Notes for MAT 331

Mathematical Problem Solving with Computers

Summer 2002

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

Contents

1 Preliminaries about Maple **1:1**

1 Starting a Maple Session . . . **1**:1
2 Basic Maple . . . **1**:3
2.1 Maple for numeric calculations . . . **1**:3
2.2 Maple for symbolic manipulations . . . **1**:5
2.3 Graphing in Maple . . . **1**:8
3 The Maple worksheet . . . **1**:9
3.1 Introducing the worksheet . . . **1**:9
3.2 Worksheet basics: the front-end and the kernel . . . **1**:9
3.3 Online Maple help . . . **1**:11
3.4 Documenting and structuring your worksheet . . . **1**:12
4 Assignments, Functions and Constants . . . **1**:12
4.1 Assignment statements . . . **1**:12
4.2 Variables and subsequent assignments: unassignments . . . **1**:13
4.3 Functions known to Maple. How to define your own functions . . . **1**:15
4.4 Special constants and reserved names . . . **1**:17
5 The limit command . . . **1**:18
6 The diff (and Diff) command . . . **1**:20
6.1 Higher order derivatives . . . **1**:21
6.2 Implicit differentiation . . . **1**:22
7 The int (and Int) command . . . **1**:23
7.1 Symbolic integrals . . . **1**:23
7.2 Numerical integration . . . **1**:25
7.3 Approximations through Riemann Sums . . . **1**:26
7.4 Multiple integrals . . . **1**:27
8 The subs command . . . **1**:27
8.1 Why is subs useful? . . . **1**:29
9 Plotting with Maple . . . **1**:30
9.1 Plotting several functions (or curves) on the same axes . . . **1**:31
9.2 Fancier plotting . . . **1**:33
10 Exercises . . . **1**:35

2 If the Curve Fits, Wear It **2:1**
1 Interpolation . . . **2**:1
1.1 Polynomial Interpolation . . . **2**:1
1.2 Connect-the-Dots and Splines . . . **2**:3
2 When the data is approximate . . . **2**:5
3 Fitting a line to data . . . **2**:6
4 Fitting a cubic to data . . . **2**:10
5 Fitting other types of funtions . . . **2**:12
6 Fitting a circle . . . **2**:13
7 Robust fitting . . . **2**:16
8 A nod toward statistics . . . **2**:20

3 The Art of Phugoid **3:1**
1 The Phugoid model . . . **3**:1
2 What do solutions look like? . . . **3**:2
3 Existence of Solutions . . . **3**:7
4 Numerical Methods . . . **3**:7
4.1 Euler's method . . . **3**:8
4.2 Numerical Solutions, Numerical Integration, and Runge-Kutta . . . **3**:9
5 Seeing the flight path . . . **3**:10
6 Fixed Point Analysis . . . **3**:13
6.1 Linear Systems of ODEs . . . **3**:14
6.2 Fixed Points for the Glider . . . **3**:18
7 Qualitative Classification of Solutions . . . **3**:19
8 Dealing with the Singularity . . . **3**:21

4 fsqFsHn sGGousG **4:1**
1 Introduction to Cryptography . . . **4**:1
2 Simple Ciphers . . . **4**:2
2.1 Simple substitution . . . **4**:2
2.2 The Caesar cipher, and the ASCII encoding . . . **4**:4
3 Defining functions with `proc`; Local and global variables . . . **4**:9
4 Caesar cipher redux . . . **4**:12
5 Improved Caesar-like ciphers . . . **4**:15
5.1 The Vignère cipher . . . **4**:15
5.2 One-time pads . . . **4**:16
5.3 Multi-character alphabets . . . **4**:19
6 Reading and Writing from a file . . . **4**:20
7 Affine enciphering . . . **4**:21
7.1 When do affine encodings fail? . . . **4**:22
7.2 Implementing and using an affine encoding . . . **4**:23

7.3 Breaking an affine cipher 4:24
8 Enciphering matrices 4:25
8.1 Treating text as vectors 4:25
8.2 Affine encoding with matrices 4:26
8.3 A Known-plaintext attack on an affine matrix cipher 4:28
9 Modern cryptography 4:30
9.1 Secure cryptosystems 4:30
9.2 Message digests 4:32
9.3 Public Key cryptography 4:32
10 Some Number Theory 4:34
10.1 The greatest common divisor and the Euclidean algorithm 4:34
10.2 The Chinese Remainder Theorem 4:37
10.3 Powers modulo n 4:38
10.4 The Euler φ-function and Euler's Theorem 4:39
11 The RSA Public key cryptosystem 4:41
11.1 Details of the RSA algorithm 4:42
11.2 Implementing RSA in Maple 4:43
12 RSA encoding a file 4:47

5 A turtle in a fractal garden 5:1
1 Turtle Graphics 5:1
2 A fractal 5:3
3 Recursion and making a Koch Snowflake with Maple 5:6
3.1 Recursive functions. 5:6
3.2 A recursive procedure to generate K_n 5:7
3.3 The Koch Snowflake 5:8
3.4 Some variations on the Koch curve 5:9
4 Making a tree 5:10
5 Fractal Dimension 5:12
5.1 Box counting dimension 5:13
5.2 Computing the box dimension of some examples 5:15
6 Cantor sets 5:17
7 The Sierpinski gasket 5:19
8 Inside the turtle's shell 5:21
9 Extending the turtle's commands 5:24

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. Calculus courses may be 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 well 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 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 courses and your own research. More advanced documentation is available on line, in the computer lab, and in the library.

Maple runs on many different types of computers and operating systems. For example, you might use Maple from a Sun workstation running Solaris (a flavor of Unix) in the computer lab, from a Windows computer in the library, and from a Macintosh at home. Maple behaves similarly on all of them; we will usually point it out when we use something specific to one type of computer.

1 Starting a Maple Session

How to start a Maple session varies somewhat in different enviroments. On the Windows machines, you can usually either start Maple by selecting the option from the start menu, or by finding the appropriate icon (it usually looks like a Maple leaf) and clicking the mouse on it. In a Unix environment, there may be an icon or menu item for Maple; you can usually always just issue the command `xmaple &` or `maple -x &` to start it.

After a brief pause, you should see a window that looks similar to that in Figure 1; this is Maple's graphical (or worksheet) interface, where you will type your commands and see the

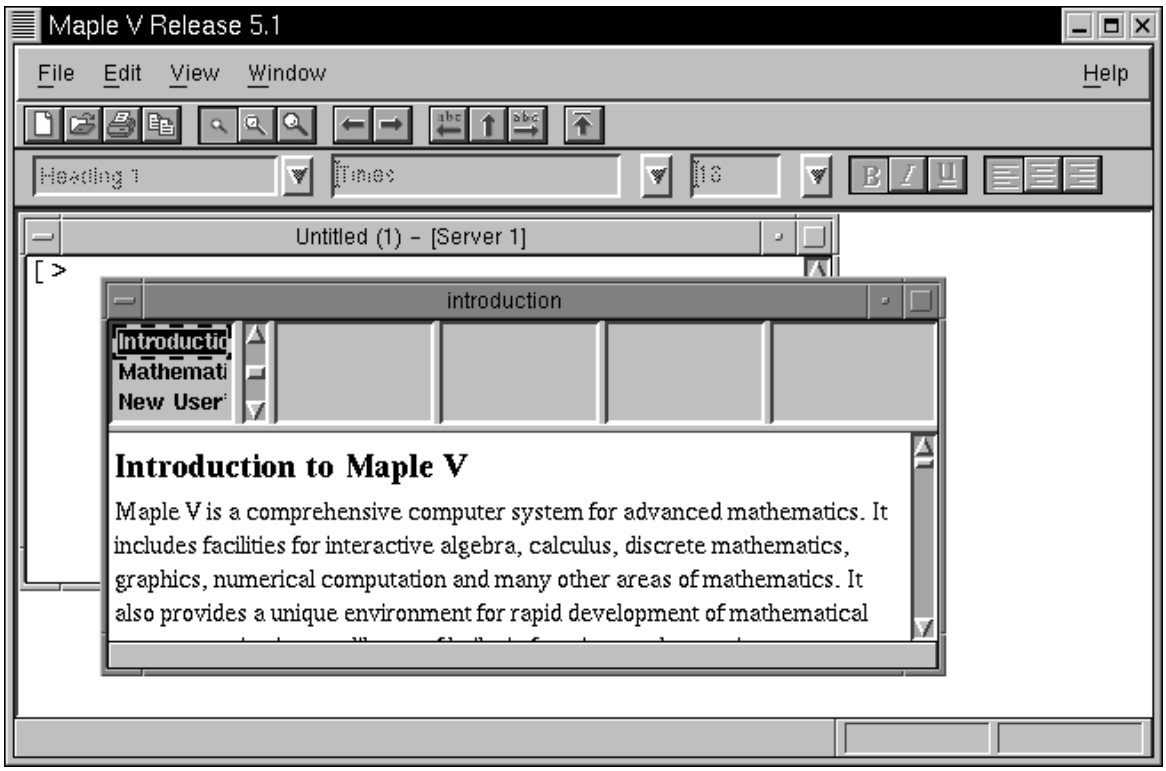
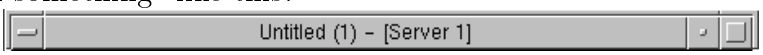


Figure 1: A newly-opened Maple worksheet.

results. We will go into more details of the worksheet interface in §3. As in the figure, you're session may have started with two windows open inside the main Maple window. The one in front in Figure 1 is a help window; we'll come back to this in §3.3. But for now, let's concentrate on a worksheet where we can enter some Maple commands. The top part of this window should look something[1] like this:



This window is a Maple worksheet, and is a place to enter your Maple commands. For now, click on the button which maximizes the current worksheet; this is on right side of the worksheet's title bar, and looks like ▣.[2] Clicking the mouse on this should make the worksheet fill most of the Maple window.

Most users typically use the Maple's graphical, or worksheet, interface; this is what starts you add the `-x` to the `maple` command in Unix. However, there is also a text-based interface, which you might see if you just type `maple`, omitting the `-x`.

[1] This particular image was made on a computer running Unix. Under Microsoft Windows or on a Macintosh, the symbols around at the edges will be slightly different, but with similar functions.

[2] Again, if you are running another type of computer, the button may look a little different. For example, under MS-Windows, it looks similar to this: ▣.

```
    |\^/|     Maple V Release 5.1 (SUNY Stony Brook)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
>
```

This interface is useful when you want to run Maple in a non-graphic environment; for example, over a dial-up connection. However, keep in mind that while all the usual Maple commands work, this interface is much more limited— it is difficult to edit your commands or save your work, and graphic commands such as `plot` are much more limited. If you get into this mode by mistake, you can just type the Maple command `quit;` to exit.

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.

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;
```

$$\frac{7}{2}$$

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

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

$$\frac{5159}{13}$$

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 may result in a syntax error message. Even if you don't get a syntax error, Maple will not treat the expression as you expect.

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);
```

$$b := \sqrt{2}$$

```
> evalf(b);
```

$$1.414213562$$

```
> evalf(b,47);
```

$$1.4142135623730950488016887242096980785696718754$$

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. The equals sign `=` is used to signify that two things are equal, not to set one to the other.

Notice the difference in the following two examples:

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

$$\frac{7381}{5040}$$

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

$$1.464484127$$

In both cases, we compute $\sum_{i=1}^{10}(2i)^{-1}$. However, in the second case, the presence of `1.0` indicates to Maple the terms in this computation are approximate, and gives you a decimal expansion of the result.

2.2 Maple for symbolic manipulations

We can ask Maple to give us a formula for the sum of the cube of the first n integers:

```
> sum(i^3,i=1..n);
```

$$\frac{1}{4}(n+1)^4 - \frac{1}{2}(n+1)^3 + \frac{1}{4}(n+1)^2$$

We can ask to factor the resulting polynomial in n:

```
> factor(%);
```

$$\frac{1}{4}n^2(n+1)^2$$

In this example, Maple's "ditto operator" `%` is used[3] 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);
```

$$\frac{1}{4}n^2(n+1)^2$$

One can expand or factor conveniently:

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

$$p := (x+1)^4(x-6)^3$$

```
> expand(%);
```

$$x^7 - 14x^6 + 42x^5 + 112x^4 - 287x^3 - 882x^2 - 756x - 216$$

```
> factor(%);
```

$$(x+1)^4(x-6)^3$$

We can use Maple to solve equations (if `b` still has the value $\sqrt{2}$ from the previous section, first *unassign* it by executing the command `b:='b'`):

[3] In Maple V release 4 and earlier, Maple used the double-quote `"` to refer to the result of the previous command instead of `%`. If you encounter an older Maple program, you might have to change this, among some other things.

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

$$quadeq := ax^2 + bx + c$$

```
> solve(quadeq=0,x);
```

$$\frac{1}{2}\frac{-\sqrt{2}+\sqrt{2-4ac}}{a}, \frac{1}{2}\frac{-\sqrt{2}-\sqrt{2-4ac}}{a}$$

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:[4]

```
> p:=x^10-1;
  q:=x-1;
  divide(p,q,b);
```

$$p := x^{10} - 1$$
$$q := x - 1$$
$$true$$

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`.[5] Maple replies with the statement `true` to indicate that the polynomial $q = x - 1$ divides the polynomial $p = x^{10} - 1$ exactly, and places the quotient in the third argument of the `divide` command. Thus, if you want to know the value of the quotient, you must see what the variable `b` stands for now:

```
> b;
```

$$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

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

```
> r:=x^2+2*x+3;
```

[4] In the worksheet interface, we can insert linebreaks by holding the *shift* key and the *return* key simultaneously. Maple does not execute what we type until we hit the return key unshifted.

[5] Of course, we could have done all of this in a single Maple command `divide(x^10-1, x-1, b);` if we had no intention of using `p` or `q` again.

$$r := x^2 + 2x + 3$$

```
> divide(p,r,c);
```

$$false$$

```
> quo(p,r,x);
```

$$x^8 - 2x^7 + x^6 + 4x^5 - 11x^4 + 10x^3 + 13x^2 - 56x + 73$$

```
> rem(p,r,x);
```

$$-220 + 22x$$

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:

```
> divide(p,r,b);
```

```
Error, wrong number (or type) of parameters in function divide
```

The above command is equivalent to `divide(p,r,x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1)`, which makes no sense at all. If we insist on calling the result of the division `b`, it would have been necessary to *unassign* the value of `b` first:

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

$$b := b$$

We could also have done this within the command itself, using `divide(p,r,'b')`.[6] Further discussion on this is given in section 4.2.

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

```
> solve({1*x+(1/2)*y+(1/6)*z =1,
        (1/2)*x+(1/6)*y+(1/24)*z =0,
        (1/6)*x+(1/24)*y+(1/120)*z =1},  {x,y,z} );
```

[6] In both cases, it is very important to use the proper quotation marks; using " or ‘ means a very different thing, and would not work. The quotation mark used here is the ‘single quote’ or ‘right quote’, and on most keyboards is on the same key as the double quoute, near the enter key.

$$\{x = 39, y = -216, z = 420\}$$

Maple also solves inequalities:

```
> solve(x^2-4*x-7>0,x);
```

$$\text{RealRange}(-\infty, \text{Open}(2 - \sqrt{11})), \text{RealRange}(\text{Open}(2 + \sqrt{11}), \infty)$$

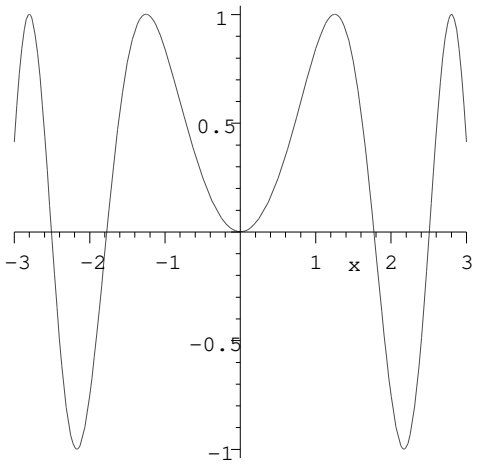
```
> solve(abs(1-x^2)<1/2,x);
```

$$\text{RealRange}\left(\text{Open}\left(-\frac{\sqrt{6}}{2}\right), \text{Open}\left(-\frac{\sqrt{2}}{2}\right)\right), \text{RealRange}\left(\text{Open}\left(\frac{\sqrt{2}}{2}\right), \text{Open}\left(\frac{\sqrt{6}}{2}\right)\right)$$

2.3 Graphing in Maple

Let us now use the graphics facilities of Maple. It is quite straightforward to draw the graph of a simple expression. For example, to graph the function $\sin(x^2)$ over the range $-3 \leq x \leq 3$, we do the following.

