CHAPTER 2

SETS AND FUNCTIONS

OBJECTIVES

1. **Define sets using different types of commands.**
2. **To define functions and properties related to it.**

1 SETS

Maxima provides functions (commands) to define sets, operations such as intersection and union for finite sets that are defined by explicit enumeration. Maxima treats lists and sets as distinct objects. This feature makes it possible to work with sets that have members which are either lists or sets.

1.1 Basic Commands

A set is a collection of well defined objects. To construct a set with members a_1', ..., a_n, either we write **set** (a_1, ..., a_n) or **{a_1, ..., a_n}**; For an empty set, write **set ()** or **{}**. In input, **set (...)** and **{ ... }** are equivalent.

```
(%i1)  kill (all)$set ();{};
(%o1)  {}
(%o2)  {}
(%i3)  set (A, B, C, D);{a, b, c, d};
(%o3)  {A, B, C, D}
(%o4)  {a, b, c, d}
```

Any two elements x and y are redundant (*i.e.*, considered the same for set construction) if and only if **is** (x = y) yields true. Note that **is** (**equal** (x, y)) can yield true while **is** (x = y) yields false; in that case, the elements x and y are considered distinct.

```
(%i5)  kill (all)$
       x:p/c + q/c;y: (p + q)/c;
       is (x = y);is (equal (x, y));
```

```
(%o1)  q/c + p/c

(%o2)  (q + p)/c

(%o3)  false

(%o4)  true

(%i5)  x - y;ratsimp (%);

(%o5)  (q + p)/c + q/c + p/c

(%o6)  0

(%i7)  A:{p, q, r};B:{p, q, {r}};is (A = B);

(%o7)  {p, q, r}

(%o8)  {p, q, {r}}

(%o9)  false
```

since is$((x-1)(x+1)=x^2-1)$ evaluates to false, $(x-1)(x+1)$ and x^2-1 are distinct set members; thus,

```
(%i10) kill (all)$
       A:{(x - 1)* (x + 1)}; B:{x^2 - 1};is (A = B);

(%o1)  {(x -) x + 1 })

(%o2)  {x^2 - 1}

(%o3)  false

(%i4)  {(x - 1)* (x + 1), x^2 - 1};
       map (rat, %);

(%o4)  {(x - 1) (x + 1), x^2 - 1}

(%o5)  /R/ {x^2 - 1}
```

To remove redundancies from other sets, one may use other simplification functions. Here is an example that uses function **trigsimp:**

```
(%i6)  {1, cos (x)^2 + sin (x)^2};
       map (trigsimp, %);

(%o6)  {1, sin (x)^2 + cos (x)^2 }

(%o7)  {1}
```

The predicate function **emptyp**(A) return true if and only if A is the empty set or the empty list.


```
(%i31) emptyp ({{}});emptyp ({});
(%o31) false
(%o32) true
```

The function **subset** (A, f) can be used to create subsets based on certain conditions. The **subset** (A, f) returns the subset of the set A that satisfies the predicate f. The subset returns a set which comprises the elements of A for which f returns anything other than false. The subset does not apply to the return value of f. The subset complains, if A is not a literal set.


```
(%i33) subset ({a, b + s, c + t}, atom);
       subset ({1, 2, 3, 4, 5, 6}, evenp);
(%o33) {a}
(%o34) {2, 4, 6}
```

In order to check whether A is a subset of B, use the predicate function **subsetp**(A, B). It returns true if and only if the set A is a subset of B. The **subsetp** complains, if either A or B is not a literal set.

```
(%i35) subsetp ({a, b, l}, {a, b, c, e, d, s});
(%o35) false
```

2 FUNCTIONS

2.1 Definition

A function is a mapping or rule which assigns to each input exactly one output. Maxima has a large number of built-in functions such as a logarithm function **(log)**, factorial function **(!)** etc. One has to provide an input or argument, and Maxima produces the output:

```
(%i36) log (10.0);
(%o36) 2.302585092994046
```

In Maxima, one can also define an own function e.g. here f is a quadratic function while g is a square root function. The function is defined by using : = (combination of : and =).

```
(%i37) f(x): = x^2 + 2*x + 3;f(2);
f(%pi) 'f(%pi);float (%);
       g (x): = sqrt (x);'g (4) = g (4);
(%o37) f(x): = x^2 + 2 x + 3
(%o38) 11
(%o39) f(π) = π^2 + 2 π + 3
```

```
(%o40)  f(3.141592653589793) = 19.15278970826894
(%o41)  g (x): =√x
(%o42)  g (4) = 2
```

In defining function g, existing built-in function **sqrt** of Maxima has been used.

There is another command called **define,** which can be used to define any function. It defines a function named f with arguments x_1, ..., x_n and function body expr. The define always evaluates its second argument (unless explicitly quoted). The function so defined may be an ordinary Maxima function (with arguments enclosed in parentheses) or an array function (with arguments enclosed in square brackets). The general form is:

define (f(x_1, ..., x_n), expr).

One can explore more syntaxes of function **define.**

```
(%i43)  kill (all)$define (f(x, y, z), x^2 + y^2 + z^2);f(x, y, z);f(1, 2, 3);
(%o1)   f(x, y, z): = z² + y² + x²
(%o2)   z² + y + x²
(%o3)   14
```

We can also define functions with a block statement. Functions that require more than one statements to be defined can use the **block** statement. The **block** statement is used if a **return** statement is to be included in the definitions of a function.

The general form of a **block-statement** function is as follows:

block ([< variables with assignments >], < expressions >)

```
(%i4)   kill (all)$
        h (x): = block (if  (0 < = x and x < 2) then return (x + 1)
        elseif  (2 < = x and x < 4) then return ((x + 1)^2)
        else return (0));
(%o1)   h (x): = block (if 0 < = x. x < 2 then return (x + 1)
        elseif 2 < = x. x < 4 then return ((x + 1)2) else return (0))
(%i2)   h (x);
(%o2)   if 0 < = x. x < 2 then return (x + 1) elseif 2 < = x. x < 4
        then return ((x + 1)²) else return (0)
```

To make it easier to understand the definition of this function, it can be re-written as:

```
(%i3)   f(x): = block (if  (0 < = x and x < 2) then
            return (x)
        elseif(2s < = x and x < 4) then
            return ((x + 1)^2)
```

```
        else
            return (0));
(%o3)   f()x : = block (if 0 < = x. x < 2 then return (x + 1)
        elseif 2 < = x. x < 4
        then return ((x + 1)^2) else return (0 ))
```

2.2 More About Functions

After defining functions, one may be interested in knowing how many functions have been defined in any particular session. The **functions** gives the list of ordinary Maxima functions in the current session. An ordinary function is a function constructed by **define** or : = and called with parentheses ().

```
(%i4) functions;
(%o4) [h (x), f(x)]
```

The function **fundef (f)** returns the definition of the function f. The argument may be the name of a macro (defined with :: =), an ordinary function (defined with : = or define), an array function (defined with : = or define, but enclosing arguments in square brackets [ ]), a subscripted function, (defined with : = or define, but enclosing some arguments in square brackets and others in parentheses ( )) one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

In contrast, function **dispfun (f)** creates an intermediate expression label and assigns the definition to the label.

```
(%i5)   fundef(f);
(%o5)   f(x): = block (if 0 < = x^x < 2 then return (x + 1) elseif
        2 < = x^x < 4 then return ((x + 1)^2) else return (0))
```

If you are interested in displaying the definition of the user-defined functions f_1, ..., f_n then use the following commands:

dispfun (f_1, ..., f_n)

dispfun (all)

The command **dispfun (all)** displays all user-defined functions as given by the functions, arrays, and macros lists, omitting subscripted functions defined with constant subscripts.

```
(%i6)   kill (all)$
        f(x, y): = x^y;
        g (x):: = x^3;
```

```
(%o1) f(x, y): = xy
(%o2) g(x):: = x^3

(%i3) dispfun (all);
(%t3) f(x, y): = x^y
(%t4) g(x):: = x^3
(%o4) [%t3, %t4]
```

It is very important to remove function from the memory of Maxima. Because one may forget about the function and later in the session try to use the same name for some other function. The function **remfunction**(f) or from **Menu** selecting **Maxima > Delete Function** ... any function can be deleted.

```
(%i5) remfunction (f);
(%o5) [f]
```

The **Show Functions** and **Delete Functions** are also available through the **Menu, Maxima >** as shown in figure.

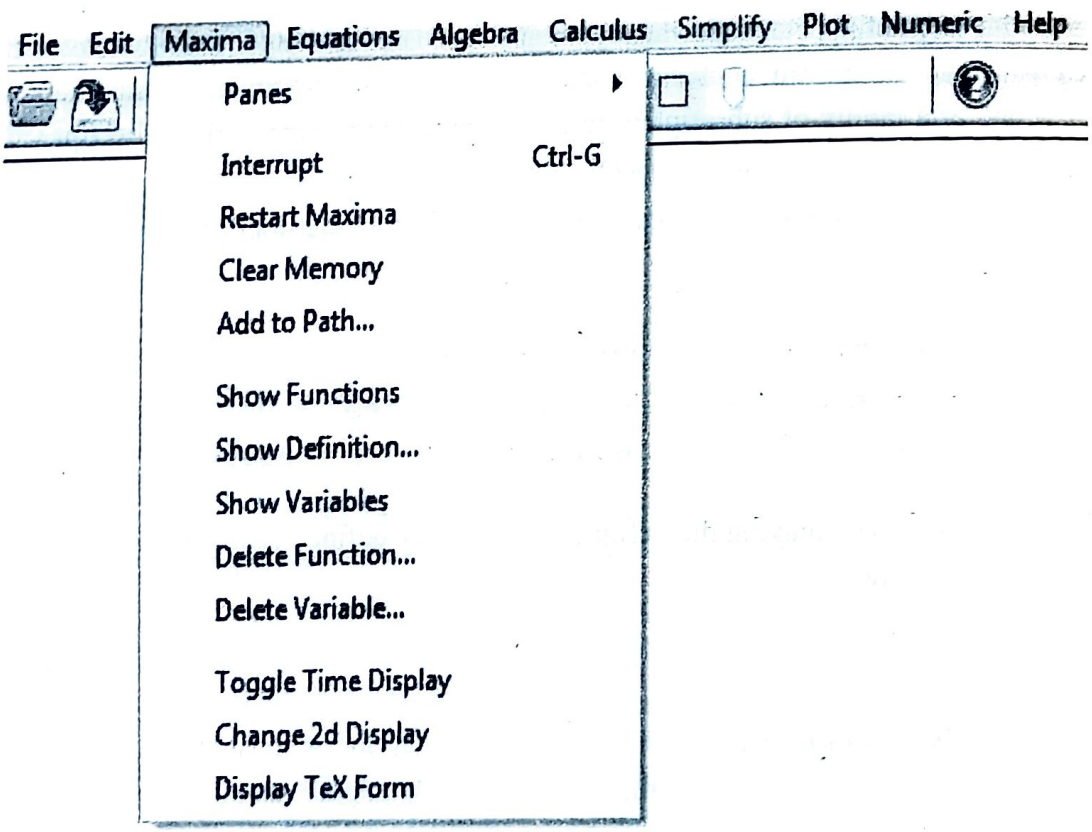


Figure 1: Various Commands related with function

Another method to delete a function is to **kill** that function.The alternate forms are **kill (all), kill (f, g)** or **kill (allbut (f, g))** etc.

```
(%i6)  w(x):=x^6;w(x);
       kill(w);
       w(x);
(%o6)  w(x):=x^6
(%o7)  x^6
(%o8)  done
(%o9)  w(x)
```

