Maxima (5.22.1) and the Calculus

Leon Q. Brin

July 27, 2011

Contents

1 Introduction **3**
- 1.1 About this document . . . 3
- 1.2 History and Philosophy . . . 3
- 1.3 Using Maxima . . . 3
- 1.4 Reading the examples . . . 4

2 Basics **4**
- 2.1 Basic arithmetic commands . . . 5
  - 2.1.1 Addition, Subtraction, Multiplication, and Division . . . 5
  - 2.1.2 Exponents and Factorials . . . 5
  - 2.1.3 Square and other roots . . . 6

3 Precalculus **6**
- 3.1 Trigonometry . . . 6
  - 3.1.1 Trigonometric functions and π . . . 6
  - 3.1.2 Inverse trigonometric functions . . . 7
- 3.2 Assignment and Function definition . . . 7
- 3.3 Exponentials and Logarithms . . . 8
- 3.4 Constants . . . 9
- 3.5 Solving equations . . . 9
  - 3.5.1 Example (Maxima fails to solve an equation) . . . 10
- 3.6 Simplification . . . 11
  - 3.6.1 Example (factoring) . . . 12
  - 3.6.2 Example (logcontract) . . . 12
  - 3.6.3 Example (trigsimp) . . . 12
  - 3.6.4 Example (other trig simplifications) . . . 12
- 3.7 Evaluation . . . 13
- 3.8 Big floats . . . 13

4 Limits **13**
- 4.1 Example (limit of a difference quotient) . . . 14

5 Differentiation **15**
- 5.1 Example (related rates) . . . 16
- 5.2 Example(optimization) . . . 16
- 5.3 Example (second derivative test) . . . 17

6 Integration **17**
6.1 Riemann Sums . . . 17
6.2 Antiderivatives . . . 18
6.2.1 Trig substitution . . . 20
6.2.2 Integration by parts . . . 20
6.2.3 Partial fractions . . . 21
6.2.4 Multiple integrals . . . 22

7 Series **23**
7.1 Scalar Series . . . 23
7.2 Taylor Series . . . 24

8 Vector Calculus **26**
8.1 Package vect . . . 26
8.1.1 `u~v` . . . 27
8.1.2 `u.v` . . . 27
8.1.3 `grad(f)` . . . 27
8.1.4 `laplacian(f)` . . . 27
8.1.5 `div(F)` . . . 27
8.1.6 `curl(F)` . . . 27
8.2 Example (Optimization) . . . 27
8.3 Example (Lagrange multiplier) . . . 28
8.4 Example (area and plane of a triangle) . . . 29

9 Graphing **29**
9.1 2D graphs . . . 29
9.1.1 Explicit and parametric functions . . . 29
9.1.2 Polar functions . . . 32
9.1.3 Discrete data . . . 33
9.1.4 Implicit plots . . . 34
9.1.5 Example (obtaining output as a graphics file) . . . 35
9.1.6 Example (using a grid and logarithmic axes) . . . 36
9.1.7 Example (setting draw() defaults) . . . 36
9.1.8 Example (including labels in a graph) . . . 37
9.1.9 Example (graphing the area between two curves) . . . 37
9.1.10 Example (creating a Taylor polynomial animation) . . . 38
9.1.11 Options . . . 39
9.1.12 Global options . . . 39
9.1.13 Local options . . . 41
9.2 3D graphs . . . 42
9.2.1 Functions and relations . . . 42
9.2.2 Discrete data . . . 45
9.2.3 Contour plots . . . 45
9.2.4 Options with 2D counterparts . . . 45
9.2.5 Options with no 2D counterparts . . . 46

10 Programming **46**
10.1 Example (Newton's Method) . . . 47
10.2 Example (Euler's Method) . . . 48
10.3 Example (Simpson's Rule) . . . 49

1 Introduction

1.1 About this document

This document discusses the use of Maxima (http://maxima.sourceforge.net/) for solving typical Calculus problems. The latest version of this document can be found at http://maxima.sourceforge.net/documentation.html. Leon Q. Brin is a professor at Southern CT State University, New Haven, CT U.S.A. Feedback is welcome at BrinL1@southernct.edu.

1.2 History and Philosophy

Maxima is an open source computer algebra system (CAS). As such it is free for everyone to download, install, and use! In fact, its (GNU Public) license, or GPL, allows everyone the freedom to modify and distribute it too, as long as its license remains with it unmodified. From the Maxima 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.

During the early days of development, the only user interface available was the command line. This option is still the only one guaranteed to work as advertised. For simplicity and the greatest compatibility, all examples in this document are presented as command line input and output (as would be seen using command line Maxima). However, several independent projects strive to give Maxima a more modern, graphical user interface. One of these projects is wxMaxima, a simple front end that allows modification of previous input and typeset output. All of the examples in this document were produced using wxMaxima. More on that later.

Any CAS may be thought of as a highly sophisticated calculator. It can be used to do any of the types of numerical calculations you might expect of a calculator such as trigonometric, exponential, logarithmic, and arithemtic computations. However, numerical calculation is not the main purpose of a CAS. A CAS' main purpose, and what sets a CAS apart from most calculators, is symbolic manipulation. As such, when asked to divide 36/72, a CAS will respond 1/2 rather than 0.5 unless explicitly commanded to respond with a decimal (called floating point in the computer world) representation. Similarly, $\sin(2)$, π, e, $\sqrt{7}$ and other irrational numbers are interpreted symbolically as $\sin(2)$, π, e, $\sqrt{7}$ and so on rather than their floating point approximations. Computer algebra systems also have the ability to perform "arbitrary precision" calculations. In other words, the user can specify how many decimal places to use in floating point calculations, and does not have to worry much about overflow errors. For example, a CAS will return all 158 digits of 100! when asked. But, as already noted, the real strength of a computer algebra system is the manipulation of variable expressions. For example, a CAS can be used to differentiate $x^2 \sin x$. It will return $2x \sin x + x^2 \cos x$ as it should. Computer algebra systems can accomplish many tasks that were not too long ago relegated to pencil and paper. The purpose of this document is to acquaint the reader with many of the features of Maxima[1] as they may be applied to solving common problems in a standard calculus sequence.

1.3 Using Maxima

Maxima itself is a command line program and can be started by issuing the command `maxima`. For a few, this is the ideal environment for computer algebra. But for most computer users it is unfamiliar and may seem quite arcane. Not by accident, Maxima includes the capability of running as a backend to a graphical user interface (GUI). One such GUI is wxMaxima. As is Maxima, wxMaxima is open source software, freely

[1] All examples were written and executed using Maxima 5.18.1. They should run on all later versions as well.

available for anyone's use. The use of wxMaxima allows for a more modern computing experience with editable inputs, menus, and buttons. For most, this will be the desired environment for using Maxima. A link to wxMaxima and a list of other Maxima-related software can be found at

http://maxima.sourceforge.net/relatedprojects.html

1.4 Reading the examples

Maxima distinguishes user input from its output by labeling each line with either `(%i#)` for input or `(%o#)` for output. It numbers input and output lines consecutively, and the output for a given input line will be labeled with the same number. The "/" at the end of a line indicates that it is continued on the next. For example, this excerpt from a Maxima session shows input 6 being 69^3 with output 6 being $328,509$ (which is 69^3); and it shows the value of 100! which covers 3 lines!

```
(%i6) 69^3;
(%o6)                           328509

(%i7) 100!;
(%o7) 9332621544394415268169923885626670049071596826438162146/
859296389521759999322991560894146397615651828625369792082722 3/
7582511852109168640000000000000000000000000
```

2 Basics

Most basic calculations in Maxima are done as would be done on a graphing calculator. Here are a few simple examples.

```
(%i5) 3-33/72+1/12;
                                 21
(%o5)                            --
                                 8

(%i6) 69^3;
(%o6)                           328509

(%i7) 100!;
(%o7) 9332621544394415268169923885626670049071596826438162146/
8592963895217599993229915608941463976156518286253697920827223/
758251185210916864000000000000000000000000

(%i8) sin(2);
(%o8)                           sin(2)
```

Notice that Maxima does exact (symbolic) calculations whenever possible. In order to force a floating point (decimal) calculation, use the `ev(·,numer)` command, or just include floating point numbers in the expression.

```
(%i8) ev(21/8,numer);
(%o8)                            2.625

(%i9) ev(sin(2),numer);
(%o9)                      0.90929742682568

(%i10) 21.0/8;
(%o10)                           2.625

(%i11) sin(2.0);
(%o11)                     0.90929742682568
```

`ev()` is an example of one Maxima command that is not intuitive. It's not something a user would likely try without somehow being informed of it first. Luckily, most commands needed for common calculus problems are intuitive, or at least not surprising. For example, `sin()`, `cos()`, `tan()` and so on are used for trig functions; `asin()`, `acos()`, `atan()` and so on for inverse trig functions; and `exp()` and `log()` for the natural base exponentials and logarithms. For more advanced computations, `diff()` and `integrate()` are used for differentiation and integration, `div()` and `curl()` for divergence and curl, and `solve()` to solve equations or systems of equations. Examples of these commands are upcoming.

2.1 Basic arithmetic commands

2.1.1 Addition, Subtraction, Multiplication, and Division

The binary operations of addition, subtraction, multiplication, and division are performed by placing the two quantities on opposite sides of the $+$, $-$, $*$, and / sign, respectively. Maxima obeys the standard order of operations so parentheses are used to apply operators in other orders.

```
(%i2) print("Maxima obeys the standard order of operations")$
      2/3+7/8;
      2/(3+7)/8;

Maxima obeys the standard order of operations
                                    37
(%o3)                               --
                                    24

                                    1
(%o4)                               --
                                    40

(%i5) print("Maxima must be told what to do with some symbolic expressions")$
      u+3*u-17*(u-v);
      expand(%);
      (u-v)*(2*u+3*v)*(u+9*v);
      expand(%);
      (u^2-v^2)/(u+v);
      fullratsimp(%);
Maxima must be told what to do with some symbolic expressions
(%o6)                     4 u - 17 (u - v)
(%o7)                        17 v - 13 u
(%o8)                (u - v) (3 v + 2 u) (9 v + u)

                             3        2       2        3
(%o9)                 - 27 v  + 6 u v  + 19 u  v + 2 u

                                  2    2
                                 u  - v
(%o10)                           -------
                                  v + u

(%o11)                            u - v
```

2.1.2 Exponents and Factorials

To raise b to the power p, use the binary operator ^ as in b^p. Use the postfix operator ! for the factorial of a quantity.

```
(%i1)  5^55;
(%o1)              277555756156289135105907917022705078125

(%i2)  121!;
(%o2)  809429852527344373968162284544935082997082306309701607045776233628497660\
04266405217133917739979101827382870741850789049568566634393183827450477162148\
11476507217602230720921600000000000000000000000000000
```

2.1.3 Square and other roots

Use the `sqrt()` command for the square root of a quantity. Use rational exponents for other roots.

```
(%i1) sqrt(100);
(%o1)                              10

(%i2) sqrt(1911805231500);
(%o2)                        54870 sqrt(635)

(%i3) sqrt(x^6);
                                      3
(%o3)                           abs(x)

(%i4) 8^(1/3);
(%o4)                              2

(%i5) 128^(1/4);
                                   1/4
(%o5)                           2 8

(%i6) 34^(1/5);
                                   1/5
(%o6)                            34

(%i7) 34.0^(1/5);
(%o7)                       2.024397458499885

(%i8) sqrt(x)^(1/4);
                                  1/8
(%o8)                            x
```

3 Precalculus

3.1 Trigonometry

3.1.1 Trigonometric functions and π

Use `sin()` to find the sine of an angle, `cos()` for the cosine, `tan()` for the tangent, `sec()` for the secant, `csc()` for the cosecant, and `cot()` for the cotangent. Angles must be given in radians. Of course, an angle measure in degrees may be converted to radians by multiplying by $\frac{\pi}{180}$. Use `%pi` for π.

```
(%i1) cos(%pi/3);
                                   1
(%o1)                              -
                                   2

(%i2) ev(sin(%pi/3),numer);
```

```
(%o2)                        0.86602540378444

(%i3) csc(45*%pi/180);
(%o3)                             sqrt(2)

(%i4) tan(%pi/8);
                                    %pi
(%o4)                           tan(---)
                                     8
```

3.1.2 Inverse trigonometric functions

Use `asin()` to find the Arcsine of a value, `acos()` for the Arccosine, `atan()` for the Arctangent, `asec()` for the Arcsecant, `acsc()` for the Arccosecant, and `acot()` for the Arccotangent. Angles will be given in radians. Of course, an angle measure in radians may be converted to degrees by multiplying by $\frac{180}{\pi}$.

```
(%i1) acos(1/2);
                                   %pi
(%o1)                              ---
                                    3

(%i2) ev(asin(sqrt(3)/2),numer);
(%o2)                        1.047197551196598

(%i3) 180*atan(1)/%pi;
(%o3)                              45

(%i4) acsc(8);
(%o4)                           acsc(8)
```

3.2 Assignment and Function definition

One of the most basic operations of any computer algebra system is variable assignment. A variable may be assigned a value using the : operator. In Maxima, the command `a:3;` sets `a` equal to `3`. But more useful than assigning numerical values to single-letter variables, Maxima allows multiple-letter variables, as do most programming languages. For example, `eqn:2*x^2+3*x-5=0;` sets the variable `eqn` equal to the equation $2x^2 + 3x - 5 = 0$, and `expr:sin(2*x)` sets `expr` equal to the expression $\sin(2x)$.