```
> plot(sin(x^2), x=-3..3);
```



Notice that within the worksheet interface, clicking the right mouse button on the displayed graph causes a menu to pop up that allows you to change some of its properties. Clicking the left mouse button causes a box to appear around the graph and the buttons on the toolbar to change. You can use the box to resize the graph, and the buttons to change attributes such as the style of the axes and the aspect ration.

In section 9 we shall discuss in further detail the graphing features of Maple.

3 The Maple worksheet

Maple's graphical interface allows you to keep a collection of commands and their results in one place, to save your work, and resume previous sessions. You can also add comments and text to your worksheet, to make it more readable and to describe the process of solving the problem. It also enables you to structure your worksheet into sections of related items.

3.1 Introducing the worksheet

As we have seen, you enter Maple commands at the `>` prompt, and the results appear in your worksheet. If you go back and change a Maple command and re-execute it, the result in the worksheet changes.

At the top of the screen is a menu bar (with entries like `File`, `Edit`, and so on): clicking on each of these words gives you a menu which allows you to affect your session in various ways. For example, clicking on the `File` menu and selecting `Save` will allow you to save your current worksheet for later use; selecting `Open` from the same menu allows you to load an existing worksheet.

Figure 2: The menu bar, button bar, and context bar.

Below the menu bar is a collection of buttons which are short-cuts for items found on the various menus. Clicking on one of these is much quicker than searching for the command on the proper menu. For example, the button that looks like a floppy disk ( ) saves the current worksheet, and the magnifying glasses ( ) adjust the zoom factor (or magnification) of the current worksheet. The stop sign interrupts a computation, which is very useful if you ask Maple to do something that takes a very long time, such as to find the factors of $2^{1000} - 1$.

Below this is the "context bar", which changes depending on what kind of item you currently have selected. For example, as mentioned in §2.3, when graphics are selected there are buttons to adjust the style of graphics (points or lines for 2-dimensional graphics, shading on 3-dimensional graphics), axes style, and so on. When a text item or comment section is chosen, selections for typeface, centering, and so on appear.

3.2 Worksheet basics: the front-end and the kernel

It is important to remember that the worksheet is only a record of a Maple session, not the session itself. Since you have the ability to edit and reorder it, it can sometimes appear to

contain inconsistent information. Here is a small example:

First, suppose we issue the following commands:

```
> one:= sin(x)^2 + cos(x)^2;
```

$$one := \sin(x)^2 + \cos(x)^2$$

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

$$0$$

The command `diff(one,x)` asks Maple to compute the derivative of the expression called `one` with respect to the variable `x`. The result is, of course, 0 because $\sin^2 x + \cos^2 x$ is a constant.

Now, we go back and change the `+` in the definition of `one` to a `-`, and execute the statement. However, we do not execute the `diff` command. Our worksheet now looks like this:

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

$$one := \sin(x)^2 - \cos(x)^2$$

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

$$0$$

This seems to contain an error (the derivative should be $4\sin(x)\cos(x)$). Obviously, this is because we we didn't execute all the statements. If you place your cursor on the line with `diff(one,x)` and hit the return key, the worksheet will be correct again.

This may seem like a silly discussion, but this apparently obvious example is to make an important point: Maple only "sees" your commands in the order you execute them, *not* in the order you see them in the worksheet. Also, if you load a worksheet from a previous session (or made by someone else), any assignments and results will not be accessible to the Maple session until you execute those statements, *even though the results appear on your screen.*

To help you understand why this is, we point out that a Maple session typically consists of two processes (or programs) which communicate with each other: the Maple kernel and the "front-end", or user interface. The front-end is the part that interacts with the user, accepting your commands, showing the results of your computations, and so on. The kernel is the part that does the actual computations. In certain situations (for example, when using a "parallel Maple kernel"), it is possible to have more than one Maple kernel associated with the same front-end.

3.3 Online Maple help

Maple has an extensive and very useful online help system, containing information on all of the commands, as well as a number of tutorials and example worksheets.[7] Spending some time exploring the help system is well worth the effort. There are a few ways to get into the system; one obvious one is to select from the `Help` menu (on the right of the menu bar). But if you know the specific item you want help on, you can also do this in a quicker way.

Suppose you want information on Maple's `factor` command. Enter the Maple command

```
> ?factor
```

(here the semicolon is optional). Then 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. You can also click in the help window, and use the arrow keys on your keyboard.

You should notice that at the end of the help window 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.

Usually there is also a section labeled "See Also", which points you to related commands. Each of these words is a link to another help page. It works like a web browser: clicking the underlined word takes you to the help page on that topic.[8] You can use the back button (⬅) to return to the previous help page.

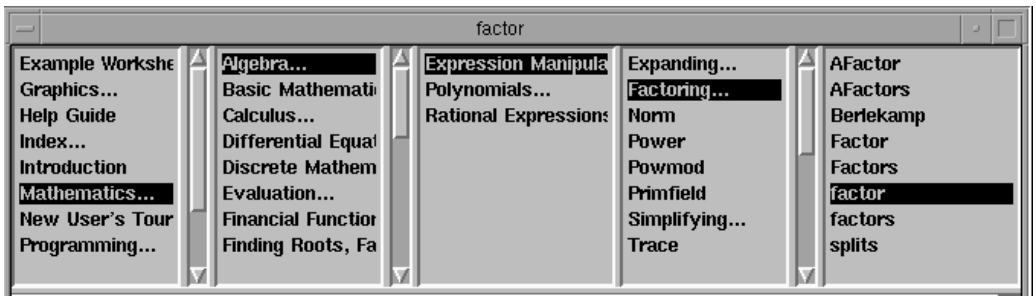


Figure 3: Maple's help index.

At the top of each help page is the help index (see Fig 3), which is a table of contents for the entire help system, organized by topic. You can use this to browse for commands you might not otherwise have known about.

Finally, on the `Help` menu are entries called "topic search" and "full text search". The first looks for help on a specific topic, and the latter searches the text of every page in the help system for the word you ask. Both of these can be valuable resources when used properly.

[7] There is even help on the help system. Select `Using Help` from the `Help` menu, or give the command `?helpguide`.

[8] If you prefer, holding the Control key while clicking on the link opens the page in a new help window, allowing you to view both at the same time.

3.4 Documenting and structuring your worksheet

Aside from the great advantage of making it easy to keep track of your commands and their results, the worksheet has some features of a word processor, allowing you to insert comments and text which describes your solution (or anything else you like). This can be very important, especially if you intend to show your worksheet to someone else (or even look at it yourself later).

To enter text into a worksheet, click on where you'd like it to go, and hit the text-mode button (it looks like T). Note that this button will convert a Maple input region (that is, one with a `>` prompt) to a text region— you'd probably prefer to *insert* your comments rather than change your commands to comments. To do this, either click before the `>` prompt, or select `Paragraph...Before` from the `Insert` menu.

In the text region, you can type whatever you like, using the context bar to adjust formatting such as the typeface, centering, etc. You can even include mathematical expressions in your text: click the "math input" button (Σ)[9] and type the Maple expression for the math you want. When you are done, use the text-mode button to return to entering text. You can also paste the output from a previously executed Maple command into the text area.

The worksheet interface also allows you to group parts of your worksheet into sections. These sections can contain paragraphs and Maple execution groups, and subsections. You can recognize a section by a large square bracket to the left, with a box at the top of it. Clicking on the box collapses the section, showing only the its title. Collapsing all sections allows you to see the structure of your document in outline form. To expand a collapsed section, click on the box next to the title.

For an extensive description of how to document and structure your worksheets, see the "Documenting your Work" section in the online help. This can be invoked with `?documenting`.

4 Assignments, Functions and Constants

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

4.1 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 4.4):

[9] Or select `Math Input` from the `Insert` menu.

```
> 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;
```

$$\frac{66}{7}$$

```
>  a-b^2;
```

$$-22$$

4.2 Variables and subsequent assignments: unassignments

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

```
> c;
```

$$\frac{22}{7}$$

```
> x;
```

$$x$$

If you specify `y` as an expression involving `x` and then assign a particular value to `x`, this value will get substituted into `y`:

```
> y:=sqrt(x+3);
```

$$y := \sqrt{x+3}$$

```
> x:=sin(w);
```

$$x := \sin(w)$$

```
> y;
```

$$\sqrt{\sin(w)+3}$$

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

```
> x:=cos(u);
```

$$x := \cos u$$

```
> y;
```

$$\sqrt{\cos u+3}$$

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';
  y;
```

$$x := x$$

$$\sqrt{x+3}$$

We can also tell Maple to forget entirely all assignments made so far using the `restart` command.

```
> restart;
  x; y; z;
```

$$x$$

$$y$$

$$z$$

Surrounding an expression in single quotes[10] tells Maple to delay evaluation of it. This is why `x:='x'` unassigns `x`— `x` is assigned to mean just `x` without any evaluation. Note that stopping evaluation does not stop simplification:

```
> 'x + 2 + 5';
```

$$x + 7$$

4.3 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)`.

A much more complete list can be found by asking Maple for help on the initially known functions; `?inifcns` does this.

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:

[10] Be careful! Most keyboards have three different kinds of quotation marks, and they each mean something different to Maple. The single quote (') delays evaluation, the double quote (") indicates a string, and the backquote (`) indicates a name (similar to a string).

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

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

Carefully notice the syntax. `f` is the name of the function and is written to the right side of `:=`. The function f sends the variable x to the expression $\sqrt{1+x^2}$; this is indicated by the arrow. The arrow `->` is made with the minus and greater-than 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);
```

$$1$$

$$\sqrt{2}$$

$$\sqrt{2}$$

$$\sqrt{2+9x^2+6x}$$

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);
```

$$f := x \to \sqrt{x-2}$$

It seemed like this worked, but in fact, the resulting `f(x)` does not work as expected. You will not be able to substitute into it or do anything you normally do with functions.[11]

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

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

$$2(x+4)^3$$

or to *delimit the arguments of a function.* In 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:

[11] Maple uses this notation for a "remember table"— you can use it to set specific values for the function. For example, if we first define f by `f:=x->sin(x)/x`, then asking for Maple for `f(0)` will give an error, since it is not a permissible value for the function. However, we can define $f(0) = 1$ with the command `f(0):=1`; after this, `f(0)` will be `1`, while $f(x) = \sin(x)/x$ for all other values of x.

```
> f(x);
```

$$\sqrt{1+x^2}$$

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

$$\sin(\sqrt{x+2}+\ln|x|)$$

4.4 Special constants and reserved names

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

```
> constants;
```

$$false, \gamma, \infty, true, Catalan, FAIL, \pi$$

The names of these constants are `false`, `gamma`, `infinity`, `true`, `Catalan`, `FAIL` and `Pi`, respectively. Be careful! Maple is case sensitive. Therefore `pi` is not the same as `Pi`. For example,

```
> cos(Pi/4);
```

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

```
> cos(pi/4);
```

$$\cos\left(\frac{1}{4}\pi\right)$$

Maple also uses `I` to represent the complex number $\sqrt{-1}$. Earlier versions used `E` for `exp(1)`; this is no longer the case. If you'd like to add `E` as a constant, you can do the following:

```
> E:=exp(1);
  protect(E);
  constants := constants, E;
```

$$E := e$$

$$constants := false, \gamma, \infty, true, Catalan, FAIL, \pi, e$$

While protecting your new constant `E` will stop you from accidentally redefining it, adding `E` to the list of known constants doesn't really affect things very much.

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 cannot use any of the following 30 keywords:

```
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
```

You can check this list for yourself using the help system by giving the command `?keywords`.

Maple also starts with a large number of predefined functions and constants. Many of these can be listed with `inifcns` and `ininames`. If you try to use one by mistake, you will receive an error:

```
> gamma:=x^2+5;
```

```
Error, attempting to assign to name `gamma` which is protected
```

If you insist, you can use `unprotect` to allow you to assign to this name. But **don't do this** unless you know what you are doing; this changes the meaning of this name even when it is used internally, and can have unexpected consequences.

5 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);
```

$$\frac{3}{4}$$

Maple can also deal with limits which do not exist, for example $\lim_{x\to 0} \frac{1}{x}$:

```
> 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);
  m:=x->slope(1,1,x,x^2);
```

$$slope := (x1, y1, x2, y2) \to \frac{y2 - y1}{x2 - x1}$$

$$m := x \to slope(1, 1, x, x^2)$$

```
> 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);
```

$$\infty$$

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

$$-\infty$$

```
> 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);
```

$$\frac{\pi}{2}$$

```
> 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';`. The previous example will fail if `y` still has the value assigned to it in section 4.2.

6 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);
```

$$ae^{ax}$$

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 in terms of the functions and their derivatives:

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

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

6.1 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 expression `3*sin(x)` and then differentiates the result to obtain the second derivative. 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);
```

$$2\frac{e^{-2x}}{x^3}+6\frac{e^{-2x}}{x^2}+12\frac{e^{-2x}}{x}-8\ln(x)e^{-2x}$$

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

$$2e^{x}y\cos(z)$$

6.2 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;
  deq:=diff(eq,x);
  solve(deq,diff(y(x),x));
```

$$eq := x^2y(x) - 3y(x)^3x = 0;$$

$$deq := 2xy(x) + x^2\left(\frac{\partial}{\partial x}y(x)\right) - 9y(x)^2\left(\frac{\partial}{\partial x}y(x)\right) - 3y(x)^3 = 0$$

$$-\frac{2xy(x) - 3y(x)^3}{x^2 - 9y(x)^2x}$$

```
> subs(y(x)=1,x=3,%);
```

$$\frac{1}{6}$$

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 8 to learn about the `subs` command).[12]

Maple actually has a built-in command to do implicit differentiation, `implicitdiff`. Thus, we could have done this same problem more concisely using the single command

```
> subs({y=1,x=3},  implicitdiff(x^2y-3y^3x=0,y,x);
```

$$\frac{1}{6}$$

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

[12] 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.

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);
```

$$f := x^3 e^y - \sin(xy)$$

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

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

7 The int (and Int) command

7.1 Symbolic integrals

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);
```

$$\frac{e^{ax}}{a}$$

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

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

$$2e^2 - 2$$

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);
  ```

  $$\int \ln(\sin(\sqrt{x^{12} - 5x^7 + 50x + 2}))\, dx$$

- 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);
  ```

  $$Si(2x)$$

  Here `Si` is the *special name* of one of these functions which appear frequently in mathematical physics. To learn more about it, ask Maple for help on it with the command `?Si`. This 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.

- Sometimes, a result may be expressed in terms of the roots of a polynomial which does not factor over the rationals. For example,

```
> int(1/(x^8 +1),x);
```

$$\sum_{_R=\mathrm{RootOf}(1+16777216_Z^8)} _R \ln(x+8_R)$$

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

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);
```

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

The inert form `Int` is also very useful in many situations when you wish to delay the evaluation of an integral, as we shall see below.

7.2 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 we are using the inert form of the integration command, `Int`. This prevents Maple from attempting to evaluate the integral symbolically and then applying `evalf` to the answer. In many cases, this can save a huge amount of time, because Maple will work very hard

to try to compute the symbolic form of the integral. For example, approximating $\int_0^1 e^{\sin(x)}\,dx$ with the command

```
> evalf(int(exp(sin(x))),x=0..1);
```

took more than 10 times the amount of time needed to execute

```
> evalf(Int(exp(sin(x))),x=0..1);
```

More complicated integrals can have even more dramatic differences.

7.3 Approximations through Riemann Sums

Maple can also compute 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):
```

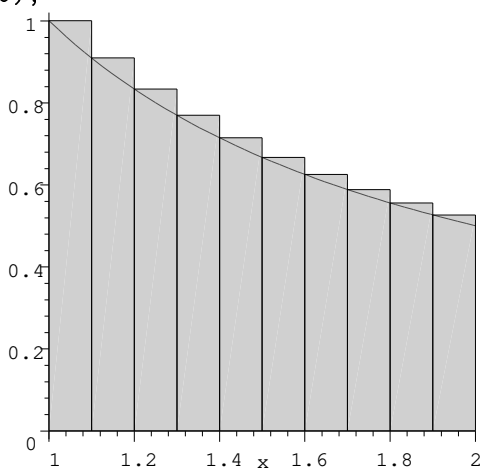
Once loaded, you can play with it. For example, if you want to use the left sum approximation to an integral, the height of each rectangle is determined by the value of the function at the left side of each interval. You may specify the number of intervals you wish to use. If you do not, Maple will use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

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

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

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);
```



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);
```

$$\frac{1}{3}\sin(1) + \frac{5}{3}\sin(25) + \frac{4}{3}\left(\sum_{i=1}^{2} 2i\sin(4i^2)\right) + \frac{2}{3}\left(\sum_{i=1}^{1}(1+2i)\sin((1+2i^2)^2)\right)$$

7.4 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

$$\int_2^3 \int_0^y x^2y^3 dxdy\,.$$

8 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);
```

$$3u + 1 + \sin(8\pi q)$$

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:

- You need not fear *circular substitutions*:

  ```
  > z;
  ```

  $$x + \sin(2h)$$

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

  $$2x + \sin(2h)$$

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

  ```
  > subs(h=x,x=z,z);
  ```

  $$x + \sin(2h) + \sin(2x + 2\sin(2h))$$

  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);
```

$$x + \sin(2h) + \sin(2x)$$

- 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. In other cases, you might consider using `algsubs`, which allows you to substitute one algebraic expression for another.

- `subs` does what is called "syntactic substitution"; it substitutes one set of symbols for another. If you are using it to evaluate an expression with certain values "plugged in", using `eval` might be more appropriate. Here is an example that should illustrate the distinction:

```
> integral := int(f(t,a), t=a..x);
```

$$\int_a^x f(t,a)\,dt$$

```
> eval(integral,{t=0,a=1});
```

$$\int_1^x f(t,1)\,dt$$

```
> subs({t=0,a=1},integral);
```

$$\int f(0,1)\,d0 = 1..x$$

The seemingly nonsensical answer using `subs` above is because Maple interpreted the expression as meaning `int(f(0,1),0=1..x)` — what you get by replacing `t` by `0` and `a` by `1` without trying to interpret the expression.

8.1 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, too many levels of recursion
```

In earlier versions of Maple, this action would have crashed your session.

On the other hand,

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

$$x := 4$$

```
> y;
```

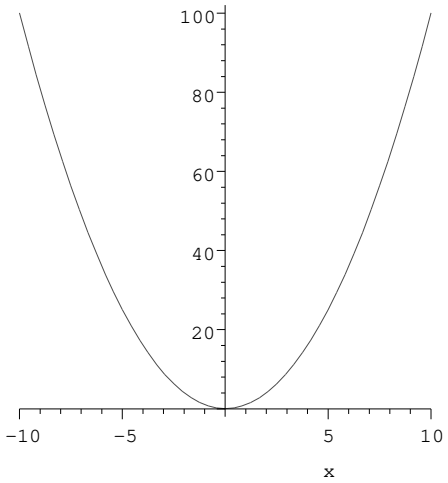
70

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

9 Plotting with Maple

Let us now use the graphics facilities of Maple:

```
> plot(x^2, x=-10..10);
```



Maple displays the graph of the function x^2 from $x = -10$ to $x = 10$ as requested. In the worksheet interface, clicking the right mouse button on the graph pops up a menu with various options which allow you to manipulate your graph, and/or change its appearance in a suitable way. Clicking the left button "selects" the graphic, drawing a black box with "handles" around it. You can drag the mouse button on these to resize the plot. Also, the coordinates of the point where you clicked appear in the upper left of the worksheet. Selecting a graphic also changes the context bar: shortcuts for manipulating the axes, aspect ratio, and plotting style appear as buttons along the top.

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. There are many options to the `plot` command; while we will mention a few of them, the on-line help system has all the details.

Try both of the two similar plot commands below:

```
> plot(x^2-x, x=-1..2, y=-1..2);
  plot(x^2-x, x=-1..2,   -1..2);
```

In both cases, the plot is displayed for values of `y` between -1 and 2, but in the first, the variable `y` is explicitly named in the command, and the vertical axis gets a label. Whenever you are plotting an expression, the variable in the domain of the function *must* be specified.