The **lambda** ([x_1, ..., x_m], expr_1, ..., expr_n) is used to define anonymous function. The function may have arguments x_1, ..., x_m which appear within the function body as a list. The return value of the function is expr_n.

```
(%i10) f:lambda([x], sin(x));f(w);
(%o10) lambda([x], sin(x))
(%o11) sin(w)
```

A **lambda** expression may appear in contexts in which a function evaluation is expected.

```
(%i12) f:lambda([x], log(x))(a);
(%o12) log(a)
```

The function **apply** (F, [x_1, ..., x_n]) constructs and evaluates an expression F(arg_1, ..., arg_n):

```
(%i13) apply(lambda([x],log(x)),[a]);
(%o13) log(a)
```

The function **map** (f, expr_1, ..., expr_n) returns an expression whose leading operator is the same as that of the expressions expr_1, ..., expr_n but whose subparts are the results of applying f to the corresponding subparts of the expressions where f is either the name of a function of n arguments or is a lambda form of n arguments.

```
(%i14) map(lambda([x], sin(x)), [a, b, c, d, e]);
(%o14) [sin(a),sin(b),sin(d),sin(c),sin(e)]

(%i15) map(sin,[a,b,c]);
(%o15) [sin(b),sin(a), sin(c)]
```

The function **depends** (f_1, x_1, ..., f_n, x_n) declares functional dependencies among variables for the purpose of computing derivatives. In absence of declared dependence, **diff**(f, x) yields zero. If **depends**(f, x) is declared, **diff**(f, x) yields a symbolic derivative.

```
(%i16) depends(u, t);
(%o16) [u(t)]
(%i17) kill(all)$depends([f, g], x);
(%o1)  [f(x),g(x)]
(%i2)  kill(all)$depends([L, S], [u, v, w]);
(%o1)  [L(u, v, w),S(u, v, w)]
```

2.3 Composite Functions

If f and g are functions, the composite function f∘g is defined as f(g (x)).

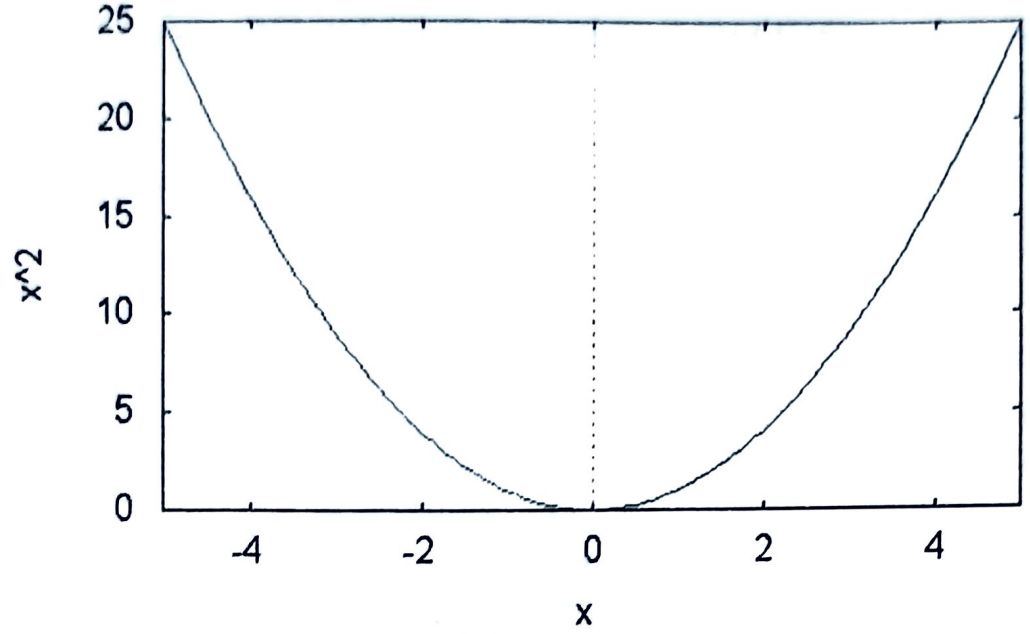
The domain f(g (x)) consists of the numbers x in the domain of g for which g (x) lies in the domain of f.

```
(%i2)  kill (all)$
       f(x): = sqrt(x);g(x):= sin(x);
(%o1)  f(x): = √x
(%o2)  g(x): = sin(x)
(%i3)  f(g(x));g (f(x));f(f(x));g (g (x));
(%o3)  √sin(x)
(%o4)  sin √x
(%o5)  x^1/4
(%o6)  sin(sin (x))
```

2.4 Even and Odd Functions

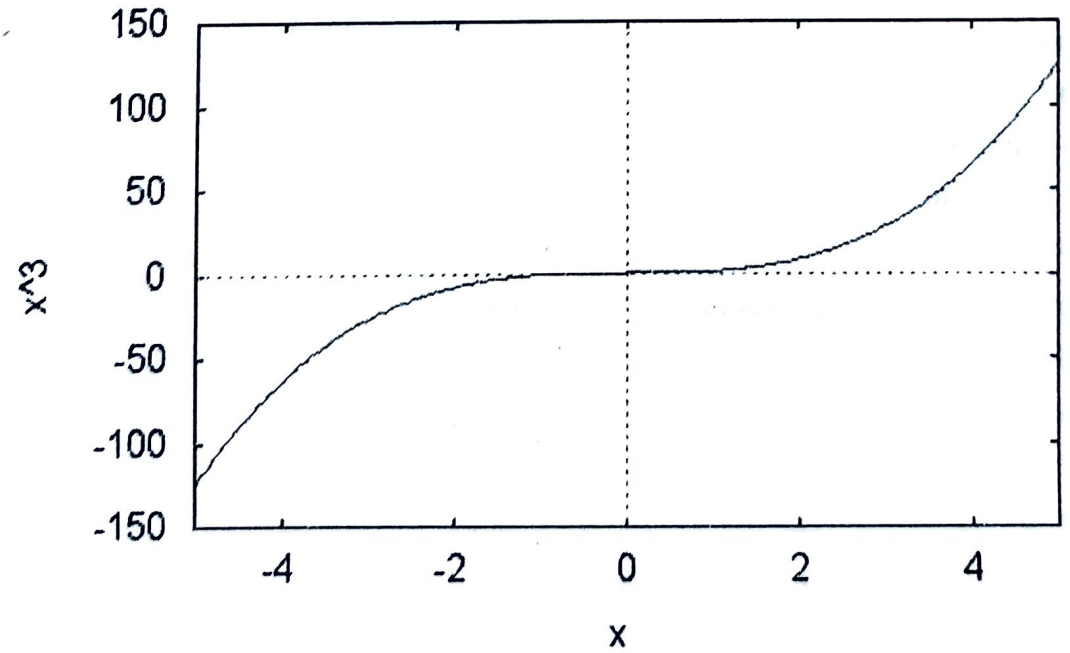
A function y = f(x) is even if f(–x) = f(x) for every x in the domain of f. The graph of even function y = f(x) is symmetric about the y-axis.

```
(%i7)  kill(all);
       f(x): = x^2;is(f(x)=f(-x));
(%o0)  f(x): = x^2
(%o1)  true
(%i2)  wxplot2d ([x^2], [x, - 5, 5])$
```



A function y = f(x) is odd, if f(x) = – f(– x) for every x in the domain of f. The graph of odd function is symmetric about the origin.

```
(%i4)   kill(all)$
        f(x):=x^3;
        is (f(x) = - f(- x));
(%o0)   f(x): = x^3
(%o1)   true
(%i3)   wxplot2d([x^3], [x, - 5, 5])$
```



OBJECTIVES

1. **Plotting explicit and implicit functions.**
2. **Understanding various kinds of advance options in plot2d.**
3. **Plotting parametric and polar curves.**
4. **Understanding contour plotting.**
5. **To know the concept of cylindrical and spherical co-ordinate system.**
6. **Plotting level curves and surfaces.**
7. **Creating animations.**
8. **Tracing conics.**

1 PLOTTING

1.1 Plotting a Function

We begin with an example of a quadratic function:

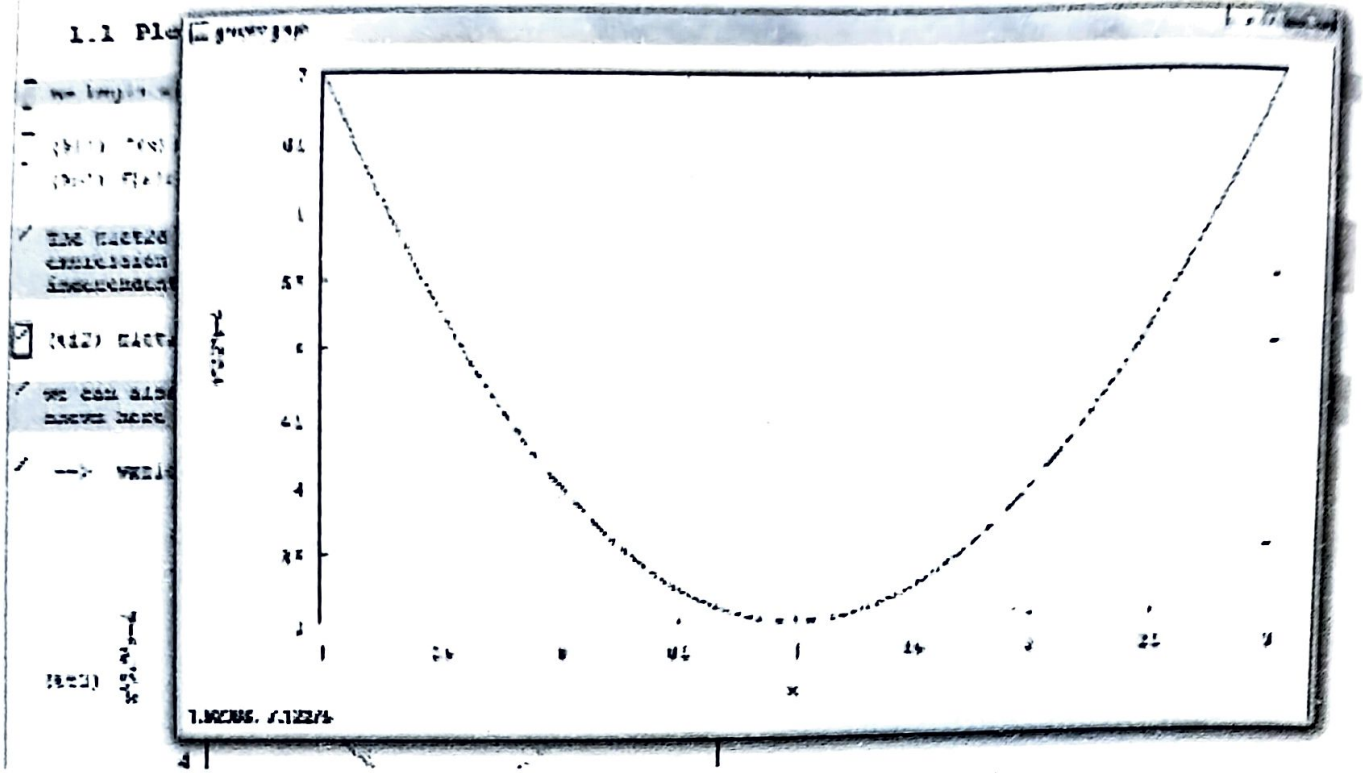
```
(%i1) f(x): = x^2 - 2*x + 4;

(%o1) f(x): = x^2 - 2x + 4
```

The **plot2d** function, in its simplest form, requires as input an expression or function name, an range of values of the independent variable.

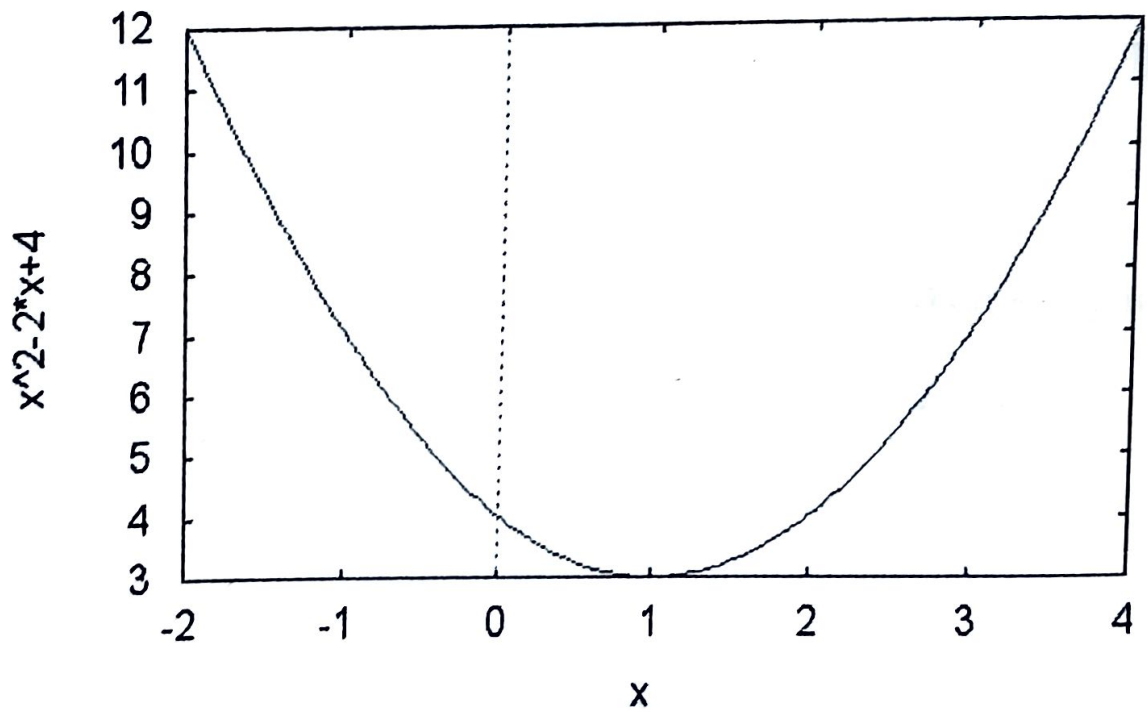
```
(%i2) plot2d(f(x), [x, -1, 3]);

(%o4)
```



we can also use **wxplot2d** function to produce inline graph as shown here :

```
(%i3) wxplot2d(f(x), [x, -2, 4]);
```

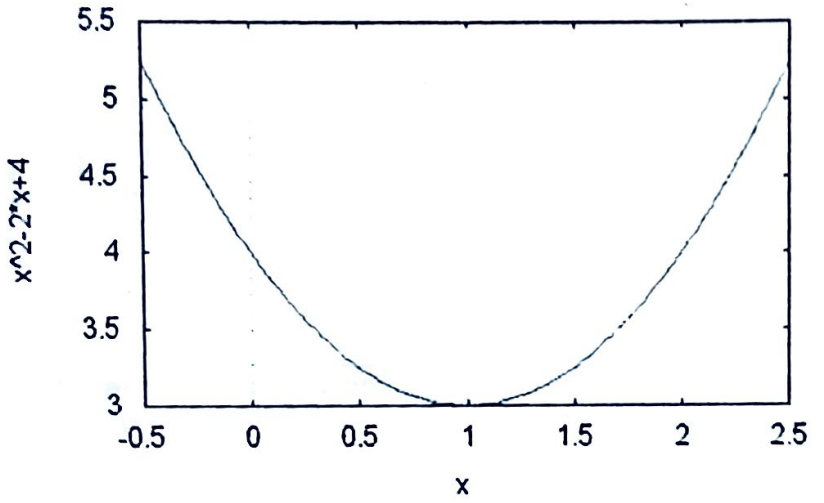


The **plot2d** and **wxplot2d** function have two arguments, separated by a comma. The first argument is the function to be graphed, and the second (in this case [x, – 4, 4]) is called an iterator. It describes the span of values that the variable x can assume; that is, it specifies the domain of the plot. The square brackets are mandatory for describing this domain. The first item x names the

variable, and the next two items give the span of values that this variable will assume (.. – 4 through 4). The values in this domain are displayed along the horizontal axis, while the values that the function assumes are displayed along the vertical axis. The plot, by default depicts the variable on X-axis and function as the range.

You can enlarge or zoom in on a particular portion of a plot simply by editing the domain specified in the iterator, then re-entering the cell.

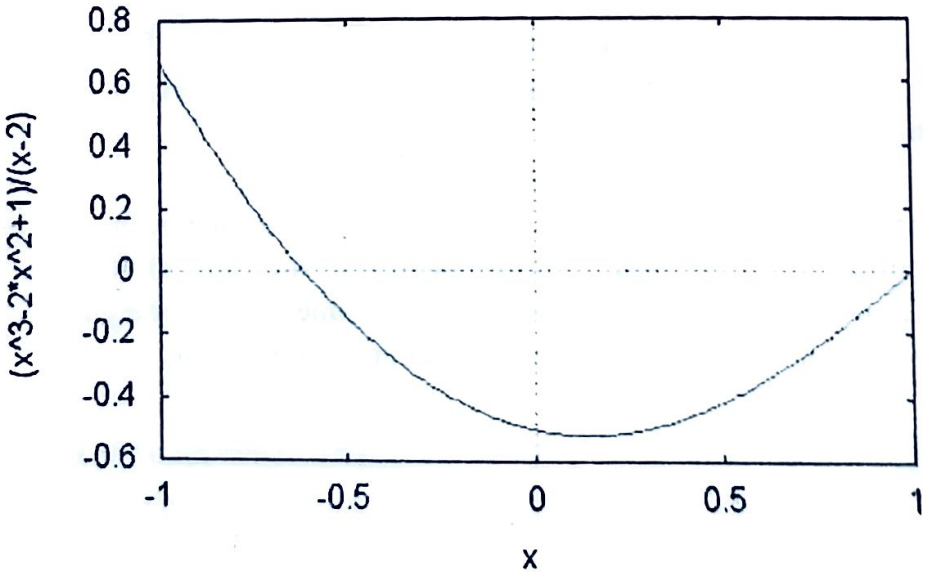
```
(%i4)  wxplot2d(f(x), [x, - 0.5, 2.5]);
```



