Notes for MAT 331

Mathematical Problem Solving with Computers

Fall 1998

Santiago R. Simanca & Scott Sutherland
The University at Stony Brook

Contents

1 Preliminaries about Maple **7**

1.1 Logging in and opening a Maple session 8
1.2 Basic Maple 8
1.2.1 Maple for numeric calculations 8
1.2.2 Maple for symbolic manipulations 10
1.2.3 Graphing in Maple 13
1.2.4 On line Maple help 13
1.3 Assignments, Functions and Constants 14
1.3.1 Assignment statements 14
1.3.2 Variables and subsequent assignments: unassignments 15
1.3.3 Functions known to Maple. How to define your own functions 16
1.3.4 Special constants 17
1.3.5 Maple reserved names 18
1.4 The limit command 18
1.5 The diff (and Diff) command 20
1.5.1 Higher order derivatives 20
1.5.2 Implicit differentiation 21
1.6 The int (and Int) command 23
1.6.1 Numerical integration 24
1.6.2 Approximations through Riemann Sums 25
1.6.3 Multiple integrals 26
1.7 The subs command 27
1.7.1 Why is subs useful? 28
1.8 Plotting with Maple 29
1.8.1 Plotting several functions (or curves) on the same axes 31
1.8.2 Fancier plotting 32
1.8.3 Printing a graph 35
1.9 Working with and from the Worksheet 36
1.9.1 Saving a resuming work from a worksheet 37
1.9.2 Quitting a Maple session and logging out 37
1.10 Exercises 37

2 Least Square Fitting **39**
2.1 Fitting a line to data . . . 40
2.2 Fitting a cubic to data . . . 43
2.3 Fitting other types of funtions . . . 45
2.4 Fitting a circle . . . 46
2.5 Robust fitting . . . 48
2.6 A nod toward statistics . . . 52

3 fsqFsHn sGGousG **53**
3.1 Introduction to Cryptography . . . 53
3.2 Simple Ciphers . . . 54
3.2.1 Simple substitution . . . 54
3.2.2 The Caesar cipher, and the ASCII encoding . . . 55
3.3 Defining functions with `proc`; Local and global variables . . . 58
3.4 Caesar cipher redux . . . 60
3.5 Improved Caesar-like ciphers . . . 62
3.5.1 The Vignère cipher . . . 62
3.5.2 One-time pads . . . 64
3.5.3 Multi-character alphabets . . . 66
3.6 Reading and Writing from a file . . . 67
3.7 Affine enciphering . . . 68
3.7.1 When do affine encodings fail? . . . 68
3.7.2 Implementing and using an affine encoding . . . 69
3.7.3 Breaking an affine cipher . . . 70
3.8 Enciphering matrices . . . 70
3.8.1 Treating text as vectors . . . 70
3.8.2 Affine encoding with matrices . . . 71
3.8.3 A Known-plaintext attack on an affine matrix cipher . . . 73
3.9 Modern cryptography . . . 74
3.9.1 Secure cryptosystems . . . 74
3.9.2 Message digests . . . 75
3.9.3 Public Key cryptography . . . 76
3.10 Some Number Theory . . . 77
3.10.1 The greatest common divisor and the Euclidean algorithm . . . 77
3.10.2 The Chinese Remainder Theorem . . . 80
3.10.3 Powers modulo n . . . 81
3.10.4 The Euler φ-function and Euler's Theorem . . . 82
3.11 The RSA Public key cryptosystem . . . 84
3.11.1 Implementing RSA in Maple . . . 85
3.11.2 Useful preliminaries . . . 85

4 A turtle in a fractal garden **89**
4.1 Turtle Graphics 89
4.2 A fractal 91
4.3 Recursion and making a Koch Snowflake with Maple 93
4.3.1 Recursive functions. 94
4.3.2 A recursive procedure to generate K_n 95
4.3.3 The Koch Snowflake 95
4.3.4 Some variations on the Koch curve 96
4.4 Making a tree 97
4.5 Fractal Dimension 99
4.5.1 Box counting dimension 100
4.5.2 Computing the box dimension of some examples 102
4.6 Cantor sets 104
4.7 The Sierpinski gasket 106
4.8 Inside the turtle's shell 108
4.9 Extending the turtle's commands 108

Chapter 1

Preliminaries about Maple

Maple is a comprehensive general purpose computer algebra system which can do symbolic and numerical calculations and has facilities for 2 and 3 dimensional graphical output. The sequence of calculus courses maybe structured so that Maple can be used as a tool that will help you gain a complete understanding of the material discussed. You will discover that the capability of Maple goes beyond the realm of calculus; it is a tool that is used more and more in education and scientific research in mathematics and engineering.

This chapter is a brief introduction to the use of Maple[1] in the environment that you will discover at Stony Brook. It is not exhaustive and its sole purpose is to guide you through the first steps needed to start using this tool. Once you become familiar with the basics of Maple and can appreciate its advantages, we encourage you to experiment with it, not only to learn mathematics in more detail but also to help you in other engineering courses and your own research. More advanced documentation is available on line, in the computer lab, and in the library.

As a student at the University, you will be given a computer account. In order to use it, you will need to keep in mind your login name and your *password*, a secret code that is used to identify and validate your right to access the computer system. This introduction includes instructions on how to get into the system and open a Maple session.

Do not reveal your password to anybody. Choose a password which is easy to remember without writing it down. Select a mixture of upper and lower case letters, numeric and punctuation characters, and spaces between them. Avoid passwords which are words in the dictionary.

[1] We should point out that the material in this chapter was originally written with release 3 of Maple in mind. You may find a few small differences with the current release which we have tried to mark in the text. These differences are extremely minor, and should be pointed out by your instructor. The later chapters in these notes are all tested with release 5 of maple, which is current as of 1998.

1.1 Logging in and opening a Maple session

When looking at the screen of any terminal in the computer lab, you will see a `login name` window in the center of the screen:

- Type in your login name and press the `return` key.
- Type in your password and press the `return` key.

To *open a Maple session* procceed as follows:

- Position the cursor in the background (off the window's area) and hold down the left button of the mouse. A `Root Menu` will be displayed.
- While still holding down the left button of the mouse, move the cursor down the `Root Menu` until you position it on the `Tools` option, and then move it a little bit to the right until you see a submenu displayed.
- Keep the left button down and move the cursor along the `Tools` submenu until it is placed on the `Maple` option. Release the left button now.

After a few seconds, a Maple window will appear on the screen. It will show a standard Maple message with some information about the software you are about to use, Maple V Release 4.

The Maple window is ready to accept your commands when the Maple prompt `>` shows on it. We will illustrate the basic use of Maple through examples assuming that you are in this input mode. Maple will record all of your commands on a worksheet. Later on, we shall tell you how to save the worksheet in case you want to use its results on another occasion.

1.2 Basic Maple

You can use Maple for numerical calculations, for symbolic manipulations, and for graphing purposes. *All Maple commands must be ended with a semicolon* `;` *and the corresponding stroke of the return key.* To familiarize yourself with it, try to run the following examples and analyze the results. The commands that you type in and the Maple results are shown here in a way that is similar to what you see on the screen.

1.2.1 Maple for numeric calculations

Most of the time you will be using Maple as a calculator. You can also write programs in Maple, but usually you will be using the tool expecting an immediate response. You type in your statement at the Maple prompt `>` and Maple responds with an answer:

```
> 1/2+3;
                                  7/2
```

The symbols for the basic operations are +, -, * and /. Exponentiation is denoted by ^:

```
> 77*(3^5+5^2)/(99-47);
                                 5159
                                 ----
                                  13
```

In the first example, the inverse of 2 is added to 3. In the second, we input $\frac{77(3^5+5^2)}{99-47}$, and Maple computes its value before printing the output.

You must always use the asterisk * for multiplication. Forgetting it in expressions like `sin(2*x)` or `4*x` is easy, and will result in a syntax error message.

Maple works primarily with whole numbers or fractions. However, you can force it to write the decimal expansion of any number with `evalf`:

```
> evalf(100*Pi);
                              314.1592654

> b:=sqrt(2);
                                   1/2
                               b := 2

> evalf(b);
                              1.414213562
```

Be careful with this example. The *assignment* of the variable `b` to $\sqrt{(}2)$ is done with := rather than with =. This will be the case of any other assignment operation you do in Maple, as we discuss below.

Notice the difference in the following two examples:

```
> sum(1/(2*i),i=1..10);
                                 7381
                                 ----
                                 5040

> sum(1.0/(2*i),i=1..10);
                              1.464484127
```

In both cases, we compute $\sum_{i=1}^{10}(2i)^{-1}$. The factor `1.0` in the second case indicates to Maple that you want to obtain the decimal expansion of the result.

1.2.2 Maple for symbolic manipulations

We can ask Maple to calculate the sum of the cube of the first n integers:

```
> sum(i^3,i=1..n);
                                4              3              2
                    1/4 (n + 1)  - 1/2 (n + 1)  + 1/4 (n + 1)
```

We can ask to factor the resulting polynomial in n:

```
> factor(");
                                       2        2
                                  1/4 n  (n + 1)
```

In this example " is used[2] to refer to the result previously obtained. Thus, in effect, the example above is equivalent to

```
> factor(1/4*(n + 1)^4 - 1/2*(n + 1)^3 + 1/4*(n + 1)^2);

                                       2        2
                                  1/4 n  (n + 1)
```

One can expand or factor conveniently:

```
> p:=(x+1)^4*(x-6)^3;
                                          4        3
                              p := (x + 1)  (x - 6)

> expand(");
           7       6       5        4        3        2
          x  - 14 x  + 42 x  + 112 x  - 287 x  - 882 x  - 756 x - 216

> factor(");
                                        4        3
                                  (x + 1)  (x - 6)
```

We can solve equations:

[2] If you have Maple V release 5 or later, you should use % to refer to the previous result instead.

```
> quadratic:=a*x^2+b*x+c;
                                        2
                          quadratic := a x  + b x + c

> solve(quadratic=0,x);
                          2         1/2              2         1/2
                  - b + (b  - 4 a c)         - b - (b  - 4 a c)
              1/2 ---------------------, 1/2 ---------------------
                            a                          a
```

Notice that `solve` provides both roots of the quadratic polynomial.

Maple also divides polynomials. Here the command you execute has a syntax which is a little bit more complicated. Consider the following example:

```
> p:=x^10-1;
                                       10
                                  p := x   - 1

> q:=x-1;
                                   q := x - 1

> divide(p,q,b);
                                      true
```

We assign the polynomials $x^{10}-1$ and $x-1$ to the variables `p` and `q`, respectively, and then ask Maple to divide `p` by `q`. Maple replies with the statement `true` to indicate that the polynomial $q = x - 1$ divides exactly the polynomial $p = x^{10} - 1$, and places the quotient in the third argument of the `divide` command. Thus, if you want to see the quotient all that you need to do is to see what the variable `b` stands for now:

```
> b;
                   9    8    7    6    5    4    3    2
                  x  + x  + x  + x  + x  + x  + x  + x  + x + 1
```

Of course, the answer produced by `divide` will be `false` in the case when the polynomials do not divide each other exactly. In that situation, one can use the commands `quo` and `rem` to find the *quotient* and *remainder* of the division:

```
> r:=x^2+2*x+3;
                                        2
                                  r := x  + 2 x + 3
> divide(p,r,c);
                                      false
```

```
> quo(p,r,x);
            8      7    6      5       4       3       2
           x  - 2 x  + x  + 4 x  - 11 x  + 10 x  + 13 x  - 56 x + 73

> rem(p,r,x);
                                    - 220 + 22 x
```

We would have obtained an error if we would have tried `divide(p,r,b)`, trying to place the quotient of the division in the variable `b`: it is no longer a variable since it has a value from the previous example. Had we wanted to proceed that way, it would have been necessary to *unassign* the value of `b` first:

```
> b:='b';
                                        b := b
```

Further discussion on this is given in section 1.3.2.

We can use `solve` for systems of equations. For example:

```
> eq1 := 1*x+(1/2)*y+(1/6)*z =1;

                           eq1 := x + 1/2 y + 1/6 z = 1

> eq2 := (1/2)*x+(1/6)*y+(1/24)*z =0;

                         eq2 := 1/2 x + 1/6 y + 1/24 z = 1

> eq3 := (1/6)*x+(1/24)*y+(1/120)*z =1;

                       eq3 := 1/6 x + 1/24 y + 1/120 z = 1

> solve({eq1,eq2,eq3},{x,y,z});

                           {x = 39, y = -216, z = 420}
```

Maple also solves inequalities:

```
> solve(x^2-4*x-7>0,x);
                                     1/2           1/2
                        {x < 2 - 11   }, {2 + 11    < x}

> solve(abs(1-x^2)<1/2,x);
                 1/2                  1/2         1/2                1/2
         {- 1/2 6    < x, x < - 1/2 2   }, {1/2 2    < x, x < 1/2 6   }
```

1.2.3 Graphing in Maple

Let us now use the graphics facilities of Maple:

```
> f:=x->sin(x^2);
```

$$f := x \to \sin(x^2)$$

```
> plot({f(x)}, x=-3..3);
```

A new window is created displaying the graph of the function $sin(x^2)$ as indicated in Fig. 1.

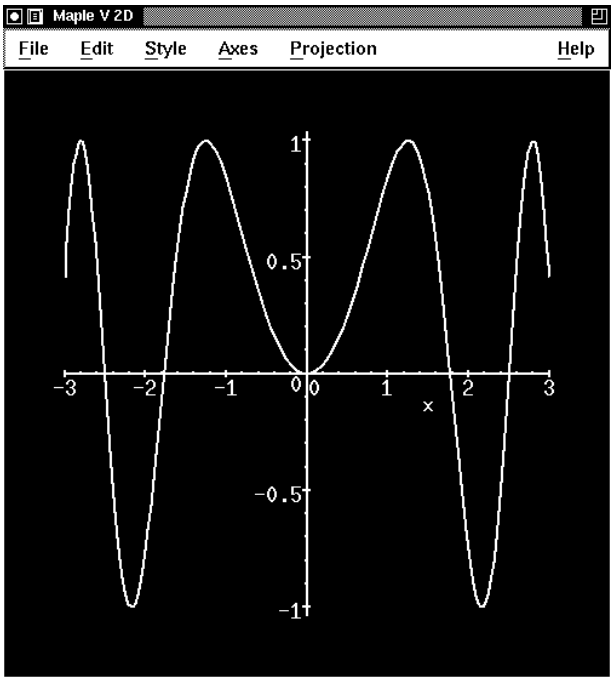


Figure 1.2.1: Window created by Maple with the plot of $sin(x^2)$ on $-3 \leq x \leq 3$

Notice the menu on top of the window displaying the graph. Clicking on `File` and selecting the `Exit` option from it, or pressing the `q` key while the cursor is on this window, makes the window go away.

In section 1.8 we shall discuss in further detail the graphing features of Maple.

1.2.4 On line Maple help

There are two kinds of help available on line in Maple. The first is useful if you know the name of the command you need to use and want to find out details on its syntax and how to use it properly. For example, to get information on Maple's `factor` command we execute

```
> ?factor;
```

Maple creates another window that very likely will contain more information than you really need or care about. You can read all or part of the information displayed in this window by moving the text down with the `scroll bar` to the right of the window. To do this, place the cursor on the `scroll bar` and move the mouse down (or up).

You should notice that at the end of all of this help windows you will find a good set of examples on the uses of the command you are inquiring about. This examples can be copied and pasted in your worksheet and executed. In this way, you can see what the command really does.

Even if the word you input is not quite a Maple command but something close, Maple will suggest relevant alternatives to try:

```
>?plotting;
Try one of the following topics:
           {plotsetup, plot, plot3d, plots, plotdevice, PLOT, PLOT3D}
```

This is an excellent way to learn about new commands.

The other way to find help about Maple is to use the `Help Browser`. Click on the `help` menu located on the right side of the top of the Maple window, and select from it the browser option. Try experimenting with it and with the other options available in this `help` menu.

1.3 Assignments, Functions and Constants

Let us discuss the syntax of some basic Maple commands, as well as some functions and constants that Maple knows.

1.3.1 Assignment statements

The assignment is the most basic and used of Maple statements. It allows you to give names to expressions you want the computer to remember for later use. It has the *colon-equals* format `name := expression`. The thing on the left side of `:=` is a *name*, i.e. a sequence of letters and digits that begins with a letter. Certain names must be avoided because Maple uses them for special purposes (see section 1.3.5):

```
> a:=3; b:=5; c:=22/7;
                                        a := 3

                                        b := 5

                                      c := 22/7
```

Now you can use a, b and c as though they are numbers. Thus, we have

```
> a+b;
                                  8
> c*a;
                                66/7
> a-b^2;
                                -22
```

1.3.2 Variables and subsequent assignments: unassignments

If you type a variable name and a semicolon, Maple will tell you what the variable stands for:

```
> c;
                                22/7
> x;
                                 x
```

If you specify y as a function of x and then assign to x a particular value, this value of x gets substituted into y:

```
> y:=sqrt(x+3);
                                      1/2
                           y := (x + 3)
> x:=sin(w);
                             x := sin(w)
> y;
                                      1/2
                           (sin(w) + 3)
```

If x gets changed, then y will also change. Maple remembers the original definition of y:

```
> x:=cos(t);
                             x := cos(t)
> y;
                                      1/2
                           (cos(t) + 3)
```

Notice that Maple evaluates x before telling you what the variable y stands for now. To undo this kind of thing, you can *unassign* variables. To do this for x execute:

```
> x:='x';
                                x := x
> y;
                                     1/2
                              (x + 3)
```

1.3.3 Functions known to Maple. How to define your own functions

Maple has all the standard mathematical functions in its library. You can make use of them by their appropriate name. We list some of these.

- Trig and inverse trig functions: `sin`, `cos`, `tan`, `cot`, `sec`, `csc`, `arcsin`, `arccos`, `arctan`, `arccot`, `arcsec`, `arccsc`.
- Exponential and logarithm: Maple uses `exp(x)` for e^x and either `log(x)` or `ln(x)` for the natural logarithm $\ln x$. Logarithm in other bases are also available. For example, `log[10](x)` is $\log_{10} x$.
- Hyperbolic functions: `sinh`, `cosh`, etc.
- Square roots and absolute values: `sqrt(x)` and `abs(x)`.

Of course, you will find yourself in a situation where you would like to define your own functions. Suppose you are analyzing the expression $\sqrt{1+x^2}$, plugging several values into it and calculating the result. Maple does not have a built-in function to do this, but you can define one yourself:

```
> f:=x->sqrt(1+x^2);
```

$$f := x \to \sqrt{1 + x^2}$$

Carefully notice the syntax. `f` is the function written to the right side of `:=`, and sends the variable x to $\sqrt{1+x^2}$. The arrow `->` is made with the minus and greater keys.

After such an assignment, you can use the name `f` in the same way you use other names of functions known to Maple, e.g. `exp` or `sin`:

```
> f(0); f(1); f(-1); f(3*x+1);
```

$$1$$

$$2^{1/2}$$

$$2^{1/2}$$

$$(2 + 9\,x^2 + 6\,x)^{1/2}$$

We are so accustomed to the notation `f(x)` to refer to a function that we might accidentally use it in Maple.

Never try to define a function with an assignment like

```
> f(x):=sqrt(x-2);
                                        1/2
                              f(x) := (x-2)
```

Maple will let you do it, but the resulting `f(x)` is useless. You will not be able to substitute into it or do anything you normally do with functions.

Maple uses parentheses `( )` to group expressions together in a convenient form as in

```
> 2*(x+4)^3;
                                          3
                                 2 (x + 4)
```

or to *delimit the arguments of a function.* In handwritten mathematics, we sometimes cheat and write $\sin x$ instead of $\sin(x)$. In Maple, this is not permissible: you must always write `sin(x)`. Nested parentheses are permitted:

```
> f(x);
                                       2 1/2
                                (1 + x )
> sin(sqrt(x+2)+ln(abs(x)));
                                   1/2
                        sin((x + 2)    + ln(abs(x)))
```

1.3.4 Special constants

Maple knows several constants. To find them out, execute the command

```
> constants;
                  false, gamma, infinity, true, Catalan, E, Pi
```

The names of these constants stand for `false`, γ, ∞, `true`, `Catalan`, e and π, respectively.[3] Be careful! Maple is case sensitive. Therefore `pi` is not the same as `Pi`. For example,

```
> cos(Pi/4);
                                        1/2
                                   1/2 2
> cos(pi);
                                   cos(pi)
```

[3] As of release 5 of Maple V, `E` no longer represents the constant $e \approx 2.71828\ldots$. Use `exp(1)` instead.

1.3.5 Maple reserved names

There are many commands and special objects in Maple that have preassigned names. In creating your own expressions and setting the name of your own variables, you should never use these particular names. They are:

```
and    by     do        done   elif    else   end    fi    for     from
if     in     intersect local  minus   mod    not    od    option  options
or     proc   quit      read   save    stop   then   to    union   while
```

The following names can be used, but it is not a good idea to do so:

```
cos  E    equation  exp   I   list  ln   log   matrix  sin   string  tan  vector
```

If you would like to use a particular name but are not certain if Maple gives it a special meaning, execute the command

```
> ?name;
```

where `name` is the name you want to check. If you get a Maple help screen for the name, it is probably safer to avoid it. You should only insist on using the name when the Maple response to the command above is something as in the following case:

```
> ?weirdname;
  The argument passed to help was:   weirdname
  Sorry, unable to explain this topic.  See  ?index  for
  a list of topics which are available.
```

1.4 The limit command

The limit command is used to compute limits. Its syntax is basically

```
> limit(f,x=a);
```

where `f` is a Maple algebraic expression which (in general) depends on the variable `x` and `a` is the expression which `x` approaches. For example, to compute

$$\lim_{x\to 2} \frac{3x-6}{x^2-4},$$

we execute the Maple command

```
> limit((3*x-6)/(x^2-4),x=2);
                                3/4
```

We can also try $\lim_{x\to 0} \frac{1}{x}$:

```
> limit(1/x,x=0);
                                  undefined
```

In the following exercise, we define the slope of a straight line passing through the points (x_1, y_1), (x_2, y_2), use this function to find the slope $m(x)$ of the line passing through the points $(1, 1)$ and (x, x^2), and finally compute the limit of $m(x)$ as $x \to 1$:

```
> slope:=(x1,y1,x2,y2)->(y2-y1)/(x2-x1);
                                                 y2 - y1
                      slope := (x1,y1,x2,y2) -> -------
                                                 x2 - x1
> m:=x->slope(1,1,x,x^2);
                                                    2
                         m := x -> slope(1, 1, x, x )

> limit(m(x),x=1);
                                       2
```

Maple is sometimes unable to determine a limit or whether it exists. In such a case, it will return nothing after you execute the limit command.

Maple computes *two-sided* limits. For example, if you specify the argument `x=1` as in the previous example, Maple assumes you mean that `x` approaches `1` from either the right or the left through real values only (as opposed to complex ones). However, Maple can compute one sided and complex limits also:

```
> limit(1/x,x=0,right);
                                    infinity

> limit(1/x,x=0,left);
                                   - infinity

> limit(x*log(x),x=0,complex);
                                       0
```

You can also compute limits as $x \to \infty$ and limits of functions of more than one variable:

```
> limit(arctan(x),x=infinity);
                                     1/2 Pi

> limit(x/(x^2+y^2),{x=0,y=0});
                                  undefined
```

A common error is to try to compute the limit of a function `f(x)` when `x` has been previously given a value. If you find yourself in this situation, unassign the value of `x` executing the command `x:='x';`.

1.5 The diff (and Diff) command

The `diff` command is used to compute derivatives of Maple expressions. Its syntax is basically

```
diff(f,x);
```

where `f` is an algebraic expression and `x` is the variable with respect to which the derivative is taken. For example, to compute

$$\frac{d}{dx}\frac{3x-6}{x^2-4},$$

we execute the Maple command

```
> diff((3*x-6)/(x^2-4),x);
```

$$\frac{3}{x^2-4} - 2\,\frac{(3x-6)\,x}{(x^2-4)^2}$$

You must specify the variable with respect to which you wish to take the derivative because, almost all the time, there are constants and other parameters around:

```
> diff(exp(a*x),x);
```

$$a\,\exp(a\,x)$$

Maple knows theoretical results about derivatives. For example, even if the functions `g(x)` and `h(x)` are not defined, Maple will give you the derivative of their product:

```
> diff(h(x)*g(x),x);
```

$$\left(\frac{d}{dx}h(x)\right) g(x) + h(x)\left(\frac{d}{dx}g(x)\right)$$

1.5.1 Higher order derivatives

To compute higher order derivatives, you can of course iterate the above command:

```
> diff(diff(3*sin(x),x),x);
```

$$-3\sin(x)$$

Such a command first calculates the derivative with respect to `x` of the function `3*sin(x)` and then differentiates the result to obtain the second order derivative of the original function. The same result can be accomplished with either one of the following commands:

```
> diff(3*sin(x),x,x);
                                  - 3 sin(x)

> diff(3*sin(x),x$2);
                                  - 3 sin(x)
```

There is an obvious extension of this command when calculating more than two derivatives. You can also calculate partial derivatives of higher order:

```
> diff(ln(x)*exp(-2*x),x$3);
          exp(- 2 x)     exp(- 2 x)      exp(- 2 x)
        2 ---------- + 6 ---------- + 12 ---------- - 8 ln(x) exp(- 2 x)
               3              2               x
              x              x

> diff(exp(x)*y^2*sin(z),x,y,z);
                               2 exp(x) y cos(z)
```

1.5.2 Implicit differentiation

Maple knows how to take derivative of both sides of an equation, and, as indicated above, Maple can also take derivatives "theoretically" even if the definition of the functions involved are not specified. However, you must always indicate *explicitly* the dependence of the functions on their variables. So, if you are thinking of `y` as a function of `x`, you must write `y(x)` in your equation rather than just `y`.

Here is a typical implicit differentiation problem. Consider the equation $x^2y - 3y^3x = 0$. Find the slope of the graph of the curve defined by this equation at the point $(3, 1)$:

```
> eq:=x^2*y(x)-3*y(x)^3*x = 0;
                                 2              3
                          eq := x  y(x) - 3 y(x)  x = 0

> deq:=diff(eq,x);
                         2 /  d      \          2   /  d      \          3
     deq := 2 x y(x) + x  |---- y(x)| - 9 y(x)  x |---- y(x)| - 3 y(x)  = 0
                          \ dx      /             \ dx      /

> solve(deq,diff(y(x),x));
```

```
                                    3
                  2 x y(x) - 3 y(x)
                - ------------------
                     2          2
                    x  - 9 y(x)  x
```

```
> subs(y(x)=1,x=3,");
                                1/6
```

The first command defines the equation relating `x` and `y`, the second defines an equation `deq` giving the relation between `x`, `y(x)` and the derivative of `y` with respect to `x`. Using `solve`, we solve `deq` to find the derivative as a function of `x` and `y(x)`, and in the result we substitute `x=3` and `y=1` to obtain the desired slope (see section 1.7 to learn about the `subs` command).

Again, the major caveat with the Maple derivative command is to avoid calculating derivatives with respect to a variable that has been previously given a value. In such an instance, unassign the value of the variable first.