9.1 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);
  m:=x->slope(1,1,x,x^2);
```

$$slope := (x1, y1, x2, y2) \rightarrow \frac{y2 - y1}{x2 - x1}$$

$$m := x \rightarrow slope(1, 1, x, x^2)$$

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 `x^2`.

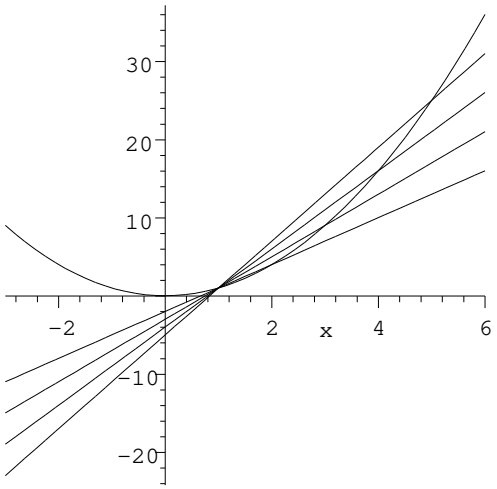
```
> line1:=m(2)*(x-1)+1;
  line2:=m(3)*(x-1)+1;
  line3:=m(4)*(x-1)+1;
  line4:=m(5)*(x-1)+1;
  plot({x^2,line1,line2,line3,line4},x=-3..6);
```

$$line1 := 3x - 2$$

$$line2 := 4x - 3$$

$$line3 := 5x - 4$$

$$line4 := 6x - 5$$



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.[13]

[13] We could have done this all in a single plot command, either explicitly typing each of the lines, or using `seq` to generate them: `plot({x^2, seq( m(i)*(x-1)+1, i=2..5)}, x=-3..6);`

9.2 Fancier plotting

The `plot` command is very powerful, and you should look for details about it using the online help facility; the command `?plot` will bring this up. There are also several related pages, and some instructive example worksheets that are well worth looking at.

To make you aware of some of its features, we will discuss a few of 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. In the worksheet interface, you can switch from one style to the other by first clicking the left mouse button on the graph, and then hitting the relevant button on the buttonbar.

  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 may notice that they will not usually be equally spaced, but concentrated in areas where the graph curves more. You can insist that more points be plotted by using the `numpoints` option:

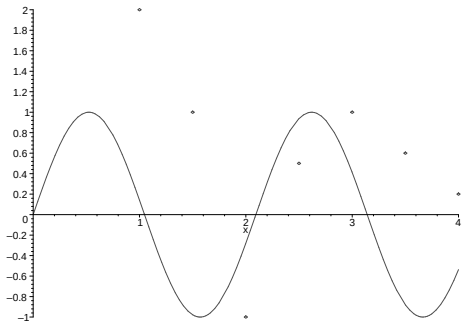
  ```
  > plot(x^2,x=-2..2,style=POINT,numpoints=150);
  ```

- Several more specialized 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);
  ```

  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 (This is the utility of `textplot`: to label 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. Note that except for the third command, we end with a colon (:) in order to suppress the Maple output.

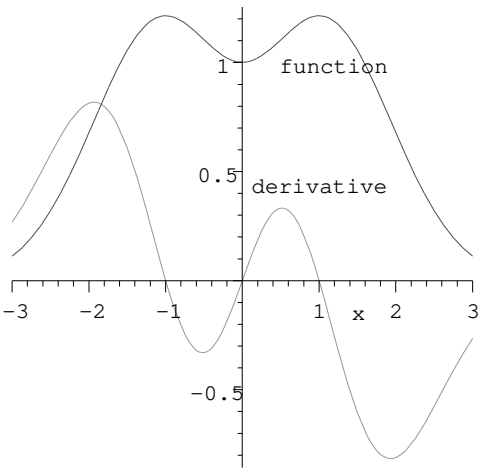
```
> plot1:=plot([points],style=POINT):
  plot2:=plot(sin(3*x),x=0..4):
  display({plot1,plot2},view=[0..4,-1..2]);
```



The `view` option 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});
```



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);
```

```
Plotting error, empty plot
```

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

As you see, the first gives an error; the second gives a plot with nothing plotted on it.

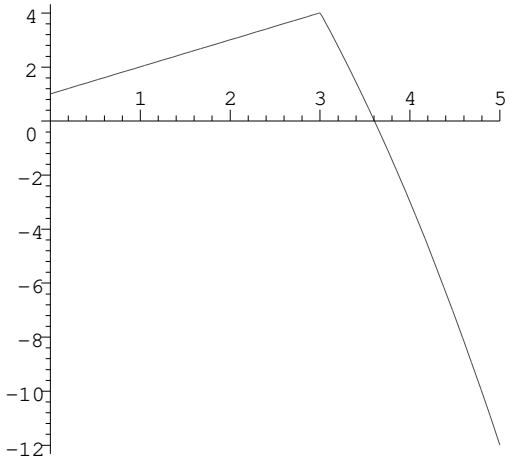
If you have a function defined using an *if-then* clause, you must use the function plotting command:

```
> 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 expression `f(x)` before it has a value for `x`. One way around this[14] is to use the command

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



You can see from the graph that the function `f` is probably continuous but not differentiable at `x=3`.

10 Exercises

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

2. Input the following as a Maple function:

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

[14] We could also delay evaluation of `f(x)` using quotes, with `plot('f(x)', x=0..5);`. Alternatively, we could define `f` using `piecewise`, as `f:=x-> piecewise(x<3, x+1, -x^2+13);`.

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

If the Curve Fits, Wear It

It is not uncommon in science and mathematics to have a collection of data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

for which we believe there is a functional relationship between the x_i and the y_i. Often, we want to get some idea of what might be happening at points other than those for which we have data. That is, we want to find some f so that $f(x_i) = y_i$ for each of our points. Of course, in order to do this, we will need to have some knowledge (or make some guesses) about the function f, or sometimes about how precisely the points are known. In this chapter, we'll look at several different approaches to this problem of fitting a curve through (or near) some given data.

1 Interpolation

First, we will look at the situation where we the data points are assumed to be known exactly (or at least to good enough precision that we ignore any errors), and we want to find a curve of some type is chosen that passes through each of the data points. This practice is called *interpolation*. In the latter part of this chapter (beginning with §2), we will relax the restriction that the curve pass exactly through the points.

Of course, there are serious restrictions involved in interpolation. Since a line is determined by two points, if we have more points, we must either use a higher degree polynomial, or use several line segments. If we have n data points we wish to interpolate, either we can fit a polynomial of degree $n-1$ or find $n-1$ line segments which "connect the dots". We'll address both of these in this section, as well as a method which is a cross between the two approaches.

1.1 Polynomial Interpolation

Given n data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, there is a unique polynomial of degree $n-1$ passing through them. Finding this polynomial is just a matter of solving the n linear equations

for the coefficients. For example, suppose we have 4 data points (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4). There is exactly one degree 3 polynomial $p(x) = ax^3 + bx^2 + cx + d$ which passes through all four points. Plugging in our data yields the equations

$$\begin{aligned} ax_1^3 + bx_1^2 + cx_1 + d &= y_1 \\ ax_2^3 + bx_2^2 + cx_2 + d &= y_2 \\ ax_3^3 + bx_3^2 + cx_3 + d &= y_3 \\ ax_4^3 + bx_4^2 + cx_4 + d &= y_4 \end{aligned}$$

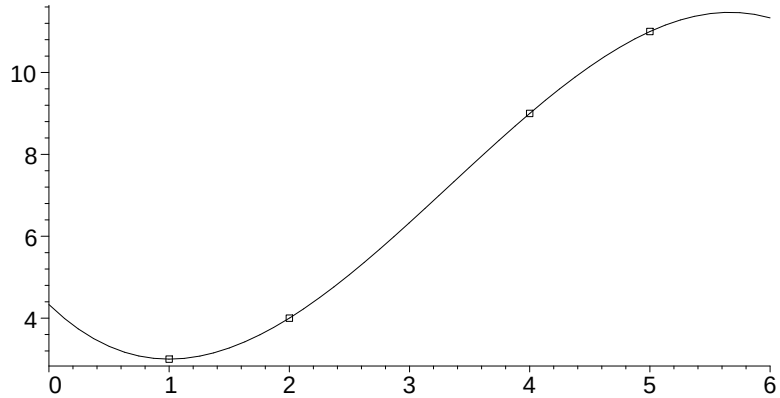
These are easily solved for the coefficients a, b, c, and d.

Maple 7[1] has a built-in command called `PolynomialInterpolation` to do this automatically for us, as part of the `CurveFitting` package. We'll do a brief example.

```
> data:=[[1,3],[2,4],[4,9],[5,11]]:

  with(CurveFitting):
  cubfit:=PolynomialInterpolation(data,x);
```

$$cubfit := -\frac{1}{6}x^3 + \frac{5}{3}x^2 - \frac{17}{6}x + \frac{13}{3}$$

```
> plots[display](
    plot(data,style=point,color=black,symbol=BOX),
    plot(cubfit,x=0..6,thickness=2));
```



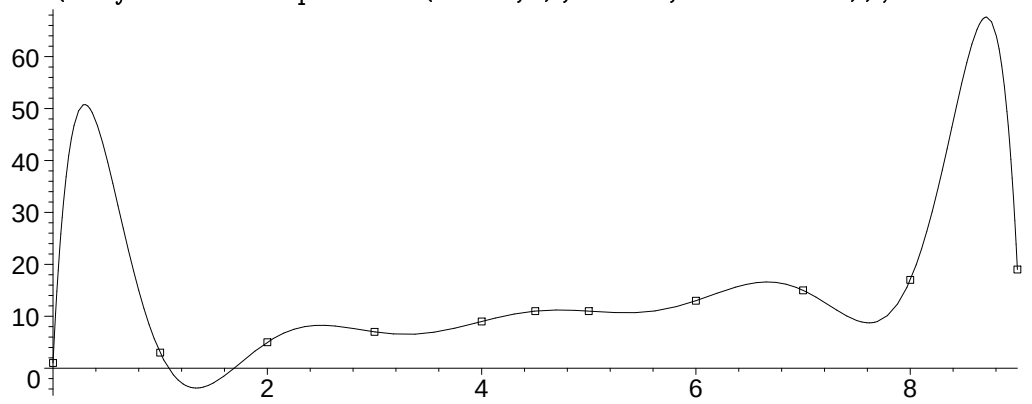
That seems to have worked pretty well. However, we should remark that polynomials are somewhat inflexible: minor variations in the data can have drastic effects on the behavior between the points. To emphasize this, we'll generate 10 points along the line $y = 2x + 1$, with one more point in the middle that is a bit above the line.

```
> data2:=[seq([i,2*i+1],i=0..4),[9/2,11],seq([i,2*i+1],i=5..9)];
```

[1] earlier versions of Maple have a similar command called `interp` with slightly different syntax.

$$data2 := [[0,\ 1],\ [1,\ 3],\ [2,\ 5],\ [3,\ 7],\ [4,\ 9],\ [\frac{9}{2},\ 11],\ [5,\ 11],\ [6,\ 13],\ [7,\ 15],\ [8,\ 17],\ [9,\ 19]]$$

```
> plots[display](
    plot(data2,style=point,color=black,symbol=BOX),
    plot(PolynomialInterpolation(data2,x),x=0..9,thickness=2));
```



This doesn't look much like a straight line, does it? Even though there is only one data point which is 1 unit above the line $y = 2x + 1$, at $x = 1/2$ the polynomial is about 50 units above this line.

1.2 Connect-the-Dots and Splines

As we said at the start of this section, another choice is not to use a single function, but to use several segments which connect the dots. The result would be a piecewise-affine function consisting of several line segments defined on a number of intervals. In the example using `data2` from the previous section, this function would be

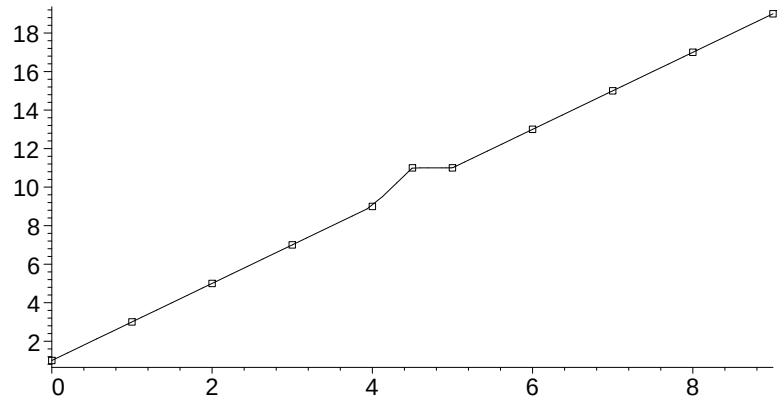
$$f(x) = \begin{cases} 2x+1 & \text{if } x \le 4 \\ 4x-7 & \text{if } 4 < x \le \frac{9}{2} \\ 11 & \text{if } \frac{9}{2} < x \le 6 \\ 2x+1 & \text{if } 6 < x \end{cases}$$

Although writing a maple procedure to figure this out for us would not be hard, such a piecewise affine curve is a version of something called a spline, which we will discuss more soon. This type of connect-the-dots curve is a linear (or degree 1) spline, and Maple can find it using the `Spline` command out of the `CurveFitting`[2] package.

```
> lspline:=Spline(data2,x,degree=1):
```

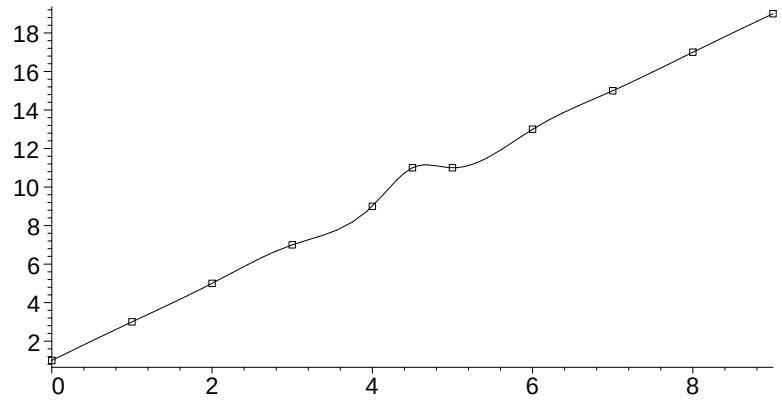
[2]versions prior to Maple 7 have a similar command called `spline` which takes slightly different arguments.

```
> plots[display](
    plot(data2,style=point,color=black,symbol=BOX),
    plot(lspline,x=0..9,thickness=2));
```



If we leave off the `degree=1`, Maple gives us a cubic spline instead:

```
> plots[display](
    plot(data2,style=point,color=black,symbol=BOX),
    plot(Spline(data2,x),x=0..9,thickness=2));
```



Notice how the cubic spline fits the points about as well as the "connect-the-dots" piecewise affine curve, but has no corners. What is this curve that Maple is showing us?

To make a spline, instead of connecting each pair of points by a straight segment, we put in a polynomial of degree d. We must ensure that the polynomial passes through the two endpoints of each segment, but this still leaves us an additional $d-1$ conditions per polynomial segment. The entire curve can be made smoother by choosing each polynomial segment so that the first $d-1$ derivatives at one endpoint agree with the derivatives of the previous segment. Since there is no segment before the first or after the last, this leaves $d-1$ more conditions still to specify. For a "natural spline", the high order derivatives at the endpoints are set to zero:

$$p^{(d-1)}(x_1) = p^{(d-2)}(x_1) = \ldots = p^{(\frac{d-1}{2})}(x_1) = 0$$

and

$$p^{(d-1)}(x_n) = p^{(d-2)}(x_n) = \ldots = p^{(\frac{d-1}{2})}(x_n) = 0$$

(if $d-1$ is odd, we have one fewer zero derivative at x_n than at x_1). Thus, for a cubic spline, the second derivatives vanish at eachpoint, and the polynomial segments have the same value, slope, and concavity at each interior point.

You might think that using higher degree polynomials in between would give a better appearance, but the same issues that arose in earlier begin to show up. Note that if we have n data points, fitting a spline of degree $n-1$ is exactly the same as the interpolating polynomial of degree $n-1$. Cubic splines are the most widely used, because they look quite smooth to the eye, but have enough freedom that they don't have to wiggle too much in order to pass through each data point. Many computer drawing programs have the ability to fit cubic splines through a set of points.

2 When the data is approximate

Now we relax the restriction that the function we are searching for must pass exactly through each of the data points. This is the typical situation in science, where we make a measurement or do an experiment to gather our y values from the input x values.

More specifically, let's 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 parameters $c_1,\ldots,c_k$ 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_{ij} and y_i are the measurements of x_j and y in the i^{th} experiment. These experimental results gives us information about the unknown coefficients c_i.

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 $(f_{c_1,\ldots,c_k}(x_1, x_2, \ldots, x_m) - y)^2$. 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 $c_1, \ldots, c_k$.

We should remark that in some cases, we might want to use $E(c_1, \ldots, c_k)/n$, the *mean-squared error*. This will not change the answer we get (because the minimum occurs at the same value of c), but does allow us to compare sets of data with different numbers of data points.

3 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). Our problem is to find the values of m and b will best fit this data. Or to put it simply and geometrically, what is the straight line that best fit the information contained in the data (x_1, y_1), ..., (x_n, y_n)?.

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

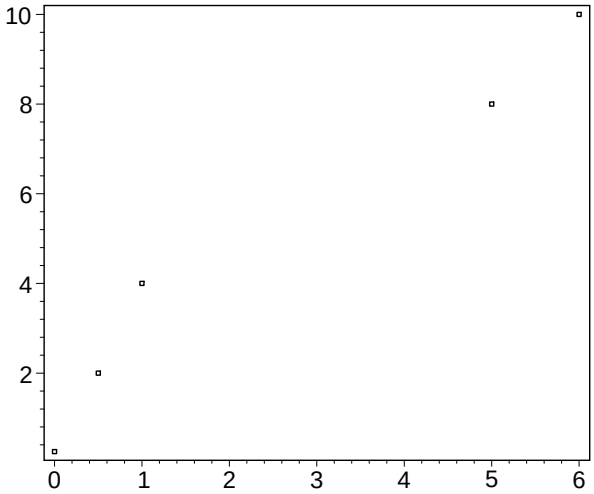
$$E(m, b) = \sum_{i=1}^{n} (mx_i + b - y_i)^2 \,.$$

The problem then becomes that of 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.

We can use the `LeastSquares` command from the `CurveFitting` package[3]

```
> CurveFitting[LeastSquares](pts,x);
```

$$1.271370968 + 1.431451613x$$

This says that the best values of m and b are $m = 1.431451613$ and $b = 1.271370968$, respectively. Now let's go through the steps ourselves, to better understand the process.

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

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

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

[3] As we noted before, the `CurveFitting` package was introduced in Maple 7. Earlier versions of Maple (as well as Maple 7) can accomplish the same thing using the `fit` command from the `stats` package, as follows.

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

As you can see, the syntax is a little different.

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

$$19.5625$$

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,pts);
```

$$3.125000000$$

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,pts), 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,pts), m)=0, diff( E(m,b,pts), 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 built-in version.