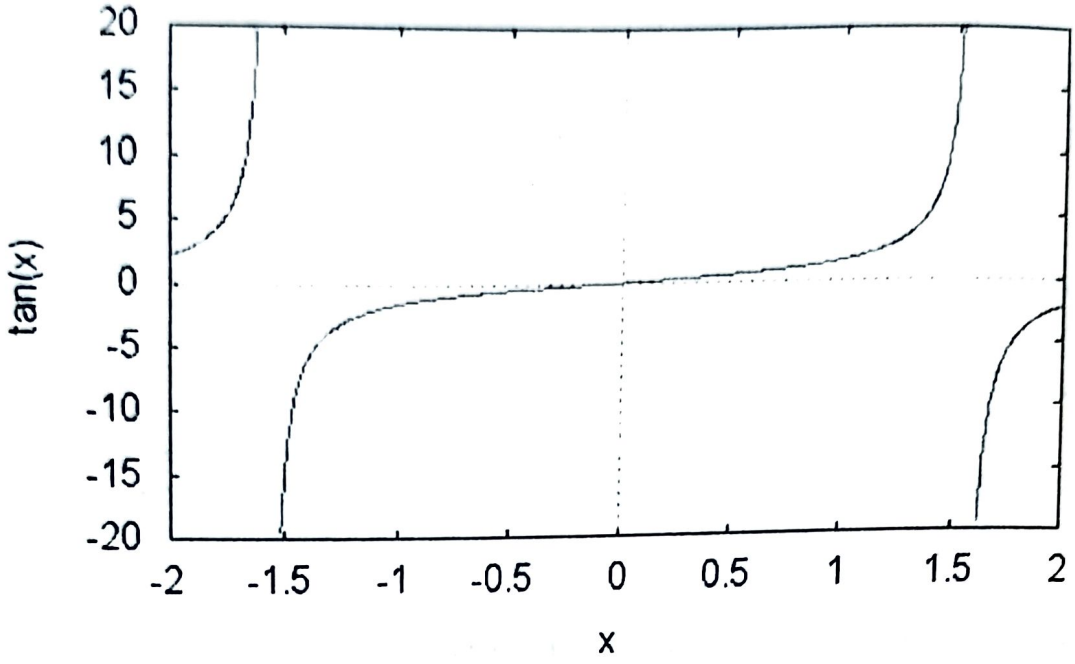
Let us plot a function k(x) which is defined as: $k(x) \quad \frac{x^3 - 2x^2 + 1}{x - 2}$.

```
(%i7)  kill(all)$
       k(x): = (x^3 - 2*x^2 + 1)/(x - 2);
       wxplot2d(k(x), [x, - 1, 1]);
```

(%o1) $k(x) := \frac{x^3 - 2\,x^2 + 1}{x - 2}$



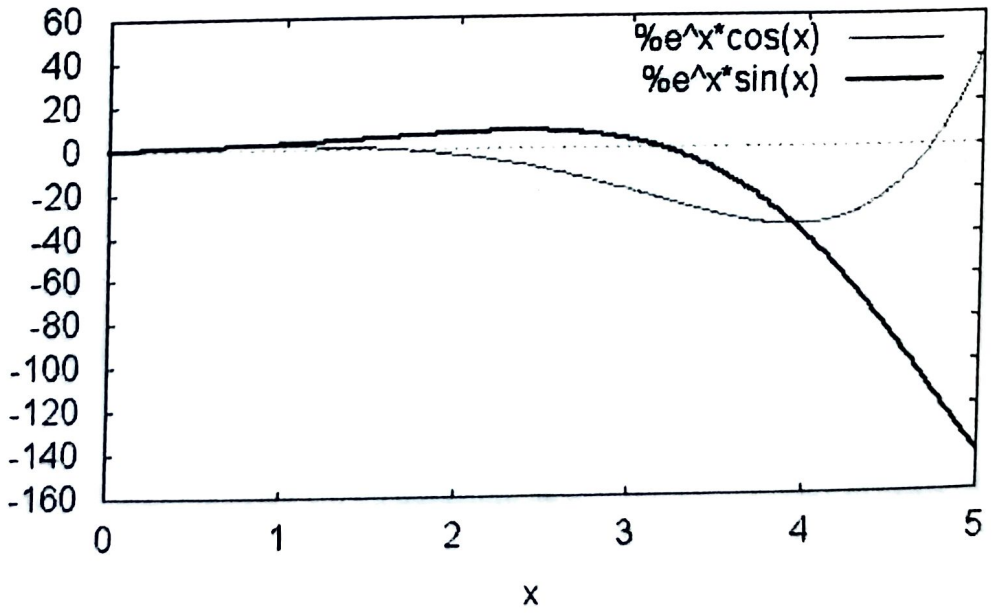
```
(%i6) wxplot2d(tan(x), [x, - 2, 2], [y, - 20, 20]);
```



1.2 Plotting More than one Function

Sometimes it becomes necessary to plot more than one function on a same graph to compare their behaviour. We can specify the list of functions or expressions instead of a single function or expression which allows the plotting of more than one curve through the use of **plot2d** (or **wxplot2d**).

```
(%i7) l(x) : = %e^x*cos(x) $
      m(x) : = %e^(x)*sin(x) $
      wxplot2d([l(x), m(x)], [x, 0, 5], [style, [lines, 1],
      [lines, 2]]);
```

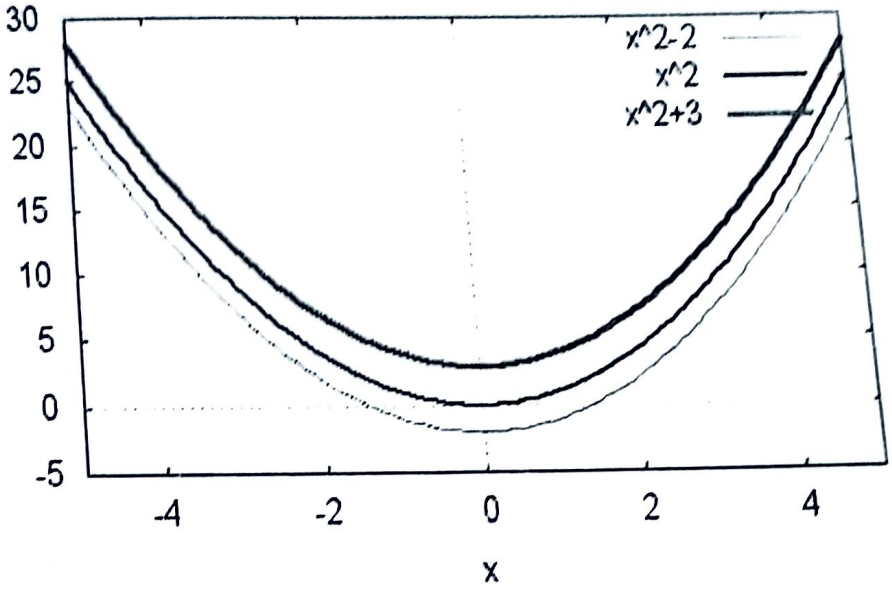


Here, an advanced option style has been used. In following pages user will learn more about it.

1.3 Shifting Graphs

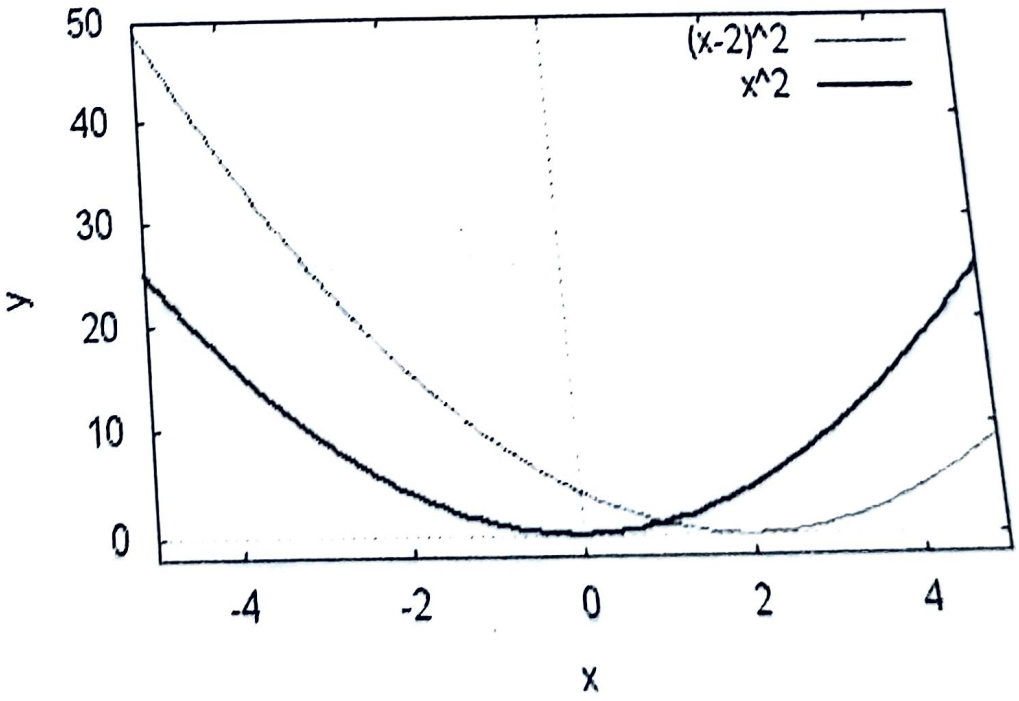
The graph of a function y = f(x) can be shifted straight up by adding a positive constant to the right-hand side of the formula y = f(x).

```
(%i10) wxplot2d([x^2 - 2, x^2, x^2 + 3], [x, - 5, 5], [style,
       [lines, 1], [lines, 2], [lines, 3]]);
```



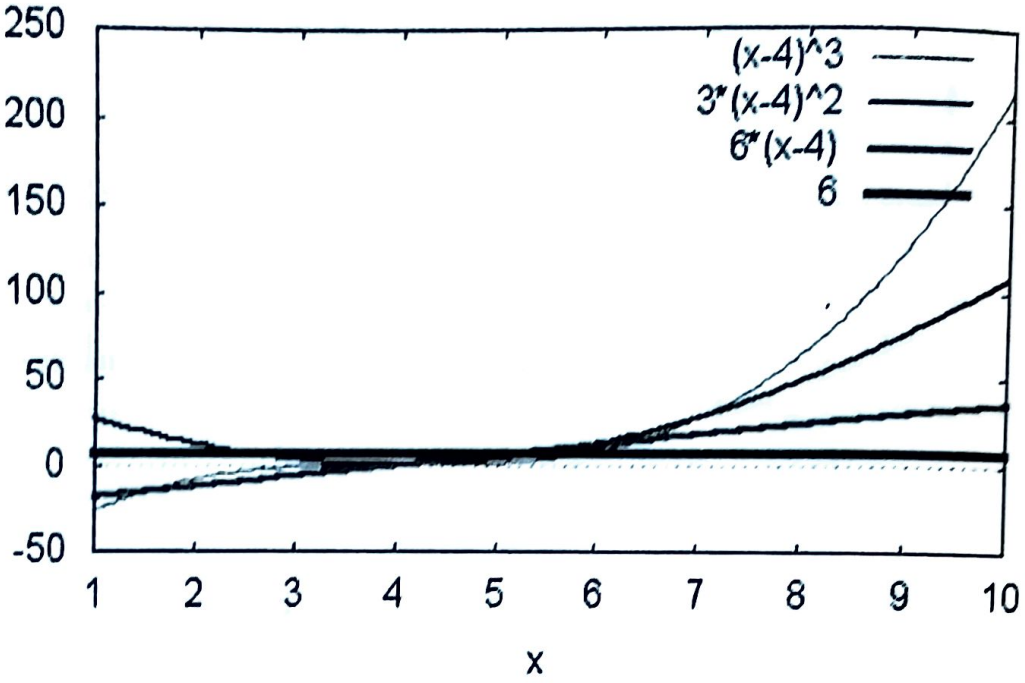
The graph of a function y = f(x) can be shifted horizontally left or right by adding or subtractir positive constant to the x in the formula y = f(x) i.e y = f(x – h) will shift h units right:

```
(%i11) wxplot2d([(x - 2)^2, x^2], [x, - 5, 5],
       [y, - 2, 50], [style, [lines, 1], [lines, 2]]);
```



Plot the function $f(x) = (x^2 - 4)^3$ and their derivatives f', f'', f'''.

