Chapter 4

Computer-assisted research: programming and graphing

4.1 Programming

4.1.1 Development process

A *program development cycle* is the incremental process of building up your code, testing and debugging. It involves four steps:

- Write and modify source code of your program in a text editor.
- Create an executable file from the program source file with a compiler.
- Run your program.
- Observe the effect and decide if further changes in the code are needed.

A computer cannot directly run a program source file, which is a human-readable text file. The source file must be translated into an executable (binary) file, which the computer understands. A *compiler* is a program that does this translation. For each programming language there is its own compiler, sometimes more than one. See information about the compilers available on local machines in Section **??**.

Often during the development cycle you have to repeat certain commands over and over again, for example,

```
gcc project2.c
a.out
```

To minimize typing, you can use a "history" keystroke: Esc-K or ↑ (see Sect. **??**).

A more fundamental time-saving suggestion concerns **testing and debugging**. Do not bother to create a friendly user interface or to add various features to your program until you achieve **basic functionality**. Suppose your program is to compute the area of a polygon given the coordinates of its vertices. Eventually you want the program to prompt the user to enter the number of vertices, N, and their coordinates, one by one, like this:

```
Please enter the number of vertices: N=  3
Vertex #1: x=  1.1
           y=  0.4
Vertex #2: x=  3.4
           y=  -4.56
Vertex #3: x=  3.12
           y=  -9.4
```

However, do not begin programming by creating the input interface. Instead, put a temporary initialization block with fixed, **hardcoded** data at the beginning of the program; use simple data to enable an easy check by hand calculation.

```
N=3
x[1]= 0
y[1]= 0
x[2]= 2
y[2]= 0
x[3]= 0
y[3]= 3
```

This set of data corresponds to a right triangle with two sides running along the coordinate axes. Then, as the mathematical part of your program matures, you should change the data: test the program on a triangle that is acute or obtuse; shift it; then test the program on a quadrilateral, etc. As you will be catching mistakes in the program, you will have to run it more than once on each sample data set, while modifying the data infrequently. Eventually you will spend much less time on data input than you would spend via the input prompt.

Syntax errors that depend on a programming language and a compiler used cannot be discussed here in any depth. We will just mention some **common errors** that appear frequently.

- Unmatched bracket in a mathematical expression (() or unmatched opening of a structure in a program, like `DO` loop without closing "`END DO`" in FORTRAN. Most notably, when you find such an error and try to fix it, there is a danger that you *misplace* the closing bracket or the closing keyword and make the problem worse, more difficult to detect.
- All variables in the program must be assigned values before their values are used for the first time. A randomly looking output is the most common consequence of the *failure to initialize*. However, a compiler may initialize your variables to 0 by default — not to the values that ought to be there; the results may look OK at first sight but still be wrong.

- Initializations should be placed *outside* loops that they are supposed to initialize. If the variable `SUM` is an accumulator for a sum computed by a loop, you should put the initialization `SUM=0` outside the loop. This problem is especially common with *nested* loops. You must carefully identify "fast" variables, which must change in the inner loop, and "slow" variables, which change only once per the outer loop.

- "Off by one" error is common even with seasoned programmers. Differentiate between *strict* and *non-strict* inequalities (`i>0` vs. `i>=0`). Check whether it is possible that your loop will skip immediately (say, the loop condition is `while(i>j)` and the initial value of `i` is equal to the initial value of `j`). If this is possible in the program, was it meant?

- Quite often, special arrangements are to be made on the first and/or last pass of a loop. For example, in a loop that produces n pairs of coordinates separated by commas, the comma should be printed only $n-1$ times, — see example on p. 86.

- Array indexing. In FORTRAN and Maple, if the array has N elements, the index runs from 1 to N, while in C and Java it runs from 0 to $(N-1)$. Also, when you update the array and the new values depend on the old values, watch the order of the update closely. Consider, for example, the cyclic permutation `x(1)`$\to$ `x(2)`$\to \ldots \to$ `x(n)`$\to$ `x(1)`:

  CORRECT

  ```
  tmp=x(n)
  FOR i=1, n-1
    x(n-i+1)=x(n-i)
  END DO
  x(1)=tmp
  ```

  WRONG

  ```
  tmp=x(n)
  FOR i=2, n
    x(i)=x(i-1)
  END DO
  x(1)=tmp
  ```

  In the wrong code, the old value of `x(1)` will propagate through the array, while the old values of `x(2), x(3), ..., x(n)` will be lost.

- When you use output to files in C or FORTRAN, make sure to *close* the file; otherwise a part of the output may be lost. Close the file *outside* the outermost loop: otherwise, a multiple closure will cause the program to crash.

- A confusion between the assignment operator and the equality condition ('`=`' vs. '`==`' in C and Java, '`:=`' vs. '`=`' in Maple and Pascal). Make sure you understand the difference! Other symbols to watch carefully are comma vs. semicolon, and various kinds of brackets.

Students often get lost when a program does not behave the way it is supposed to. How is it possible to find errors that affect functionality and are not easy to catch? A regular approach to find and fix a mistake is to insert **temporary output operators** and, using simple test data (cf. p. **??**), to trace the intermediate results comparing them with those calculated by hand. If the program's functioning disagrees with your mathematical algorithm (most often, due to a typo), you will detect a discrepancy at some point. Debugging is more complicated if the mathematical method is flawed in itself. Dissecting the problem (and the program) into smaller steps is still a dependable approach.

4.1.2 Programming style

Your program code should be reasonably self-contained and documented. Put the following at the top of a program:

- Author's name
- Date
- Course and project number
- A brief description of the program (what it does)
- Additional information if several programs have been written for this project

Note that the readability of a program is improved and debugging is helped immensely by the generous insertion of **comments**. Ask yourself: will you be able to understand your program in a half a year period of time?

Use **indentation** to improve readability of loops, especially long ones, and if/else clauses.

GOOD

```
for (i=1; i<=n; i++)
{
   if (a[i]>0)
      sum=sum+a[i];
   else
      sum=sum-a[i];
}
```

BAD

```
for (i=1; i<=n; i++)
{
if (a[i]>0)
sum=sum+a[i];
else  sum=sum-a[i];
}
```

Use **meaningful names** of variables and functions, but don't make them too long. A variable name like `AreaOfTriangle` is hardly better than just `Area`. If practical, use very short names that match the notation you use in the description of the mathematical method. Strive for **consistency** in your programming style, as in everything else.

Good programmers tend to use **modular approach** (subroutines in FORTRAN, functions in C, classes in C++ and Java) to make the structure of a program more transparent and to confine those few cumbersome mathematical lines of code. Modularity also makes debugging easier. However, in programs with simple "linear" structure or in short programs modularity can be a burden rather than a benefit.

Appropriate **generalization** is another feature of a solid style. Make your program flexible, make it easy to play with parameters. An example of a coordinate-generating code on the next page will help you to grasp the idea.

4.1.3 Generating graphics data with your own program

We will discuss two-dimensional graphs only. Essentially, every graph you generate is determined by points, each point being a pair of coordinates (x, y). Continuous mathematical curves, consisting of infinitely many points, are most commonly approximated by polygonal lines consisting of segments whose endpoints are to be computed by the program.

When writing a program, you have to make a decision as to what the **bounds for x and y** should be (if the mathematical curve is infinite), and **by how many points** you want to define the curve. To get a rough estimate of a reasonable number, let us take 5 *in* as a largest dimension of a picture and note that human eye, even sharp, can hardly resolve distances less than 1/200 of an inch. Thus $5 \times 200 = 1000$ data points across a sheet is perhaps enough in most cases. The smoother a curve, the smaller number of points will suffice: often 50 or even 10.

A good idea is to have **variables** in your program for the x and y limits and for the number of points, rather than to use specific numbers throughout the code. Compare the two fragments of FORTRAN code:

GOOD

```
xmin=-5
xmax=5
numpoints=20
xstep=(xmax-xmin)/numpoints
DO i=0,numpoints
   x=xmin+i*xstep
   y=SIN(x)
   PRINT *,"(",x,",",y,")"
END DO
```

BAD

```
DO i=0,20
   x=-5+i*0.5

   PRINT *,"(",x,",",SIN(x),")"
END DO
```

The "bad" code does not look bad at all: it is concise, easy to understand, and correct. But it has two drawbacks:
(1) the code conceals the meaning of the numbers: what are those "20" and "-5", and "0.5"?;
(2) if the values that are denoted `xmin`, `xmax`, `numpoints` in the "good" code occur somewhere else in the program and you wish to change all or some of the values, it is easy to do: you just need to change the values assigned to the symbolic names, in one place only.

In simple cases, like in the example above, the points are generated independently of each other; as soon as a point has been computed, it can be printed immediately. In more complicated cases, when dependency of some sort is present, you may have to create an array and to complete computation of all the points before you can print them out.

Another suggestion: use **scaling** parameters and **translation** parameters. The "good" code above is good, in particular, because it is easy to implement this suggestion. Modify the output operator as follows:

```
PRINT *, "(",xscale*(x-xorigin),",",yscale*(y-yorigin),")"
```

This stretches (or shrinks) the distances by a factor `xscale` in the x-direction and by a factor `yscale` in the y-direction. In addition, the point where (`x,y`)=(`xorigin,yorigin`) will be printed as (0,0), that is, it will become the origin of the coordinate system on the plotting device. The trivial default values can be initialized: `xscale=1`, `yscale=1`, `xorigin=0`, `yorigin=0`. If you are not satisfied with size or position of your graph, it will be very easy to change.

Formatting coordinates

You must format the coordinates in accordance with the method you intend to use to render your graph. Every pair of numbers must be "framed" with *opening* symbol or keyword preceding the x value and *closing* symbol or keyword following the y value, and a *separator* must be inserted between the x and y values. The list of coordinates as a whole has its own opening (before the first point), closing (after the last point) and a separator (between the successive points). Some of these can be void or blank space.

Graphing facility	LaTeX picture	Postscript	Maple	Gnuplot
Before x	`(`	none	`[`	none
Between x and y	`,`	space	`,`	space
After y	`)`	`moveto` or `lineto`	`]`	none
List opening	`\join`	`newpath`	`[`	none
Between points	none	line break	`,`	line break
List closing	none	`stroke` or `fill`	`]`	none

Table 4.1: Openings, closings, and separators for coordinates

A summary of the various coordinate formats mentioned in this Manual is given in Table 4.1. Consider for example a line defined by three points $(-1,1)$, $(0,0)$ and $(1,1)$ (a very rough approximation to a graph of $y = x^2$). Below we show what the data file produced by your program should look like in different cases. In LaTeX and Postscript, a change of scale may be necessary to actually see the picture.

- LaTeX: you must include package `2130.sty` or `curvesb.sty` to enable the `join` command. Import the data file by the `input` command, see p. 120.

```
\join (-1,1)(0,0)(1,1)
```

- Postscript: you must add the heading `%!PS-Adobe-2.0` by hand or have your program to print it automatically.

```
newpath
-1 1 moveto
 0 0 lineto
 1 1 lineto
stroke
```

- Maple: you must cut and paste this array to Maple's `plot` command.

```
[[-1,1],[0,0],[1,1]]
```

- Gnuplot: feed the file to Gnuplot's `plot` command; use option `with lines`.

```
-1 1
 0 0
 1 1
```

Since the list opening and list closing are to be printed only once, this can be done outside of the coordinate-generating loop. On the other hand, the separator between the pairs must be printed after each pair save the last one. So it must be done within the loop; the last pass of the loop must be a little different. We present short FORTRAN and C codes that produce the data in **Maple's style**. For simplicity of presentation, we sacrificed any flexibility, in violation of a good programming style we promote.