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.[4]

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

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

```
> m;
```

$$1.431451613$$

[4]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 this operator instead.

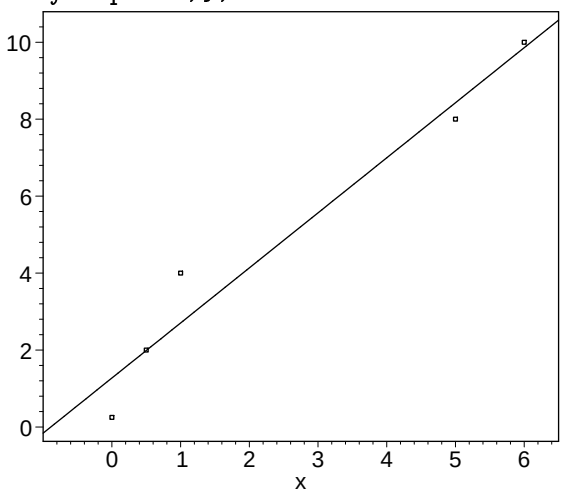
```
> E(m,b,pts);
```

$$2.929334677$$

The value of $E(m, b, pts)$ 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):
  display({plot(m*x+b, x=-1..6.5, axes=boxed),
           plot(pts, style=point)}):
```



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 resulting 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, and the discrepancy above is attributable to round-off errors..

As a final remark, we observe that if you now execute the command `diff( E(m,b,pts),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 $E(m, b, pts)$ for general m and b, you will have to unassign the values of m and b first:

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

4 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 fitting package.

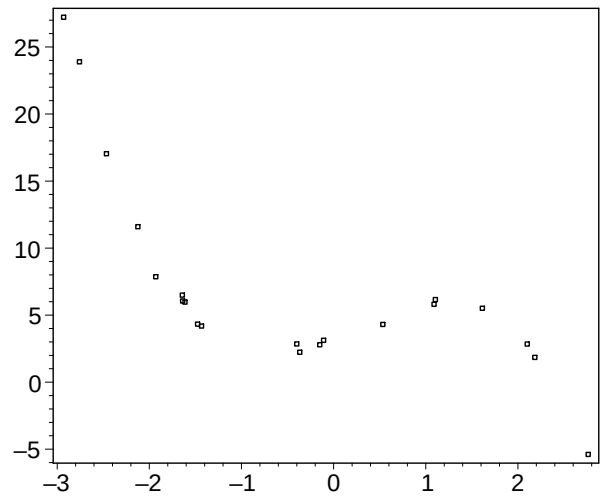
We randomly generate a set of 21 data points to be fitted to a cubic using a Maple program. (Don't worry about how this program works just now. We'll get into those details later).

```
> 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);
```



Now 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;
```

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

and then define the error function:

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

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

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

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

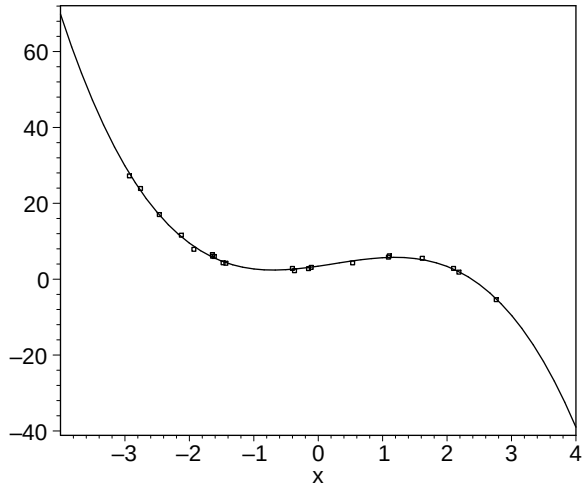
Of course, we do not see the answer, but the resulting values have already being assigned to the coefficients a, b, c, d:

```
> 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(CurveFitting):
  LeastSquares(data,x,curve=A*x^3+B*x^2+C*x+D);
```

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

For this particular version of least square fitting, we may once again interpret the error function E geometrically as 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 why it is of necessity a global minimum.

5 Fitting other types of funtions

In the previous constructions, we dealt with functions that depended linearly upon the parameters to be fitted. We found them by solving a linear system of equations, a relatively easy task.

However, as explained at the beginning of this chapter, the same scheme works equally

well for any function. In the general case, though, the resulting system that we must solve to find the best fit could depend non-linearly on the parameters, and the mere existence of solutions to such systems is not a trivial problem to settle. Take for instance, a function such as $y = \sum_{i=1}^{n} \sin(m_i x)$. Assuming measurements $(x_1, y_1), \ldots, (x_k, y_k)$, we may take as our error the function

$$E(m_1, \ldots, m_n) = \sum_{j=1}^{k} (y_j - \sum_{i=1}^{n} \sin(m_i x_j))^2 .$$

Its partial derivatives are quite easy to calculate, but it is far from clear if there are values of $m_1, \ldots, m_n$ where they all vanish simultaneously. For a particular set of values, we may ask Maple to solve the resulting system and get absolutely nothing. This either indicates an inability of Maple to handle such a system, or worse yet, the fact that such a system has no solution at all. Even when the latter happens, Maple itself might not be able to tell us so.

Sometimes, even if the function does not depend linearly on the coefficients, it can 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

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

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.

6 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.

We assume here that there is a routine called `circle_pts` which gives us points which lie near a circle of unknown radius and center. The one we use here is similar to that in §4, and is defined in `lsq_data.txt`.

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

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

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

$$E := (a,\, b,\, r,\, pts) \rightarrow \sum_{i=1}^{\mathrm{nops}(pts)} \varepsilon(pts_i,\, a,\, b,\, r)^2$$

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

$$\begin{aligned} sol := [\quad & \{r = 0.,\, b = 8.598845417 - 5.760818468\, I,\, a = 10.74842676 - 3.044017219\, I\}, \\ & \{r = 0.,\, a = 10.74842676 + 3.044017219\, I,\, b = 8.598845417 + 5.760818468\, I\}, \\ & \{r = 0.,\, b = 9.123213697,\, a = 8.304601815\}, \\ & \{r = 0.,\, a = 10.50158731 + 4.256560324\, I,\, b = 11.57538622 - 5.447431926\, I\}, \\ & \{b = 11.57538622 + 5.447431926\, I,\, r = 0.,\, a = 10.50158731 - 4.256560324\, I\}, \\ & \{a = 8.224446708,\, r = -4.468796472,\, b = 8.538288341\}, \\ & \{a = 8.224446708,\, b = 8.538288341,\, r = 4.468796472\}] \end{aligned}$$

Here we find a number of critical points for the function E, but from physical considerations, the only reasonable choice is the circle for which $r > 0$.

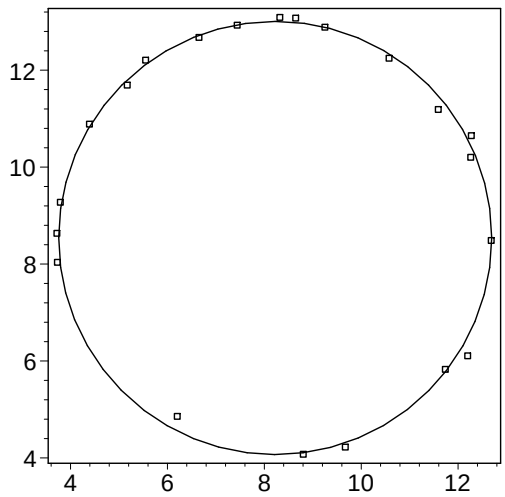
This is the seventh entry in the above list, so we could refer to it just by `sol[7]`. However, with a different set of data, it might be the second solution, or the fourth, etc. To ensure that we always get the right one, no matter what the order is, we can use Maple's `select` command to pull out the one with $r > 0$. This looks more daunting than it really is: we just define a function that returns `true` if that $r > 0$ for that solution, and `false` if not. Then `select` returns only those elements for which the function is true. We need to use `op`, because we have a list of sets, and we just want the one answer.

```
> goodsol:=op(select(s->if (subs(s,r)>0) then true else false fi,sol));
```

$$goodsol := \{a = 8.224446708,\ b = 8.538288341,\ r = 4.468796472\}$$

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.

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



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?

7 Robust fitting

Let us return momentarily to the problem discussed in §3: given some data $(x_1, y_1), \ldots, (x_n, y_n)$, we found the best line that fits it. This was accomplished by measuring the square of the *vertical* distance from each data point and the line $y = mx+b$, which led us to consider the error function

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

The problem was solved by finding the values of m and b where E achieves its minimum.

However, the square of the vertical distance is only one of many reasonable ways of measuring the distance from the data points to the line $mx + b$. For example, it is perfectly reasonable to consider instead the expression $1 + (mx_i + b - y_i)^2$. If (x_i, y_i) is on the line, this value would be equal to 1, and its logarithm would then be zero. Thus, the function

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

is also a reasonable way of measuring the distance from the data points to the line $y = mx + b$, and its minimum would produce a best fit line in this other sense. Similarly, We may argue that the function

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

is another reasonable alternative, though this time E is not even differentiable in the region of interest (if a data point (x_i, y_i) happens to be exactly on the line, $mx_i + b - y_i = 0$ and the absolute value is not differentiable at 0).

There is a priori no particular reason to prefer one error function over another. And for each choice of such we could end up with a problem whose solution exists and produces a line that best fit the data. Then we could compute the value of the error function for the best fit line, value that depends upon the choice of function made. A *canonical* way of choosing the best error function could be that for which this number is the smallest. But finding such a function is quite a difficult, if not impossible, problem to solve. If we limit our attention to *linear estimators*, in a sense to be explained in the last section of this chapter, the solution obtained in §3 is optimal. In here, we pursue further the problem of finding the best fit according to the two error functions introduced above.

Notice that if $\epsilon_i = mx_i + b - y_i$, data with large errors ϵ_i excert a huge influence on the original error function of §3 (because we use $\sum \epsilon_i^2$). The two error functions mentioned above

grow linearly (or sub-linearly) with $|\epsilon_i|$, causing the tails to count much less heavily. We will see this when comparing the different results.

In order to solve the problem now at hand, we first generate some "noisy" data. We load the routines in `lsq_data.txt`:

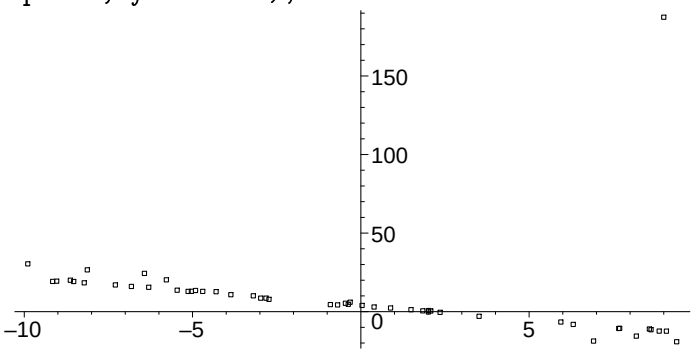
```
> read(`lsq_data.txt`);
```

```
defined line_pts(), bad_line_pts(), quadratic_pts(), cubic_pts(), and circle_pts()
```

The data and its graphical visualization can then be obtained by:

```
> pts:=bad_line_pts():
```

```
> plot(pts,style=point,symbol=box);
```



You should notice that while most of the data lies fairly near a line, there is one point which is extremely far away from the rest of the data.

In order to compare results, let us first find the line given by least squares. First, we define a function ϵ, which gives the signed vertical distance between a point `pt` and a line of slope `m` and intercept `b`

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

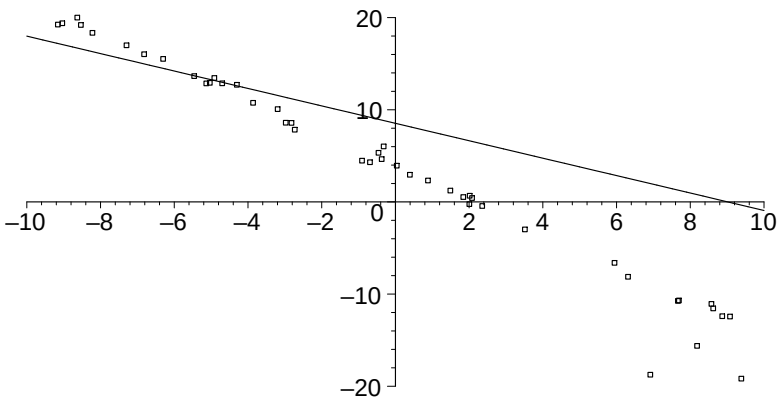
$$\varepsilon := (pt,\, m,\, b) \rightarrow m\, pt_1 + b - pt_2$$

Now we compute the regular least squares fit.

```
> H:=(m,b,pts)->sum(epsilon(pts[i],m,b))^2,i=1..nops(pts)):
  sol:=solve({diff(H(m,b,pts),m)=0, diff(H(m,b,pts),b)=0},{m,b});
```

$$sol := \{m = -.9455490512,\, b = 8.528125863\}$$

```
> display({plot(pts,style=point,symbol=box),plot(subs(sol,m*x+b),x=-10..10)},
          view=-20..20);
```



In the above, we restricted our attention to the region $-20 < y < 20$, where the line and the majority of the data lies. As you can see, the fit to the majority of the data is very poor, because of the influence exerted by the anomalous data point.

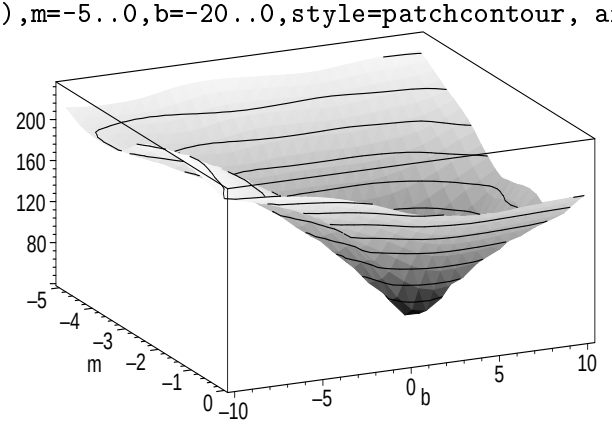
Now, let's try again, this time minimizing $\ln(1+\epsilon_i^2/2)$, which behaves like ϵ^2 for small errors, but grows very slowly for large ones.

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

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

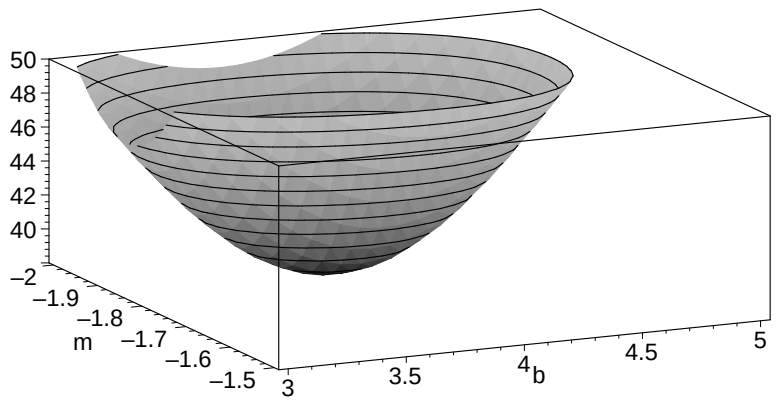
Here's what the functional we want to minimize looks like:

```
> plot3d(R(m,b,pts),m=-5..0,b=-20..0,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. By examining the plot and zooming in on the minimum, we can choose an appropriate region.

```
> plot3d(R(m,b,pts),m=-2..-1.5,b=3..5,view=38..50, axes=boxed);
```



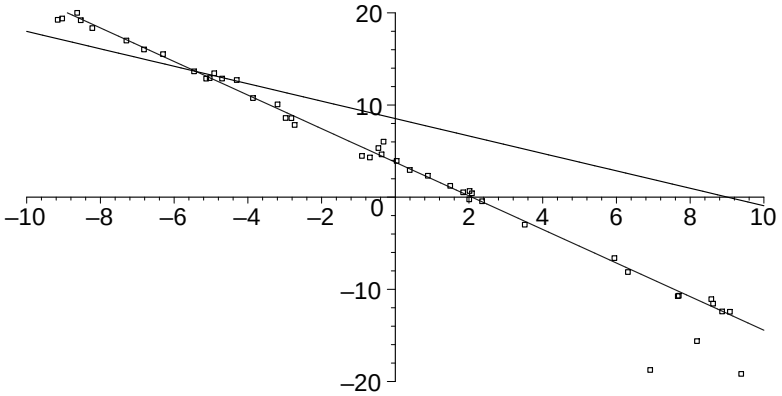
Once we have an appropriate region, we can ask Maple to look for the solution numerically using `fsolve`.