Occasionally, to make your worksheets easier to read, you may wish to have Maple display a derivative in standard mathematical notation without evaluating it. For this purpose there is an *inert* capitalized form of the `diff` command: `Diff`. The two forms `diff` and `Diff` are usually combined to produce meaningful sentences:

```
> Diff(exp(x)/(1-x),x);
```

$$\frac{\partial}{\partial x}\frac{e^x}{1-x}$$

```
> Diff(exp(x)/(1-x),x)=diff(exp(x)/(1-x),x);
```

$$\frac{\partial}{\partial x}\frac{e^x}{1-x} = \frac{e^x}{1-x} + \frac{e^x}{(1-x)^2}$$

This command is particularly useful to produce easy to read worksheets that involve partial derivatives:

```
> f:=x^3*exp(y)-sin(x*y);
                                  3
                      f := x  exp(y) - sin(x y)
```

```
> Diff(f,x,x,y)=diff(f,x,x,y);
```

$$\frac{\partial^3}{\partial y \partial x^2} x^3 e^y - \sin(xy) = 6xe^y + \cos(xy)xy^2 + 2\sin(xy)y$$

1.6 The int (and Int) command

The `int` command is used to compute both definite and indefinite integrals of Maple expressions. Its syntax is basically

```
int(f,x);
```

where `f` is an algebraic expression and `x` is the integration variable. For example, to compute

$$\int \frac{3x-6}{x^2-4}\,dx\,,$$

we execute the Maple command

```
> int((3*x-6)/(x^2-4),x);
                                3 ln(x + 2)
```

Notice that Maple does not provide the constant of integration. You will occasionally have to take this into account and provide your own constant.

You must specify the variable of integration. In expressions involving other parameters, Maple assumes that you want the integral of the expression as the variable you specify changes and that all other parameters in the expression represent constants:

```
> int(exp(a*x),x);
                                  exp(a x)
                                  --------
                                      a
```

To compute a definite integral, the *range* over which the integration variable moves must be specified:

```
> int(x^2*exp(x),x=0..2);
                                2 exp(2) - 2
```

The `int` command in Maple has a particular behavior in certain situations:

- If Maple cannot compute the integral in *closed form*, it will return it unevaluated:

```
> int(ln(sin(sqrt(x^12-5*x^7+50*x+2))),x);
                      /
                     |           12      7            1/2
                     |  ln(sin((x   - 5 x  + 50 x + 2)   )) dx
                     |
                    /
```

- Sometimes the integral cannot be evaluated in closed form in terms of *elementary* functions, but the answer has a special name in mathematical circles due to its importance in applications. For example,

```
> int(sin(2*x)/x,x);
                                    Si(2 x)
```

  Here `Si` is the *special name* of one of these functions which appear frequently in mathematical physics. To learn more about it, execute the command

```
> ?Si;
```

  which will bring up a window with information about the function `Si`. If Maple responds to an integral with one of these functions, it is quite likely that the integral cannot be evaluated in terms of elementary functions.

- Very complicated answers contain the `%n` notation:

```
> int(1/(x^8 +1),x);
                                 -----
                                  \
                                   )     _R ln(x + 8 _R)
                                  /
                                 -----
                                _R = %1

                                                      8
%1 :=                       RootOf(1 + 16777216 _Z )
```

  In this answer, the sum is taken over all roots R of the polynomial $1 + 16777216z^8$, and the summand is $R \log(x + 8R)$.

1.6.1 Numerical integration

With the `evalf` command, you can force Maple to apply a numerical approximation technique for definite integration:

```
> evalf(Int(sqrt(1+x^10),x=0..1));
                                  1.040899075
```

Notice that the command for integration is *capitalized*. This prevents Maple from attempting to evaluate the integral symbolically and then just `evalf`ing the answer.

1.6.2 Approximations through Riemann Sums

Maple also computes expressions that approximate a definite integral using rectangles, trapezoids, etc. It can also do approximations using Simpson's rule. The only difficulty is that these commands are contained in a separate library called `student` and you must load this library into computer memory before you can use it:

```
> with(student):
 [D, Int, Limit, Sum, changevar, combine, completesquare, distance, intercept,

     intparts, isolate, leftbox, leftsum, makeproc, maximize, middlesum,

     midpoint, minimize, powsubs, rightbox, rightsum, showtangent, simpson,

     slope, trapezoid, value]
```

Once loaded, you can play with it. For example, if you want to use the left sum approximation to an integral, the height of each rectangle is determined by the value of the function at the left side of each interval. You may specify the number of intervals you wish to use. If you do not, Maple will use four intervals by default:

```
> leftsum(x^4*ln(x),x=1..4,10);
```

$$\sum_{i=0}^{9} \frac{3}{10}\left(1+\frac{3}{10}i\right)^4 \ln\left(1+\frac{3}{10}i\right)$$

Maple can also draw pictures of the rectangles used to approximate the integral of a function on a given interval:

```
> leftbox(1/x,x=1..2,10);
```

and the resulting figure is displayed in Fig. 2.

The following is the result of using the Simpson's rule for the function $xsin(x^2)$ on the interval $1 \leq x \leq 5$. Since the number of intervals is not specified, Maple assumes four equal intervals by default:

```
> simpson(x*sin(x^2),x=1..5);
```

$$\sum_{i=0}^{2}$$

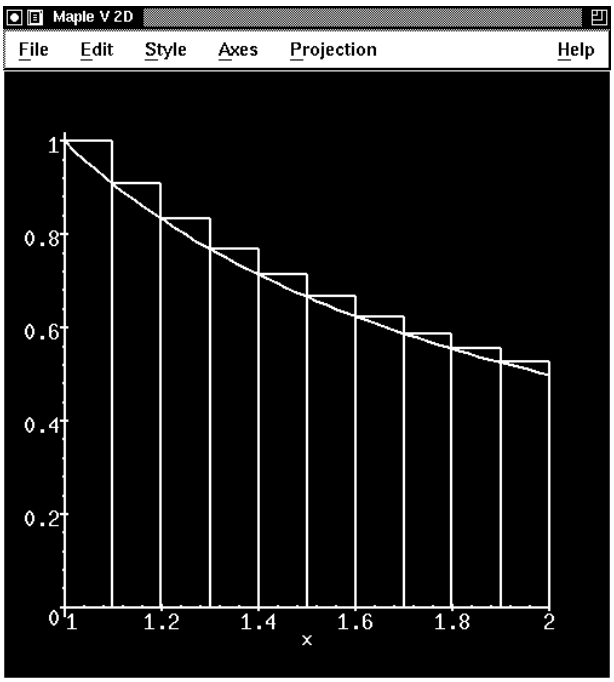


Figure 1.6.2: Left sum approximation of $\int_1^2 \frac{1}{x} dx$ with 10 rectangles

```
                                 |  )    2 i sin(4 i )|
1/3 sin(1) + 5/3 sin(25) + 4/3   | /                  |
                                 |-----               |
                                 \i = 1               /

          /  1                                \
          |-----                              |
          | \                              2  |
    + 2/3 |  )    (1 + 2 i) sin((1 + 2 i) )  |
          | /                                 |
          |-----                              |
          \i = 1                              /
```

1.6.3 Multiple integrals

Maple does iterated integrals:

```
> int(int(x^2*y^3,x=0..y),y=2..3);
```

$$\frac{2059}{21}$$

This is literally an iterated integral: the integral of x^2y^3 with respect to x on the interval $[0, y]$ is nested inside the integral with respect to y on the interval $[2, 3]$, in effect calculating

$$\int_2^3 \int_0^y x^2 y^3 dx dy \, .$$

As with `diff`, there is an *inert* form `Int` of the integral command which can be used in combination with `int` to produce easily readable worksheets:

```
> Int(ln(1+3*x), x=1..4);
```

$$\int_1^4 \ln(1+3x) dx$$

```
> Int(ln(1+3*x), x=1..4)=int(ln(1+3*x),x=1..4);
```

$$\int_1^4 \ln(1+3x) dx = \frac{13}{3}\ln(13) - 3 - \frac{4}{3}\ln(4)$$

A typical error is to try to integrate with respect to a variable that has been given a particular value you forgot about. Unassign the variable first.

1.7 The subs command

You have learned in the previous sections that if you assign a particular value to a variable, Maple will remember it for the rest of the session until you unassign it. At times, you might like to *plug-in* some values into an expression without altering the variable (or variables) involved. Maple has the `subs` command which reports the result of making such a substitution:

```
> y:=x+3;
```

```
y := x + 3
```

```
> subs(x=2,y);
```

```
5
```

Notice that Maple reports the value obtained when plugging in `x=2` into `y` without giving that value to `x` or altering `y` in any way:

```
> x;
```

```
x
```

```
> y;
```

```
x + 3
```

You can make more than one substitution:

```
> z:=x+sin(2*h):

> subs(x=3*u+1,h=4*Pi*q,z);
                              3 u + 1 + sin(8 Pi q)
```

In this example, the first assignment is ended with a colon : to suppress the display. You can do this whenever you want to make an assignment which you do not want to see displayed. Then, plugging $x = 3u + 1$ and $h = 4\pi q$ into the expression for z, we obtain $3u + 1 + \sin(8\pi q)$. Notice that the values of `x` and `h` that we plug in do not need to be enclosed in braces when issuing the command.

You should be aware of a few things when using the `subs` command:

- In certain situations, `subs` will not be able to find the expression you are substituting for if it is complicated. In those cases, try to *reword* your substitution request so that Maple *understands it*, or break the task into several manageable ones.

- You need not fear *circular substitutions*:

  ```
  > z;
                                      x + sin(2 h)

  > subs(x=2*x,z);
                                    2 x + sin(2 h)
  ```

  On the other hand, long chains of self-referential substitutions may produce results which are hard to predict:

  ```
  > subs(h=x,x=z,z);
                          x + sin(2 h) + sin(2 x + 2 sin(2 h))
  ```

  Simultaneous substitutions are made enclosing the values you plug in in braces. This is in contrast with the left-to-right substitution above:

  ```
  > subs({h=x,x=z},z);
                                 x + sin(2 h) + sin(2 x)
  ```

1.7.1 Why is subs useful?

Often it is better to use `subs` rather than to make assignments to the variables in an expression. For example, if

```
> y:=3*x^2+5*x+2;
```

$$y := 3x^2 + 5x + 2$$

and you want to compute the difference quotient $(y(x+h)-y(x))/h$, then you should execute

```
> dq:=(subs(x=x+h,y)-y)/h;
```

$$dq := \frac{3(x+h)^2 + 5h - 3x^2}{h}$$

because if we set

```
> x:=x+h;
```

```
Warning: Recursive definition of name
```

$$x := x + h$$

any call to `y` will result in

```
> y;
Error, STACK OVERFLOW
```

and quite likely, your Maple session will crash. On the other hand,

```
> x:=4;
```

$$x := 4$$

```
> y;
```

$$70$$

is allright, but now `y` is a number and we cannot plug any other values of `x` into it. `subs` is a better way to do the plugging in.

1.8 Plotting with Maple

Let us now use the graphics facilities of Maple:

```
> f:=x->x^2;
```

$$f := x \to x^2$$

```
> plot({f(x)}, x=-10..10);
```

```
>
```

A new window is created displaying the graph of the function x^2 from $x = -10$ to $x = 10$ as indicated in Fig. 3. Notice that the window has a menu which allows you to manipulate your graph or even get help with a click of the mouse. To close the window, place the cursor in the `File` menu and, holding down the left key of the mouse, select `close`.

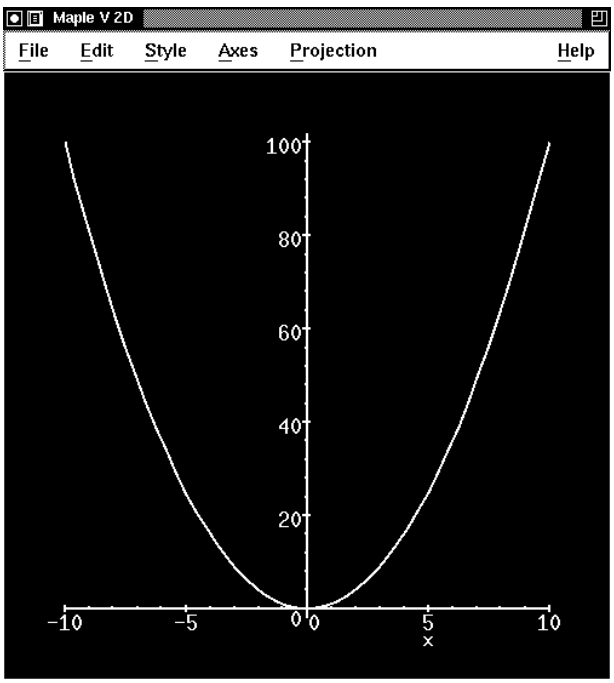


Figure 1.8.3: Window created by Maple with the plot of x^2 on $-10 \leq x \leq 10$

The basic syntax of the plot command is

```
> plot(f,range);
```

where `f` is the expression to be plotted and `range` is the range of the parameter(s) for which you would like to see the plot of `f`. In the example above, we indicated the range of `x` by `x=-10..10`. Maple automatically chooses a scale on the vertical axis. Observe what happens when executing the next two commands:

```
> plot(x^2-x,x=-2..2,y=-2..2);
```

```
> plot(x^2-x,x=-2..2,-2..2);
```

In both cases, the plot is displayed for values of `y` between -2 and 2, but in the example where the variable `y` is explicitly used in the command, the vertical axis has a label. The variable in the domain of the function *must always* be specified.

1.8.1 Plotting several functions (or curves) on the same axes

We can plot more than one function (or curve) on the same coordinate system. Consider the following example, recalling the slope function of the line passing through the points (x_1, y_1), (x_2, y_2), used to define the slope $m(x)$ of the line passing through $(1, 1)$ and (x, x^2):

```
> slope:=(x1,y1,x2,y2)->(y2-y1)/(x2-x1);
                                              y2 - y1
                      slope := (x1,y1,x2,y2) -> -------
                                              x2 - x1
> m:=x->slope(1,1,x,x^2);
                                                 2
                          m := x -> slope(1, 1, x, x )
```

We now define the equation of the lines passing through $(1, 1)$ and (x, x^2) for $x = 2, 3, 4, 5$ and plot the four lines together with the function `f(x)`. The result is shown in Fig. 4:

```
> line1:=subs(x=2,m(x))*(x-1)+1;
                                line1 := 3 x - 2

> line2:=subs(x=3,m(x))*(x-1)+1;
                                line2 := 4 x - 3

> line3:=subs(x=4,m(x))*(x-1)+1;
                                line3 := 5 x - 4

> line4:=subs(x=5,m(x))*(x-1)+1;
                                line4 := 6 x - 5

> plot({f(x),line1,line2,line3,line4},x=-3..6);

>
```

Observe that the expressions to be plotted are *enclosed* in braces `{ }`. Besides specifying the range on which `x` varies as done above, you can also specify and label the `y` range proceeding exactly in the same manner as when plotting a single function.

The most common errors in basic plotting are:

- Using a domain variable that already has a specific value.
- Not specifying a domain variable or specifying the wrong domain variable.

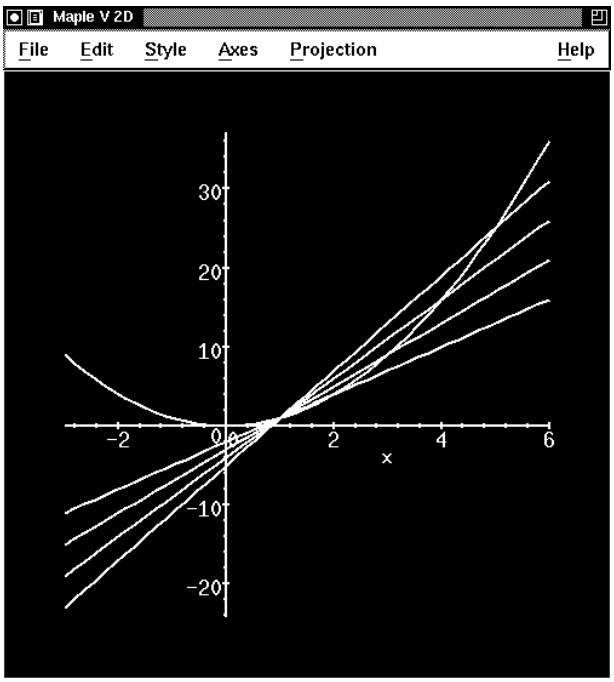


Figure 1.8.4: Window created by Maple with the plot of several functions on the same axes

1.8.2 Fancier plotting

The `plot` command is very powerful, and you should look for details about it using the online help facility:

```
> ?plot;
```

To make you aware of some of its features, we discuss them here:

- It is possible to have Maple *plot points*. This can be done in two ways depending upon how the points are generated. If you have a list of specific points, you can assign them to a name and then plot them as follows:

  ```
  > points:=[1,2],[1.5,1],[2,-1],[2.5,0.5],[3,1],[3.5,0.6],[4,0.2]:

  > plot([points],style=POINT);

  >
  ```

  If `style=POINT` is replaced by `style=LINE`, the dots will be connected by segments of lines.

On the other hand, if the points result from evaluating an expression at several values of `x`, you can use `plot` in its usual form, but specifying `style=POINT`:

```
> plot(x^2,x=-2..2,style=POINT);

>
```

Approximately 50 points will be plotted this way. You can insist that more points be plotted by using the *numpoints* option:

```
> plot(x^2,x=-2..2,style=POINT,numpoints=150);

>
```

- Maple's fanciest plotting functions are contained in a separate library called `plots`. For basic plotting, there are two commands from this library that are especially useful. They are `textplot` and `display`. To load these commands into the computer memory, execute the statement

```
> with(plots,textplot,display);
                              [display,textplot]
>
```

The `display` command is useful to combine different kinds of plots into one picture. You can merge standard plots of expressions, plots of points, plots of text (what `textplot` is for, useful for labelling things), and animations. In the next example, we combine the point plot of the variable `points` defined above with a plot of `sin(3*x)`. We first define them as different plots and then display them together:

```
> plot1:=plot([points],style=POINT):

> plot2:=plot(sin(3*x),x=0..4):

> display({plot1,plot2},view=[0..4,-1..2]);

>
```

Notice that `view` controls the horizontal and vertical ranges to be displayed.

You can attach labels to your plots by combining `textplot` with `display`. The basic syntax of `textplot` is to use it with an argument of the form `[a,b,'name']` which places the word name inside the quotes on the plot so that it is centered at the point (a, b). For example:

```
> y:=(1+x^2)*exp(-x^2/2): d:=diff(y,x):

> F:=plot({y,d},x=-3..3):

> G:=textplot({[1,1,'function'],[0.75,0.45,'derivative']}):

> display({F,G});

>
```

The result in this case is displayed in Fig. 5.

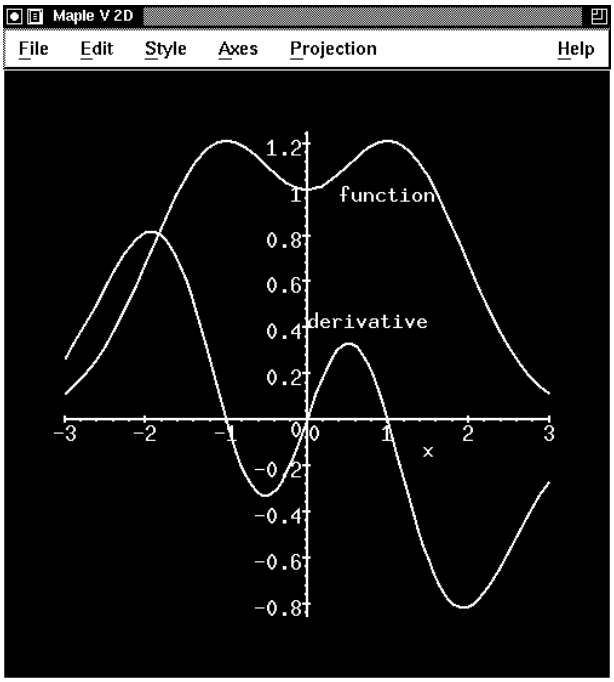


Figure 1.8.5: A plot generated combining textplot and display together

We end with a word about plotting functions (as opposed to expressions), and a situation in which it is a good idea to do so. Sometimes you will have a relationship that you want to plot in the form of a function rather than an expression:

```
> f:=x->x^3*exp(-x):
```

In this situation, you can plot `f(x)` as explained above. Alternatively, you may use:

```
> plot(f,0..3);

>
```

The two ways of plotting a function should not be confused. Neither of the following two statements will work correctly:

```
> plot(f(x),0..3);
Warning in iris-plot: empty plot

> plot(f,x=0..3);

>
```

If you have a function defined using an *if-then* clause, you must use the function plotting command:

```
> f:=x-> if x<3 then x+1 else -x^2+13 fi:

> plot(f(x),x=0..5);
Error, (in f) cannot evaluate boolean
```

The error resulted because Maple attempts to understand the function before it has a value for `x`. In this case, the command

```
> plot(f,0..5);

>
```

will work correctly as indicated in Fig. 6:
You can see from the graph that the function `f` is probably continuous but not differentiable at `x=3`.

1.8.3 Printing a graph

In the `File` menu at the top of any Maple window containing the plot of a curve (look at any of the previously given Figures), you can find a `Print` option which itself has several alternatives. You can save the figure into a postscript file by responding appropriately to the questions you are asked. You can give any name to the file where you want to save the figure, say `figure.ps`. Once you complete this processes, you will have a file with this name in the directory from which you launched Maple. You can print that file with the `lpr` command. However, you have to do so from any other window, not the Maple one.

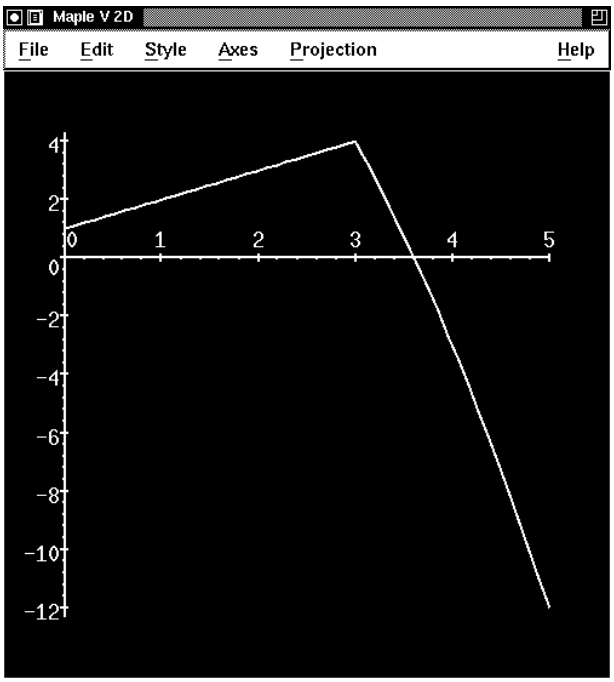


Figure 1.8.6: Plot of $f(x) = x + 1$ if $x < 3$ and $f(x) = -x^2 + 13$ if $x \geq 3$, displayed on the interval $[0, 5]$

1.9 Working with and from the Worksheet

Maple accepts your commands when the Maple prompt `>` shows on it. The session is in `Input mode`. Anything you type in while in this mode is understood as a Maple command and Maple will try to execute it. If the prompt is not present, the session is in `Text mode`. You can type anything you want and Maple will interpret it as text to be recorded but not executed. You can move from the `Input mode` to the `Text mode` by positioning the cursor on the `Input` box and holding down its left button. Such an action will display a menu from which you select the `Text` item. Similarly, you can move back from `Text` mode to `Input` mode by positioning the cursor on the `Edit` menu, holding down the left button and selecting the `Insert Prompt` from the submenu.

To make your worksheets easy to read, in particular, when you have to turn them in as homework, you should mix some amount of text with your Maple commands explaining to the reader what is being done. This will be very useful for you when you save your Maple session and resume work from the saved worksheet at a later date. By that time, you may have forgotten all that you were doing and the inserted text will make easier for you to remember it.

1.9.1 Saving a resuming work from a worksheet

You may save your Maple session to a file so that you can use it again at a later date: with the cursor on the `File` menu of the Maple window, press the left button of the mouse and select the the entry `Save As...`. A small window will appear where you can choose the directory where you want to save the worksheet. In the `Selection` field of this window you can type in a name under which you want to save it.

You may resume work on a previously saved worksheet: use the left button to click on File and select Open from the menu. A Load Session window will appear from where you can choose the file you want to use. Click on it.

When working of a set of Maple problems, it is a good idea to start a new session and worksheet for each problem. This avoids the problem of remembering which variables you have used before and saves you from loosing a whole assignment if something goes wrong with the computer. Try to save your worksheets often, in particular after doing something that you do not want to do again.

1.9.2 Quitting a Maple session and logging out

To quit a Maple session place the cursor on the `File` menu, and while holding down the left button of the mouse select the `Exit` option. You may or may not want to save the worksheet as indicated previously. Answer the question accordingly. To log out of the system, position the cursor in the background and select the `Exit` option. This will terminate your login session.

1.10 Exercises

1. Evaluate $\pi^{\sqrt{2}}$ to 30 decimal places.

2. Input the following as a Maple function:

$$f(x) = \frac{\sqrt{x}(-116x^3 + 8x^4 + 558x^2 - 891x)}{2x^5 - 29x^4 + 140x^3 - 225x^2}\,.$$

   a) Write this in a simpler form (that is, factor and reduce it).

   b) Draw the graph of the function for x between 0 and 1). Adjust the vertical range so that some detail can be seen.

   c) Compute the area of the part of the curve that lies above the x-axis and between $x = 0$ and $x = 5$ (that is, integrate the function over the appropriate range of x values). Give your answer both in actual form and as a decimal approximation to about 10 places.

   d) What is the value of the integral if you remove the factor of $\sqrt{x}$ from the numerator?

e) Use the derivative of $f(x)$ to determine for what real value x the function has a local maximum. Use an approximation to about 8 decimal places.

3. Use the commands `seq` and `ithprime` to generate a list of the first 20 primes. Compute the sum of these 20 primes, and give its integer factors.

4. Find the solutions of the system of equations $\{x^2 - y^2 = 4, x - 2y = 2\}$.

5. Draw a graph showing both $\cos x$ and its fifth Taylor polynomial (that is, $1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4$) for x between -4 and 4. How many terms do you seem to need to get *good agreement* in this range.

   Hint: use a variation of the command `convert(taylor(cos(x),x,5),polynom)` to make this work.

   Think of a suitable way to demonstrate that the approximation you have taken is *good*.

Chapter 2

Least Square Fitting

Let us assume that $y = f_{c_1,\ldots,c_k}(x_1,\ldots,x_m)$ is a real-valued function of $(x_1,\ldots,x_m)$ which depends upon k parameters $c_1,\ldots,c_k$. These parameters are unknown to us. However, suppose that we can perform repeated experiments that for given values of $(x_1,\ldots,x_m)$ allows us to measure output values for y. How can we estimate the parameter $c_1,\ldots,c_k$ that best correspond with this information?

Let us assume that the experiment measuring the value $y = f_c(x)$ for specific input values $x = (x_1,\ldots,x_m)$ is repeated n-times. We will then obtain a system of equations

$$\begin{aligned} f_{c_1,\ldots,c_k}(x_{11},x_{12},\cdots,x_{1m}) &= y_1 \\ f_{c_1,\ldots,c_k}(x_{21},x_{22},\cdots,x_{2m}) &= y_2 \\ &\vdots \\ f_{c_1,\ldots,c_k}(x_{n1},x_{n2},\cdots,x_{nm}) &= y_n \end{aligned}$$

where $x_{\imath\jmath}$ and $y_\imath$ are the measurements of $x_\jmath$ and y in the $\imath$ experiment. These experimental results gives us information about the unknown coefficients $c_\imath$'s.

