and **`*/`**, which only need to occur at the very beginning and end of the comment, even if the comment extends over many lines.

Here is the beginning of Sec.3, Entering Commands, from the Xmaxima manual. We have replaced some irrelevant or obsolete material with updated instructions.

> Most commonly, you will enter Maxima commands in the last input line that appears on the text Window. That text will be rendered in weak green. If you press **Enter**, without having written a command-termination character (either **`;`** or **`$`**) at the end, the text will remain green and you can continue to write a multi-line command. When you type a command-end character and press the **Enter** key, the text will become light blue and a response from Maxima should appear in light black. You can also use the **UpArrow** or **DownArrow** keys to move to a new line without sending the input for Maxima evaluation yet. If you want to **clear** part of the current input from the beginning to some point, position your cursor at that point (even if the region thereby selected spans several lines) and then use **Edit, Clear input** or the two-key command **Ctrl+u**.
>
> If you move the cursor over the **`(%i1)`** input label, or any other label or output text (in black), you will not be able to type any text there; that feature will prevent you from trying to enter a command in the wrong place, by mistake. If you really want to insert some additional text to modify Maxima's output, and which will not be interpreted by Maxima, you can do that using cut and paste (we will cover that later).
>
> You can also write a new input command for Maxima on top of a previous input line (in blue), for instance, if you do not want to write down again everything but just want to make a slight change. Once you press the **Enter** key, the text you modified will appear at the last input line, as if you had written it down there; the input line you modified will continue the same in Xmaxima's and Maxima's memory, in spite of having changed in the screen.

For example, suppose you entered **`a: 45;`** in input line **`(%i1)`**, and something else in **`(%i2)`**.

```
(%i1) a:45;
(%o1)                          45
(%i2) b:30;
(%o2)                          30
```

You then move up over the **`(%i1) a: 45;`** and change the **`5`** to **`8`**. You can then press **End** to get the cursor at the end of the command, and then press **Enter** to submit the new (edited) command. At that point the screen looks like:

```
(%i1) a:48;
(%o1)                          45
(%i2) b:30;
(%o2)                          30
(%i3) a:48;
(%o3)                          48
```

But if you now enter **`(%i1);`** as input **`(%i4)`** and press **Enter**, the output **`(%o4)`** will be **`a: 45`**. The screen will now look like:

```
(%i1) a:48;
(%o1)                          45
(%i2) b:30;
(%o2)                          30
(%i3) a:48;
(%o3)                          48
(%i4) (%i1);
(%o4)                        a : 45
```

Maxima knows the current binding of both **`(%i1)`** (which is the output **`(%o4)`**) and **`a`**

```
(%i5) a;
(%o5)                          48
```

If you navigate through the input lines history (see next section), you will also see that the first input keeps its original value.

Speeding up Your Work with XMaxima

When you want to edit your previous command, use **Alt+p** to enter the previous input (or use enough repetitions of **Alt+p** to retrieve the command you want to start with). If the code extends over several screen lines, and/or your editing will include deleting lines, etc., delete the command-completion character at the end (**`;`** or **`$`**) first, and then edit, and then restore the command completion character to run the edited code.

The use of the keyboard keys **Home**, **End**, **PageUp**, **PageDown**, **Ctrl+Home**, and **Ctrl+End** (as well as **UpArrow** and **DownArrow** greatly speeds up working with Xmaxima

For example to rerun as is or to copy a command which is located up near the top of your current Xmaxima session, first use **Home** to put the cursor at the **beginning** of the current line, then **PageUp** or **Ctrl+Home** to get up to the top region fast.

If you simply want to rerun that command as is, use **End** to get the cursor at the end of the entry (past the command-completion character), and simply press **Enter** to retry that command. The command will be evaluated with the state of information Maxima has after the last input run. That entry will be automatically entered into your session (with new output) at the bottom of your session screen. You can get back to the bottom using the **Ctrl+End** two-key command.

Alternatively, if you don't want the retry the exact same command, but something similar, then select the part of the code you want to use as a starting point for editing and press **Ctrl+c** to copy your selection to the Window's Clipboard. To select, you can either drag over the code with the mouse while you hold down the left mouse button, or else hold down the **Shift** key with your left hand and and combine with the **End**, the **LeftArrow**, and the **DownArrow** keys to help you select a region to copy.

Once you have selected and copied, press **Ctrl+End** to have the cursor move to the bottom of your workspace where XMaxima is waiting for your next input and press **Ctrl+v** to paste your selection. If the selection extends over multiple lines, use the down cursor key to find the end of the selection. If your selection included the command-completion character, remove it (backspace over that final symbol) before starting to edit your copied selection.

You are then in the driver's seat and can move around the code and make any changes without danger of Xmaxima preemptively sending your work to the Maxima engine until you finally have completed your editing, and move to the very end and provide the proper ending (either `;` or `$`) and then press **Enter** to evaluate the entry.

Using the Input Lines History

If your cursor is positioned next to the active input prompt at the bottom of the screen (ie., where **Ctrl+End** places the cursor), you can use the key combinations **Alt+p** and **Alt+n** to recover the previous or next command that you entered. For example, if you press **Alt+n**, you will enter the first input **`(%i1)`**, and if you continue to press **Alt+n**, you will get in sucession **`(%i2)`**, **`(%i3)`**,...

Alternatively, if the active input prompt is **`(%i10)`** and you press **Alt+p** repeatedly, you will get inputs **`(%i9)`**, **`(%i8)`**, **`(%i7)`**, ...

Searching for a String in Your Previous Inputs

Those same two-key combinations can also be used to search for a previous input lines containing a particular string. Suppose you have one or more previous lines that included (among other functions) **sin(something)**. At the last input prompt, type **sin** and then use either of the two-key commands **Alt+p** or **Alt+n** repeatedly until the particular input line containing the instance of **sin** you are looking for appears. You can then either immediately rerun that input line (press **End** to get the cursor at the end of the input and then press **Enter**) or you can edit the input line using **RightArrow** and **LeftArrow**, **Home**, and **End** to move around, and finally complete your editing and press **Enter**. In summary, you first write down the string to search, and then **Alt+p**, to search backwards, or **Alt+n** to search forward. Pressing those key combinations repeatedly will allow you to cycle through all the lines that contain the string. If you want to try a different string in the middle of the search, you can delete the current input, type the new string, and start the search again.

Cutting and Pasting

You can cut or copy a piece of text that you select, from anywhere on the text window (ie., the main top window of Xmaxima); not only from the input lines but also from the output text in black. To select the text, you can drag the cursor with the mouse while you keep its left button depressed, or you can hold the **Shift** key with one finger, while you move the cursor with the mouse or with the arrow keys.

Once you have selected the text, you can either cut it, with **Edit, cut** or the shortcut **Ctrl+x**, **or** copy it to an internal buffer using **Edit, copy** or **Ctrl+c**. The text that has been cut or copied most recently can be pasted anywhere, even in the output fields, using **Edit, paste** or **Ctrl+v**.

There is a command similar to 'cut', called 'kill', accessed via either **Edit, kill** or **Ctrl+k**, with two major differences: it only works in input fields (blue or green) and instead of cutting a text that was selected, it will cut all the text from the cursor until the **end** of the input line.

The command **Edit, Clear input** or **Ctrl+u** is similar to **Edit, kill**, but it will only work on the last input line (ie the current input line) and will clear all from the beginning of that input line to the cursor position.

To paste the **last** text that you have cut with either ‘kill’ or ‘clear input’, you should use the ‘yank’ command **Edit, yank** or **Ctrl+y**. If you use the Clear Input command, **Ctrl+u**, you can immediately restore the line with the yank command **Ctrl+y** in a sort of “UnDo”.

Other Keyboard Shortcuts

Other useful key combinations are:
Ctrl+f, the same as **RightArrow**,
Ctrl+b, the same as **LeftArrow**,
Ctrl+p, the same as **UpArrow**,
Ctrl+n, the same as **DownArrow**,
Either **Ctrl+a** or **Home** moves to the left end of the current line (to the left of `(%xn)`),
Either **Ctrl+e** or **End** moves to the right end of the current line,
Ctrl+Home moves to the first character at the top of the text window,
Ctrl+End, moves to the last character at the bottom of the text window.

Save Xmaxima Session Record as a Text File

The menu command **Edit, Save Console to File** will bring up a dialog which allows you to select the folder and file name with the default extension **.out**, and Xmaxima will save the current session screen, as it appears to you, to a text file with that name. This can be a convenient way to keep a record of your work, particularly if you use the day’s date as part of the name. You can then open that text file with any text editor, such as **Notepad2**, and edit it as usual, and you can also change the name of the file. This session record is not in the form of inputs which you could use immediately with Xmaxima, although you could copy and paste command inputs one at a time into a later Xmaxima session.

Save All Xmaxima Inputs as Lisp Code for Later Use in Maxima

The menu command **File, Save Expressions to File** will open a dialog which will save every input as Lisp code with a file name like **sat.bin** or **sat.sav**. Although you can read such a file with a normal text editor, its main use to to rerun all the inputs in a later session by using **load(”sat.bin”)**, for example. All the variable assignments and function definitions will be absorbed by Maxima, but you don’t see any special output on the screen. Instead of that menu route, you could just type the input `save("C:/work2/sat1.bin",all);` to create the Lisp code record of session inputs.

Save All Xmaxima Inputs in a Batch File Using stringout

If you type the input `stringout("c:/work2/sat1.mac",input);`, the inputs will be saved in the form of Maxima syntax which can later be batched into a later Maxima session. In a later session, you type the input `batch("c:/work2/sat1.mac");` and on the screen will appear each input line as well as what each output is in detail. (Or you could use the menu route **File, Batch File**, which will open a dialog which lets you select the file).

Quiet Batch Input

If you use the menu route **File, Batch File Silently**, you will not see the record of inputs and outputs in detail, but all the bindings and definitions will be absorbed by Maxima. There will be no return value and no indication which file has been loaded, so you might prefer typing `load("c:/work2/sat1.mac");`, or `batchload("c:/work2/sat1.mac");` just to have that record in your work.

1.5 Creating and Using a Startup File: maxima-init.mac

You can create a startup file which will be read by Maxima at the start (or restart) of a new Maxima session. If you either have not already created your startup file or have not interactively changed the binding of **`maxima_userdir`**, you can find where Maxima expects to find a startup file as follows. We purposely start Xmaxima from a link to **..bin/xmaxima.exe** on the desktop, so Xmaxima has no clue where our work folder is.

```
(%i1) maxima_userdir;
(%o1)           C:/Documents and Settings/Edwin Woollett/maxima
```

Before we show the advantages of using your startup file, let's show what you need to do to load one of your ***.mac** files in your work folder into Maxima without that startup file helping out. In my work folder **c:/work2** is a file **qfft.mac**, (available with Ch. 11 material on the author's web page) and here is an effort to load the file in Xmaxima.

```
(%i2) load(qfft);
Could not find `qfft' using paths in file_search_maxima,file_search_lisp.
 -- an error.  To debug this try debugmode(true);
(%i3) load("qfft.mac");
Could not find `qfft.mac' using paths in file_search_maxima,file_search_lisp.
 -- an error.  To debug this try debugmode(true);
(%i4) load("c:/work2/qfft.mac");
 type qfft_syntax(); to see qfft and qift syntax
(%o4)                         c:/work2/qfft.mac
```

On the other hand, if we have a link to **..bin/xmaxima.exe** sitting in our work folder **c:/work2** and we start Xmaxima using that folder link we get

```
(%i2) load(qfft);
Could not find `qfft' using paths in file_search_maxima,file_search_lisp.
 -- an error.  To debug this try debugmode(true);
(%i3) load("qfft.mac");
 type qfft_syntax(); to see qfft and qift syntax
(%o3)                             qfft.mac
```

which shows one of the virtues of starting up Xmaxima using a link in your work folder.

Now that you know where Maxima is going to look for your startup file, make sure such a folder exists and create a file named **maxima-init.mac** in that folder. You can have Maxima display a simple message when it reads your startup file as an added check on what exactly is going on. For example, you could have the line **disp (”Hi, Cindy”)$**.

After saving the current version of that file in the correct folder, that message should appear after the Maxima version and credits message.

Below is the author's startup file, in which he has chosen to tell Maxima to look in the **c:/work2** folder for `*.mac` or `*.mc` files, as well as the usual search paths. The chapter 1 utility file **mbe1util.mac** is loaded into the session.

```
/* this is c:\Documents and Settings\Edwin Woollett\maxima\maxima-init.mac  */
/* last edit: 7-28-09  */
maxima_userdir: "c:/work2" $
maxima_tempdir : "c:/work2"$
file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$
file_search_lisp : append(["c:/work2/###.lisp"],file_search_lisp )$
load(mbe1util)$
print("   mbe1util.mac functions ", functions)$
disp("Maxima is the Future!")$
```

Naturally, you need to replace `c:/work2` by the path to your own work folder.