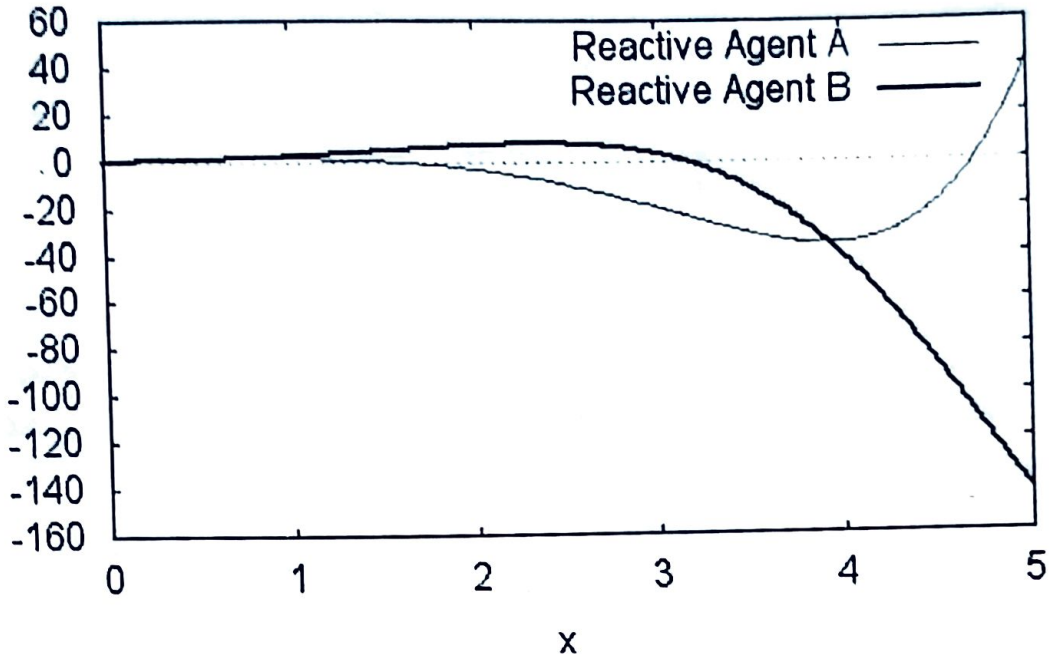
```
(%i12) f(x) := (x - 4)^3;
       g(x) := diff(f(x), x);
       h(x) := diff(f(x), x, 2);
       i(x) := diff(f(x), x, 3);
       wxplot2d([f(x), g(x), h(x), i(x)], [x, 1, 10],
       style, [lines, 1], [lines, 2], [lines, 3], [lines, 4]);
(%o14) f(x) := (x - 4)^3
(%o13) g(x) := diff(f(x), x)
(%o14) h(x) := diff(f(x), x, 2)
(%o15) i(x) := diff(f(x), x, 3)
```



1.4 Advanced Options in Plot

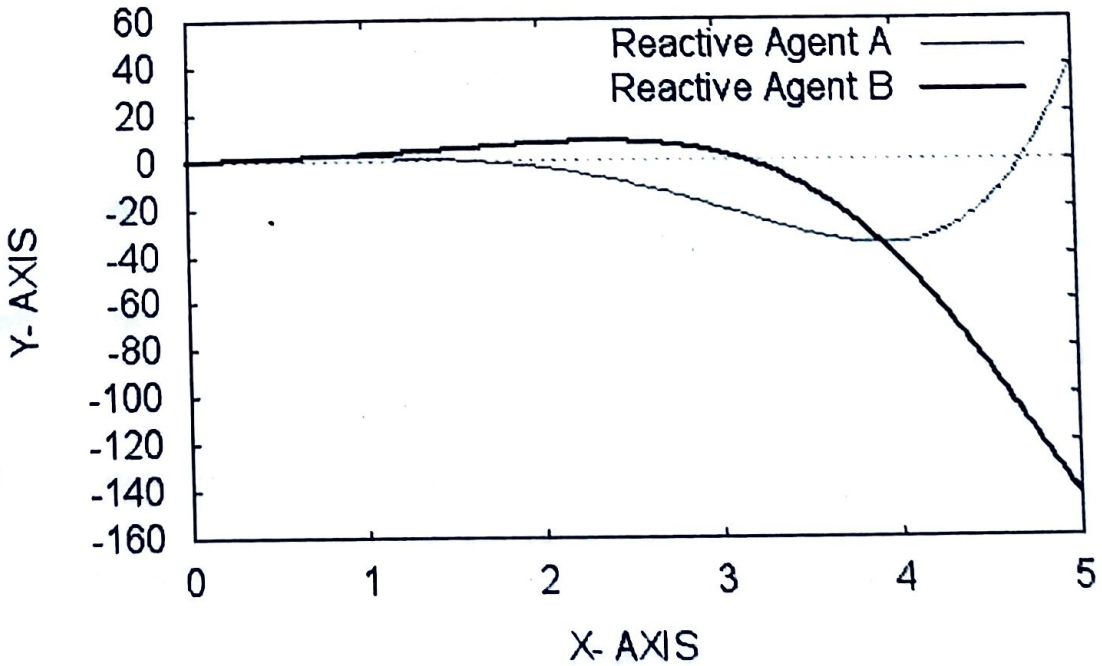
Maxima provides various options like legends, labels, styles, etc. The plot above shows the function expressions as the legends for the two curves. The user may specify the legends to be included by using the legend option as shown here:

```
(%i17) l(x) := %e^x*cos(x) $
       m(x) := %e^(x)*sin(x) $
       wxplot2d([l(x), m(x)], [x, 0, 5], [legend, "Reactive
       Agent A","Reactive Agent B"], [style, [lines, 1],
       [lines, 2]]);
```



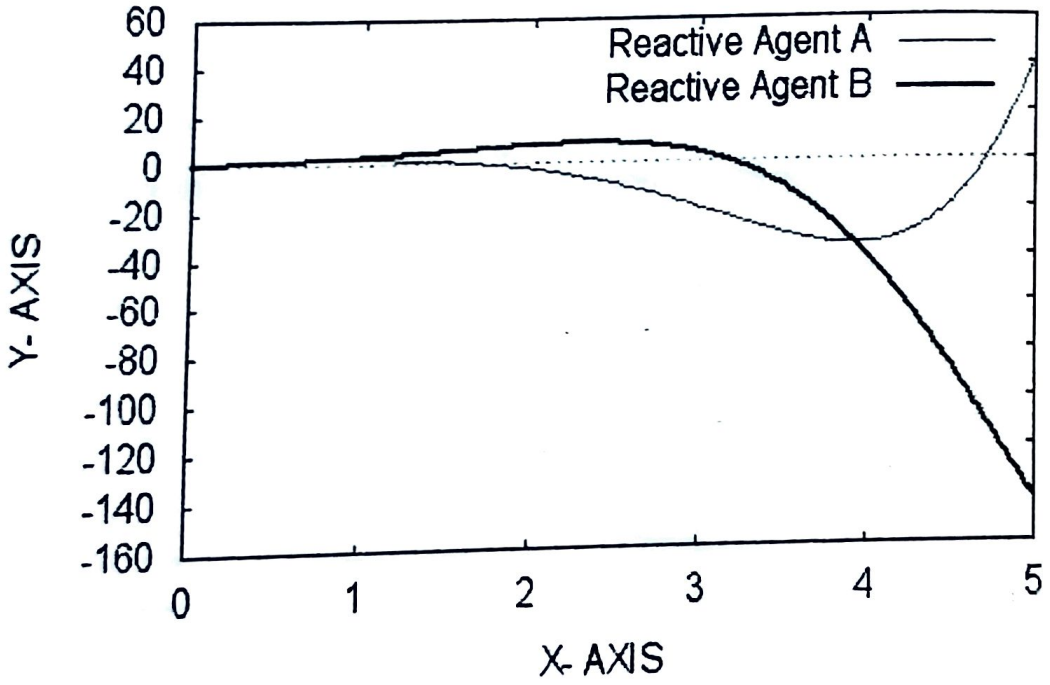
The **xlabel** and **ylabel** can also be specified as labels in the graph to make it more interactive:

```
(%i20) l(x) := %e^x*cos(x) $
       m(x) := %e^(x)*sin(x) $
       wxplot2d([l(x), m(x)], [x, 0, 5], [legend, "Reactive Agent A",
       "Reactive Agent B"], [xlabel, "X-AXIS"], [ylabel, "Y-AXIS"],
       [style, [lines, 1], [lines, 2]]);
```



By default, the expressions are plotted as a set of line segments joining adjacent points within a set of points, which is either given in the discrete form, or calculated automatically from the expression given. It uses an algorithm that automatically adapts the steps among points using as an initial estimate of the total number of points the value set with the **nticks** option. It is to be noted here that more is the number of **nticks** smoother the graph curve is.

```
(%i23) l(x) := %e^x*cos(x) $
       m(x) := %e^(x)*sin(x) $
       wxplot2d([l(x), m(x)], [x, 0, 5], [legend, "Reactive Agent A",
       "Reactive Agent B"], [xlabel, "X-AXIS"], [ylabel, "Y-AXIS"],
       [nticks, 10], [style, [lines, 1], [lines, 2]]);
```



In plots, we can also define styles. The styles that will be used for the various functions or sets of data in a 2d plot. Each **style** can be either

1. **lines** for line segments,
2. **points** for isolated points,
3. **linespoints** for segments and points, or dots for small isolated dots.