FORTRAN

```
OPEN (UNIT=1, FILE ="line1.dat")
WRITE(1,*) "[" !List opening
DO i=0,2
   x=i-1
   y=x**2
   WRITE(1,*),"[",x,",",y,"]"
   IF (i<2) THEN
     WRITE(1,*),"," !separator
   END IF
END DO
WRITE(1,*) "]" !List closing
CLOSE(1)
```

C

```
FILE *f =fopen("line1.dat","w");
fprintf(f,"[");//List opening
for (int i=0; i<=2; i++)
{
   x=i-1;
   y=pow(x,2);
   fprintf(f,"[%f,%f]",x,y);
   if (i<2)
     fprintf(f,","); //separator
}
fprintf(f,"]");//List closing
fclose(f);
```

If, instead of connecting the points by lines, you need to render them differently, your program can be written accordingly. For example, the following line in a C program will print a LaTeX command that puts a small solid circle in a specified position.

```
printf("\put(%f,%f){\circle*{0.1}}",x,y);
```

An equivalent FORTRAN code is

```
PRINT *, "\put(", x, "," ,y, "){\circle*{0.1}}"
```

4.2 An introduction to Maple

Maple is a computer algebra system (CAS) created around 1980 by a team of researchers based at the University of Waterloo. Currently it is commercial software supported by Maplesoft, Inc. and it is available to MUN students through the LabNET-wide licence. It may be necessary for you to set up your account so that you can use Maple — see Section **??**.

Maple's most vigorous competitor is CAS *Mathematica*, a product of Wolfram Research, Inc. (USA). Another software that has many similar capabilities but focuses on matrix computations at the expense of sophisticated symbolic manipulations is *Matlab*. Users familiar with one of these systems will have little difficulty with another as soon as they understand the basic syntax and work out a few examples. In Math 2130, we focus on Maple.

Maple's functionality and interface have evolved over about 30 years. As of now, there exist three types of user interface in Maple. Two of them are graphical user interfaces: *standard* (modern) and *classic worksheet*, and the third is non-graphical *text-based* interface, which can be used in a command-line mode. Of the two graphical interfaces, the classic worksheet is the one which is easier to transform to a printed document. Our presentation will be based on the classic worksheet. The examples below were tested on Maple version 11 in November 2008. Maple graphics is dealt with in Sect. 4.3.2.

Maple can, in principle, save a worksheet in LaTeX format. However, this feature doesn't seem to be implemented carefully. We suggest that you paste fragments of your Maple code into your reports by hand (cf. Sect. **??**).

4.2.1 Basic Arithmetic and Algebra

To start Maple, open a terminal window and at the prompt, simply type

```
xmaple
```

Maple also has a classic worksheet option, which can be accessed by typing

```
xmaple -cw
```

at the prompt. Maple has a very useful help area, where you can find instructions on the many operations it can perform. In the classic worksheet, the *Help* command is located in the top right-hand corner. In the standard Maple, it is located under *Tools*. Also, in all interfaces of Maple, help can be accessed by typing

```
?help
```

Maple can be used as a calulator. Hit *Enter* to execute the command. The keystroke *Shift-Enter* makes carriage return without an immediate execution. Our first examples are:

```
> 4+3;
                                  7
> 2*5;
  6^2;
                                 10
```

```
36
```

Note the difference between exact and floating point operations:

```
> 2^64;
```

$$18446744073709551616$$

```
> 2.0^64;
```

$$1.844674407 \cdot 10^{19}$$

Let's see how to do slightly more interesting operations. Symbolic names can be used:

```
> y1:=x^3/2-9/2*x^2-2*x+6;
  y2:=(x^4-x^3-15*x^2+9*x+54)/(2*x^3-2*x^2-8*x+8);
```

$$y1 := \frac{1}{2}x^3 - \frac{9}{2}x^2 - 2x + 6$$

$$y2 := \frac{x^4 - x^3 - 15x^2 + 9x + 54}{2x^3 - 2x^2 - 8x + 8}$$

Expressions can be symbolically factored:

```
> y1f:=factor(y1);
```

$$y1f := \frac{(x-1)(x^2 - 8x - 12)}{2}.$$

Or expanded:

```
> expand(y1f);
```

$$y1 := \frac{1}{2}x^3 - \frac{9}{2}x^2 - 2x + 6$$

A remark on symbolic names. Some names in our examples (`x`, `y1`, `y2`, `y1f`) denote *user-defined variables* — these names are arbitrary, you can change them any way you like. Other names are *keywords* known to Maple (like `factor`, `expand`). These names are protected; an attempt to assign a value to them will prompt an error message. Maple knows a few special constants, which are also protected. The most famous one is `PI` ($\pi = 3.14159\ldots$); not so many students are familiar with Euler's constant `gamma` ($\gamma = 0.57721\ldots$). You may be surprised that the symbols `e` and `E` are not protected; the natural logarithm base $e = 2.71828\ldots$ can be accessed in Maple as `exp(1)` (in symbolic calculations) or as `exp(1.0)` (numerically). The availability of the symbol `e` is convenient for astronomers who use e to denote eccentricities of planetary orbits. The fact that `gamma` is reserved is unfortunate for geometers who like to denote the angles of a triangle as α, β, γ.

We continue a tour of basic Maple commands. The `simplify` command applies a bunch of algorithms to transform expressions to a simpler form. For example, it will identify common factors in the numerator and denominator and remove them. In our fraction `y2` defined above, Maple finds that the common factor $(x + 2)$ can be canceled:

```
> simplify(y2);
```

$$\frac{x^3 - 3x^2 - 9x + 27}{2(x^2 - 3x + 2)}$$

The `simplify` command also knows trigonometric identities:

```
> simplify(sin(theta)^2+cos(theta)^2);
```

$$1$$

Maple's simplification algorithms are powerful but not perfect. For example, Maple fails to notice that $(x+1)^{2n} - (x^2+2x+1)^n = ((x+1)^2)^n - (x^2+2x+1)^n = 0$:

```
> simplify((x+1)^(2*n)-(x^2+2*x+1)^n);
```

$$(x+1)^{(2n)} - (x^2+2x+1)^n$$

For any particular n, Maple will simplify correctly, but it can take a long time.

```
> simplify((x+1)^(2*700)-(x^2+2*x+1)^700);
```

$$0$$

Another useful command is substitution, `subs`:

```
> subs(x=Pi/4, sin(x)):    simplify(%);
```

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

The colon suppresses printout of a result (try to put a semicolon instead to see the effect). The percent sign refers to the most recent result.

4.2.2 Equations

Maple has built-in commands to solve equations automatically.

```
> solve(x^2+x-12=0,x);
```

$$3, \ -4$$

```
> XX:=solve(x^2+6*x+3,x);
```

$$-3+\sqrt{6}, \ \ -3-\sqrt{6}$$

The command `evalf` takes exact answers like those above and spits them out in decimal form:

```
> evalf(XX);
        -.550510257, -5.449489743
```

Higher accuracy is available through an optional argument of the `evalf` command.

```
> evalf(XX,20);
        -.55051025721682190 18, -5.4494897427831780982
```

Maple knows complex numbers, too. Note that symbol "I" in Maple is the imaginary number $\sqrt{-1}$, usually denoted as i.

```
> solve(x^2+x+1=0,x);
```

$$-\frac{1}{2}+\frac{1}{2}\mathrm{I}\sqrt{3},\quad -\frac{1}{2}-\frac{1}{2}\mathrm{I}\sqrt{3}$$

Maple's exact answers to equations of degrees 3 and 4 can be impractical. For roots of polynomials of degree 5 and higher no general formulas exist and, unless the equation can be factored, Maple will return gibberish. In all such cases, `evalf` can be used to get an approximation of the roots. The command `fsolve` returns approximations of real roots only.

```
> evalf(solve(x^3+x+1=0,x));
    -.6823278040, 0.3411639019-1.161541400 I, 0.3411639019+1.161541400 I
> fsolve(x^3+x+1=0,x);
                                -.6823278038
```

Maple can solve not only algebraic equations but many others, too. Beware, however, of its simplistic approach. Every math student knows that the equation $\cos x = 0$ has infinitely many solutions — but not Maple!

```
> solve(cos(x)=0);
```

$$-\frac{1}{2}\pi$$

Nor can Maple find all approximate solutions in a given interval containing many roots:

```
> fsolve(cos(x)=0, x=-100..200);
                                48.69468613
```

4.2.3 Calculus

Maple can do calculus both symbolically and numerically. Recall the expression `y1` from our example on page 88; we can use Maple for differentiation, indefinite and definite integration:

```
> y1;
```

$$\mathrm{y1} := \frac{1}{2}\mathrm{x}^3-\frac{9}{2}\mathrm{x}^2-2\mathrm{x}+6$$

```
> diff(y1,x);
```

$$\frac{3}{2}\mathrm{x}^2-9\mathrm{x}-2$$

```
> Iy1:=int(y1,x);
```

$$\mathrm{Iy1} := \frac{1}{8}\mathrm{x}^4-\frac{3}{2}\mathrm{x}^3-\mathrm{x}^2+6\mathrm{x}$$

```
> int(y1,x=-1..1);
```

$$9$$

Maple has commands that find extreme values of functions.

```
> maximize(Iy1);
```

$$\infty$$

```
> minimize(Iy1);
```

$$\frac{(4+2\sqrt{7})64}{8} - \frac{3(4+2\sqrt{7})^3}{2} - (4+2\sqrt{7})^2 + 24 + 12\sqrt{7}$$

The percent symbol can be used as a substitute for the result of the last executed command:

```
> evalf(%);
                          -302.1620735
```

The symbols `%%`, `%%%`, etc. refer to the results obtained so many steps back. By Maple's design, you can execute commands that are typed in your worksheet in any order (moving back and forth across the worksheet) simply by hitting *Enter* on a command. This practice should be avoided in worksheets that are to be saved and later read by you or another person, otherwise the results can mislead the reader. In particular, instead of using the percent sign, it is preferable to assign symbolic names to the results you want to re-use.

The `maximize` and `minimize` commands work only on certain functions — namely where no critical points exist or the equation for critical points can be solved exactly. Not the case here:

```
> maximize(x*cos(x), x=0..2);
 RootOf(tan(_Z) _Z -1, 0.8603335890) cos(RootOf(tan(_Z) _Z - 1, 0.8603335890))
```

The command `numapprox[infnorm]` can find the *maximum abslolute value* of more general functions. (The reader can rightly be curious about the command's name; look it up!)

```
> numapprox[infnorm](x*cos(x), x=0..2);
                          0.8322936731
```

Here we encounter for the first time an example of a function from a Maple *package*. The whole package whose name is `numapprox` can be uploaded by the command `with (numapprox);` and then you can use the `infnorm` command without prefix `numapprox`.

Maple is quite knowledgeable in Calculus. It knows limits and Taylor series.

```
> limit(sin(2*x)/ln(1-x), x=0);
                               -2

> taylor(tan(t),t, 6);
```

$$t + \frac{1}{3}t^3 + \frac{2}{15}t^5 + O(t^7)$$

We leave it to the reader to find out the meaning of the "big Oh" symbol `O(..)`.