With this startup file in place, here is the opening screen of Maxima, using the link to Xmaxima in my work folder to start the session, and setting input **(%i1)** to be a request for the binding of **maxima_userdir**:

```
Maxima 5.19.0 http://maxima.sourceforge.net
Using Lisp GNU Common Lisp (GCL) GCL 2.6.8 (aka GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
   mbe1util.mac functions  [qplot(exprlist, prange, [hvrange]), rtt(e),
ts(e, v), to_atan(e, y, x), to_atan2(e, y, x), totan(e, v), totan2(e, v),
mstate(), mattrib(), mclean(), fll(x) ]
                             Maxima is the Future!

(%i1) maxima_userdir;
(%o1)                              c:/work2
```

Now it is easy to load in the package **qfft.mac**, and see the large increase in the number of user defined functons.

```
(%i2) load(qfft)$
(%i3) functions;
(%o3) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x), nyquist(ns, fs), sample(expr, var, ns, dvar), vf(flist, dvar),
current_small(), setsmall(val), _chop%(ex), fchop(expr), fchop1(expr, small),
_fabs%(e), kg(glist), spectrum1(glist, nlw, ymax),
spectrum(glist, nlw, ymax, [klim]), spectrum_eps(glist, nlw, ymax, fname,
[klim])]
```

We can use the utility function **mclean()** to remove those **qfft** functions.

```
(%i4) mclean();
----- clean start

(%o0)
(%i1) functions;
(%o1) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
```

If we had instead used **kill (all)** , the intial set of utilities loaded in from **mbe1util.mac** would have also vanished.

```
(%i2) kill(all);
(%o0)                                done
(%i1) functions;
(%o1)                                 []
```

Of course you could then use a separate **load(mbe1util)** command to get them back.

1.6 Maxima Expressions, Numbers, Operators, Constants and Reserved Words

The basic unit of information in Maxima is the expression. An expression is made up of a combination of operators, numbers, variables, and constants. Variables should have names which are not reserved words (see below), and can represent any type of data structue; there is no requirement to "declare" a variable to be of a certain type.

Numbers in Maxima

Maxima uses

- Integers, such as **123456**,
- Rational numbers, such as $\mathbf{3/2}$, ratios of integers,
- Floats and bigfloats such as **1.234**, **1.234e-6**, and **1.234b5**,
- Complex numbers, such as **`4 + 2*%i`** and **`a + b*%i`**. Maxima assumes the symbols **`a`** and **`b`** represent real numbers by default.

Operators in Maxima

The table below lists some Maxima operators in order of priority, from lowest to highest.
The input **`x^2+3`** means $\mathbf{x^2 + 3}$, and not $\mathbf{x^{2+3}}$. The exponentiation has higher precedence than addition. The input **`2^3^4`** stands for **`2^(3^4)`**. Parentheses can be used to force order of operations, or simply for clarity.

```
(%i1) x^2+3;
                                    2
(%o1)                              x  + 3
(%i2) 2^3^4;
(%o2)                     2417851639229258349412352
(%i3) 2^(3^4);
(%o3)                     2417851639229258349412352
(%i4) (2^3)^4;
(%o4)                               4096
```

Since the operator $*$ has precedence over $+$, $\mathbf{a + b * c}$ means $\mathbf{a + (b * c)}$, rather than $\mathbf{(a + b) * c}$.

Operator	Description
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$-$	negation
ˆ	exponentiation
.	non-commutative multiplication
ˆ ˆ	non-commutative exponentiation
!	factorial
!!	double factorial

Table 1: Operators in Order of Increasing Precedence

Constants in Maxima

The following table summarizes predefined constants.

Constant	Description
%e	Base of the natural logarithms $(\mathbf{e})$
%i	The square root of (-1) $(\mathbf{i})$
%pi	The transcendental constant pi $(\boldsymbol{\pi})$
%phi	The golden mean $(1+\sqrt{5})/2$
%gamma	The Euler-Mascheroni constant
inf	Real positive infinity (∞)
minf	Real negative infinity $(-\infty)$

Table 2: Maxima Predefined Constants

Here are the numerical values to 16 digit precision.

```
(%i5) float( [%e,%pi,%phi,%gamma] );
(%o5) [2.718281828459045, 3.141592653589793, 1.618033988749895,
                                                  0.57721566490153]
```

Reserved Words

There are a number of reserved words which, if used as variable or function names, might be confusing to both the user and Maxima. Their use might cause a possibly cryptic syntax error. Here are some of the "well known" and "less well known but short" reserved words Of course there are many other Maxima function names, global option variables, and

af	else	ic2	plog
and	elseif	if	psi
av	erf	ift	product
args	ev	ilt	put
array	exp	in	rat
at	f90	ind	rem
bc2	fft	inf	rk
carg	fib	inrt	some
cf	fix	integrate	step
cint	for	is	sum
col	from	li	then
cov	gcd	limit	thru
cv	gd	min	und
del	get	next	unless
diag	go	not	vers
diff	hav	op	while
do	ic1	or	zeta

Table 3: Some Simple Reserved Words

system option variables which you might also try to avoid when naming your own variables and function. One quick way to check on "name conflicts" is to keep the html Maxima help manual up in a separate window, and have the Index panel available to type in a name you want to use. The painful way to check on name conflicts is to wait for Maxima to give you a strange error message as to why what you are trying to do won't work. If you get strange results, try changing the names of your variables and or functions.

An important fact is that Maxima is case sensitive, so you can avoid name conflicts by capitalizing the names of your user defined Maxima functions. Your **Solve** will not conflict with Maxima's **solve**. This is a dumb example, but illustrates the principle:

```
(%i6) Solve(x):= x^2;
                                                  2
(%o6)                          Solve(x) := x
(%i7) Solve(3);
(%o7)                                 9
```

Of course, it takes more typing effort to use capitalized function names, which is why they are not popular among power users.

1.7 Input and Output Examples

In the following we are using the **Xmaxima** interface.

As discussed in Sec. 1.3.1, a rational expression is a ratio of polynomials or a sum of such. A special case is a polynomial. A rational expression is a special case of what is called an expression in Maxima.

Here we write a rational expression without binding it to any particular symbol.

```
(%i1) x/(x^3+1);
                                    x
(%o1)                             ------
                                   3
                                  x  + 1
```

The input line is typed in a "one dimensional" version and, if the input completion character is a semi-colon `;` then Xmaxima displays the expression in a text based two-dimensional notation. You can force one-dimensional output if you set **display2d** to **false**:

```
(%i2) display2d:false$
(%i3) %o1;
(%o3) x/(x^3+1)
(%i4) display2d:true$
(%i5) %o1;
                                    x
(%o5)                             ------
                                   3
                                  x  + 1
```

When you want to show a piece of code to the Maxima mailing list, it is recommended that you show the output in the one-dimensional form since otherwise, in the received message, exponents can appear in a shifted position which may be hard to interpret.

The output `(%o6)` is a symbolic expression which can be manipulated as a data structure. For example, you can add it to itself. The Maxima symbol `%` refers to the last output line.

```
(%i6) % + %;
                                   2 x
(%o6)                             ------
                                   3
                                  x  + 1
```

Notice the automatic simplification carried out by default. Maxima could have left the result as a sum of two terms, but instead recognised that the summands were identical and added them together producing a one term result. Maxima will automatically (in default behavior) perform many such simplifications.

Here is an example of automatic simplification of a trig function:

```
(%i7) sin(x - %pi/2);
(%o7)                            - cos(x)
```

1.8 Maxima Power Tools at Work

1.8.1 The Functions apropos and describe

Function apropos

`apropos("foo")` returns a list of core Maxima names which have `foo` appearing anywhere within them. For example, `apropos ("exp")` returns a list of all the core flags and functions which have **exp** as part of their names, such as **expand**, **exp**, and **ratexpand**. Thus if you can only remember part of the name of something, you can use this command to find the correct complete name. Here is an example:

```
(%i1) apropos ("exp");
(%o1) [askexp, auto_mexpr, besselexpand, beta_expand, cfexpand, comexp,
domxexpt, dotexptsimp, errexp, errexp1, errexp2, errexp3, exp, exp-form,
expand, expandwrt, expandwrt_denom, expandwrt_factored, expandwrt_nonrat,
expansion, expint, expintegral_chi, expintegral_ci, expintegral_e,
expintegral_e1, expintegral_ei, expintegral_hyp, expintegral_li,
expintegral_shi, expintegral_si, expintegral_trig, expintexpand, expintrep,
explicit, explose, expon, exponentialize, expop, expr, exprlist, expt,
exptdispflag, exptisolate, exptsubst, Expt, facexpand, factorial_expand,
gamma_expand, logexpand, macroexpand, macroexpand1, macroexpansion, matrixexp,
poisexpt, psexpand, radexpand, ratexpand, ratsimpexpons, sexplode,
solveexplicit, sumexpand, taylor_logexpand, texput, trigexpand,
trigexpandplus, trigexpandtimes, tr_exponent, tr_warn_fexpr]
```

Function describe

describe(e), describe(e, exact), describe(e, inexact)
describe(e) is equivalent to **describe(e, exact)** and prints to the screen the manual documentation of **e**.

describe(e, inexact) prints to the screen a numbered list of all items documented in the manual which contain “e” as part of their name. If there is more than one list element, Maxima asks the user to select an element or elements to display.

SHORTCUTS:
At the interactive prompt, `? foo` (with a space between `?` and `foo`) and NO ENDING SEMICOLON (just press Enter) is equivalent to either **describe(foo)** or **describe(foo, exact)**, and `?? foo` (with a space between `??` and `foo`) and NO ENDING SEMICOLON (just press Enter) is equivalent to **describe(foo, inexact)**.

In the latter case, the user will be asked to type either a set of space separated numbers to select some of the list elements, such as `2 3 5` followed by Enter, or the word `all` followed by Enter, or the word `none` followed by Enter.

Here is an example of interactive use of the single question mark shortcut.

```
(%i2) ? exp
 -- Function: exp (<x>)
     Represents the exponential function.  Instances of `exp (<x>)' in
     input are simplified to `%e^<x>'; `exp' does not appear in
     simplified expressions.

     `demoivre' if `true' causes `%e^(a + b %i)' to simplify to `%e^(a
     (cos(b) + %i sin(b)))' if `b' is free of `%i'. See `demoivre'.

     `%emode', when `true', causes `%e^(%pi %i x)' to be simplified.
     See `%emode'.

     `%enumer', when `true' causes `%e' to be replaced by 2.718...
     whenever `numer' is `true'. See `%enumer'.
```

```
  There are also some inexact matches for `exp'.
  Try `?? exp' to see them.

(%o2)                               true
```

1.8.2 The Function ev and the Properties evflag and evfun

Function ev

ev is a "jack-of-all-trades swiss army knife" which is frequently useful, occasionally dangerous, and complex to describe.

In brief, the syntax is

```
     ev ( expr, options );
        or more explicitly,
     ev (expr, arg_1, ..., arg_n)
```

with the interactive mode shortcut

```
     expr, options ;
       or again more explicitly
     expr, arg_1, ..., arg_n ;
```

The interactive mode shortcut form cannot be used inside user defined Maxima functions or blocks.

The Manual description of **ev** begins

```
Evaluates the expression expr in the environment specified by the
  arguments arg_1, ..., arg_n. The arguments are switches (Boolean flags),
  assignments, equations, and [Maxima] functions. ev returns the result
  (another expression) of the evaluation.
```

One option has the form `V : e`, or `V = e`, which causes `V` to be bound to the value of `e` during the evaluation of `expr`. If more than one argument to **ev** is of this type, then the binding is done in parallel, as shown in the following example.

```
(%i1) x+y, x  =  a+y;
(%o1)                              2 y + a
(%i2) %, y = 2;
(%o2)                               a + 4
(%i3) x+y, x = a+y, y = 2;
(%o3)                             y + a + 2
(%i4) x+y, [x = a+y, y = 2];
(%o4)                             y + a + 2
```

If `V` is a non-atomic expression, then a substitution rather than a binding is performed.

This example illustrates the subtlety of the way **ev** in designed to work, and shows that some experimentation should be carried out to gain confidence in the result returned by **ev**.

The next example of **ev** shows its use to check the correctness of solutions returned by **solve**.

```
(%i1) eqns : [-2*x -3*y = 3, -3*x +2*y = -4]$
(%i2) solns : solve (eqns);
                                   17      6
(%o2)                       [[y = - --, x = --]]
                                   13      13
(%i3) eqns, solns;
(%o3)                        [3 = 3, - 4 = - 4]
```

Our final example shows the use of **rectform** and **ratsimp** to make explicit the fourth roots of -1 returned by **solve**.

```
(%i1) solve ( a^4 + 1 );
                 1/4                 1/4              1/4               1/4
(%o1) [a = (- 1)    %i, a = - (- 1)   , a = - (- 1)    %i, a = (- 1)    ]
(%i2) % , rectform, ratsimp;
           sqrt(2) %i - sqrt(2)        sqrt(2) %i + sqrt(2)
(%o2) [a = --------------------, a = - --------------------,
                    2                           2
                                 sqrt(2) %i - sqrt(2)      sqrt(2) %i + sqrt(2)
                           a = - --------------------, a = --------------------]
                                          2                         2
(%i3) %^4, ratsimp;
                      4         4         4         4
(%o3)              [a  = - 1, a  = - 1, a  = - 1, a  = - 1]
```

Both **rectform** and **ratsimp** are Maxima functions which have the property **evfun** (see next entry), which means that **`ev(expr, rectform, ratsimp)`** is equivalent to **`ratsimp ( rectform ( ev(expr) ) )`** .