Similar to assignment is function definition. The primary difference is a function is designed for evaluation at various values of the independent variables where expressions are generally not. In the end, though, which one to use will be a matter of preference more than anything pragmatic. The notation for function definition in Maxima is almost identical to that of pencil and paper mathematics. Simply use function notation with a := where you would use = on paper. For example, `f(x):=2*sin(x^3)+cot(x)` in Maxima is equivalent to $f(x) = 2\sin(x^3) + \cot(x)$ on paper.

```
(%i1) a:3$
       display(a)$
                                    a = 3
(%i3) f(x):=a*x^2+b*x+c;
      expr:a*x^2+b*x+c;
      f(2);
      ev(expr,x=2);
                                     2
(%o3)                      f(x) := a x  + b x + c
                                 2
(%o4)                        3 x  + b x + c
```

```
(%o5)                          c + 2 b + 12
(%o6)                          c + 2 b + 12
(%i7) ev(f(x),b=-5,c=12);
      ev(expr,b=-5,c=12);
                                  2
(%o7)                          3 x  - 5 x + 12
                                  2
(%o8)                          3 x  - 5 x + 12
(%i9) g(x,y):=x^2-y^2;
      g(expr,y);
      g(expr,f(y));
                                         2    2
(%o9)                       g(x, y) := x  - y
                                2           2    2
(%o10)                     (3 x  + b x + c)  - y
                           2           2       2           2
(%o11)               (3 x  + b x + c)  - (3 y  + b y + c)
(%i12) expand(%);
           4        3         2    2  2                4        3        2
(%o12) - 9 y  - 6 b y  - 6 c y  - b  y  - 2 b c y + 9 x  + 6 b x  + 6 c x
                                                                2  2
                                                            + b  x  + 2 b c x
```

3.3 Exponentials and Logarithms

In addition to using the `^` operator for exponentials, Maxima provides the `exp()` function for exponentiation base e, so `exp(x)` is the same as `%e^x`. Maxima only provides the natural (base e) logarithm. Therefore, a useful definition to make is

```
logb(b,x):=log(x)/log(b);
```

for calculating $\log_b(x)$.

```
(%i35) log(%e);
       log(3);

(%o35)                              1
(%o36)                           log(3)

(%i37) expr:log(x*%e^y);
       expr=radcan(expr);

                                        y
(%o37)                           log(x %e )
                                    y
(%o38)                     log(x %e ) = y + log(x)

(%i39) expr:%e^(r*log(x));
       expr=radcan(expr);

                                 r log(x)
(%o39)                         %e
                               r log(x)    r
(%o40)                       %e         = x

(%i41) logb(b,x):=log(x)/log(b);
       a:logb(3,27)$
       a=radcan(a);
```

```
                                        log(x)
(%o41)                    logb(b, x) := ------
                                        log(b)
                               log(27)
(%o43)                         ------- = 3
                               log(3)
(%i44) a:log(30*x^2/z^5);
       ev(a,logexpand=super);
       radcan(a);
                                       2
                                   30 x
(%o44)                         log(-----)
                                     5
                                    z
(%o45)                - 5 log(z) + 2 log(x) + log(30)
(%o46)         - 5 log(z) + 2 log(x) + log(5) + log(3) + log(2)
```

3.4 Constants

The ubiquitous constants π, e, and i are known to Maxima. Use `%pi` for π, `%e` for e, and `%i` for i.

3.5 Solving equations

In precalculus, students learn to solve equations by performing operations equally on both sides of a given equation, ultimately isolating the desired variable. Maxima makes it easy to demonstrate this process electronically. This gives students a way to check their work, and test their equation solving skills. In order to apply a given operation to both sides of an equation, simply assign a variable to the equation and apply the operation to that variable.

```
(%i47) eqn:(3*x-5)/(17*x+4)=2;
                              3 x - 5
(%o47)                        -------- = 2
                              17 x + 4
(%i48) eqn2:eqn*(17*x+4);
(%o48)                   3 x - 5 = 2 (17 x + 4)
(%i49) eqn3:expand(eqn2);
(%o49)                     3 x - 5 = 34 x + 8
(%i56) eqn4:eqn3+5;
(%o56)                       3 x = 34 x + 13
(%i57) eqn5:eqn4-34*x;
(%o57)                         - 31 x = 13
(%i58) eqn6:eqn5/-31;
                                      13
(%o58)                          x = - --
                                      31
(%i64) print("Checking our work:")$
       ev(eqn,eqn6);
       ev(eqn,eqn6,pred);
Checking our work:
(%o65)                             2 = 2
(%o66)                             true
```

Of course when the solution process is not important, Maxima provides a single command for solving equations. Use `solve()` to solve equations or systems of equations. Systems of equations are accepted by Maxima using vector notation. Delimit the system using square brackets ([]), separating equations by commas.

```
(%i6) solve(eqn);
                                                13
(%o6)                                [x = - --]
                                                31
(%i7) print("When it is not obvious for which variable Maxima",
          "should solve, specify it:")$
      solve(a*x^2-b*x+c,x);
When it is not obvious for which variable Maxima
should solve, specify it:
                         2                            2
                  sqrt(b  - 4 a c) - b         sqrt(b  - 4 a c) + b
(%o8)       [x = - --------------------, x = --------------------]
                          2 a                          2 a
(%i9) solve(a*x^2-b*x+c,b);
                                       2
                                    a x  + c
(%o9)                          [b = --------]
                                       x
(%i10) solve([3*x+4*y=c,2*x-3*y=d],[x,y]);
                            4 d + 3 c      2 c - 3 d
(%o10)                 [[x = ---------, y = ---------]]
                               17             17
```

3.5.1 Example (Maxima fails to solve an equation)

Solve for x:

$$2 - \frac{x}{\sqrt{1-x^2}} = 0$$

```
(%i1)   eqn:2-x/sqrt(1-x^2)=0;
        print(“Maxima fails to solve the equation:”)$
        xx:solve(2-x/sqrt(1-x^2)=0);
        print(“But with a little help...”)$
        print(“(squaring both sides)”)$
        xx:solve(xx[1]^2);
        print(“Check results:”)$
        ev(eqn,xx[1]);
        ev(eqn,xx[1],pred);
        ev(eqn,xx[2]);
        ev(eqn,xx[2],pred);


                                       x
(%o1)                      2 - ------------ = 0
                                           2
                                 sqrt(1 - x )

Maxima fails to solve the equation:
                                               2
(%o3)                      [x = 2 sqrt(1 - x )]
But with a little help...
(squaring both sides)
```

```
                                 2            2
(%o6)                  [x = - -------, x = -------]
                              sqrt(5)      sqrt(5)

Check results:
(%o8)                          4 = 0
(%o9)                          false
(%o10)                         0 = 0
(%o11)                          true
```

$x = \dfrac{2}{\sqrt{5}}$ is the only solution.

3.6 Simplification

Simplification of expressions is one of the most difficult jobs for a computer algebra system even though there are established routines for standard simplification procedures. The difficulty is in choosing which simplification procedures to apply when. For example, certain mathematical situations require factoring while others require expanding. A computer algebra system has no way to determine what the situation demands. Therefore, different simplification procedures are available for different purposes. Very little simplification is done automatically. More than one simplification procedure may be applied to a single expression.

Command	**Action**	Examples
`fullratsimp()`	A somewhat generic simplification routine. Start with this. If it does not do what you hope, try one of the more specific routines.	§2.1.1, §8.4
`expand()`	Products of sums and exponentiated sums are multiplied out. Logarithms are not expanded.	§2.1.1, §3.2, §3.5
`factor()`	If the argument is an integer, factors the integer. If the argument is anything else, factors the argument into factors irreducible over the integers.	below
`radcan()`	Simplifies logarithmic, exponential, and radical expressions into a canonical form.	§3.3
`ev(·,logexpand=super)`	Expands logarithms of products, quotients and exponentials.	§3.3
`logcontract()`	Contracts multiple logarithmic terms into single logarithms.	below
`trigsimp()`	Employs the identity $\sin^2 x + \cos^2 x = 1$ to simplify expressions containing tan, sec, etc., to sin and cos.	below
`trigexpand()`	Expands trigonometric functions of angle sums and of angle multiples. For best results, the argument should be expanded. May require multiple applications.	below
`trigreduce()`	Combines products and powers of sines and cosines into sines and cosines of multiples of their argument.	below
`trigrat()`	Gives a canonical simplified form of a trigonometric expression.	below

For hyperbolic trig simplification, use `trigsimp()`, `trigexpand()` and `trigreduce()`.

3.6.1 Example (factoring)

```
(%i7) factor(1001);
(%o7)                              7 11 13
(%i8) factor(4*x^5-4*x^4-13*x^3+x^2-17*x+5);

                                  2           2
(%o8)                (2 x - 5) (x  + 1) (2 x  + 3 x - 1)
```

Note that the juxtaposition of the 7, 11, and 13 implies multiplication.

3.6.2 Example (logcontract)

```
(%i9) logcontract(log(3*x)-2*log(5*y));

                                        3 x
(%o9)                              log(-----)
                                          2
                                      25 y
```

3.6.3 Example (trigsimp)

```
(%i10) trigsimp(tan(x));
                                     sin(x)
(%o10)                               ------
                                     cos(x)

(%i11) trigsimp(tan(x)^2+1);
                                       1
(%o11)                               -------
                                        2
                                     cos (x)

(%i13) trigsimp(csc(x)^2-cot(x)^2);
(%o13)                                 1
```

3.6.4 Example (other trig simplifications)

```
(%i22) expr:sin((a+b)*(a-b))+sin(x)^3*cos(x)^2;

                          2       3
(%o22)                 cos (x) sin (x) + sin((a - b) (b + a))

(%i23) trigexpand(expr);

                          2       3
(%o23)                 cos (x) sin (x) + sin((a - b) (b + a))

(%i24) trigexpand(expand(expr));

                    2       3             2       2          2       2
(%o24)           cos (x) sin (x) - cos(a ) sin(b ) + sin(a ) cos(b )
```

```
(%i25) trigreduce(expr);

                - sin(5 x) + sin(3 x) + 2 sin(x)         2    2
(%o25)          -------------------------------- + sin(a  - b )
                               16

(%i26) trigrat(expr);
                                                          2    2
                sin(5 x) - sin(3 x) - 2 sin(x) + 16 sin(b  - a )
(%o26)        - ------------------------------------------------
                                       16
```

Compare trigexpand(expr) to trigexpand(expand(expr)) to see that trigexpand() is more effective when the arguments of the trig functions are expanded.

3.7 Evaluation

The ev() command is a very powerful command for both simplification and evaluation of expressions. ev() can be used for simplification as in ev(·,logexpand=super) [§3.3] or ev(·,trigsimp) which has exactly the same effect as trigsimp(·), but its main use is for evaluating expressions in different manners. ev(·,numer) [§2] converts numeric expressions to floating point values. ev(·,equation(s)) [§3.2, §3.5] evaluates an expression, substituting the values given by the equation(s). ev(·,pred) [§3.5] evaluates expressions as if they are predicates (true or false). ev() has many more features as well, a few of which will be discussed later.

3.8 Big floats

On occasion it may be desired to compute a floating point value to more precision than the default 16 significant figures. The bfloat() and fpprec() commands give you that capability. For example, if you want to compute π to 100 decimal places, set fpprec (short for floating point precision) to 101 and then enter bfloat(%pi). This will show 101 digits of π (1 to the left of the decimal point and 100 to the right).

```
(%i11) fpprec: 101;
(%o11)                                101
(%i12) bfloat(%pi);
(%o12) 3.1415926535897932384626433832795028841971693993751058209749445923078164\
062862089986280348253421170679b0
```

Notice that bfloats (big floats) are given in scientific notation, using the letter b to separate the coefficient from the exponent.

4 Limits

As with solving equations, Maxima has a single command for computing limits, but is also useful for demonstrating the ideas. One of the first looks at limits often involves a table of values where the independent variable approaches some given value. The table will often show the dependent variable approaching some determinable value, the limit. With the help of the fpprintprec flag and the maplist() command, Maxima can produce useful tables. The floating point print precision (fpprintprec) flag determines how many significant digits of a floating point value should be displayed when printed. The default is 16, so does not make for concise listing. The value of fpprintprec is set using the assignment operator, :, just like any variable. maplist() is used to compute the values of the dependent variable.

```
(%i1) fpprintprec:7$
      f(x):=sin(x)/x;
      a:[1.0,1/4.0,1/16.0,1/64.0,1/256.0,1/1024.0];
```

```
      maplist(f,a);
                                          sin(x)
(%o2)                             f(x) := ------
                                            x
(%o3)      [1.0, 0.25, 0.0625, 0.015625, 0.0039063, 9.765625E-4]
(%o4)         [0.84147, 0.98962, 0.99935, 0.99996, 1.0, 1.0]
```

If only the result is desired, use the `limit()` command. The command requires an expression and a variable with the value it is to approach. For limits to infinity or minus infinity, use `inf` and `minf` respectively. The result may be a number, `ind` (indeterminate but bounded), `inf`, `minf`, `infinity` (complex infinity), or `und` (undefined). One-sided limits may be computed by adding a “plus” or “minus” argument to indicate the right-hand limit and left-hand limit respectively.

```
(%i1) f(x):=sin(x)/x;
      limit(f(x),x,0);
      limit(f(x),x,inf);
                                           sin(x)
(%o1)                              f(x) := ------
                                             x
(%o2)                                     1
(%o3)                                     0
(%i4) f(x):=atan(x);
      limit(f(x),x,minf);
(%o4)                              f(x) := atan(x)
                                          %pi
(%o5)                                   - ---
                                           2
(%i6) f(x):=(x^2-3*x+8)/(x+2);
      limit(f(x),x,-2,minus);
      limit(f(x),x,-2,plus);
      limit(f(x),x,-2);
                                         2
                                        x  - 3 x + 8
(%o6)                           f(x) := ------------
                                           x + 2
(%o7)                                   minf
(%o8)                                    inf
(%o9)                                    und
(%i10) limit(sin(1/x),x,0);
(%o10)                                    ind
(%i11) limit((sqrt(3+2*x)-sqrt(3-x))/x,x,0);
                                            3
(%o11)                                  ---------
                                        2 sqrt(3)
```

4.1 Example (limit of a difference quotient)

Let $f(x) = x^3 \tan x$. Find $\lim_{h \to 0} \dfrac{f(x+h) - f(x)}{h}$.

```
(%i1) f(x):=x^3*tan(x);
      dq:(f(x+h)-f(x))/h;
      limit(dq,h,0);
                                          3
(%o1)                             f(x) := x  tan(x)
```

```
                             3                3
                     (x + h)  tan(x + h) - x  tan(x)
(%o2)                -------------------------------
                                    h

                                                3
                                2              x
(%o3)                      3 x  tan(x) + -------
                                                2
                                             cos (x)
```

$\lim_{h \to 0} \frac{f(x+h)-f(x)}{h} = 3x^2 \tan(x) + \frac{x^3}{\cos^2(x)}$.

5 Differentiation

As shown in example 4.1, Maxima is capable of computing derivatives based on the definition of derivative. Of course this is not at all the best way to do so when it is only the derivative, and not the process, that is important. Maxima provides a single differentiation command, `diff()`, that computes derivatives in a much more efficient manner. The arguments to diff() are the expression to be differentiated, the variable with respect to which to differentiate, and optionally, a positive integer indicating how many times to differentiate with respect to that variable. More than one variable/number pair may be specified to compute mixed partial derivatives. Maxima makes no distinction between derivatives and partial derivatives.

```
(%i1) diff(%e^sqrt(sin(x)),x);
                                         sqrt(sin(x))
                                cos(x) %e
(%o1)                           ---------------------
                                   2 sqrt(sin(x))
(%i2) diff(f(x)/g(x),x);
                             d                 d
                             -- (f(x))   f(x) (-- (g(x)))
                             dx                dx
(%o2)                        --------- - ----------------
                               g(x)            2
                                              g (x)
(%i3) f(x):=sin(x)$
      diff(f(x)/g(x),x);
      ratsimp(%);
                                               d
                                        sin(x) (-- (g(x)))
                              cos(x)           dx
(%o4)                         ------ - ------------------
                               g(x)            2
                                              g (x)

                                   d
                            sin(x) (-- (g(x))) - g(x) cos(x)
                                   dx
(%o5)                     - --------------------------------
                                             2
                                            g (x)
(%i6) f(x,y,z):=x^2*y^3*z^4$
      diff(f(x,y,z),x,1,z,2);
```

```
                                         3  2
(%o7)                              24 x y  z
```

5.1 Example (related rates)

A spherical balloon is releasing air in such a way that at the time its radius is 6 inches, its volume is decreasing at a rate of 3 cubic inches per second. At what rate is its radius decreasing at this time?

```
(%i1) eqn:V(t)=4/3*%pi*r(t)^3;
                                                3
                                         4 %pi r (t)
(%o1)                            V(t) = -----------
                                              3
(%i2) diff(eqn,t);
                        d                       2     d
(%o2)                   -- (V(t)) = 4 %pi r (t) (-- (r(t)))
                        dt                            dt
(%i3) ev(%,diff(r(t),t)=drdt,diff(V(t),t)=-3,r(t)=6);
(%o3)                            - 3 = 144 %pi drdt
(%i4) solve(%);
                                               1
(%o4)                             [drdt = - ------]
                                             48 %pi
```

Its radius is decreasing at $\frac{1}{48\pi}$ inches per second.