4.2.4 Arrays

Data in Maple can be grouped to form an array. An array is bounded by square brackets and elements are separated by commas. The elements of an array can be objects of like or different nature; they can themselves be arrays. For example,

```
> A:=[1, 2, [red,blue], x^2-5*x+6, plot1];
```

The elements can be referenced using forward or backward indexing:

```
> A[1];
       1
> A[3];
      [red,blue]
> A[3][2];
       blue
> A[-1];
       plot1
```

The command `nops` returns the number of elements in an array:

```
> nops(A);
       5
```

A sub-array can be selected:

```
> A[3..4];
      [[red,blue], x^2-5*x+6]
```

It is straightforward to change the values of elements of an existing array by assignments like `A[1]:=...`, but adding new elements is tricky. We need the command '`op`' whose effect is just to remove the bounding brackets around the whole array:

```
> op(A);
      1, 2, [red,blue], x^2-5*x+6, plot1
```

To add a new element, say, `elem6`, the following command can be used:

```
> A:=[op(A), elem6];
      A:=[1, 2, [red,blue], x^2-5*x+6, plot1, elem6]
```

The command `seq` provides a convenient way to initialize an array with elements generated according to a given rule:

```
> B:=[seq(i^2, i=3..9)];
      B:=[9,16,25,36,49,64,81]
```

The command `map` performs the specified action on all elements of the array at once:

```
> map(sqrt, B);
      [3,4,5,6,7,8,9]
```

In the commands `seq` and `map` it is possible to use your own function. For example, the following will increment all elements of the array `B` by one:

```
> map(x->x+1, B);
      [10,17,25,37,50,65,82]
```

4.2.5 Linear Algebra

Linear algebra is available in Maple via either of two packages, `linalg` or `LinearAlgebra`. The former is not being updated anymore and will be eventually phased out. We will work with the latter package. First, load the library.

```
> with(LinearAlgebra):
```

Here are some basic operations: to create a matrix, to find the inverse, the determinant, the transpose, the characteristic polynomial, the eigenvalues and eigenvectors.

```
> A:=Matrix([[2, 4],[6,8]]);
```

$$\mathtt{A} := \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

```
>  A^(-1);
```

$$\begin{bmatrix} -1 & \frac{1}{2} \\ \frac{3}{4} & -\frac{1}{4} \end{bmatrix}$$

```
> Determinant(A);
```

$$-8$$

```
> Transpose(A);
```

$$\begin{bmatrix} 2 & 6 \\ 4 & 8 \end{bmatrix}$$

```
> CharacteristicPolynomial(A,lambda);
```

$$\lambda^2 - 10\lambda - 8$$

```
> Eigenvalues(A);
```

$$\begin{bmatrix} 5+\sqrt{33} \\ 5-\sqrt{33} \end{bmatrix}$$

```
> EValVec:=Eigenvectors(A);
```

$$\mathtt{EValVec} := \begin{bmatrix} 5+\sqrt{33} \\ 5-\sqrt{33} \end{bmatrix} \qquad \begin{bmatrix} \frac{4}{3+\sqrt{33}} & \frac{4}{3+\sqrt{33}} \\ 1 & 1 \end{bmatrix}$$

Note that the `Eigenvectors` command return the eigenvalues as well as the eigenvectors. We can select the matrix comprised of the eigenvectors:

```
> EVec:=EValVec[2];
```

$$\texttt{EVec} := \begin{bmatrix} \dfrac{4}{3+\sqrt{33}} & \dfrac{4}{3+\sqrt{33}} \\ 1 & 1 \end{bmatrix}$$

Now, if we want to separate the eigenvectors from one another and to make an array of them, the following command will do it:

```
> EVecArray:=[seq(Column(EVec,i), i=1..2)];
```

$$\texttt{EVecArray} := \left[\begin{bmatrix} \dfrac{4}{3+\sqrt{33}} \\ 1 \end{bmatrix}, \begin{bmatrix} \dfrac{4}{3+\sqrt{33}} \\ 1 \end{bmatrix} \right]$$

Matrix multiplication (and dot product of vectors) are denoted by a single dot (period).

```
> B:=Matrix([[2,0,-3],[1,2,1]]);
```

$$\texttt{B} := \begin{bmatrix} 2 & 0 & -3 \\ 1 & 2 & 1 \end{bmatrix}$$

```
> A.B;
```

$$\begin{bmatrix} 8 & 8 & -2 \\ 20 & 16 & -10 \end{bmatrix}$$

An attempt to multiply matrices when dimensions don't match leads to an error message.

```
> B.A;
```

```
Error, (in MatrixMatrixMultiply) first matrix column dimension (3) <> second
matrix row dimension (2)
```

4.2.6 Programming

Maple can be used for programming. It allows conditionals, loops, and user-defined functions.

Compared to languages such as FORTRAN or C, programming in Maple is more convenient: there is no need to bother about input/output operations and data types, commands can be executed immediately, and you have access to all of the built-in commands and available packages. The price to pay is efficiency. Maple runs loops a lot slower than compiled programs; also it runs out of memory on much smaller sizes of data. Yet, for many applications and many Math-2130 projects these issues are not critical.

Our first example is a summation loop, which computes the arithmetic sum $11 + 32 + 53 + 74 + 95$. Remember to use *Shift-Enter* to type multi-line commands into Maple (cf. page 87).

```
> tot := 0;
  for i from 11 by 21 while (i < 100) do
     tot := tot + i
  end do;
                                    tot :=   0
                                    tot :=  11
                                    tot :=  43
                                    tot :=  96
                                    tot := 170
                                    tot := 265
```

If you are not interested to see the intermediate results, simply replace the semicolons by colons in the above program and add the line `> tot;` to print the final result.

There are usual conditional commands `if-then-else`. For example, the following command finds the maximum of two numbers.

```
> a:=4: b:=2:
  if (a > b) then  a  else  b  end if;
                                      4
> a:=1: b:=7:
  if (a > b) then a else b end if;
                                      7
```

The closing commands `end do` and `end if` can be replaced by less traditional `od` and `fi`, respectively.

Consider an example of a very simple user-defined function, or procedure. It just adds one to a given number.

```
> increment:=proc(x) return (x+1): end proc:
> increment (2008);
                                  2009
```

Procedures can be much more involved; they may have many input values, local variables, and return values of any type. Procedures can contain loops, conditionals and calls to other procedures or to Maple's built-in functions.

There are also commands to test whether inputs are real, complex, matrices, or whatever.

```
> type(4,realcons);  type(c,realcons);
                                     true
                                     false
```

These could be used in `if-then` statements. For example, the following procedure returns the square root of the argument if the root is a real number, and prints an error message otherwise.

```
> SafeSqrt:=proc(x) local sqrtx:
    sqrtx:=sqrt(x):
    if  type(sqrtx,realcons) then
       return evalf(sqrtx)
    else
       print (Sqrt - Error)
    end if:
  end proc:

>  SafeSqrt(3);
              1.732050808
>  SafeSqrt(-3);
                Sqrt - Error
>  SafeSqrt(x^2);
                Sqrt - Error
```

4.3 Drawing graphs

Knowing how to generate graphs and incorporate them into a paper is a valuable skill for all technical writers. In this chapter we review a number of ways to generate plots with software.

The most common method to import computer-generated graphs into a LaTeX document involves **Encapsulated Postscript** files. We begin this section with basic information about Postscript, Encapulated Postscript, and LaTeX imports.

We then explain how graphs can be generated in a **Maple** worksheet and include brief descriptions of the **Gnuplot** and **Xfig** graphing packages.

LaTeX has its own graphical facility, the **picture environment**. Its drawing functions are very limited, but there are powerful enhancements, in particular, those included with `2130.sty` file. Besides, the *picture* environment can be used to create a desired layout of imported graphs or to superimpose graphs and text.

We do not advocate exclusive usage of any one of these graphing utilities. Try to draw pictures, and draw your conclusions. In all cases, you have to write a program which in part generates graphics data and it is up to you to select the utility that will handle a current graphics task best.

A practice **not allowed** in this course (except with an express permission of the instructor) is to import graphics files that are not your own production (downloaded from the Internet). Also, do not use bitmap graphics and image compression formats (`gif`, `jpg`, `png`), in particular, scanned pictures and digital photos.

4.3.1 Postscript Files

Many graphics utilities, Maple and Gnuplot for example, generate *Postscript* files, which are easily recognized by the `.ps` or `.eps` extension. Postscript is a popular graphical format introduced by Adobe, Inc. It belongs to the category of *vector* graphics, as opposed to *bitmap* graphics, a typical representative of which is the `bmp` format. Postscript is a parent (pretty much alive!) of the Portable Document Format (PDF) also designed by Adobe, Inc.

EPS stands for *Encapsulated Postscript*, which is now the best-supported format for the inclusion of graphics into LaTeX documents. To include an `.eps` file into a `.tex` file is easy.

1. First, you must have the line `\usepackage{graphicx}` in the preamble of you document (between the `\documentclass` and `\begin{document}`). Note the peculiar "x" at the end of the package name.

2. At the spot where you want to drop the `.eps` file into your document, use the command `\includegraphics`. For example, if the name of your graph is `fig1.eps`, the following line will include it into the LaTeX file:

   ```
   \includegraphics{fig1}
   ```

   The extension `.eps` in the file name should be omitted. Often the `\includegraphics` command is used within the *picture* environment or *figure* environment. See p. **??** and p. 114 for information about these environments.

3. Then proceed with your LaTeX file in the usual way.

For most students, the above algorithm of integration of EPS with LaTeX is all that is needed.

For all practical purposes, the only difference between `.ps` and `.eps` files is that the latter have a *BoundingBox* line, which is usually the second line in the file (but sometimes it is found at the very end). If your graphing program generates `.ps` file, but not an `.eps`, you need to **convert** it — see instructions on p. **??**. Unless you use raw Postscript programming or older versions of Gnuplot, you will likely never need this.

Optional arguments of `\includegraphics`

Consider a more sophisticated version of the above example:

```
\includegraphics[height=8cm, angle=90]{fig1}
```

The expressions `height=8cm` and `angle=90` are optional arguments to the `\includegraphics` command and, as with all optional arguments in LaTeX, they are included within square brackets. Any valid TeX unit of length can be used. Counter-clockwise direction of rotation is deemed positive and the angle is measured in degrees. Other possible optional arguments for `\includegraphics` are `totalheight`, `width`, and `origin`. The difference between `height` and `totalheight` is that the former specifies the elevation of the graphics above the baseline, while the latter equals to height plus depth (the part below baseline). The argument `origin` specifies what point to use for a rotation origin (`origin=c` rotates about the centre).

Postscript programming

Postscript (`.ps`, `.eps`) files are usually created by specialized graphing or more universal programs — like Gnuplot, XFig, or Maple. However, Postscript by itself is a human-readable language, and one can write a "program" in Postscript describing a picture. Also, you can generate a Postscript file automatically by your own FORTRAN or C program.

As an example, the following short program in raw Postscript (Fig. 4.1) describes two Pythagorean triangles, shown on the right. Type the program in a text editor and save the file as, say, `triangles.ps`. The unit length in Postscript is *point*, which is 1/72 of an inch.

```
%!PS-Adobe-2.0
% First triangle (contour)
newpath
200 125 moveto
300 125 lineto
200 200 lineto
200 125 lineto
stroke
% Second triangle (filled)
0.8 setgray
newpath
200 0   moveto
275 0   lineto
275 100 lineto
200 0   lineto
fill
showpage
```

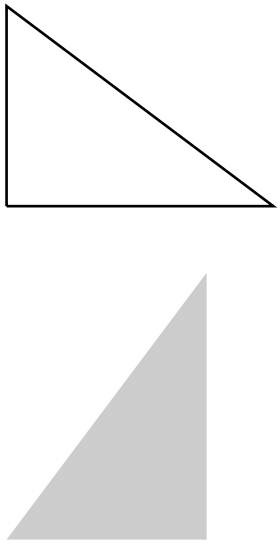


Figure 4.1: A simple Postscript program and its effect

You can immediately open the file `triangles.ps` with Postscript viewer Ghostview:

```
gv triangles.ps
```

You can now insert the picture to a LaTeX document. A conversion to `.eps` is required, but if you want a quick try, just replace the first line `%!PS-Adobe-2.0` in the file by two other lines:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 198 0 303 202
```

Save the file as `triangles.eps`. You can now type the line `\includegraphics{triangles}` in a LaTeX document and the picture will be inserted. To put the triangles beside the Postscript program on Figure 4.1, we used a superimposition trick described in Section 4.4.4.

4.3.2 Maple graphics

Maple has versatile plotting capabilities. They are realized by means of two basic commands: `plot`, `plot3d`, and more functions provided by the `plots` package. We will illustrate the usefulness of these commands with very basic examples. To get a more elaborate description of plots and their options, use Maple's Help and other available sources.

We begin with a graph of the function $y = \frac{\sin x}{x}$ on a specified interval $(-15, 15)$. The formula does not make sense when $x = 0$, but the limit of y as $x \to 0$ exists. Setting $y = 1$ when $x = 0$ makes our function continuous everywhere. Maple knows such tricks as extension by continuity and it automatically determines the y-range necessary for the plot.