Each of the styles can be enclosed inside a list with some additional parameters. A brief discription is given here:

The **lines** accept one or two parameters: the width of the line and second integer identifies a color. The default color codes are:

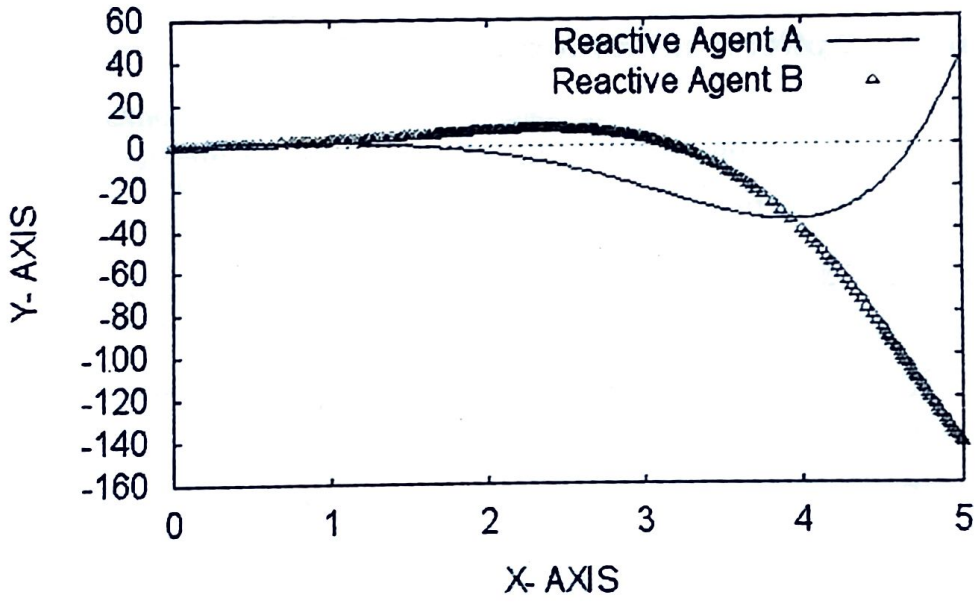
1: blue, 2: red, 3: magenta, 4: orange, 5: brown, 6: lime and 7: aqua.

The **points** accept one, two or three parameters; the first parameter is the radius of the points. The second parameter is an integer that selects the color, using the same code used for lines. The default types of objects are:

1: filled circles, 2: open circles, 3: plus signs, 4: x, 5: *, 6: filled squares, 7: open squares, 8: filled triangles, 9: open triangles, 10: filled inverted triangles, 11: open inverted triangles, 12: filled lozenges and 13: open lozenges.

The **linesdots** accepts up to four parameters: line width, points radius, color and type of object to replace the points.

```
(%i26) l(x) : = %e^x*cos(x) $
       m(x) : = %e^(x)*sin(x) $
       wxplot2d([l(x), m(x)], [x, 0, 5], [legend, "Reactive Agent A",
       "Reactive Agent B"], [xlabel, "X-AXIS"], [ylabel,
       "Y-AXIS"], [nticks, 10], [style, [lines, 1, 2],
       [points, 1, 3, 9]]);
```



1.5 Plotting Discrete Data

In various statistical applications, it becomes important to plot discrete data and analyze it. If the data to be plotted consists of discrete data points, say,

```
(%i29) x : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] $
       y : [0, 1.8, 3.5, 10.5, 12.0, 15.5, 17.3, 18.2, 19.7] $
```

The discrete option is available in a plot along with the list of datapoints x and y, to produce a plot, e.g.,

CHAPTER 14

PROGRAMMING WITH MAXIMA

OBJECTIVES

1. **Understand, Maxima Verbs and Nouns.**
2. **Maxima's database commands.**
3. **Data types and Structures.**
4. **Introduction to various input-output and system commands.**
5. **To know relational and logical operators.**
6. **Understanding different control statements.**
7. **Some important commands including date-time commands.**

1 INTRODUCTION

Maxima is written in Lisp but their symbols are distinguished by naming conventions. The Lisp codes can be executed within Maxima session. The Maxima is case-sensitive i.e it distinguishes between lower and uppercase letters in identifiers.

```
(%i1) kill(all)$is(AB = aB);is(Int = int);
(%o1) false
(%o2) false
```

If you need to accomplish certain task which requires a set of commands, you need to put them in a sequence and even sometimes you must have certain specific statements. Like any other CAS (Computer Alebra System) or programming language, the Maxima provides a fairly good number of commands to accomplish any task. These commands can also be used at command prompt of the Maxima or converted to a block of statements called programs. Being highly structured programming language, you need to know certain programming skills before you begin writing programs. You can save your programs as Maxima files (.wxm) for further reference.

2 VERBS AND NOUNS

Like any other language, Maxima also distinguishes between nouns and verbs. A verb is an operator which can be executed while a noun is an operator which appears as a symbol in an expression, without being executed. By default, function names are verbs.

'expr prevents evaluation of expression expr (called quote operator) while "expr causes evaluation of expression expr (called quote-quote operator). (quote and quote-quote operators are available on a standard keyboard near **Enter** key)

The single quote operator ' prevents evaluation of an expression. This can be useful if one needs to "mask" a global variable. When used on a function, it returns the noun form of the function call. The quote-quote operator " (two single quote marks) causes evaluation of an expression expr when evaluation is otherwise suppressed, such as in function definitions. The value of expr is then substituted for expr in the input expression. Applied to the operator of an expression, the quote-quote operator changes the operator from a noun to a verb (if it is not already a verb).

```
(%i3)  kill(all)$ x:3$
       'x^3 = x^3;
       f(x, y): = x^3 + y^3;
       f(2, 3);'f(2, 3);"f(2, 3);
(%o2)  x^2 = 27
(%o3)  f(x, y): = x^3 + y^3
(%o4)  35
(%o5)  f(2, 3)
(%o6)  35
(%i7)  kill(all)$f(x): = x^2 + "y^2;f('x);
(%o1)  f(x): = x^2 + y^2
(%o2)  y^2 + x^2
```

3 MAXIMA'S DATABASE

3.1 Functions and Variables for Properties

The **declare**(a_1, p_1, a_2, p_2, ...) assigns the atom or list of atoms a_i the property or list of properties p_i. When a_i and/or p_i are lists, each of the atoms gets all of the properties. It may be noted that **declare** quotes its arguments and **declare** always returns done as output.

Following properties are available:

evfun, evflag, bindtest, noun, constant, scalar, nonscalar, mainvaralphabetic, feature, rassociative, lassociative, nary, symmetric, antisymmetric, commutative, oddfun, evenfun, outative, multiplicative, additive, linear, integer, noninteger, even, odd, rational, irrational, real, imaginary, complex, increasing, decreasing, posfun, integervalued.

```
(%i3)  kill(all)$declare(z, complex);
       properties(z);
(%o1)  done
(%o2)  [database info, kind(z, complex)]
```

We can use the function **properties**(a) to retrieve the properties associated with the atom a.

3.2 Functions and Variables for Facts

The **assume**(pred_1, ..., pred_n) command adds predicates pred_1, ..., pred_n to the current context. If a predicate is inconsistent or redundant with the predicates in the current context, it is not added to the context.

The predicates pred_1, ..., pred_n can only be expressions with the relational operators <, <=, equal, notequal, >=, and >. Predicates cannot be literal equality = or literal inequality # expressions, nor can they be predicate functions such as **integerp.**

```
(%i3)  kill(all)$assume(a > b);is(a > b);facts();contexts;
(%o1)  [a > b]
(%o2)  true
(%o3)  [a > b]
(%o4)  [initial, global]
```

If you want to know the context of the item, **facts**(item) returns a list of the facts in the specified context. It returns a list of the facts known about item in the current context. Facts that are active, but in a different context, are not listed.

The function **facts**() (i.e., without an argument) lists the current context.

The function **forget** removes predicates established by **assume.** The predicates may be expressions equivalent to (but not necessarily identical to) those previously assumed. The general syntax is:

forget (pred_1, ..., pred_n) where pred1, ...pred_n is a list of predicates.

The function **forget**(L), where L is a list of predicates, forgets each item on the list.

```
(%i5)  forget(a > b);facts();
(%o5)  [a > b]
(%o6)  []
```

In order to determine whether the predicate expr **is** provable from the facts in the assume database use function **is:**

is(expr)

If the predicate is provably true or false, **is** returns true or false, respectively.

```
(%i7)  kill(all)$assume(a > b);is(a > b);
(%o1)  [a > b]
(%o2)  true
```

To determine whether the predicate expr is provable from the facts in the assume database use function **maybe:**

If the predicate is provably true or false, the function **maybe**(expr) returns true or false, respectively. Otherwise, **maybe** returns unknown.

```
(%i3)  maybe(a > b);maybe(%e < %pi);
(%o3)  true
(%o4)  true
```

3.3 Functions and Variables for Predicates

The function **compare**(x, y) returns a comparison operator op (<, <=, >, >=, =, or #) such that is(x op y) evaluates to true; when either x or y depends on %i and x # y, return notcomparable; when there is no such operator or Maxima isn't able to determine the operator, returns unknown.

```
(%i5)  compare(%e, %pi);
(%o5)  <
```

An expression is considered a constant expression if its arguments are numbers (including rational numbers, as displayed with /R/), symbolic constants such as π, **e**, and **i**, variables bound to a constant or declared constant by **declare**, or functions whose arguments are constant.

The function **constantp**(expr) returns true if expr is a constant expression, otherwise returns false.

```
(%i6)  constantp(%e);
(%o6)  true
```

The function **equal**(a, b) represents equivalence, that is, equal value. By itself, the function equal does not evaluate or simplify.