```
> rs:=fsolve( {diff(R(m,b,pts),m)=0, diff(R(m,b,pts),b)=0},
              {m,b},m=-2..-1.5, b=3..5);
```

$$rs := \{m = -1.822071903,\ b = 3.789519722\}$$

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=-20..20);
```



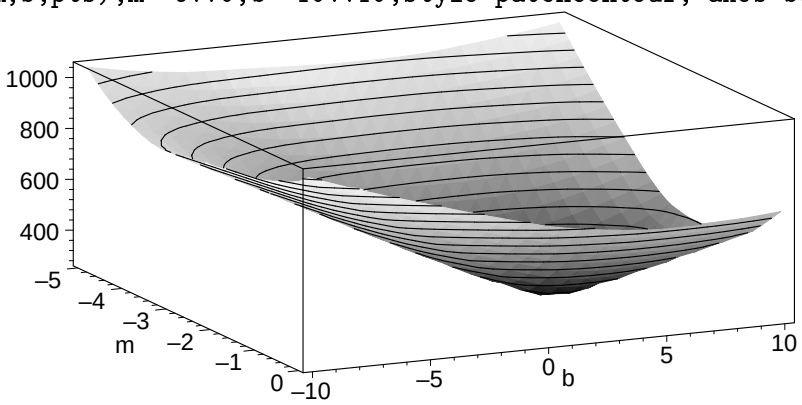
Let us now try to minimize $\sum |\epsilon_i|$. Since this function is not even differentiable near $\epsilon_i = 0$, we have to work harder to get less.

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

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

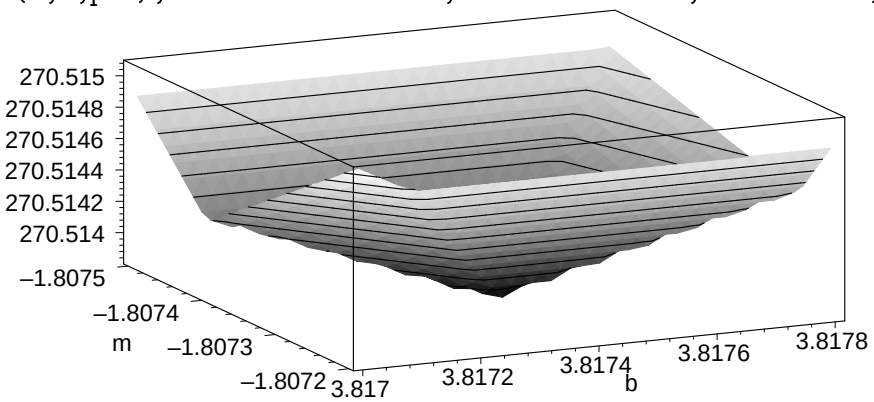
Displaying the graph of this function gives us an idea about where its minimum lies:

```
> plot3d(A(m,b,pts),m=-5..0,b=-10..10,style=patchcontour, axes=boxed);
```



But coercing Maple into finding a numerical approximation to it is not so easy. There *are* reliable and efficient ways to determine the location of the minimum to any precision (at least given some hints), but they are inappropriate for the scope of this chapter, and will not be considered here. Instead we may zero in on the minimum by looking at where the lowest point is and repeatedly restricting the domain. After several iterations, we get the following.

```
> plot3d(A(m,b,pts),m=-1.0875..-1.0872,b=3.817..3.8178, axes=boxed);
```



This leads us to a choice of $m = -1.80733$, $b = 3.8174$ as our "best" line. This is almost (but not quite) the same line found using our distance $R(m, b)$.

8 A nod toward statistics

In the problem we have treated so far, there have been a distinction made between the x and y coordinate of a data point (x_i, y_i): the x coordinate is thought as the predictor variable, while the y coordinate is the predicted value. The distinction is significant. If, for example, we think of x as a function of y, the best line that fits the data using the method of least square is, in general, different than the corresponding line when thinking of y as a function of x. Indeed, if we use the same data as in §3, the best line $x = ny + c$ that fits it is given by

$x = 0.6677953348y - 0.738807374$. Solving in terms of y yields $y = 1.497464789x + 1.106338028$; this differs significantly from the line $y = 1.431451613x + 1.271370968$ found in §3.

In this case, we use as error the square of the *horizontal* distance, and it is somewhat perturbing that this similarly reasonable approach leads to a best fit that differs from the one obtained when employing vertical distances.

The situation can be made even worse if neither x nor y is a predictor variable. In this case, we want to minimize the shortest distance to a line $y = mx + b$ rather than the vertical (or horizontal) distance. 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.

In practice, the perturbing fact that different but reasonable error functions lead to best fits that are different is resolved by knowing what part of the data is the input and what part is predicted from it. This happens often enough, though not always. We thus can make a good choice for E and solve a minimization problem. That settled, we are still left with the problem of demonstrating why our choice for E is a good one.

It is rather easy to write down the solution to the problem in §3: if

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i\,, \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i\,,$$

then

$$\hat{m} = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{j=1}^{n} x_j\right)\left(\sum_{j=1}^{n} y_j\right)}{n\sum_{i=1}^{n} x_i x_i + \left(\sum_{j=1}^{n} x_j\right)^2} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}\,, \quad \hat{b} = \bar{y} - m\bar{x}\,.$$

If y_i are assumed to be the values of random variables Y_i which depend *linearly* upon the x_i,

$$Y_i = mx_i + b + \varepsilon_i\,,$$

with errors ε_i that are independent from each other, are zero on average and have some fixed variance, then the value of m given above is the *best* estimator of the slope among all those linear estimators that are unbiased. "Best" here is measured by calculating the deviation from the mean, and this best estimator is the one that produces the smallest such deviation.

This conclusion follows by merely making assumptions about the inner products of the data points $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$. Staticians often would like to answer questions such as the degree of accuracy of the estimated value of m and b. For that one would have to assume more about the probability distribution of the error variables Y_i. A typical situation is to assume that the ε_i above are normally distrubuted, with mean 0 and variance σ^2. Under these assumptions, the values of m and b given above are the so-called *maximum likelihood estimators* for these

two parameters, and there is yet one such estimator for the variance σ^2. But, since we assumed more, we can also say more. The estimators $\hat{m}$ and $\hat{b}$ are normally distributed and, for example, the mean of $\hat{m}$ is m and its variance is $\sigma^2 / \sum_{i=1}^{n}(x_i - \bar{x})^2$. With this knowledge, one may embark into determining the confidence we could have on our estimated value for the parameters. We do not do so in here, but want to plant the idea in the interested reader, whom we refer to books on the subject.

Chapter 3

The Art of Phugoid

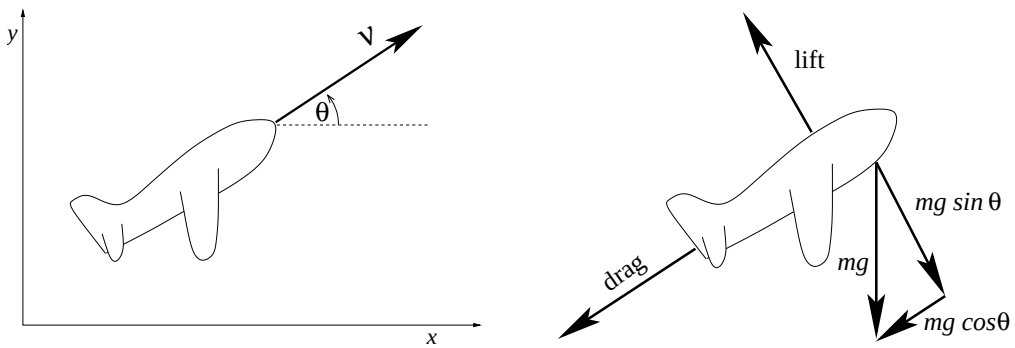
In this chapter, we will explore some aspects of mathematical model of glider flight. This model is called Lanchester's *phugoid theory*[1], developed by Frederick Lanchester [Lan] at beginning of the twentieth century. While this model has its drawbacks, it still is used today to explain oscillations and stalls in airplane flight.

Since this model, like many models, is a set of differential equations, we will need some results which are typically covered in a course on differential equations. We will cover the relevant material briefly here, but readers needing a more in-depth treatment are encouraged to look in a text on ordinary differential equations, such as [BDH], [HW], or [EP]. Of these, the approach in [BDH] is perhaps closest to the one presented here.

1 The Phugoid model

The Phugoid model is a system of two nonlinear differential equations in a frame of reference relative to the plane. Let $v(t)$ be the speed the plane is moving forward at time t, and $\theta(t)$ be the angle the nose makes with the horizontal. As is common, we will suppress the functional notation and just write v when we mean $v(t)$, but it is important to remember that v and θ are functions of time.

[1] Folklore has it that this name was chosen by Lanchester because he wanted a classically-based name for his new theory of oscillations occuring during flight. Since the Greek root *phug* ($\phi\nu\gamma\eta$, pronounced "fyoog") as well as the Latin root *fug-* both correspond to the English word "flight", he decided on the name "phugoid". Unfortunately, in both Greek and Latin, this means "flight" as in "run away" instead of what birds and airplanes do— the same root gives rise to the words "fugitive" and "centrifuge". The Latin for flight in appropriate sense is *volatus*; the Greek word is *potê* ($\pi o\tau\eta$).



If we apply Newton's second law of motion (force = mass × acceleration) and examine the major forces acting on the plane, we see easily the force acting in the forward direction of the plane is

$$m\frac{dv}{dt} = -mg\sin\theta - \text{drag}.$$

This matches with our intuition: When θ is negative, the nose is pointing down and the plane will accelerate due to gravity. When $\theta > 0$, the plane must fight against gravity.

In the normal direction, we have centripetal force, which is often expressed as mv^2/r, where r is the instantaneous radius of curvature. After noticing that that $\frac{d\theta}{dt} = v/r$, this can be expressed as $v\frac{d\theta}{dt}$, giving

$$mv\frac{d\theta}{dt} = -mg\cos\theta + \text{lift}.$$

Experiments show that both drag and lift are proportional to v^2, and we can choose our units to absorb most of the constants. Thus, the equations simplify to the system

$$\frac{dv}{dt} = -\sin\theta - Rv^2 \qquad \frac{d\theta}{dt} = \frac{v^2 - \cos\theta}{v}$$

which is what we will use henceforth. Note that we must always have $v > 0$.

It is also common to use the notation $\dot{v}$ for $\frac{dv}{dt}$ and $\dot{\theta}$ for $\frac{d\theta}{dt}$. We will use these notations interchangeably.

2 What do solutions look like?

Now that we have some equations in hand, our first goal is to understand what the solutions look like. Note that the equations are nonlinear, and we cannot hope to find explicit formulae for the solutions— they don't exist except in very special cases.

Let's deal with the easiest case first: pretend that drag doesn't exist and set $R = 0$. Does this help?

The equations become

$$\frac{dv}{dt} = -\sin\theta \qquad \frac{d\theta}{dt} = \frac{v^2 - \cos\theta}{v}$$

Now, let's see what we can say about the solutions to this equation. First, notice that there must be many solutions: Any choice of initial conditions $\theta(0) = \theta_0, v(0) = v_0$ gives rise to a different solution. (Informally, if the glider starts with a different initial angle and/or velocity, it will fly along a different path. If we say it this way, it seems so obvious we shouldn't need to say it at all.) Can we find *any* solutions at all?

There is one very easy solution to find. Notice that if we can find a choice of θ and v so that $\frac{dv}{dt} = 0 = \frac{d\theta}{dt}$, then since their derivatives are zero, $v(t)$ and $\theta(t)$ must be constant functions. But solving this is simple:

$$0 = -\sin\theta \qquad\qquad 0 = \frac{v^2 - \cos\theta}{v}$$

is easily seen to be true when $\theta = 0$ and $v = 1$. In fact, θ can be any multiple of 2π. So we have found the special solution

$$v(t) \equiv 1 \qquad\qquad \theta(t) \equiv 0.$$

This is a glider which always flies level, with a constant velocity of 1.

What about other solutions? Now we must work harder. There is no hope to find a general analytic expression for $v(t)$ and $\theta(t)$, but that does not mean all is lost. We can get an idea of how other solutions behave by thinking about what it means to have a solution $\varphi(t) = [\theta(t), v(t)]$.

First, notice that such a solution φ can be thought of as a parametric curve in the (θ, v)-plane which passes through the point $\varphi(0) = [\theta_0, v_0]$. At any point along this parametric curve, we can calculate its derivative:

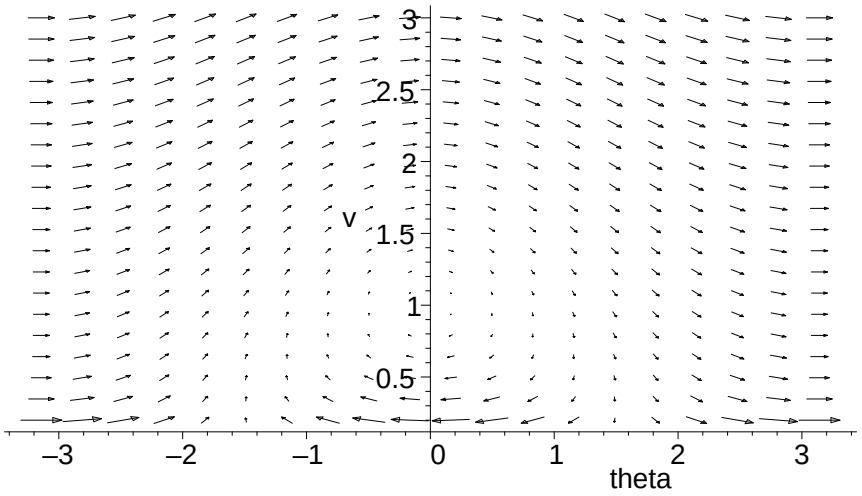
$$\varphi'(t) = \left[\frac{d}{dt}\theta(t), \frac{d}{dt}v(t)\right] = \left[\dot\theta(t), \dot v(t)\right].$$

But we started out with expressions for $\dot\theta$ and $\dot v$: our original differential equation. Because our differential equations only involve t implicitly via θ and v (the system is *autonomous*), we need only know the values of θ and v to know the tangent vector to any solution curve passing through that point.

We can get an idea of how the solutions behave by making a picture of the vector field corresponding to this: At each value for the pair (θ, v), we draw the vector $\langle\dot\theta, \dot v\rangle$, which only depends on (θ, v).

To get maple to do this, we can use the `fieldplot` command. Because the arrows in the `fieldplot` show not only direction but also relative magnitude, we shouldn't get too close to the $v = 0$ axis or they will overwhelm the smaller arrows for which occur for larger v.

```
> plots[fieldplot]( [ (v^2 - cos(theta))/v, -sin(theta) ],
                    theta=-Pi..Pi, v=0.2..3, arrows=slim);
```



We can certainly see that near the constant solution $\{\theta = 0, v = 1\}$, nearby solutions rotate around it in a clockwise manner. This "rotation" in the $\theta - v$ coordinates corresponds to a flight path which wobbles: the nose angle alternately increases and decreases, as does the velocity. For v large, it seems that the solutions move from left to right— θ constantly increases, which means that the glider is always increasing its nose angle. This means it is doing loops.

We can also use the `DEplot` command from Maple's `DEtools` package to produce a direction field. The direction field is obtained from the vector field by rescaling all nonzero vectors to be of unit length. Sometimes discarding the extra information makes it easier to see what is happening. In addition, `DEplot` has other very useful options, and will be one of our primary tools throughout this chapter.

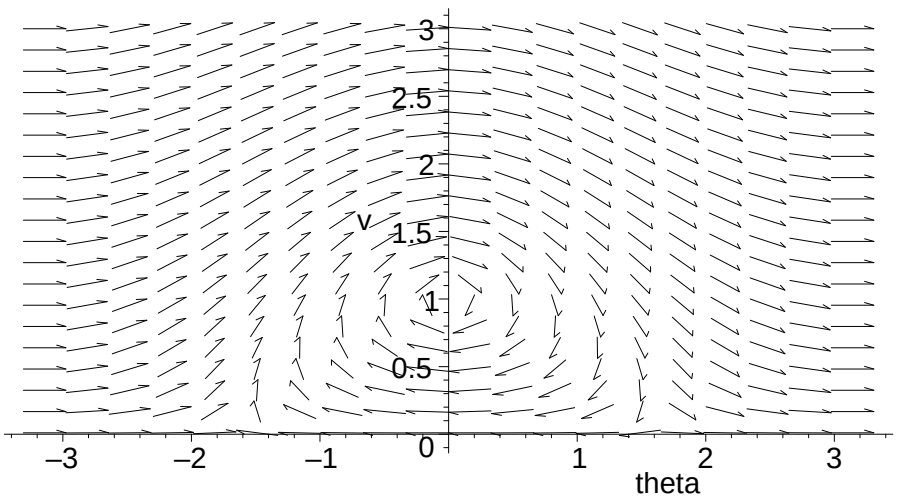
For convenience, we will name our system of differential equations as `phug`, so that we needn't keep retyping them.

```
> phug := [ diff(theta(t),t) =  (v(t)^2 - cos(theta(t)))/v(t),
            diff(v(t),t)     =  -sin(theta(t)) - R*v(t)^2 ];
  R:=0;
```

$$phug := [\frac{\partial}{\partial t}\,\theta(t) = \frac{\mathrm{v}(t)^2 - \cos(\theta(t))}{\mathrm{v}(t)},\ \frac{\partial}{\partial t}\,\mathrm{v}(t) = -\sin(\theta(t))]$$

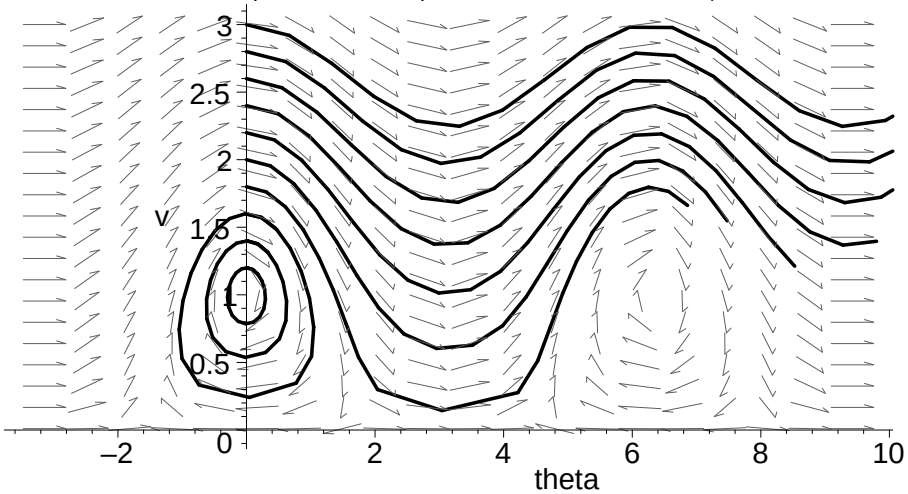
$$R := 0$$

```
> with(DEtools):
  DEplot( phug, [theta(t), v(t)], t=0..1, theta=-Pi..Pi, v=0.01..3);
```



We can also ask maple to plot a number of solutions corresponding to several initial conditions. Rather than typing in 11 initial conditions, we can use `seq` to generate them. We tell maple that the solution curves should be black, because otherwise it uses a sickly yellow. If we had wanted each solution curve to be a different color, we could use something like `linecolor=[seq(COLOR(HUE,i/10),i=0..10)]`.

```
> DEplot( phug, [theta(t), v(t)], t=0..5,
          [seq([theta(0)=0, v(0)=1+0.2*i], i=0..10)],
          theta=-Pi..3*Pi, v=0.01..3, linecolor=black );
```

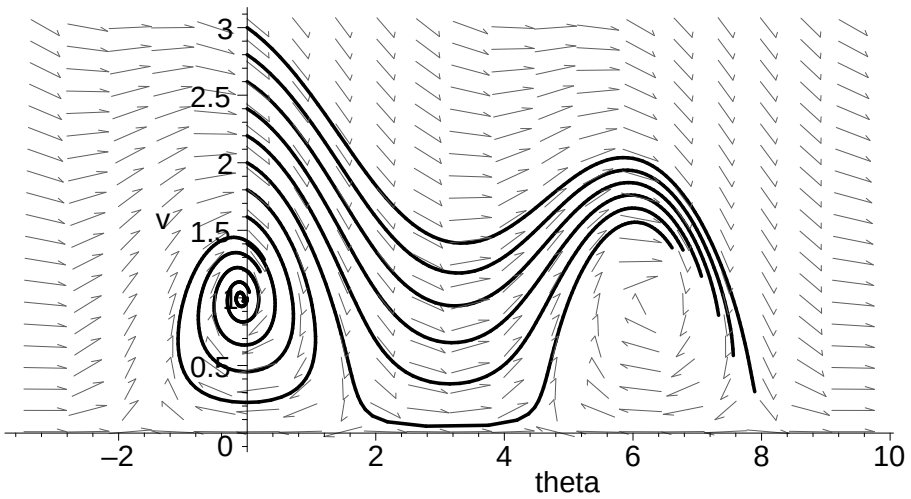


The above figure confirms our earlier impressions about the "wobbling" and "looping" solutions. We'll see how to get Maple to demonstrate this more explicitly in section 5, where we will plot the path of the glider through space (i.e., in x-y coordinates).