Property evflag

When a symbol **`x`** has the **evflag** property, the expressions **`ev(expr, x)`** and **`expr, x`** (at the interactive prompt) are equivalent to **`ev(expr, x = true)`**. That is, **`x`** is bound to **true** while **`expr`** is evaluated.
The expression **`declare(x, evflag)`** gives the **evflag** property to the variable **`x`**.
The flags which have the **evflag** property by default are the following:

```
algebraic, cauchysum, demoivre, dotscrules, %emode, %enumer, exponentialize,
exptisolate, factorflag, float, halfangles, infeval, isolate_wrt_times,
keepfloat, letrat, listarith, logabs, logarc, logexpand, lognegint, lognumer,
m1pbranch, numer_pbranch, programmode, radexpand, ratalgdenom, ratfac,
ratmx, ratsimpexpons, simp, simpsum, sumexpand, and trigexpand.
```

Even though the Boolean switch **numer** does not have the property **evflag**, it can be used as if it does.

```
(%i4) [exponentialize,float,numer,simp];
(%o4)                     [false, false, false, true]
(%i5) properties(exponentialize);
(%o5) [system value, transfun, transfun, transfun, transfun, transfun,
                                                              transfun, evflag]
(%i6) properties(numer);
(%o6)                   [system value, assign property]
(%i7) properties(float);
(%o7) [system value, transfun, transfun, transfun, transfun, transfun,
                                                    evflag, transfun, transfun]
(%i8) properties(simp);
(%o8)                       [system value, evflag]
```

Here are some examples of use.

```
(%i9) [ ev (exp(3/29), numer ), ev (exp(3/29), float) ];
(%o9)              [1.108988430411017, 1.108988430411017]
(%i10) [ ev (exp (%pi*3/29), numer), ev (exp (%pi*3/29), float) ];
                                        0.10344827586207 %pi
(%o10)             [1.384020049155809, %e                    ]
(%i11) 2*cos(w*t) + 3*sin(w*t), exponentialize, expand;
                     %i t w                      - %i t w
            3 %i %e            %i t w   3 %i %e              - %i t w
(%o11)    - ------------- + %e        + --------------- + %e
                  2                            2
```

(For the use of the “key word” **expand** in this context, see below.)

Property evfun

When a Maxima function `F` has the **evfun** property, the expressions `ev(expr, F)` and `expr, F` (at the interactive prompt) are equivalent to `F ( ev (expr))`.
If two or more **evfun** functions `F`, `G`, etc., are specified, then **ev (expr, F, G)** is equivalent to **G (F (ev(expr)))**.

The command `declare(F, evfun)` gives the **evfun** property to the function `F`.
The functions which have the **evfun** property by default are the following very useful and single argument funtions: **bfloat, factor, fullratsimp, logcontract**, **polarform, radcan, ratexpand, ratsimp, rectform**, **rootscontract, trigexpand, and trigreduce**.

Note that **rat**, and **trigsimp** do not, by default, have the property **evfun**.

Some of the other "key words" which can be used with **ev** are **expand, nouns, diff, integrate** (even though they are not obviously boolean switches and do not have the property **evflag**).

```
(%i12) [diff,expand,integrate,nouns];
(%o12)                  [diff, expand, integrate, nouns]
(%i13) properties(expand);
(%o13)         [transfun, transfun, transfun, transfun, transfun]
(%i14) (a+b)*(c+d);
(%o14)                           (b + a) (d + c)
(%i15) (a+b)*(c+d),expand;
(%o15)                        b d + a d + b c + a c
(%i16) ev((a+b)*(c+d),expand);
(%o16)                        b d + a d + b c + a c
```

The undocumented property **transfun** apparently has something to do with translation to Lisp.

1.8.3 The List functions and the Function fundef

functions

functions is the name of a list maintained by Maxima and this list is printed to the screen (as would any list) with the names of the current available non-core Maxima functions (either user defined interactively or defined by any packages loaded into the current session). Here is the example we saw previously based on the author's startup file:

```
(%i1) functions;
(%o1) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
```

fundef

fundef(name) prints out the definition of a non-core Maxima function (if currently known by Maxima). Here is an example related to the previous example which said that Maxima knew about a function called **rtt**.

```
(%i2) fundef(rtt);
(%o2)               rtt(e) := radcan(trigrat(trigsimp(e)))
(%i3) fundef(cos);
cos is not the name of a user function.
 -- an error.  To debug this try debugmode(true);
```

1.8.4 The Function kill and the List values

kill

kill(a,b) will eliminate the objects **a** and **b**. Special cases are **kill(all)** and **kill(allbut(x,y))**. Here is an example which removes our user defined Maxima function **rtt**.

```
(%i1) kill(rtt);
(%o1)                                done
(%i2) functions;
(%o2) [qplot(exprlist, prange, [hvrange]), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
```

values

values is another list maintained by Maxima which contains the names of currently assigned scalar values which have been set by the user interactively or by packages which have been loaded. Here is a simple example.

```
(%i3) [ a:2, b:5, e: x^2/3 ];
                                          2
                                         x
(%o3)                           [2, 5, --]
                                         3
(%i4) values;
(%o4)                            [a, b, e]
```

1.8.5 Examples of map, fullmap, apply, grind, and args

Let **f** be an unbound symbol. When **f** is then mapped onto a list or expression, we can see what will happen if we **map** a Maxima function on to the same kind of object.

```
(%i1) map('f, [x, y, z] );
(%o1)                       [f(x), f(y), f(z)]
(%i2) map('f, x + y + z);
(%o2)                       f(z) + f(y) + f(x)
(%i3) map('f, [a*x,b*exp(y),c*log(z)]);
                                      y
(%o3)                 [f(a x), f(b %e ), f(c log(z))]
(%i4) map('f, a*x + b*exp(y) + c*log(z) );
                                           y
(%o4)                 f(c log(z)) + f(b %e ) + f(a x)
```

The use of **map** with the Maxima function **ratsimp** allows a separate simplification to be carried out on each term of the following expression.

```
(%i5) e : x/(x^2+x)+(y^2+y)/y ;
                                 2
                                y  + y     x
(%o5)                           ------ + ------
                                  y       2
                                         x  + x
(%i6) e, ratsimp;
                               (x + 1) y + x + 2
(%o6)                          -----------------
                                     x + 1
(%i7) map('ratsimp,e);
                                       1
(%o7)                           y + ----- + 1
                                    x + 1
```

We create a list of equations from two lists using **map**.

```
(%i8) map( "=", [x,y,z],[a,b,c] );
(%o8)                        [x = a, y = b, z = c]
```

We compare **map** and **fullmap** when appied to an expression.

```
(%i9) expr : 2*%pi + 3*exp(-4);
                                                - 4
(%o9)                           2 %pi + 3 %e
(%i10) map('f, expr);
                                                  - 4
(%o10)                        f(2 %pi) + f(3 %e    )
(%i11) fullmap('f, expr);
                                                    f(- 4)
(%o11)                   f(2) f(%pi) + f(3) f(%e)
```

apply

The Maxima function **apply** can be used with a list only (not an expression).

```
(%i12) apply('f, [x,y,z]);
(%o12)                            f(x, y, z)
(%i13) apply("+",[x,y,z]);
(%o13)                             z + y + x
(%i14) dataL : [ [1,2], [2,4] ]$
(%i15) dataM : apply('matrix, dataL );
                                   [ 1  2 ]
(%o15)                              [      ]
                                   [ 2  4 ]
(%i16) grind(%)$
matrix([1,2],[2,4])$
```

The function **grind** allows one dimensional display suitable for copying into another input line. The last example above created a Maxima **matrix** object from a nested list. Maxima has many tools to work with **matrix** objects. To achieve one dimensional display of such **matrix** objects (saving space), you can use `display2d:false`. To go from a Maxima **matrix** object to a nested list, use **args**.

```
(%i17) args(dataM);
(%o17)                          [[1, 2], [2, 4]]
```

1.8.6 Examples of subst, ratsubst, part, and substpart

subst

To replace `x` by `a` in an expression `expr`, you can use either of two forms.

```
   subst ( a, x, expr );
     or
   subst ( x = a, expr );
```

To replace `x` by `a` and also `y` by `b` one can use

```
  subst ( [ x = a, y = b ], expr );
```

`x` and `y` must be atoms (ie., a number, a string, or a symbol) or a complete subexpression of `expr`. When `x` does not have these characteristics, use `ratsubst( a, x, expr )` instead.

Some examples:

```
(%i1) e : f*x^3 + g*cos(x);
                                                        3
(%o1)                          g cos(x) + f x
(%i2) subst ( x = a, e );
                                                      3
(%o2)                          cos(a) g + a  f
(%i3) e1 : subst ( x = a + b, e );
                                                           3
(%o3)                     cos(b + a) g + (b + a)  f
(%i4) e2 : subst ( a + b = y, e1 );
                                                        3
(%o4)                          g cos(y) + f y
(%i5) e3 : f*x^3 + g*cos(y);
                                                        3
(%o5)                          g cos(y) + f x
(%i6) e4 : subst ( [x = a+b, y = c+d],e3 );
                                                           3
(%o6)                     cos(d + c) g + (b + a)  f
(%i7) subst ([a+b = r, c+d = p],e4 );
                                           3
(%o7)                          f r  + g cos(p)
```

ratsubst

```
(%i8) e : a*f(y) + b*g(x);
(%o8)                          a f(y) + b g(x)
(%i9) e1 : ratsubst( cos(y),f(y),e );
(%o9)                         a cos(y) + b g(x)
(%i10) e2 : ratsubst( x^3*sin(x),g(x),e1 );
                                                  3
(%o10)                      a cos(y) + b x  sin(x)
```

part and substpart

The Maxima functions **part** and **substpart** are best defined by a simple example.

```
(%i11) e : a*log(f(y))/(b*exp(f(y)));
                                   - f(y)
                               a %e       log(f(y))
(%o11)                          --------------------
                                         b
(%i12) length(e);
(%o12)                                     2
(%i13) [part(e,0),part(e,1),part(e,2)];
                                   - f(y)
(%o13)                     [/, a %e       log(f(y)), b]
(%i14) substpart("+",e,0);
                                 - f(y)
(%o14)                      a %e       log(f(y)) + b
(%i15) length(part(e,1));
(%o15)                                     3
(%i16) [part(e,1,0),part(e,1,1),part(e,1,2),part(e,1,3)];
                                     - f(y)
(%o16)                      [*, a, %e      , log(f(y))]
(%i17) length(part(e,1,3));
(%o17)                                     1
(%i18) [part(e,1,3,0),part(e,1,3,1)];
(%o18)                                [log, f(y)]
```

```
(%i19) substpart(sin,e,1,3,0);
                                      - f(y)
                             a %e       sin(f(y))
(%o19)                       --------------------
                                      b
(%i20) length( part(e,1,3,1) );
(%o20)                                 1
(%i21) [part(e,1,3,1,0), part(e,1,3,1,1)];
(%o21)                              [f, y]
(%i22) substpart(x,e,1,3,1,1);
                                                - f(y)
                             a log(f(x)) %e
(%o22)                       --------------------
                                      b
```

By judicious use of **part** and **substpart**, we see that we can modify a given expression in all possible ways.

1.8.7 Examples of coeff, ratcoef, and collectterms

collectterms

Use the syntax

```
    collectterms ( expr, arg1, arg2, ...)
```

This Maxima function is best explained with an example. Consider the expression:

```
(%i1) ex1 :  a1*(b + c/2)^2 + a2*(d + e/3)^3 , expand;
              3          2                          2
         a2 e     a2 d e         2         3    a1 c                2
(%o1)    ----- + ------- + a2 d  e + a2 d  + ----- + a1 b c + a1 b
          27        3                             4
```

How can we return this expanded expression to the original form? We first use **collectterms**, and then **factor** with **map**.

```
(%i2) collectterms(ex1,a1,a2);
                       3      2                          2
                      e    d e     2      3          c           2
(%o2)            a2  (-- + ---- + d  e + d ) + a1  (-- + b c + b )
                      27    3                        4
(%i3) map('factor, %);
                                          3                2
                           a2  (e + 3 d)     a1  (c + 2 b)
(%o3)                      ------------- + -------------
                               27                4
```

Maxima's core simplification rules prevent us from getting the $3^3 = 27$ into the numerator of the first term, and also from getting the $2^2 = 4$ into the numerator of the second term, unless we are willing to do a lot of **substpart** piecework (usually not worth the trouble).

coeff

The syntax

```
     coeff ( expr, x, 3 )
```

will return the coefficient of `x^3` in `expr`.