```
> plot(sin(x)/x,x=-15..15);
```

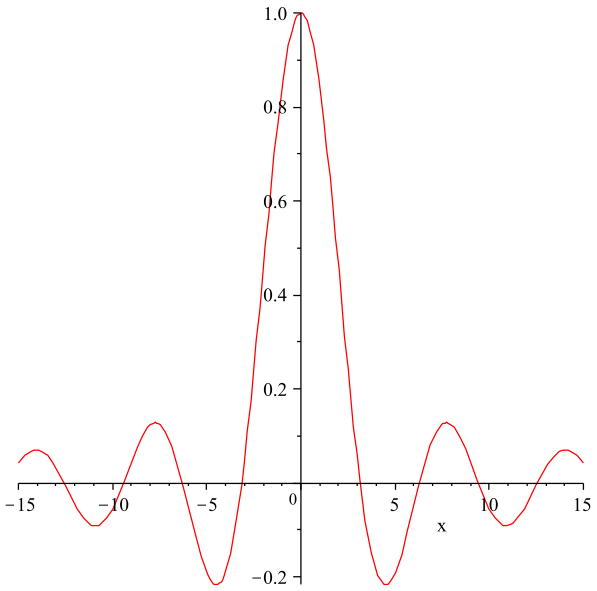


Figure 4.2: Maple's graph of smooth function: $y = \dfrac{\sin x}{x}$

Let's try a graph with vertical asymptotes. Consider the function `y2` on p. 89, which can be factorized as $(x+3)(x-3)^2/(2(x-1)(x-2))$.

```
> plot(y2,x=-10..10);
```

The graph, Figure 4.3, is not looking particularly illuminating. The vertical range can be altered with this graph to enable a clearer picture (Figure 4.4, left):

```
> plot(y2,x=-10..10,y=-40..40);
```

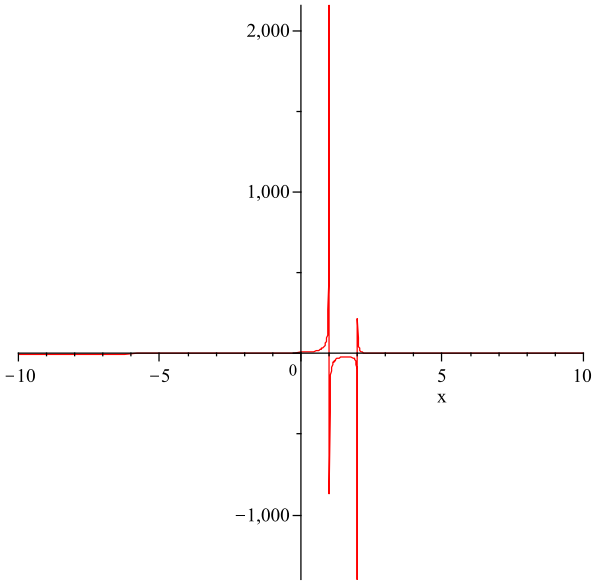


Figure 4.3: Maple's plot of $y = \dfrac{(x+3)(x-3)^2}{2(x-1)(x-2)}$. Default view.

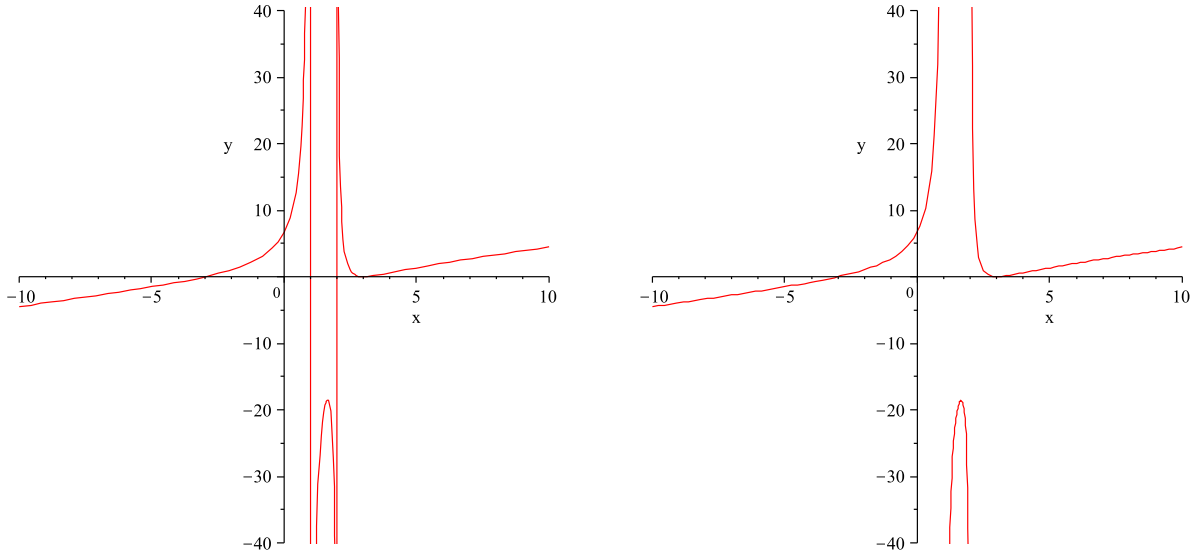


Figure 4.4: Plot with restricted y-range: discontinuity exhibited (left) or hidden (right).

A discontinuity detector can be used to remove the unnecessary lines, as on Figure 4.4, right:

```
> plot(y2,x=-10..10,y=-40..40,discont=true);
```

Coordinates can be plotted using the following format, yielding Figure 4.5.

```
> X:=[[-2, 4],[-1,1],[-1/2,1/4],[0,0],[1/2,1/4],[1,1],[2,4]];
```

$$\mathtt{X} := \left[[-2,4],[-1,1],\left[-\frac{1}{2},\frac{1}{4}\right],[0,0],\left[\frac{1}{2},\frac{1}{4}\right],[1,1],[2,4]\right]$$

```
> plot(X);
```

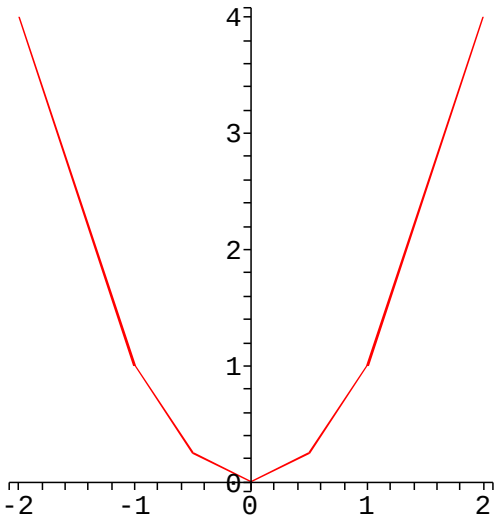


Figure 4.5: Maple's graphs defined by an array of coordinate pairs

Maple can plot parametric curves. In the following example we equalize the x and y scales using the option `scaling=constrained`. Without this option the graph, which is an ellipse (Fig. 4.6), would look like a circle.

```
> plot([5/2*cos(t), 5/3*sin(t), t=0..2*Pi],scaling=constrained);
```

Attention! A slight syntactic alteration of the command — moving the bracket `']'` — completely changes the picture: instead of a parametric curve we obtain two curves on the same graph (Fig. 4.7), with t being the independent variable (instead of being a parameter). The option `scaling=constrained` is unimportant in this example and omitted.

```
> plot([5/2*cos(t), 5/3*sin(t)], t=0..2*Pi);
```

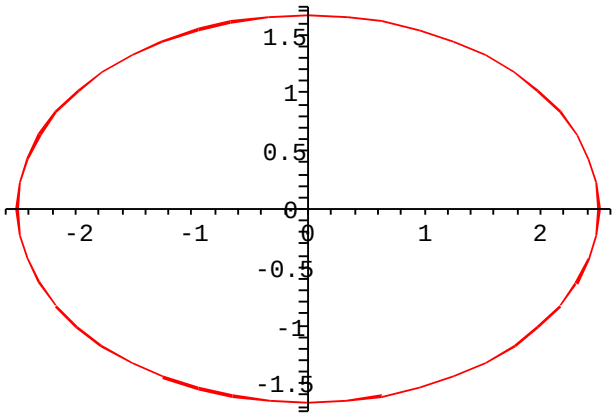


Figure 4.6: An ellipse described parametrically: $x = \frac{5}{2}\cos t,\ y = \frac{5}{3}\sin t.$

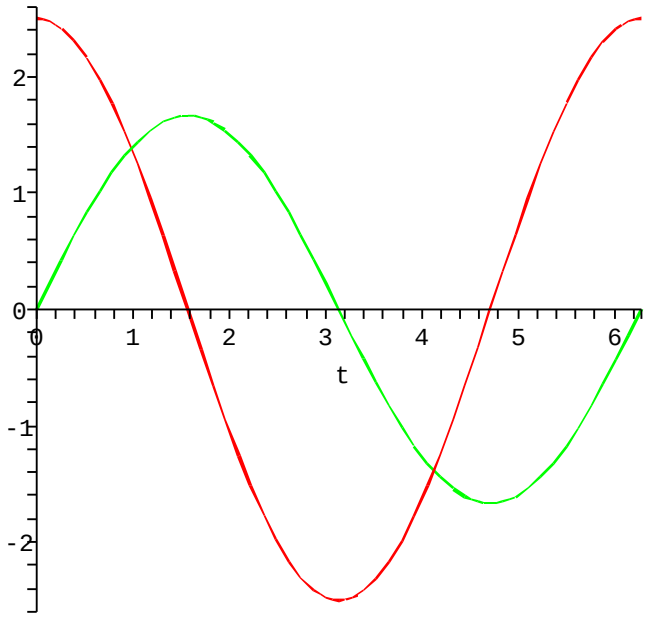


Figure 4.7: Curves $y = \frac{5}{2}\cos t$ and $y = \frac{5}{3}\sin t$ in the (t, y) axes.

Here is an example of a 3D graph. (Compare Maple's graph with Gnuplot's — Fig. 4.12(D).)

```
> F:=sin(x^2+y^2)/(x^2+y^2);
```

$$F := \frac{\sin(x^2 + y^2)}{x^2 + y^2}$$

```
> plot3d(F,x=-3..3,y=-3..3);
```

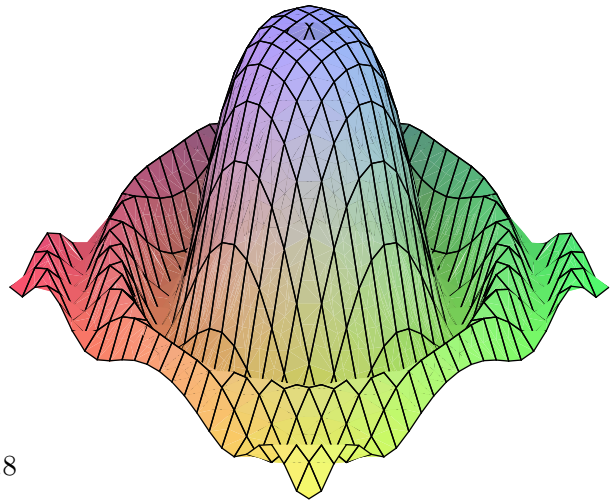


Figure 4.8

The default is not to show the axes. However, they can be added:

```
> plot3d(F, x=-3..3, y=-3..3, axes=boxed);
```

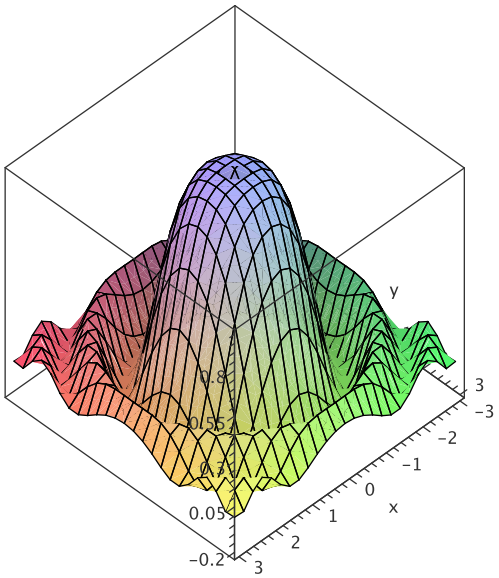


Figure 4.9

For more advanced tasks, we can load the plotting library, `plots`.