Just to see how things differ if we make R nonzero, let's change R and get Maple to show us what happens. We need to adjust the stepsize to get a reasonable accuracy; what precisely this does is described in the next section.

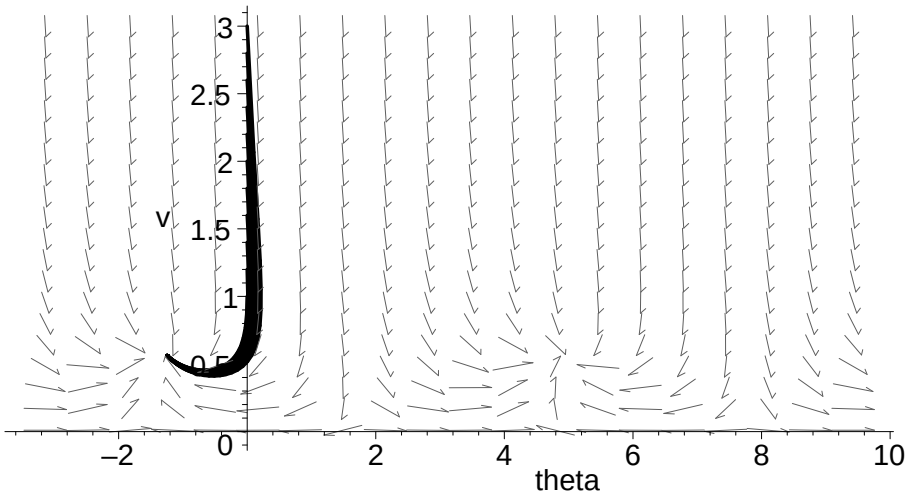
```
> R:=0.1;
  DEplot( phug, [theta(t), v(t)], t=0..5,
          [seq([theta(0)=0, v(0)=1+0.2*i], i=0..10)],
          theta=-Pi..3*Pi, v=0.01..3, linecolor=black, stepsize=0.1 );
```

$$R := .1$$




```
> R:=3;
  DEplot( phug, [theta(t), v(t)], t=0..5,
          [seq([theta(0)=0, v(0)=1+0.2*i], i=0..10)],
          theta=-Pi..3*Pi, v=0.01..3, linecolor=black,stepsize=0.1 );
```

$$R := 3;$$



The behavior of the glider appears to be dramatically affected by the drag parameter R. One of our main goals in this chapter will be to classify the types of solutions that are possible as a function of the drag parameter R.

3 Existence of Solutions

We've been acting as though just by specifying an initial condition, there must be a solution, and it must be unique (that is, the only one corresponding to that initial condition). And, in fact, this is typically true for any "nice" differential equation. But which differential equations are "nice" enough?

We won't prove the theorem here, but we will state it. The reader is encouraged to look up the proof in any differential equations text, such as [BDH] or [HW].

Theorem 3.1 *Consider the differential equation*

$$\frac{d\vec{x}}{dt} = \vec{F}(\vec{x}, t).$$

If $\vec{F}$ is continuous on an open set in containing $(\vec{x}_0, t_0)$, then there is a solution $\vec{x}(t)$ which is defined on some interval $(t_0 - \epsilon, t_0 + \epsilon)$ which satisfies the initial value problem

$$\frac{d\vec{x}}{dt} = \vec{F}(\vec{x}, t) \qquad \vec{x}(t_0) = \vec{x}_0.$$

Furthermore, if $\vec{F}$ has continuous partial derivatives, this solution is unique.

In fact, the hypothesis can be weakened a little bit and still preserve the uniqueness. As long as $\vec{F}$ is at least Lipshitz in $\vec{x}$, the solution will be unique. A function $\vec{F}$ if called Lipschitz if there is some K so that

$$|\vec{F}(\vec{x}_1, t) - \vec{F}(\vec{x}_2, t)| < K|\vec{x}_1 - \vec{x}_2|$$

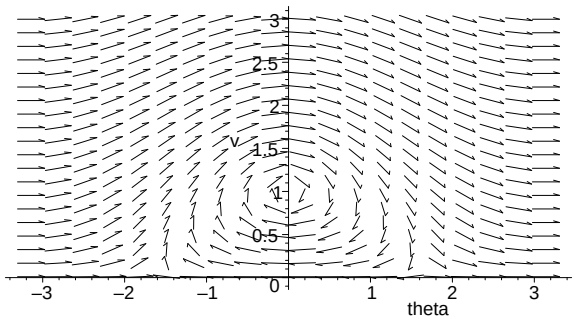
for all t, $\vec{x}_1$, and $\vec{x}_2$. This condition says, roughly, that $\vec{F}$ doesn't spread out too quickly. All functions with continuous partials are automatically Lipshitz.

This means that for us, as long as we stay away from the place where our differential equation isn't defined (i.e. ensure that $v > 0$), we can be assured that there is a solution through every point, and that solution is unique. Notice that since our equations are autonomous (the right-hand side has no explict dependence on t), uniqueness of solutions means that solutions cannot cross in the (θ, v)-plane.

4 Numerical Methods

What does Maple do when you ask it to display a solution of a differential equation with `DEplot`? Obviously, it cannot solve the equations analytically, since solutions don't always exist in closed form. Instead, it approximates them numerically.

4.1 Euler's method



To get an idea of how this can be done, take a look again at the direction field for the glider. By looking at how the arrows point, you can almost guess how the solution must turn. From a given initial condition, move in the direction of the arrow to the next position. At that new point, step in the direction of the vector at that point, and so on. This is the idea behind the simplest numerical integration scheme, called Euler's method.

More precisely, given a differential equation and an initial condition

$$\frac{d\vec{x}}{dt} = \vec{F}(\vec{x}, t) \qquad\qquad \varphi(t_0) = \vec{x}_0,$$

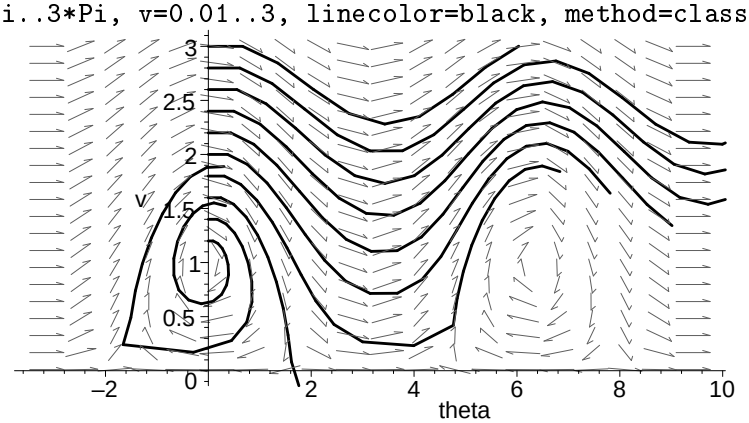
the goal is to construct a sequence of points $\vec{x}_0, \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$ which approximate the solution φ at times $t_0, t_1, t_2, \ldots, t_n$. We take our initial point to be the initial condition, and choose some small number $h > 0$ called the *stepsize*. Then we set

$$\vec{x}_{i+1} = \vec{x}_i + h\vec{F}(\vec{x}_i, t_i) \qquad\qquad t_{i+1} = t_i + h$$

for $0 \le i \le n-1$. This very simple scheme works best for small values of h; the error in the approximation for a fixed time interval is proportional to h. This means that to improve the accuracy of a numerical solution by a factor of 10, we need to do about 10 times more work.

Maple doesn't typically use Euler's method because the errors accumulate too quickly. To see this, we reproduce one of the earlier figures from §2 using Euler's method.

```
> DEplot( phug, [theta(t), v(t)], t=0..5,
      [seq([theta(0)=0, v(0)=1+0.2*i], i=0..10)],
      theta=-Pi..3*Pi, v=0.01..3, linecolor=black, method=classical[foreuler]);
```



While the errors in the above figures are considerable, using a much smaller stepsize (say, `stepsize=0.01`) will essentially reproduce the earlier figure, with a lot more computation.

4.2 Numerical Solutions, Numerical Integration, and Runge-Kutta

To get some ideas about improving on Euler's method, let's first notice that Euler's method corresponds exactly to the left-hand Riemann sum when computing a numerical approximation to an integral. You may recall from calculus that the left-hand sum converges to the value of the integral, but you must take a large number of subdivisions to get any accuracy.

A more efficient method is the trapezoid rule, which is the average of the left-hand and right-hand sum. There is a numerical scheme for integrating ODEs which corresponds to this, known variously as the *improved Euler method* or the *Heun formula*. In the trapezoid rule, we compute the function at the left and right endpoints of the function and average the result; the analog of this would require us to average the value of the vector field at our current point and at the next point. Perhaps you see the circularity implicit in that reasoning: to know the next point, we would need to solve the differential equation. So, instead we fudge a bit: we get our approximation of the next point by using Euler's method. More specifically, using the notation of §4.1, given an approximation $\vec{x}_i$ at time t_i, we find the next one using the scheme below.

$$\begin{aligned} t_{i+1} &= t_i + h \\ \vec{k}_\ell &= \vec{F}(\vec{x}_i, t_i) && \text{slope at the left side of the interval} \\ \vec{k}_r &= \vec{F}(\vec{x}_i + h\vec{k}_\ell, t_{i+1}) && \text{slope at the right side of the interval} \\ \vec{x}_{i+1} &= x_i + h(\vec{k}_\ell + \vec{k}_r)/2 \end{aligned}$$

To do the improved Euler method, we need to evaluate the vector field twice as often as the regular Euler method, but there is a big gain in accuracy. The error of the approximation for a fixed time interval is proportional to h^2. Thus, if we decrease h by a factor of 10, we should expect to reduce the error 100-fold. You can get Maple to use the improved Euler method by specifying `method=classical[heunform]` as an argument to `DEplot` and related commands.

You might remember from your calculus course that Simpson's rule can get very accurate numerical approximations for integrals with a very few number of points. This is because Simpson's rule is a fourth-order method: the error in the approximation is proportional to h^4. For Simpson's rule, we evaluate the function at the right endpoint, the left endpoint, and the middle, and then take a weighted average of those values.

The analog of Simpson's rule for differential equations is called the Runge-Kutta method[2], developed by the German mathematicians C. Runge and W. Kutta at the end of the nineteenth

[2]In fact, there is a whole family of such methods, of various orders. But generally, if someone says "Runge-Kutta" without specifying the order, they mean this fourth-order Runge-Kutta.

century. It takes a linear combination of four slopes, as below:

$$
\begin{array}{rcll}
t_{i+1} & = & t_i + h & \\
\vec{k}_\ell & = & \vec{F}(\vec{x}_i, t_i) & \text{slope at the left side of the interval} \\
\vec{k}_m & = & \vec{F}(\vec{x}_i + \frac{h}{2}\vec{k}_\ell, t_i + \frac{h}{2}) & \text{slope at the middle of the interval} \\
\vec{k}_c & = & \vec{F}(\vec{x}_i + \frac{h}{2}\vec{k}_m, t_i + \frac{h}{2}) & \text{corrected slope at the middle of the interval} \\
\vec{k}_r & = & \vec{F}(\vec{x}_i + h\vec{k}_c, t_{i+1}) & \text{slope at the right side of the interval} \\
\vec{x}_{i+1} & = & x_i + h(\vec{k}_\ell + 2\vec{k}_m + 2\vec{k}_c + \vec{k}_r)/6 &
\end{array}
$$

When the slope depends only on t and not on $\vec{x}$, then $\vec{k}_m$ and $\vec{k}_c$ are the same, and the formula reduces exactly to that of Simpson's rule.

Runge-Kutta is probably the most commonly used method of numerical integration, because of its generally high accuracy. Like Simpson's rule, it is a fourth-order method, so the error is proportional to h^4. Maple has several numerical methods for ODEs built in to it; see the help page on `dsolve[numeric]` for more information about them; the ones we have described are "classical" methods, and are described (along with others) on Maple's help page for `dsolve[classical]`. Unless asked to do otherwise, Maple's `DEplot` command uses the fourth-order Runge-Kutta method described above (Maple's name for this method is `classical[rk4]`), although the `numeric` option of `dsolve` defaults to a variation called *fourth-fifth order Runge-Kutta-Fehlberg*, which Maple refers to as `rkf45`.

5 Seeing the flight path

In §2, we exhibited a number of solutions to the phugoid model with various initial conditions and values for the drag, as plots of the parametric curves $[\theta(t), v(t)]$. From these, we could deduce an approximate path of the glider through space via reasoning like "$\theta(t)$ is monotonically increasing, so the glider must be doing loop-the-loops". We will now discuss how to solve for the position $[x(t), y(t)]$ directly.

First, notice that if we know the value of v and θ at some time t, we also have $\frac{dx}{dt}$ and $\frac{dy}{dt}$ at that t, since

$$\frac{dx}{dt} = v\cos\theta \qquad \text{and} \qquad \frac{dy}{dt} = v\sin\theta.$$

So, one way to find $x(t)$ and $y(t)$ would be to solve the original ODE (numerically) for $\theta(t)$ and $v(t)$, obtain a new differential equation, and then solve that numerically for $x(t)$ and $y(t)$.

Aside from the fact that there is an easier way, there is a serious practical problem with this approach. When we get a numerical solution $[\theta(t), v(t)]$, this is a collection of points $(\theta_0, v_0, t_0), (\theta_1, v_1, t_1), \ldots, (\theta_n, v_n, t_n)$ along a curve. This means we only know the vector field for (x, y) at those points. It is highly unlikely that these will be the points we need to use Runge-Kutta. We could use these to approximate our (x, y)-solution using Euler's method,

although even then we would have to program it ourselves rather than use Maple's built-in method.

If you think for just a moment, you should realize that there is no reason we must find $\theta(t)$ and $v(t)$ before attempting to find $x(t)$ and $y(t)$. Instead, we can augment the original system, adding in the new equations for $\dot{x}$ and $\dot{y}$. Then, we can use numerical methods to solve for $[\theta(t), v(t), x(t), y(t)]$ all in one step. The methods we discussed in §4.2 apply equally well to vector fields in any number of variables.

Consequently, we rewrite our phugoid system as

$$\frac{d\theta}{dt} = \frac{v^2 - \cos\theta}{v} \qquad \frac{dv}{dt} = -\sin\theta - Rv^2 \qquad \frac{dx}{dt} = v\cos\theta \qquad \frac{dy}{dt} = v\sin\theta.$$

Using this system, we need to specify our initial conditions as $(\theta_0, v_0, x_0, y_0)$, and each solution is a curve in $\mathbb{R}^4$ parameterized by time. We can take a projection of this curve to get our solutions in θ-v or x-y coordinates.

Asking Maple to solve this new system is almost the same as before, but we must specify the additional variables and use the `scene` option to `DEplot` to tell Maple which projection we want to see. Note that we reset R to ensure that it appears in the equations as an arbitrary constant, rather than with any previously assigned value.