The syntax

```
    coeff ( expr, x )
```

will return the coefficient of **x** in **expr**.
x may be an atom or a complete subexpression of **expr**, such as **sin(y)**, **a[j]**, or **(a+b)**. Sometimes it may be necessary to expand or factor **expr** in order to make **x^n** explicit. This preparation is not done automatically by **coeff**.

```
(%i4) coeff(%, a2);
                                               3
                                     (e + 3 d)
(%o4)                                ----------
                                         27
(%i5) coeff(%, e + 3*d, 3);
                                         1
(%o5)                                    --
                                         27
```

ratcoef (also ratcoeff)

The function **ratcoef** (or **ratcoeff**) has the same syntax as **coeff** (except that **n** should not be negative), but expands and rationally simplifies the expression before finding the coefficient, and thus can produce answers different from **coeff**, which is purely syntactic.

```
(%i6) ex2 : (a*x + b)^2;
                                               2
(%o6)                                (a x + b)
(%i7) coeff(ex2,x);
(%o7)                                    0
(%i8) ratcoeff(ex2,x);
(%o8)                                  2 a b
(%i9) ratcoef(ex2,x);
(%o9)                                  2 a b
(%i10) ratcoeff(ex2, x, 0);
                                         2
(%o10)                                   b
(%i11) ratcoef(ex2, x, 0);
                                         2
(%o11)                                   b
```

Both **coeff** and **ratcoef** can be used with equations (as well as expressions).

```
(%i12) eqn : (a*sin(x) + b*cos(x))^3 = c*sin(x)*cos(x);
                                            3
(%o12)              (a sin(x) + b cos(x))   = c cos(x) sin(x)
(%i13) ratcoef(eqn,sin(x),0);
                                    3     3
(%o13)                             b   cos  (x) = 0
(%i14) ratcoef(eqn,sin(x));
                                  2     2
(%o14)                      3 a b   cos  (x) = c cos(x)
```

1.8.8 Examples of rat, diff, ratdiff, ratexpand, expand, factor, gfactor and partfrac

Maxima provides tools which allow the user to write an expression in multiple forms. Consider the tenth order polynomial in x given by

```
(%i1) e:(x + 3)^10;
                                              10
(%o1)                                  (x + 3)
```

We can use **diff (expr, x)** to find the first derivative of **expr**, and **diff (expr, x, 2)** to find the second derivative of **expr**, and so on.

```
(%i2) de1 : diff ( e, x );
                                                9
(%o2)                               10 (x + 3)
```

Because this expression is a polynomial in **x**, we can also use **ratdiff (expr, x)**:

```
(%i3) de1r : ratdiff ( e, x );
         9        8         7          6           5           4          3
(%o3) 10 x  + 270 x  + 3240 x  + 22680 x  + 102060 x  + 306180 x  + 612360 x
                                                               2
                                                + 787320 x  + 590490 x + 196830
(%i4) factor ( de1r );
                                                9
(%o4)                               10 (x + 3)
```

We show three tools for expansion of this polynomial.

```
(%i5) rat(e);
         10       9        8         7          6          5           4
(%o5)/R/ x   + 30 x  + 405 x  + 3240 x  + 17010 x  + 61236 x  + 153090 x
                                                  3          2
                                    + 262440 x  + 295245 x  + 196830 x + 59049
(%i6) ratexpand (e);
      10       9        8         7          6          5           4
(%o6) x   + 30 x  + 405 x  + 3240 x  + 17010 x  + 61236 x  + 153090 x
                                                  3          2
                                    + 262440 x  + 295245 x  + 196830 x + 59049
(%i7) expand (e);
      10       9        8         7          6          5           4
(%o7) x   + 30 x  + 405 x  + 3240 x  + 17010 x  + 61236 x  + 153090 x
                                                  3          2
                                    + 262440 x  + 295245 x  + 196830 x + 59049
(%i8) factor (%);
                                              10
(%o8)                                  (x + 3)
```

If you want the lowest powers displayed first, you have to change the setting of **powerdisp** to **true**.

```
(%i9) powerdisp;
(%o9)                                    false
(%i10) powerdisp : true$
(%i11) %o7;
                                   2           3           4          5
(%o11) 59049 + 196830 x + 295245 x  + 262440 x  + 153090 x  + 61236 x
                                                6         7        8       9    10
                                   + 17010 x  + 3240 x  + 405 x  + 30 x  + x
(%i12) powerdisp : false$
```

Next we consider a sum of rational expressions:

```
(%i13) expr: (x - 1)/(x + 1)^2 + 1/(x - 1);
                                x - 1       1
(%o13)                         -------- + -----
                                      2   x - 1
                               (x + 1)
(%i14) expand ( expr );
                           x              1           1
(%o14)                ------------ - ------------ + -----
                       2              2             x - 1
                      x  + 2 x + 1   x  + 2 x + 1
(%i15) ratexpand ( expr );
                               2
                            2 x                 2
(%o15)                 --------------- + ---------------
                        3    2            3    2
                       x  + x  - x - 1   x  + x  - x - 1
```

We see that **ratexpand** wrote all fractions with a common denominator, whereas **expand** did not. In general, **rat** and **ratexpand** is more efficient at expanding rational expressions. Here is a very large expression with many **%pi**'s which can be simplified easily by the "rat tribe", but **expand** takes a very long time and returns a mess.

```
(%i16) e :
   ((2/%pi-1)*(((%pi/2-1)/(%pi-1)-1)*((6*(2/%pi-2*(4-%pi)/%pi)/%pi-(6*(2*(4-%pi)
     /%pi-(%pi-2)/(%pi/2-1))/(%pi-1)-6*((%pi-2)/(%pi/2-1)-2)*(%pi/2-1)
     /(%pi*(%pi-1)))/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1)) +2)))
      /((((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))+2)
     -(6*(2*(2*%pi-5)/%pi-2/%pi)/%pi-(6*(2/%pi-2*(4-%pi)/%pi)/%pi
     -(6*(2*(4-%pi)/%pi-(%pi-2)/(%pi/2-1))/(%pi-1)-6*((%pi-2)/(%pi/2-1)-2)
     *(%pi/2-1)/(%pi*(%pi-1)))/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2)))
     /(2*(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))+2)))
     /(2*(2-1/(4*(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))
     +2)))*(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))
     +2))+2)))/((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2)+(6*(2*(4-%pi)/%pi-(%pi-2)
     /(%pi/2-1))/(%pi-1)-6*((%pi-2)/(%pi/2-1)-2)*(%pi/2-1)/(%pi*(%pi-1)))
     /((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))/2+6*((%pi-2)/(%pi/2-1)-2)/%pi)
     *(x^3-x)/6  + 2*x$
(%i17) ratsimp (e);
              2                   3           3          2
      (128 %pi  - 520 %pi + 528) x  + (194 %pi  - 248 %pi  + 400 %pi - 528) x
(%o17)-----------------------------------------------------------------------
                                  3         2
                            97 %pi  - 60 %pi  - 60 %pi
(%i18) rat (e);
(%o18)/R/
                2                   3           3          2
        (128 %pi  - 520 %pi + 528) x  + (194 %pi  - 248 %pi  + 400 %pi - 528) x
        -----------------------------------------------------------------------
                                    3         2
                              97 %pi  - 60 %pi  - 60 %pi
```

```
(%i19) ratexpand (e);
                          3                            3
                528 x                      128 %pi x
(%o19) -------------------------- + ---------------------
                3         2                   2
          97 %pi  - 60 %pi  - 60 %pi   97 %pi  - 60 %pi - 60
                    3                                                  2
             520 x                        528 x                 194 %pi  x
 - --------------------- - -------------------------- + ---------------------
              2                      3         2                  2
       97 %pi  - 60 %pi - 60   97 %pi  - 60 %pi  - 60 %pi   97 %pi  - 60 %pi - 60
             248 %pi x                  400 x
 - --------------------- + ---------------------
              2                      2
       97 %pi  - 60 %pi - 60   97 %pi  - 60 %pi - 60
```

There is a more general form **expand (expr, p, n)** available to control which parts of fractions are expanded and how (see the Manual index entry for **expand**). There is also a more general form for `rat (expr, x_1, ..., x_n)`.

gfactor

The "g" in **gfactor** comes from factorization over the Gaussian integers.

```
(%i20) gfactor ( x^2 + 1 );
(%o20)                          (x - %i) (x + %i)
```

partfrac

We have earlier emphasized the usefulness of **ratsimp**. A Maxima function which in some cases can be considered the "opposite" of **ratsimp** in its results is **partfrac**, which has the syntax `partfrac ( expr, var )`, and which expands `expr` in partial fractions with respect to `var`. Here is the example presented in the Manual.

```
(%i21) e : 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
                                2       2        1
(%o21)                        ----- - ----- + --------
                              x + 2   x + 1          2
                                               (x + 1)
(%i22) e, ratsimp;
                                             x
(%o22)                          - --------------------
                                     3      2
                                    x  + 4 x  + 5 x + 2
(%i23) partfrac (%, x);
                                2       2        1
(%o23)                        ----- - ----- + --------
                              x + 2   x + 1          2
                                               (x + 1)
```

1.8.9 Examples of integrate, assume, facts, and forget

An important symbolic tool is indefinite integration, much of which is performed algorithmically. The original integration package, written by Joel Moses, incorporates the non-algebraic case of the Risch integration algorithm, and the package is called **sin.lisp** ("sin" is a mnemonic from "Symbolic INtegrator").

Consider the indefinite integral of a rational expression:

```
(%i1) e : x/(x^3 + 1);
                                    x
(%o1)                             ------
                                   3
                                  x  + 1
(%i2) ie : integrate (e, x);
                                      2 x - 1
                  2              atan(-------)
             log(x  - x + 1)          sqrt(3)    log(x + 1)
(%o2)        --------------- + ------------- - ----------
                    6              sqrt(3)         3
```

Indefinite integration can usually be checked by differentiating the result:

```
(%i3) diff (ie, x);
                    2                2 x - 1            1
(%o3)       ------------------ + -------------- - ---------
                        2              2            3 (x + 1)
              (2 x - 1)          6 (x  - x + 1)
            3 (---------- + 1)
                   3
```

This answer, as it stands, is not identical to the starting integrand. The reason is that Maxima merely differentiates term by term. Maxima automatically applies only those simplification rules which are considered "obvious" and "always desirable". A particular simplification rule might make one expression smaller but might make another expression larger. In such cases, Maxima leaves the decision to the user.

In this example, the starting integrand can be recovered by "rational simplification" using the **ratsimp** function.

```
(%i4) %, ratsimp;
                                    x
(%o4)                             ------
                                   3
                                  x  + 1
```

We met **ratsimp** and **fullratsimp** (Sec. 1.3.1) in our brief discussion of the buttons and menu system of **wxMaxima**, where the **Simplify** button made use of **ratsimp**, a reflection of the frequent use you will make of this function. You can think of the word **ratsimp** being a mnenomic using the capitalised letters in RATional SIMPlification.

If you try to use **ratdiff** on the above indefinite integration result, you get an obscure error message:

```
(%i5) ratdiff (ie, x);
`ratdiff' variable is embedded in kernel
 -- an error.  To debug this try debugmode(true);
```

The error message is returned because **ratdiff (expr, x)** can only be used for expressions which are either a polynomial in **x** or a ratio of polynomials in **x**.

integrate consults the **assume** database when making algebraic decisions. The content of that database is returned by **`facts();`**. You can sometimes avoid questions from **integrate** if you first provide some assumptions with **assume**. Here is an example of a definite integral in which we start with no assumptions, and need to answer a question from **integrate**.

```
(%i1) facts();
(%o1)                               []
(%i2) integrate(x*exp(-a*x)*cos(w*x),x,0,inf);
Is  a  positive, negative, or zero?

p;
                                     2    2
                                    w  - a
(%o2)                          - -----------------
                                  4      2  2    4
                                 w  + 2 a  w  + a
```

We can now tell Maxima that the parameter **`a`** should be treated as a positive number. We thereby avoid the question.

```
(%i3) ( assume(a > 0), facts() );
(%o3)                             [a > 0]
(%i4) integrate(x*exp(-a*x)*cos(w*x),x,0,inf);
                                     2    2
                                    w  - a
(%o4)                          - -----------------
                                  4      2  2    4
                                 w  + 2 a  w  + a
```

You can now tell Maxima to **forget** that assumption.

```
(%i5) (forget(a > 0), facts() );
(%o5)                               []
```

1.8.10 Numerical Integration and Evaluation: float, bfloat, and quad_qags

Let's try a harder integral, this time a definite integral which is expressed by Maxima in terms of a "special function".

```
(%i1) is : integrate( exp(x^3),x,1,2 );
                                  1                          1
      (sqrt(3) %i - 1) (gamma_incomplete(-, - 8) - gamma_incomplete(-, - 1))
                                  3                          3
(%o1) ----------------------------------------------------------------------
                                 6
(%i2) float(is);
(%o2) 0.16666666666667 (- 719.2849028287006 %i
 - 1.0 (0.66608190774162 - 3.486369946257051 %i) - 412.6003937374765)
 (1.732050807568877 %i - 1.0)
(%i3) expand(%);
(%o3)                        275.5109837634787
(%i4) ival:%;
(%o4)                        275.5109837634787
```

You can look up the **`gamma_incomplete(a,z)`** definition in the Maxima manual, where you will find a reference to the incomplete upper gamma function **A&S 6.5.2**, which refers to Eq. 6.5.2 in **Handbook of Mathematical Functions**, Edited by Milton Abramowitz and Irene A. Stegun, Dover Publications, N.Y., 1965.