Since we may perform the experiments as many times as we wish, we may end up with more relations of the type above than unknowns (that is, n is larger than k). The larger the n, the more information we have collected about the coefficients. However, even if the experiments are carried out with great care, they unavoidably will contain some error. The question remains: how may we estimate judiciously the coefficients $c_1,\ldots,c_k$ using the collected information about $y = f_c(x)$? What is the best fit?

A very common method to respond to this question is known as the *method of least-squares*. The idea is simple: per realization of the experiment, we measure the fitting error by the distance from the real number $f_c(x_1,x_2,\cdots,x_m)$ and the observed value of y. The best fit for the distance though will also lead to a best fit for the square of the distance. To avoid absolute values, we change our viewpoint slightly and measure the fitting error by $(f_{c_1,\ldots,c_k}(x_1,x_2,\cdots,x_m)-y)^2$.

The error function that considers all the information obtained from the n experiments is then

$$E(c_1, \ldots, c_k) = \sum_{i=1}^{n} ((f_{c_1,\ldots,c_k}(x_{i1}, x_{i2}, \cdots, x_{im}) - y_i)^2 ,$$

where $(x_{i1}, x_{i2}, \cdots, x_{im})$ and y_i are the i-th input for x and value of the measurement for y.

This turns out to be a function of $c = (c_1, \ldots, c_k)$. Mathematically, our best fit problem is now reduced to finding the value of c which produces a minimum for this error function E. The details of how this can be done depends intrinsically upon the assumed form of the function f, and its relation to the parameters $c_1, \ldots, c_k$.

2.1 Fitting a line to data

Suppose that we are in the situation described above, with $m = 1$, and that the function f is a polynomial of degree 1. Hence, the number of parameters is two, and for convenience we write

$$f(x) = mx + b ,$$

in order to interpret the parameters m and b as the slope and y-intersect of the graph of f. We have data points (x_1, y_1), ..., (x_n, y_n). The problem is now what values of m and b will best fit this data. Or put it simply, what is the straight line that best fit this information.

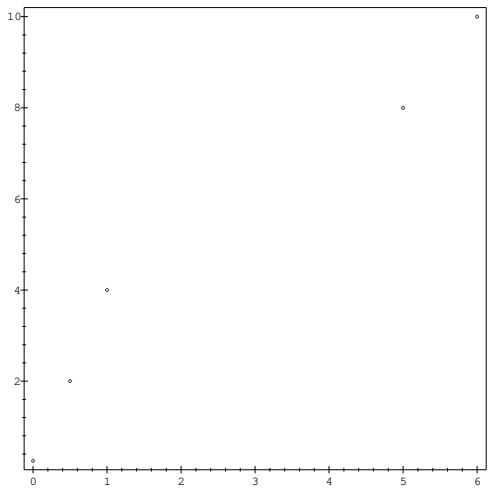
The contribution to the error function coming from the i-th piece of data, (x_i, y_i), is given by $(mx_i + b - y_i)^2$, and therefore, the total error is

$$E(m, b) = \sum_{i=1}^{m} (mx_i + b - y_i)^2 .$$

The problem then becomes simply searching for the values of m and b where the error $E(m, b)$ achieves a minimum. Let us solve this problem using Maple.

First, let us type in some data to try to fit our line to:

```
> pts := [ [0,0.25], [1/2,2], [1,4], [5,8], [6,10] ]:

> plot(pts, style=point, axes=boxed);
```



In order to solve the problem at hand, we could use Maple's built-in regression package to find the answer, but that would not help much in explaining what is going on. However, let us do this anyway.

Since the least-squares package expects the x and y values in separate lists, we have to make a slight adjustment to our data:

```
> with(stats):
  fit[leastsquare[[x,y]]]([[ pts[i][1] $i=1..5], [ pts[i][2] $i=1..5]]);
```

$$y = 1.271370968 + 1.431451613x$$

This says that the best values of m and b are $m = 1.431451613$ and $b = 1.271370968$, respectively. Let us do this *by hand*.

First, we define the error function according to the data points to fit that we have. Notice again that this function is the square of the distance from the line to the data:

```
> E := (m,b)-> sum( ((m*pts[i][1]+b)-pts[i][2] )^2, i=1..5);
```

$$E := (m, b) \rightarrow \sum_{i=1}^{5} (m\, pts_{i1} + b - pts_{i2})^2$$

For example, had we guessed that the line $y = 2x + 1$ is the answer to our problem, we could compute the distance:

```
> E(2,1);
```

$$19.5625$$

However, decreasing m and increasing b produces a smaller value of E, so that guess cannot be the best fit:

```
> E(1.5,1.2);
```

$$3.125000000$$

We want to minimize the error function. Since minima of differentiable functions occur at critical points, we begin by solving for the values of (m, b) which annihilate the two partial derivatives. Notice that Maple happily computes partial derivatives:

```
> diff( E(m,b), m);
```

$$\frac{249}{2}m + 25b - 210$$

We may then ask Maple to solve the system of equations obtained by equating the two partial derivatives by

```
> solve( { diff( E(m,b), m)=0, diff( E(m,b), b)=0 }, {m,b});
```

m = 1.431451613, b = 1.271370968

Maple thus finds only one solution, and not surprisingly, it coincides with the solution found using the statistics package.

If we now want to use this values of m and b without retyping it, one way to do that is to use the command `assign(%)` to let b and m be given these constant values.[1]

```
> assign(%);
```

Maple executes this assignment silently, without reporting the assignment. However, now both m and b have the assigned values:

```
> m;
```

1.431451613

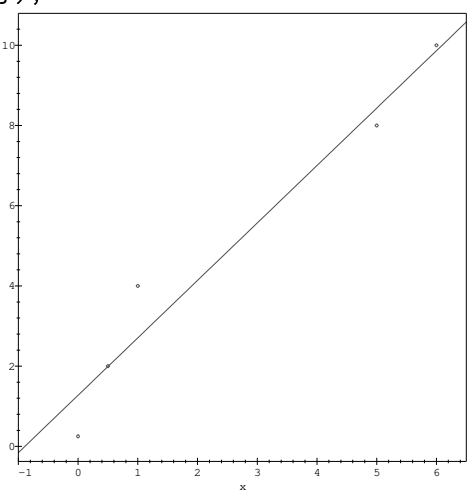
```
> E(m,b);
```

2.929334677

The value of $E(m, b)$ calculated above, is the value of E when (m, b) is the critical point found earlier.

We may visualize how good our fit is, using `plot` to display the data points and the fit line on the same graph. We load the plots package, so that we can use display:

```
> with(plots):
  lplot := plot(m*x+b, x=-1..6.5, axes=boxed):
  pplot := plot(pts, style=point):
  display({lplot, pplot});
```



[1] We are using maple's *ditto operator % to refer to the result of the previous command. In versions of maple prior to release 5 of Maple V, the double quote (") was used for the ditto operator.*

Notice that at this point, m and b have constant values previously assigned to them, the values producing the minimum of E up to 9 decimal places. Let us try to verify that our solution is indeed correct, via an argument which is independent of the one described above.

Geometrically, the error function E is the square of the Euclidean distance between $m\vec{x}+\vec{b}$ and the vector $\vec{y}$, where $\vec{x}=(x_1,x_2,\ldots,x_n)$, $\vec{b}=(b,b,\ldots,b)$ and $\vec{y}=(y_1,\ldots,y_n)$. Clearly then, the best fit is achieved when the coefficients m and b are chosen so that $m\vec{x}+\vec{b}$ is the orthogonal projection of the vector $\vec{y}$ onto the subspace of $\mathbb{R}^n$ spanned by $\vec{x}$ and $(1,1,\ldots,1)$. That is to say, the best fit occurs when the vectors $\vec{y}-(m\vec{x}+\vec{b})$ and $m\vec{x}+\vec{b}$ are perpendicular to each other. Let us verify this:

```
> sum( (pts[i][2]-m*pts[i][1]-b)*(m*pts[i][1]+b),i=1..5);
```

$$-.16\cdot 10^{-7}$$

The inner product is not quite zero, but the first non-zero digit of its decimal expansion occurs in the eighth decimal place. This is because the calculations were only done with finite precision.

As a final remark, we observe that if you now execute the command `diff( E(m,b),m )`, you will obtain an error. This is due to the fact that m and b are constants at this point, and not independent variables. If you would like to continue making use of the function $E(m,b)$ for general m and b, you will have to unassign the values of m and b first:

```
> m := 'm':
  b := 'b':
```

2.2 Fitting a cubic to data

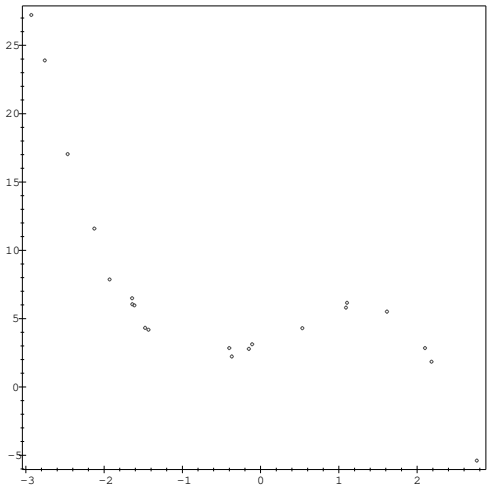
Let us now try to fit a cubic polynomial to some data. We do so directly, without using Maple's built in statistical package.

We randomly generate a set of 21 data points to be fitted to a cubic using a Maple program. (Don't worry about how this program works just now. We'll get into those details later).

```
> Seed:=readlib(randomize)():
  with(stats):
  myrand := (a,b) -> evalf(a+(b-a)*(rand()/10^12)):
  points_wanted := 21:
> cubic_pts := proc()
    local fuzz, a, b, c, d,s,x,i;
    a:= myrand(-3,3);
    b:= myrand(-3,3);
    c:= myrand(-3,3);
    d:= myrand(-10,10);
    if (myrand(-10,10) > 0) then s:=1 else s:=-1 fi;
    fuzz := [ random[normald[0,0.5]](points_wanted)];
    x    := [ random[uniform[-3,3]](points_wanted)];
    RETURN([seq([x[i],s*(a-x[i])*(b-x[i])*(c-x[i])+d+fuzz[i]],
                i=1..points_wanted)]);
  end:
```

Now we can put the data to be fitted into a list, and visualize the result with `plot`:

```
> data:=cubic_pts():
  plot(data, style=point, axes=boxed);
```



Never mind the way we generated the data points. As far as the problem to solve is concerned, we have a set of 21 data points which are to be fitted to a cubic polynomial. A polynomial function of degree n is determined by $n+1$ coefficients. So we define

```
> cub := (x,a,b,c,d)->a*x^3+b*x^2+c*x+d;
```

$$cub := (x,\, a,\, b,\, c,\, d) \to a\,x^3 + b\,x^2 + c\,x + d$$

and then define the error function:

```
> E:=(a,b,c,d)->sum((cub(data[i][1],a,b,c,d)-data[i][2])^2,i=1..nops(data));
```

$$E := (a,\, b,\, c,\, d) \to \sum_{i=1}^{\mathrm{nops}(data)} (cub(data_{i1}, a, b, c, d) - data_{i2})^2$$

We have four parameters to determine, the coefficients of `cub`. We find its values by

```
> assign(solve({diff(E(a,b,c,d),a)=0,
                diff(E(a,b,c,d),b)=0,
                diff(E(a,b,c,d),c)=0,
                diff(E(a,b,c,d),d)=0},
               {a,b,c,d}));
```

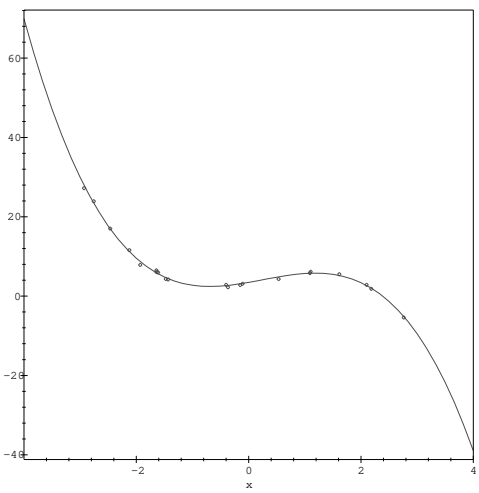
We do not see the answer, but the resulting values have already being assigned to the coefficients a, b, c, d.

```
> cub(x,a,b,c,d);
```

$$-1.006141915x^3 + .7505869176x^2 + 2.487673553x + 3.430571969$$

Finally, let's make a picture of the result, showing both the data and the fitted curve:

```
> with(plots):
  cplot := plot(cub(x,a,b,c,d), x=-4..4, axes=boxed):
  pplot := plot(data, style=point):
  display({cplot,pplot});
```



For comparison, let us find the answer using Maple's built-in statistical package:

```
> with(stats):
  fit[leastsquare[[x,y],y=A*x^3+B*x^2+C*x+D]]
          ([[data[i][1]$i=1..nops(data)],[data[i][2]$i=1..nops(data)]]);
```

$$y = -1.006141912x^3 + .7505869193x^2 + 2.487673537x + 3.430571965$$

For this particular version of least square fitting, the geometric interpretation of the error function E is the square of the Euclidean distance between $a\vec{x}_3 + b\vec{x}_2 + c\vec{x}_1 + \vec{d}$ and the vector $\vec{y}$, where $\vec{x}_j = (x_1^j, x_2^j, \ldots, x_n^j)$, $\vec{d} = d(1, 1, \ldots, d)$ and $\vec{y} = (y_1, \ldots, y_n)$. The best fit is thus achieved when the coefficients a, b, c and d are chosen so that $a\vec{x}_3 + b\vec{x}_2 + c\vec{x}_1 + \vec{d}$ is the orthogonal projection of the vector $\vec{y}$ onto the subspace of $\mathbb{R}^n$ spanned by $\vec{x}_3$, $\vec{x}_2$, $\vec{x}_1$ and $(1, 1, \ldots, 1)$. That is to say, the best fit occurs when the vectors $\vec{y} - (a\vec{x}_3 + b\vec{x}_2 + c\vec{x}_1 + \vec{d})$ and $a\vec{x}_3 + b\vec{x}_2 + c\vec{x}_1 + \vec{d}$ are perpendicular. You should verify that this is the case for the solution given above.

In both cases, you should consider why there is only one critical point for the error function, and that is of necessity a global minimum.

2.3 Fitting other types of funtions

Note that all the previous constructions relied upon was that the type of function we try to fit depend linearly on the coefficients. The same scheme works equally well to fit functions such as $\sum_{i=1}^{n} \sin(nx)$, although note that here we need to solve n simultaneous linear equations.

Even if the function does not depend linearly on the coefficients, it can sometimes be transformed to one which does. For example, if our data points $\{(x_i, y_i)\}$ were believed to approximate an exponential function of the form $y = ae^{kx}$, then setting

$$E(a, k) = \sum_{i=1}^{m} (y_i - ae^{kx_i})^2 .$$

would require us to solve the system

$$\sum_{i=1}^{m}(y_i - ae^{kx_i})e^{kx_i} = 0 \quad \sum_{i=1}^{m}(y_i - ae^{kx_i})ax_ie^{kx_i} = 0\,.$$

While this isn't impossible, it is much more straightforward to make a change of variables.

Assuming the y_i are all positive (which is reasonable since we believe the data is exponential), we can let $z_i = \ln y_i$. Then we are trying to fit a function $z = \ln\left(ae^{kx}\right)$, which reduces to the line $z = \ln a + kx$. This is familiar territory, and we can just proceed as before.

2.4 Fitting a circle

In a slightly different vein, suppose that the data points $\{(x_i, y_i)\}$ lie near a circle of unknown center and radius. Note that if we did know the center, this problem would be very simple: the desired radius would be the average distance from the points to the center. If you don't think very hard, you might suspect that the desired center would be very near the center of mass of the points $\{(x_i, y_i)\}$, but this will only be the case if the points are evenly distributed around the circle.

Instead, we can come up with a function that measures the error between an arbitrary circle and our data points.

Recall that the general equation of a circle centered at (a, b) with a radius of r is

$$(x - a)^2 + (y - b)^2 = r^2.$$

Unlike the previous cases, there is no independent variable. (Note that we *could* try to fit $y_i \simeq b \pm \sqrt{r^2 - (x_i - a)^2}$, but not only would the resulting equations be messy, this would bias things very badly; do you see why?). Nevertheless, we press on.

One reasonable measure of the distance between the points and a circle is the "area difference", that is

$$E(a, b, r) = \sum_i \left((x_i - a)^2 + (y_i - b)^2 - r^2\right)^2\,.$$

One difficulty with this is that E is not quadratic in a, b, and r. However, Maple is able to solve the resulting equations anyway:

```
> cpts:=circle_pts():
  epsilon:=(pt,a,b,r) -> (pt[1]-a)^2 + (pt[2]-b)^2 -r^2;
```

$$\varepsilon := (pt,\, a,\, b,\, r) \to (pt_1 - a)^2 + (pt_2 - b)^2 - r^2$$

```
> E:= (a,b,r) ->   sum( epsilon(cpts[i],a,b,r)^2,  i=1..nops(cpts));
```

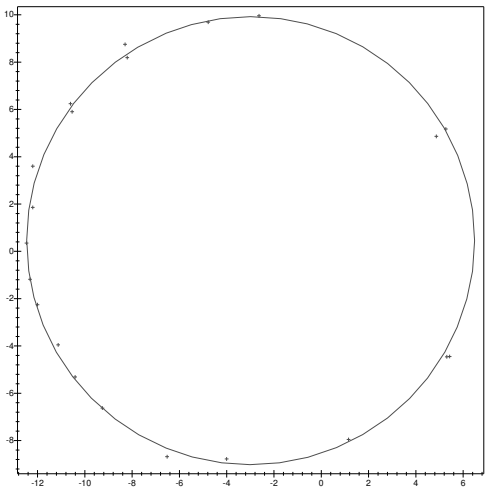
$$E := (a,\, b,\, r) \to \sum_{i=1}^{\mathrm{nops}(cpts)} \varepsilon(cpts_i,\, a,\, b,\, r)^2$$

```
> sol:= solve({diff(E(a,b,r),a)=0,
               diff(E(a,b,r),b)=0,
               diff(E(a,b,r),r)=0},
              {a,b,r});
```

$$
\begin{aligned}
sol := \quad & \{a = -7.279165053 - 12.95193570\,I,\ b = -2.210665541 + 2.912161761\,I,\ r = 0\},\\
& \{a = -7.279165053 + 12.95193570\,I,\ b = -2.210665541 - 2.912161761\,I,\ r = 0\},\\
& \{b = .4328495120,\ a = -4.481107455,\ r = 0\},\\
& \{b = -14.49583861 - 13.00051962\,I,\ a = 4.676168966 - 18.08449394\,I,\ r = 0\},\\
& \{a = 4.676168966 + 18.08449394\,I,\ b = -14.49583861 + 13.00051962\,I,\ r = 0\},\\
& \{r = 9.471322781,\ a = -2.987792224,\ b = .4467421770\},\\
& \{r = -9.471322781,\ a = -2.987792224,\ b = .4467421770\}
\end{aligned}
$$

Here we find a number of critical points for the function E, but from physical considerations, the only reasonable choice is the circle with center at $(-2.987792224, .4467421770)$ and radius 9.471322781. In order to see the fit, we plot the points and the circle. Note that we represent the circle parametrically, and use `subs` to substitute the desired solution.

```
> pp:=plot(cpts,style=point,scaling=constrained,axes=boxed):
  cp:=plot(subs(sol[6],[a+r*cos(t),b+r*sin(t),t=0..2*Pi])):
  plots[display](pp,cp);
```



The fact that there is only one interesting critical point is no accident, however. If we let $k = a^2 + b^2 - r^2$, then the resulting error function $H(a, b, k) = E(a, b, \sqrt{a^2 + b^2 - k})$ is quadratic in a, b, and k, and so we really only need to solve a linear system. It is easy to check that this new functional H still satifies all the criteria we wanted (that is $H(a, b, k) = 0$ if and only if all the data points lie on the circle $C(a, b, k)$, H is non-negative, and it is smooth), so fitting a circle becomes a linear problem after all. The reader should verify that, in fact, the unique minimum of H corresponds exactly to the critical points the original function E for which $r \neq 0$.

The interested reader might want to consider a similar approach to fitting other conic sections, such as an ellipse or a hyperbola, to given data. Do you expect to be able to make the problem linear, as in the case of the circle?

Another interesting variation on least-squares fitting is if we consider the measurement errors to occur in both x_i and y_i. In this case, we want to minimize the shortest distance to a line $y = mx + b$ rather than the vertical distance. In this case, the resulting equations will not be linear, nor can they be made linear. However, maple will be able to find the critical points with no trouble. There will always be at least two (there may be a third with a huge slope and intercept)– one is the minimum, and the other is a saddle point. It is worth thinking for a few minutes about the geometric interpretation of the saddle point in terms of the problem at hand.

2.5 Robust fitting

A discussion needs to be inserted about fitting lines using

$$E(m,b) = \sum_{i=1}^{n} ln\left(1 + (mx_i + b - y_i)^2\right) \ .$$

and

$$E(m,b) = \sum_{i=1}^{n} |mx_i + b - y_i| \ .$$

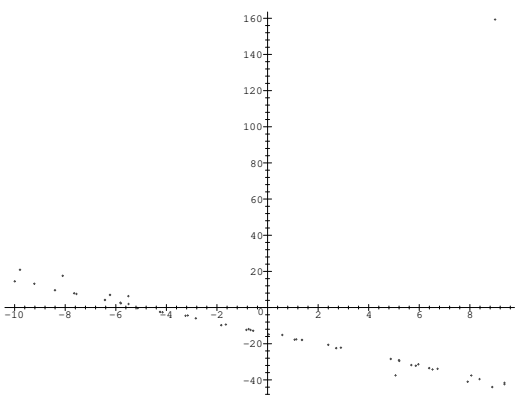
The basic idea is that there is nothing truly magical about minimizing sums of squares, although it is the appropriate thing to do if the errors are Gaussian (see next section). But note that if one has data with large errors ϵ_i, these will have a huge influence (because we use ϵ_i^2). The two mentioned above grow linearly (or sub-linearly) with $|\epsilon_i|$, causing the tails to count much less heavily.

```
> read(`lsq_data.txt`);
```

```
defined line\_pts(), bad\_line\_pts(), quadratic\_pts(), cubic\_pts(),
and circle\_pts()
```

First, we generate some "noisy" data.

```
> pts:=bad_line_pts():
> plot(pts,style=point);
```



```
> epsilon:= (pt,m,b) -> (m*pt[1]+b-pt[2]);
```

$$\varepsilon := (pt,\, m,\, b) \to m\, pt_1 + b - pt_2$$

First, let's find the line given by least squares:

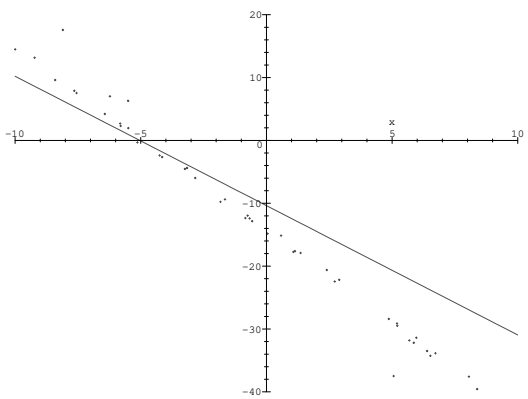
```
> H:=(m,b)->sum(epsilon(pts[i],m,b)^2,i=1..nops(pts));
```

$$H := (m,\, b) \to \sum_{i=1}^{\mathrm{nops}(pts)} \varepsilon(pts_i,\, m,\, b)^2$$

```
> sol:=solve(\{diff(H(m,b),m)=0, diff(H(m,b),b)=0\},\{m,b\});
```

$$sol := \{m = -2.058151695,\, b = -10.36674445\}$$

```
> p := plot(pts,style=point):
  l := plot(subs(sol,m*x+b),x=-10..10):
  plots[display](p,l,view=-40..20);
```



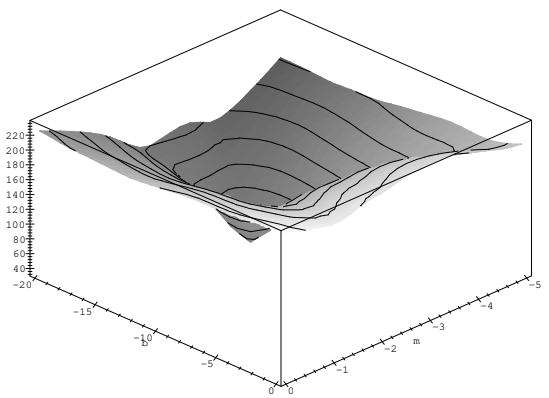
Now, let's try it minimizing $\ln(1+\epsilon_i^2)/2$, which behaves like ϵ^2 for small errors, but grows very slowly for large ones.

```
> R:=(m,b)->sum(ln(1+epsilon(pts[i],m,b)^2/2),i=1..nops(pts));
```

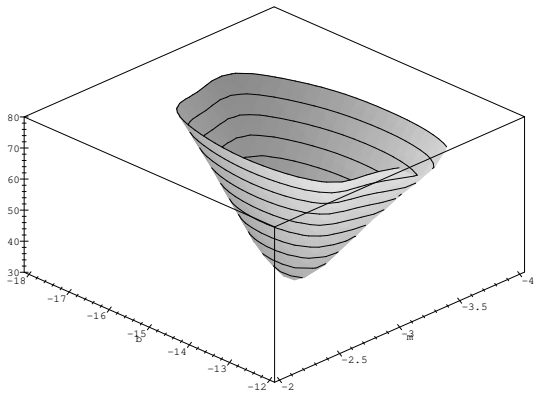
$$R := (m,\, b) \to \sum_{i=1}^{\mathrm{nops}(pts)} \ln(1 + \frac{1}{2}\,\varepsilon(pts_i,\, m,\, b)^2)$$

Here's what the functional we want to minimize looks like:

```
> plot3d(R(m,b),m=-5..0,b=-20..0,style=patchcontour, axes=boxed);
```



```
> plot3d(R(m,b),m=-4..-2,b=-18..-12,view=30..80, style=patchcontour, axes=boxed);
```



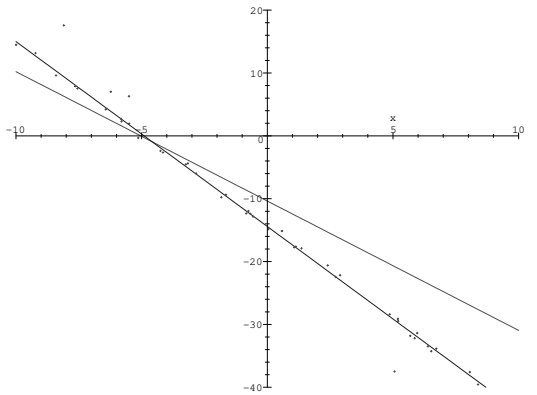
The equations we want to solve are not linear. In fact, they're messy enough that maple needs some help to find the minimum. From the pictures above, we can choose an appropriate region.

```
> rs:=fsolve( \{diff(R(m,b),m)=0, diff(R(m,b),b)=0\},
              \{m,b\},m=-3.1..-2.8, b=-15..-14);
```

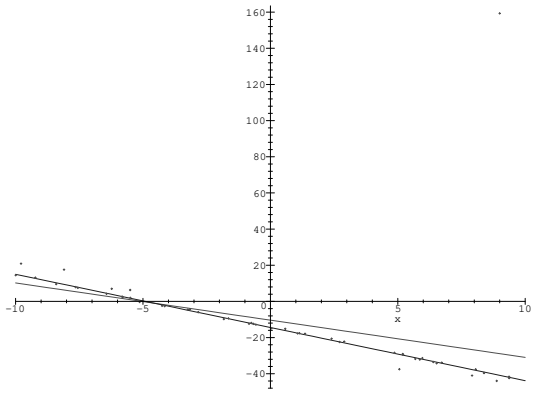
$$rs := \{b = -14.41947456,\ m = -2.942250792\}$$

And here we can compare the two lines found.

```
> p := plot(pts,style=point):
  l := plot(subs(sol,m*x+b),x=-10..10):
  s := plot(subs(rs,m*x+b),x=-10..10,color=blue):
  plots[display](p,l,s,view=-40..20);
```




```
> plots[display](p,l,s);
```



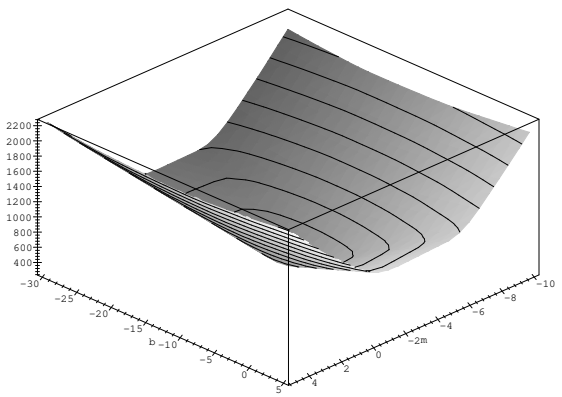
We can also try to minimize $\sum |\epsilon_i|$. Of course, this isn't even differentiable near $\epsilon_i = 0$, so we have to work harder to get less.

```
> A:=(m,b)->sum(abs(epsilon(pts[i],m,b)),i=1..nops(pts));
```

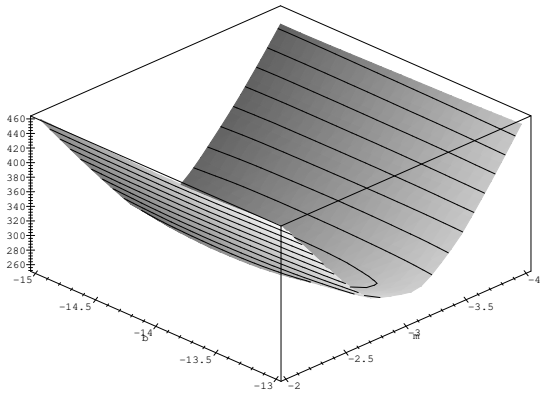
$$A := (m,\, b) \rightarrow \sum_{i=1}^{\mathrm{nops}(pts)} |\varepsilon(pts_i,\, m,\, b)|$$

We can determine graphically about where the minimum is, but to coerce maple into finding the minimum, we would have to work rather hard. There *are* reliable and efficient ways to determine the location of the minimum to any precision (at least given some hints), but they inappropriate for consideration here.

```
> plot3d(A(m,b),m=-10..5,b=-30..5,style=patchcontour, axes=boxed);
```




```
> plot3d(A(m,b),m=-4..-2,b=-15..-13,style=patchcontour, axes=boxed);
```



Repeatedly narrowing in on the minimum graphically will locate the choice of m and b.