```
 >with(plots);
[Interactive, animate, animate3d, animatecurve, arrow, changecoords,
  complexplot, complexplot3d, conformal, conformal3d, contourplot,
  contourplot3d, coordplot, coordplot3d, cylinderplot, densityplot, display,
  display3d, fieldplot, fieldplot3d, gradplot, gradplot3d, graphplot3d,
  implicitplot, implicitplot3d, inequal, interactive, interactiveparams,
  listcontplot, listcontplot3d, listdensityplot, listplot, listplot3d,
  loglogplot, logplot, matrixplot, multiple, odeplot, pareto, plotcompare,
  pointplot, pointplot3d, polarplot, polygonplot, polygonplot3d,
  polyhedra_supported, polyhedraplot, replot, rootlocus, semilogplot,
  setoptions, setoptions3d, spacecurve, sparsematrixplot, sphereplot,
  surfdata, textplot, textplot3d, tubeplot]
```

Plotting the graph of an implicit equation $f(x, y) = 0$ now becomes possible. Try

```
> f:=4*x^2+9*y^2-25;
> implicitplot(f, x=-4..4, y=-2..2, scaling=constrained);
```

The result is identical to Figure 4.6. It should not be surprising: the parametrized coordinates $x = \frac{5}{2}\cos t$ and $y = \frac{5}{3}\sin t$ satisfy exactly our present equation $4x^2 + 9y^2 = 25$. Note however that plotting functions implicitly is more difficult for Maple than plotting parametric curves with a command like that on p. 101. The quality of parametric plots will generally be better.

The library `plots` also makes available plotting multiple graphs on the same picture. Each graph (plot) can be created separately; then all of them are submitted at once to the `display` command. When creating the individual plots, use *colon* as a terminator; otherwise Maple will spit out the long and nasty internal representation of a plot. Example: the following commands produce the plot identical to Figure 4.7 save for a label on the horizontal axis.

```
 > sinplot:=plot(5/3*sin(x), x=0..2*PI, color=green):
 > cosplot:=plot(5/2*cos(x), x=0..2*PI, color=red):
 > display([sinplot, cosplot]);
```

While in this case the two curves can be conveniently plotted together using just the regular `plot` command as on p. 101, it is easily conceivable that individual plots can be too many or too complicated, so that computing them separately and then using the `display` command can be the only practical option. The following example, showing an equilateral triangle together with its inscribed and circumscribed circles (Fig. 4.10) would be too cumbersome to describe by a single `plot` command.

```
> triangle:=plot([[1,0],[-1/2, sqrt(3)/2],[-1/2,-sqrt(3)/2],[1,0]], color=red):
> circumcircle:=plot([cos(t),sin(t),t=0..2*Pi],color=blue):
> incircle:=plot([cos(t)/2,sin(t)/2,t=0..2*Pi],color=green):
> display([triangle,circumcircle,incircle],scaling=constrained, axes=none);
```

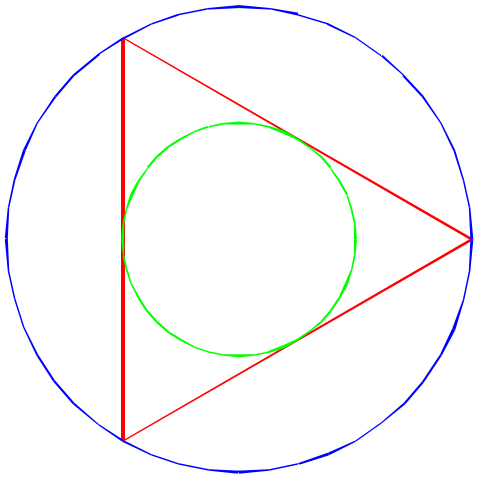


Figure 4.10: Equilateral triangle with inscribed and circumscribed circles

Our last example demonstrates a combination of Maple elements just described and those described in Section 4.2. We construct three similar parametric curves, called cycloids, each given by the parametric equations

$$x = C(0.3t - \sin t), \qquad y = C \cdot 0.6(1 - \cos t). \tag{4.1}$$

The parameter C equals 0.5, 1, and 2 for the three curves, respectively. To make the x-span of the curves approximately equal, we choose the t-range the bigger the smaller C is. Specifically, we set $t \in [-\pi, 9\pi]$ for $C = 0.5$, $t \in [-\pi, 5\pi]$ for $C = 1$, and $t \in [-\pi, 3\pi]$ for $C = 2$. The pattern is: $t_{\min} = -\pi$ and $t_{\max} = (1 + 4/C)\pi$.

Instead of creating plots of the three curves individually, we take advantage of the clear pattern and create a procedure. The first argument is the value C in Eq. (4.1), and the second argument is the color to strike the curve with.

```
> cycloidC:=proc(C,col) local t,tmax,fx,fy:
      fx:=0.3*t-sin(t):  fy:=0.6*(1-cos(t)):  tmax:=(1+4/C)*Pi:
      return plot([C*fx, C*fy, t=-Pi..tmax], color=col):
  end proc:
```

We then assign the colors:

```
> cycloid_colors:=[red,blue,black]:
```

In the final plotting command, we use the array manipulation methods from Section 4.2.4.

```
> display([seq(cycloidC(2^(n-2), cycloid_colors[n]),n=1..3)] );
```

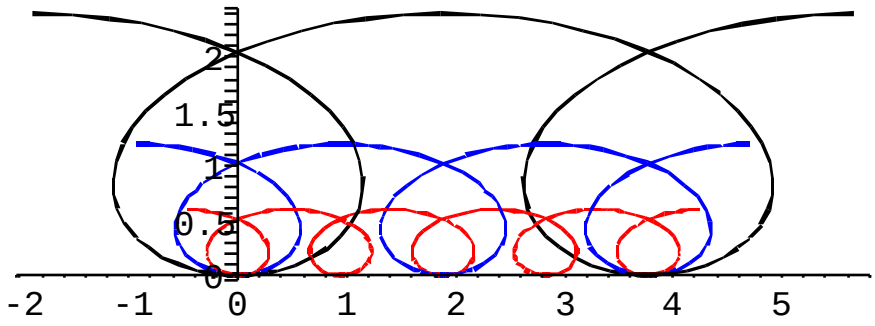


Figure 4.11: Cycloids given by Eq. 4.1 with $C = 0.5$ (red), $C = 1$ (blue), and $C = 2$ (black).

For comparison, here is an equivalent, more explicit Maple code for the same plot.

```
> a:=0.3:   fx:=a*t-sin(t):   fy:=a*(1-cos(t)):
> p0:=plot([0.5*fx,   fy, t=-Pi..9*Pi], scaling=constrained, color=red):
> p1:=plot([    fx, 2*fy, t=-Pi..5*Pi], scaling=constrained, color=blue):
> p2:=plot([  2*fx, 4*fy, t=-Pi..3*Pi], scaling=constrained, color=black):
> plots[display]({p0,p1,p2});
```

How to insert a Maple graph in a LaTeX document.

First, you have to save your graph as an Encapsulated Postscript file. The simplest method is to right-click on the graph you want to import and to *save image as EPS file*. Alternatively, especially if you have many pictures in your file, a convenient way to save all of them at once is to click on menu *File/ExportAs* and choose file type LaTeX.

If, say, the name of the Maple worksheet is `cycloids.mw`, then the name of the Maple-created LaTeX file is `cycloids.tex`, and the first plot will be saved as `cycloidsplot1.eps`, the second plot (if exists) — as `cycloidsplot2.eps`, and so on.

As advised on page 87, ignore the LaTeX file that Maple creates, but pick the companion `eps` files.

Once you have extracted an `eps` file or files from a Maple worksheet, import them in your master LaTeX file by means of the `\includegraphics` command.

As a final remark, we ask you **not** to include graphics **modified via the pop-up menu** that appears when you right-click on a picture. While this menu provides convenient manipulations, no record remains of the ultimate parameters. Hence the instructor may not be in position to tell how exactly your graph was generated based on the Maple worksheet you would submit electronically. Of course, if you fully document any manipulations, then these restriction will not apply.

4.3.3 Gnuplot

Gnuplot used to be thought of as a UNIX command-line utility, and it still is, but now it is also available for Windows, with a handy clickable menu. It is a freely distributed software.

Gnuplot can display graphs on a computer screen and save them as files of various graphics types. The kind of output display device is described by the parameter `terminal` of Gnuplot's `set` command.

The program is controlled by user's commands typed in the command line; several commands can be put together into a file to form a re-usable script.

Gnuplot can handle two-dimensional and three-dimensional data and graphs. It automatically changes line styles when several curves are plotted. It automatically produces a legend for each curve and permits the user to include such things as arrows and mathematical text anywhere on the graph.

Gnuplot is somewhat clumsy to use even with the simplest of data because of much typing necessary. In exchange, it provides a lot of flexibility in regard to line styles, labeling, and such, — perhaps, significantly more than Maple. It is also a very straightforward tool to plot graphs given as pairs of coordinates. On the other hand, Maple's versatility and symbolic manipulation power is incomparably higher.

The best way to use Gnuplot is to interactively piece together (step-by-step), vary and adjust all the essentials for the desired plot, all the while watching the plot displayed in its own window at every stage. Once the desired plot is generated and fine-tuned, it is easy to redirect the plot into a file.

To start a Gnuplot session, simply type `gnuplot` in the command line and the Gnuplot prompt will appear. A Gnuplot session is ended by typing `quit` at the prompt.

The two main plotting commands are `plot` (for 2D data) and `splot` (for 3D data). Their behaviour is controlled by a wide range of options. Most of the options are introduced by the command `set`.

Gnuplot's originally intended and still most popular use is to render plots based on externally prepared data fed to it from a file. Yet modern Gnuplot knows many standard mathematical functions and it is quite capable of plotting on its own, as sample graphs that follow demonstrate. Here we are only able to help you get started with Gnuplot. An interested reader can find out about numerous options and features in Gnuplot's Help and in online tutorials, for example, `http://www.ibm.com/developerworks/library/l-gnuplot`.

Plotting functions with Gnuplot

The figures on the next page demonstrate how Gnuplot can be used directly, without data importing. The commands used to generate each plot are given, but we are not attempting to explain the options involved. Look up a reference and play around with them to see, for instance, what effect the absence of the command `unset key` will produce or what will a different numerical value of `spacing`, `samples`, `isosamples`, etc. do.

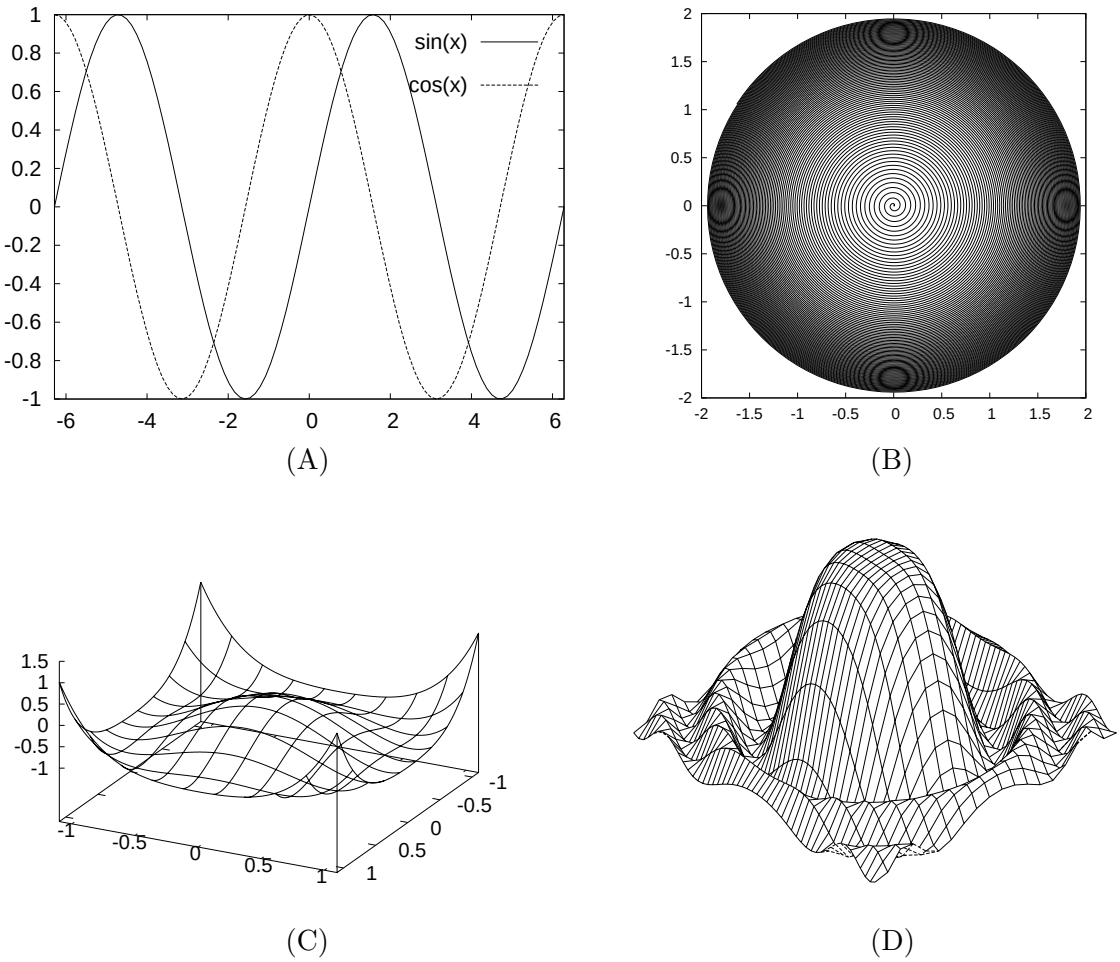


Figure 4.12: Graphs of functions produced by Gnuplot on its own