We know that since we are integrating along the real x axis from 1 to 2, our integrand in **`(%i1)`** is real, so that the numerical value of this integral must be a real number. In using **float**, we see that the real answer will be obtained by cancellation of imaginary pieces, so we may have some roundoff errors.

Let's check the accuracy of using **float** here by comparing that answer to the answer returned by the use of **bfloat** together with **fpprec** set to 20.

```
(%i5) bfloat(is),fpprec:20;
(%o5) 1.6666666666666666667b-1 (1.7320508075688772935b0 %i - 1.0b0)
 (- 1.0b0 (6.6608190774161802181b-1 - 3.4863699462570511861b0 %i)
 - 7.1928490282826659929b2 %i - 4.1260039373722578387b2)
(%i6) expand(%),fpprec:20;
(%o6)       5.7824115865893569814b-19 %i + 2.7551098376331160126b2
(%i7) tval20: realpart(%);
(%o7)                      2.7551098376331160126b2
(%i8) abs(ival - tval20);
(%o8)                       1.670912297413452b-10
```

We see that the numerical answer found using **float** only has twelve digit accuracy. Let's look at the numerical values of the `gamma_incomplete` functions involved here as returned by **float**:

```
(%i9) float(gamma_incomplete(1/3,-8));
(%o9)            - 719.2849028287006 %i - 412.6003937374765
(%i10) float(gamma_incomplete(1/3,-1));
(%o10)             0.66608190774162 - 3.486369946257051 %i
```

Finally, let's use a pure numerical integration routine , `quad_qags`, which returns a list consisting of `[answer, est-error, num-integrand-eval, error-code]`.

```
(%i11) quad_qags(exp(x^3),x,1,2);
(%o11)         [275.5109837633116, 3.2305615936931465E-7, 21, 0]
(%i12) abs(first(%) - tval20);
(%o12)                      2.842170943040401b-14
```

and we see that the use of `quad_qags` for a numerical value was more accurate than the comination of **integrate** and **float**.

1.8.11 Taylor and Laurent Series Expansions with taylor

When computing exact symbolic answers is intractable, one can often resort to series approximations to get approximate symbolic results. Maxima has an excellent Taylor series program. As an example, we can get truncated series representations for **sin(x)** and **cos(x)** as follows. According to the Maxima manual

> **taylor (expr, x, a, n)** expands the expression **expr** in a truncated Taylor or Laurent series in the variable **x** around the point $\mathbf{x = a}$, containing terms through $\mathbf{(x - a)^n}$.

Here we expand around the point $\mathbf{x = 0}$.

```
(%i1) taylor( sin(x),x,0,5 );
                                       3    5
                                      x    x
(%o1)/T/                         x - -- + --- + . . .
                                      6    120
(%i2) taylor( cos(x),x,0,5);
                                       2    4
                                      x    x
(%o2)/T/                          1 - -- + -- + . . .
                                      2    24
```

A truncated series in denoted by the "`/T/`" symbol next to the line label and also by the trailing dots. Maxima retains certain information about such expansions, such as the "quality" of the approximation, so when several series expansions are combined, no terms are computed beyond the degree of the approximation. In our example, our starting expansions are only good through terms of order $\mathbf{x^5}$, so if we multiply the expansions, any terms smaller (in order of magnitude) than

x^5 are neglected.

Here we use **`%th(n)`**, which refers to the n'th to last line.

```
(%i3) % * %th(2);
                                   3      5
                                2 x    2 x
(%o3)/T/                    x - ---- + ---- + . . .
                                 3      15
```

The name "taylor" is only an artifact of history, since the Maxima function **taylor** can handle expansions of functions with poles and branch points and automatically generates Laurent series when appropriate as shown here:

```
(%i4) taylor( 1/ (cos(x) - sec(x))^3, x,0,5 );
                                                 2          4
            1     1       11      347    6767 x    15377 x
(%o4)/T/  - -- + ---- + ------ - ----- - ------- - -------- + . . .
             6      4        2   15120   604800    7983360
            x    2 x    120 x
```

1.8.12 Solving Equations: solve, allroots, realroots, and find_root

The **solve** function is described in the Maxima manual

> Function: **solve (expr, x)**
> Function: **solve (expr)**
> Function: **solve `([eqn_1, ..., eqn_n], [x_1, ..., x_n])`**
> Solves the algebraic equation **expr** for the variable **x** and returns a list of solution equations for **x**. If **expr** is not an equation, the equation **expr** $= \mathbf{0}$ is assumed in its place. **x** may be a function (e.g. f(x)), or other non-atomic expression except a sum or product. **x** may be omitted if **expr** contains only one variable. **expr** may be a rational expression, and may contain trigonometric functions, exponentials, etc.

Example 1

Here we look for solutions of the equation $\mathbf{x^6} = \mathbf{1}$ or $\mathbf{x^6} - \mathbf{1} = \mathbf{0}$. Once we have found candidate solutions, we check them one at a time.

```
(%i1) eqn : x^6 - 1 = 0$
(%i2) solns : solve(eqn);
          sqrt(3) %i + 1      sqrt(3) %i - 1                sqrt(3) %i + 1
(%o2) [x = --------------, x = --------------, x = - 1, x = - --------------,
                2                   2                              2
                                                          sqrt(3) %i - 1
                                                    x = - --------------, x = 1]
                                                                2
(%i3) for i thru length(solns) do
        disp ( ev( eqn, solns[i], ratsimp ) )$
                                        0 = 0
                                        0 = 0
                                        0 = 0
                                        0 = 0
                                        0 = 0
                                        0 = 0
```

One often uses **solve** in the hope of finding useful symbolic expressions for roots of an equation. It may still be illuminating to look at the numerical values of the roots returned by the polynomial root finder **allroots**, which we do for the above example:

```
(%i4) fpprintprec:8$
(%i5) nsolns : allroots(x^6 - 1);
(%o5) [x = 0.866025 %i + 0.5, x = 0.5 - 0.866025 %i, x = 0.866025 %i - 0.5,
                                    x = - 0.866025 %i - 0.5, x = 1.0, x = - 1.0]
(%i6) for i thru length(nsolns) do
        disp ( ev( x^6 - 1, nsolns[i],expand ))$
                      - 4.4408921E-16 %i - 5.55111512E-16
                       4.4408921E-16 %i - 5.55111512E-16
                       4.4408921E-16 %i - 5.55111512E-16
                      - 4.4408921E-16 %i - 5.55111512E-16
                                         0.0
                                         0.0
```

The numerical roots found by **allroots** satisfy the starting equation to within floating point errors.

Example 2

You can use **solve** to look for solutions to a set of equations. Here we find four sets of solutions satisfying two simultaneous equations, and we check the solutions.

```
(%i1) fpprintprec:8$
(%i2) eqns : [4*x^2 - y^2 - 12 = 0, x*y - x - 2 = 0]$
(%i3) solns : solve(eqns,[x,y]);
(%o3) [[x = 2, y = 2], [x = 0.520259 %i - 0.133124,
y = 0.0767838 - 3.6080032 %i], [x = - 0.520259 %i - 0.133124,
y = 3.6080032 %i + 0.0767838], [x = - 1.7337518, y = - 0.153568]]
(%i4) eqns,solns[1],ratsimp;
(%o4)                           [0 = 0, 0 = 0]
(%i5) for i:2 thru 4 do
        disp ( ev (eqns, solns[i],expand) )$
        [1.77635684E-15 - 6.66133815E-16 %i = 0, - 2.22044605E-16 = 0]
        [6.66133815E-16 %i + 1.77635684E-15 = 0, - 2.22044605E-16 = 0]
                  [- 1.13954405E-6 = 0, - 9.38499825E-8 = 0]
(%i6) solns[4];
(%o6)                  [x = - 1.7337518, y = - 0.153568]
```

We only use **ratsimp** on the first (integral) solution, which is an exact solution. For the numerical solutions, we use instead **expand**. The second and third (numerical) solutions have an accuracy of about 15 digits. The fourth (numerical) solution has about five digit accuracy.

Example 3: realroots

A Maxima user asked (on the Mailing List) how to find the value of **r** implied by the equation

```
 275.0 * exp(-r) + 275.0 * exp(-2*r) + 275.0 * exp(-3*r)
   + 275.0 * exp(-4*r) + 5275.0 * exp(-5*r) = 4750.0
```

Straightforward use of **solve** does not return an explicit solution. Here is an approach suggested by Maxima developer Stavros Macrakis.

First use **rat** to convert approximate numbers (275.0) to exact (275), replace **exp(r)** by **z**, and then try **solve**.

```
(%i1) fpprintprec:8$
(%i2) eqn : 275.0 * exp(-r) + 275.0 * exp(-2*r) + 275.0 * exp(-3*r)
    + 275.0 * exp(-4*r) + 5275.0 * exp(-5*r) - 4750.0 = 0$
(%i3) rat(eqn),ratprint:false;
(%o3)/R/
                r 5           r 4           r 3           r 2          r
        4750 (%e )  - 275 (%e )  - 275 (%e )  - 275 (%e )  - 275 %e  - 5275
      - ------------------------------------------------------------------- = 0
                                               r 5
                                            (%e )
(%i4) zeqn : ratsubst(z,exp(r),%);
                     5         4         3         2
              4750 z  - 275 z  - 275 z  - 275 z  - 275 z - 5275
(%o4)       - ------------------------------------------------- = 0
                                        5
                                       z
(%i5) soln : solve(zeqn);
                           5       4       3       2
(%o5)          [0 = 190 z  - 11 z  - 11 z  - 11 z  - 11 z - 211]
```

We see that **solve** returned a list containing one element: a simplified form of our polynomial (in **z**) equation, but was unable to return an algebraic solution. We must then consider an approximate solution, and suppose we are only interested in real roots **z**, in which case we can use **realroots**.

```
(%i6) rr : realroots(soln[1]);
                                         35805935
(%o6)                               [z = --------]
                                         33554432
(%i7) zeqn, rr, ratsimp;
                     10113653003589120573178835908551 38
(%o7)              - ------------------------------------- = 0
                    2354155174380926222089675135724933837 5
(%i8) %, numer;
(%o8)                           - 4.29608596E-4 = 0
```

We see that the returned real root is not very accurate, so we override the default value of **rootsepsilon** (which is one part in 10^7)

```
(%i9) rr : realroots( soln[1],1.0e-16);
                                     38446329182573765
(%o9)                           [z = -----------------]
                                     36028797018963968
(%i10) zeqn, rr, numer;
(%o10)                          - 6.5731657E-13 = 0
```

If $z = e^r$, then $\ln(z) = \ln(e^r) = r$. We then get an accurate numerical answer for **r** via

```
(%i11) rval : log(rhs(rr[1])),numer;
(%o11)                                 0.0649447
(%i12) eqn, r = rval;
(%o12)                         - 1.70530257E-12 = 0
```

Example 4: Using find_root

Let's go thru the first manual example for **find_root**, which has the syntax

```
    find_root ( expr, var, a, b )
```

which assumes **expr** is a function of **var**, and Maxima is to look for a numerical root of the equation **expr = 0** in the range **a <= var <= b**.

Here we find the value of **x** such that **sin(x) - x/2 = 0**. The function **solve** cannot cope with this type of nonlinear problem.

```
(%i1) e : sin(x) - x/2;
                                              x
(%o1)                               sin(x) - -
                                              2
(%i2) solve(e);
(%o2)                            [x = 2 sin(x)]
```

Let's use **plot2d** for a quick look at this expression for positive **x**.

```
(%i3) plot2d(e,[x,0,2*%pi],
          [style,[lines,5]],[ylabel," "],
           [xlabel," sin(x) - x/2 " ],
           [gnuplot_preamble,"set zeroaxis lw 2"])$
```

which produces the plot:

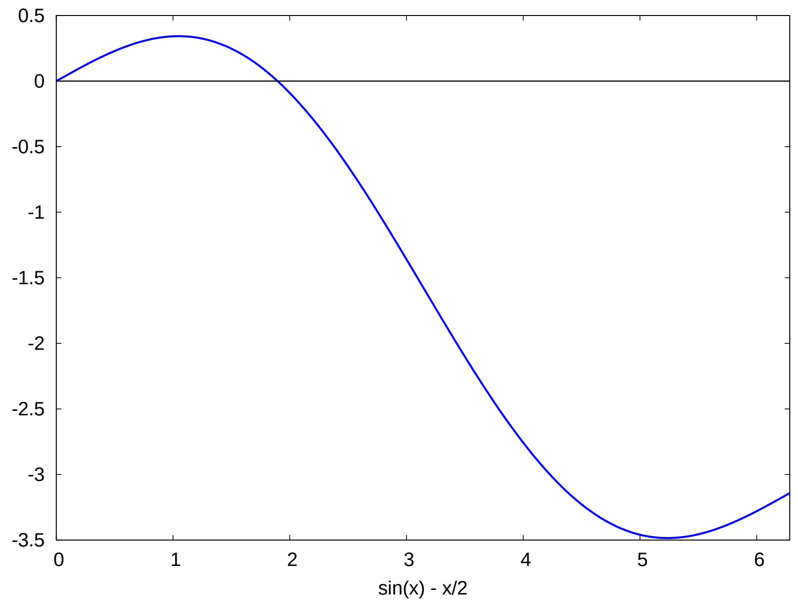

Figure 1: $\sin(x) - x/2$

We see from the plot that there is a root in the range **0 < x < %pi** . We can now ask **find_root** to locate this root numerically.

```
(%i4) xr : find_root(e,x,0.1,%pi);
(%o4)                         1.895494267033981
(%i5) e, x = xr;
(%o5)                                0.0
```

Example 5: Helping solve Find the Exact Roots

Let's find the values of **x** which satisfy the equation

$$\mathbf{a}\,(\mathbf{1} - \sin(\mathbf{x})) = \mathbf{2}\,\mathbf{b}\cos(\mathbf{x}) \tag{1.1}$$

As this equation stands, **solve** is not able to make progress.

```
(%i1) eqn : a*(1-sin(x)) - 2*b*cos(x) = 0$
(%i2) solve ( eqn, x );
                                       2 b cos(x) - a
(%o2)                      [sin(x) = - --------------]
                                             a
```

There are several ways one can help out **solve** to get the solutions we want.