NOTE: In `(%i3)`, the substitution `diff(r(t),t)=drdt` is necessary to prevent `diff(r(t),t)` from evaluating to zero when the substitution `r(t)=2` is made. Along these same lines, the order in which these substitutions is listed, relative to one another, in `(%i3)` is critical. The `ev()` command

```
(%i3) ev(%,diff(V(t),t)=-3,r(t)=6,diff(r(t),t)=drdt);
```

would result in the equation `-3=0` because `diff(r(t),t)` will have already been evaluated to zero by the time the substitution `diff(r(t),t)=drdt` is considered.

5.2 Example(optimization)

A cylindrical can with a bottom but no lid is to be made out of 300π cm^2 of sheet metal. Find the maximum volume of such a can.

```
(%i23) Vol(r,h):=%pi*r^2*h;
                                                   2
(%o23)                           Vol(r, h) := %pi r  h
(%i24) SA(r,h):=%pi*r^2+2*%pi*r*h;
                                             2
(%o24)                      SA(r, h) := %pi r  + 2 %pi r h
(%i25) solve(SA(r,h)=300*%pi,h);
                                           2
                                          r  - 300
(%o25)                            [h = - --------]
                                            2 r
```

```
(%i26) V:ev(Vol(r,h),%[1]);
                                        2
                              %pi r (r  - 300)
(%o26)                      - ----------------
                                     2
(%i27) eqn:diff(V,r)=0;
                                          2
                               2   %pi (r  - 300)
(%o27)                 - %pi r  - -------------- = 0
                                          2
(%i28) critical:solve(eqn);
(%o28)                      [r = - 10, r = 10]
(%i29) ev(V,critical[2]);
(%o29)                           1000 %pi
```

1000π cm^3.

5.3 Example (second derivative test)

Use the second derivative test to find the relative extrema of $f(x) = \sec x$ on $(-\pi/2, \pi/2)$.

```
(%i1) f(x):=sec(x);
      first:diff(f(x),x);
      second:diff(f(x),x,2);
      critical:solve(first=0);
(%o1)                         f(x) := sec(x)
(%o2)                          sec(x) tan(x)
                                      2        3
(%o3)                    sec(x) tan (x) + sec (x)
`solve' is using arc-trig functions to get a solution.Some solutions will be lost.
(%o4)                        [x = 0, x = asec(0)]
(%i5) ev(second,critical[1]);
      ev(f(x),critical[1]);
(%o5)                                1
(%o6)                                1
```

$(0, 1)$ is a local minimum.

6 Integration

6.1 Riemann Sums

Perhaps the most tedious part of learning the calculus is computing Riemann sums. Here is a fantastic opportunity to involve the computer. After all, computers are much more adept at tedious computation than we are. Consider the following two methods for computing a Riemann sum using Maxima. The first method is very utilitarian. It gets the job done, but doesn't do a very good job of illustration.

```
(%i1) fpprintprec:5$
      f(x):=1+3*cos(x)^2/(x+5);
      a:2$
      b:4$
      n:12$
      rightsum:ev(sum((b-a)/n*f(a+i*(b-a)/n),i,1,n),numer);
```

```
                                             2
                                        3 cos (x)
(%o2)                       f(x) := 1 + ---------
                                          x + 5
(%o6)                              2.5392
```

This second method does a much better job of illustrating the area computation. It uses the package `draw` to take care of the graphics. To load package draw, include the line

```
load(draw);
```

somewhere before the graphics are needed. The results of the following code are shown in Figure 1.

```
(%i322) fpprintprec:5$
        f(x):=1+3*cos(x)^2/(x+5);
        a:2$
        b:4$
        n:12$
        print("Left endpoints:")$
        leftend:makelist(a+i*(b-a)/n,i,0,n-1);
        print("Right endpoints:")$
        rightend:makelist(a+i*(b-a)/n,i,1,n);
        print("Heights:")$
        height:makelist(ev(f(leftend[i]),numer),i,1,n);
        area:sum((rightend[i]-leftend[i])*height[i],i,1,n)$
        print("Riemann sum =",area)$
        /* Create and display graphics */
        rects:makelist(rectangle([leftend[i],0],[rightend[i],height[i]]),i,1,n)$
        graph:[explicit(f(x),x,a-(b-a)/12,b+(b-a)/12)]$
        options:[yrange=[0,2]]$
        scene:append(options,rects,graph)$
        apply(draw2d,scene)$
                                             2
                                        3 cos (x)
(%o323)                     f(x) := 1 + ---------
                                          x + 5
Left endpoints:
                      13  7  5  8  17     19  10  7  11 23
(%o328)           [2, --, -, -, -, --, 3, --, --, -, --, --]
                      6   3  2  3  6      6   3   2  3   6
Right endpoints:
                   13  7  5  8  17     19  10  7  11  23
(%o330)           [--, -, -, -, --, 3, --, --, -, --, --, 4]
                   6   3  2  3  6      6   3   2  3   6
Heights:
(%o332) [1.0742, 1.1319, 1.1952, 1.2567, 1.3095, 1.3477, 1.3675, 1.3671,
Riemann sum = 2.5278
```

6.2 Antiderivatives

To calculate the antiderivative of an expression, use the `integrate()` command. As might be expected, `integrate()` takes the integrand, the variable against which to integrate, and limits of integration, if any, as its arguments. So, some typical examples of its usage are as follows.

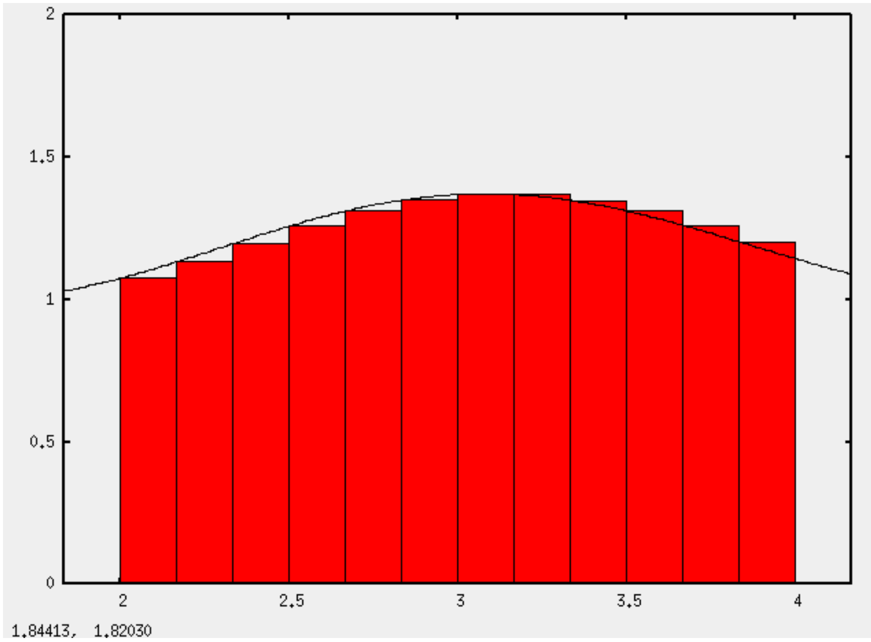


Figure 1: Rieman Sum

```
(%i5) 'integrate(x,x)=integrate(x,x);

                                   /         2
                                   [        x
(%o5)                              I x dx = --
                                   ]        2
                                   /

(%i7) 'integrate(exp(t),t,2,log(12))=integrate(exp(t),t,2,log(12));

                              log(12)
                             /
                             [          t              2
(%o7)                        I         %e  dt = 12 - %e
                             ]
                             /
                              2

(%i8) 'integrate(x^2*y-sqrt(x+y),y)=integrate(x^2*y-sqrt(x+y),y);

                 /                                2  2             3/2
                 [   2                           x  y     2 (y + x)
(%o8)            I (x  y - sqrt(y + x)) dy = ----- - ------------
                 ]                                 2          3
                 /
```

Note that the constant of integration is not reported. Also note the use of the single quote before the `integrate()` command. This tells Maxima to simply display the integral instead of evaluate it, also known as the noun form. The single quote can be used before any command in order to suppress evaluation. Some more involved examples are given below.

6.2.1 Trig substitution

Use trig substitution to evaluate $\int_0^1 t^3\sqrt{1+t^2}\,dt$.

```
(%i92) print("Maxima can simply integrate...")$
       'integrate(t^3*sqrt(1+t^2),t,0,1)=integrate(t^3*sqrt(1+t^2),t,0,1);
       print("...or Maxima can be used to illustrate the steps:")$
       integrand:t^3*sqrt(1+t^2)$
       subt:cos(h)$
       subintegrand:ev(integrand,t=subt)*diff(subt,h)$
       lower:ev(h,solve(subt=0))$
       upper:ev(h,solve(subt=1))$
       'integrate(integrand,t,0,1)='integrate(subintegrand,h,lower,upper);
       print("which of course evaluates to")$
       integrate(subintegrand,h,lower,upper);
```

```
Maxima can simply integrate...

                      1
                     /
                     [   3       2           2 sqrt(2)   2
(%o93)               I  t  sqrt(t  + 1) dt = --------- + --
                     ]                          15       15
                     /
                      0

...or Maxima can be used to illustrate the steps:
`solve' is using arc-trig functions to get a solution.Some solutions will be lost.
`solve' is using arc-trig functions to get a solution.Some solutions will be lost.

                                %pi
                                ---
         1                       2
        /                       /
        [   3       2           [       3            2
(%o100) I  t  sqrt(t  + 1) dt = I    cos (h) sqrt(cos (h) + 1) sin(h) dh
        ]                       ]
        /                       /
         0                       0

which of course evaluates to

                               2 sqrt(2)   2
(%o102)                        --------- + --
                                  15       15
```

6.2.2 Integration by parts

Use integration by parts to evaluate $\int \sin(\log(x))\,dx$.

```
(%i1) u:sin(log(x))$
      dv:1$ f(x):=u*dv$
      'integrate(f(x),x);
      integrate(f(x),x);
```

```
                                 /
                                 [
(%o4)                            I sin(log(x)) dx
                                 ]
                                 /

                           x (sin(log(x)) - cos(log(x)))
(%o5)                      -----------------------------
                                         2
(%i6) du:diff(u,x);
      v:integrate(dv,x);
                                    cos(log(x))
(%o6)                               -----------
                                         x

(%o7)                                    x
(%i8) u*v-'integrate(v*du,x);
      u*v-integrate(v*du,x);

                                         /
                                         [
(%o8)                    x sin(log(x)) - I cos(log(x)) dx
                                         ]
                                         /

                                   x (sin(log(x)) + cos(log(x)))
(%o9)              x sin(log(x)) - -----------------------------
                                                 2

(%i10) ev(equal(%o5,%o9),pred);
(%o10)                                 true
```

6.2.3 Partial fractions

Maxima has a simple command for finding partial fraction decompositions aptly named `partfrac()`. If the fraction to decompose consists of only one variable, the command may be called in the form

`partfrac(expression)`.

But if there is more than one variable, the variable of interest must be specified as in

`partfrac(expression,variable)`.

In the following example, Maxima decomposes $\frac{3n^2+2n}{n^3-3n^2+2n-6}$ (with respect to n) and

$$\frac{2xy^3-2y^3+6x^2y^2+5xy^2-4x^3y-2x^2y-8x^4}{xy^3-x^2y^2-2x^3y}$$

with respect to x and with respect to y.

```
(%i27) numerator:3*n^2+2*n$
       denominator:n^3-3*n^2+2*n-6$
       numerator/denominator=partfrac(numerator/denominator);
       numerator:2*x*y^3-2*y^3+6*x^2*y^2+5*x*y^2-4*x^3*y-2*x^2*y-8*x^4$
       denominator:x*y^3-x^2*y^2-2*x^3*y$
```

```
       numerator/denominator; print("equals")$
       partfrac(numerator/denominator,x);
       print("equals")$
       partfrac(numerator/denominator,y);

                                      2
                                   3 n  + 2 n          2        3
(%o29)                       ------------------ = ------ + -----
                              3      2               2      n - 3
                             n  - 3 n  + 2 n - 6   n  + 2

                         3      3      2  2        2      3        2        4
                   2 x y  - 2 y  + 6 x  y  + 5 x y  - 4 x  y - 2 x  y - 8 x
(%o32)             --------------------------------------------------------
                                        3    2  2      3
                                     x y  - x  y  - 2 x  y

equals

                                      3       2 y     4 x   2
(%o34)                              ----- - ------- + --- - -
                                    y + x   2 x - y    y    x

equals

                                   3       4 x     4 x   2 x - 2
(%o36)                           ----- + ------- + --- + -------
                                 y + x   y - 2 x    y       x
```

6.2.4 Multiple integrals

Multiple integrals are evaluated using multiple calls to the `integrate()` function. The calls may be nested as the example illustrates in calculating

$$\int_0^1 \int_0^{2x} y e^{x^3}\, dy\, dx$$

```
(%i17) print("Maxima can handle nested integrate() commands.")$
       exmpl:'integrate('integrate(y*exp(x^3),y,0,2*x),x,0,1)$
       exmpl=ev(exmpl,nouns);
       print("For clarity, however, it may be simpler to use separate commands:")$
       inner:integrate(y*exp(x^3),y,0,2*x)$
       intermediate:'integrate(inner,x,0,1)$
       print(exmpl,"=",intermediate,"=",ev(intermediate,nouns))$

Maxima can handle nested integrate() commands.

                                  1       2 x
                                 /     3 /
                                 [    x  [                  %e   1
(%o19)                           I  %e   I    y dy dx = 2 (-- - -)
                                 ]       ]                  3    3
                                 /       /
                                  0       0

For clarity, however, it may be simpler to use separate commands:
```

```
 1       2 x             1
 /     3 /               /        3
 [    x  [               [   2   x          %e   1
 I  %e   I    y dy dx = 2 I  x  %e   dx = 2 (-- - -)
 ]       ]               ]                  3    3
 /       /               /
 0       0               0
```