2.6 A nod toward statistics

Here we really need a more complete discussion about *why* the sums of squares is what is usually mimimized (there are other considerations besides the fact that it gives a set of linear equations), and that we need to account for the variance of the ϵ_i. Further, we need to discuss correlation and goodness of fit (maybe with a mention of the incomplete gamma function?), possibly the normal distribution, etc. In particular, it is worth noting that the sums of the squares comes out of taking the log of the product of the errors, if errors are normal, and so on.

Chapter 3

fsqFsHn sGGousG

3.1 Introduction to Cryptography

If some information is meant to be kept private, the best means is to keep it well hidden. This is not, of course, always possible. One way around this is to hide the information or message in plain sight, that is, encode it in some way so that even if it is seen, it will be unreadable. The study of such encoding processes is called *Cryptography*[1].

The usual jargon is as follows: the message you want to hide is called the *plaintext*, and the act of encoding it is called *encryption* or *enciphering*. The encoded plaintext is called the *crypttext* or the *ciphertext*, and the act of decoding it is called *decryption* (or "cracking the code", if the decoder wasn't the intended reader). Usually, an encryption system (also called a *cipher*) has an auxiliary piece of information called the *key* needed for the encoding and decoding process. Mathematically, we can represent the encryption process as

$$f_{key}(\mathcal{P}) = \mathcal{C},$$

where $\mathcal{P}$ is the plaintext and $\mathcal{C}$ is the ciphertext. To decrypt the message, one applies f^{-1} to the ciphertext. Note that to be able to decipher a message without ambiguity, f^{-1} must be a well-defined function (although f needn't be!).

[1] Cryptography is related to, but distinct from, *steganography*, which is the process of hiding the fact that a message exists at all. Using "invisible ink" is steganography, as is hiding messages within other messages. For example, looking at the second letter of each word in the message below (which was actually sent by a German spy during World War I),

> Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on byproducts, ejecting suets and vegetable oils.

one finds a rather different message, namely

> Pershing sails from NY June I.

Steganography is often used to augment cryptography, and is also related to "digital watermarking". We will not be considering steganography here.

3.2 Simple Ciphers

3.2.1 Simple substitution

One of the most common (and very easy to crack) ciphers is substitution. One sometimes sees these in a newspaper somewhere near the crossword puzzle. This scheme is also the one used in Poe's story "The Gold Bug", and the Sherlock Holmes "Mystery of the Dancing Men". The idea is straightforward: choose a rearrangement of the letters of the alphabet, and replace each letter in the plaintext by its corresponding one.

As an example, let's implement this in Maple. Recall that we need to use a backquote (`) around character strings in Maple.[2]

```
> Alphabet :=`abcdefghijklmnopqrstuvwxyz `:
> Cryptabet:=`THEQUICKBROWNFXJMPSVLAZYDG `:
> Scramble:= msg ->
    cat(seq(substring(Cryptabet,
                      searchtext(substring(msg,i),
                                 Alphabet)),
            i=1..length(msg)));
```

$$Scramble := msg \rightarrow \mathrm{cat}(\mathrm{seq}(\mathrm{substring}(Cryptabet, \mathrm{searchtext}(\mathrm{substring}(msg, i), Alphabet)), i = 1..\mathrm{length}(msg)))$$

```
> Scramble(`please do not read this`);
```

$$JWUTSU\ QX\ FXV\ PUTQ\ VKBS$$

Probably some comments are in order. First, we tell maple what our permutation is by defining the `Alphabet` that our plaintext is written in, and the corresponding `Cryptabet`. Our encoding function `Scramble` operates as follows (read from the middle out)

- `substring(msg,i)` extracts the i-th character from the input string `msg`.
- `searchtext` looks up where the character is in `Alphabet`.
- `substring` is used again, to extract the corresponding character from `Cryptabet`.
- this is done for each i ranging from 1 to the length of `msg`, giving a sequence of characters.
- finally, all the characters are smashed together again into one long string using `cat`.

Note that our "alphabet" contains a space, which corresponds to a space in our key. We didn't need to do this, but it left the message in words of the same length. This, of course, makes the message easier to guess. Also, notice that we used `searchtext`, which ignores case in the search (that is, "A" and "a" are the same). Had we wished to, we could have distinguished between upper an lower case using `SearchText` in which case we would also need to use a

[2] In Maple V release 5 and later, strings may be surrounded by double quotes ("), which is a more common way of denoting character strings. However, using a backquote works in all versions.

53 letter alphabet (and a 53 letter key). In this implementation, we also had to limit our input message to characters occurring in our alphabet. Had our message contained numbers or punctuation, the function `Scramble` would have failed with an error.

The usual way to break such a code is a combination of "frequency analysis" (that is, knowing that the most commonly occurring letter in the English language is "E", followed by "T"), plus a bit of trial and error. Leaving the spaces between words helps the cracker greatly.

3.2.2 The Caesar cipher, and the ASCII encoding

A variation on the arbitrary permutation discussed above is the "Caesar cipher", named after Julius Caesar, who supposedly invented it himself. This is also what was done by a "Captain Midnight Secret Decoder Ring", popular in the 1930s and 40s. Here we convert our alphabet to numeric equivalents (with, say A=0, B=1, and so on), add an offset to each numeric equivalent (Caesar used 3), then re-encode the numbers as letters. We do the addition modulo the length of the alphabet; that is, when we shift a Y by 3, we get B, since $24+3=27$, and $27 \equiv 1 \pmod{26}$. However, we won't always restrict ourselves to a 26 letter alphabet— we may want punctuation, numbers, and to preserve upper and lower case distinctions.

Treating characters as numbers

Computers already have a built-in numeric representation of the alphabet— the ASCII encoding.[3] At their lowest level, digital computers operate only on numeric quantities— they just manipulate ones and zeros. The smallest piece of information is the state of a switch: it is off (0) or it is on (1). This is called a *bit*, which is short for **b**inary digi**t**. A group of 8 bits is called a *byte*, and is a single "character" of information. A byte can have $2^8 = 256$ different states:

$$00000000, 00000001, 00000010, 00000011, \ldots, 11111111$$

which are generally interpreted either as representing the integers 0 through 255, or the highest bit is often taken to represent the sign, in which case we can represent integers between -128 and 127. (We get an "extra" negative number because we don't need -0, so the byte 10000000 can mean -128 instead.)[4] For notational convenience, bytes are often written in hexadecimal (that is, in base 16), grouping the bits into two blocks of 4 bits (sometimes called a *nybble*, half a byte), and letting the characters A–F stand for the integers 10–15. Thus, the byte 01011111 would be written as 5F in hexadecimal.

[3] In fact, while most computers these days use ASCII, it is not the only such arrangement. IBM mainframes, for example, primarily use another encoding called EBCDIC. But this makes no real difference to our discussion.

[4] The next larger unit in a computer is a *word*. However, different makes of computers have different sizes of words. Most computers today have 4 byte words (32 bits), but word sizes of 2 bytes (16 bits) and 8 bytes (64 bits) are not uncommon. 8-bit bytes are now essentially universal although in the early days of computing, there were computers which used other sizes, such as 6 bit bytes.

So how does a computer represent characters? There is an agreed upon standard encoding of characters, punctuation, and control codes into the first 127 integers. This is called ASCII, an acronym for American Standard Code for Information Interchange. "Extended ASCII" encodes more characters into the range 128–255 (this is where you would find characters for å and ±, for example), although the characters found in this range vary a bit. Not all programs can deal successfully with characters in the extended range.[5]

Maple has a built-in routine to allow us to convert characters to and from their ASCII equivalents. If it is called with a list of numbers, it returns a string of characters, and if called with a character string, out comes a list of numbers. For example,

```
> convert(`How do I work this?`,bytes);
```

[72, 111, 119, 32, 100, 111, 32, 73, 32, 119, 111, 114, 107, 32, 116, 104, 105, 115, 63]

```
> convert([72, 117, 104, 63],bytes);
```

Huh?

We can use this to make a list of the ASCII codes, if we like:

```
> seq([i,convert([i],bytes)],i=0..127);
```

[0,], [1, □], [2, □], [3, □], [4, □], [5, □], [6, □], [7, \a], [8, \b], [9, \t], [10, \n],
[11, \v], [12, \f], [13, \r], [14, □], [15, □], [16, □], [17, □], [18, □], [19, □], [20, □],
[21, □], [22, □], [23, □], [24, □], [25, □], [26, □], [27, \e], [28, □], [29, □], [30, □],
[31, □], [32,], [33, !], [34, \], [35, #], [36, $], [37, %], [38, &], [39, '], [40, (],
[41,)], [42, *], [43, +], [44, ,], [45, -], [46, .], [47, /], [48, 0], [49, 1], [50, 2],
[51, 3], [52, 4], [53, 5], [54, 6], [55, 7], [56, 8], [57, 9], [58, :], [59, ;], [60, <],
[61, =], [62, >], [63, ?], [64, @], [65, A], [66, B], [67, C], [68, D], [69, E], [70, F],
[71, G], [72, H], [73, I], [74, J], [75, K], [76, L], [77, M], [78, N], [79, O], [80, P],
[81, Q], [82, R], [83, S], [84, T], [85, U], [86, V], [87, W], [88, X], [89, Y], [90, Z],
[91, [], [92, \], [93,]], [94, ^], [95, _], [96, `], [97, a], [98, b], [99, c], [100, d],
[101, e], [102, f], [103, g], [104, h], [105, i], [106, j], [107, k], [108, l], [109, m], [110, n],
[111, o], [112, p], [113, q], [114, r], [115, s], [116, t], [117, u], [118, v], [119, w], [120, x],
[121, y], [122, z], [123, {], [124, |], [125, }], [126, ~], [127, □]

Render unto Caesar

Now we are nearly ready to write a version of the Caesar cipher. The basic idea is that for each character of the plaintext, we convert it to its ASCII encoding, add a pre-specified offset modulo 127, then convert back to characters. It will look very similar to the `Scramble` function we wrote in §3.2.1, except we needn't declare an alphabet explicitly. However, in addition to the maple commands `convert( ,bytes)`, we will use another new command, `map`.

[5] In the late 1980s and early 1990s, there was a big push to make programs "8-bit clean", that is, to be able to handle these "unusual" characters.

The `map` command allows us to apply a function to every element of a vector or list in one call. For example, to compute the square root of each of the first 10 integers, we could do

```
> map(sqrt,[1,2,3,4,5,6,7,8,9,10]);
```

$$[1,\ \sqrt{2},\ \sqrt{3},\ 2,\ \sqrt{5},\ \sqrt{6},\ \sqrt{7},\ 2\sqrt{2},\ 3,\ \sqrt{10}]$$

If the function we wish to apply takes two variables, we can give the second one as a third argument to `map`. For example, the command

```
> map(modp,[1,2,3,4,5,6,7,8,9,10],3);
```

$$[1,\ 2,\ 0,\ 1,\ 2,\ 0,\ 1,\ 2,\ 0,\ 1]$$

computes $n \bmod 3$ as n ranges over the list $[1, 2, \ldots, 10]$. (Recall that $n \bmod p$ is the remainder after dividing n by p.

We *finally* are ready. Remember to read from the middle outward. Note that we are defining the function we are `map`ping directly.

```
> Caesar:= (plain, shift) ->
    convert(map((x,y)->(x+y-1) mod 127 + 1,
                convert(plain,bytes),
                shift),
            bytes);
```

$$Caesar := (plain,\ shift) \to \text{convert}(\text{map}((x,\ y) \to ((x + y - 1) \bmod 127) + 1,\ \text{convert}(plain,\ bytes),\ shift),\ bytes)$$

Let's give it a try:

```
> Caesar(‘Hello there‘,3);
```

Khoor#wkhuh

It seems to work. Note that we didn't really compute $(x + \texttt{shift}) \bmod 127$; instead we subtracted one from the sum, computed mod 127, then added 1 again. Why? Consider what would happen if, for example, our character was "k" (ASCII code 107) and we shifted by 20. Then $107 + 20 \bmod 127$ gives 0, which is the code for the null byte, indicating the end of a string. This would be a problem. So instead, we add 1 afterwards to ensure that our answer is in the range $1, \ldots, 127$. And just to avoid confusing ourselves, we subtract the 1 first, so that shifting by 0 doesn't change anything.

Also, you might wonder why we can't use

```
> map(x -> (x+shift-1) mod 127 + 1, convert(plain,bytes))
```

in the middle of the function. If you try it, you'll see it does not work. This is because maple tries to fill in the value of `shift` at the wrong time.

Now, aside from the fact that this function is a bit confusing to read and offers essentially no cryptographic security, what else is wrong with it? Consider the following example:

```
> Caesar(‘Who put the benzedrine in Mrs. Murphy’s Ovaltine?‘, 86);
```

.?FvGLKvK?<v9<EQ<;I@E<v@Ev$IJ□v$LIG?P}Jv&M8CK@E< □

In addition to the fact that the result looks funny, if we were to try to decipher this from printed output, we would have a little (but not much) trouble, because the characters "." and "?", after encoding, print out as □.[6] This happens because the numeric values of these characters, after shifting by 86, are 5 and 22, respectively. Neither of those corresponds to a printing character, so we can't tell them apart. Of course, if we had no intention of dealing with our encoded message in printed form, this would be no problem.

How can we get around this? We can restrict ourselves to printing characters using a variation of the idea in `Scramble` of §3.2.1, that is, giving an explicit `Alphabet` that we will deal in.

Before doing this, however, it will be helpful to discuss the `proc` command, which allows us to construct functions that consist of more than one maple command.

3.3 Defining functions with `proc`; Local and global variables

We already know how to define functions in maple. However, all of the functions we have defined so far have consisted of a single maple expression (no matter how complicated it might be). For example,

```
> sqr := x -> sqrt(x);
```

$$sqr := \mathrm{sqrt}$$

```
> sqr(25);
```

$$5$$

However, maple has a more versatile way of defining functions using the `proc` command. The function `sqr2` below behaves exactly the same as `sqr` defined above. (When using `proc`, the function begins with the keyword `proc`, followed by any parameters in parenthesis, then one or more maple statements, and finally the keyword `end`. The result of the function is the result of the last maple command executed.

```
> sqr2 := proc(x)
     sqrt(x);
  end;
```

$$sqr2 := \mathbf{proc}(x)\,\mathrm{sqrt}(x)\,\mathbf{end}$$

```
> sqr2(25);
```

$$5$$

```
> sqr2(-25);
```

$$5\,I$$

[6] Of course, it is easy to make examples where significant parts of the message all come out as □. One such is the same message with a shift of 50.

Unlike the functions defined by `->`, however, we can use several statements when using `proc`:

```
> sqr3 := proc(x)
    if (x >= 0) then
       sqrt(x);
    else
       print(‘Square root of ‘,x,‘is not a real number‘);
    fi;
  end;
```

$$sqr3 := \mathbf{proc}(x) \; \mathbf{if}\, 0 \le x \,\mathbf{then}\, \mathrm{sqrt}(x) \,\mathbf{else}\, \mathrm{print}(\text{`}\textit{Square root of}\text{`}, x, \text{`}\textit{is not a real number}\text{`})\, \mathbf{fi} \; \mathbf{end}$$

```
> sqr3(-25);
```

$$\textit{Square root of } , -25, \textit{ is not a real number}$$

```
> sqr3(25);
```

$$5$$

Here is another variation on the same idea: we compute a variable $y = x^2$. If $y > 25$, the result is 25, otherwise it is y. [7]

```
> try:= proc(x)
    y := x^2;
    if ( y>25) then
       25;
    else
       y;
    fi;
  end;
```

```
Warning, ‘y‘ is implicitly declared local
```

$$try := \mathbf{proc}(x)\, \mathbf{local}\, y;\; y := x^2 \,;\; \mathbf{if}\, 25 < y \,\mathbf{then}\, 25 \,\mathbf{else}\, y \,\mathbf{fi}\; \mathbf{end}$$

What does that warning mean? It means that maple has assumed that the variable `y` used in `try` is *local* to that procedure. That is, it is different from a variable `y` that may be used outside the context of the procedure. Let's check that. First, we set `y` to 35, invoke `try(19)` (which sets *its* copy of `y` to 361), then check what the value of `y` is.

```
> y:=35;
```

$$y := 35$$

```
> try(19);
```

$$25$$

```
> y;
```

$$35$$

[7] Note that this could also be done using `piecewise`.

It is still 35. Now, let's add a line saying that `y` has a *global* definition— the `y` within the procedure is the same as the `y` outside.

```
> try2:= proc(x)
    global y;

    y := x^2;
    if ( y>25) then
       25;
    else
       y;
    fi;
  end;
```

$$try2 := \mathbf{proc}(x)\,\mathbf{global}\,y;\ y := x^2\ ;\ \mathbf{if}\,25 < y\,\mathbf{then}\,25\,\mathbf{else}\,y\,\mathbf{fi}\,\mathbf{end}$$

```
> y;
```

$$35$$

```
> try2(19);
```

$$25$$

```
> y;
```

$$361$$

This time, of course, `y` was changed by invoking `try2`.

The opposite of `global` is, not surprisingly, `local`. In order to stop maple from giving warnings about variables being implicitly declared local, we can add a the statement `local y;` just after the `proc` statement. This is a good habit to get into, because it lessens the chance of accidentally using a global variable or of misspelling the name of a variable.

3.4 Caesar cipher redux

Now that we know about `proc`, lets implement the Caesar cipher again, this time with an arbitrary alphabet.

First, we define a global variable `Alphabet` that lists all the characters we intend to encode. Note that, in this case, the newline character is in our alphabet, as well as a space and a few punctuation characters.

```
> Alphabet := `ABCDEFGHIJKLMNOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz., `;
```

$$Alphabet := ABCDEFGHIJKLMNOPQRSTUVWXYZ\backslash$$
$$abcdefghijklmnopqrstuvwxyz.,$$

Now, we can define two procedures, `ToNum` and `FromNum`, which convert character strings to and from lists of numbers. The numbers correspond to the position of the character in `Alphabet`, with (in this case) A=0. As in `Scramble` of §3.2.1, we use `substring` to pick out the i-th character, and `SearchText` to find its position in `Alphabet`. Note that `Alphabet` is declared as a global variable.

```
> ToNum := proc(text)
    local i;
    global Alphabet;
    [seq(SearchText( substring(text,i),Alphabet)-1, i=1..length(text))];
  end:

  FromNum := proc(numlist)
    local i;
    global Alphabet;
    cat(seq( substring(Alphabet,numlist[i]+1), i=1..nops(numlist)));
  end:
```

Now the Caesar cipher is written in terms of these functions. Note that this is a little easier to understand than the previous version.

```
> Caesar2:= proc(plaintext, shift)
    local textnum,codenum,i,p;
    global Alphabet;

    p       := length(Alphabet);
    textnum := ToNum(plaintext);
    codenum := [seq( modp(textnum[i]+shift, p),
                     i=1..length(plaintext)) ];
    FromNum(codenum);
  end:
```

It works as follows.

- We are given a `plaintext` message, and a shift amount. The temporary variables that we will use during the conversion process are declared as local, and the `Alphabet` is declared as global.

- The number of characters in the alphabet is saved as `p`.

- The plaintext is converted to a list of numbers called `textnum`.

- Each number in `textnum` is shifted, and the result is taken mod `p`. Note that since `ToNum` yields result in $\{0, 1, \ldots, p-1\}$, we don't have to monkey around to get the result into the proper range like we did in the first `Caesar` (§3.2.2, p. 57). Also note that we use `modp(a,p)` to compute $a \bmod p$, instead of writing `a mod p` as before. Either way works fine.

- Finally, the result is converted back to characters.

Let's see how it works.

```
> Caesar2(‘Veni, Vidi, Vici‘,3);
```

YhqlBCYlglBCYlfl

```
> test:=‘I have heard the mermaids singing, each to each
  I do not think that they will sing to me.‘;
```

$$test := I\ have\ heard\ the\ mermaids\ singing,\ each\ to\ each\backslash \\ I\ do\ not\ think\ that\ they\ will\ sing\ to\ me.$$

```
> encoded:=Caesar2(test,38);
```

$$encoded := tkQJdNkQNJ\backslash \\ MkbQNkVN\backslash \\ VJRMakaRWPRWPjkNJLQkbXkNJLQItkMXkWXbkbQRWTkbQJbkbQNgk\backslash \\ eRUUkaRWPkbXkVNi$$

Note that since our alphabet treats an end-of-line as a character, the linebreaks in the encoded version are significant. We can use the same function to decode, but giving the negative of the shift as the key.

```
> Caesar2(encoded,-38);
```

$$I\ have\ heard\ the\ mermaids\ singing,\ each\ to\ each\backslash \\ I\ do\ not\ think\ that\ they\ will\ sing\ to\ me.$$

Finally, notice what happens if we try to encode characters that do not occur in our alphabet. To better see what happens, we will shorten the alphabet dramatically.

```
> Alphabet:=‘abcdefghijklmop_‘;
```

$$Alphabet := abcdefghijklmop_$$

Now let's re-encode the test message, with a shift of zero. Ordinarily, a shift by 0 would not change the text at all. But with the shortened alphabet, something happens to the characters that aren't mentioned.

```
> Caesar2(test,0);
```

$$_ha_e_hea_d__he_me_maid__i_gi_g__each__o_each__do__o__hi_k__ha__he__ill_i_g__o_me_$$

Can you explain why this happens?

3.5 Improved Caesar-like ciphers

Certainly the Caesar cipher offers no cryptographic security at all: if you know the alphabet the message was encoded in, you need only guess one character to crack the code. Even if you don't know the alphabet, guessing the correspondence is not very hard with a little patience.

In this section, we will discuss a few approaches to improving the security, while retaining the basic idea of character shifting.

3.5.1 The Vignère cipher

One way to make a Caesar cipher a bit harder to break is to use different shifts at different positions in the message. For example, we could shift the first character by 25, the second

by 14, the third by 17, and the fourth by 10. Then we repeat the pattern, shifting the fifth character by 25, the sixth by 14, and so on, until we run out of characters in the plaintext. Such a scheme is called a Vignère cipher[8], which was first used around 1600, and was popularly believed to be unbreakable. [9]

In our example, the key consists of the four shifts $[25, 14, 17, 10]$, which are the numerical equivalents of the string "ZORK" in a 26-letter alphabet consisting of the letters A–Z. It is common practice to think of our key as plaintext letters, rather than their numerical equivalents, but either will do. We can encode the string "CRYPTOGRAPH" as

C	R	Y	P	T	O	G	R	A	P	H
$+25$	$+14$	$+17$	$+10$	$+25$	$+14$	$+17$	$+10$	$+25$	$+14$	$+17$
B	F	P	Z	S	C	X	B	Z	D	Y

Note that in the above, the letter "R" in the plaintext encodes to both "F" and "B" in the crypttext, depending on its position. Similarly, the two "Z"s in the crypttext come from different plain characters.

But, still, this is not very hard to crack. If we know that the key is of a certain length, say 4, and our plaintext is sufficiently long, then we can perform frequency analysis on every fourth letter. Even if we don't know the length, it is not too hard to write a computer program to try all the lengths less than, say, 10, and pick the one that looks best.

```
> Alphabet := `ABCDEFGHIJKLMNOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz., `:

  Vignere:= proc(plaintext, key)
    local textnum,codenum,i,p,offsets,keylen;
    global Alphabet;

    p       := length(Alphabet);
    offsets := ToNum(key);
    keylen  := length(key);
    textnum := ToNum(plaintext);
    codenum := [seq(
                   modp(textnum[i] + offsets[modp(i-1,keylen)+1],
                        p),
                   i=1..length(plaintext)) ];
    FromNum(codenum);
  end:
```

[8]This cipher takes its name after Blaise de Vignère, although it is actually a corruption of the one he introduced in 1585. Vignère's original cipher changed the shift amount each letter based on the result of the last encoding, and never repeated. This scheme is *much* harder to break. However, one reason for its lack of popularity was probably due to the fact that a single error renders the rest of the message undecipherable.