Method 1 is to convert the trig functions to their complex exponential forms using **exponentialize**.

```
(%i3) eqn, exponentialize;
                   %i x     - %i x
             %i (%e     - %e      )               %i x     - %i x
(%o3)     a (---------------------- + 1) - b (%e     + %e      ) = 0
                        2
(%i4) solns : solve ( %, x );
                      2 %i b         a
(%o4) [x = - %i log(---------- + ----------),
                    2 b - %i a   2 b - %i a
                                                                  a          2 %i b
                                                x = - %i log(---------- - ----------)]
                                                             2 b - %i a   2 b - %i a
(%i5) solns : solns, rectform, ratsimp;
                                                  2    2
                      %pi                  4 b  - a      4 a b
(%o5)             [x = ---, x = - atan2(---------, ---------)]
                       2                  2    2      2    2
                                       4 b  + a   4 b  + a
(%i6) eqn, solns[1], ratsimp;
(%o6)                                 0 = 0
(%i7) eqn, solns[2], ratsimp;
(%o7)                                 0 = 0
(%i8) x2 : atan2(a^2 - 4*b^2, 4*a*b)$
(%i9) eqn, x = x2, ratsimp;
                              4      2  2    4          2    3
                  a sqrt(16 b  + 8 a  b  + a ) - 4 a b  - a
(%o9)           - ----------------------------------------- = 0
                                       2    2
                                    4 b  + a
(%i10) scanmap( 'factor, % );
(%o10)                                0 = 0
```

Method 2 is to enlarge the system of equations to include the trig identity

$$\cos^2 x + \sin^2 x = 1 \tag{1.2}$$

which one would hope would be used creatively by **solve** to find the solutions of our equation.

Let's replace `cos(x)` by the symbol `c` and likewise replace `sin(x)` by the symbol `s`, and start with a pair of equations.

```
(%i1) eqns : [a*(1-s) - 2*b*c = 0, c^2 + s^2 - 1 = 0]$
(%i2) solns : solve ( eqns, [s,c] );
                                                2    2
                                             4 b  - a         4 a b
(%o2)        [[s = 1, c = 0], [s = - ---------, c = ---------]]
                                                2    2          2    2
                                             4 b  + a       4 b  + a
(%i3) eqns, solns[1], ratsimp;
(%o3)                          [0 = 0, 0 = 0]
(%i4) x1soln : solve ( sin(x) = 1, x );
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
                                          %pi
(%o4)                               [x = ---]
                                          2
(%i5) eqn, x1soln, ratsimp;
(%o5)                               0 = 0
```

```
(%i6) eqns, solns[2], ratsimp;
(%o6)                           [0 = 0, 0 = 0]
(%i7) tan(x) = s/c, solns[2], ratsimp;
(%o7) tan(x) = -(4*b^2-a^2)/(4*a*b)
(%i8) x2soln : solve (%, x), ratsimp;
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
                                          2    2
                                       4 b  - a
(%o8)                       [x = - atan(---------)]
                                         4 a b
```

Bearing in mind that $\arctan(-x) = -\arctan(x)$, we get the same set of solutions using method 2.

1.8.13 Non-Rational Simplification: radcan, logcontract, rootscontract, and radexpand

The wxMaxima interface has the **Simplify(r)** button allows uses the **radcan** function on a selection. The "r" stands for **radcan** although you could just as well think "radical", since it is equivalent to the wxMaxima menu choice **Simplify, Simplify Radicals**. You can also think of the work **radcan** as a mnenomic constructed from the capitalised letters in "RADical CANcellation", but this does not hint at the full power of this function.

The manual asserts that **radcan** is useful in simplifying expressions containing logs, exponentials, and radicals. Here is an expression containing exponentials which **radcan** simplifies.

```
(%i1) expr : (exp(x)-1)/(exp(x/2)+1);
                                       x
                                     %e  - 1
(%o1)                               ---------
                                      x/2
                                    %e    + 1
(%i2) expr, ratsimp;
                                       x
                                     %e  - 1
(%o2)                               ---------
                                      x/2
                                    %e    + 1
(%i3) expr, radcan;
                                      x/2
(%o3)                               %e    - 1
```

We see that **radcan** is able to simplify this expression containing exponentials, whereas **ratsimp** cannot.

Here is some Xmaxima work showing the differences between the use of **ratsimp**, **radcan**, and **logcontract** on an expression which is a sum of logs.

```
(%i4) logexpr : log ( (x+2)*(x-2) ) + log(x);
(%o4)                    log((x - 2) (x + 2)) + log(x)
(%i5) logexpr, ratsimp;
                                 2
(%o5)                        log(x  - 4) + log(x)
(%i6) logexpr, fullratsimp;
                                 2
(%o6)                        log(x  - 4) + log(x)
(%i7) logexpr, radcan;
(%o7)                  log(x + 2) + log(x) + log(x - 2)
(%i8) %, logcontract;
                                     3
(%o8)                            log(x  - 4 x)
```

```
(%i9) logexpr, logcontract;
                                    3
(%o9)                          log(x  - 4 x)
```

We see that use of **logcontract** provided the same quality of simplification as the two step route **radcan** followed by **logcontract**.

Here we expolore the use of **radcan** and **rootscontract** to simplify square roots:

```
(%i10) sqrt(2)*sqrt(3), radcan;
(%o10)                          sqrt(2) sqrt(3)
(%i11) sqrt(2)*sqrt(3), rootscontract;
(%o11)                              sqrt(6)
(%i12) sqrt(6)*sqrt(3), radcan;
(%o12)                            3 sqrt(2)
(%i13) sqrt(6)*sqrt(3), rootscontract;
(%o13)                            3 sqrt(2)
(%i14) sqrt(6)/sqrt(3), radcan;
(%o14)                              sqrt(2)
(%i15) sqrt(6)/sqrt(3), rootscontract;
(%o15)                              sqrt(2)
```

radexpand

The Maxima manual describes the option variable **radexpand** which can have the values **true** (the default), **all**, or **false**. The setting of **radexpand** controls the automatic simplifications of radicals.

In the default case (**radexpand** set to **true**), `sqrt(x^2)` simplifies to `abs(x)`. This is because, by default, all symbols, in the absence of special declarations, are considered to represent real numbers. You can declare that all symbols should (unless declared otherwise) be considered to represent complex numbers using the **domain** flag, which has the default value **real**.

```
(%i16) sqrt(x^2);
(%o16)                              abs(x)
(%i17) domain:complex$
(%i18) sqrt(x^2);
                                         2
(%o18)                            sqrt(x )
(%i19) domain:real$
(%i20) sqrt(x^2);
(%o20)                              abs(x)
```

Next we show the simplifications which occur to the expression `sqrt(16*x^2)` for the three possible values of **radexpand**.

```
(%i21) radexpand;
(%o21)                               true
(%i22) sqrt(16*x^2);
(%o22)                            4 abs(x)
(%i23) radexpand:all$
(%i24) sqrt(16*x^2);
(%o24)                               4 x
(%i25) radexpand:false$
(%i26) sqrt(16*x^2);
                                           2
(%o26)                           sqrt(16 x )
```

1.8.14 Trigonometric Simplification: trigsimp, trigexpand, trigreduce, and trigrat

Here are simple examples of the important trig simplification functions **trigsimp**, **trigexpand**, **trigreduce**, and **trigrat**.

trigsimp converts trig functions to sines and cosines and also attempts to use the identiy $\cos(x)^2 + \sin(x)^2 = 1$.

```
(%i1) trigsimp(tan(x));
                                   sin(x)
(%o1)                              ------
                                   cos(x)
(%i2) sin(x+y), trigexpand;
(%o2)                   cos(x) sin(y) + sin(x) cos(y)
(%i3) x+3*cos(x)^2 - sin(x)^2, trigreduce;
                     cos(2 x)      cos(2 x)    1        1
(%o3)                -------- + 3 (-------- + -) + x - -
                        2             2        2        2
(%i4) trigrat ( sin(3*a)/sin(a+%pi/3) );
(%o4)                  sqrt(3) sin(2 a) + cos(2 a) - 1
```

The manual entry for **trigsimp** advises that **trigreduce**, **ratsimp**, and **radcan** may be able to further simplify the result. The author has the following function defined in **mbe1util.mac**, which is loaded in during startup.

```
    rtt(e) := radcan ( trigrat ( trigsimp (e)))$
```

This function can be equivalent to **trigreduce** if only sines and cosines are present, but otherwise may return different forms, as we show here, comparing the opposide effects of **trigexpand** and **trigreduce**. Note that **trigexpand** expands trig functions of sums of angles and/or trig functions of multiple angles (like `2*x` at the "top level", and the user should read the manual description to see options available.

```
(%i5) e: sin(x+y), trigexpand;
(%o5)                   cos(x) sin(y) + sin(x) cos(y)
(%i6) e, trigreduce;
(%o6)                            sin(y + x)
(%i7) rtt (e);
(%o7)                            sin(y + x)
(%i8) e :  tan(x+y), trigexpand;
                               tan(y) + tan(x)
(%o8)                         -----------------
                              1 - tan(x) tan(y)
(%i9) e, trigreduce;
                          tan(y)             tan(x)
(%o9)                - ----------------- - -----------------
                       tan(x) tan(y) - 1   tan(x) tan(y) - 1
(%i10) %, ratsimp;
                                tan(y) + tan(x)
(%o10)                        - -----------------
                                tan(x) tan(y) - 1
(%i11) rtt (e);
                                 sin(y + x)
(%o11)                           ----------
                                 cos(y + x)
(%i12) trigsimp( tan(x+y) );
                                 sin(y + x)
(%o12)                           ----------
                                 cos(y + x)
(%i13) e :  cosh(x + y), trigexpand;
(%o13)                sinh(x) sinh(y) + cosh(x) cosh(y)
(%i14) e, trigreduce;
(%o14)                           cosh(y + x)
```

```
(%i15) rtt (e);
                              - y - x    2 y + 2 x
                            %e        (%e          + 1)
(%o15)                      ---------------------------
                                         2
(%i16) expand(%);
                                y + x     - y - x
                              %e        %e
(%o16)                        ------- + ---------
                                 2          2
```

In the following, note the difference when using `trigexpand(expr)` compared with `ev ( expr, trigexpand )`. This difference is due to the fact that **trigexpand** has both the **evfun** and the **evflag** properties.

```
(%i1) e1 : trigexpand( tan(2*x + y) );
                               tan(y) + tan(2 x)
(%o1)                        -------------------
                              1 - tan(2 x) tan(y)
(%i2) e2 : tan (2*x + y), trigexpand;
                                         2 tan(x)
                             tan(y) + -----------
                                               2
                                        1 - tan (x)
(%o2)                        --------------------
                                  2 tan(x) tan(y)
                             1 - ---------------
                                            2
                                    1 - tan (x)
(%i3) rtt (e1);
                                  sin(y + 2 x)
(%o3)                             ------------
                                  cos(y + 2 x)
(%i4) rtt (e2);
                                  sin(y + 2 x)
(%o4)                             ------------
                                  cos(y + 2 x)
(%i5) trigsimp (e1);
                         cos(2 x) sin(y) + sin(2 x) cos(y)
(%o5)                  - ---------------------------------
                         sin(2 x) sin(y) - cos(2 x) cos(y)
(%i6) trigsimp (e2);
                        2
                 (2 cos (x) - 1) sin(y) + 2 cos(x) sin(x) cos(y)
(%o6)          - -----------------------------------------------
                                                      2
                 2 cos(x) sin(x) sin(y) + (1 - 2 cos (x)) cos(y)
```

1.8.15 Complex Expressions: rectform, demoivre, realpart, imagpart, and exponentialize

Sec. 6.2 of the Maxima manual has the introduction

> A complex expression is specified in Maxima by adding the real part of the expression to `%i` times the imaginary part. Thus the roots of the equation `x^2 - 4*x + 13 = 0` are `2 + 3*%i` and `2 - 3*%i`. Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the **realpart**, **imagpart**, **rectform**, **polarform**, **abs**, **carg** functions.