The negation of equal is notequal.

```
(%i7)  kill(all)$a:2;is(equal(a, 2));
(%o1)  2
(%o2)  true
```

Maxima recognizes certain mathematical properties of functions and variables. These are called **features.**

The predicate **featurep**(x, pro) returns true if x has the pro property, and false otherwise. The infolist features are integer, noninteger, even, odd, rational, irrational, real, imaginary, complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, commutative, lassociative, rassociative, symmetric, and antisymmetric, plus any user-definedfeatures.

```
(%i3)  declare(const, constant);
       featurep(const, constant);
(%o3)  done
(%o4)  true
```

The predicate **numberp**(expr) returns true if expr is a literal integer, rational number, floating point number, or bigfloat, otherwise false. **numberp** returns false if its argument is a symbol, even if the argument is a symbolic number such as π or **i**, or declared to be even, odd, integer, rational, irrational, real, imaginary, or complex.

```
(%i5)  a:9;numberp(9);numberp(%e);numberp(a);
(%o5)  9
(%o6)  true
(%o7)  false
(%o8)  true
```

4 DATA TYPES AND STRUCTURES

4.1 Numbers

The integers, floating numbers and complex numbers are available in Maxima.

The function **declare** assigns the atom or list of atoms a_i the property or list of properties p_i. The syntax is: **declare**(a_1, p_1, a_2, p_2, ...).

When a_i and/or p_i are lists, each of the atoms gets all of the properties.

The feaures are :

integer	noninteger	even
odd	rational	irrational
real	imaginary	complex
analytic	increasing	decreasing
oddfun	evenfun	posfun
commutative	lassociative	rassociative
symmetric	antisymmetric	

plus any user-defined feature.

4.2 Strings

Strings (quoted character sequences) are enclosed in double quote marks "for input, and displayed with or without the quote marks, depending on the global variable **stringdisp.**

Strings may contain any characters, including embedded tab, newline, and carriage return characters. The sequence \" is recognized as a literal double quote, and \\ as a literal backslash. When backslash appears at the end of a line, the backslash and the line termination (either newline or carriage return and newline) are ignored, so that the string continues with the next line.

There is no character type in Maxima; a single character is represented as a one-character string.

The **stringproc** add-on package contains many functions for working with strings.

```
(%i9)  'x = x:"Gur";'y = y:"preet";
       concat(x, y);
(%o9)  x = Gur
(%o10) y = preet
(%o11) Gurpreet
(%i12) concat("Gurpreet", " Singh");
(%o12) Gurpreet Singh
```

4.3 Constants

Some of the constants defined in the Maxima are:

1. **%e** represents the base of the natural logarithm, also known as Euler's number. Its value is (2.718281828459045d0).
2. **%i** represents the imaginary unit, sqrt(−1).
3. **%gamma** represents the Euler-Mascheroni constant, 0.5772156649015329 ..
4. **ind** represents a bounded, indefinite result.
5. **inf** represents real positive infinity.
6. **infinity** represents complex infinity.
7. **minf** represents real minus (i.e., negative) infinity.
8. **%phi** represents the so-called golden mean, (1 + sqrt(5))/2. The numeric value of **%phi** is the double-precision floating-point value 1.6180339887498895d0.
9. **%pi** represents the ratio of the perimeter of a circle to its diameter. The numeric value of **%π** is the double-precision floating-point value 3.141592653589793d0.
10. **true** represents the boolean constant of the same name. Maxima implements true by the value T in Lisp.(Likewise false).
11. **und** represents an undefined result.
12. **zeroa** represents an infinitesimal above zero. **zeroa** can be used in expressions. **limit** simplifies expressions which contain infinitesimals.
13. **zerob** represents an infinitesimal below zero. **zerob** can be used in expressions. **limit** simplifies expressions which contain infinitesimals.

4.4 Lists

Lists are the basic building block for Maxima and Lisp. All data types other than arrays, hash tables, numbers are represented as Lisp lists. These are represented as:

```
(%i13) [1, %pi, 2, 45, i + j, a];
(%o13) [1, p, 2, 45, j + i, 9]
```

Above, represents a list of six elements.

Operator: [ ]

[and] mark the beginning and end, respectively, of a list. [and] also enclose the subscripts of a list, array, hash array, or array function.

```
(%i14) kill(all)$L:[x, y, z];L[2];
(%o1)  [x, y , z]
(%o2)  y
(%i3)  g[k]: = 1/(k^2 + 1);
       g[3];
(%o3)  g_k:=1/(k^2+1)
(%o4)  1/10
```

4.5 Arrays

The function **arrays** creates an n-dimensional array. n may be less than or equal to 5. The subscripts for the i^{th} dimension are the integers running from 0 to dim_i.

There are three types of **array** functions:

1. **array**(name, dim_1, ..., dim_n) creates a general array.
2. **array**(name, type, dim_1, ..., dim_n) creates an array, withelements of a specified type.
3. **array**([name_1, ..., name_m], dim_1, ..., dim_n) creates m arrays, all of defined dimensions.

These comprise arrays declared by array, hashed arrays constructed by implicit definition (assigning something to an array element), and array functions defined by : = and define.

```
(%i5)  kill(all)$array(A, 2, 2);A[0, 0]:4;arrayinfo(A);
(%o1)  A
(%o2)  4
(%o3)  [declared, 2, [2, 2]]
```

The function **listarray**(A) returns a list of the elements of the array A. The argument A may be a declared array, an undeclared(hashed) array, an array function, or a subscripted function.

```
(%i4) listarray(A);
(%o4) [4, #####, #####, #####, #####, #####, #####, #####, #####]
```

4.6 Structures

Maxima provides a simple data aggregate called a structure. A structure is an expression in which arguments are identified by **name**(the field name) and the expression as a whole is identified by its **operator**(the structure name). A field value can be any expression.

defstruct(S(a_1, ..., a_n))
defstruct(S(a_1 = v_1, ..., a_n = v_n))

The function **defstruct** defines a structure, which is a list of named fields a_1, ..., a_n associated with a symbol S. An instance of a structure is just an expression which has operator S and exactly n arguments. The function **new**(S) creates a new instance of structure S.

An argument which is just a symbol a specifies the name of a field. An argument which is an equation a_1 = v_1 specifies the field name a_1 and its default value v_1. The default value can be any expression.

```
(%i5)  kill(all)$declare([i, j, k], integer)$
       defstruct(DATE(i, j, k));
       u:new (DATE(83, 16, 200));u@k;
(%o2)  [DATE(i, j, k)]
(%o3)  DATE(83, 16, 200)
(%o4)  200
```

The function **new** creates new instances of structures. The variants are:

new(S)
new(S (v_1, ..., v_n))

The new(S) creates a new instance of structure S in which each field is assigned its default value, if any, or no value at all if no default was specified in the structure definition.

The **new**(S(v_1, ..., v_n)) creates a new instance of S in which fields are assigned the values v_1, ..., v_n.

```
(%i5)  defstruct(DATE(a, b));
       dt:new(DATE(a = 2, b = 3));
(%o5)  [DATE(a, b)]
(%o6)  DATE(a = 2, b = 3)
```

The symbol @ is the structure field access operator. The expression x@ a refers to the value of field a of the structure instance x. The field name is not evaluated.

```
(%i7) dt@b;
(%o7) b = 3
```

5 OUTPUT/INPUT COMMANDS

5.1 Output Command

For providing information at any line we need to have a comment symbol. A comment in Maxima input is any text between **/* and */** .
/* Comments can be nested to any level by inserting /* */ .

```
(%i8)  /*This is a comment /*It is a subcomment*/*/1 + 9;
(%o8)  10
```

The function **display** displays equations whose left side is expr_i unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and for statements in order to have intermediate results displayed. The arguments to display are usually atoms, subscripted variables, or function calls. The general form is :
display(expr_1, expr_2, ...expr_n).

```
(%i9)  a = 7;display("This is display command", a*8, 9^3);
(%o9)  a = 7
This is display command = This is display command
a 8 = 8 a
9^3 = 729
(%o10) done
```

The **disp**(expr_1, expr_2, ...) is like function display but only the value of the arguments are displayed rather than equations. This is useful for complicated arguments which don't have names or where only the value of the argument is of interest rather than its name.

```
(%i11) a = 7;disp("This is disp command", a*8, 9^3);
(%o11) a = 7
This is disp command
8 a
729
(%o12) done
```

The function **ldisp**(expr_1, ..., expr_n) displays expressions expr_1, ..., expr_n to the console as printed output. **ldisp** assigns an intermediate expression label to each argument and returns the list of labels.

```
(%i13) a = 7;ldisp("This is ldisp command", e:a*8, f:9^3);
(%o13) a = 7
(%t14) This is ldisp command
(%t15) 8 a
(%t16) 729
(%o16) [%t14, %t15, %t16 ]
```

The function **ldisplay**(expr_1, ..., expr_n) evaluates and displays expressions expr_1, ..., expr_n to the console as printed output. Each expression is printed as an equation of the form lhs = rhs in

which lhs is one of the arguments of **ldisplay** and rhs is its value. Typically, each argument is a variable. **ldisp** assigns an intermediate expression label to each equation and returns the list of labels.

```
(%i17) a = 7;ldisplay("This is ldisplay command", e:a*8, f:9^3);
(%o17) a = 7
(%t18) This is ldisplay command = This is ldisplay command
(%t19) e = 8 a
(%t20) f = 729
(%o20) [%t18, %t19, %t20]
```

The function **print**(expr_1, …, expr_n) evaluates and displays expr_1, …, expr_n one after another, from left to right, starting at the left edge of the console display. The value returned by the function **print** is the value of its last argument. The function **print** does not generate intermediate expression labels.

```
(%i21)  print("Hello World", 6 + 7)$
Hello World 13
```

5.2 Input Command

The input to Maxima can be either from console or from a file.