7 Series

7.1 Scalar Series

Sums of scalars can be calculated using the `sum()` command. The `sum()` command requires a formula for the summands, the variable with respect to which the sum is to be calculated, a lower limit, and an upper limit as arguments. The upper limit may be ∞, but there is no guarantee Maxima will be able to evaluate any given infinite series. Infinity is denoted by `inf` in Maxima. As a first example, the sum

$$\sum_{i=1}^{3} i$$

would be written `sum(i,i,1,3)`, as shown below. The first `i` is the formula and the second `i` indicates the variable whose limits are 1 and 3.

```
(%i127) sum(i,i,1,3);
(%o127)                             6
```

When the limits of the summation are specific integers, as above, the default action is to evaluate the limit. However, when either one of the limits is variable or infinity, the default action is to return the noun form. To force an evaluation of the sum, use the `ev(·,simpsum)` command.

```
(%i129) sum(i,i,1,n);
        ev(%,simpsum);
                                     n
                                    ====
                                    \
(%o129)                              >    i
                                    /
                                    ====
                                    i = 1

                                     2
                                    n  + n
(%o130)                             ------
                                      2

(%i140) exmpl:6*sum(1/i^2,i,1,inf)$
        exmpl=ev(exmpl,simpsum);
                                inf
                                ====
                                \      1        2
(%o141)                       6  >     -- = %pi
                                /       2
                                ====   i
                                i = 1
```

Of course Maxima can handle much more complicated sums such as this one of Ramanujan's (to a finite number of terms):

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!(1103 + 26390n)}{(n!)^4 \cdot 396^{4n}}$$

```
(%i25) fpprec:60$
       total:sqrt(8)/9801*'sum(((4*n)!*(1103+26390*n))/((n!)^4*396^(4*n)),n,0,6)$
       reciprocalNoun:1/total$
       reciprocalEv:ev(reciprocalNoun,nouns)$
       print(reciprocalNoun)$ print("     equals")$
       print(reciprocalEv)$ print("     which is approximately")$
       print(bfloat(reciprocalEv))$ print("     pi is approximately")$
       print(bfloat(%pi))$

                  9801
 -----------------------------------------
              6
             ====
             \      (26390 n + 1103) (4 n)!
(2 sqrt(2))   >     -----------------------
             /               4 n   4
             ====         396    n!
             n = 0

    equals

   155495205271253997199430020302790505394544386181106983690567 68
---------------------------------------------------------------------
349987175974771049984276898878450737381678902268863173904792 5 sqrt(2)

    which is approximately

3.14159265358979323846264338327950288419716939937510582102093b0

     pi is approximately

3.14159265358979323846264338327950288419716939937510582097494b0
```

7.2 Taylor Series

Maxima has two commands for calculating Taylor Series: one for computing the first several terms, `taylor()`, and one for determining a summation formula, `powerseries()`. Each command takes an expression (function), the variable of expansion, and a point about which to expand as arguments. Additionally, the `taylor()` command requires the number of terms to compute.

```
(%i54) fn:1/(1-x)^2$
       taylor(fn,x,0,8);
       niceindices(powerseries(fn,x,0));

                             2       3       4       5       6       7       8
(%o55)/T/ 1 + 2 x + 3 x  + 4 x  + 5 x  + 6 x  + 7 x  + 8 x  + 9 x  + . . .
```

```
                                 inf
                                 ====
                                 \              i
(%o56)                            >    (i + 1) x
                                 /
                                 ====
                                 i = 0

(%i57) fn:atan(x)$
       taylor(fn,x,0,8);
       niceindices(powerseries(fn,x,0));

                                  3    5    7
                                 x    x    x
(%o58)/T/                    x - -- + -- - -- + . . .
                                 3    5    7

                               inf
                               ====       i  2 i + 1
                               \     (- 1)  x
(%o59)                          >    ---------------
                               /         2 i + 1
                               ====
                               i = 0

(%i60) fn:exp(x^2)$
       taylor(fn,x,0,8);
       niceindices(powerseries(fn,x,0));

                                     4    6    8
                                2   x    x    x
(%o61)/T/                  1 + x  + -- + -- + -- + . . .
                                    2    6    24

                                    inf
                                    ====   2 i
                                    \     x
(%o62)                               >    ----
                                    /      i!
                                    ====
                                    i = 0

(%i69) fn:cos(sin(x))$
       taylor(fn,x,0,8);
       niceindices(powerseries(fn,x,0));

                           2      4       6        8
                          x    5 x    37 x    457 x
(%o70)/T/             1 - -- + ---- - ----- + ------ + . . .
                          2     24     720    40320

(%o71)                           Unable to expand

(%i39) fn:sin(x)$
```

```
taylor(fn,x,%pi/2,8);
niceindices(powerseries(fn,x,%pi/2));

                 %pi 2        %pi 4        %pi 6        %pi 8
             (x - ---)    (x - ---)    (x - ---)    (x - ---)
                  2            2            2            2
(%o40)/T/ 1 - ---------- + ---------- - ---------- + ---------- + . . .
                2            24           720          40320

                            inf        i      %pi 2 i
                            ====  (- 1)  (x - ---)
                            \                  2
(%o41)                       >    ------------------
                            /           (2 i)!
                            ====
                            i = 0
```

8 Vector Calculus

8.1 Package vect

Vectors in Maxima are denoted by enclosure in square brackets, `[]`. Basic manipulations such as the sum and difference and scalar multiplication of vectors are part of the standard library. The magnitude of a vector is not defined, so a useful definition to make is

```
Norm(a):=sqrt(a.conjugate(a));
```

for calculating magnitudes. This definition will work for vectors of both real and imaginary quantities. Common vector calculus manipulations are not part of the standard library either. They are included in the `vect` package. To use `vect`, put the line

```
load(vect);
```

at some point before the vector calculus is needed.

Package `vect` supplies vector calculus functions such as `div`, `grad`, `curl`, `Laplacian`, dot product (`.`) and cross product (`~`). It (re)defines the dot product to be commutative. The default form for all functions except the dot product is the noun form. This means they will simply be regurgitated as inputted. They will not be evaluated unless explicitly forced. Therefore it is useful to define the generic evaluation function

```
evalV(v):=ev(express(v),nouns);
```

for use when evaluation is required. `express(v)` alone is also sometimes useful. Try it out to see what it does.

The default coordinate system is the Cartesian coordinate system in the three variables x, y, z, and affects `grad`, `div`, `curl`, and `Laplacian`. Hence, $\text{grad}(t^2 - 6s^3)$ will evaluate to $[0, 0, 0]$ and $\text{grad}(x^3 - 3y^2)$ will evaluate to $[3x^2, -6y, 0]$ by default. To access a particular component of a vector, follow the vector with the index of the component (starting with 1 for the first component) in square brackets.

```
(%i1) evalV(grad(t^2-6*s^3));
(%o1) [0,0,0]
(%i2) evalV(grad(x^3-3*y^2));
(%o2) [3x^2,-6y,0]
(%i3) %[2];
(%o3) -6y
```

To change the coordinate system, use the `scalefactors()` command. For example, to work in $\mathbb{R}^2$ with independent variables x and y, use the command `scalefactors([[x,y],x,y])`. To work in elliptical coordinates u and v, use `scalefactors([[(u^2-v^2)/2,u*v],u,v])`.

8.1.1 `u~v`

Returns the cross product of vectors `u` and `v`.

8.1.2 `u.v`

Returns the dot product of vectors `u` and `v`.

8.1.3 `grad(f)`

Returns the gradient of function `f`.

8.1.4 `laplacian(f)`

Returns the Laplacian of function `f`.

8.1.5 `div(F)`

Returns the divergence of vector field `F`.

8.1.6 `curl(F)`

Returns the curl of vector field `F`.

8.2 Example (Optimization)

Find the maximum and minimum values of $f(x, y) = 3x - 4y$ subject to the constraint $x^2 + 2y = 1$.

```
(%i1)   f(x,y):=3*x-4*y;
        constraint:x^2+2*y=1;
        subfory:solve(constraint,y);
        fofx:ev(f(x,y),subfory);
        eqn:diff(fofx,x)=0;
        criticalx:solve(eqn);
        criticaly:ev(subfory,criticalx[1]);
        extreme:ev(f(x,y),criticalx,criticaly);
        testpoint:f(0,0);
        print("Since",testpoint,">",extreme,",",extreme,"is the minimum value.")$
        print("There is no maximum value.")$

(%o1)                   f(x, y) := 3 x - 4 y

                                  2
(%o2)                      2 y + x  = 1

                                   2
                                  x  - 1
(%o3)                     [y = - ------]
                                    2

                               2
(%o4)                    2 (x   - 1) + 3 x
(%o5)                       4 x + 3 = 0

                                    3
(%o6)                         [x = - -]
                                    4
```

```
                                        7
(%o7)                              [y = --]
                                        32

                                       25
(%o8)                                - --
                                       8

(%o9)                                  0

            25     25
Since 0 > - -- , - -- is the minimum value.
            8      8

There is no maximum value.
```

8.3 Example (Lagrange multiplier)

Find the maximum and minimum values of $f(x, y) = 2x + y$ subject to the constraint $x^2 + y^2 = 1$.

```
(%i1)   f(x,y):=2*x+y;
        g(x,y):=x^2+y^2-1;
        delf:evalV(grad(f(x,y)));
        delg:evalV(lambda*grad(g(x,y)));
        eq1:delf[1]=delg[1];
        eq2:delf[2]=delg[2];
        soln:solve([eq1,eq2,g(x,y)=0]);

(%o1)   f(x, y) := 2 x + y
                    2    2
(%o2)   g(x, y) := x  + y  - 1
(%o3)   [2, 1]
(%o4)   [2 x lambda, 2 y lambda]
(%o5)   2 = 2 x lambda
(%o6)   1 = 2 y lambda

                1              sqrt(5)          2
(%o7)   [[y = -------, lambda = -------, x = -------],
              sqrt(5)              2         sqrt(5)

                    1                sqrt(5)              2
             [y = - -------, lambda = - -------, x = - -------]]
                    sqrt(5)                2             sqrt(5)

(%i8)   print("Extreme values:")$
        ev(f(x,y),soln[1]);
        ev(f(x,y),soln[2]);

Extreme values:
                                     5
(%o9)                             -------
                                  sqrt(5)

                                     5
(%o10)                          - -------
                                  sqrt(5)
```

8.4 Example (area and plane of a triangle)

Find the area of the triangle with vertices $(1, 3, 4)$, $(5, -3, 9)$ and $(9, -11, 8)$ and an equation of the plane containing it.

```
(%i1)   P:[1,3,4]$
        Q:[5,-3,9]$
        R:[9,-11,8]$
        PQ:Q-P;
        PR:R-P;
        n:evalV(PQ~PR);
        area:Norm(n)/2;
        eqn:n.([x,y,z]-P)=0;
        fullratsimp(eqn);

(%o4)                          [4, - 6, 5]
(%o5)                         [8, - 14, 4]
(%o6)                         [46, 24, - 8]
(%o7)                           sqrt(689)
(%o8)        - 8 (z - 4) + 24 (y - 3) + 46 (x - 1) = 0)
(%o9)                - 8 z + 24 y + 46 x - 86 = 0
```

Area $= \sqrt{689}$ and an equation of the plane is $46x + 24y - 8z = 86$.

9 Graphing

One of the most common practices in studying the calculus is sketching graphs. Maxima supplies a very rich set of commands for producing graphs of functions and relations. The examples and explanations in this section all refer to the use of Maxima's `draw` package which relies on gnuplot version 4.2 or later. If you have an older version of gnuplot, you will have to use Maxima's old school plotting routines (not covered in this document). If you don't know what version of gnuplot you have, try the routines in this section. If they work, you are all set. If not, you may be able to get some help from the Maxima Documentation website:

http://maxima.sourceforge.net/documentation.html

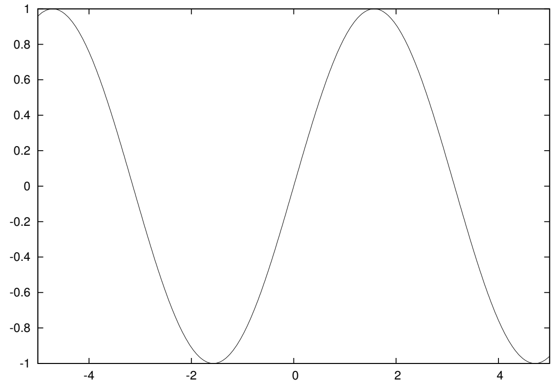
Basic usage of the `draw` package is explained in §9.1.1-§9.1.4. Further information and additional commands are covered in the examples of §9.1.5-§9.1.10. Finally, a detailed look at common graphing options is covered in §9.1.11-§9.1.13.