```
          (A)
set key spacing 2
plot [0:2*pi] sin(x),cos(x)
```

```
          (B)
unset key;  set parametric;
set trange [1:7]; set samples 10000
plot log(t)*cos(100*t), log(t)*sin(100*t)
```

```
          (C)
unset key; set ztics -0.5,0.5,1.5
set xrange [-1:1]
set yrange [-1:1]
splot (x**2+y**2)*(x**2+y**2-1.5)
```

```
          (D)
unset tics; unset border; unset key
set isosamples 50,20; set view 18,45,1,3
set xrange [-3:3]; set yrange [-3:3];
splot  (sin(x**2+y**2)/(x**2+y**2))
```

Notes: (B) represents a parametric curve — logarithmic spiral $\begin{bmatrix} x \\ y \end{bmatrix} = \ln t \begin{bmatrix} \cos(100t) \\ \sin(100t) \end{bmatrix}$.

(D) displays the same function as Figure 4.8 on p. 103.

Two-dimensional data plots

We will now use Gnuplot to plot data from a file supplied by the user. For two-dimensional plots, the contents of the file, by default, should be a collection of ordered pairs x, y separated by white space, one pair per line.

Let's say that you wish to plot two curves on the interval $x \in [-2, 2]$:

$$y = x^3 - 3x \quad \text{and} \quad y = \frac{1}{4}x^3.$$

- Choose the number of points to be used. Let's say, 100 points. Since the length of our interval is 4, the value of x will be incremented by 0.04 from one point to the next.
- Write a program to generate the data sets for both functions. For example, here is a fragment of a C program that does it for the first function.

  ```
  for (x=-2; x<=2; x=x+0.04){
    printf("%f %f\n", x, pow(x,3)-3*x);}
  ```

  (This code prints the data to the terminal, but you can re-direct the output to a file: type `a.out > plot1.dat` — see Section **??**).
- A data file named `plot1.dat` has been created with 100 data pairs. For example, the first and the last lines in the file will be like these:

  ```
  -2.000000 -2.000000
      ......
  2.000000 2.000000
  ```

  Modify the program and create the file `plot2.dat` for the second function similarly.
- Now the command

  ```
  gnuplot> plot 'plot1.dat' with lines, 'plot2.dat' with lines
  ```

  generates the plot displayed in Figure 4.13.

On your terminal, the curves will be drawn in like-style but with different colours. When a hard-copy is generated, `gnuplot` knows that colour is not generally available and thus differentiates between different curves on the basis of different line styles.

Displaying a surface in 3D space

Gnuplot accepts 3D data in the format similar to 2D: each line in the file represents the coordinates x, y, z of a single point on the surface. There is an option to remove hidden lines, so that foreground and background can be differentiated.

Figure 4.14 was produced from the file `glass.dat` (author: Gershon Elber, 1990), which is a part of the repository of samples provided with Gnuplot distributions. It can be downloaded separately from `http://www.challenge.nm.org/ctg/graphics/glass.dat`.

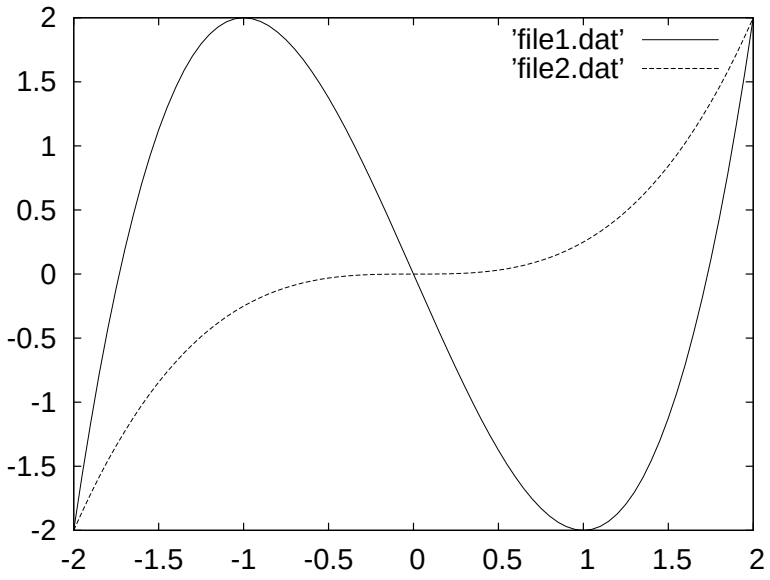


Figure 4.13: Graphs of $y = x^3 - 3x$ and $y = \frac{1}{4}x^3$ produced from data files.

The glass is a surface of revolution, which can be mathematically described by an equation of the form $r = f(z)$ in the cylindrical coordinates in $\mathbb{R}^3$, with $r = \sqrt{x^2 + y^2}$. The function $f(z)$ in this case is not given by a formula, but it is rather a result of an artwork.

Rendering three-dimensional surfaces is no simple business. If you venture to try, we recommend that all slices (level curves) have the same number of points per slice or level curve. The reason for this is that given a set of, say, x-slices, Gnuplot attempts to work out the missing y-slice data in order to perform the hidden line removal. Gnuplot will not necessarily fail if this advice is not followed, but its behaviour is then somewhat unpredictable.

The data file `glass.dat` contains $16 \times 16 = 256$ coordinate triples (x, y, z). There are 16 data blocks, each describing the level curve $z = z_i$ of the glass (that is, the circle with radius $f(z_i)$), for 16 different (not equally spaced) values z_i in the range from $z_1 = -0.911$ to $z_16 = 1.101$. Each block, in its turn, consists of 16 points, equally distributed along the level circle, with angular separation $360°/16$.

Here are the commands used to obtain figure 4.14, and a brief explanation.

`unset key`	Do not show legend
`set hidden`	Do not show hidden lines
`set xtics -0.5 0.5 0.5`	Set ticks on x axis starting from -0.5, with step 0.5, ending at 0.5
`set ytics -0.5 0.5 0.5`	
`set xyplane 0`	Adjust the position of the base xy plane
`set border 4095`	Draw a box around the plot
`splot 'glass.dat' using 2:1:3 with lines`	2:1:3 means interchange x and y data

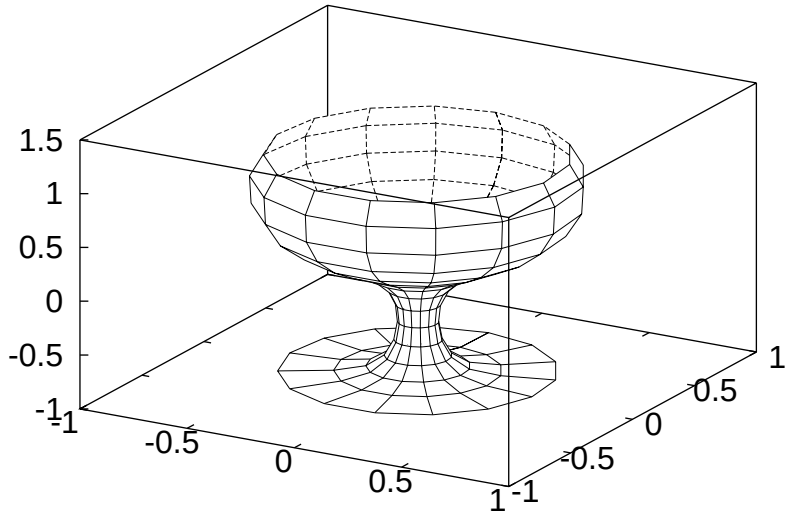


Figure 4.14: Plot produced from file `glass.dat`, with hidden lines removed.

The order in which the columns in the data file are used was changed with the "`using 2:1:3`" option in an effort to get a solid line to represent the forefront portion of the glass. (Otherwise the dashed line appears on the front somehow.) This is a harmless move in this case thanks to the rotational symmetry of the glass.

Generating Postscript with Gnuplot

Once, in the course of a Gnuplot session, you have a complete graph exactly as you want it, then simply issue the commands

```
> set terminal postscript eps
> set output 'figure1.eps'
> replot
```

The first of these (abbreviations `term` and `post` can be used) tells Gnuplot you wish to produce data in Encapsulated Postscript format, the second redirects the output into a file (in this case named `figure1.eps`), and the last command just redraws the plot last displayed into the file.

The file `figure1.eps` has now been created in your working directory. You can view it with Ghostview or insert it in your LaTeX document as descrined in Section 4.3.1. A scaling option (`height` or `width` in `\includegraphics`) may be helpful. (If you `set term post` without the `eps` option, then also conversion `ps` to `eps` and rotation by 270° will be required.)

The size of the picture as it will appear in your paper will often be smaller than what you see when previewing the generated file in Ghostview. Have mercy on your instructor's eyes, **use larger font size in labels.** This can be done as follows:

```
> set term post eps "Helvetica" 24
```

4.3.4 Using XFig to make diagrams

The program XFig is a handy tool that allows you to quickly draw diagrams consisting of simple shapes, to drag and drop objects, and even to put labels and formulas on your figures in a LaTeX format. It is great for creating **"hand-drawn" diagrams**, but less suitable when it comes to graphs where the positions of objects must be specified by coordinates. In XFig, you build a diagram **by mouse manipulations** and immediately see the result. Thus, as we see, the creation of XFig diagrams is not relied on a digital algorithm (although the resulting file is a digital description of the figure). As such, XFig should not be used in the cases where you must generate definite, unambiguously reproducible graphics. It can be used for auxiliary, artwork-like illustrations or schemes.

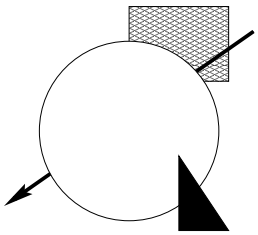
To start XFig, just enter the command