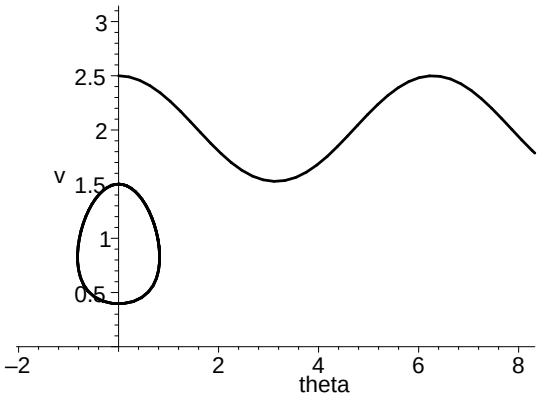
```
> R:='R':
  xphug :=[ diff(theta(t),t) =  (v(t)^2 - cos(theta(t)))/v(t),
            diff(v(t),t)     =  -sin(theta(t)) - R*v(t)^2 ,
            diff(x(t),t)     =  v(t)*cos(theta(t)),
            diff(y(t),t)     =  v(t)*sin(theta(t))];
```

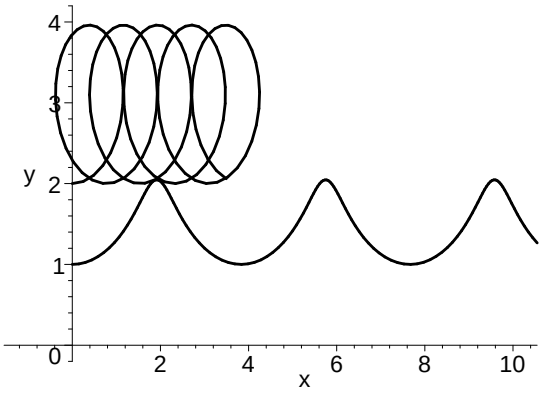
$$xphug := [\tfrac{\partial}{\partial t}\,\theta(t) = \frac{\mathrm{v}(t)^2 - \cos(\theta(t))}{\mathrm{v}(t)},\ \tfrac{\partial}{\partial t}\,\mathrm{v}(t) = -\sin(\theta(t)) - R\,\mathrm{v}(t)^2,$$
$$\tfrac{\partial}{\partial t}\,\mathrm{x}(t) = \mathrm{v}(t)\cos(\theta(t)),\ \tfrac{\partial}{\partial t}\,\mathrm{y}(t) = \mathrm{v}(t)\sin(\theta(t))]$$

Now we ask Maple to plot two solution curves in both θ, v coordinates and x, y. One is started with an initial angle of $\theta = 0$ and velocity $v = 1.5$ at a height $y(0) = 1$; the other starts with $\theta = 0$, $v = 2.5$, and initial height $y = 2$. We also use `display` with an `array` to show the two scenes side by side.

```
> R:=0;
  plots[display](array([
   DEplot(xphug, [theta,v,x,y], t=0..15,
    [[theta(0)=0,v(0)=1.5,x(0)=0,y(0)=1],[theta(0)=0,v(0)=2.5,x(0)=0,y(0)=2]],
    theta=-Pi..5*Pi/2, v=0.1..3, x=-1..10, y=0..4,
    linecolor=black, stepsize=0.1, scene=[theta,v]),

   DEplot(xphug, [theta,v,x,y], t=0..15,
    [[theta(0)=0,v(0)=1.5,x(0)=0,y(0)=1],[theta(0)=0,v(0)=2.5,x(0)=0,y(0)=2]],
    theta=-Pi..5*Pi/2, v=0.1..3, x=-1..10, y=0..4,
    linecolor=black, stepsize=0.1, scene=[x,y]),
   ])));
```

$R := 0;$

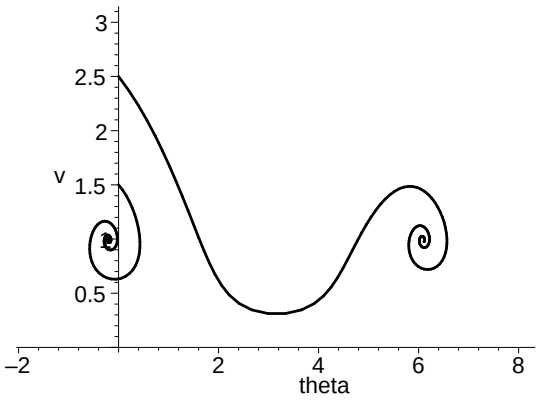


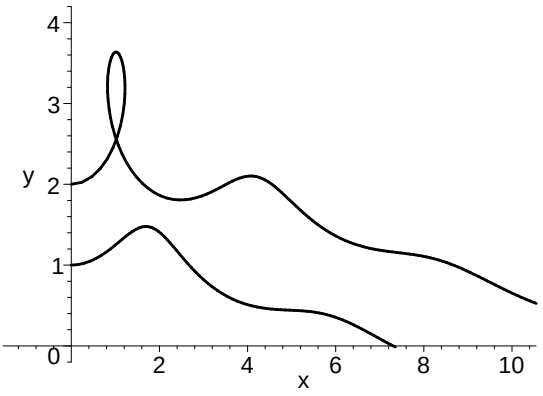


If we change the value of R to 0.2 and repeat the command, we can see how things change when there is friction. Notice how spiraling towards the fixed point in the θ-v coordinates corresponds to a plane which is diving with an oscillatory motion.

```
> R:=0.2;
  plots[display](array([
   DEplot(xphug, [theta,v,x,y], t=0..15,
    [[theta(0)=0,v(0)=1.5,x(0)=0,y(0)=1],[theta(0)=0,v(0)=2.5,x(0)=0,y(0)=2]],
    theta=-Pi..5*Pi/2, v=0.1..3, x=-1..10, y=0..4,
    linecolor=black, stepsize=0.1, scene=[theta,v]),

   DEplot(xphug, [theta,v,x,y], t=0..15,
    [[theta(0)=0,v(0)=1.5,x(0)=0,y(0)=1],[theta(0)=0,v(0)=2.5,x(0)=0,y(0)=2]],
    theta=-Pi..5*Pi/2, v=0.1..3, x=-1..10, y=0..4,
    linecolor=black, stepsize=0.1, scene=[x,y]),
   ]));
```

$R := 0.2;$

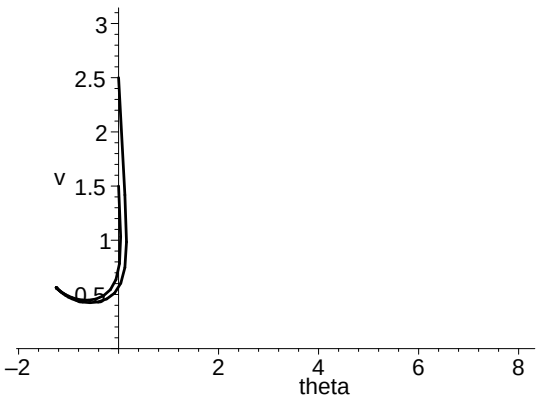


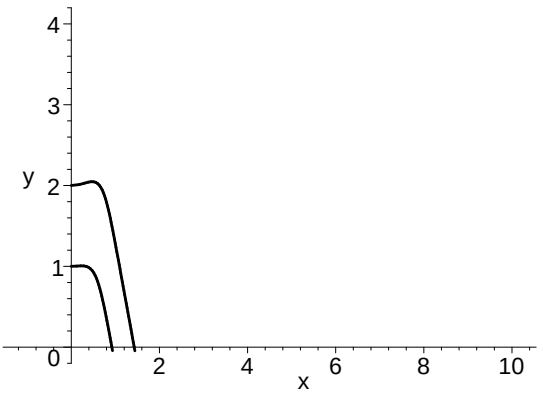


Finally, we redo the pictures with a large amount of friction ($R = 3$). Note how the glider acts more like a rock than a glider, and there is no oscillation at all.

```
> R:=3;
  plots[display](array([
   DEplot(xphug, [theta,v,x,y], t=0..15,
    [[theta(0)=0,v(0)=1.5,x(0)=0,y(0)=1],[theta(0)=0,v(0)=2.5,x(0)=0,y(0)=2]],
    theta=-Pi..5*Pi/2, v=0.1..3, x=-1..10, y=0..4,
    linecolor=black, stepsize=0.1, scene=[theta,v]),

   DEplot(xphug, [theta,v,x,y], t=0..15,
    [[theta(0)=0,v(0)=1.5,x(0)=0,y(0)=1],[theta(0)=0,v(0)=2.5,x(0)=0,y(0)=2]],
    theta=-Pi..5*Pi/2, v=0.1..3, x=-1..10, y=0..4,
    linecolor=black, stepsize=0.1, scene=[x,y]),
   ])));
```

$$R := 3;$$





6 Fixed Point Analysis

So far, the cases we've looked at seem to have a number of common features. Each has has a solution with $\theta(t)$ and $v(t)$ constant; there are solutions which immediately tend toward the constant solution (or oscillate around it), and solutions which do some number of loops first (for $R = 3$, we didn't demonstrate this– if the inital angle is 0, a solution needs an initial velocity larger than 86.3 before it will do a loop). How do the possible solutions depend on the value of R? For example, the behavior of solutions for $R = 0.1$ and $R = 0.2$ are quite similar, but are dramatically different from $R = 0$ and $R = 3$.

One way we can get a better handle on exactly how the behavior of the solutions depends on R is to examine what happens to solutions near the constant solution. If we look in the θ-v plane, this solution corresponds to a single point, which is often referred to as a "fixed point". We explicitly found the fixed point $\{\theta(t) = 0, v(t) = 1\}$ solution for $R = 0$ in §2 by noticing

that whenever $\dot{\theta} = 0$ and $\dot{v} = 0$, we must have a constant solution (and vice-versa). Using `solve`, we can get maple to tell us the where the fixed point is for arbitrary values of R. We use `convert` to insist that maple represent the result as a radical, instead of using the `RootOf` notation.

```
> R:='R':
  FixPoint:=convert(
            solve({(v^2-cos(theta))/v=0,-sin(theta)-R*v^2=0}, {theta,v}),
            radical);
```

$$FixPoint := \{v = (\frac{1}{R^2+1})^{(1/4)},\ \theta = \arctan(-R\sqrt{\frac{1}{R^2+1}},\ \sqrt{\frac{1}{R^2+1}})\}$$

So, we see that there is a fixed point for all values of R (since $1 + R^2$ is always positive, the radical always takes real values). For $R > 0$, $\theta(t)$ at this fixed solution is negative, so this corresponds to a diving solution. As R increases, the angle of the dive becomes steeper and steeper.

What can we say about behaviour of solutions *near* this fixed solution?

In order to answer this, notice that we can apply Taylor's theorem to the right-hand side of our system of differential equations. That is, think of our differential equation as being in the vector form

$$\frac{d\vec{X}}{dt} = \vec{F}(\vec{X}),$$

where (in our case) $\vec{X}$ is $[\theta(t), v(t)]$, and $\vec{F}(X) = [(v^2 - \cos(\theta))/v,\ -\sin(\theta) - Rv^2]$. Taylor's theorem says that for $\vec{X}$ near $\vec{X}_0$,

$$\vec{F}(\vec{X}) = \vec{F}(\vec{X}_0) + D\vec{F}(\vec{X}_0)(\vec{X} - \vec{X}_0) + \text{higher order terms},$$

where $D\vec{F}(\vec{X}_0)$ is the Jacobian of $\vec{F}$ evaluated at $\vec{X}_0$. Since $\vec{X}_0$ is a fixed point, $\vec{F}(\vec{X}_0)$ is the zero vector. Hence, we can get a good idea of what is happening close to the fixed point by studying the *linear* system of differential equations given by the derivative at the fixed point.

First, we give a brief refresher about linear 2×2 systems of equations. The reader is refered to the texts in the references for more comprehensive treatment of this topic.

6.1 Linear Systems of ODEs

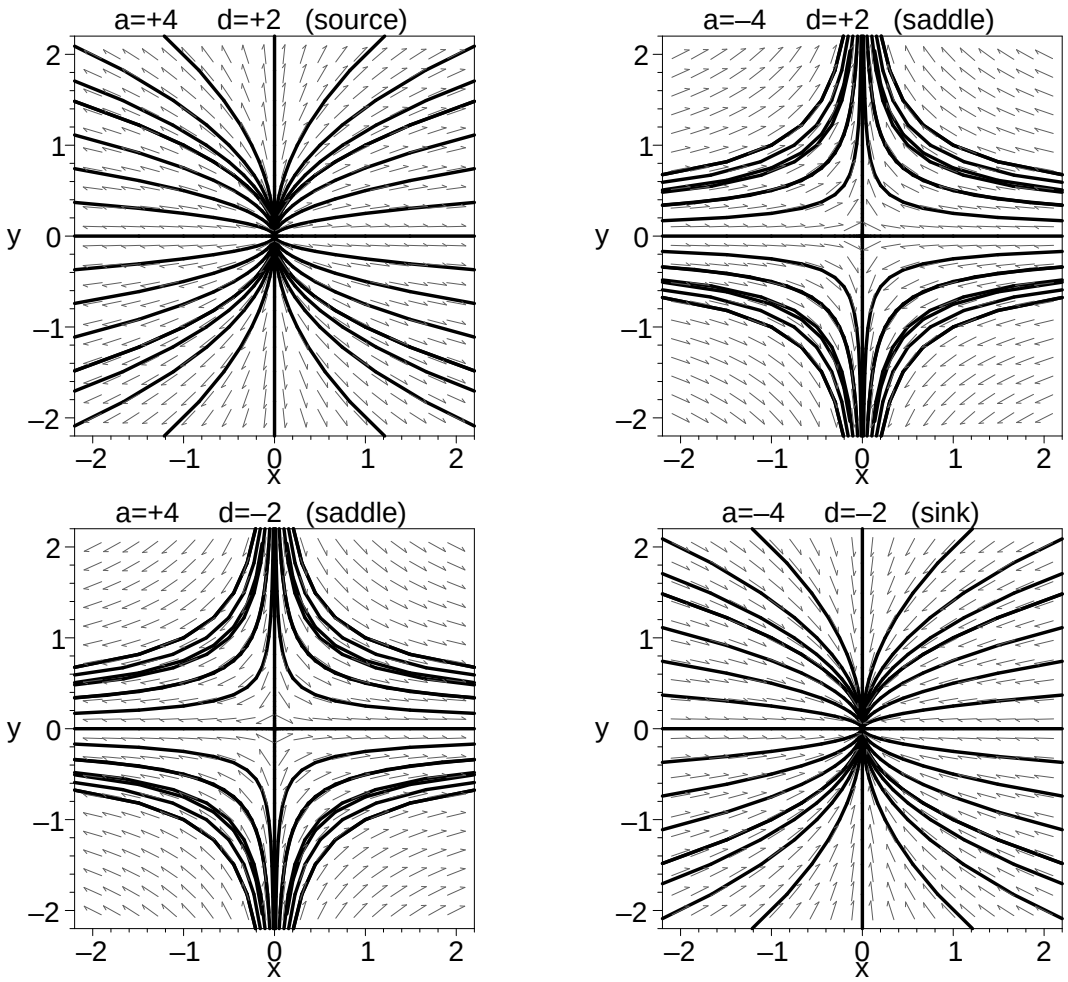
Linear differential equations (with constant coefficients) are the simplest ones to solve and understand. Such equations have the form $\frac{dX}{dt} = AX$, where A is an $n \times n$ matrix and X is an n-vector. We'll be content with $n = 2$.

First, notice that $X = 0$ is always a fixed point for this system, and it is the only one. Secondly, if $X_1(t)$ and $X_2(t)$ are solutions to the system, then so is $X_1(t) + X_2(t)$. This property is called *superposition* and will be very useful.

Let's look at the simplest case for the 2×2 system: $A = \begin{pmatrix} a & 0 \\ 0 & d \end{pmatrix}$, that is,

$$\begin{aligned} \dot{x} &= ax \\ \dot{y} &= dy \end{aligned}$$

If we have an initial condition with $y(0) = 0$, then $y(t) = 0$. This means that the problem reduces to the one-dimensional case, and $x(t) = x(0)e^{at}$. If $a > 0$, the solution moves away from the origin along the x-axis as t increases; if $a < 0$, the solution moves toward the origin. If the initial condition is on the y-axis, the sign of d controls whether the solution moves away ($d > 0$) or towards ($d < 0$) the origin. Because of the superposition property mentioned before, an initial condition with $x(0) \neq 0$ and $y(0) \neq 0$ gives rise to a solution which is a combination of these two behaviours.



A large number of linear systems behave very similarly to the cases above, except the straight-line solutions may not lie exactly along the coordinate axes.

Suppose there is a solution of $\dot{X} = AX$ which lies along a straight line, that is, along some vector $\vec{v}$. Because the system is linear, this means the tangent vectors to this solution must be of the form λX for some number λ. Such a number λ is called an *eigenvalue*, and the corresponding vector $\vec{v}$ is the corresponding *eigenvector*. Note that in the case above, we had eigenvalues a and d with eigenvectors $[1, 0]$ and $[0, 1]$.

To find the eigenvalues, we need to solve

$$AX = \lambda X, \quad \text{or equivalently,} \quad (A - \lambda I)X = 0.$$

This can only happen if X is the zero vector, or the determinant of $A - \lambda I$ is zero. In the latter case, we must have

$$(a - \lambda)(d - \lambda) - bc = 0,$$

That is,

$$\lambda^2 - (a + d)\lambda + (ad - bc) = 0.$$

Using the quadratic formula, we see that the eigenvalues must be

$$\lambda = \frac{(a + d) \pm \sqrt{(a + d)^2 - 4(ad - bc)}}{2}.$$

The quantity $a + d$ is called the *trace* of A (more generally, the trace of a matrix is the sum of diagonal entries), and $ad - bc$ is the determinant of A. The eigenvalues of a 2×2 matrix can be expressed in terms of the trace and the determinant[3] as

$$\frac{TrA \pm \sqrt{(TrA)^2 - 4\,detA}}{2}.$$

We'll use this form again in a little while.

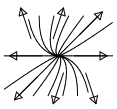
For linear systems, the eigenvalues determine the ultimate fate of solutions to the ODE. From the above, it should be clear that there are always two eigenvalues for a 2×2 matrix (although sometimes there might be a double eigenvalue, when $(TrA)^2 - 4\,detA = 0$). Let's call them λ_1 and λ_2.

We've already seen prototypical examples where the eigenvalues are real, nonzero, and distinct. But what if the eigenvalues are complex conjugate (which happens when $(TrA)^2 < 4\,detA$)? In this case, there is no straight line solution in the reals. Instead, solutions turn around the origin. If the real part of the eigenvalues is positive, solutions spiral away from the origin, and if it is negative, they spiral towards it. If the eigenvalues are purely imaginary, the solutions neither move in nor out; rather, they circle around the origin.

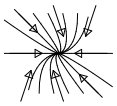
We can summarize this in the table below. We are skipping over dealing with the degenerate cases, when $\lambda_1 = \lambda_2$ and when one of the eigenvalues is zero.

[3] It is true for all matrices that the trace is the sum of the eigenvalues, and the determinant is their product.