Here is a brief look at some of these functions, starting with **exponentialize**.

```
(%i1) cform : [cos(x),sin(x),cosh(x)], exponentialize;
             %i x     - %i x          %i x     - %i x     x     - x
           %e     + %e            %i (%e     - %e      )  %e  + %e
(%o1)     [-----------------, - ----------------------, -----------]
                  2                        2                  2
(%i2) cform, demoivre;
                                                x     - x
                                              %e  + %e
(%o2)                       [cos(x), sin(x), -----------]
                                                   2
(%i3) cform, rectform;
                                                x     - x
                                              %e  + %e
(%o3)                       [cos(x), sin(x), -----------]
                                                   2
(%i4) realpart ( cform );
                                                x     - x
                                              %e  + %e
(%o4)                       [cos(x), sin(x), -----------]
                                                   2
(%i5) imagpart ( cform );
(%o5)                              [0, 0, 0]
```

1.8.16 Are Two Expressions Numerically Equivalent? zeroequiv

The function **zeroequiv(expr, var)** uses a series of randomly chosen values of the variable **var** to test if **expr** is equivalent to zero. Although there are cases where **dontknow** is returned (see the manual cautions), this function can be useful if you arrange that **expr** is the difference of two other expressions which you suspect are numerically equivalent (or not).

As a pedagogical example consider showing that

$$\cos^2(x - 1) = \frac{1}{2}\left(\sin(2)\ \sin(2\,x) + \cos(2)\ \cos(2x) + 1\right) \tag{1.3}$$

is true. Although it is not difficult to show the equivalence using the standard Maxima tools (see Sec. 10.3.6 in Ch.10), we use here **zeroequiv**.

```
(%i1) e1:cos(x-1)^2;
                                        2
(%o1)                                cos  (x - 1)
(%i2) e2 : (sin(2)*sin(2*x)+cos(2)*cos(2*x)+1)/2;
                       sin(2) sin(2 x) + cos(2) cos(2 x) + 1
(%o2)                  -------------------------------------
                                         2
(%i3) zeroequiv ( e1-e2, x );
(%o3)                                  true
```

An alternative (and complementary) approach is to plot both expressions `e1` and `e2` on the same plot.

1.9 User Defined Maxima Functions: define, fundef, block, and local

A Maxima function can be defined using the `:=` operator. The left side of the function definition should be the name of the function followed by comma separated formal parameters enclosed in parentheses. The right side of the Maxima function definition is the "function body". When a Maxima function is called, the formal parameters are bound to the call arguments, any "free" variables in the function body take on the values that they have at the time of the function call, and the function body is evaluated. You can define Maxima functions which are recursive to an "arbitrary" depth (of course there are always practical limits due to memory, speed,...). After a Maxima function is defined, its name is added to the Maxima list **functions**.

Problems may sometimes arise when passing to a Maxima function (as one of the calling arguments) an expression which contains a variable with the same name as a formal parameter used in the function body (name conflicts).

1.9.1 A Function Which Takes a Derivative

In this section we discuss a question sent in to the Maxima Mailing List (which we paraphrase).

```
I want to make the derivative of a function a function itself.
Naively, I tried the following.
First define a Maxima function f(x) as a simple polynomial.
Next define a second Maxima function fp(x) ("f-prime")
  which will take the first derivative of f(x).
Then try to use fp(x) to get the derivative of f at x =1.

This did not work, as shown here:
------------------------------------------------
(%i1) f(x):= -2*x^3 + 3*x^2 + 12*x - 13$
(%i2) fp(x):= diff(f(x),x)$
(%i3) fp(1);
diff: second argument must be a variable; found 1
#0: fp(x=1)
 -- an error.  To debug this try debugmode(true);
---------------------------------------------
What can I do?
Thanks
```

The error message returned by Maxima in response to input `%i3` means that Maxima determined that the job requested was to return the result of the operation `diff( f(1), 1 )`. In other words, "variable substitution" occurred before differentiation, whereas, what is wanted is first differentiation, and then variable substitution.

Let's pick a simpler function, and compare three ways to try to get this job done.

```
(%i1) f(y) := y^3;
                                                3
(%o1)                             f(y) := y
(%i2) f(z);
                                           3
(%o2)                                     z
(%i3) f1(x) := diff ( f(x), x);
(%o3)                       f1(x) := diff(f(x), x)
(%i4) f2(x) := ''(diff ( f(x), x));
                                                  2
(%o4)                            f2(x) := 3 x
(%i5) define ( f3(x), diff ( f(x), x) );
                                                  2
(%o5)                            f3(x) := 3 x
```

```
(%i6) [f1(z), f2(z), f3(z) ];
                                      2      2      2
(%o6)                              [3 z , 3 z , 3 z ]
(%i7) [f1(1), f2(1), f3(1) ];
diff: second argument must be a variable; found 1
#0: f1(x=1)
 -- an error.  To debug this try debugmode(true);
(%i8) [f2(1), f3(1)];
(%o8)                                    [3, 3]
```

As long as we request the derivative of f(z), where z is treated as an undefined variable (ie., **z** has not been bound to a number, in which case **`numberp(z)`** will return **`false`**), all three definitions give the correct result. But when we request the derivative at a specific numerical point, only the last two methods give the correct result without errors, and the first (”delayed assignment operator method”) does not work.

Note that the “quote-quote method” requires two single quotes, and we need to surround the whole expression with parentheses: **`''(...)`**. This “quote-quote” method is available only in interactive use, and will not work inside another function, as you can verify. The conventional wisdom denigrates the quote-quote method and advises use of the **define** method.

The **define** function method can not only be used interactively, but also when one needs to define a function like this inside another Maxima function, as in

```
(%i9) dplay(g,a,b) := block ( [val1,val2 ],
        local (dg),
          define ( dg(y), diff (g(y), y)),
          val1 : g(a) + dg(a),
          val2 : g(b) + dg(b),
          display (val1,val2),
          val1 + val2)$
(%i10) g(x) := x;
(%o10)                              g(x) := x
(%i11) dplay(g,1,1);
                                    val1 = 2

                                    val2 = 2

(%o11)                                  4
(%i12) g(x) := x^2;
                                                2
(%o12)                             g(x) := x
(%i13) dplay(g,1,2);
                                    val1 = 3

                                    val2 = 8

(%o13)                                 11
(%i14) dg(3);
(%o14)                                dg(3)
```

You can find the manual discussion of the quote-quote operator near the top of the index list where misc. symbols are recorded. There you find the comment “Applied to the operator of an expression, quote-quote changes the operator from a noun to a verb (if it is not already a verb).”

The use of **local** inside our Maxima function **dplay** prevents our definition of the Maxima function **dg** from leaking out into the top level context and keeps the knowledge of **dg** inside **dplay** and any user defined Maxima functions which are called by **dplay**, and inside any user defined Maxima functions which are called by those functions, etc...

Maxima developer Stavros Macrakis has posted a good example which emphasizes the difference between the delayed assignment operator `:=` and the **define** function. This example uses Maxima's **print** function.

```
(%i1) f1(x):=print (x)$
(%i2) define ( f2(x), print (x) )$
x
```

Notice that the delayed assignment definition for `f1` did not result in the actual operation of **print(x)**, whereas the **define** definition of `f2` resulted in the operation of **print(x)**, as you can see on your screen after input `%i2`. Quoting from Macrakis:

> In the case of `':='`, both arguments are unevaluated. This is the normal way function definitions work in most programming languages.
>
> In the case of `'define'`, the second argument is evaluated normally [in the process of absorbing the definition], not unevaluated. This allows you to `*calculate*` a definition.

We can see the difference between **f1** and **f2** by using **fundef**, which returns what Maxima has accepted as a definition in the two cases.

```
(%i3) fundef (f1);
(%o3)                          f1(x) := print(x)
(%i4) fundef (f2);
(%o4)                             f2(x) := x
```

Let's now compare these definitions at runtime. `f1` will print its argument to the screen, but `f2` will not, since in the latter case, all reference to Maxima's **print** function has been lost.

```
(%i5) f1(5);
5
(%o5)                                  5
(%i6) f2(5);
(%o6)                                  5
```

Macrakis continues:

> `'define'` is useful for calculating function definitions which will later be applied to arguments. For example, compare:

```
(%i7) define (f(x), optimize (horner (taylor (tanh(x),x,0,8),x)));
                                   2  x (%1 (%1 (42 - 17 %1) - 105) + 315)
(%o7) f(x) := block([%1], %1 : x , ------------------------------------)
                                                   315
(%i8) f(0.1);
(%o8)                          0.099667994603175
(%i9) g(x) := optimize (horner (taylor (tanh(x),x,0,8),x));
(%o9)        g(x) := optimize(horner(taylor(tanh(x), x, 0, 8), x))
(%i10) g(0.1);
Variable of expansion cannot be a number: 0.1
#0: g(x=0.1)
 -- an error.  To debug this try debugmode(true);
```

> `g(.1) => ERROR` (because it is trying to do `taylor(tanh(.1),.1,0,8)`).

1.9.2 Lambda Expressions

Lambda notation is used to define unnamed "anonymous" Maxima functions. Lambda expressions allow you to define a special purpose function which will not be used in other contexts (since the function has no name, it cannot be used later).

The general syntax is
lambda (arglist, expr1, expr2, ...,exprn)
where arglist has the form `[x1, x2, ...,xm]` and provide formal parameters in terms of which the `expr1, expr2,...` can be written, (use can also be made of `%%` to refer to the result of the previous expression `exprj`). When this function is evaluated, unbound local variables `x1,x2,...` are created, and then `expr1` through `exprn` are evaluated in turn. The return value of this function is `exprn`.

```
(%i1) functions;
(%o1) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
(%i2) f : lambda([x,y],x^2 + y^3);
                                                   2    3
(%o2)                           lambda([x, y], x  + y )
(%i3) f (1,1);
(%o3)                                         2
(%i4) f(2,a);
                                            3
(%o4)                                     a   + 4
(%i5) apply ( f, [1,2] );
(%o5)                                       9
(%i6) functions;
(%o6) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
(%i7) map ( 'sin, [1,2,3] );
(%o7)                          [sin(1), sin(2), sin(3)]
(%i8) map ( lambda([x],x^2 + sin(x) ),[1,2,3] );
(%o8)                    [sin(1) + 1, sin(2) + 4, sin(3) + 9]
(%i9) lambda ([x],x+1 )(3);
(%o9)                                       4
```

1.9.3 Recursive Functions; factorial, and trace

Here is an example which does the same jub as the Maxima function **factorial**. We turn on **trace** to watch the recursion process.

```
(%i1) myfac(n) := if n = 0 then 1 else n*myfac(n-1)$
(%i2) map ('myfac, [0,1,2,3,4] );
(%o2)                             [1, 1, 2, 6, 24]
(%i3) [0!,1!,2!,3!,4!];
(%o3)                             [1, 1, 2, 6, 24]
(%i4) map ('factorial, [0,1,2,3,4] );
(%o4)                             [1, 1, 2, 6, 24]
(%i5) trace (myfac);
(%o5)                                  [myfac]
```

```
(%i6) myfac(4);
1 Enter myfac [4]
 2 Enter myfac [3]
  3 Enter myfac [2]
   4 Enter myfac [1]
    5 Enter myfac [0]
    5 Exit  myfac 1
   4 Exit  myfac 1
  3 Exit  myfac 2
 2 Exit  myfac 6
1 Exit  myfac 24
(%o6)                               24
(%i7) untrace(myfac);
(%o7)                             [myfac]
(%i8) [myfac(4),factorial(4),4!];
(%o8)                          [24, 24, 24]
```

Legendre Polynomial

As a second example of a recursive definition of a Maxima function, one can define the Legendre polynomial $\mathbf{P_n(x)}$ via a recursive relation that relates three different **n** values (for a given **x**). We can check our design by using the Maxima function **legendre_p (n,x)**, which is part of the orthogonal polynomial package **orthopoly.lisp**.

```
(%i1) p(n,x) := if n=0 then 1 elseif n=1 then x else
              expand( ( (2*n-1)/n )*x*p(n-1,x) - ((n-1)/n)*p(n-2,x) )$
(%i2) [p(0,x),p(1,x),p(2,x),p(3,x)];
                                   2          3
                                3 x    1   5 x    3 x
(%o2)                   [1, x, ---- - -, ---- - ---]
                                 2     2   2      2
(%i3) map ( lambda([nn], p (nn,x) ),[0,1,2] );
                                         2
                                      3 x    1
(%o3)                         [1, x, ---- - -]
                                       2     2
(%i4) map ( lambda([nn], expand (legendre_p (nn,x))),[0,1,2] );
                                         2
                                      3 x    1
(%o4)                         [1, x, ---- - -]
                                       2     2
```

1.9.4 Non-Recursive Subscripted Functions (Hashed Arrays)

"Undeclared arrays" are called "hashed arrays" and they grow dynamically as more "array elements" are assigned values or otherwise defined. Here is a non-recursive simple subscripted function having one subscript **k** and one formal argument **x**.

```
(%i1) f[k](x) := x^k + 1;
                                           k
(%o1)                           f (x) := x  + 1
                                 k
(%i2) arrays;
(%o2)                                 [f]
(%i3) makelist ( f[nn](y),nn,0,3 );
                                      2       3
(%o3)                    [2, y + 1, y  + 1, y  + 1]
(%i4) arrayinfo(f);
(%o4)                   [hashed, 1, [0], [1], [2], [3]]
```

```
(%i5) map ( lambda ( [m], f[m](y) ), [0,1,2,3] );
                                             2        3
(%o5)                      [2, y + 1, y  + 1, y  + 1]
```