[9]In fact, as late as 1917, this cipher was described as "impossible of translation" in a respected journal (Scientific American), even though the means to break it had been well known among cryptographers for at least 50 years.

```
> coded:=Vignere(test,‘Prufrock‘);
```

$$coded := Yr\backslash$$
$$EkRcQuQjHrfINPbXVbNJMFrkMcTJWwquIQPIkGduIQPIIYrWSr\backslash$$
$$PbPi\backslash$$
$$McXcbxQlfiUFgPlaPaoTRAWuXdoNNN$$

We can make the decoding function from the original (let's call it `unVignere`) by changing exactly one + sign to a $-$.[10] We omit the change here so perhaps you will figure it out for yourself. But we will test it, to show you that it does work.

```
> unVignere(code,‘Prufrock‘);
```

I have heard the mermaids singing, each to each
I do not think that they will sing to me.

3.5.2 One-time pads

Note that the longer the key is in the Vignère cipher, the harder it is to break (provided the key-string isn't a well-known text, like the Gettysburg address or Hamlet's soliloquy). A *one-time pad* is essentially a Vignère cipher with an infinitely long, preferably random, key. Theoretically, this provides an unbreakable cryptosystem, because deciphering part of the message gives no information about the rest. The catch is that, to decode it, you must know what the infinitely long random sequence the message was encoded with.

Such systems were used by spies, where agents were furnished with codebooks containing pages and pages of random characters. Then the key to the encryption is given by the page on which to begin. It is, of course, important that each page be used only once (hence the name "one-time pad"), because otherwise if a codebreaker were able to intercept a message and (via some other covert means) its corresponding translation, that could be used to decipher messages encoded with the same page. This sort of setup makes sense if an agent in the field is communicating with central command (but not with each other). Each agent is given his own codebook, and he uses one page per message.

A variation on this theme is the "Augustus cipher", where instead of a random sequence of shifts, a phrase or passage from a text which is as long as the plaintext is used. The trouble with this is that, because of the regularities in the key, a statistical analysis of the crypttext allows one to break the cipher.

We can easily modify our `Vignere` program to be a one-time pad system, using maple's random number generator to make our one-time pad.[11] You might think that generating a random

[10] Note that we could also use the `Vignere` routine, but with the inverse of the key. For example, in the preceding example, the inverse of `Prufrock` is `oMJYMPaT`: the numeric code of `P` plus the numeric code of `o` is 56 (the length of the alphabet), similarly for `r` and `U`, and so on.

[11] Technically speaking, this is not a one-time pad, but a one-time stream. The distinction is subtle, and we will ignore it here.

sequence of numbers would be inherently unreproducible, so the message would be indecipherable unless we record the "random stream". However, computers can not usually generate a truly random sequence. Instead, they generate a "pseudo-random" sequence $s_1, s_2, s_3, \ldots$ where the pattern of the numbers s_i is supposed to be unpredictable, no matter how many of the values of the sequence you already know. However, to start the sequence off, the pseudo-random number generator requires a *seed*— whenever it is started with the same seed, the same sequence results. We can use this to our advantage, taking the seed to be our key.[12] Note that in order to decode the message by knowing the key (the seed), the recipient must use the *same* pseudo-random number generator.

Maple's random number generator gives different results when called in different ways. If called as `rand()`, it gives a pseudo-random non-negative 12 digit integer. When called as `rand(p)`, it gives a procedure to generate a random integer between 0 and `p`; it is this second version that we will use. The seed for maple's random number generators is the global variable `_seed`.[13]

```
> OneTimePad := proc(plaintext, keynum)
    local textnum,codenum,i,p,randnum;
    global Alphabet,_seed;

    p       := length(Alphabet);
    randnum := rand(p);
    _seed   := keynum;
    textnum := ToNum(plaintext);
    codenum := [seq((textnum[i] + randnum()) mod p, i=1..length(plaintext))];
    FromNum(codenum);
  end:
```

In this implementation, it is assumed that the key is a positive integer (it can be as large as you like). It would be easy to change it to use a string of characters, however, by converting the string to a number first. One way to do that is discussed in the next section.

It should probably also be mentioned that usually, a one-time pad is described by taking the exclusive-or (XOR) of the random number and the integer of the plaintext, rather than adding them as we have done. This does not significantly change the algorithm if the random sequence is really random, since adding a random number is the same as XOR'ing a different random number. However, the XOR approach has the advantage that the enciphering transformation is its own inverse. That is, if we produce the crypttext using `crypt:=OneTimePadXOR(plain,key):`, then `OneTimePadXOR(crypt,key):` will give the decryption with the same key. This is not the case for the version given above; we would need to modify the routine above to make a decryption proc by changing the $+$ to a $-$.

[12] Pseudo-random number generators appropriate for cryptography are rare. Most implementations (including maple's) are good enough for everyday use, but not enough to be cryptographically secure. By analyzing the output of a typical random number generator, a good cryptanalyst can usually determine the pattern. We shall ignore this problem here, however.

[13] We can choose a "random" seed (based on the computer's clock) using the function `randomize()`.

3.5.3 Multi-character alphabets

We can also improve security a bit by treating larger chunks of text as the characters of our message. For example, if we start with the usual 26-letter alphabet A–Z, we can turn it into a 676-letter alphabet by treating pairs of letters as a unit (such pairs are called *digraphs*), or a 26^3-letter alphabet by using trigraphs, or triples of letters. This makes frequency analysis much harder, and is quite easy to combine with the other crytptosystems already discussed.

To convert the digraph "HI" to an integer (using a length 26^2 alphabet of digraphs), one simple way is to just treat it as a base-26 number. That is, "HI" becomes $7 * 26 + 8 = 190$, assuming the usual correspondence of H=7, I=8. To convert back, we look at the quotient and remainder when dividing by 26. For example, $300 = 26 * 11 + 14$, yielding "LO".

Below is an example of how we might implement this conversion. We assume our usual functions `ToNum` and `FromNum` are defined, as well as the global `Alphabet`. The routine below converts `text` into a list of integers, treating each block of `k` letters as a unit. It is assumed that the length of `text` is divisible by `k`.[14] A block of `k` characters $c_1c_2c_3\ldots c_k$ is assigned the numeric value $\sum_{i=0}^{k-1} x_i k^i$, where x_i is the value of `ToNum`(c_i). The only tricky part is getting the right characters in the right position.

```
> KgraphToKnums := proc(text, k)
    local textnums, i, j, p;
    global Alphabet;
    p:= length(Alphabet);

    textnums := ToNum(text);
    [seq( sum( textnums[(j-1)*k + i]* p^(k-i), i=1..k),
          j=1..length(text)/k)];
  end:
```

This may look a bit impenetrable at first, but when reading it, just keep in mind that `j` selects which `k`-block we are looking at, and `i` is the position of the character within that block.

To undo the transformation, we note that if

$$X = x_1 p^{k-1} + x_2 p^{k-2} + \ldots + x_{k-1} p + x_k,$$

then $x_i \equiv \left[X/p^{k-i}\right] \mod p^{k+1-i}$, where $[a]$ means the integer part of a (that is, we divide and discard any remainder).[15] We use maple's `iquo` function to do the division without remainder.

```
> KnumsToText := proc(numlist, k)
    local p;
    global Alphabet;
    p:=length(Alphabet);
    FromNum([seq(seq(iquo(modp(numlist[j], p^(k-i+1)),
                          p^(k-i)),
                     i=1..k),
                 j=1..nops(numlist))]);
  end:
```

[14] See the routine `TextToVects` in §3.8.1 for an way around this

[15] There are more computationally efficient ways of doing this, but we aren't worrying about that.

In the examples below, we are using the 26-character alphabet discussed above. Of course, this will work on any alphabet.

```
> KgraphToBignums(`HILOHOLA`,2);
```

[190, 300, 196, 286]

We can also use 4-graphs:

```
> KgraphToBignums(`HILOHOLA`,4);
```

[128740, 132782]

```
> KnumsToText([128740, 132782],4);
```

HILOHOLA

Another way to treat multiple characters together is to think of them as vectors. For example, the digraph "HI" might correspond to the vector [7, 8]. We will treat this approach in §3.8.

3.6 Reading and Writing from a file

While none of the methods we have covered yet are secure enough to foil a good cryptanalyst, the `OneTimePad` procedure of §3.5.2 is good enough for household use (unless you live with cryptanalysts!). Should we want to really use it, however, we would find it extremely tedious to type in our message each time, and then copy the encrypted text from the worksheet.

Fortunately, maple is able to read and write data from a file, using `readline` and `writeline`. These read (or write) a single line of data from a specified file.[16]. When reading from a file, `readline` returns either a line of data, or 0 when the end of the file is reached.

Although it is not strictly necessary, it is a good idea to explicitly `open` the file before dealing with it, and `close` it when you are done. In addition to being good practice (like explicitly declaring variables), it also allows you to specify certain options, like whether you intend to read or write to the file. In fact, I recommend using `fopen`, which also allows you to specify whether the file should be treated as text or binary.[17] Below is a small example of a procedure which encrypts a file one line at a time using a procedure `EncryptLine`. It should be clear how to modify it to use your favorite encryption routine instead.

[16] If you are treating a newline as a character, you will need to deal with the file on a byte-by-byte basis, using `readbytes` and `writebytes`

[17] On several operating systems, including MS-Windows and MacOS (but not Unix), files may be treated as containing text or binary data. When dealing with a text file, some control characters (such as the end-of-line marker) are translated to a format more suitable for use. When opening the file, we can tell maple whether it should be dealt with as text or binary. If you neglect to specify which, you may find your file has one very long line if you open it in a text editor. Under Unix, no translation is necessary, but it doesn't hurt.

```
> Cryptfile := proc(plainfilename,cryptfilename,key)
    local line, cline, file, cryptf;

    file  :=fopen(plainfilename,READ,TEXT);
    cryptf:=fopen(cryptfilename,WRITE,TEXT);

    line :=readline(file);
    while (line <> 0) do
       writeline(cryptf,EncryptLine(line,key));
       line :=readline(file);
    od;
    close(file);
    close(cryptf);
  end:
```

3.7 Affine enciphering

We have just examined a trivial encryption scheme, the Caesar cipher, which can be described as applying $x \mapsto x + b \pmod{p}$ to each of the characters x in a message. As we have seen, this provides no security at all (a message encoded in this way can be cracked in a short while by a third grader), but minor modifications such as those in §3.5 greatly improve the security.

We now return to trivialities, but make it one step harder. Instead of encoding each character by just adding a constant, we can multiply by a constant as well. That is, we can use $x \mapsto ax + b \pmod{p}$ to encode each character. This not only helps a tiny bit (to crack it, you need to find out *two* characters instead of just one, and the encoded message looks a little more "mixed up", since letters adjacent in the alphabet no longer encode to adjacent letters), it helps us set up a little more conceptual machinery.

3.7.1 When do affine encodings fail?

You may wonder, however, will this work? Certainly when we were just shifting the letters by a fixed amount, there was no chance that the enciphering transformation f could fail to be invertible. Does the same hold true if we multiply the numeric codes for the letters? That is, do different letters always encode to different letters?

The answer is, unfortunately, not always. For example, suppose we use a 27 character alphabet (A-Z and blank). Then if we encode using the transformation $x \mapsto 6x + 2 \pmod{27}$, both "A"(0) and "J"(9) encode to the character "C"(2). What went wrong? Recall that when we work modulo 27, any numbers that differ by a multiple of 27 are equivalent. Since 27 has nontrivial factors, several characters will collide whenever the multiplicand (6) shares a common divisor with 27. This will never happen if $\gcd(a, p) = 1$.

Let's make that precise:

Proposition 3.7.1 *Let a, b, x, and y be integers in the set $\{0, 1, \ldots, p-1\}$, with $\gcd(a, p) = 1$. Then*

$$ax + b \equiv ay + b \ (mod p) \quad \Longleftrightarrow \quad x \equiv y \ (mod p)$$

Proof. Suppose $ax + b \equiv ay + b \pmod{p}$. We want to show that $x \equiv y \pmod{p}$. Certainly we have $ax \equiv ay \pmod{p}$, so there must be integers k and m so that $kp + ax = mp + ay$. But we can then rewrite this as

$$a(x - y) = (k - m)p$$

that is, $a(x - y)$ is a multiple of p. If a and p share no common divisors, then $x - y$ must be a multiple of p, so $x \equiv y \pmod{p}$.

If $x \equiv y \pmod{p}$, then certainly $ax + b \equiv ay + b \pmod{p}$. □

Thus, by the above, we can see that the affine encoding scheme will be just fine if the multiplicand a is relatively prime to the length of our alphabet (p). Henceforth, we will always take an alphabet whose length is a prime number, so we need not worry about this issue. An alphabet of length 97 works quite well, because we can include all the usual printable ASCII characters (see §3.2.2, especially the table on page 56). The keyboard I am typing this on produces 96 distinct characters (including space, tab, and newline), without resorting to control keys. If you plan to use the full ASCII or extended ASCII character set (including the null character), then you will need to make sure you choose a to be an odd number.

3.7.2 Implementing and using an affine encoding

Implementing this scheme is a trivial modification to the `Caesar2` procedure we wrote in §3.4. We only need add another parameter `a`, and change the encoding calculation. (We also renamed `shift` as `b` to correspond to our discussion here).

```
> AffineCode:= proc(plaintext, a, b)
    local textnum,codenum,i,p;
    global Alphabet;

    p       := length(Alphabet);
    textnum := ToNum(plaintext);
    codenum := [seq( modp(a*textnum[i]+b, p), i=1..length(plaintext)) ];
    FromNum(codenum);
  end:
```

What about decoding a message encoded by an affine cipher? If we know the original `a` and `b`, we could either write `AffineDecode`, which calculates `modp( (textnum[i]-b)/a, p)` instead, or, we could use `AffineCode(1/a, -b/a)`, which is really the same thing.

If you think about it for a second, you may wonder whether $1/a \bmod p$ makes any sense at all. It does, but only if we interpret division in the right way. We should not divide by a in the "ordinary way" and then reduce modulo p, but instead interpret it to mean the solution x to

the equation $ax \equiv 1 \pmod{p}$. That is, $1/a \bmod p$ is the multiplicative inverse of a in the group $0, 1, \ldots, p-1$. (As before, we can run into trouble here if a and p are not relatively prime.) Fortunately for us, maple knows about division modulo p:

```
> 1/6 mod 97;
```

$$81$$

3.7.3 Breaking an affine cipher

Suppose we have a message that was encrypted with an affine cipher on an alphabet we also know. Suppose also that we know (or manage to guess) the encoding of two letters. How do we then decipher the message? It is quite straightforward: since we know the message was enciphered with $x \mapsto y = (ax + b) \bmod p$, and we know two examples of x and y, we need only solve a pair of linear equations $\bmod p$. This is quite straightforward to do by hand, but maple will also do it for us.

For example, suppose we receive the message

nrIbQyxEbkrI bwr EUbJaPybL abpyJJy UbUlrrxU

which we know is encoded in the 53 letter alphabet consisting of the letters A–Z, a blank, then a–z. Suppose we also guess that the letter b (28) is the encoding for a space (26), and that U (20) is an encoded s (45) . We can figure out the encoding transformation by solving the pair of equations

$$26a + b \equiv 28 \pmod{53} \qquad 45a + b \equiv 20 \pmod{53}.$$

We can ask maple to do this for us using `msolve`:

```
> msolve({a*26 + b = 28, a*45 +b = 20},53);
```

$$\{b = 25,\ a = 47\}$$

but, it will be more convenient to solve the *deciphering* transformation instead:

```
> msolve({A*28 + B = 26, A*20 +B = 45},53);
```

$$\{A = 44,\ B = 13\}$$

```
> AffineCode(‘nrIbQyxEbkrI bwr EUbJaPybL abpyJJy UbUlrrxU‘,44,13);
```

You bend your words like Uri Gellers spoons

3.8 Enciphering matrices

3.8.1 Treating text as vectors

We can let our enciphering transformation f operate on larger blocks of text than just a single character at a time. However, instead of treating a multi-byte string as a "super-character", as in §3.5.3, we could think of it as a vector of characters. Thus, if we were using, say, 2-vectors, then the name "Dougal" might become the three vectors [36, 79], [85, 71] and [65, 76]. If the length of the text we were converting was not divisible by 2, we would have to extend it by

adding a space. Thus the name "Ian" occurring at the end of a message would become the pair of vectors [41, 65] and [78, 0].

Below is an implementation of this conversion, `TextToVects`, which takes as input the text to convert to n-vectors and the length of the vectors, n. It uses the `ToNum` procedure from §3.4, as well as the usual maple commands.

```
> TextToVects:= proc(text,n)
    local i,j,textnums,vectors;

    textnums := ToNum(cat(text, seq(` `,
                                i=1..modp(n-length(text),n))));
    [seq([seq(textnums[(j-1)*n + i], i=1..n)],
          j=1..nops(textnums)/n)];
  end:
```

The first thing it does is extend the length of the text to a multiple of `n`, if needed. This relies on the observation that $k + ((n-k) \bmod n)$ is a multiple of n, and on the fact that maple will do the right thing with a command like `seq(i,i=1..0)`, that is, produce nothing whatsoever. Then the text, extended by the correct number of blanks, is converted to a list of numbers.

We now parcel up the list of numbers `textnums` into chunks of size `n`. This is done by noting that the characters in the `j`-th vector will come from positions $n(j-1)+1$ through $n(j-1)+n$, so the inner `seq` gives us the `j`-th vector, as the outer one ranges over all the vectors. If you don't see how this works right off, just take a moment to write out an explicit example for yourself.

Converting back from a list of n-vectors to a character string is easy, because we already have `FromNum` to convert a list of numbers to a character string for us. All we need to do to change a list of lists to just a list is discard the inner sets of braces. We can use `map` and `op` to do this for us:

```
> map(op,[[36, 79], [85, 71], [65, 76]]);
```

$$[36, 79, 85, 71, 65, 76]$$

One small problem is that maple distinguishes between a vector and a list of numbers— a vector has a little more structure. While we can use the two almost interchangeably (in the same way that we can think of a point in $\mathbb{R}^n$ as an n-vector, or of as an n-vector as a point), some functions such as `op` that deal with the internal structure will give different results. So, to be sure that `op` will work, we need to convert the vectors to lists first. The result is

```
> VectsToText:= vects -> FromNum(map(op, map(convert,vects,list))):
```

3.8.2 Affine encoding with matrices

Now that we know how to think of character strings as vectors, we can manipulate them. Our enciphering transformation is just a permutation of the n-vectors.

For example, the Vignère cipher (§3.5.1) can be interpreted as the applying the map $v \mapsto v + b \bmod p$, where v and b are vectors the length of the key.

We can do a more general trick using the affine transformation

$$v \mapsto Av + b \bmod p$$

, where A is an $n \times n$ matrix, and b and v are n-vectors. As before, we need certain restrictions on the matrix A. In particular, we need it to be an invertible matrix (so that we can decipher the encoding). This is guaranteed by requiring that the determinant of A be nonzero and share no common factors with p. As before, choosing an alphabet of prime length makes our lives much easier. The linear version of this encoding, $v \mapsto Av$, is sometimes called a "Hill cipher".

In maple, one does matrix arithmetic pretty much as you would expect, except that you need to use `&*` to indicate matrix multiplication. This is because matrix multiplication is not commutative, while `*` is. Also, maple doesn't carry out the operations until you execute `evalm`. Thus, to compute $Av + b$, we would write `evalm(A &* v + b)`.

```
> AffineMatEncode := proc(text,A,B)
  local   vtext,vcrypt,i,n,p;
  global  Alphabet;

  p:=length(Alphabet);
  n:=nops(convert(B,list));

  vtext := TextToVects(text, n);
  vcrypt:= [seq( map(modp,
                     evalm(  A &* vtext[i]  + B),
                     p),
                 i=1..nops(vtext))];
  VectsToText(vcrypt);
 end:
```

Note that matricies allow the encodings of nearby characters in the message to interact. For example, we can exchange every other character in a message using the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$:

```
> AffineMatEncode('I'm so mixed up!', [[0,1],[1,0]], [0,0]);
```

'I mosm xideu !p

To decode a message encoded using A, b, we can encode the crypttext with the matrix A^{-1} and the shift vector $-A^{-1}b$. You should check for yourself why this is the proper transformation. To obtain the inverse matrix in maple, you can use `inverse`. This is part of the linear algebra package, so you must either first issue the command `with(linalg,inverse)` or refer to the function as `linalg[inverse]`.

```
> A := matrix([[67, 23], [69, 81]]); B:=[5, 47];
```

$$A := \begin{bmatrix} 67 & 23 \\ 69 & 81 \end{bmatrix}$$

$$B := [5, 47]$$

```
> AffineMatEncode(‘Lester Hill‘,A, B);
```

nG=a/$TvPuUU

```
> AffineMatEncode(‘nG=a/$TvPuUU‘, inverse(A), evalm(-inverse(A) &* B));
```

Lester Hill

3.8.3 A Known-plaintext attack on an affine matrix cipher

As in §3.7.3, we can determine the key if we know what certain characters correspond to. If the message is encoded using n-vectors, we will need to know the decodings for $n+1$ vectors, that is, $n(n+1)$ characters. We then solve the $n+1$ vector equations simultaneously to find out the deciphering matrix and shift vector.

We will give a small example: Suppose we receive a message that we know was encoded using an affine cipher on 2-vectors, in our usual 97-letter alphabet. Suppose we also know that the plaintext "secret" corresponds to the crypttext "oHyeAH". This is sufficient information to decipher the message, because we know the encodings of three different 2-vectors.

```
> plain:=TextToVects(‘secret‘,2);
```

$$plain := [[83, 69], [67, 82], [69, 84]]$$

```
> crypt:=TextToVects(‘oHyeAH‘,2);
```

$$crypt := [[79, 40], [89, 69], [33, 40]]$$

These three vectors give us three equations to solve:

$$Ac_1 + B \equiv p_1 \qquad Ac_2 + B \equiv p_2 \qquad Ac_3 + B \equiv p_3$$

where the c_i are the crypttext vectors and the p_i are the corresponding plaintext, and the equations are taken mod 97. (If this seems backwards, remember we are trying to find the deciphering transformation.) If the code were linear, then the deciphering matrix would be PC^{-1}, where P and C are the matrices whose columns are made up of the vectors p_i and c_i respectively.[18]

Since the code is affine, we can either play around manipulating matrices, or we can just translate everything into 6 linear equations in 6 variables. We will take the latter approach.

```
> A:=matrix([[a,b],[c,d]]);
  B:=[e,f];
```

$$A := \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$B := [e, f]$$

[18] You probably want to check that this works for yourself.

Note that we wish to solve $Ac_i + B \equiv p_i \pmod{97}$ for A and B. If we execute the statement `evalm(A &* crypt[i] + B - plain[i])`, we will get a vector which should be the zero vector. For example, using the first sample, we obtain

```
> evalm( A&*crypt[1] + B - plain[1]);
```

$$[79\,a + 40\,b - 83 + e,\ 79\,c + 40\,d - 69 + f]$$

This gives us a vector of two expressions we would like to set to zero. If we do this for all three of our samples, and unwrap the vectors using `op` and `convert( ,list)`, we will get a system of equations appropriate to solve:

```
> eqns:= {seq(op(convert(
                    evalm(A&*crypt[i] + B - plain[i]),
                    list)), i=1..3)};
```

$$eqns := \{33\,a + 40\,b - 69 + e,\ 33\,c + 40\,d - 84 + f,\ 79\,a + 40\,b - 83 + e,$$
$$79\,c + 40\,d - 69 + f,\ 89\,a + 69\,b - 67 + e,\ 89\,c + 69\,d - 82 + f\}$$

Now, we use `msolve` to find out the decoding matrix and shift vector.

```
> Answer:= subs(msolve(eqns,97),[evalm(A),B]);
```

$$Answer := [\begin{bmatrix} 72 & 85 \\ 6 & 82 \end{bmatrix}, [16,\ 1]]$$

We can decode our message now, by encoding the crypttext using the matrix A and shift B found above. A is the inverse of the matrix used for encoding, and B is A times the negative of the original shift vector.

3.9 Modern cryptography

3.9.1 Secure cryptosystems

As mentioned before, most of the ciphers we have examined are not really cryptographically secure, although they may have been adequate prior to the widespread availability of computers. A cryptosystem is called *secure*, if a good cryptanalyst, armed with knowledge the details of the cipher (but not the key used), would require a prohibitively large amount of computation to decipher a plaintext. At first, this may seem like an unreasonably strong definition; you may be thinking that, given a mass of encrypted data, how is the cryptanalyst to know *how* it was encrypted? If you are encrypting your own files, for personal use, this is probably so. But today, most encryption is done by software or hardware that the user did not produce himself. One can reverse engineer a piece of software (or hardware) and make the cryptographic algorithm apparent, so we cannot rely on the secrecy of the method as a good measure of security.

There are several widely used ciphers which are believed to be fairly secure. We will not go into the details of any of these; most of the algorithms are quite involved and require manipulations at the bit level— maple is not really the appropriate tool for them. Implementations

are widely available, both commercially and as free public software (see, for example, the International Cryptography page[19]).

Probably the most commonly used cipher today is DES (the Data Encryption Standard), which was developed in the 1970s and has been adopted as a standard by the US government. The standard implementation of DES operates on 64-bit blocks (that is, it uses an alphabet of length 2^{64}— each "character" is 8 bytes long), and uses a 56-bit key. Unfortunately, the 56-bit key means that DES, while secure enough to thwart the casual cryptanalyst, is attackable with special hardware by governments, major corporations, and probably well-heeled criminal organizations. One merely needs to try a large number of the possible 2^{56} keys.[20] This is a formidable, but not insurmountable, computational effort. A common variation of DES, called Triple-DES, uses three rounds of regular DES with three different keys (so the key length is effectively 168 bits), and is considerably more secure.

Other commonly used ciphers are IDEA (the International Data Encryption Algorithm, developed at the ETH Zurich in Switzerland), which uses 128-bit keys, and Blowfish (developed by Bruce Schneier), which uses variable length keys of up to 448 bits. Both of these are currently believed to be secure. Another common cipher, RC4 (developed by RSA Data Security) can use variable length keys and, with sufficiently long keys, is believed to be secure. Some versions the Netscape web browser used RC4 with 40 bit keys for secure communications. A single encrypted session was broken during 1995 in about 8 days using 112 networked computers (see Damien Doligez's web page about cracking SSL[21] for more details). Even though this algorithm has been "cracked", it seems that this scheme is certainly good enough for everyday use considering the resources required to decrypt a single session. In any case, current web browsers offer much more security— a 40-bit key is the weakest offered.

There are, of course, many other ciphers in common use, and more being developed all the time.

3.9.2 Message digests

Related to encryption are "Message Digests", or cryptographic hash functions. A message digest takes an arbitrary length string and computes a number, or hash, from it. Changing the string changes the value of the hash. Hashing functions also play a major role in searching and sorting, and hence in computer databases. While it is obvious that several plaintexts must correspond to the same hash value, it is a difficult problem in general to construct such a plaintext from the value of the hash alone. Thus, one can use a good message digest to *authenticate* a message— appending an encrypted version of the hash to an otherwise clear text message can act as a seal, showing that the message was not otherwise tampered with, since it would be infeasably hard to construct a message with the same hash.