9.1 2D graphs

9.1.1 Explicit and parametric functions

The `draw2d()` command is used to plot functions or expressions of one variable. The arguments must contain the function(s) to be graphed, and may include options. Options are either global or not. Global options may be set once per plot, and apply to all functions being plotted by that command. Global options may be placed anywhere in the sequence of arguments. Local options apply only to the functions that follow them in the list of arguments. Local arguments may be specified any number of times in a single call to `draw2d()`. One or more of the arguments must be the function to plot. It may be explicit, implicit, polar, parametric, or points. Here are some basic examples. Remember, the `draw2d()` command is part of the `draw` package, so it will have to be loaded before it can be used.

```
See Figure 2
   Example 1:
(%i2) load("draw")$
      draw2d(explicit(sin(x),x,-5,5))$
```



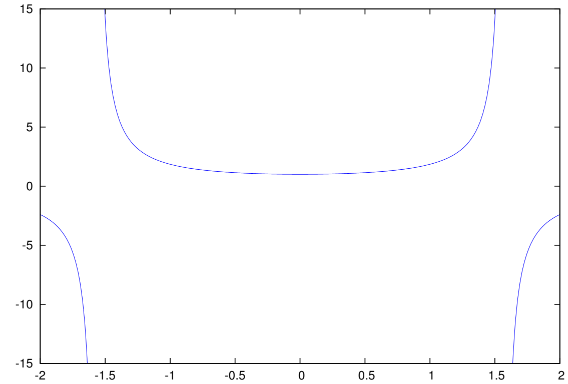


Figure 2: Maxima plots of $\sin(x)$ and $\sec(x)$.

```
   Example 2:
(%i3) draw2d(
          color=blue,
          explicit(sec(v),v,-2,2),
          yrange=[-15,15]
      )$
```

Here are a couple more involved examples. Explanations for these four examples follow.

```
See Figure 3
   Example 3:
(%i12) load(“draw”)$
       expr:x^2$
       F(x) := if x<0 then x^4-1 else 1-x^5$
       draw2d(
           key="x",
           color="blue",
           explicit(expr,x,-1,1),
           key="x^3",
           color="red",
           explicit(x^3,x,-1,1),
           key="F(x)",
           color="magenta",
           explicit(F(x),x,-1,1),
           yrange=[-1.5,1.5]
       )$

   Example 4:
(%i2)crv1:parametric(cos(t)/2,(sin(t)-0.8)/3,t,-7*%pi/8,0)$
     crv2:parametric(cos(t),sin(t),t,-%pi,%pi)$
     crv3:parametric(0.35+cos(t)/5,0.4+sin(t)/5,t,-%pi,%pi)$
     crv4:parametric(-0.35+cos(t)/5,0.4+sin(t)/5,t,-9*%pi/8,%pi/8)$
     draw2d(
         xrange=[-1.2,1.2],
         yrange=[-1.2,1.2],
         line_width=3,
         color=red,
         proportional_axes=xy,
         xaxis=true,
```

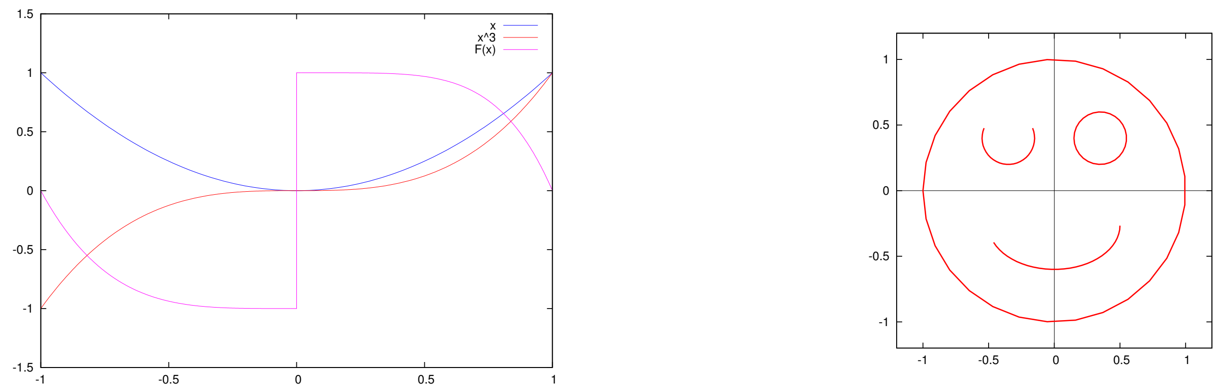


Figure 3: A couple more involved plots.

```
        xaxis_type=solid,
        yaxis=true,
        yaxis_type=solid,
        crv1,crv2,crv3,crv4
)$
```

As seen in the first example, the most basic way to graph a function using `draw2d()` is the form

`draw2d(function);`

The example shows a graph of an explicitly defined function, $f(x) = \sin(x)$. The name of the independent variable and its interval domain must be specified within the call to `explicit()`. In the example,

`explicit(sin(x),x,-5,5)`

represents the function $f(x) = \sin(x)$ over the domain $[-5, 5]$. In this case, the displayed range (y-values) will be decided by Maxima. However, when the dependent variable is unbounded over the specified domain or a specific range for the dependent variable is desired, it is best to supply a `yrange` as in the second example,

`draw2d(color=blue,explicit(sec(v),v,-2,2),yrange=[-15,15]);`

Note that the independent variable may take any name. The third example shows how to plot more than one function on the same set of axes. You simply list more than one function in the `draw2d()` command. Maxima will not by default plot each function in a different color nor include a legend identifying which graph is which. In order to produce a useful legend, each function should be given a `key` and a `color`. The `key` is the name that will be used for the function in the legend and the `color` is of course the color that will be used to graph the function. The `key` and `color` options are not global options, so their placement in the sequence of arguments matters. Each one applies to any function that follows it up to the point where the option is given again. In contrast, the `yrange` option is global. It will apply to the whole graph no matter how many functions are being plotted. As a global option, it may only be specified once, and its placement in the sequence of arguments is immaterial. The fourth example demonstrates how to make a parametric plot and illustrates a few more options that are available for customizing the output. The basic form for a parametric plot is

`draw2d(parametric(x(t),y(t),trange));`

For example, the command `draw2d( parametric( cos(t), sin(t), t, -%pi, %pi ) );` ostensibly plots the unit circle (See figure 4, left side). However, due to the aspect ratio, it is oblong when it should not be. And upon close inspection, you may notice it is a 28-gon, not a circle! For this size graphic, that may not be a problem, but for larger plots, the polygonal shape will be clear. These problems can be fixed by adding options to the draw command. Plot options are always specified in the `option = value` format. The

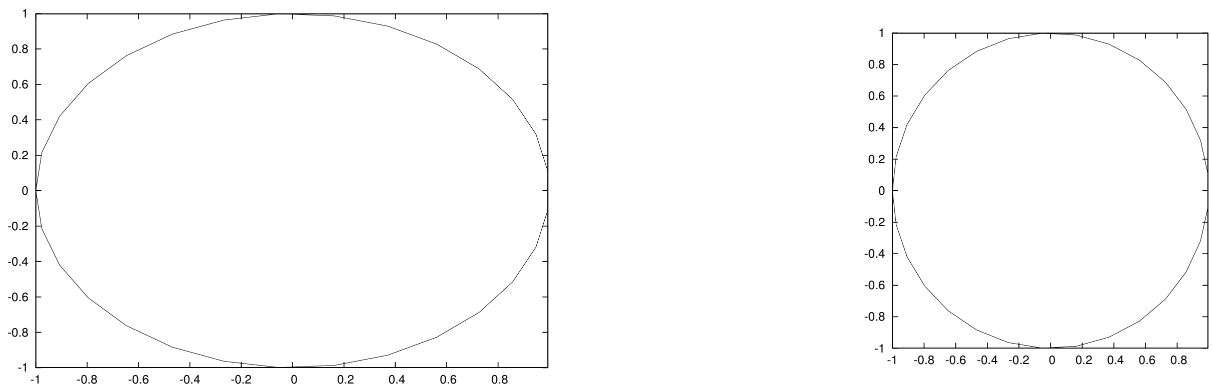


Figure 4: This is a circle?

option `proportional_axes = xy` will fix the aspect ratio so the resulting figure actually appears circular. And the option `nticks = 80` will force Maxima to draw an 80-gon which will appear much more circular. Here is the complete command: `draw2d( nticks = 80, parametric( cos(t), sin(t), t, -%pi, %pi ), proportional_axes = xy);` Note well, the placement of the `nticks = 80` option is important. Since it is not a global option, it must come before the graph to which it is to apply. See the results of both circle plots in figure 4. There are many other plot options, a few of which are used in the fourth example. In particular, the smiley face plot (the fourth example) uses the options

- `line_width = 3`
- `xaxis = true`
- `xaxis_type = solid`
- `xrange = [-1.2,1.2]`

The `line_width` option is of course used to modify the width of the plotted line. The default width is 1, but the line width may be set to any positive integer. The example shows how to draw a solid x-axis using the options `xaxis = true` and `xaxis_type = solid`. Finally, the `xrange` tells Maxima what values of the independent variable to show on the horizontal axis. Note that this interval is independent of the specified domain of any explicit function or the implied domain of any parametric function. Of course, the y-axis is controlled analogously.

9.1.2 Polar functions

`draw2d()` has the capability of graphing polar functions using the `polar()` construct. The `polar()` call is used just as the `explicit()` call except that the independent variable is interpreted as the angle and the dependent variable the radius. For example, `polar ( 3, th, 0, 2*%pi )` specifies the circle of radius 3 centered at the origin. Another simple example is the hyperbolic spiral $r(\theta) = 10/\theta$. The following example graphs this function for $\theta \in [1, 15\pi]$, once using all the default options, and then again using some reasonable options to make the graph presentable.

```
(%i28) load(“draw”)$
(%i29) draw2d(polar(10/theta,theta,1,15*%pi))$
(%i30) draw2d(
          user_preamble="set grid polar",
          color=red,
          line_width=3,
          xrange=[-4,4],
          yrange=[-3,3],
```

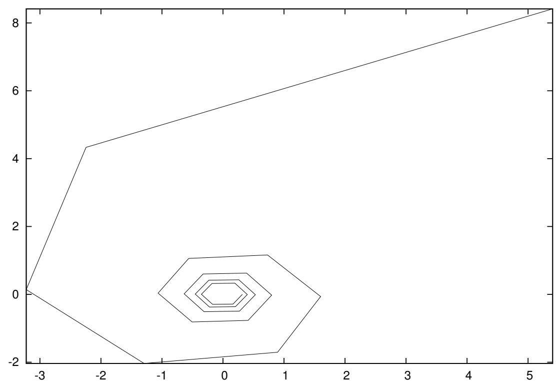
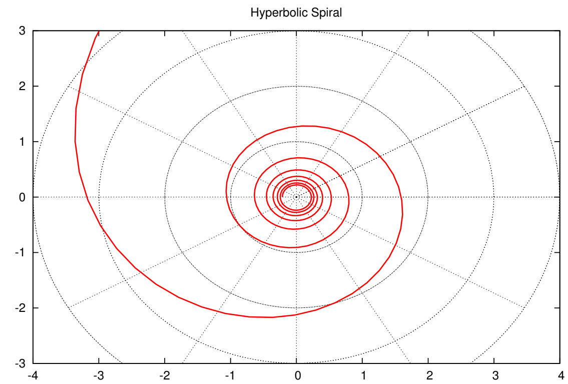



Figure 5: A hyperbolic spiral with default options (left) and more reasonable options (right).

```
        nticks=300,
        title="Hyperbolic Spiral",
        polar(10/theta,theta,1,15*%pi)
    )$
```

The options are discussed in §9.1.1 and §9.1.11. The cryptic format of the `user_preamble` option is due to the fact that it must contain gnuplot commands. Gnuplot is Maxima's plotting engine, and its syntax and conventions are independent of those of Maxima. See the results of the two hyperbolic spiral graphs in figure 5. Note that setting the `xrange`:`yrange` ratio to 4:3 is another way to ensure that the aspect ratio of a plot is reasonably close to 1:1. This method, however, is not as precise as the `proportional_axes = xy` option.

9.1.3 Discrete data

Graphs of discrete data sets are also graphed using the `draw2d()` command. The `points()` construct tells Maxima to plot data points. Suppose you want to plot the data set

$$\{(0,1),(5,5),(10,5),(15,4),(20,6),(25,5)\}.$$

By default, Maxima will produce a scatter plot of the points. If this is what you want, then you are all set. You just need to provide Maxima with the data and call the `draw2d()` command:

```
(%i20) load("draw")$
       xx:[0,5,10,15,20,25]$
       yy:[1,5,5,4,6,5]$
       draw2d(
           color=red,
           point_size=2,
           point_type=filled_circle,
           points(xx,yy)
       );
```

If you prefer a trend line with no points, change the `point_type` to `none` and specify `points_joined=true` as in `draw2d ( point_type = none, points_joined = true, points( xx, yy ) );` The results of these two plots are shown in Figure 6. See §9.1.11 for a more complete explanation of the options. You may wish to get even fancier by plotting both the points and the connecting line segments. This can be achieved by plotting the data twice, once with each set of options as above. But be careful, once an option is set, it applies to all graphs that follow it, so it will be necessary to unset some options sometimes (as in `points_joined = false` in the example). Or perhaps you would like to plot a best-fit line along with the data. These graphs are shown in the next two examples. See Figure 7 for the results.

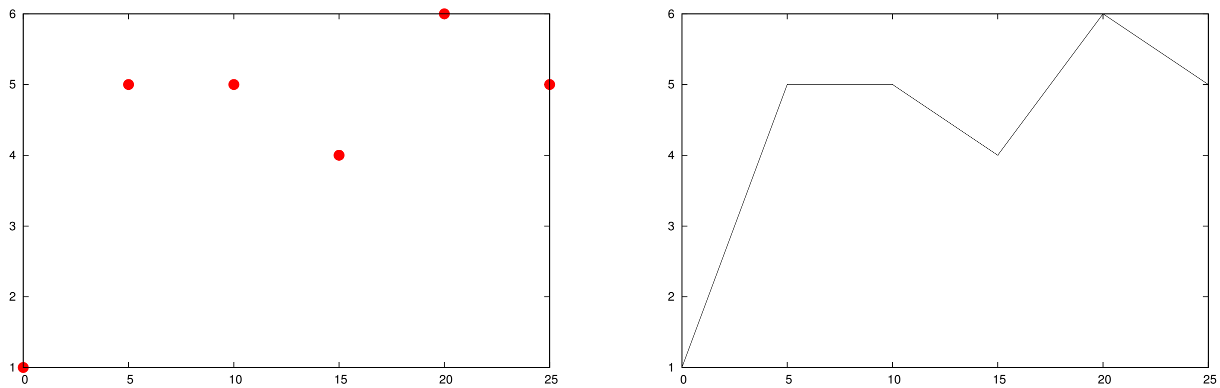


Figure 6: Plotting discrete data.

```
(%i7) load(“draw”)$
      xx:[0,5,10,15,20,25]$
      yy:[1,5,5,4,6,5]$
      xy:makelist([xx[i],yy[i]],i,1,6);
      draw2d(
          point_type=none,
          points_joined=true,
          points(xy),
          point_size=2,
          point_type=filled_circle,
          color=red,
          points_joined=false,
          points(xy)
      )$

(%o7) [[0,1],[5,5],[10,5],[15,4],[20,6],[25,5]]

(%i8) bestfit:0.1257*x+2.7619$
      draw2d(
          key="best fit line",
          explicit(bestfit,x,0,25),
          point_size=2,
          point_type=filled_circle,
          color=red,
          key="data",
          points(xy),
          yrange=[0,10],
          user_preamble="set key bottom"
      );
```

Notice that the `makelist()` command was used to combine the data into a single list of $[x, y]$ ordered pairs. This is an alternative format for inputting the data for a discrete plot.