The **read**(expr_1, ..., expr_n) command prints expr_1, ..., expr_n, then reads one expression from the console and returns the evaluated expression. The expression is terminated with a semicolon ; or dollar sign $.

```
(%i22) kill(all)$block(num:read("Give a number: "),
       print("You have given number : ", num));
Give a number:
(%i1)  a:read("Give a number or alphabet: ");a;
Give a number or alphabet:
You have given number : a
(%o1)  a
```

The **readonly**(expr_1, ..., expr_n) prints expr_1, ..., expr_n, then reads one expression from the console and returns the expression (without evaluation). The expression is terminated with a ; (semicolon) or $ (dollar sign).

```
(%i2)  a:readonly("Give a number: ")$a^2;
```

Give a number: 23;

```
(%o3)  23
```

5.3 System Variables

__ (Double dash): __ is the input expression currently being evaluated i.e while an input expression expr is being evaluated, __ is expr. It is assigned the input expression before the input is simplified or evaluated. However, the value of __ is simplified (but not evaluated) when it is displayed. The __ is recognized by the functions **batch** and **load** both.

```
(%i4)  kill(all)$a:4;b:19;a.b;
       print("is the product of a and b", __);
(%o1)  4
(%o2)  19
(%o3)  76
is the product of a and b print(is the product of a and b, __)
(%o4)  print(is the product of a and b, __)
```

_ (Single dash): The operator _ is the most recent input expression (e.g., %i1, %i2, %i3, ...). The _ is assigned the input expression before the input is simplified or evaluated. However, the value of _ is simplified (but not evaluated) when it is displayed. The _ is also recognized by the functions **batch** and **load** both.

```
(%i5)  23*89;
(%o5)  2047
(%i6)  _;
(%o6)  2047
```

%: The operator % is the output expression (e.g., %o1, %o2, %o3, ...) most recently computed by Maxima, whether or not it was displayed. The % is recognized by both **batch** and **load** functions. In a file processed by **batch,** % has the same meaning as at the interactive prompt. In a file processed by **load,** the % is bound to the output expression most recently computed at the interactive prompt or in a batch file; the % is not bound to output expressions in the file being processed.

6 RELATIONAL OPERATORS

Maxima has the inequality operators < , < = , > = , > , # (not equal) and equality **operator** = (equal). It is to be noted that Maxima has different assignment operator and relational operator of equality.

```
(%i7)  'x = x:9;'y = y:3 + 6;is(x#y);
(%o7)  x = 9
(%o8)  y = 9
(%o9)  false
```

```
(%i10) is(2 < 3);
(%o10) true
```

7 LOGICAL OPERATORS

The logical operators defined in Maxima are: **and, not & or.**

1. **and** : The logical conjunction operator. The operator **and** is an n-ary infix operator; its operands are boolean expressions, and its result is a boolean value. The operator **and** is not commutative: a **and** b might not be equal to b **and** a due to the treatment of indeterminate operands.
2. **not** : The logical negation operator. The operator **not** is a prefix operator; its operand is a Boolean expression, and its result is a boolean value.
3. **or** : The logical disjunction operator. The operator **or** is an n-ary infix operator; its operands are boolean expressions, and its result is a boolean value.

```
(%i11) kill(all)$T:true;F:false;
       T and F;
       T or F;
(%o1)  true
(%o2)  false
(%o3)  false
(%o4)  true
```

8 CONTROL STATEMENTS

Here, we study the following control statements:

1. **Block** statement.
2. Logical **if.**
3. **Do** statement.

8.1 Block Statement

The **block** statement can be used to bind certain statements to produce a clean program.

The general forms are:

block([v_1, ..., v_m], expr_1, ..., expr_n)

block(expr_1, ..., expr_n)

The block evaluates expr_1, ..., expr_n in a sequence and returns the value of the last expression evaluated. Some variables v_1, ..., v_m can be declared local to the block; these are distinguished from global variables of the same names. If no variables are declared local, then the list may be omitted. Within the block, any variable other than v_1, ..., v_m is a global variable.

The **return**(value) may be used to exit explicitly from a block.

The go(tag) within a block is used to transfer control to the statement of the block which is tagged with the argument **go**. To tag a statement, precede it by an atomic argument as another statement in the block.

```
(%i5)  block([x, y, z], cos(x), tan(y), cot(z));
(%o5)  cot(z)
(%i6)  block(x:4, loop, display(x),
       if (x > 1) then (x:x - 1, go(loop))
       else return(0));
                                    x = 4
                                    x = 3
                                    x = 2
                                    x = 1
(%o6)  0
```

8.2 Logical If

The control statement **logical if** represents the conditional evaluation. The various forms of if expressions are recognized in Maxima.

1. if cond_1 **then** expr_1 **else** expr_0 evaluates to expr_1 if cond_1 evaluates to true, otherwise the expression evaluates to expr_0.(Single branching)
2. if cond_1 **then** expr_1 **elseif** cond_2 **then** expr_2 **elseif ... else** expr_0 evaluates to expr_k if cond_k is true and all preceding conditionsare false. If none of the conditions are true, the expression evaluates to expr_0. A trailing else false is assumed if else is missing. (Multiple branching). The alternatives expr_0, ..., expr_n may be any Maxima expressions, including **nested if** expressions.

The alternatives are neither simplified nor evaluated unless the corresponding condition is true. The conditions cond_1, ..., cond_n are expressions which potentially or actually evaluate to true or false i.e relational expressions.

```
(%i7)  if (12 > 3) then "True" else "False";
(%o7)  True
```

Now, we can write a function called **signum.** If a given number is positive then the function returns 1, for negative numbers – 1 , else 0.

```
(%i8)  signum(n): = block(if (n > 0) then return(1)
       elseif (n < 0) then return(- 1)
       else return(0));
(%o8)  signum(n): = block
```

The go(tag) within a block is used to transfer control to the statement of the block which is tagged with the argument **go**. To tag a statement, precede it by an atomic argument as another statement in the block.

```
(%i5)  block([x, y, z], cos(x), tan(y), cot(z));
(%o5)  cot(z)
(%i6)  block(x:4, loop, display(x),
       if (x > 1) then (x:x - 1, go(loop))
       else return(0));
                                    x = 4
                                    x = 3
                                    x = 2
                                    x = 1
(%o6)  0
```

8.2 Logical If

The control statement **logical if** represents the conditional evaluation. The various forms of if expressions are recognized in Maxima.

1. if cond_1 **then** expr_1 **else** expr_0 evaluates to expr_1 if cond_1 evaluates to true, otherwise the expression evaluates to expr_0.(Single branching)
2. if cond_1 **then** expr_1 **elseif** cond_2 **then** expr_2 **elseif ... else** expr_0 evaluates to expr_k if cond_k is true and all preceding conditionsare false. If none of the conditions are true, the expression evaluates to expr_0. A trailing else false is assumed if else is missing. (Multiple branching). The alternatives expr_0, ..., expr_n may be any Maxima expressions, including **nested if** expressions.

The alternatives are neither simplified nor evaluated unless the corresponding condition is true. The conditions cond_1, ..., cond_n are expressions which potentially or actually evaluate to true or false i.e relational expressions.

```
(%i7)  if (12 > 3) then "True" else "False";
(%o7)  True
```

Now, we can write a function called **signum.** If a given number is positive then the function returns 1, for negative numbers – 1 , else 0.

```
(%i8)  signum(n): = block(if (n > 0) then return(1)
       elseif (n < 0) then return(- 1)
       else return(0));
(%o8)  signum(n): = block
```

Operation	Symbol	Type
less than	<	relational infix
less than or equal to	<=	relational infix
equality (syntactic)	=	relational infix
negation of =	#	relational infix
equality (value)	equal	relational function
negation of equal	notequal	relational function
greater than or equal to	>=	relational infix
greater than	>	relational infix
and	and	logical infix
or	or	logical infix
not	not	logical prefix

(if $n > 0$ then return(1) elseif $n < 0$ then return(– 1) else return(0))

```
(%i9)  signum(- 9);signum(10);
(%o9)  -1
(%o10) 1
```

Write a program to read a number and then display whether the number is either positive or negative?

```
(%i11) block(kill(all), [a], a:read("Give a number"),
       if (a > = 0) then print(a, "is positive") else print(a, "is
       negative"))$

Give a number - 9;
 - 9 is negative
```

Write a program to read a number and then display whether the number is either even or odd.

```
(%i1)  block(kill(all), [x], x:read("Give a number"),
       if(mod(x, 2) = 0) then print(x, "is even")
       else print(x, "is odd"))$

Give a number 77;
77 is odd
```

Find prime numbers between two given integers and count them.

```
(%i1)  block(kill(all), [min, max, i, j:1],
       min:read("Give lower limit"),
```

```
        max:read("Give upper limit"),
        for i:min thru max do
        (if (primep(i) = true) then (print(j," ",i), j:j + 1)),
        print("Total Prime Numbers:", j - 1));
Give lower limit 10;
Give upper limit 20;
1 11
2 13
3 17
4 19
Total Prime Numbers: 4
(%o0)  4
```

8.3 DO STATEMENT

The **do** statement is used for performing iterations. There are three variants of this form any they differ only in their terminating conditions. They are:

1. **for variable:** initial_value **step** increment **thru** limit **do** body
2. **for variable:** initial_value **step** increment **while** condition **do** body
3. **for variable:** initial_value **step** increment **unless** condition **do** body

```
(%i1) for a:0 thru 10 step 2 do display(a);
      a = 0
      a = 2
      a = 4
      a = 6
      a = 8
      a = 10
(%o1) done
(%i2) for a:0 while a < 5 do display(a);
      a = 0
      a = 1
      a = 2
      a = 3
      a = 4
(%o2) done
```

```
(%i3) for a:0 unless a > 5 do display(a);
       a = 0
       a = 1
       a = 2
       a = 3
       a = 4
       a = 5
(%o3) done
```

Write a program to find sum of digits of an integer given by a user.

```
(%i4) block(kill(all), [sum, nu, i, dig1, num],
       num:read("Give an integer whose digit to be added:"),
       sum:0, nu:num,
       for i:1 thru 10 do
           (dig1:mod(num, 10), sum:sum + dig1, num:fix(num/10)),
                print("Sum of digits of", nu, "is: ", sum))$
Give an integer whose digit to be added: 54689;
Sum of digits of 54689 is: 32
```

To find absolute value of a number given by a user.

```
(%i1) block(kill(all), [num], num:read("Give an integer:"),
       if (num > 0) then
           print("The absolute value is :", num)
       else
           print("The absolute value is :", - num))$
Give an integer: - 65;
The absolute value is : 65
```

Following program finds the root of any given equation f(x) using bisection Method. User can remove quote from **display**(a) in else block to get output of each iteration. The function f(x) can be redifined by the user:

```
(%i1) kill(all)$x0:0$x1:2.0$f(x): = x^2 - 2; i:0$
       if (float(f(x0)*f(x1)) > 0)
           then print("change values")
       else
           for i:1 thru 50 do (a:(x0 + x1)/2,
           if (f(a)*f(x1) > 0)
```