Here is a example which uses Maxima's anonymous **lambda** function to define a subscripted function which does the same job as above.

```
(%i1) h[nn] := lambda ([xx], xx^nn + 1);
                                                     nn
(%o1)                    h    := lambda([xx], xx   + 1)
                          nn
(%i2) arrays;
(%o2)                                  [f, h]
(%i3) arrayinfo(h);
(%o3)                              [hashed, 1]
(%i4) h[2];
                                                  2
(%o4)                          lambda([xx], xx  + 1)
(%i5) arrayinfo(h);
(%o5)                           [hashed, 1, [2]]
(%i6) h[2](y);
                                          2
(%o6)                                    y  + 1
(%i7) map ( lambda ( [mm], h[mm](x) ),[0,1,2,3] );
                                           2        3
(%o7)                      [2, x + 1, x  + 1, x  + 1]
(%i8) arrayinfo(h);
(%o8)                    [hashed, 1, [0], [1], [2], [3]]
```

1.9.5 Recursive Hashed Arrays and Memoizing

We again design a homemade factorial "function", this time using a "hashed array" having one "index" **n**.

```
(%i1) arrays;
(%o1)                                  []
(%i2) a[n] := n*a[n-1]$
(%i3) a[0] : 1$
(%i4) arrays;
(%o4)                                  [a]
(%i5) arrayinfo(a);
(%o5)                           [hashed, 1, [0]]
(%i6) a[4];
(%o6)                                  24
(%i7) arrayinfo(a);
(%o7)                  [hashed, 1, [0], [1], [2], [3], [4]]
(%i8) a[n] := n/2$
(%i9) a[4];
(%o9)                                  24
(%i10) a[7];
                                        7
(%o10)                                  -
                                        2
(%i11) arrayinfo(a);
(%o11)              [hashed, 1, [0], [1], [2], [3], [4], [7]]
```

We see in the above example that elements **1** through **4** were computed when the call **a[4]** was first made. Once the array elements have been computed, they are not recomputed, even though our definition changes. Note that after **(%i8) a[n] := n/2$**, our interogation **(%i9) a[4];** does not return our new definition, but the result of the original calculation. If you compute **a[50]** using our first recursive factorial definition, all values **a[1]** through **a[50]** are remembered. Computing **a[51]** then takes only one step.

1.9.6 Recursive Subscripted Maxima Functions

We here use a "subscripted function" to achieve a recursive definition of a Legendre polynomial $P_n(x)$. (The author thanks Maxima developer Richard Fateman for this example.)

```
(%i1) ( p[n](x) := expand(((2*n-1)/n)*x*p [n-1](x) - ((n-1)/n)*p [n-2](x) ),
         p [0](x) := 1, p [1](x) := x ) $
(%i2) makelist( p[m](x),m,0,4 );
                             2          3             4       2
                          3 x    1  5 x    3 x  35 x    15 x    3
(%o2)            [1, x, ---- - -, ---- - ---, ----- - ----- + -]
                           2     2   2      2     8       4     8
(%i3) arrays;
(%o3)                                   [p]
(%i4) arrayinfo(p);
(%o4)                  [hashed, 1, [0], [1], [2], [3], [4]]
(%i5) [p[3](x),p[3](y)];
                                   3           3
                                5 x    3 x  5 y    3 y
(%o5)                          [---- - ---, ---- - ---]
                                 2      2    2      2
(%i6) map ( lambda([nn], p[nn](x)),[0,1,2,3] );
                                       2         3
                                    3 x    1  5 x    3 x
(%o6)                        [1, x, ---- - -, ---- - ---]
                                     2     2   2      2
```

Note that the input line `%i1` has the structure
    `( job1, job2, job3 )$`. This is similar to using one input line to bind values to several variables.

```
(%i7) ( a:1, b:2/3, c : cos(4/3) )$
```

If you want a visual reminder of the bindings, then the `[job1,job2,..]` notation is more appropriate.

```
(%i8) [ a:1, b:2/3, c : cos(4/3) ];
                                        2      4
(%o8)                               [1, -, cos(-)]
                                        3      3
```

1.9.7 Floating Point Numbers from a Maxima Function

Let's use our recursive subscripted function definition of $P_n(x)$ and compare the "exact" value with the default floating point number when we look for $P_3(8/10)$. Let `p3t` hold the exact value `2/25`. Maxima's **float** function returns `0.08` from `float(2/25)`, but internally Maxima holds a binary representation with has a small error when converted to a decimal number. We can see the internally held decimal number by using `?print` which accesses the lisp function **print** from Maxima.

```
(%i1) ( p[n](x) := expand(((2*n-1)/n)*x*p [n-1](x) - ((n-1)/n)*p [n-2](x) ),
         p [0](x) := 1, p [1](x) := x ) $
(%i2) p3 : p[3](x);
                                          3
                                       5 x    3 x
(%o2)                                  ---- - ---
                                        2      2
(%i3) p3t : p3, x = 8/10;
                                           2
(%o3)                                      --
                                           25
```

```
(%i4) float(p3t);
(%o4)                                0.08
(%i5) ?print(%);
0.08000000000000002
(%o5)                                0.08
(%i6) p3f : p3, x = 0.8;
(%o6)                                0.08
(%i7) ?print(%);
0.08000000000000071
(%o7)                                0.08
(%i8) abs (p3f - p3t);
(%o8)                       6.9388939039072284E-17
(%i9) abs (%o4 - p3t);
(%o9)                                 0.0
(%i10) ?print(%);
0.0
(%o10)                                0.0
```

The input line **`(%i6) p3f : p3, x = 0.8;`** replaces **`x`** by the floating point decimal number **`0.8`** (which has no exact binary representation) and then uses binary arithmetic to perform the indicated additions, divisions, and multiplications, and then returns the closest decimal number corresponding to the final binary value. The result of all this hidden work is a number with some ”floating point error”, which we calculate in line **`%i8`**.

We can come to the same conclusion in a slightly different way if we first define a Maxima function **`P3(x)`** using **`p[3]`**, say.

```
(%i11) define (P3(y), p[3](y))$
(%i12) P3(x);
                                      3
                                   5 x    3 x
(%o12)                             ---- - ---
                                    2      2
(%i13) P3f1 : float (P3(8/10));
(%o13)                               0.08
(%i14) ?print(%);
0.08000000000000002
(%o14)                               0.08
(%i15) P3f2 : P3(0.8);
(%o15)                               0.08
(%i16) ?print(%);
0.08000000000000071
(%o16)                               0.08
(%i17) abs (P3f2 - P3f1);
(%o17)                      6.9388939039072284E-17
```

We have used **`?print(...)`** above to look at the approximate decimal digit equivalent of a binary representation held internally by Maxima for a floating point number. We can sometimes learn new things by looking at Lisp translations of input and output using **`?print( Maxima stuff )`**, **`:lisp #$ Maxima stuff $`**, and **`:lisp $_`**. (See html Help Manual: Click on Right Panel, Click on Contents, Sec. 3.1, Lisp and Maxima.)

```
(%i18) ?print(a + b)$
((MPLUS SIMP) $A $B)
(%i19) :lisp $_
((PRINT) ((MPLUS) $A $B))
(%i19) :lisp #$a+b$
((MPLUS SIMP) $A $B)
(%i19) ?print( a/b )$
((MTIMES SIMP) $A ((MEXPT SIMP) $B -1))
(%i20) :lisp $_
((PRINT) ((MQUOTIENT) $A $B))
```

```
(%i20) :lisp #$a / b$
((MTIMES SIMP) $A ((MEXPT SIMP) $B -1))
(%i21) bfloat(2/25),fpprec:30;
(%o21)                              8.0b-2
(%i22) ?print(%);
((BIGFLOAT SIMP 102) 3245185536584267267831560205762 -3)
(%o22)                              8.0b-2
(%i23) p3, x = 8/10;
                                      2
(%o23)                                 --
                                      25
(%i24) :lisp $_
(($EV) $P3 ((MEQUAL) $X ((MQUOTIENT) 8 10)))
(%i24) :lisp #$ p3, x=8/10 $
((RAT SIMP) 2 25)
```

1.10 Pulling Out Overall Factors from an Expression

If you have an expression of the form `expr : u * ( fac*c1 + fac*c2 )`, you can pull out the common factor `fac` using `fac * ratsimp ( expr / fac )` . Of course there may also be other methods which will work for a particular expression.

Here is one example.

```
(%i1) e1 : x*(A*cos(d1)^3 - A*cos(d2)^3);
                                3            3
(%o1)                    x (cos  (d1) A - cos  (d2) A)
(%i2) A * ratsimp (e1/A);
                               3          3
(%o2)                     (cos  (d1) - cos  (d2)) x A
(%i3) rat (e1);
                                3          3
(%o3)/R/                 (- cos  (d2) + cos  (d1)) x A
```

Here is a second example.

```
(%i4) e2 : x*(A*a*b1 - A*a*b2);
(%o4)                        x (a b1 A - a b2 A)
(%i5) A*a*ratsimp ( e2/( A*a));
(%o5)                          a (b1 - b2) x A
(%i6) factor(e2);
(%o6)                         - a (b2 - b1) x A
(%i7) rat(e2);
(%o7)/R/                     (- a b2 + a b1) x A
```

We see above that **rat** does not pull out all the common factors in this second example. We also find that **factor** does not simply pull out factors, but also rewrites the trig functions, when used with the first expression.

```
(%i8) factor (e1);
                                2                            2
(%o8)  - (cos(d2) - cos(d1)) (cos  (d2) + cos(d1) cos(d2) + cos  (d1)) x A
```

More discussion of display and simplification tools will appear in the upcoming new Chapter 4.

1.11 Construction and Use of a Test Suite File

It is useful to construct a code test which can be run against new versions of your own personal code, as well as against new versions of Maxima. We show here a very simple example of such a code test file and use.

The test of floating point division (below) is sensitive to the specific Lisp version used to compile the Maxima source code; different Lisp versions may have slightly different floating point behaviors and digits of precision. Here are the contents of **mytest.mac**, which contains the tests: addition of integers, multiplication of integers, integer division, and floating point division. The convention is to include an input Maxima command followed by the expected Maxima answer (with the addition of a semicolon). The command `batch("mytest.mac",test)` will then use Maxima's test utilities. (You will find much useful and practical information in the file `README.how-to` in the `...\maxima\5.19.0\tests` folder.)

```
/* this is mytest.mac */
1 + 2;
3;
2 * 3;
6;
2/3;
2/3;
2.0/3;
0.66666666666666663;
```

Here is an example of running **mytest.mac** as a **batch** file in **test** mode using one dimensional output display.

```
(%i1) display2d:false$
(%i2) 2.0/3;
(%o2) 0.66666666666667
(%i3) ?print(%);
0.66666666666666663
(%o3) 0.66666666666667
(%i4) batch ("mytest.mac",test )$
Error log on #<output stream mytest.ERR>
********************** Problem 1 ***************
Input:
2+1
Result:
3
... Which was correct.

********************** Problem 2 ***************
Input:
2*3
Result:
6
... Which was correct.

********************** Problem 3 ***************
Input:
2/3
Result:
2/3
... Which was correct.

********************** Problem 4 ***************
Input:
2.0/3
Result:
0.66666666666667
... Which was correct.
4/4 tests passed.
```

Note that to avoid an error return on the floating point division test, we needed to use for the answer the result returned by the Lisp print function, which is available from Maxima by preceding the Lisp function name with a question mark (?).

1.12 History of Maxima's Development

From the Unix/Linux Maxima manual:

> MACSYMA (Project MAC's SYmbolic MAnipulation System) was developed by the Mathlab group of the MIT Laboratory for Computer Science (originally known as Project MAC), during the years 1969-1972. Their work was supported by grants NSG 1323 of the National Aeronautics and Space Administration, N00014-77-C-0641 of the Office of Naval Research, ET-78-C-02-4687 of the U.S. Department of Energy, and F49620-79-C-020 of the U.S. Air Force. MACSYMA was further modified for use under the UNIX operating system (for use on DEC VAX computers and Sun workstations), by Richard Fateman and colleagues at the University of California at Berkeley; this version of MACSYMA is known as VAXIMA. The present version stems from a re-working of the public domain MIT MACSYMA for GNU Common Lisp, prepared by William Schelter, University of Texas at Austin until his passing away in 2001. It contains numerous additions, extensions and enhancements of the original.

From the first page of the Maxima html help manual:

> Maxima is derived from the Macsyma system, developed at MIT in the years 1968 through 1982 as part of Project MAC. MIT turned over a copy of the Macsyma source code to the Department of Energy in 1982; that version is now known as DOE Macsyma. A copy of DOE Macsyma was maintained by Professor William F. Schelter of the University of Texas from 1982 until his death in 2001. In 1998, Schelter obtained permission from the Department of Energy to release the DOE Macsyma source code under the GNU Public License, and in 2000 he initiated the Maxima project at SourceForge to maintain and develop DOE Macsyma, now called Maxima.

Maxima is now developed and maintained by the Maxima project at
`http://maxima.sourceforge.net`.