9.1.4 Implicit plots

Yet another graphing feature of the `draw2d()` command is the ability to graph implicitly defined relations. The syntax is very much like that of `explicit()` but `implicit()` requires that intervals for both variables involved in the relation be specified. Maxima does not assume that the variable x is to be plotted against the horizontal axis and that y is to be plotted againt the vertical. The variable first listed in the implicit call

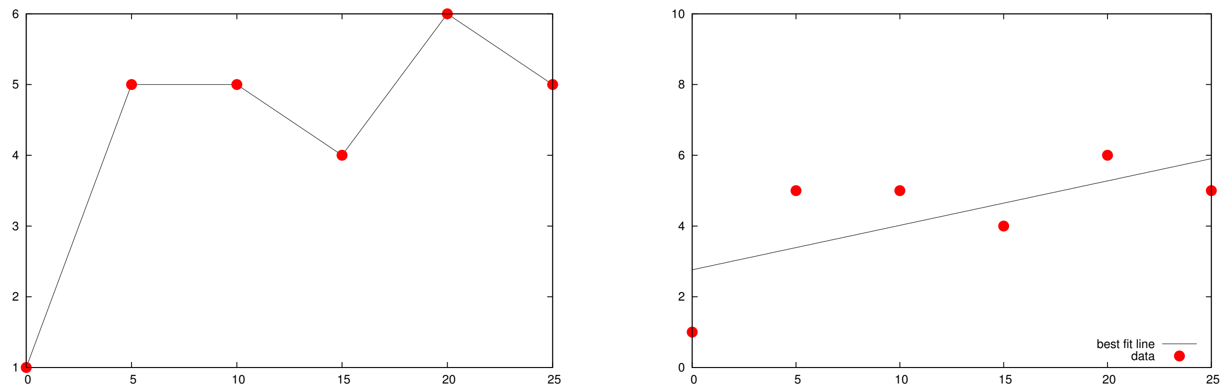


Figure 7: Fancier discrete data plots.

will be graphed against the horizontal axis and the second listed will be graphed against the vertical axis. And neither one has to be x and neither one has to be y. Here are two examples. The first one just produces an implicit plot. The second one finds an equation of the tangent line to an implicit graph and plots both. See figure 8.

```
(%i11) load(“draw”)$
       eqn:(u^2+v^3)*sin(sqrt(4*u^2+v^2))=3$
       draw2d(implicit(eqn,u,-5,5,v,-5,5))$

(%i15) eqn:s^2=t^3-3*t+1$
       s0:1$
       t0:0$
       depends(t,s)$
       subst([diff(t,s)=D,s=s0,t=t0],diff(eqn,s))$
       D0:subst(solve(%,D),D)$
       tanline:t-t0=D0*(s-s0)$
       draw2d(
           color=blue,
           key="s^2=t^3-3*t+1",
           implicit(eqn,s,-4,4,t,-2.5,3.5),
           color=red,
           key="tangent line",
           implicit(tanline,s,-4,4,t,-2.5,3.5),
           xaxis=true, xaxis_type=solid,
           yaxis=true, yaxis_type=solid,
       )$
```

Note that an alternative way to graph the tangent line is to define `tanline` explicitly as in `tanline : D0 * ( s - s0 ) + t0$` and then use an explicit call as in `explicit ( tanline, s, -4, 4 )` instead of the implicit call used above.

9.1.5 Example (obtaining output as a graphics file)

To obtain graphics file output such as EPS, PNG, GIF, or JPG, use the `file_name` and `terminal` options. The value for `file_name` should be the desired file name without extension. The `terminal` type will determine what format the output will take. The following example will create a graph of the sine function in the file `sine.png`. A more involved example can be found in §9.1.10.

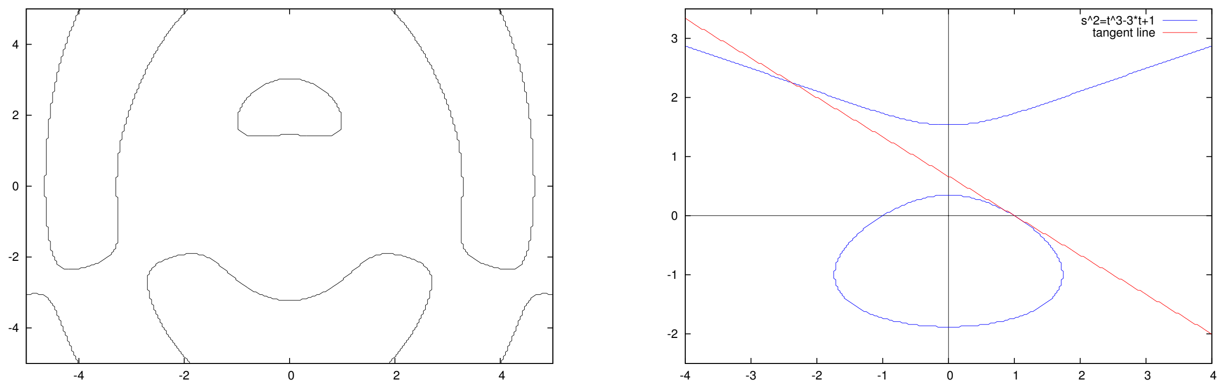


Figure 8: Implicit plot examples.

```
(%i10) load("draw")$
       draw2d(
           explicit(sin(x),x,-5,5),
           file_name="sine",
           terminal='png
       );
```

NOTE: If you are using wxMaxima, the output graphics file may not be created. However, a file named `maxout.gnuplot` will be. In order to get your graphics file (`sine.png` in the example), you will need to locate `maxout.gnuplot` and, from the command line, run the command `gnuplot maxout.gnuplot`.

9.1.6 Example (using a grid and logarithmic axes)

Plotting a grid is as easy as setting `grid` to true. Setting the axes to use a logarithmic scale is similarly simple. The example shows how a polynomial appears logarithmic while an exponential function appears linear when the y-axis is logarithmic. To see how to set the grid for a polar plot, refer to §9.1.2.

```
(%i9) load("draw")$
      draw2d(
          logy=true,
          grid=true,
          explicit(exp(x),x,1,40),
          explicit(x^10,x,1,40)
      );
```

See figure 9.

9.1.7 Example (setting draw() defaults)

If you are creating multiple graphs, each of which share some common set of options, you may find the use of `set_draw_defaults()` helpful. Instead of setting the common options for each graph, you set them once in the `set_draw_defaults()` command. In the example, all graphs will be plotted with thick blue lines, axes showing, no upper or right boundary lines, and with the same x and y ranges.

```
(%i88) load("draw")$
       set_draw_defaults(
           xaxis=true,
           yaxis=true,
           xrange=[-4,4],
           yrange=[-3,3],
```

```
        axis_right=false,
        axis_top=false,
        color=blue,
        line_width=4,
        terminal=eps
    );
    draw2d(explicit(x^2,x,-4,4),file_name="quadratic");
    draw2d(explicit(x^3,x,-4,4),file_name="cubic");
    draw2d(explicit(x^4,x,-4,4),file_name="quartic");
    draw2d(explicit(x^5,x,-4,4),file_name="quintic");
    draw2d(explicit(x^6,x,-4,4),file_name="hexic");
```

Also see §9.1.10 for an example using `set_draw_defaults()` to create a Taylor Polynomial animation.

9.1.8 Example (including labels in a graph)

Including a label in a graph is as easy as placing a `label()` construct in the sequence of arguments of a `draw2d()` command. The `label()` construct takes three arguments: the label and the coordinates where the label should be placed. The alignment and orientation of labels are controlled by the `label_alignment` and `label_orientation` options.

```
(%i1) load(“draw”)$
      draw2d(
          label(["crest",%pi/2,1.05]),
          label(["trough",-%pi/2,-0.93]),
          label_alignment='left,
          label(["baseline",-2,0.05]),
          label_alignment=center,
          label_orientation=vertical,
          label(["amplitude",%pi/2-0.1,0.5]),
          rectangle([%pi/2,0],[%pi/2,1]),
          color=goldenrod,
          line_width=3,
          explicit(sin(x),x,-3,3),
          title="Parts of a sinusoidal wave",
          yrange=[-1,1.2],
          grid=true,
          xaxis=true, xaxis_type=solid,
          yaxis=true, yaxis_type=solid,
      );
```

See figure 9.

9.1.9 Example (graphing the area between two curves)

Use the `filled_func` and `fill_color` options to graph the area between two curves. Set `filled_func` to one of the functions and afterward plot the other function using `explicit`. When `filled_func` is used, the functions themselves are not graphed. Only the region between them is graphed. Therefore, in the example functions f and g are graphed afterward in thick black lines. To graph the area between a function and the x-axis, set `filled_func=0`.

```
(%i1) load(“draw”)$
      f:x^2$
      g:x^3$
      draw2d(
          filled_func=f,
```

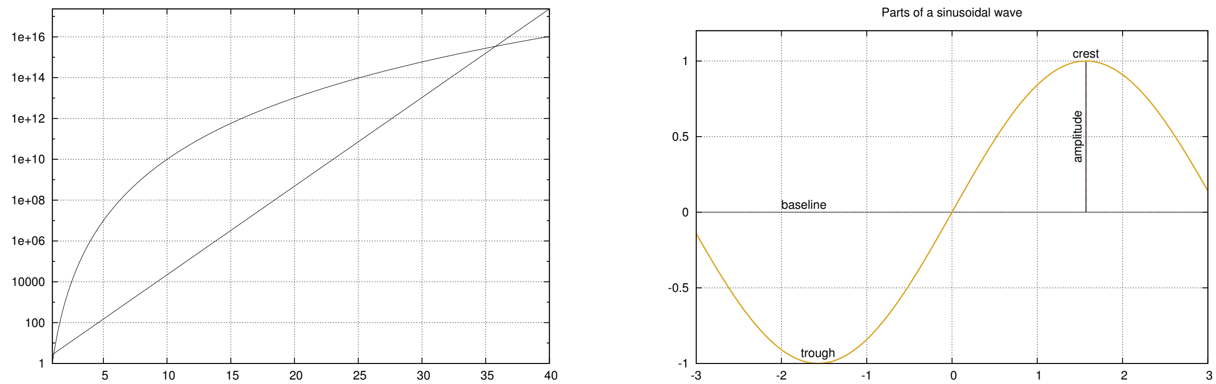


Figure 9: Logarithmic axes on the left and a labeled graph on the right.

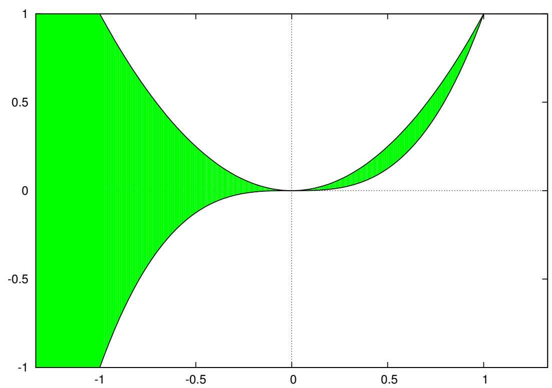
```
            fill_color=green,
            explicit(g,x,-4/3,4/3),
            line_width=2,
            filled_func=false,
            explicit(f,x,-4/3,4/3),
            explicit(g,x,-4/3,4/3),
            yrange=[-1,1],
            xaxis=true,
            yaxis=true
        );
```

See figure 10.

9.1.10 Example (creating a Taylor polynomial animation)

The following set of commands can be used as a template for creating animations of Taylor Polynomials converging to a function. Only the first 4 lines need to be modified to change the function of interest. As it stands, it will show the first 18 (counting the 0^{th}) Taylor polynomials for $\sin(x)$.

```
(%i8) load(“draw”)$
      f(x):=sin(x);
      graph_name:"sin(x)"$
      set_draw_defaults(xaxis=true, yrange=[-4,4])$
      options:[file_name="sine",terminal=animated_gif,delay=100]$
      graph_title:["Zeroth Taylor Polynomial", "First Taylor Polynomial",
                    "Second Taylor Polynomial", "Third Taylor Polynomial",
                   "Fourth Taylor Polynomial", "Fifth Taylor Polynomial",
                   "Sixth Taylor Polynomial", "Seventh Taylor Polynomial",
                   "Eighth Taylor Polynomial", "Ninth Taylor Polynomial",
                   "Tenth Taylor Polynomial", "Eleventh Taylor Polynomial",
                   "Twelfth Taylor Polynomial", "Thirteenth Taylor Polynomial",
                   "Fourteenth Taylor Polynomial", "Fifteenth Taylor Polynomial",
                   "Sixteenth Taylor Polynomial", "Seventeenth Taylor Polynomial"]$
      graph_color:["red", "yellow", "green", "blue", "cyan", "magenta",
                   "turquoise", "pink", "goldenrod", "salmon",
                   "red", "yellow", "green", "blue", "cyan", "magenta",
                   "turquoise", "pink", "goldenrod", "salmon"]$
      scene:makelist(gr2d(
                          key=graph_name,
```



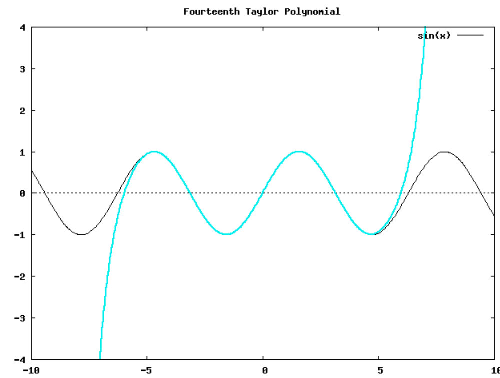


Figure 10: Area between curves on the left and one frame of a Taylor polynomial animation on the right.

```
                        explicit(f(x),x,-10,10),
                        title=graph_title[i],
                        color=graph_color[i],
                        line_width=2,
                        key="",
                        explicit(taylor(f(x),x,0,i-1),x,-10,10)
                    ),i,1,18)$
    apply(draw,append(options,scene));
```

See figure 10.

9.1.11 Options

Some of the most useful options for 2D graphs are listed below. For a complete list of options and complete description of the options below see the Maxima Reference Manual at

http://maxima.sourceforge.net/documentation.html,

section 51 draw. In the following discussion, possible values where appropriate will be listed in *italic* and the default value will be highlighted in ***bold italic***.

Options are either global or local. Global options may be placed anywhere in the sequence of arguments and may be set only once per plot. Global options apply to all functions being plotted by that command. Local options apply only to the functions that follow them in the list of arguments. Local options may be specified any number of times in a single plot. Each successive value overrides the previous value.