[19] http://www.cs.hut.fi/ssh/crypto/
[20] that is, 72,057,594,037,927,936 keys.
[21] http://pauillac.inria.fr/ doligez/ssl/

Some commonly used Message digests are MD5 (developed by RSA Data Security), and SHA or SHS (the Secure Hash Algorithm/Standard, developed by the US government).

3.9.3 Public Key cryptography

All of the ciphers so far discussed are symmetric encryption routines (also called "secret-key" or "traditional" ciphers): if one knows the key used to encode the message, one also can decrypt the message without too much work. For example, in an affine cipher (§3.7) $x \mapsto ax + b \bmod p$, if we know the value of a and b to used encode the message, we can quickly compute the deciphering key $1/a \bmod p, -b/a \bmod p$. Also, decrypting the message usually also yields the key.[22]

This was true of all cryptosystems up until the mid 1970s: knowledge of how to encode a message allowed one also to decipher it. However, in 1976 W. Diffie and M. Hellman invented "public key cryptography". In a public key system, someone who knows how to encipher a message cannot determine how to decipher the message without a prohibitively large computation. That is, given the enciphering transformation $f_{key} : \mathcal{P} \mapsto \mathcal{C}$, discovering f_{key}^{-1} is not computationally realistic. It is important to realize that there may be a procedure for doing so, but to carry out this process would take a prohibitively long time (say, hundreds of years on the fastest known computers).

What good is a public key cryptosystem? Since knowledge of the encoding key K_E does not give information about the decoding key K_D, the encoding key K_E can be made public (hence the name "public key"). This allows anyone to encode messages that only the recipient can decode. For example, suppose you want to make a credit card purchase over the Internet. Data sent across the Internet unencrypted is not secure: someone with access to the transmission lines can listen in and capture sensitive data.[23] However, if the merchant provides a public key,

[22] A story related to this appears in Casanova's memoirs (1757). He was given an encrypted manuscript by a Madame d'Urfé, which he deciphered. Later, upon learning this, Madame d'Urfé was incredulous, believing he must know the keyword to do so.

> I then told her the key-word, which belonged to no language, and I saw her surprise. She told me that it was impossible, for she believed herself the only possessor of that word which she kept in her memory and which she had never written down.
>
> I could have told her the truth— that the same calculation which had served me for deciphering the manuscript had enabled me to learn the word— but on a caprice it struck me to tell her that a genie had revealed it to me. This false disclosure fettered Madame d'Urfé to me. That day, I became the master of her soul, and I abused my power. Every time I think of it, I am distressed and ashamed, and I do penance now in the obligation under which I place myself of telling the truth in writing my memoirs.

[23] This is not as hard as you might expect. In many places, such as universities, portions of the network backbone pass through public places like dormitories and classrooms. One merely needs to attach a computer to the network lines to "sniff" the network traffic. This will, of course, be limited to users communicating to or from that location, but it is still quite a lot. Also, there are many segments of the Internet which consist of microwave links. To listen in on a huge part of network traffic, all you need is an appropriately placed microwave

you can encrypt your credit card number and transmit it without worry. Only the merchant can decrypt the message, even though anyone may send one.[24]

Another feature of public key cryptography is "authentication". In many public key systems, either the enciphering key K_E or the deciphering key K_D may be made public without revealing the other one. In order to "digitally sign" a message, I could append my name encrypted with my enciphering key, which I keep secret. I could also make public the deciphering key K_D. Then anyone can check that I was the actual sender, because only I could have encoded my name. Of course, to stop someone from merely appending a valid encrypted signature to a forged message, I should also encrypt something specific to that message, such as a message digest (see §3.9.2) computed on its contents. This feature is commonly used to "digitally sign" email by software such as PGP[25].

A common public key system in current use is RSA (named for its inventors: Rivest, Shamir and Adelman), which we will look at in detail in §3.11. Others are the Diffie-Hellman key exchange system, El Gamal, Subset-sum (or Knapsack) systems, and systems based on elliptic curves. We will not discuss these here.

3.10 Some Number Theory

Most public key systems rely on number-theoretic results. Before we can discuss the implementation of one, we need to quickly go over the necessary background. We have already used a tiny amount of number theory (in our discussion of computing $\bmod p$ and of the greatest common divisor). Of course, this must be done briefly, and we will only touch on a small part of a large and ancient field— the interested reader would do well to consult a text on number theory for more information.

3.10.1 The greatest common divisor and the Euclidean algorithm

We have already met the greatest common divisor, or gcd, which is the largest integer which divides both of a pair of numbers. Two numbers are said to be *relatively prime* if their greatest common divisor is 1. As we have already seen, finding two relatively prime numbers has important applications in many cryptosystems.

How can we determine the gcd of two numbers? If the numbers are not too large, just looking at their factors does the trick. For example,

$$\gcd(138, 126) = 6 \quad \text{since} \quad 138 = 2 \cdot 3 \cdot 23 \quad \text{and} \quad 126 = 2 \cdot 3^2 \cdot 7$$

receiver.

[24] Of course, you should still only deal with reputable merchants. You probably wouldn't give your credit card number to a guy selling stuff on the street; neither should you automatically trust anyone with a web site.

[25] http://www.pgpi.com

However, there must be a more efficient way, since maple can calculate the gcd of two 60-digit numbers in well under a second, while taking a long time (about 30 minutes on a 133 MHZ pentium) to factor either one of them.

One such algorithm is the Euclidean algorithm, which has been around for thousands of years. It works by computing successive differences— to compute $\gcd(a, b)$, where $a > b$, we first compute $r_1 = a - k_1 b$, where $k_1 b$ is the largest multiple of b that is less than a. Then we repeat this, finding $r_2 = b - k_2 r_1$, $r_3 = r_1 - k_3 r_2$, and so on until the remainder is zero. The last nonzero remainder is the gcd. For example, to calculate $\gcd(138, 126)$,

$$\begin{array}{rrcr} a & & = & 138 \\ & b & = & 126 \\ a & -b & = & 12 \\ -10a & +11b & = & 6 \\ 21a & -23b & = & 0 \end{array}$$

This not only gives $\gcd(138, 126)$, but expresses it as a difference of multiples of the two numbers: $6 = 11 \cdot 126 - 10 \cdot 138$. We can write this procedure more formally as a theorem.

Theorem 3.10.1 (The Euclidean Algorithm) *Let a and b be two positive integers, with $b < a$. Then we can construct two decreasing sequences of positive integers r_i and k_i as follows:*

$$\begin{array}{lcccll} k_1 b & + & r_1 & = & a & 0 < r_1 < b \\ k_2 r_1 & + & r_2 & = & b & 0 < r_2 < r_1 \\ k_3 r_2 & + & r_3 & = & r_1 & 0 < r_3 < r_2 \\ k_4 r_3 & + & r_4 & = & r_2 & 0 < r_4 < r_3 \\ & & & \vdots & & \\ k_n r_{n-1} & + & r_n & = & r_{n-2} & 0 < r_n < r_{n-1} \\ k_{n+1} r_n & & & = & r_{n-1} & \end{array}$$

and r_n is the greatest common divisor of a and b.

We have already seen that the Euclidean algorithm also gives us the following:

Corollary 3.10.2 *Let a and b be two positive integers. Then there are integers x and y so that $ax + by = \gcd(a, b)$.*

Before proving Thm. 3.10.1, we will need a lemma.

Lemma 3.10.3 *Let a be a positive integer, and b be a nonnegative integer. If*

$$b = aq + r, \qquad \text{with } q \text{ and } r \text{ positive integers}$$

then $\gcd(a, b) = \gcd(a, r)$.

Proof. Let $d = \gcd(a, b)$. Then since d is a divisor of a and a divisor of b, it is also a divisor of $b - aq$. Since $r = b - aq$, we have d a divisor of r. So it must divide $\gcd(a, r)$.

Now, in order to prove equality, we want to show that also $\gcd(a, r)$ is a divisor of d. Certainly $\gcd(a, r)$ divides b, since $b = r + aq$. It also divides a, and so we have $\gcd(a, r)$ as a divisor of $\gcd(a, b)$.

Thus, $\gcd(a, r) = \gcd(a, b)$. □

Now we are ready to prove Thm. 3.10.1:

Proof. Notice that the sequence $b > r_1 > r_2 > r_3 > \ldots$ is a strictly decreasing sequence of positive integers, so it must eventually terminate with some $r_n > 0$. The last equation, $k_{n+1}r_n = r_{n-1}$ says that r_n is a divisor of r_{n-1}, so certainly $r_n = \gcd(r_n, r_{n-1})$. Applying the lemma to the equation above it gives $\gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2})$. We can construct the following chain in this way:

$$r_n = \gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2}) = \ldots = \gcd(r_1, b) = \gcd(a, b)$$

□

Recall that if r is such that $ar = 1$, then r is called the multiplicative inverse of a. We have already dealt with such objects extensively. We now state a theorem which tells us exactly when such inverses exist. Note that this is really just Prop. 3.7.1 slightly reworded.

Theorem 3.10.4 *Let a and n be integers, with $n \geq 2$. Then a has a multiplicative inverse modulo n if and only if* $\gcd(a, n) = 1$.

Proof. First, suppose that a has an inverse modulo n. Then there is a k so that

$$ak \equiv 1 \bmod n.$$

In particular, n is a divisor of $ak - 1$, so there must be some integer t so that $ak - 1 = nt$. Rewriting this as

$$ak - nt = 1,$$

we see that $\gcd(a, n) = 1$.

Now suppose that $\gcd(a, n) = 1$. Then by Cor. 3.10.2, there are integers r and s with $ar + ns = 1$. This means $ar - 1$ is divisible by n, so

$$ar \equiv 1 \bmod n.$$

But this says that r is the inverse of $a \bmod n$. □

Notation. We haven't yet used this notation, but it is common to denote the set of integers modulo n as $\mathbb{Z}_n$. That is,

$$\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$$

As we have already know, this set is closed under addition and multiplication, and the usual associative, distributive, and commutativity laws hold. In fact, this set forms a *ring*.

Notation. We will use $\mathbb{Z}_n^*$ to denote the elements of $\mathbb{Z}_n$ which that have multiplicative inverses. In light of the last theorem, we have

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}$$

3.10.2 The Chinese Remainder Theorem

In this section, we state a very old theorem about simultaneous solutions to congruences. This theorem may have been known to the eight-century Buddhist monk I-Hsing, and certainly appears in Ch'in Chiu-shao's *Mathematical Treatise in Nine Sections*, written in 1247. It is sometimes (rarely) referred to as the "Formosa Theorem".

Theorem 3.10.5 *Let $m \geq 2$ and $n \geq 2$ be integers with $\gcd(m, n) = 1$. Then for any integers a and b, the system of equations*

$$\begin{aligned} x &\equiv a \mod m \\ x &\equiv b \mod n \end{aligned}$$

has a solution. Furthermore, there is only one solution x between 0 and $mn - 1$.

Proof. Since m and n are relatively prime, there are integers k and t so that $mk + nt = 1$. Then

$$c = bkm + ant$$

is the desired solution. To verify this, we need only check:

$$c \equiv ant \equiv a(1 - mk) \equiv a \mod m$$

and

$$c \equiv bkm \equiv b(1 - nt) \equiv b \mod n.$$

To check uniqueness, suppose there are two solutions c and d. Since $c \equiv a \bmod m$ and $d \equiv a \bmod m$. We have $c - d \equiv 0 \bmod m$ and $c - d \equiv 0 \bmod n$. Since both m and n divide $c - d$, and m and n are relatively prime, we have that mn divides $c - d$. Thus $c - d \equiv 0 \bmod mn$, that is, $c \equiv d \bmod mn$.

3.10.3 Powers modulo n

Notation. In this and future sections, we shall use the notation $[\![a]\!]_b$ to denote $a \bmod b$.

First, let's take a look at a few examples of what happens when we reduce the sequence $a, a^2, a^3, a^4, \ldots$ modulo n. Consider $[\![3^k]\!]_{20}$:

$$3,\ [\![3^2]\!]_{20} = 9,\ [\![3^3]\!]_{20} = 7,\ [\![3^4]\!]_{20} = [\![21]\!]_{20} = 1,\ [\![3^5]\!]_{20} = 3, \ldots$$

The sequence consists of 4 numbers $3, 9, 7, 1$ and then it repeats. Now consider the powers of 11:

$$11,\ [\![11^2]\!]_{20} = [\![121]\!]_{20} = 1,\ [\![11^3]\!]_{20} = 11, \ldots$$

This sequence is periodic of period 2. What about 4? We have

$$4,\ [\![4^2]\!]_{20} = 16,\ [\![4^3]\!]_{20} = 4,\ [\![4^4]\!]_{20} = 16, \ldots$$

This is periodic, but the sequence does not contain 1. Finally, let us consider the powers of 6:

$$6,\ [\![6^2]\!]_{20} = [\![36]\!]_{20} = 16,\ [\![6^3]\!]_{20} = [\![16 \cdot 6]\!]_{20} = [\![96]\!]_{20} = 16,\ [\![6^4]\!]_{20} = 16, \ldots$$

The first two examples give rise to the following definition:

Definition 3.10.6 *If there is a positive integer k so that $[\![a^k]\!]_n = 1$, we say that a has* finite multiplicative order *modulo n. The smallest such integer k is called the* order of *a modulo n.*

We can readily categorize the elements with finite order. If you look at the examples above, 3 and 11 have finite order mod 20, while 4 and 6 do not. You probably have already guessed that this is because $\gcd(3, 20) = 1$ and $\gcd(11, 20) = 1$, while $\gcd(4, 20) = 4$ and $\gcd(6, 20) = 2$.

Theorem 3.10.7 *Let a and $n > 2$ be integers. Then a has finite order modulo n if and only if* $\gcd(a, n) = 1$.

Proof. First, suppose a has finite order mod n. Then there is some k so that

$$1 = [\![a^k]\!]_n = [\![a]\!]_n \cdot [\![a^{k-1}]\!]_n.$$

Thus, a has a multiplicative inverse, namely $[\![a^{k-1}]\!]_n$. By Thm. 3.10.4, this means $\gcd(a, n) = 1$.

Now we suppose $\gcd(a, n) = 1$, and we want to find the inverse of a. Consider the sequence

$$a, [\![a^2]\!]_n, [\![a^3]\!]_n, \ldots, [\![a^n]\!]_n, [\![a^{n+1}]\!]_n.$$

Since there are $n + 1$ elements and we are reducing modulo n, there must be at least two which are equal. Let's denote these by $[\![a^k]\!]_n$ and $[\![a^t]\!]_n$, with $1 \leq t < k$. Since

$$[\![a^k]\!]_n = [\![a^t]\!]_n$$

we have

$$\left[\!\left[a^{k-t}\right]\!\right]_n = 1.$$

Thus, a has order $k-t$ mod n. □

So, we have seen that any invertible element of $\mathbb{Z}_n$ has finite order. In particular, if p is a prime number, every element except 0 is invertible and so has finite order. What are the possible orders? A theorem of Pierre de Fermat gives a result in that direction.

Theorem 3.10.8 (Fermat's Little Theorem, 1640) *Let p be a prime. If* $\gcd(a,p) = 1$, *then*

$$a^{p-1} \equiv 1 \ mod p.$$

For any integer a,

$$a^p \equiv a \ mod p.$$

Among other things, this theorem says that if p is prime, then the order of a mod p always divides $p-1$. Note that it doesn't say the order *is* $p-1$, just that it is a divisor of it. Also, it says nothing about order with modulo a non-prime. We won't give the proof of this theorem here. Rather, we will give a proof of a more general result in the next section.

3.10.4 The Euler φ-function and Euler's Theorem

Recall that we denote the set of invertible elements of $\mathbb{Z}_n$ by $\mathbb{Z}_n^*$. That is,

$$\mathbb{Z}_n^* = \{a \in Z_n \mid \gcd(a,n) = 1\}.$$

This set is sometimes called a *reduced residue system mod n.*

Definition 3.10.9 *Let n be a positive integer. Then $\varphi(n)$ is the number of elements in $\mathbb{Z}_n^*$, that is, the number of positive integers less than n which are relatively prime to n. This function is called the* Euler φ-function,[26] *or sometimes Euler's totient function.*

We can compute a few values of $\varphi(n)$ just by counting.

$$\begin{aligned}
\varphi(2) &= 1 \quad \text{since} \quad \mathbb{Z}_2^* = \{1\} \\
\varphi(3) &= 2 \quad \text{since} \quad \mathbb{Z}_3^* = \{1,2\} \\
\varphi(4) &= 2 \quad \text{since} \quad \mathbb{Z}_4^* = \{1,3\} \\
\varphi(5) &= 4 \quad \text{since} \quad \mathbb{Z}_5^* = \{1,2,3,4\} \\
\varphi(6) &= 2 \quad \text{since} \quad \mathbb{Z}_6^* = \{1,5\} \\
\varphi(7) &= 6 \quad \text{since} \quad \mathbb{Z}_7^* = \{1,2,3,4,5,6\} \\
\varphi(8) &= 4 \quad \text{since} \quad \mathbb{Z}_8^* = \{1,3,5,7\} \\
\varphi(9) &= 6 \quad \text{since} \quad \mathbb{Z}_9^* = \{1,2,4,5,7,8\} \\
\varphi(10) &= 4 \quad \text{since} \quad \mathbb{Z}_{10}^* = \{1,3,7,9\} \\
\varphi(11) &= 10 \quad \text{since} \quad \mathbb{Z}_{11}^* = \{1,2,3,4,5,6,7,8,9,10\} \\
\varphi(12) &= 4 \quad \text{since} \quad \mathbb{Z}_{10}^* = \{1,5,7,11\}
\end{aligned}$$

[26] Although the function was invented by Euler, the notation $\varphi(n)$ to represent it is actually due to Gauss.

Of course, maple also knows about the function φ in the `numtheory` package, so we could generate more values by asking maple. But let's try to understand a bit more. We can readily prove some properties of φ.

Proposition 3.10.10 *Let $p \geq 2$ be a prime. Then*

a. $\varphi(p) = p - 1$.

b. *If n is a positive integer, then $\varphi(p^n) = p^n - p^{n-1}$.*

c. *If a and b are relatively prime, then $\varphi(ab) = \varphi(a)\varphi(b)$.*

Proof. **(a)** This is immediate, since all non-zero elements of $\{0, 1, \ldots, p-1\}$ are invertible.

(b) We prove this part just by counting. Since p is prime, the only numbers in $\mathbb{Z}_p$ which are not relatively prime to p^n are multiples of p. These are $0, p, 2p, 3p, \ldots, p^2, (p+1)p, \ldots, (p^{n-1}-1)p$. Note that $p^{n-1}p = p^n$, so there are exactly p^{n-1} multiples of p less than p^n (including 0). All the others are relatively prime to p— this gives the result.

(c) We need to show that the number of elements in $\mathbb{Z}_{ab}^*$ is the same as the product of the size of $\mathbb{Z}_a^*$ and $\mathbb{Z}_b^*$. We do that by showing that for each $t \in \mathbb{Z}_{ab}^*$, there is exactly one pair (r, s) with $r \in \mathbb{Z}_a^*$ and $s \in \mathbb{Z}_b^*$.

So, let $r \in \mathbb{Z}_a^*$ and $s \in \mathbb{Z}_b^*$. Then, by the Chinese Remainder Theorem (Thm. 3.10.5), there is an integer $t < ab$ with

$$t \equiv r \bmod a \qquad \text{and} \qquad t \equiv s \bmod b.$$

Since t is a solution of the first, we can find an integer k so that $r = t + ka$, so $\gcd(t, a) = \gcd(a, r) = 1$ by Lemma 3.10.3. Similarly, $\gcd(t, b) = 1$. Since t is relatively prime to both a and b, it is relatively prime to their product, so $\gcd(t, ab) = 1$. Thus $t \in \mathbb{Z}_{ab}^*$.

Now take $t \in \mathbb{Z}_{ab}^*$, and we must find a corresponding r and s. Let $r = \llbracket t \rrbracket_a$. Now, since $\gcd(t, ab) = 1$, certainly $\gcd(t, a) = 1$. Since $r \equiv t \bmod a$, there is an integer k so that $t = r + ka$, so we have $\gcd(r, a) = 1$. Thus $r \in \mathbb{Z}_a^*$. Similarly, we let $s = \llbracket t \rrbracket_b$, and we see that $s \in \mathbb{Z}_b^*$.

Since we have paired each $t \in \mathbb{Z}_{ab}^*$ with a unique pair $(r, s) \in \mathbb{Z}_a^* \times Z_b^*$, they have the same cardinality. The first set has $\varphi(ab)$ elements, and the latter has $\varphi(a)\varphi(b)$. □

We now come to Euler's generalization of Fermat's result. This result is the central idea underlying the RSA public key cryptosystem.

Theorem 3.10.11 (Euler, 1750) *Let a and n be relatively prime integers, with $n \geq 2$. Then*

$$\llbracket a^{\varphi(n)} \rrbracket_n = 1.$$

Proof. Let a be relatively prime to n, and consider the set

$$a\mathbb{Z}_n^* = \left\{ \left[\!\left[ab\right]\!\right]_n \,\middle|\, b \in \mathbb{Z}_n^* \right\}.$$

Since a and n are relatively prime, $\left[\!\left[a\right]\!\right]_n \in \mathbb{Z}_n^*$. Thus every element of $a\mathbb{Z}_n^*$ is in $\mathbb{Z}_n^*$. But also every element of $\mathbb{Z}_n^*$ is an element of $a\mathbb{Z}_n^*$, since $\left[\!\left[ab\right]\!\right]_n = \left[\!\left[ac\right]\!\right]_n$ if and only if $\left[\!\left[b\right]\!\right]_n = \left[\!\left[c\right]\!\right]_n$. This means that the sets $a\mathbb{Z}_n^*$ and $\mathbb{Z}_n^*$ are equal.

Now, multiply all the elements of $\mathbb{Z}_n^*$ together; call the result N. Then $\left[\!\left[N\right]\!\right]_n \in \mathbb{Z}_n^*$. If we multiply all the elements of $a\mathbb{Z}_n^*$ together, we get $a^{\varphi(n)}N$, and since the two sets are the same, we have

$$\left[\!\left[a^{\varphi(n)}N\right]\!\right]_n = \left[\!\left[N\right]\!\right]_n.$$

Thus,

$$\left[\!\left[a^{\varphi(n)}\right]\!\right]_n = 1.$$

□

3.11 The RSA Public key cryptosystem

In 1978, R. Rivest, A. Shamir, and L. Adelman published the description of the RSA public key system, which we will describe here. As discussed in §3.9.3, a public key system allows anyone to encode messages that only the intended recipient may decode. This algorithm relies on the fact that unless one knows some special facts about n, calculating the Euler φ-function $\varphi(n)$ is as hard as factoring n, which, for very large n (say, 200 digits), is very hard indeed.

To set up the system, we pick at random two large primes p and q, of about 100 digits each. We then compute the **base** n as the product of p and q. Note that $\varphi(n) = (p-1)(q-1)$. We also pick some other large number $a < \min(p,q)$, called the **exponent**, which is relatively prime to $\varphi(n)$. We make the numbers (n,a) public— these form the key needed for encoding. We also use the Euclidean algorithm to compute x which satisfies $ax - y\varphi(n) = 1$; that is, so that $\left[\!\left[ax\right]\!\right]_{\varphi(n)} = 1$. The number x is the part of the key we keep private.

To encode a message: The sender divides his message up into blocks of length k, and to each block assigns an integer β (for example, by treating the characters of the message as a k-graph, as in §3.5.3, with $0 \le \beta < \min(p,q)$. For each block of plaintext, the sender transmits $\left[\!\left[\beta^a\right]\!\right]_n = m$.

To decode the message: For each unit m of the crypttext received, the recipient computes $\left[\!\left[m^x\right]\!\right]_n$, which is the integer representing of a k-graph of the message. We can see that this is so, because

$$\left[\!\left[m^x\right]\!\right]_n = \left[\!\left[\beta^{ax}\right]\!\right]_n = \left[\!\left[\beta^{1+\varphi(n)y}\right]\!\right]_n = \left[\!\left[\beta\right]\!\right]_n \left[\!\left[\beta^{y\varphi(n)}\right]\!\right]_n = \left[\!\left[\beta\right]\!\right]_n \left[\!\left[(\beta^y)^{\varphi(n)}\right]\!\right]_n = \left[\!\left[\beta\right]\!\right]_n \cdot 1.$$

This relies on Euler's theorem, so that $\left[\!\left[\gamma^{\varphi(n)}\right]\!\right]_n = 1$, where $\gamma = \beta^y$. The requirement that β be less than both p and q was to ensure that $\gcd(\beta, n) = 1$, so that we can apply the theorem.

Note that the symmetry between a and x means that we can make either one public. Thus, RSA works just as well for authentication as it does for public key encryption.

3.11.1 Implementing RSA in Maple

Now that we have covered how the RSA system works in theory, it is fairly simple to implement it in maple.