```
xfig
```

The XFig window has several icons along the left, several menu options along the top, including Help, rulers along the top and right, as well as a few setting boxes on the bottom. Playing with the icons on the left and the bottom is perhaps the best way to discover what you can do. The "scull and bones" button allows you to delete previously created objects. Once you get going, a few of the setting boxes on the bottom are worth noting, such as the *Point Posn*, *Depth*, and *Fill Style*.

Notice that the functionality of each of the 3 mouse buttons varies from option to option, and is displayed in the upper right corner of the window.

For example, the following diagram contains 4 objects that were each drawn with a different *depth* setting.

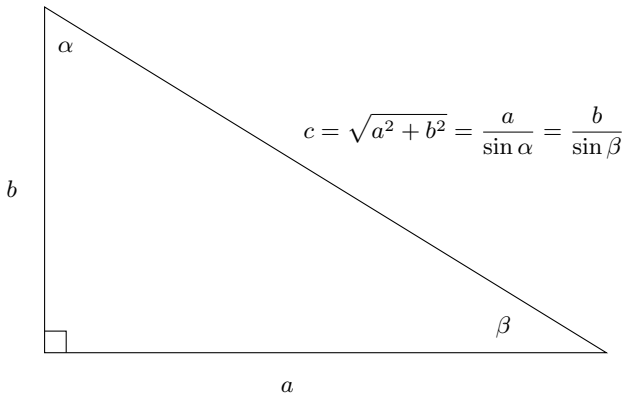


Once you've got a diagram ready for inclusion in a LaTeX document, first use the *File* menu option and save the figure in a file with a `.fig` extension, such as `diagram.fig`. Then use the *Export* menu option to export the figure using the *Encapsulated Postscript* language. Notice the default here will be to create a file named `diagram.eps`, which you can subsequently include in your document by following the instructions in Section 4.3.1.

You will likely want to label your diagrams on occasion, sometimes even using the math mode of LaTeX to display mathematical symbols. For this, you ought to start XFig with the command:

```
xfig -P -specialtext -latexfonts
```

Using the *text* drawing tool in XFig, you can then label your figures as you please, even including math mode text (which you can accomplish by using dollar signs). While XFig will not display your math mode labels very nicely, they will come out just fine in your document, such as in the following figure:



However, when using math mode, you need to select the export option of *Combined PS/LaTeX* rather than *Encapsulated Postscript*. You also should use the magnification option of the XFig *Export* window in order to fine-tune the size of your figure as it appears in your document. This time, to include the diagram in your master LaTeX file, use a different LaTeX command:

```
\input diagram.pstex_t
```

In this import mechanism, only the intermediary file with extention `pstex_t` must be directly referenced from your LaTeX document; yet the actual figure is still an `eps` file (which in this case has extension `pstex`). Thus your **electronic submission** must include both the `.pstex_t` and `.pstex` files besides the main `.tex` file.

4.4 The LaTeX `picture` environment and enhancements

4.4.1 Introduction

The most straightforward way to draw diagrams in a LaTeXed document is to use the **picture** environment:

`\begin{picture}`(*xlen*,*ylen*)(*leftcornerx*,*leftcornery*)
`...`
`\end{picture}`

Here ***`xlen`*** and ***`ylen`*** are the ranges (from zero) of the x and y coordinates. The **optional** parameters, ***`leftcornerx`*** and ***`leftcornery`*** change the bottom left hand corner from the origin to the new coordinates. They may be omitted if the bottom left hand corner is the origin.

The picture environment essentially allows you to do only one thing — to `\put`:

`\put`(*xcoor*,*ycoor*)`{`*object*`}`

An *object* can be a LaTeX's graphics element (`\line`,`\circle`), or a text, formula, paragraph (`\parbox`), table, etc. It can also be the `\includegraphics` command referring to an `.eps` file to be imported.

The dimensions of the picture (in the environment's header) and the coordinates in `\put` commands are given just as numbers, without specyfying the unit of length. The default **unitlength** is 1 point ($= 1/72\,in = 0.35\,mm$). It is rather too small for practical purposes and for convenience it can be reset by a command like

```
\setlength{\unitlength}{1cm}
```

printed just before the beginning of the picture environment.

LaTeX has very limited and poor graphical faclities of its own. Its `\line` command cannot even create arbitrary lines. However, with **enhancements** provided by additional packages, LaTeX's picture environment becomes competitive in its ability to generate interesting quality graphics. From now till the last part of this chapter, Section 4.4.4, we will stay entirely within a LaTeX document. No other graphics formats or imports will be involved.

We will describe in some detail the enhancements offered by the `math2130.sty` package.

In particular, a necessity to decide which unit length to set for the given picture can be avoided by use of an enhanced picture environment called **scaledpicture**. It is used thus:

`\begin{scaledpicture}{percent}(xlen,ylen)[(`***`leftcornerx`***`,`***`leftcornery`***`)]`
`...`
`\end{scaledpicture}`

Here, `percent` is a number less than or equal to 100, where 100 gives a diagram that almost fills the entire text width, and any smaller number gives a diagram that spans a percentage of the corresponding 100% diagram.

In most respects, `scaledpicture` behaves like the standard LaTeX `picture` environment, but it has additional features to make it easier to produce diagrams quickly and easily.

4.4.2 Lines

A major restriction in the LaTeX picture environment lies with the available slopes of lines that can be drawn. These slopes are restricted to *rationals* of the form p/q, where p, q are coprime and less than or equal, in size, to 6. The slopes are written as ordered pairs, (p, q) to allow for vertical lines, given by $(0, 1)$ or $(0, -1)$, depending on the direction.

Another complication in the syntax of a line segment, which is given by:

```
\line(p,q){len}
```

is that `len` refers, not to the length on the line, but to its projection in the x–direction (unless, of course, it is vertical, when `len` means the vertical length).

Even worse, each line segment must be `put` somewhere with a `\put` command of the form:

```
\put(a,b){\line(p,q){len}}.
```

You can see that the simple process of drawing a straight line is quite complicated.

4.4.3 Enhanced Pictures

However, help is at hand with enhanced picture styles. Since you will have started almost every document for this course with

```
\documentclass{article}
\usepackage{2130}
```

it makes sense to learn a few new commands available in the **picture** environment.

The `\join` command

The most useful command is:

```
\join(x1,y1)(x2,y2)
```

This draws a straight line from `(x1,y1)` to `(x2,y2)`. Already you can see an advantage over the standard `\line` command. The restrictions on slope no longer apply. Even better, by using

```
\join(x1,y1)(x2,y2)(x3,y3)
```

we get a straight line from `(x1,y1)` to `(x2,y2)` followed by a straight line from `(x2,y2)` to `(x3,y3)`. By doing this successively, with the points sufficiently close together, we can draw curves. So, you must first calculate the appropriate data set for the curve (see Sect. 4.1.3). The computed data set can be either pasted into the `.tex` file or, alternatively, it can be kept in its own file, say, `curve.tex` and the command

```
\input{curve}
```

can be used to import the data into your master LaTeX file.

Dotted lines

`\dottedline[`*`dotchar`*`]{dotgap}(x1,y1)(x2,y2)...(xn,yn)`

This draws a dotted line joining `(x1,y1)` to `(x2,y2)` and so on, to `(xn,yn)`. The `dotgap` is given in the units equal to the `\unitlength` defined and needs not be an integer. The optional *`dotchar`* may be omitted to give the default of a small dot, but any character may be used.

You can use this, with the appropriate `dotgap`, as a method for putting markers on curves.

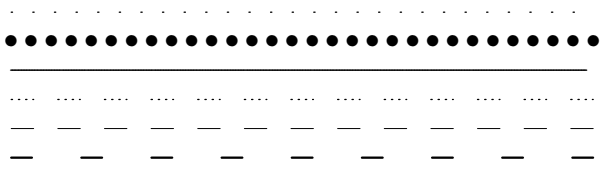
Dashed lines

`\dashline[`*`stretch`*`{dashlen}[`*`dotgap for dash`*`](x1,y1)(x2,y2)...(xn,yn)`

This draws a dashed line joining `(x1,y1)` to `(x2,y2)` and so on, to `(xn,yn)`. The `dashlen` is the length of the dash. Each dash is in fact a dotted line, and the optional *`dotgap for dash`* is the gap between each dot that is used to construct the dash. Both are in current unitlengths. The optional *`stretch`* is an integer between -100 and $+\infty$. You should experiment with these to find the appropriate relationship among the parameters to suit your purpose.

Here are some examples created by:

```
\begin{center}\setlength{\unitlength}{1em}
\begin{picture}(20,7)
   \dottedline{.7}(0,6)(20,6)
   \dottedline[$\bullet$]{.7}(0,5)(20,5)\join(0,4)(20,4)
   \dashline{.8}[0.2](0,3)(20,3)
   \dashline{.8}(0,2)(20,2)
   \thicklines\dashline[-30]{.8}(0,1)(20,1)
\end{picture}\end{center}
```



Grids

Grids can be created in many ways using:

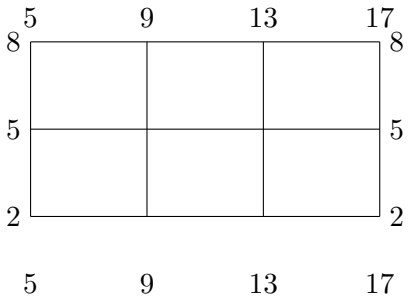
`\grid(xlen,ylen)(`Δ`xlen,`Δ`ylen)[init-x,init-y]`

This creates a grid measuring `xlen`×`ylen` with each `xlen` interval being Δ`xlen`, and each `ylen` interval being Δ`ylen`. The inputs of `init-x` and `init-y` give the coordinates of the bottom left hand corner of the grid.

The next diagram is generated within picture environment by the command

```
\grid(12,6)(4,3)[5,2]
```

(Picture dimensions are (12,6) and `unitlength` is set equal to 1em).



Circles

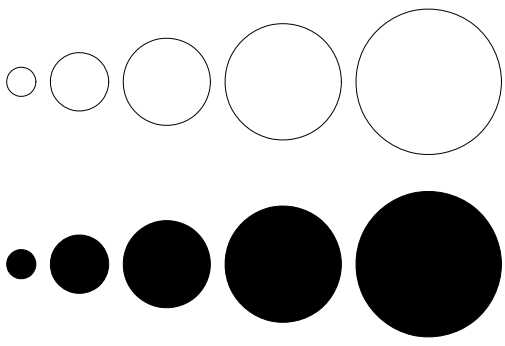
The standard LaTeX picture environment allows you to draw circles using the command:

```
\put(x,y){\circle{diam}}
```

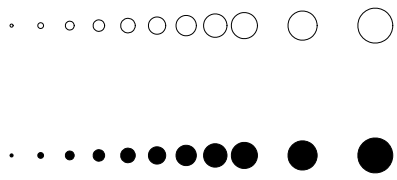
The parameter `diam` is the diameter of the circle (measured in `unitlength`), centered on `(x,y)`, and is an integer in the range $0 \leq$ `diam` ≤ 5. However, there is also a restriction on the maximum diameter of circle that can be drawn in absolute units: it cannot exceed 15 points (about 0.5 cm). In the enhanced picture enviroment, these restrictions no longer apply, although large circles will turn out to be rectangles with rounded corners.

There is a variation, `\circle*{diam}`, which gives a solid disk instead of a hollow circle. If you try to make a solid circle that is too large, LaTeX will not fill it in!