9.1.12 Global options

Global options may only be specified once per plot. Their placement in the sequence of arguments is unimportant. These options affect the overall appearance of a plot irrespective of the graphs (functions) being plotted.

axis_bottom, axis_left, axis_right, axis_top Possible values are ***true*** and *false*. Determines whether to draw the bottom, left, right, or top of the bounding box for a 2D graph.

eps_width, eps_height The width and height in centimeters of the Postscript graphic produced by terminals `eps` and `eps_color`. Default values are ***12*** and ***8***, respectively.

file_name The desired name of the graphics file to be produced by terminals `eps`, `eps_color`, `png`, `jpg`, `gif`, and `animated_gif` without extension. For example, `file_name=“circle”` would cause terminal `eps` to produce a file named `circle.eps` and would cause terminal `png` to produce a file named `circle.png`. The default name is ***maxima_out***.

grid Possible values are *true* and ***false***. Determines whether a grid is to be shown on the x-y plane.

logx, logy Possible values are *true* and ***false***. Determines whether the specified axis should be drawn in logarithmic scale or not.

pic_width, pic_height The width and height in pixels of the graphic produced by terminals `png` and `jpg`. Default values are ***640*** and ***480***, respectively.

terminal Possible values are *eps*, *eps_color*, *jpg*, *png*, *gif*, *animated_gif*, and ***screen***. Determines where the drawing of the graph will be done. The default value, `screen`, indicates that the graph should be shown on the screen. Each of the other values indicates that a graphic file (of type equal to the terminal value) should be created instead. No graph will be shown on screen. Instead, a graphic file will be created on disk.

title The main title to be used for a graph. Default value is "" (no title).

user_preamble Will insert gnuplot commands at the beginning of the gnuplot command list. Default value is "" (no preamble). Some features of gnuplot can only be accessed this way. For example, some possible preambles are

- "set size ratio 1" (Sets the height:width ratio for a graph to 1. Useful for making circles look like circles, for example. Has the same effect as the option `proportional_axes=true`)
- "set grid polar" (Makes a polar grid appear on the graph.)
- "set key bottom" (Makes the legend appear at the bottom of the graph instead of the default position, top.)

xaxis, yaxis Possible values are *true* and ***false***. When true the specified axis will be drawn.

xaxis_color, yaxis_color See `color` option for usage. Determines the color to use for the specified axis when it is drawn.

xaxis_type, yaxis_type Possible values are *solid* and ***dots***. Determines the type of line to use for the specified axis when it is drawn.

xaxis_width, yaxis_width Possible values are positive numbers. Default value is ***1***. Determines the width of the line to use for the specified axis when it is drawn.

xlabel, ylabel The title to be used for the specified axis. Default value is "" (no title).

xrange, yrange Possible values are ***auto*** or an interval in the form $[\min, \max]$. Specifies the interval to be shown for the indicated axis. Note that this range can be set independently of the domain interval that must be specified for the independent variable in explicit and parametric plots.

xtics, ytics Affects the drawing of tic marks along the indicated axis. Possible values and their appearance as described in the following table. The default value is ***auto***.

Value	Appearance
`auto`	tic marks are automatically drawn
`none`	no tic marks are drawn
`positive number`	tic marks will be spaced this far apart
`[start,inc,end]`	tic marks will be placed from `start` to `end` in increments of `inc`
$\{r_1, r_2, \ldots, r_n\}$	tic marks will be placed at the values $r_1, r_2, \ldots, r_n$
$\{["l_1", r_1], ["l_2", r_2], \ldots, ["l_3", r_3]\}$	tic marks will be placed at the values $r_1, r_2, \ldots, r_n$ and will be labeled $l_1, l_2, \ldots, l_n$

9.1.13 Local options

Local options may be specified as many times as desired in a given plot. Each time a local option is specified, it affects every graphic object that follows it in the list of arguments.

color Possible values are names of colors or a hexadecimal RGB code in the format #rrggbb. The default color is ***black***. Other possible color names are *white*, *gray*, *red*, *yellow*, *green*, *blue*, *cyan*, *magenta*, *turquoise*, *pink*, *salmon*, and *goldenrod*. Each of these colors except for black and white may be prefixed by either *"light-"* or "*dark-*" as in *light-red*. When specifying a color with a "-" in the name, it must be enclosed in parentheses. For example, `color=blue` is OK, but `color=light-blue` is not. To specify light-blue, use `color=‘‘light-blue’’`. Affects lines, points, borders, polygons, and labels.

fill_color See `color` for possible values. Default value is ***red***. Affects the filling of polygons and filled functions.

filled_func Possible values are *true*, ***false***, and a *function expression*. When true, will cause the region between an `explicit` function and the bottom of the graphing window to be filled in `fill_color`. When supplied with a function expression, will fill the region between the supplied function and an `explicit` function with `fill_color`.

key The name to use for a function in the legend. Default value is "" (no name).

label_alignment Possible values are ***center***, *left*, and *right*. Determines how a label will be justified relative to its specified coordinates.

label_orientation Possible values are ***horizontal*** and *vertical*. Affects the orientation of labels.

line_width Possible values are positive integers. Default value is ***1***. Affects points, rectangle, explicit, implicit, parametric, and polar.

line_type Possible values are ***solid*** and *dots*. Affects points, rectangle, explicit, implicit, parametric, and polar.

nticks Possible values are positive integers. Default value is ***29***. Specifies the initial number of points to use in the adaptive routine for explicit plots. Also determines the number of points to plot in parametric and polar graphs. Note that no adaptive routine is used for parametric and polar graphs, so `nticks` often must be set higher for plots of these types.

point_size Possible values are non-negative integers. Default value is ***1***. Affects all points except those with `point_type` dot.

point_type Possible values are listed in the table below. Default value is ***1*** (***plus***). Determines the type of points to plot. May be specified using either the numeric value or the name.

Numeric value	Name	Appearance (Approximately)
-1	`none`	
0	`dot`	·
1	`plus`	+
2	`multiply`	×
3	`asterisk`	∗
4	`square`	⊡
5	`filled_square`	■
6	`circle`	⊙
7	`filled_circle`	●

8	`up_triangle`	△
9	`filled_up_triangle`	▲
10	`down_triangle`	▽
11	`filled_down_triangle`	▼
12	`diamant`	◇
13	`filled_diamant`	◆

points_joined Possible values are *true* and ***false***. When true, points will be connected with line segments.

9.2 3D graphs

It is recommended that you read §9.1 before continuing. Much of what is discussed there applies here. 3D graphs are created using the `draw3d()` command whose syntax is very similar to that of the `draw2d()` command but with necessary modifications involving the independent variables. For example, instead of graphing an explicit function with a call like `draw2d(explicit(sin(x),x,-5,5))`, you use a call like `draw3d(explicit(sin(x*y),x,-2,2,y,-2,2))`. 3D extensions exist for the constructs explicit, implicit, parametric, and points. Additionally, the common 3D analogs of polar, namely cylindrical and spherical coordinates, are available. Finally, Maxima also offers a facility for creating contour plots and parametric surfaces.

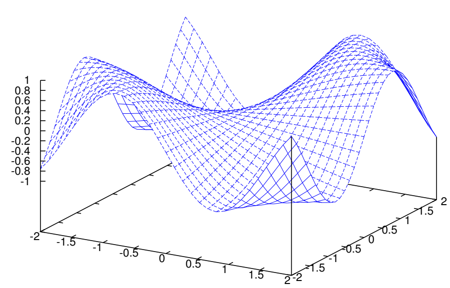
There are many options available for controlling a graph's appearance, some of which will be discussed as they are used in the examples. Be aware that the options discussed in §9.1.11-§9.1.13 also apply to 3D graphs, so they will be used but not re-explained here. Only new options will be explained as they are encountered.

9.2.1 Functions and relations

As with 2D graphs, Maxima has the facility to draw 3D explicit, implicit, and parametric graphs. In fact, the constructs and syntax are nearly identical. The only modification needed for these constructs is the addition of the third variable. Refer to the corresponding section on 2D graphs for more information. In the following examples, the `(%i#)` will be suppressed as there is no chance for confusion between input (the Maxima code) and output (the graphs).

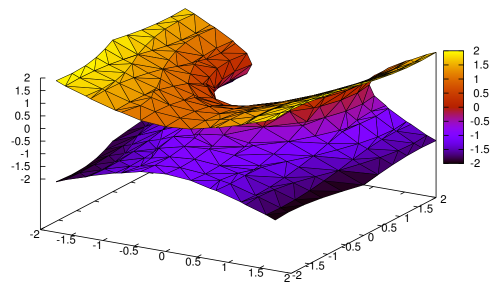
Explicit Graph the function $f(x, y) = sin(xy)$ over the rectangle $[-2, 2] \times [-2, 2]$. The `surface_hide` option tells Maxima not to show hidden surfaces. When `surface_hide` is true, each surface will be treated as if it were opaque. When surface_hide is false, the surface will be shown as a wire frame only.

```
load(draw)$
  draw3d(
      surface_hide=true,
      color=blue,
      explicit(sin(x*y),x,-2,2,y,-2,2),
  );
```



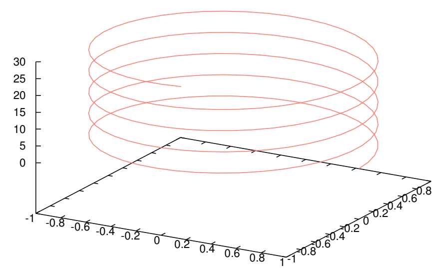
Implicit Graph the solutions of $x^2 - sin(y) = z^2$ in the cube $[-2, 2] \times [-2, 2] \times [-2, 2]$. When enhanced3d is true, Maxima will plot the surface in color and display a colorbox indicating what values the colors represent.

```
load(draw)$
draw3d(
    enhanced3d=true,
    implicit(x^2-sin(y)=z^2,
             x,-2,2,y,-2,2,z,-2,2)
);
```



Parametric Graph the spring defined by the parametric equations $x = \cos(t)$, $y = \sin(t)$, $z = t$, $t \in [0, 30]$.

```
load(draw)$
draw3d(
    nticks=200,
    line_width=2,
    color=salmon,
    parametric(cos(t),sin(t),t,t,0,30)
);
```

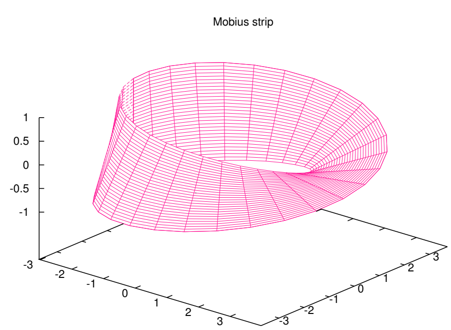


Maxima also has the capability of drawing parametric surfaces plus functions defined in cylindrical and spherical coordinates.

Parametric surface Parametric surfaces are defined by supplying the `parametric_surface` construct with the 3 coordinate functions (of 2 variables) plus the ranges for the two independent variables. The syntax for graphing a parametric surface is

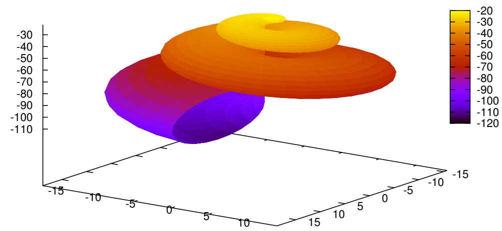
```
parametric_surface(x(u,v),y(u,v),z(u,v),u,umin,umax,v,vmin,vmax)
```

```
load(“draw”)$
draw3d(
    title="Mobius strip",
    color="dark-pink",
    surface_hide=true,
    rot_vertical=54,
    rot_horizontal=40,
    parametric_surface(
        cos(x)*(3+y*cos(x/2)),
        sin(x)*(3+y*cos(x/2)),
        y*sin(x/2),
        x,-%pi,%pi,y,-1,1
    )
);
```



A second example

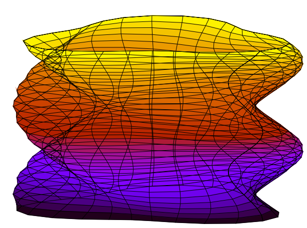
```
load("draw")$
draw3d(
    enhanced3d=true,
    xu_grid=100,
    yv_grid=25,
    parametric_surface(
        0.5*u*cos(u)*(cos(v)+1),
        0.5*u*sin(u)*(cos(v)+1),
        u*sin(v)-((u+3)/8*%pi)^2-20,
        u,0,6*%pi,v,-%pi,%pi
    ),
    rot_horizontal=126,
    rot_vertical=67
);
```



Cylindrical Functions in cylindrical coordinates are defined by supplying the `cylindrical` construct with an explicit expression for radius as a function of the azimuth, θ, and z-coordinate. θ is the angle within the x-y plane measured counterclockwise from the positive x-axis, as in polar coordinates. θ is often in the range from 0 to 2π. The syntax for graphing a cylindrical function is

```
cylindrical(r(z,t),z,zmin,zmax,t,tmin,tmax)
```

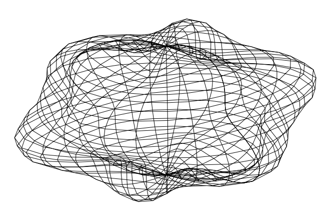
```
draw3d(
    cylindrical(5+cos(3*th+z),
            z,0,12,th,0,2*%pi),
    axis_3d=false,
    xtics=false,
    ytics=false,
    ztics=false,
    rot_horizontal=44,
    rot_vertical=76,
    enhanced3d=true,
    colorbox=false
);
```



Spherical Functions in spherical coordinates are defined by supplying the `spherical` construct with an explicit expression for magnitude as a function of the azimuth, θ, and zenith, ϕ. θ is the angle within the x-y plane measured counterclockwise from the positive x-axis, often in the range from 0 to 2π. ϕ is the angle measured from the positive z-axis, often in the range from 0 to π. The syntax for graphing a spherical function is

```
spherical(M(t,p),t,tmin,tmax,p,pmin,pmax)
```

```
draw3d(
    spherical(5+cos(3*th)*sin(4*ph),
            th,0,2*%pi,ph,0,%pi),
    axis_3d=false,
    xtics=false,
    ytics=false,
    ztics=false,
    rot_horizontal=50,
    rot_vertical=56
);
```



9.2.2 Discrete data

See §9.1.3 for a discussion of how to plot discrete data. The differences for plotting points in 3D are that you must

- use `draw3d()` instead of `draw2d()`
- give 3 coordinates for each point instead of 2

For example, the following code will plot the data set

$$\{(1,5,2),(2,4,4),(6,2,3),(3,7,1),(4,4,6),(2,3,9),(5,5,4)\}.$$

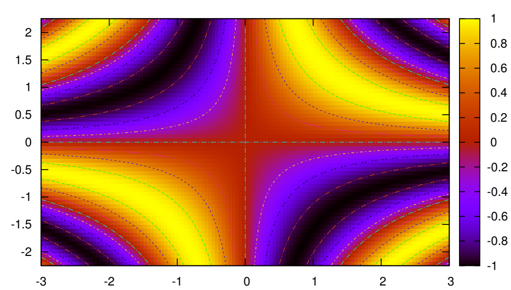
Of course point and line options such as `point_size`, `line_width`, and `point_type` all apply to 3D plots exactly as they do to 2D plots.

```
(%i31) load("draw")$
       xx:[1,2,6,3,4,2,5]$
       yy:[5,4,2,7,4,3,5]$
       zz:[2,4,3,1,6,9,4]$
       draw3d(points(xx,yy,zz));
```

9.2.3 Contour plots

A contour plot is just an explicit plot with appropriate options set. So to create a contour plot, do exactly as you would if you were planning to plot a 3D explicit function; then add the information about how you want the contours to look using the `contour` option and, optionally, the `contour_levels` option.

```
load("draw")$
draw3d(
    user_preamble="set pm3d at s;unset key",
    xu_grid=120,
    yv_grid=60,
    explicit(sin(x*y),x,-3,3,y,-9/4,9/4),
    contour=map,
    contour_levels={-.9,-0.6,-0.3,0,.3,.6,.9}
 );
```



The `user_preamble` "`set pm3d at s;unset key`" does two things. It tells Maxima to add color and a color scale to the contour plot, and to turn off the contour line key that normally accompanies a contour plot. The same code will work perfectly well without the `user_preamble`, but no coloring will be done.