3.11.2 Useful preliminaries

First, we define a procedure which allows us to read in a file, and convert it into a list of large integers. This is done using `readbytes` to obtain `chunksize` bytes from the file. Each byte is represented by its ASCII code[27] (that is, as an integer between 0 and 128), and each group of `chunksize` characters is viewed as a number expressed in base $128^{\texttt{chunksize}}$. We convert each of these numbers to decimal, as in §3.5.3. The result is a list of several rather large integers representing the contents of the file. The choice of `chunksize` is fairly arbitrary, provided you ensure that $128^{\texttt{chunksize}}$ is less than your chosen primes. Here we use 47, which gives us 99 digit numbers, because 128^{47} is a 99-digit number.

```
> ReadBignums:= proc(filename,chunksize)
    local file,chunk,i,p,numlist;

    p       := 128;
    numlist := [];
    file    := fopen(filename, READ, BINARY);
    chunk   := readbytes(file,chunksize);
    while (chunk <> 0) do
        numlist := [op(numlist),
                    sum( chunk[i]* p^(chunksize-i), i=1..nops(chunk))];
        chunk := readbytes(file,chunksize);
    od;
    close(file);
    RETURN(numlist);
  end:
```

For example, if we read in the contents of the file `/home/mat331/data/Extra` using 47-byte chunks, we obtain a list of 39 numbers of 99 digits each.

```
> data:=ReadBignums(`/home/mat331/data/Extra`,47):
> nops(data);
```


[27] We assume here that the data in the file is plain text. If it is expected that there may be extended ASCII codes in the file (or if the file contains "binary data", for example, if it is compiled computer code or an image), the variable `p` should be set to 256 instead.

```
> data[2];
```

$$882221479087147858856410399353400203005463343834450424221849495430681\backslash$$
$$46627358316365192956608760830695$$

It will also be useful for us to be able to convert our big integers back into text, so that we can read our message when we are done.[28]

```
> BignumToText:=proc(num,k)
    local i,p;
    p:=128;
    convert([seq(  iquo(modp(num,p^(k-i+1)),p^(k-i)), i=1..k)],'bytes');
  end:
> BignumToText(data[2],47);
```

g phrase, which was encoded using a affine
enco*

Basic Setup: choice of primes, base, exponent, and decoding exponent

We wish to choose two primes p and q of about 100 digits each "at random", and also a random exponent relatively prime to $\varphi(pq)$. At first, this may seem to be a very hard task: how do you know if an arbitrary 100-digit number is prime unless you try to factor it? However, there are a number of efficient ways to test if a number is prime without actually determining its factors. Maple's `isprime` function uses such methods. Thus, all we really need is to generate some random numbers of about 100 digits, then take the next prime bigger. We use Maple's `rand` to create two functions, `BigRand` and `MedRand`, which give us random numbers of around 100 digits and less than 90 digits, respectively. We also have to remember to use `randomize()` to ensure that we get a different random number each time (see the discussion of pseudo-random numbers in §3.5.2, page 64).

```
> BigRand := rand(10^100..10^105):
  MedRand := rand(50..10^90):
> readlib(randomize)();
```

892060192

Now we pick our primes p and q, set `phi` to be $\varphi(pq)$, and compute our base $n = pq$.

```
> p:=nextprime(BigRand());
  q:=nextprime(BigRand());
  phi:=(p-1)*(q-1);
  n:=p*q;
```

$$p := 168148344694840258566703067925059488643059382079284570104453791\backslash$$
$$3863119894063560618824877602199869749797 31$$

$$q := 759147509371863656660616161981274323356040723687316036705747045\backslash$$
$$9407203590662948347772517936413042598626 47$$

[28] Of course, if you anticipate 8-bit data in your input, the number `p` here will need changing, as well.

$$\phi := 12764939708008960576348156136037597347167169331860339503010769812699154114822845932599138076073100856701277725767099009433810307591820570871142464403355472591199517479954681219469395759024153764817168\\8434165580$$

$$n := 12764939708008960576348156136037597347167169331860339503010769812699154114822845932599138076073100856701370455352505679825333039514811204252342374413932132651880537563687384454316660848690127720203297\\9669007957$$

To choose our public exponent a, we pick a number at random and check if it is relatively prime to $\varphi(n)$. If not, we just pick another and try again. Usually, a suitable exponent will be found after only a few tries.

```
> a:=MedRand();
  while( gcd(a,phi) <> 1) do;
      a:= MedRand();
  od;
```

$$a := 23761842762862818026813705680848933572643894137172039301368795022049395663850363780640313 5$$

$$a := 31980669092497318435254308131511089664020984689213578921745876063910817576939119242719832 0$$

$$a := 31313498486294424311286178163628183559546522196716411592492267847892755233552248294501656 7$$

Finally, we choose x to be the inverse of a mod $\varphi(n)$.

```
> x:= modp( 1/a, phi);
```

$$x := 16445856393380707369282446076185129880820098375956674147069759642064503707641544915321985624002512316665365378468661955298331280883922977475968127108659879154551620218885868380952053297001254368234914\\293234343$$

Encoding and Decoding

We've already done all the hard work; now writing the actual encoding and decoding routines is easy. Lets write `RSAEncodeNum` which encodes a single large number β by computing $\beta^n \bmod n$.

```
> RSAEncodeNum := proc(bignum, a, n)
    modp( bignum &^ a, n);
  end:
> RSAEncodeNum(data[1],a,n);
```

```
650640533631773130566525918154403948990214682783764008970814724574 98\
    95141947823205579680281872956674957884890678666592759764003894389372\
    05499067528982324528262147573016171219176997233479109398560183821065\
    60214
```

We can easily write a one-liner to encode a file with RSA by using `map` to apply `RSAEncodeNum` to the result of `ReadBignums`.

```
> RSAEncodeFile := proc(file,n,a)
    map(RSAEncodeNum, ReadBignums(file,47), a, n)
  end:
> encoded := RSAEncodeFile(‘/home/mat331/data/Extra‘, a, n):
```

To check that it worked, we can decode the message we just encoded by using `RSAEncodeNum` with the inverse of `a` that we computed earlier, and then apply `BignumToText` so we can read the result.

```
> decoded := map(RSAEncodeNum, encoded, x, n):
> cat(op(map(BignumToText,decoded,47)));
```

Here is exercise #3.

[1.] Decode the following phrase, which was encoded using a affine encoding cipher on a 27 − letter alphabet (the letters a − z and a blank). Note that a = 0.

xgb dkjvmbkffcmtkrv m lkaerask mdbmwkfmezremzkmxrcm kkmfvkrx mxgvkmaskrvsc

Chapter 4

A turtle in a fractal garden

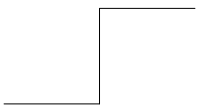
4.1 Turtle Graphics

Imagine you have a small turtle who responds to certain commands like "move forward a step", "move back a step", "turn right", and "turn left". Imagine also that this turtle carries a pen (or just leaves a trail of green slime wherever he crawls— the pond he came out of was none too clean). As long as our turtle follows our commands without tiring, we can get him to make rather intricate patterns easily.[1] The turtle graphics commands described here are not a standard part of maple. Rather, they are in a file "turtle.txt" which we will examine in more detail later in this chapter. Maple has a somewhat different implementation included in the share library. In order to get our turtle commands known to maple, we must first load them with a command such as

```
> read(`turtle.txt`);
```

Let us suppose our turtle is given a string such as "FLFRF" and interprets it as the sequence of instructions "Move Forward. Turn Left. Move Forward. Turn Right. Move Forward." Then we should see the turtle move in a zigzag pattern:

```
> TurtleCmd(`FLFRF`);
```



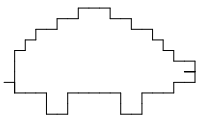
We shall begin assuming the turtle knows the following commands:

F — move forward one step	R — turn right (but don't move)
B — move backward one step	L — turn left

[1] This is the premise of the computer language LOGO, invented in the late 1960s and often used to teach children the basic ideas of programming. The "turtle graphics" we do here has very little to do with real LOGO, which is a much more powerful system.

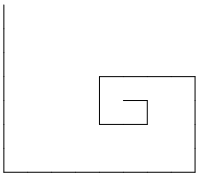
We will point out some more commands later, although this set is sufficient to do quite a lot.[2] For example, we can ask our turtle to make a self portrait:

```
> TurtleCmd(‘FRFLFFF RFFLFFLFFR FFFFF RFFLFFLFFR FFFLFR FFLF RBFL
            FLFF RFLFRFLFRFLF FRFLFFRFLFFF LFRFFLF RFFLF RFLFRFLF FFF‘);
```



Or, we can make a spiral:

```
> TurtleCmd(‘FRFR FFRFFR FFFFRFFFFR FFFFFFFFRFFFFFFFR‘);
```

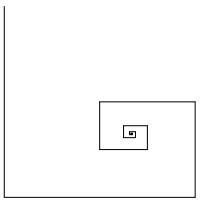


This last can be represented more succinctly if we add a pair of new commands, namely

G — “Grow”, i.e. double the unit of length S — “Shrink” (halve the unit of length)

Then the last picture can be produced by the string “FRFRG FRFRG FRFRG FRFR”, or, even better, we can make it spiral much more by repeating “FRFRG” twenty times:

```
> TurtleCmd(cat(seq(‘FRFRG‘,i=1..20)));
```

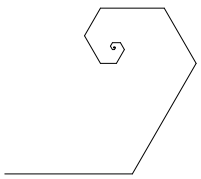


You may have noticed that the command `TurtleCmd` always resizes the resulting graphic to be of constant width. While this may seem a minor point, it is not. In any single string of turtle commands, “FF” will always be twice as long as “F”; however, `TurtleCmd(‘F‘)` and `TurtleCmd(‘FFFFFFFFFF‘)` produce identical results. This is because the a single “F” in the second case is 1/10 the width of the figure, while it is the full width in the first case.

We can also adjust some of the turtle’s attributes. For example, the turtle need not make only 90 degree turns. We use the command `ResetTurtle()` to set the turtle back to its initial state, then `SetTurtleAngle` to specify the angle (in degrees) that the turtle will turn left or right when it encounters an L or R command.

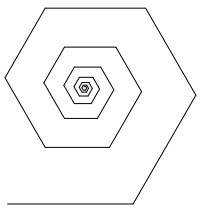
[2] In fact, this set of commands is already redundant. We could replace L with RRR and B with RRFLL.

```
> ResetTurtle(); SetTurtleAngle(60);
  TurtleCmd(cat(seq(`FRFRG`,i=1..20)));
```



`SetTurtleScale` allows us to modify the behavior of the S and G commands. Its argument is the factor the unit of length changes when an S is encountered.

```
> ResetTurtle(); SetTurtleAngle(60); SetTurtleScale(.8);
  TurtleCmd(cat(seq(`FRFRG`,i=1..20)));
```



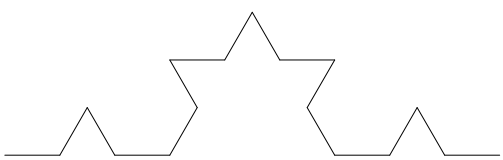
4.2 A fractal

Now let's try to do something more interesting. Begin with a straight segment (command "F"). Now remove the middle third of the segment, and replace it with the top two sides of an equilateral triangle. Since the overall figure produced by the turtle is always rescaled to unit size, we can represent the latter with the command "F LFRRFL F", where the part LFRRFL describes the bump we added. Of course, we have to set the angle the turtle turns to be 60°, and we need the RR in the middle to make a 120° turn.

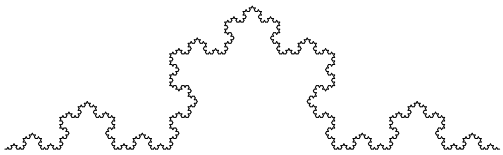
```
> SetTurtleAngle(60);
  plots[display](array([TurtleCmd(`F`), TurtleCmd(`FLFRRFLF`)]));
```

Now let's put a bump on each segment in the second figure. This is the same as replacing each F in the second command with "FLFRRFLF", giving us

```
> TurtleCmd(`FLFRRFLF L FLFRRFLF RR FLFRRFLF L FLFRRFLF`);
```



Putting bumps on each segment of that figure, and then adding smaller bumps to each segment of *that* figure, and then doing it yet again, gives us a very wiggly curve:



Notice that the width of each bump added is 1/3 of the one added previously, so by the time we have done this five times (as in the previous figure), the bumps are $1/3^5$ the size of the whole figure. If we continue much more, the changes become smaller than we can discern. It should be clear that there is a well defined "curve" which corresponds to the limit of doing this process infinitely often. We shall try to make this statement a little more precise now.

Let $\mathcal{K}_0$ be the figure at the 0-th stage, that is, the straight segment, and let $\mathcal{K}_1$ be the curve after we have added one bump. In general, let $\mathcal{K}_n$ be the curve after the n-th step. We claim that there is a well-defined limit curve $\mathcal{K}_\infty = \lim_{n\to\infty} \mathcal{K}_n$.

To make sense of this statement, we need a way to measure how far apart two sets in the plane are. That is, if S_1 and S_2 are two sets of points, we shall define the "Hausdorff distance" between S_1 and S_2 as follows. First, define the distance from a point x to a set S as the distance between x and the closest point in S, that is

$$d(x, S) = \inf_{y\in S} d(x, y).$$

Then let the Hausdorff distance between S_1 and S_2 be the greatest of all such distances, as x moves through each set.

$$d(S_1, S_2) = \max\left\{\sup_{x\in S_1} d(x, S_2), \quad \sup_{x\in S_2} d(x, S_1)\right\}$$

Once you have absorbed that, it should be simple to see that that we now need only show that for any $\epsilon > 0$, we can always find N sufficiently large so that $d(\mathcal{K}_N, \mathcal{K}_n) < \epsilon$ for all $n > N$. This follows from the fact that the size of the bumps goes down by a factor of 1/3 at each stage. Thus, $d(\mathcal{K}_n, \mathcal{K}_{n+1}) < 1/3^{n+1}$, and

$$d(\mathcal{K}_N, \mathcal{K}_n) < \sum_{j=N}^{n} 1/3^j < \frac{1}{2\cdot 3^{N-1}}.$$

So there certainly is a well-defined limit curve $\mathcal{K}_\infty$.

Now, this "curve" $\mathcal{K}_\infty$ has some interesting properties. First, notice that the distance in the plane from one end to the other is 1, but the "curve" itself is infinitely long. In fact, the length of any small part of the curve is infinite, as well. We can see that by examining how the lengths of each $\mathcal{K}_n$ varies with n:

curve	# segments	segment length	total length
$\mathcal{K}_0$	1	1	1
$\mathcal{K}_1$	4	1/3	4/3
$\mathcal{K}_2$	16	1/9	16/9
$\mathcal{K}_3$	64	1/27	64/27
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathcal{K}_n$	4^n	$1/3^n$	$4^n/3^n$

At each stage, every segment is replaced by four segments 1/3 as long. Thus, the overall length grows by a factor of 4/3 at each step. For any finite length ℓ we choose, we can find an n so that the length of $\mathcal{K}_n$ is greater than ℓ.

Note also that $\mathcal{K}_\infty$ is nowhere differentiable: it has corners densely throughout it. Finally, note that $\mathcal{K}_\infty$ is "self-similar". By this we mean that if you take any small piece of it, no matter how small, there is a small copy of the whole $\mathcal{K}_\infty$ within it.

The curve $\mathcal{K}_\infty$ is an example of a fractal.[3] This particular example is called the von Koch curve, and was discovered by H. von Koch in the late nineteenth century. This curve, and others like it, caused quite a stir in the mathematics community at the time because of its peculiar properties. We will explore additional fractals and some of their properties in the remainder of this chapter.

The particular way we generated the Koch curve is called an "L-system" or a "Lindenmeyer system". In an L-system, one begins with an initial figure (called the initiator), and a set of rules for modifying any figure to obtain the next in the sequence. In our example above, the initiatior was a straight segment (F) and the recursion rule was F $\mapsto$ FLFRRFLF.

4.3 Recursion and making a Koch Snowflake with Maple

In the previous section, we didn't give the specifics of exactly how we generated the Koch curve of level 5 which we showed. The general procedure was pointed out ("Begin with F. Replace each F with FLFRRFLF. Repeat 5 times."), but if you actually tried to do it yourself, you might have found it a bit tricky. Since the turtle command for the curve has 2388 letters, certainly it isn't something you would expect someone to type in directly.

Instead, a small maple procedure was written to generate it. Before giving the details, let's analyze the process a bit. We shall rely heavily on the self-similar nature of the Koch curve $\mathcal{K}_\infty$.

[3]The term "fractal" was coined by Benoit Mandelbrot in the late 1970s. Unfortunately, there is no universally accepted definition for this term. Some definitions require that a fractal be a self-similar object, but this excludes the most well-known example of a fractal, the Mandelbrot set, which is only approximately self-similar. We shall take it to mean any set whose Hausdorff dimension exceeds its topological dimension (we will see what this means in section 4.5).

Notice that the curve $\mathcal{K}_\infty$ is made up of four small copies of itself, arranged as

$$\mathcal{K}_\infty \mathrm{L}\ \mathcal{K}_\infty \mathrm{RR}\ \mathcal{K}_\infty \mathrm{L}\ \mathcal{K}_\infty .$$

If we use K_n to denote the set of turtle commands used to produce $\mathcal{K}_n$, then a similar statement is true for each K_n, namely

$$K_n = \begin{cases} K_{n-1}\mathrm{L}K_{n-1}\mathrm{RR}K_{n-1}\mathrm{L}K_{n-1} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} .$$

4.3.1 Recursive functions.

This is an example of a recursive definition. You've probably encountered such definitions previously in math classes. For example, the factorial function can be defined either as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

or

$$n! = \begin{cases} n \cdot (n-1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} .$$

The second definition is much simpler to implement in a programming language which allows recursive functions, such as maple.[4] To see this, let's implement the factorial function in both ways:

```
> fact1 := proc(n::posint)
    local i,ans;
    ans := 1;
    for i from 1 to n do
      ans := ans*i;
    od;
    ans;
  end:
> fact2 := proc(n::posint)
    if (n=1) then
        1;
    else
       n*fact2(n-1);
    fi;
  end:
```

Notice how much shorter and clearer the second version is.[5] However, there is a price to pay: a recursively defined function has a bit more overhead than a non-recursive one. However,

[4] Most programming languages these days do allow recursive function calls, although this was not the case until 1985 or so.

[5] We've stretched things a bit here. We could have implemented the factorial even more simply as `fact3:=n->product(i,i=1..n);`. However, this relies on the fact that the factiorial is a special product.

the savings in simplicity often overwhelms the minor loss in speed.[6]

To test your understanding, you might try to write a procedure to generate the Fibonacci numbers, which are defined by

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{if } n > 2 \\ 1 & \text{if } n = 1 \text{ or } n = 2 \end{cases} .$$

Try to implement this with both a recursive and non-recursive procedure. The non-recursive one is much more complicated!

4.3.2 A recursive procedure to generate K_n

With the idea of recursion in our bag of tools, we can now easily write our procedure to generate the Koch curve, using the observation we made in §4.3, namely that

$$K_n = \begin{cases} K_{n-1}\mathrm{L}K_{n-1}\mathrm{RR}K_{n-1}\mathrm{L}K_{n-1} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} .$$

```
> Koch:= proc (n::nonnegint)
    if (n=0) then
       `F`;
    else
        cat( Koch(n-1), `L`, Koch(n-1), `RR`, Koch(n-1), `L`, Koch(n-1));
    fi;
  end:
> Koch(2);
```

FLFRRFLFLFLFRRFLFRRFLFRRFLFLFLFRRFLF

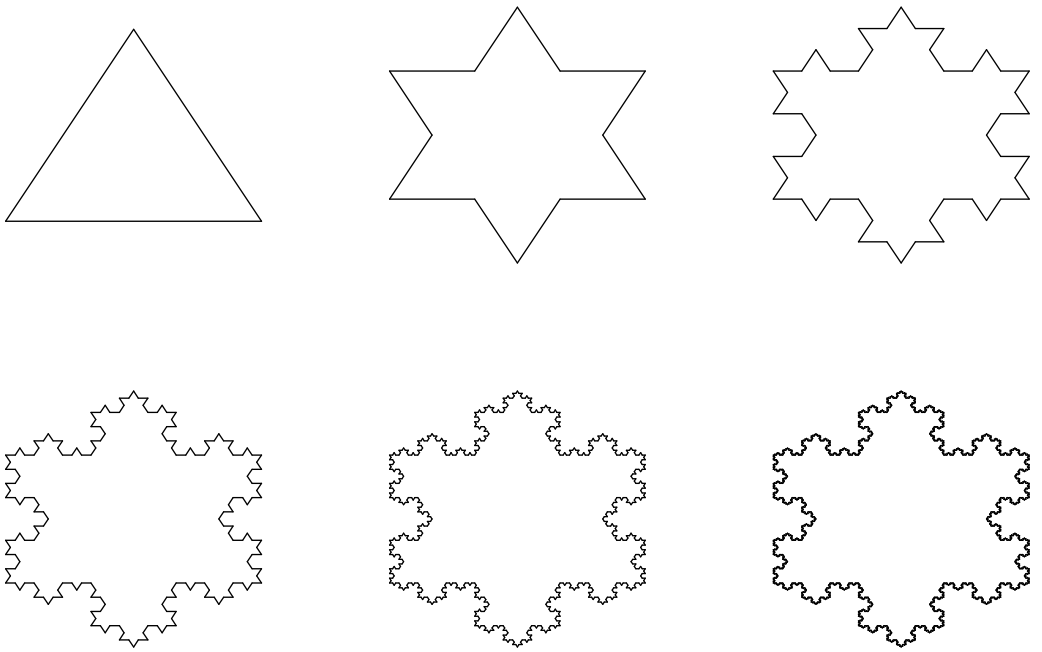
If you think about it for a second, you might notice that what we have just done is to write a little program (`Koch`) whose output is a program in another language (the string of turtle commands which describe K_n). While this may seem odd at first, doing this sort of thing is not at all uncommon.

4.3.3 The Koch Snowflake

Often, one sees the curve $\mathcal{K}_\infty$ as one side of a closed plane figure, the Koch Snowflake (sometimes also called the "Koch Island"). This is an infinite length "curve" which bounds a finite area, and resembles a snowflake. It is made by sticking together three copies of $\mathcal{K}_\infty$ meeting each other at 60° angles so that they close up. We do this now:

[6] Also, symbolic systems such as maple often have additional means to speed up recursive procedures. If you expect your function to be called several times with the same argument, you can add `options remember` to the definition. This instructs maple to save the result of previous function calls. When the function is called again, maple just gives the same result as before, rather than computing it again. But be careful: you get the same answer, even if you change the function later.

```
> KochFlake := proc(n::nonnegint)
   ResetTurtle();
   SetTurtleAngle(60);
   cat(`L`,Koch(n),`RR`,Koch(n),`RR`,Koch(n));
  end:
> plots[display](array([[seq(TurtleCmd(KochFlake(i)),i=0..2)],
                        [seq(TurtleCmd(KochFlake(i)),i=3..5)]]));
```



4.3.4 Some variations on the Koch curve

What happens to the curve $\mathcal{K}_\infty$ if we change the height of the bumps we add at each stage? We can do this easily by changing the angle the turtle turns when it encounters an R or an L. For example, an angle of 10° gives a curve which is nearly smooth.

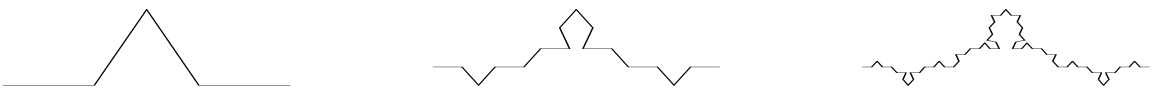
```
> ResetTurtle(); SetTurtleAngle(10);
  TurtleCmd(Koch(6));
```


But an angle of 80° gives a curve which wiggles wildly and almost seems to take up area.

```
> ResetTurtle(); SetTurtleAngle(80);
  TurtleCmd(Koch(6));
```

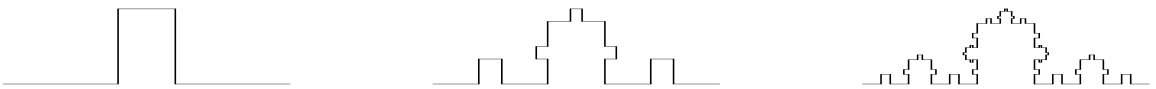
We shall make this observation more precise in section 4.5. You might try some additional experiments on your own, to gain intuition about how the limit curve depends on the parameters. For example, what would happen if you make the angle be 90°? Do you find the result surprising?

Some other variations you might want to try on your own might be to modify the `Koch` procedure so that the bumps are added alternately above and below the curve, as shown in the following figure:



Here the angle used was less than 60°. What happens for an angle of exactly 60°? How about a larger angle? Can you adjust things so that using 75° angle does not cause self-intersections?

Or, you might try using a different shape of a bump. For example, below we used squares, shrinking the scale first so the bumps don't run into each other. What happens if you vary the scale factor with `SetTurtleScale`?



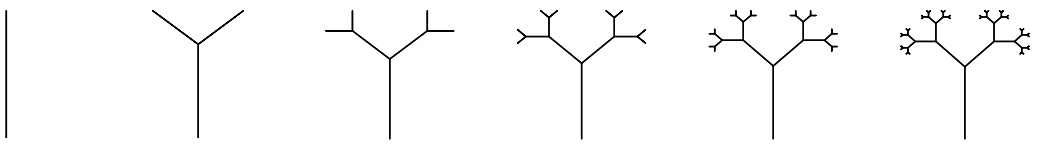
4.4 Making a tree

We now try to get our turtle to trace out a tree. In our tree, we start with a trunk, and then add two branches at the top. Then, at the end of each branch, we will add more branches.

First, we prefer our trees to grow up, so we change the turtle's initial heading. We also set our angle to be 45°.

```
> read(‘turtle.txt‘);
  SetInitialTurtleHeading(90);
  SetTurtleAngle(45);
```

The trunk of our tree will have no branching, and will just be the single command F. We will call this $\mathcal{T}_0$. For $\mathcal{T}_1$, we add a pair of branches to the top, to make a Y shape.[7] At each subsequent stage, we add another level of branches, as below:



How do we go about implementing this? As in the case of the Koch curve, we can use recursion, by noticing that $\mathcal{T}_n$ contains two small copies of $\mathcal{T}_{n-1}$ forming the left and right limbs

[7] As in §4.3, we will use a calligraphic font ($\mathcal{T}_n$) for the curve and ordinary roman (T_n) to denote the commands to produce the curve.

of the tree. Thus, we can define T_n recursively. Before proceeding, you might want to take a minute to think about how to do this on your own.

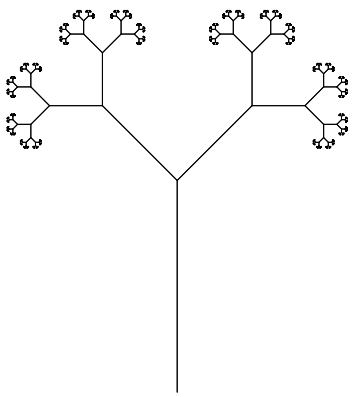
Notice that the letter Y cannot be drawn without either retracing part of the curve or lifting your pen. Rather than using a `PenUp` and `PenDown` command to tell the turtle to stop drawing,[8]we will have our turtle back up over a branch after we draw it. We also have to make sure our turtle is pointing in the right direction.

Thus, $T_0 =$ FB, and we can take T_1 as "F S LFBR RFBL G B". The "LFBR" part goes out the left branch and returns, and "RFBL" goes out the right branch and returns. To get T_2, we replace each copy of "FB" in T_1 with a scaled version of all of T_1, after turning a bit to the left or right. In general, we have

$$T_n = \begin{cases} \text{FSL}T_{n-1}\text{RR}K_{n-1}\text{LGB} & \text{if } n > 0 \\ \text{FB} & \text{if } n = 0 \end{cases} .$$

Using this, it is now easy to write the generating procedure.

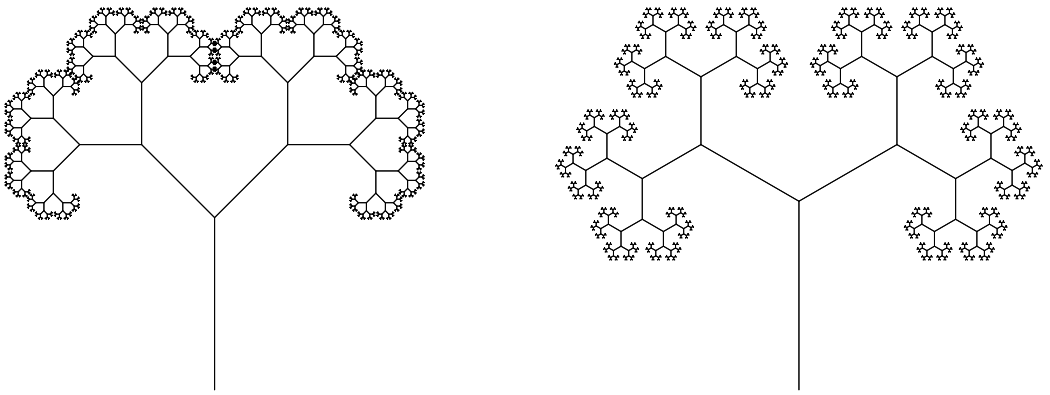
```
> Tree:= proc(n::nonneg)
    options remember;
    if (n=0) then
       `FB`;
    else
       cat(`FSL`, Tree(n-1), `RR`, Tree(n-1), `LGB`);
    fi;
  end:

> TurtleCmd(Tree(10));
```



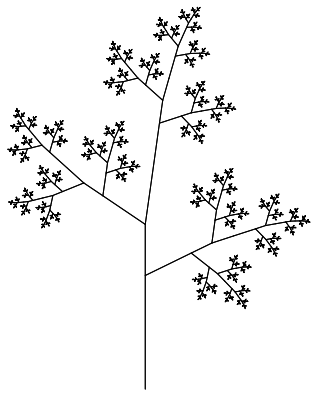
You might want to experiment with varying the angle and the scale factor, and see how the tree changes. For example, if we use `SetTurtleScale(.6)`, the branches of $\mathcal{T}_8$ just touch, and they overlap in $\mathcal{T}_n$ if $n > 8$. However, if we change the angle to 60°, the branches stay away from each other.