```
\setlength{\unitlength}{1em}
\begin{center}
\begin{picture}(24,5)
\put(2,3){\circle{0}}   \put(3,3){\circle{1}}   \put(5,3){\circle{2}}
\put(8,3){\circle{3}}  \put(12,3){\circle{4}}  \put(17,3){\circle{5}}
\end{picture}
\end{center}
```



```
\setlength{\unitlength}{0.1em}
\begin{center}
\begin{picture}(140,50)
\put(0,30){\circle{0}}  \put(10,30){\circle{1}}
\put(20,30){\circle{2}} \put(30,30){\circle{3}}
\put(40,30){\circle{4}} \put(50,30){\circle{5}}
\put(60,30){\circle{6}} \put(70,30){\circle{7}}
\put(80,30){\circle{8}} \put(90,30){\circle{9}}
\put(110,30){\circle{10}} \put(135,30){\circle{12}}
\end{picture}
\end{center}
```



In the **scaledpicture** environment, drawing circles is made easy, and consistently correct, with the command `\arc`. This command has three parameters, and is used thus:

`\put(a,b){\arc(p,q){deg}}`

Here, the centre of the arc is at `(a,b)`; the point where the arc begins is `(p,q)`, with this being taken **relative to the centre of the arc**; and the arc is drawn `deg` degrees in the positive (counter-clockwise) sense from the starting point. For example, `\put(0,1){\arc(2,0){360}}` will draw a circle of radius `2`, centered at `(0,1)`.

Labels

Labels can be placed on a diagram using the command `\put(x,y){label}`. A label is usually a letter or a number typed in math mode, like `$A$`, to produce a slanted A on the picture (for numbers, the math mode ensures that the negative sign will have an appropriate length). While easy in its syntax, this command requires some experience and judgement as to where one should "put" the label if, say, the label is referring to the point at the top right corner of a square. Try it for yourself. Try to label a simple square $ABCD$ and you will need to adjust the values of `x` and `y` several times before the labels finally look nice.

In the **scaledpicture** environment, this is made easy. There is one general command, and several short forms. The great advantage is that they refer to the point that is being labelled, say `(a,b)`.

The general command

`\angleput{deg}[scale](a,b){label}` — the default scale number is `1`.

Short forms

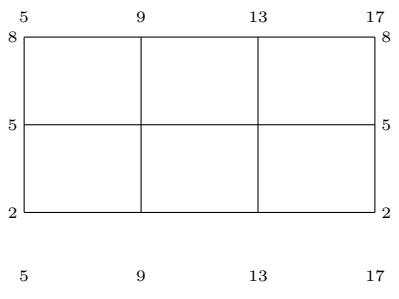
1. `\cput(a,b){label}` — `deg=0`, `scale=0` — centred on the point.

   In the next eight commands, `scale=1`.

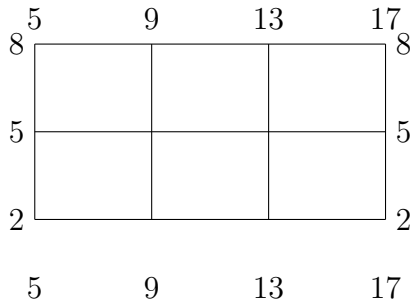
2. `\eput(a,b){label}` — `deg=0` — east of the point
3. `\nput(a,b){label}` — `deg=90` — north of the point
4. `\wput(a,b){label}` — `deg=180` — west of the point
5. `\sput(a,b){label}` — `deg=-90` — south of the point
6. `\neput(a,b){label}` — `deg=45` — northeast of the point
7. `\nwput(a,b){label}` — `deg=135` — northwest of the point
8. `\swput(a,b){label}` — `deg=-135` — southwest of the point
9. `\seput(a,b){label}` — `deg=-45` — southeast of the point

In the **scaledpicture** environment, the font size is chosen accordingly to the `percent` parameter. You may override this by the usual font sizing commands. Compare:

```
\begin{Scaledpicture}{36}(12,6)
\grid(12,6)(4,3)[5,2]
\end{scaledpicture}
```




```
\begin{Scaledpicture}{36}(12,6)
{\large \grid(12,6)(4,3)[5,2]}
\end{scaledpicture}
```



A `scaledpicture` is always centered. In fact, the environment used here was `Scaledpicture`, which produces an uncentred `scaledpicture`. This is useful for putting several diagrams on one line.

To further demostrate the use of labeling commands provided by `scaledpicture`, here is a set of commands for a cubic graph:

```
\begin{scaledpicture}{70}(8,6)(-4,-3)
\xaxis \yaxis \xnums{1} \ynums{1}
\ticks{1}[-0.1] \thicklines
\input{cubic_graph}
\put(-2.07936,0){\circle*{0.1}} \put(0.46295,0){\circle*{0.1}}
\put(3.1164,0){\circle*{0.1}} \put(2,-2.33333){\circle*{0.1}}
\put(-1,2.16667){\circle*{0.1}} \put(0,1){\circle*{0.1}}
\put(-4,-3.7){\large The graph of $f(x)=\frac13 x^3-\frac12 x^2-2x+1$}
\end{scaledpicture}
```

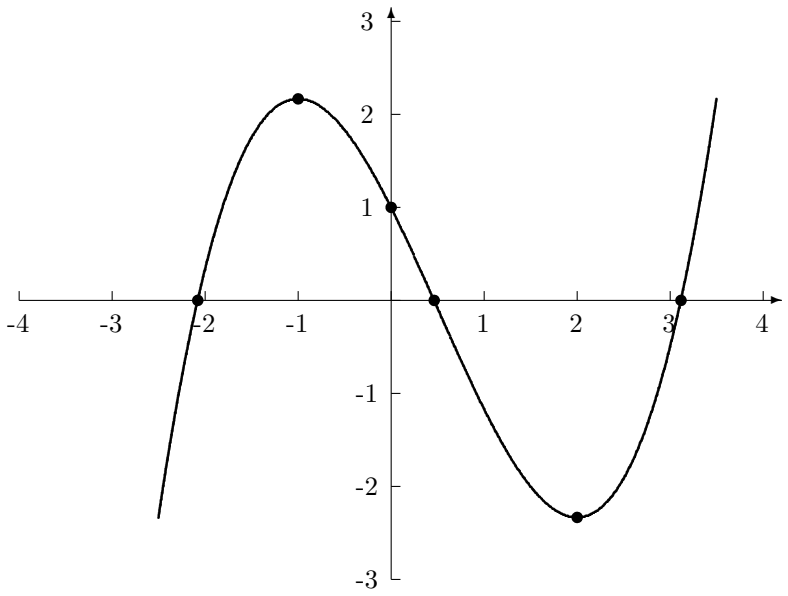


Figure 4.15: The graph of $f(x) = \frac{1}{3}x^3 - \frac{1}{2}x^2 - 2x + 1$

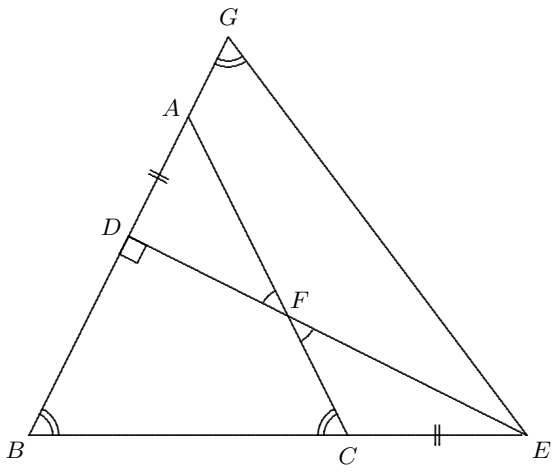
The `\input{cubic_graph}` command imports the contents of the file `cubic_graph.tex` whose first few lines are

```
\join(-2.50,-2.333)(-2.48,-2.200)(-2.46,-2.068)(-2.44,-1.939)(-2.42,-1.812)
(-2.400,-1.688)(-2.380,-1.566)(-2.360,-1.446)(-2.340,-1.329)(-2.320,-1.214)
(-2.300,-1.101)(-2.280,-0.990)(-2.260,-0.882)(-2.240,-0.775)(-2.220,-0.671)
(-2.200,-0.569)(-2.180,-0.470)(-2.160,-0.372)(-2.140,-0.277)(-2.120,-0.183)
(-2.100,-0.092)(-2.080,-0.003)(-2.060,0.084)(-2.040,0.169)(-2.020,0.252)
```

This is a typical example of a file that you can generate by your own program as described in Section 4.1.3.

Finally, here is the set of commands for the diagram that follows:

```
\begin{scaledpicture}{50}(13,12)(0,-1)
\join(0,0)(12.5,0)(5,10)(0,0) \join(12.5,0)(2.5,5) \join(4,8)(8,0)
\swput(0,0){$B$} \seput(12.5,0){$E$} \sput(8,0){$C$} \nput(5,10){$G$}
\angleput{153}[1](2.5,5){$D$} \angleput{153}[1](4,8){$A$}
\angleput{55}[1](6.5,3){$F$} \put(2.5,5){\rtangle{243}{.5}}
\put(10.25,0){\join(.05,-.25)(.05,.25)\join(-.05,-.25)(-.05,.25)}
\put(3.25,6.5){\rotate{63}{\join(.05,-.25)(.05,.25)
\join(-.05,-.25)(-.05,.25)}}
\put(6.5,3){\arc(.5,-.5){15}\arc(.5,-.5){-18} \arc(-.5,.5){15}
\arc(-.5,.5){-18}} \put(0,0){\arc(.6,0){64}\arc(.75,0){64}}
\put(8,0){\arc(-.6,0){-64}\arc(-.75,0){-64}} \put(5,10){\arc(0,-.6){34}
\arc(0,-.6){-27}\arc(0,-.75){34} \arc(0,-.75){-27}}
\end{scaledpicture}
```



Two commands here, `\rotate` and `\rtangle`, are defined in the `2130.sty` file using a bit of Postscript programming and by means of a command `special`. When the `special` command is used, the result may not always be dispalyed correctly. In this case, the `.dvi` picture may have some unrotated elements, but when a `.dvi` file is converted to `.pdf`, the correct rotation is put in place.

Some other picture commands in `2130.sty`

`\closecurve(a,b,c,d,e,f)` produces a triangle on the vertices (a, b), (c, d), and (e, f).

`\curve(a,b,c,d,e,f)` produces two segments joining (a, b) to (c, d) to (e, f).

`\rotate{deg}{object}` rotates the object through "deg" degrees. This needs Postscript.

`\rtangle{deg}{size}` is used in diagrams to produce a right angle marker. Needs Postscript.

4.4.4 Superimposition

The `picture` environment can be used to manipulate position of graphics and text by hand if needed. While this should not be considered as a good practice in general (Do not fight LaTeX! It knows better!), sometimes knowing how to fine-tune your document may help.

As an example, consider the layout of Figure 4.1 on page 98. On the left, we have the text of a program made within the `verbatim` environment, and on the right there is an `eps` picture of the two triangles inserted by means of the `incudegraphics` command. The question is how it is possible to make LaTeX to put the graphics in such a non-standard place. The key trick is the picture environment with a tiny height, which however enables precise positioning of any objects (graphics or text) via the coordinates in the `\put` command.

```
\begin{figure}[H]
\begin{verbatim}
    Text of the Postscript program
\end{verbatim}

\begin{picture}(400,1)
\put(300,30){\includegraphics{triangles}}
\end{picture}

\caption{A simple Postscript program and its effect}
\end{figure}
```

The main difficulty in such cases is determining the coordinates where to `\put` an object. It is, essentially, a trial and error business; the "convergence rate" of the process and precision with which you can drop the thing where you want it to be greatly improves as you gain experience.

Using superimposition within the master LaTeX document, it is possible to insert labels, on graphs created by various software tools. The advantage of this approach is that your labels will always be in the same font style as the rest of your document and their size will not depend on the scaling you apply to the imported graphics.

The pattern is simple:

```
\begin{picture}(...)
\put(0,0){\includegraphics{ your .eps file}}
\put(...){$ label$ or text}
\end{picture}
```

For the dimensions of the picture (in points), you can take the dimensions of the EPS graph, which can be calculated based on `BoundingBox` information. (The `%%BoundingBox` line can be found in most EPS files near the top of the file.) Suppose, for example, that the `BoundingBox` numbers are `50 60 410 302`. Then the horizontal size of the graph is $410 - 50 = 360$ and the vertical size is $302 - 60 = 242$. Thus you can use `\begin{picture}(360,242)`. Setting the picture dimensions precisely is not necessary and you can always make adjustments if you don't like the way the compiled LaTeX document looks.