9.2.4 Options with 2D counterparts

Each of these options works just like its x and y counterparts. So for information on `logz`, see `logx` (§9.1.12) and for information on `zaxis`, see `xaxis`, and so on.

```
logz, zaxis, zaxis_color, zaxis_type, zaxis_width, zlabel, zrange, ztics
```

9.2.5 Options with no 2D counterparts

All of the following options except for `xu_grid` and `yv_grid` are global options.

axis_3d Possible values are ***true*** and *false*. Determines whether to draw a bounding box for a 3D graph.

colorbox Possible values are ***true*** and *false*. Determines whether to draw a color scale for colored 3D graphs. If `enhanced3d` is false, this setting has no effect.

contour Possible values are ***none***, *base*, *surface*, *both*, and *map*. The effects of these options are as follows.

- `none`: no contour lines are plotted.
- `base`: contour lines are added to the x-y plane.
- `surface`: contour lines are added on the surface.
- `both`: contour lines are added to the x-y plane and on the surface.
- `map`: contour lines are drawn in 2D. No surface is shown.

contour_levels Possible values are positive integers, a list of three numbers, or a set of numbers. Default value is ***5***. The effects of the three types of values are as follows.

- positive integer: Sets the number of contour lines to be drawn. They will be drawn at equal intervals within the range.
- list of three numbers, `[low, step, high]`: Sets contour lines to be plotted from `low` to `high` in increments of `step`.
- set of numbers, $\{v_1, v_2, \ldots\}$: Contour lines will be plotted at the values, v_1, v_2, and so on, specified in the set.

enhanced3d Possible values are *true* and ***false***. Determines whether to draw 3D surfaces in color.

rot_horizontal Possible values are numbers from 0 to 360. The default value is ***30***. Sets the angle of rotation about the z-axis for 3D scenes.

rot_vertical Possible values are numbers from 0 to 180. The default value is ***60***. Sets the angle of rotation about the x-axis for 3D scenes.

surface_hide Possible values are *true* and ***false***. Determines whether to draw hidden parts of 3D surfaces.

xu_grid Possible values are positive integers. Default value is 30. Sets the number of values of the first coordinate to use in building the grid of sample points. The less even a surface is, the higher this number will have to be in order to capture the details of the surface.

yv_grid Possible values are positive integers. Default value is 30. Sets the number of values of the second coordinate to use in building the grid of sample points. The less even a surface is, the higher this number will have to be in order to capture the details of the surface.

10 Programming

Iterative methods such as Newton's Method, Euler's Method, and Simpson's Rule are often part of the common Calculus sequence. Sometimes they are simply demonstrated and sometimes students are asked to implement the algorithms in some type of programming language or another. All of them can easily be programmed and executed using Maxima. The main ingredient not yet covered in this manual is the loop. In Maxima, there is only one looping structure: the `do()` command. In its simplest form, it is used with no prefix. In this case, its arguments are to be interpreted as a list of commands to be executed repeatedly (looped) ad infinitum. Of course, to be practical, such a loop must have an exit procedure. This is supplied by the `return()` command. When a `return()` is reached, the `do()` loop is exited and the argument of the `return()` command becomes the loop's return value. So, an "infinite" `do()` loop will typically have the form

```
do(
    command1(),
    command2(),
        ⋮
    commandN(),
    if conditionMet then return(value)
)
```

`do()` loops may also be prefixed with conditions on how many times to excute the loop and for what values of the looping variable. If you are comfortable with "for" loops from other programming languages, this will look very familiar. The possible forms for such loops are

- `for` *variable*: *startvalue* `thru` *endvalue* `step` *increment* `do(`*commands*`)`
- `for` *variable*: *startvalue* `while` *condition* `step` *increment* `do(`*commands*`)`
- `for` *variable*: *startvalue* `unless` *condition* `step` *increment* `do(`*commands*`)`

The only difference between the three forms is the exit condition. The first will exit after the loop has executed for the *endvalue*. The second will only exit when the `while` condition fails to be met. The third will exit when the `unless` condition is met. So, these three `do()` loops are equivalent:

- `for i:1 thru 10 step 1 do(`*commands*`)`
- `for i:1 while i<11 step 1 do(`*commands*`)`
- `for i:1 unless i>10 step 1 do(`*commands*`)`

In fact, when *increment* is 1, you can omit the `step` as in

`for i:1 while i<11 do(`*commands*`)`

10.1 Example (Newton's Method)

Let's look at Newton's Method with the intent of creating a reusable (functional) implementation. As a quick review, if you have a function $f(x)$ and an initial approximation x_0, then Newton's Method is to compute iteratively $x_1, x_2, x_3, \ldots, x_n$ according to the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

The number of iterations must be monitored in some way since Newton's Method is not guaranteed to converge in general. So, the function that we build must at a minimum apply the above formula iteratively and count the number of iterations. It is customary to include the ability to stop iterating when $|f(x_i)|$ is less than some tolerance, ϵ. So, a good implementation will have this ability as well. Here is one such implementation.

```
newton(f,x0,tol,maxits):=([t,df,c],
    df:diff(f(t),t),
    c:0,
    do(
        x0:x0-f(x0)/ev(df,t=x0),
        if abs(f(x0))<tol then return (x0),
        c:c+1,
        if c=maxits then return ("maximum iterations reached")
    )
);
```

The inputs to the function are `f` (the function whose zeroes are desired), `x0` (the initial approximation), `tol` (the tolerance), and `maxits` (the maximum number of iterations to attempt). The variables `t`, `df`, and `c` are declared to be local to this function by enclosing them in square brackets immediately following the open parenthesis for the function. Each line of the function except the last is terminated with a comma instead of the usual semicolon or dollar sign. This is how to include more than one command in a function. Similarly the `do()` command consists of multiple lines, each except the last terminated by a comma. This form of the `do()` command will simply loop until a `return()` command is encountered. Notice there are two conditions under which the `do()` command will exit: if `abs(f(x0))<tol` or if `c=maxits`. In other words, if $|f(x_i)| < \epsilon$ or the number of iterations has reached its maximum. In case of the first condition, the value of the current iteration is returned. In case of the second condition, a message stating that the maximum number of iterations was reached is returned instead, indicating failure to achieve the desired accuracy. Notice the straightforward use of the `if ... then ...` construct. To call the function, you need four arguments. For example, finding a zero of $f(x) = x - \cos(x)$ to within $5(10)^{-10}$ accuracy using an initial approximation of 300 could be done as follows.

```
(%i115) f(x):=x-cos(x);
        newton(f,300.0,5e-10,50);

(%o115) f(x):=x-cos(x)
(%o116) .7390851332151606
```

Of course the implementation could be modified to include a `print()` statement within the `do()` loop to report each iteration if such information were desirable. And to make calling the function simpler, a tolerance and maximum number of iterations could be hard-coded into the function, thus reducing the number of arguments.

10.2 Example (Euler's Method)

Euler's Method is a first-order numerical method for solving differential equations based on the simple assumption that

$$y(t+h) \approx y(t) + h \cdot y'(t, y(t)).$$

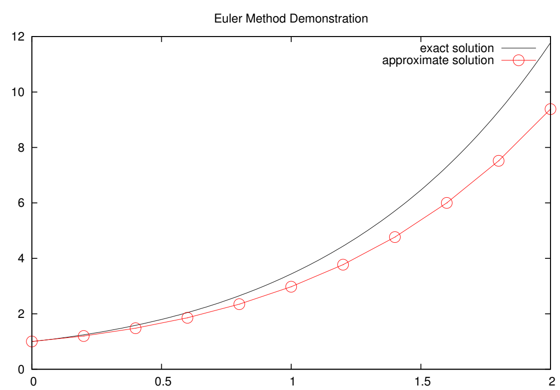
Let's construct an implementation designed to execute Euler's Method and display the results graphically. We will assume a differential equation of the form

$$y' = f(t, y).$$

The method will proceed from an initial condition $y(t_0) = y_0$, calculating $y_{i+1} = y_i + h \cdot f(t_i, y_i)$ and $t_{i+1} = t_i + h$ for $i = 0, 1, 2, \ldots, n-1$. This time, we will forego writing a multiline function definition in favor of a code whose first several lines should be modified but last several should not. The first several lines will contain the information about the differential equation and its initial conditions. The last several lines will contain the implementation of the algorithm and the code to display the results. Here is one way to do so.

```
load(draw)$
/* Set up DE and parameters */
/* Modify these values to demo */
/* solution of other DEs */
f(t,y):=t+y$
tt:[0.0]$
yy:[1.0]$
h:0.2$
n:10$
yactual(t):=-1-t+2*exp(t)$
yr:[0,12]$ /* yrange fro the graph */
/* Execute the method */
```

```
/* Do not modify any lines */
/* below this one */
for j:1 thru n do(
    yy:append(yy,[yy[j]+h*f(tt[j],yy[j])]),
    tt:append(tt,[tt[j]+h])
)$
/* Plot results */
draw2d(
    key="exact solution",
    explicit(yactual(x),x,t[0],t[0]+h*n),
    key="approximate solution",
    points_joined=true,
    color=red,
    point_type=circle,
    point_size=2,
    points(tt,yy),
    title="Euler Method Demonstration",
    yrange=yr
);
```



Notice that the program is 31 lines long, but the heart of Euler's Method only accounts for 4 of them! The rest of the lines are a matter of convenience and readability. All text delimited by `/*` and `*/` is treated as a comment. These parts of the code do nothing but instruct the reader. The `do()` command in this form executes once for each integer from 1 through `n`.

10.3 Example (Simpson's Rule)

Our implementation of Simpson's Rule will be a hybrid of the implementations of Newton's Method and Euler's Method. We will produce a no-frills multiline Simpson's Rule function and wrap it with both utilitarian and illustrative code. Starting with Simpson's Rule, we will use the `do()` command with the `step` option.

```
simpsons(f,x1,x2,n):=([j,h,total],
    total:0.0,
    h:ev((x2-x1)/(2*n),numer),
    for j:0 thru 2*(n-1) step 2 do(
        total:total + f(x1+j*h) + 4.0*f(x1+(j+1)*h) + f(x1+(j+2)*h)
    ),
    h*total/3.0
)$
```

There are more efficient ways to program this computation, but this one favors simplicity. It may make a nice exercise to rewrite this function to do the computation in a more efficient manner. In any case, note that the function requires 4 arguments: the function to integrate (`f`), the limits of integration (`x1` to `x2`), and the number of intervals to use in Simpson's Rule (`n`). Since this function has no bells or whistles, it can be used in both a utilitarian fashion as in

```
(%i69) f(x):=sin(x)*exp(x)+1$
       simpsons(f,0,%pi,10);
(%o70) 15.21177471542183
```

and in a more frilly fashion as in

```
(%i80) f(x):=sin(x)*exp(x)+1$
       x1:0$
       x2:%pi$
       n:10$
       exact:'integrate(f(x),x,x1,x2)$
       print(exact)$
       print("is approximately ",simpsons(f,x1,x2,n))$
       print("and is exactly ",ev(exact,nouns))$
       print("which is about ",ev(%,numer))$

 %pi
/
[       x
I    (%e  sin(x) + 1) dx
]
/
 0

is approximately  15.21177471542183

                                %pi
                2 %pi + %e    + 1
and is exactly  -----------------
                        2

which is about  15.21193896997943
```

Index

ˆ, 6, 8
π, 6, 9
%pi, 6, 9
e, 9
%e, 9
i, 9
%i, 9
:, 7
:=, 7
[], 10

addition, 5
antiderivatives, 18
Arccosecant, 7
Arccosine, 7
Arccotangent, 7
Arcsecant, 7
Arcsine, 7
Arctangent, 7
arithmetic, 5
assignment, 7

CAS, *see* computer algebra system
command
    acos(), 7
    acsc(), 7
    atan(), 7
    bfloat(), 13, 24
    cos(), 6
    csc(), 7
    display(), 7
    do(), 46–49
    draw2d(), 29–31, 33–35
    draw3d(), 42–45
    ev(·,*equations*), 7, 27
    ev(·,logexpand), 9
    ev(·,nouns), 26
    ev(·,numer), 4, 6, 7
    ev(·,pred), 9, 10
    ev(·,simpsum), 23
    ev() summary, 13
    evalV(), 26
    exp(), 5, 8
    exp(), 19, 22, 25
    expand(), 5
    fpprec, 13, 24
    fpprintprec, 13
    fullratsimp(), 5
    if...then..., 47
    ind, 14
    inf, 14, 23
    integrate(), 19
    limit(), 14
    log(), 8
    logb(), 8
    makelist(), 34
    maplist(), 14
    minf, 14
    minus, 14
    niceindices(), 24–26
    Norm(), 26
    partfrac(), 21
    plus, 14
    powerseries(), 24–26
    print(), 5
    radcan(), 8
    return(), 46–48
    scalefactors(), 26
    set draw defaults(), 36, 38
    solve(), 10, 27
    sqrt(), 6
    sum(), 23
    sum, 23
    tan(), 7
    taylor(), 24–26
    und, 14
computer algebra system, 3
constants, 9
cosecant, 6
cosine, 6
cotangent, 6
cross product, *see* product, cross
cube root, *see* root, other than square
curl, 27
cylindrical coordinates, 44

decimal representation, *see* floating point
degrees to radians, 6
difference
    arithmetic, *see* subtraction
divergence, 27
division, 5
dot product, *see* product, dot

e, 9
evaluation, 13
exponents, 5, 8
    ˆ, 6, 8

factorial, 5
floating point, 4, 13
for loop, 47
function definition, 7

gradient, 27
graphing, 29
    2D, 29
    3D, 42
    contour plots, 45
    discrete data, 33, 45
    implicit relations, 34
    parametric equations, 31
    polar functions, 32
    scatter plot, 33
    `cylindrical`, 44
    `explicit`, 30, 31, 34, 36–39, 42, 45
    `implicit`, 34, 35, 42, 43
    `parametric surface`, 43, 44
    `parametric`, 30–32, 43
    `points`, 33, 34, 45, 49
    `polar`, 32, 33
    `spherical`, 44

i, 9
infinity, 14, 23
integration, 18
    multiple, 22

Laplacian, 27
limits, 13–15
logarithms, 8

multiplication, 5

noun form, 19, 26

package
    `draw`, 18, 29
    `vect`, 26
partial fractions, 21
pi, 6, 9
plotting, *see* graphing
power, *see* exponents
product
    arithmetic, *see* multiplication
    cross, 27
    dot, 27

quotient
    arithmetic, *see* division

radians to degrees, 7
Ramanujan, 24
root
    other than square, 6
    square, 6

secant
    trigonometric, 6
series, 23
    scalar, 23
    Taylor, 24
sine, 6
solving equations, 9–11
spherical coordinates, 44
square root, *see* root, square
subtraction, 5
sum, 23
    arithmetic, *see* addition
    infinite, 23

tangent
    trigonometric, 6
trigonometric functions, 6

vector
    access single component, 26
    magnitude, 26

wxMaxima, 3