[8]The file turtle.txt does, in fact, implement these commands as D and U— see §4.9 for more details.

```
> ResetTurtle(); SetInitialTurtleHeading(90);
  SetTurtleAngle(45); SetTurtleScale(.6);
  t10a:=TurtleCmd(Tree(10)):
> SetTurtleAngle(60);
  t10b:=TurtleCmd(Tree(10)):
> plots[display](array([t10a, t10b]));
```



As an extra challenge, try your hand at producing a fern like the one below.



4.5 Fractal Dimension

From an early age, we that learn lines and curves are one-dimensional, planes and surfaces are two-dimensional, solids such as a cube are three dimensional, and so on. More formally, we say a set is n-dimensional if we need n independent variables to describe a neighborhood of any point. This notion of dimension is called the *topological dimension* of a set.[9] The dimension of the union of finitely many sets is the largest dimension of any one of them, so if we "grow hair" on a plane, the result is still a two-dimensional set. We should note here that if we take

[9]In fact, there is a much more precise definition of topological dimension, but to give it requires more background material than we are prepared to give here.

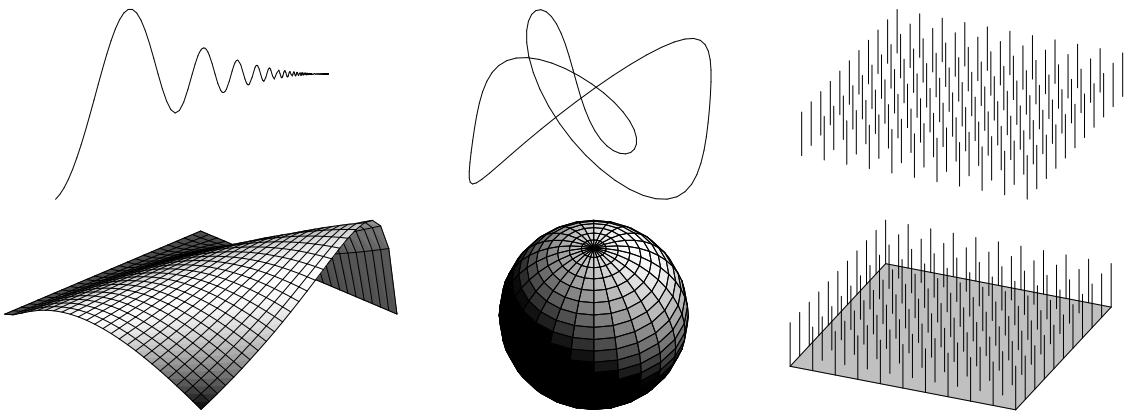


Figure 4.5.1: Some one- and two-dimensional sets (the sphere is hollow, not solid).

the union of an infinite collection of sets, the dimension can grow. For example, a line, which is one-dimensional, is the union of an infinite number of points, each of which is a zero-dimensional object.

There can occaisionally be a little confusion about the dimension of an object: sometimes people call a sphere a three-dimensional object, because it can only exist in space, not in the plane. However, a sphere is two-dimensional: any little piece of it looks like a piece of the plane $\mathbb{R}^2$, and in such a small piece, you only need two coordinates to describe the location of a point.[10]

But, what about the fractals we have been considering? For example, what is the dimension of the Koch snowflake? It has topological dimension one, but it is by no means a curve— the length of the "curve" between any two points on it is infinite. No small piece of it is line-like, but neither is it like a piece of the plane or any other $\mathbb{R}^n$. In some sense, we could say that it is too big to be thought of as a one-dimensional object, but too thin to be a two-dimensional object. Maybe its dimension should be a number *between* one and two. In order to make this kind of thinking more precise, let's look at the dimension of familiar objects another way.

4.5.1 Box counting dimension

What is the relationship between an object's length (or area or volume) and its diameter? The answer to this question leads to another way to think about dimension. Let us consider a few examples.

[10] In fact, this is just a different measure of dimension, called the *embedding dimension*: a set has embedding dimension n if n is the smallest integer for which it can be embedded into $\mathbb{R}^n$ without intersecting itself. Thus, the embedding dimension of a plane is 2, the embedding dimension of a sphere is 3, and the embedding dimension of a klein bottle is 4, even though they all have (topological) dimension two. A famous theorem (the Whitney embedding theorem) says that if a manifold has topological dimension n, its embedding dimension is at most $2n$.

If we try to cover the unit square with little squares of side length ϵ, how many will we need? Obviously, the answer is $1/\epsilon^2$. How about to cover a segment of length 1? Here we need only $1/\epsilon$ little squares. If we think of the square and segment as sitting in space and try to cover them with little cubes ϵ on a side, we get the same answer. And if we use the little cubes to cover a $1 \times 1 \times 1$ cube, how many will we need? Exactly $1/\epsilon^3$. Note that the exponent here is the same as the dimension of the thing we are trying to cover. This is no coincidence.

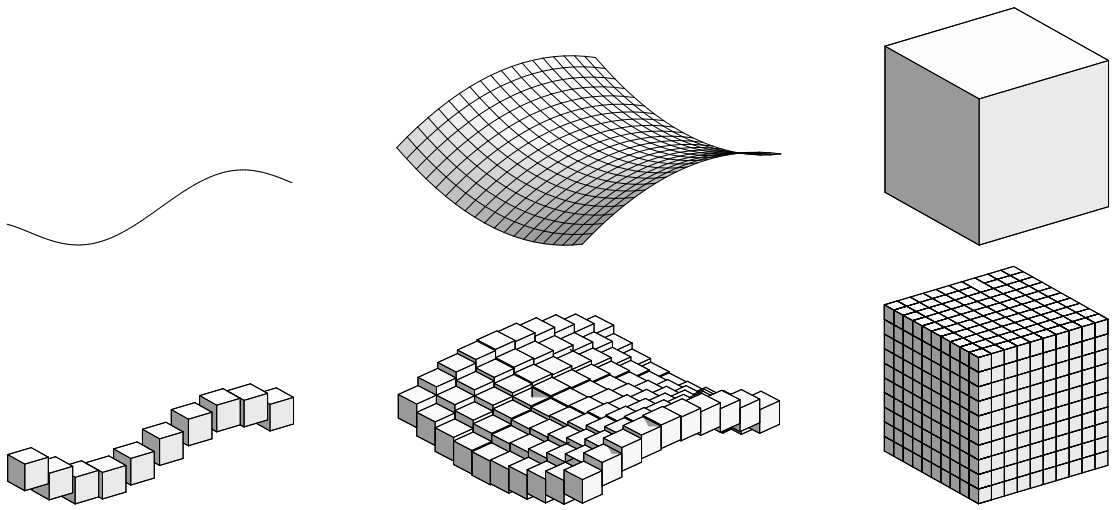


Figure 4.5.2: Covering a curve, a surface, and a solid cube with cubes of diameter ϵ.

We define the *box-counting dimension* (or just "box dimension") of a set $\mathcal{S}$ contained in $\mathbb{R}^n$ as follows: For any $\epsilon > 0$, let $N_\epsilon(\mathcal{S})$ be the minimum number of n-dimensional cubes of side-length ϵ needed to cover $\mathcal{S}$. If there is a number d so that

$$N_\epsilon(\mathcal{S}) \sim 1/\epsilon^d \qquad \text{as} \qquad \epsilon \to 0,$$

we say that the box-counting dimension of $\mathcal{S}$ is d. We will denote this by $\dim_\square(\mathcal{S}) = d$.

Note that the box-counting dimension is d if and only if there is some positive constant k so that

$$\lim_{\epsilon \to 0} \frac{N_\epsilon(\mathcal{S})}{1/\epsilon^d} = k.$$

Since both sides of the equation above are positive, it will still hold if we take the logarithm of both sides to obtain

$$\lim_{\epsilon \to 0} (\ln N_\epsilon(\mathcal{S}) + d \ln \epsilon) = \ln k.$$

Solving for d gives

$$d = \lim_{\epsilon \to 0} \frac{\ln k - N_\epsilon(\mathcal{S})}{\ln \epsilon} = -\lim_{\epsilon \to 0} \frac{N_\epsilon(\mathcal{S})}{\ln \epsilon}.$$

Note that the $\ln k$ term drops out, because it is constant while the denominator becomes infinite as $\epsilon \to 0$. Also, since $0 < \epsilon < 1$, $\ln \epsilon$ is negative, so d is positive as we would expect.

We should remark that there are some sets $\mathcal{S}$ for which $\dim_\square(\mathcal{S})$ cannot be defined because there is no d for which the limit converges.[11] We will not encounter such examples here, however. Since the box-counting dimension is so often used to calculate the dimensions of fractal sets, it is sometimes referred to as "fractal dimension". We prefer the term box dimension, however, because sometimes the term "fractal dimension" might refer to box dimension, Hausdorff dimension, or even other measures of dimension such as the information dimension or capacity dimension.

When computing box dimension, several simplifications can be made.

- We need not use boxes if some other shape is more convenient: if we cover our set with disks of diameter ϵ, or even stars or mickey-mouses of diameter ϵ, we will get the same answer.
- Not every possible ϵ need be considered. It is enough to consider the limit of $\ln N_{\epsilon_i}/\ln \epsilon_i$ where ϵ_i is a sequence converging to zero. Choosing a convenient sequence often makes the calculations much easier.

Sometimes box counting dimension is referred to as "similarity dimension" in the context of self-similar sets. If a set is self-similar, there is an expansion factor r by which one can blow up a small copy to get the whole set. If there are exactly N such small copies that make up the entire set, the box dimension is easily seen to be $\ln N/\ln r$.

4.5.2 Computing the box dimension of some examples

Not surprisingly, the box dimensions of ordinary Euclidean objects such as points, curves, surfaces, and solids coincide with their topological dimensions of 0, 1, 2, and 3— this is, of course, what we would want to happen, and follows from our the discussion at the beginning of §4.5.1. But what about other, more complicated sets? Let's try a few simple examples for some subsets of the unit interval $[0, 1]$. In these cases, our ϵ-cubes can be closed intervals of length ϵ.

Consider the set of points $\mathcal{A} = \left\{0, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots\right\}$. For any $n \geq 0$, we can cover $\mathcal{A}$ with n intervals of length $1/2^n$: we need one for the elements between 0 and $\frac{1}{2^n}$, and another interval for each of the remaining $n-1$ elements of $\mathcal{A}$. This means that $\dim_\square(\mathcal{A}) = \lim_{n\to\infty} n/2^n = 0$. The box dimension and the topological dimension of $\mathcal{A}$ are the same.

Let $\mathcal{Q}$ be the set of rational numbers in the interval $[0, 1]$, that is,

$$\mathcal{Q} = \left\{ \frac{p}{q} \,\middle|\, p, q \text{ are relatively prime integers with } p \leq q \right\}.$$

[11] Mathematicians often use a more general measure called *Hausdorff dimension*, which is defined for every such set. One difficulty with the Hausdorff dimension is that it is often very hard to compute. The Hausdorff and box dimensions coincide for compact, self-similar fractals, so we will not concern ourselves with the distinction.

What is $\dim_\square(\mathcal{Q})$? Since the rationals are dense in $[0,1]$, any interval we choose contains some. This means for every $\epsilon > 0$, we will need $1/\epsilon$ intervals to cover $\mathcal{Q}$. Thus, $N_\epsilon(\mathcal{Q}) = 1/\epsilon$. Consequently, $\dim_\square(\mathcal{Q}) = \lim_{\epsilon\to 0} \frac{1/\epsilon}{1/\epsilon} = 1$.

Recall that any real number x can be represented as a (possibly infinitely long) decimal. For example, $1/4 = .25$, $1/11 = .0101\overline{01}\ldots$, and $\pi = 3.14159265\ldots$. The decimal expansion is not quite unique— for example, $1/2$ can be written as either $.4999999\overline{9}\ldots$ or $.5000000\overline{0}\ldots$. This is the only possible point of confusion, however: any real number x ending in all zeros has another representation ending in all nines.

Let us determine the box-counting dimension of the set

$$\mathcal{D} = \{x \in [0,1] \,|\, x \text{ has a decimal expansion containing no 4s or 5s}\}\,.$$

First, let's note a few things about $\mathcal{D}$:

- It is *totally disconnected*: that is, between any two points of $\mathcal{D}$, there is another point which is not in $\mathcal{D}$.
- It is *closed*: the limit of any convergent sequence of points in $\mathcal{D}$ is still in $\mathcal{D}$. Note that this is not the case for the set $\mathcal{Q}$ above— a sequence of rational numbers can converge to an irrational.
- It has uncountably many points. Much like the irrational numbers, there are far too many elements of $\mathcal{D}$ to enumerate them. Also like the irrational numbers, we *can* enumerate what is not in $\mathcal{D}$.
- We shall see that $\mathcal{D}$ is also self-similar: any small piece of it can be scaled up to look like the whole thing just by multiplying by an appropriate power of 10.

So, what is $\dim_\square(\mathcal{D})$? Let's construct a sequence of covers of $\mathcal{D}$ whose diameter tends to zero, and count the number of pieces we need. Note that while $\mathcal{D}$ contains the points[12] .4 and .6, it does not contain the open interval $(.4, .6)$, so we can cover it by two intervals of length .4, namely $[0, .4]$ and $[.6, 1]$. This means $N_{.4} = 2$.

At the next finer level, we can see that $\mathcal{D}$ can be covered with the intervals

$$\begin{array}{cccccccc} [0,.04], & [.06,.10], & [.10,.14], & [.16,.20], & [.20,.24], & [.26,.30], & [.30,.34], & [.36,.40], \\ [.60,.64], & [.66,.70], & [.70,.74], & [.76,.80], & [.80,.84], & [.86,.90], & [.90,.94], & [.96,1] \end{array}$$

That is, we need 16 intervals of length .04. This means that $N_{.04} = 16$.

For $\epsilon = .004$, we will need 8 times as many intervals to cover $\mathcal{D}$. This pattern continues: each time we shrink ϵ by another factor of 10, we need 8 times as many intervals.

[12] $\mathcal{D}$ contains .4 because it can be written as $0.399\overline{9}\ldots$, which contains no 4s or 5s.

This means that

$$\dim_{\square}(\mathcal{D}) = -\lim_{n\to\infty} \frac{\ln 8^{n-1}}{\ln\left(4\cdot 10^{-n}\right)} = -\lim_{n\to\infty} \frac{(n-1)\ln 8}{\ln 4 - n\ln 10} = \frac{\ln 8}{\ln 10} = \frac{3\ln 2}{\ln 10} \approx .903089987$$

The box-counting dimension of $\mathcal{D}$ is not an integer. Note that the topological dimension of $\mathcal{D}$ is zero.

4.6 Cantor sets

The set $\mathcal{D}$ of the previous section is an example of a Cantor set. Often, a Cantor set is described as the result of an iterative process, much in the same way we described the Koch curve in §4.2. We'll describe the most common Cantor set, the "middle-thirds Cantor set", this way now.

Begin with the interval $[0,1]$ as the set $\mathcal{C}_0$. Now remove the open interval $(\frac{1}{3},\frac{2}{3})$ from $\mathcal{C}_0$ to obtain

$$\mathcal{C}_1 = \left\{ \left[0,\frac{1}{3}\right],\ \left[\frac{2}{3},1\right] \right\}.$$

To get the second stage, we remove the middle third of each interval in $\mathcal{C}_1$. This gives

$$\mathcal{C}_2 = \left\{ \left[0,\frac{1}{9}\right],\ \left[\frac{2}{9},\frac{1}{3}\right],\ \left[\frac{2}{3},\frac{7}{9}\right],\ \left[\frac{8}{9},1\right] \right\}.$$

We continue in this way, removing the middle third of each interval in $\mathcal{C}_n$ to obtain $\mathcal{C}_{n+1}$ (See Fig. 4.6.3). Note that $\mathcal{C}_n \subset \mathcal{C}_m$ for all $n > m$. What is left as we let $n \to \infty$ is the Cantor set. That is,

$$\mathcal{C} = \bigcap_{n=0}^{\infty} \mathcal{C}_n.$$

Figure 4.6.3: The first few stages $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_5$ in the construction of the middle-thirds Cantor set $\mathcal{C}$.

It may not be clear at first that $\mathcal{C}$ even exists. It seems we are taking just about everything out when we remove the intervals. Indeed, the total length of intervals we remove is

$$\frac{1}{3} + \frac{2}{9} + \frac{4}{27} + \frac{8}{81} + \ldots = 1.$$

That is, the total length of the segments removed is equal to the segment we started with! For this reason, a set like $\mathcal{C}$ is sometimes called a "Cantor dust". But notice that the endpoints of each interval of $\mathcal{C}_n$ must be in $\mathcal{C}$, because they never get removed at any level. Thus, $\mathcal{C}$ is nonempty, since it contains at least the points $\left\{0, 1, \frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{2}{9}, \frac{7}{9}, \frac{8}{9}, \frac{1}{27}, \frac{2}{27}, \ldots\right\}$. And it contains quite a few more points, as well. To see that, we can view $\mathcal{C}$ in a manner similar to the description of the set $\mathcal{D}$ given in §4.5.2. However, $\mathcal{C}$ is more naturally described in base 3, rather than base 10. Let's briefly review what we mean by this.

When we express a number x in base 10, we are expressing it as sums of powers of ten. Thus, when we write $e^5 \approx 148.413$, we are actually saying that

$$e^5 \approx 1 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0 + 4 \cdot 10^{-1} + 1 \cdot 10^{-2} + 3 \cdot 10^{-3}.$$

There is nothing magic about powers of 10, of course (except that it is the base we learned as children). For example, we could express this[13] in ternary (that is, base 3), where we would have

$$e^5 \approx 12111.102011_3 = 1{\cdot}3^4{+}2{\cdot}3^3{+}1{\cdot}3^2{+}1{\cdot}3^1{+}1{\cdot}3^0{+}1{\cdot}3^{-1}{+}0{\cdot}3^{-2}{+}2{\cdot}3^{-3}{+}0{\cdot}3^{-4}{+}1{\cdot}3^{-5}{+}1{\cdot}3^{-6}.$$

In computer applications, the bases 16 (hexadecimal) and 2 (binary) are very common, because the computer represents everything internally in binary.[14]

Returning to the Cantor set $\mathcal{C}$, note that when we remove the interval (1/3, 2/3), we are removing all the numbers whose ternary expansion has a 1 immediately after the decimal place.[15] At the next stage, we remove those points which have a 1 in the second place after the decimal, and so on. Thus, in the limit, we obtain the following alternate description of $\mathcal{C}$:

$$\mathcal{C} = \{x \in [0,\, 1] \,|\, x \text{ has a ternary expansion containing no 1s}\}.$$

You may remember that rational numbers correspond to numbers with a decimal expansion that eventually repeats; the same holds true of the expansion in any base (how might you prove this if you assume it for decimals?). It is easy to see using this description that $\mathcal{C}$ is closed (that is, it contains all its limit points), totally disconnected, and contains an uncountable number of elements (there are a *lot* of sequences of 0s and 2s), just like the set $\mathcal{D}$ of the previous section. It is also self-similar, because if we take any small section of it and expand it by an appropriate power of 3, it looks like the entire set.

[13] Maple will convert integers to other bases, using a command such as `convert(148, base, 3)`. One has to work a little bit to convert fractions— can you think of a way to do this?

[14] Hexadecimal is convenient for use with a binary system, because we can group blocks of 4 binary digits to obtain one hexadecimal digit (by convention, the letters A–F are used to represent the integers 10–15). This is quite handy in computer applications, because a single byte is represented by 8 binary digits, or two hex digits. Before 8-bit bytes became standard, octal (base 8) was very common as well. Octal is much less convenient for 8-bit quantities, however, because 8 bits doesn't break up nicely into groups of 3 bits.

[15] As in decimal expansions, some points have two representations. Thus, the point $1/3$ can be written either as $.1_3$ or as $.0222\overline{2}\ldots$. As in the construction of $\mathcal{D}$ in §4.5.2, we keep such points in our set.

What is its box dimension? The description by the sets $\mathcal{C}_n$ give us an effective way to calculate the limit: at the n^{th} stage, we have 2^n intervals of length 3^{-n}. Using this, we can immediately calculate that

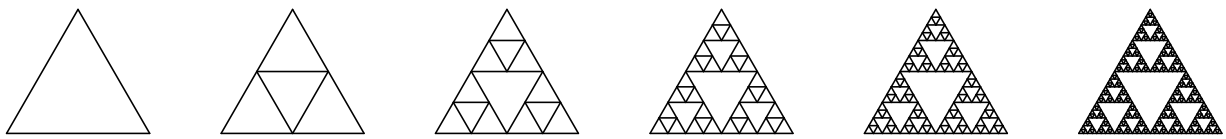
$$\dim_{\square}(\mathcal{C}) = \lim_{n\to\infty} \frac{\ln 2^n}{\ln 3^n} = \frac{\ln 2}{\ln 3} \approx 0.63093.$$

You may have noticed the strong similarity in the construction of the Koch curve $\mathcal{K}$ of §4.2 and the middle-thirds Cantor set $\mathcal{C}$. In fact, notice that the intersection of $\mathcal{K}$ with its base (the initial segment $\mathcal{K}_0$) is exactly the middle-thirds Cantor set.

4.7 The Sierpinski gasket

Another very commonly encountered fractal is the Sierpinski gasket, which can be described as follows:

We start with an equilateral triangle $\mathcal{S}_0$, and replace it by three equilateral triangles with a base half the size of the original, stacked so the original perimiter is kept, but leaving a hole in the center. We then replace each of those triangles by three more triangles, to obtain $\mathcal{S}_2$ as nine triangles, each with a base of length 1/4. Continuing in this fashion infinitely many times yields the Sierpinski gasket $\mathcal{S}$. The sets $\mathcal{S}_0$ through $\mathcal{S}_6$ are shown below.



It is easy to see that $\dim_{\square}(\mathcal{S})$ is $\ln 3/\ln 2$, or about 1.58. You should try to confirm this fact for yourself. The self-similarity of the Sierpinski gasket should help this calculation considerably.

How might we coerce our turtle into making a Sierpinski gasket? While the procedure is simple enough to describe, explicitly writing it down in symbols requires a bit of thought. Give it a try before continuing.

One way to accomplish this[16] involves taking a slightly different viewpoint. Rather than making the gasket by replacing triangles with smaller triangles, we construct a fractal "curve" that traverses the base and the perimeters of the "holes". This traces out the same sets $\mathcal{S}_n$, except for two segments which we can easily add later.

We view S_0 as the straight horizontal segment F, along with two other segments which are not horizontal. To emphasize the difference between the horizontal and the non-horizontal segments, we will traverse the non-horizontal segments backwards, and use the command B. Thus, our first triangle is $S_0 =$ FRBLLB.

[16] We will give another solution which is closer to our original description following §4.9.

Now, to obtain S_1 from S_0, we replace the horizontal segment F by two smaller ones with an inverted triangle between them. As with S_0, we use F to denote the horizontal parts and B for the non-horizontal portions. Thus, we apply the transformation F $\mapsto$ SF RBLFLBR FG. This gives us S_1 from S_0 as the command "SF RBLFLBR FG RBLLB". If we replace all three Fs in S_1 using this transformation, we will obtain S_2.

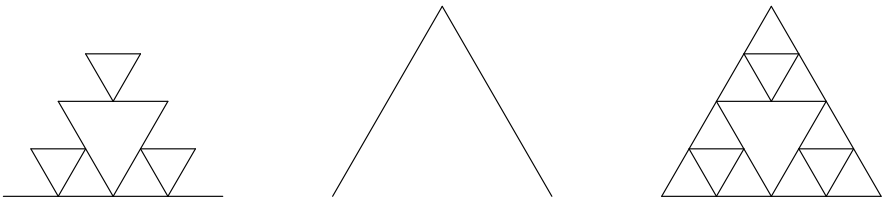


Figure 4.7.4: The sets produced by the turtle commands for s_2, h, and S_2 in the construction of the Sierpinski gasket.

Before we try to describe any of the S_n recursively, note that in this description, each one really consists of two pieces: there is the complicated, recursive portion which describes most of the gasket, and then there is the final RBLLB which makes the outer "hat" (see figure 4.7.4). Let's call the commands to produce this first part s_n and call commands for the hat h, that is, $S_n = s_n\, h$. Then we can describe S_n as follows

$$S_n = s_n\, h \qquad s_n = \begin{cases} \mathrm{S}s_{n-1}\mathrm{RBL}s_{n-1}\mathrm{LBR}s_{n-1}\mathrm{G} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} \qquad h = \mathrm{RBLLB}.$$

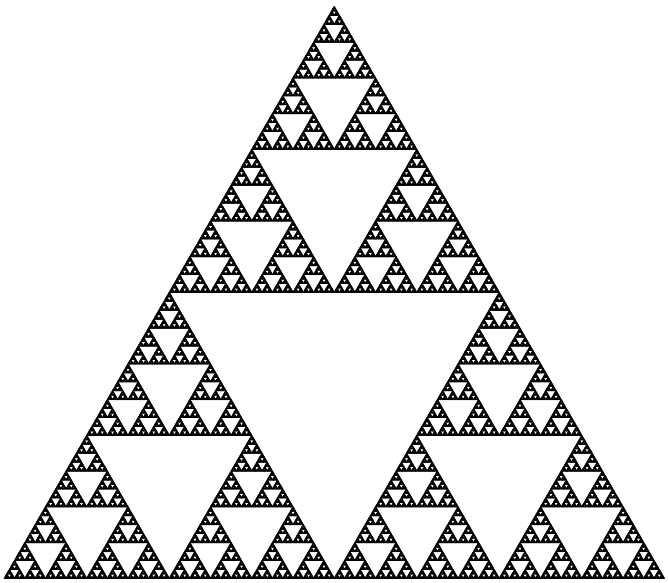
As in the previous sections, once we have worked out the algorithm, writing the maple code to implement it is quite straightforward. We we will use the name `Sierp` for the procedure that produces s_n, and `Sierpinski` for the procedure to produce S_n.

```
> Sierp:=proc(n::nonnegint)
    if (n=0) then
       `F`;
    else
       cat(`S`,Sierp(n-1), `RBL`, Sierp(n-1), `LBR`, Sierp(n-1), `G`);
    fi;
  end:
> Sierpinski:= n -> cat( Sierp(n), `RBLLB`):
```

Since we also need to set the angle and scale appropriately, we can combine them all into a single command `DrawSierp`:

```
> DrawSierp:= proc(n::nonnegint)
     SetTurtleAngle(60);
     SetInitTurtleHeading(0);
     SetTurtleScale(.5);
     TurtleCmd(Sierpinski(n));
  end:
```

```
> DrawSierp(9);
```



Warning: Drawing even $\mathcal{S}_8$ takes a significant amount of time and is almost indistinguishable from $\mathcal{S}_7$. If you try to draw $\mathcal{S}_9$, be prepared to wait quite a while, and make sure your computer has plenty of memory.

4.8 Inside the turtle's shell

We have how to use the turtle package in several ways; now it is time to look at how the package itself is implemented. This was done in several layers

4.9 Extending the turtle's commands