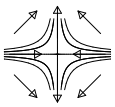
$\lambda_1 > \lambda_2 > 0$
source



$\lambda_1 < \lambda_2 < 0$
sink



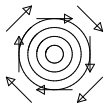
$\lambda_1 < 0 < \lambda_2$
saddle



λ_i complex, $Re\lambda > 0$
spiral source

λ_i complex, $Re\lambda < 0$
spiral sink

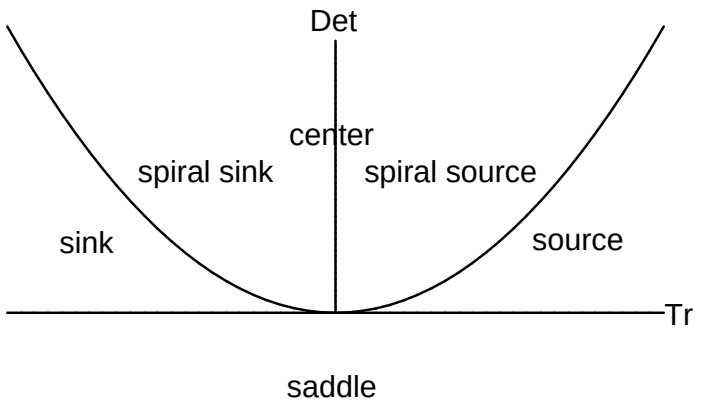
λ_i complex, $Re\lambda = 0$
center



Finally, we remark that there is a convenient way to organize this information into a single diagram. Since the eigenvalues of A can be written as

$$\frac{TrA \pm \sqrt{(TrA)^2 - 4\,detA}}{2},$$

we can consider the matrix A as being a point in the Trace-Determinant plane. Then the curve where $(TrA)^2 = 4\,detA$ is a parabola. If A is above this parabola, it has complex eigenvalues. Furthermore, if the determinant is negative, we must have a saddle because[4] $\sqrt{(TrA)^2 - 4\,detA} > TrA$, which means there is one positive and one negative eigenvalue. Finally, if the determinant is positive, the eigenvalues (or their real part, if they are complex) is the same sign as the trace of A. We summarize this in the diagram below.



[4]Since the determinant is the product of the eigenvalues, the only way it can be negative is if they are of opposite signs.

6.2 Fixed Points for the Glider

We now return to our nonlinear system, and look at the linearization near the fixed point. As we saw earlier in this section, for every value of R the Phugoid model

$$\frac{d\theta}{dt} = \frac{v^2 - \cos\theta}{v} \qquad \frac{dv}{dt} = -\sin\theta - Rv^2$$

has a fixed point at

$$\theta = -\arctan\left(R\sqrt{\frac{1}{1+R^2}}\right) \qquad v = \sqrt[4]{\frac{1}{1+R^2}}.$$

We now calculate the Jacobian matrix, using Maple.

```
> with(linalg):
  R:='R':
  J:=jacobian( [(v^2-cos(theta))/v, -sin(theta)-R*v^2], [theta, v]);
```

$$J := \begin{bmatrix} \dfrac{\sin(\theta)}{v} & 2 - \dfrac{v^2 - \cos(\theta)}{v^2} \\ -\cos(\theta) & -2\,R\,v \end{bmatrix}$$

Once we know the trace and determinant of the Jacobian matrix at the fixed point, we can use that information to determine how its type (sink, source, saddle, etc.) depends on R. Rather than solve for the eigenvalues directly, we will compute the trace and determinant, which have a simpler form in this case.

First, let's find the fixed point again, issuing the same command we did earlier. Then, we'll substitute that into the result of `trace` and `det` to calculate the trace and determinant at the fixed point.

```
> FixPoint:=convert(
            solve({(v^2-cos(theta))/v=0,-sin(theta)-R*v^2=0}, {theta,v}),
            radical);
```

$$FixPoint := \{\theta = \arctan(-R\sqrt{\frac{1}{R^2+1}}, \sqrt{\frac{1}{R^2+1}}),\ v = (\frac{1}{R^2+1})^{(1/4)}\}$$

```
> trfix := simplify(subs(FixPoint, trace(J)));
  detfix:= simplify(subs(FixPoint, det(J)));
```

$$trfix := \frac{-3R}{(R^2+1)^{1/4}}$$

$$detfix := 2\sqrt{R^2+1}$$

So, we see that for all real values of R, the determinant is positive (meaning saddles are impossible). Furthermore, for $R > 0$, the trace is negative, so the fixed point is always a sink. (When $R = 0$, the trace is zero, which means it is possible for this fixed point to be a center.[5]) From the plots we made earlier, we expect that for small values of R, we will have a spiral sink, but it will probably not spiral for sufficiently large values of R. We can check this by solving for the point where $Tr(J)^2 = 4\det(J)$, which happens when $R = 2\sqrt{2}$.

```
> solve( (trfix)^2 = 4*detfix, R);
```

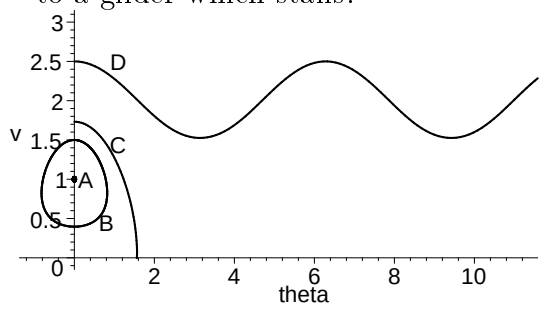
$$2\sqrt{2},\ -2\sqrt{2}$$

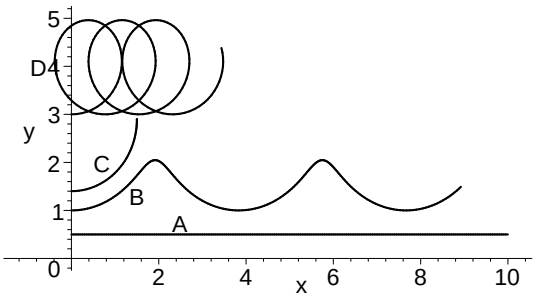
7 Qualitative Classification of Solutions

This analysis of the previous section allows us to completely classify what happens in the Phugoid model for all $R \in [0, \infty)$. In all cases, the fixed point in (θ, v) coordinates corresponds to the solution

$$\theta(t) = -\arctan\left(R\sqrt{\frac{1}{1+R^2}}\right) \quad v(t) = \sqrt[4]{\frac{1}{1+R^2}} \quad x(t) = \frac{t}{(1+R^2)^{\frac{3}{4}}} \quad y(t) = \frac{-R\,t}{(1+R^2)^{\frac{3}{4}}}$$

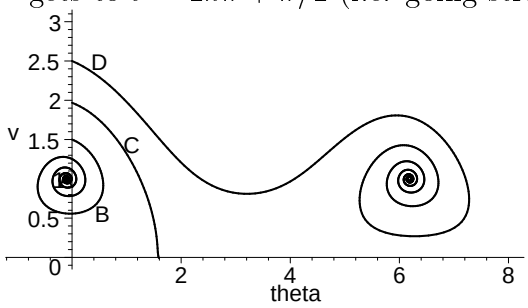
$R = 0$**:** In this case, there is no drag. The fixed point at $\theta = 0, v = 1$ corresponds to a glider flying level with a constant speed (labeled `A` in the figures below). The fixed point is a center in θ, v coordinates: nearby solutions are closed curves and correspond to a glider with an oscillatory path, alternately diving and climbing (labeled `B`). For initial conditions further away from the fixed point, $v(t)$ oscillates, but $\theta(t)$ constantly increases (see `D`). Such solutions correspond to a glider endlessly looping, and a pilot with a severe case of nausea. Between these two is a solution which cannot be continued beyond a certain time (`C`), because $v(t)$ becomes zero and our equations are no longer defined. This corresponds to a glider which stalls.

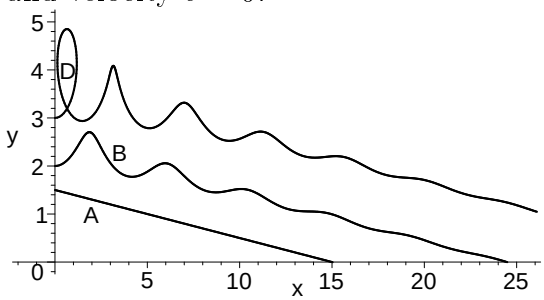




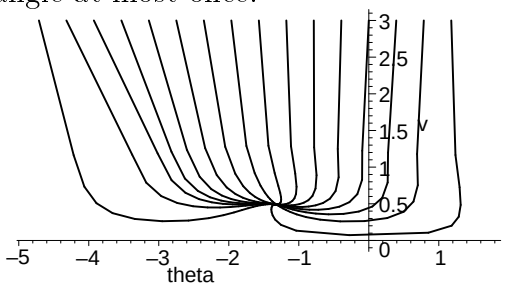
[5] We didn't address this point. If the trace is nonzero, it is impossible to have a center. But if the trace is zero, then the higher-order terms in the Taylor series play a role, and anything is possible. In fact, it can be shown that the fixed point for $R = 0$ is indeed a center. One way to do this is to show that the quantity $E(\theta, v) = v^3 - 3v\cos\theta$ must be constant for any solution when $R = 0$. Thus, the solutions are the level sets of E, which has a local minimum at $\theta = 0, v = 1$. Around a local minimum, the level sets are simple closed curves, so the fixed point is indeed a center.

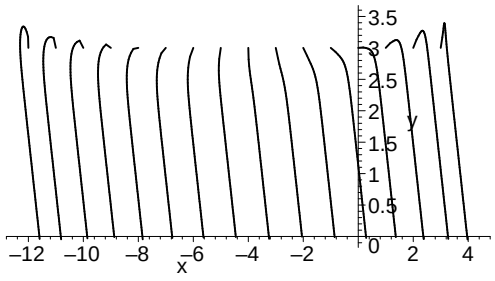
$0 < R < 2\sqrt{2}$**:** Here the fixed point (A) corresponds to a glider which dives with a constant velocity and not too steep an angle (the steepest angle is slightly less than $\pi/4$ below horizontal). The fixed point is a spiral sink: nearby initial conditions spiral into it (B). Such a spirals corresponds to a glider for which the angle and velocity constantly oscillate, but these oscillations get smaller and smaller as time passes, limiting on the same angle and velocity as the fixed point. Solutions with initial conditions further away (D) from the fixed point do some number of loops before settling into a pattern of oscillations which limit on the diving solution. Between the solutions which do k loops and those which do $k+1$ loops lies a solution which stalls out after k loops (C) — at some time the solution gets to $\theta = 2k\pi + \pi/2$ (i.e. going straight up) and velocity $v = 0$.





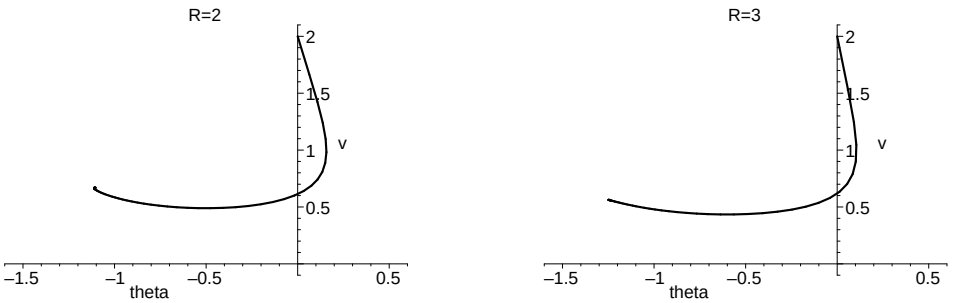
$R \geq 2\sqrt{2}$**:** The fixed point is a sink corresponding to a steeply diving solution.[6] Causing the glider to loop becomes increasingly difficult as R increases. For example when $R = 3$, if the initial angle θ is zero, an initial velocity of larger than 86.3 is required to get the glider to do one loop. If the initial angle differs from that of the fixed point, the glider fairly quickly turns towards that angle. No oscillation occurs; the angle can pass the limiting angle at most once.



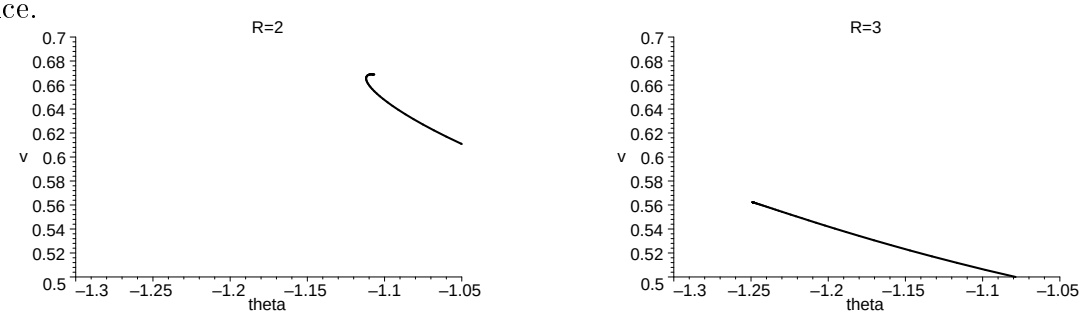


We have shown that for $R < 2\sqrt{2}$, the fixed point is a spiral sink: nearby solutions oscillate toward it. We should, however, point out that while the oscillations are always present mathematically, they can be very hard to discern. To emphasize this, we will compare a solution in the θ, v-plane for $R = 2$ (a spiral sink) and $R = 3$ (a sink with two real eigenvalues).

[6] For $R = 2\sqrt{2}$, the linearization corresponds to a degenerate case with a double eigenvalue, rather than a regular sink. However, what we say here still applies.



The two plots look rather similar. However, if we look closely at the solutions near the fixed point (we can use `zoom` to do this without recomputing the picture), we see a significant difference.



For $R = 2$, there is a "hook" at the end, but for $R = 3$, the solution goes straight in to the fixed point. Further magnifications show the same pattern, as you may want to verify for yourself. The spiral for $R = 2$ is always present, but very tight and hard to detect. Similarly, the oscillations in the x, y-plane when R is just slightly less than $2\sqrt{2}$ are nearly indiscernable, but present.

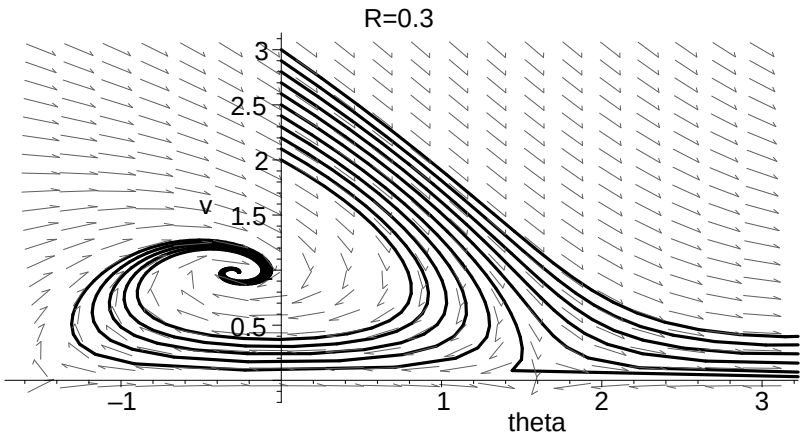
8 Dealing with the Singularity

The phugoid equations

$$\dot{\theta} = \frac{v^2 - \cos\theta}{v} \qquad \dot{v} = -\sin\theta - Rv^2$$

run into trouble when $v = 0$, because $\dot{\theta}$ becomes infinite as $v \to 0$. Since there are very few solutions for which v is ever zero, you might be tempted to ignore this issue, as we have done so far. But in fact, since we are solving the system numerically (rather than exactly), we run into problems even when v is merely small. So far, we have taken care to avoid such situations. But now let's throw caution to the wind, and see what happens.

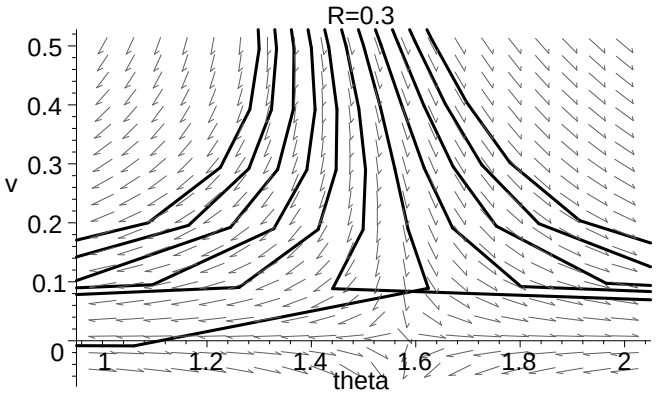
```
> R:=0.3:
  DEplot([diff(theta(t),t)=(v(t)^2 - cos(theta(t)))/v(t),
          diff(v(t),t)    = -sin(theta(t)) - R*v(t)^2],
    [theta,v], t=0..10,
    [seq([theta(0)=0,v(0)=2+i/10],i=0..10)],
    theta=-1.5..3, v=-0.05..3, title="R=0.3",linecolor=black,stepsize=0.1);
```



Something looks very wrong with the solution which starts at $\theta = 0, v = 2.5$. Notice how the solution makes a sharp turn, heading in a completely different direction from the vector field. Why is this happening?

Just to verify that it isn't a fluke, let's examine several solutions with initial conditions $\theta(0) = 0$ and $2.4 < v(0) < 2.6$, looking in the problem area. We'll use `obsrange=false` to tell Maple we want it to continue computing the solutions, even when they go outside of the viewport.

```
> DEplot( [diff(theta(t),t) =  (v(t)^2 - cos(theta(t)))/v(t),
           diff(v(t),t)     =  -sin(theta(t)) - R*v(t)^2 ],
    [theta,v], t=0..10,
    [seq([theta(0)=0,v(0)=2.4+.02*i],i=0..10)],
    theta=1..2, v=-0.05..0.5, obsrange=false,
    title="R=0.3",linecolor=black,stepsize=0.1);
```



Even worse! Not only does the solution for $v(0) = 2.5$ go the wrong way, the one with $v(0) = 2.52$ also turns the wrong way, crossing over another solution. Since our equations are autonomous, this is *never* supposed to happen.

We can remedy this by decreasing the stepsize significantly. Decreasing the stepsize by a factor of 10 (to 0.01) in the above example gives solutions which do the right thing. But a significant price in computation must be paid, and this doesn't really solve the problem. There are other initial conditions nearby which lead to the same trouble.

In order to decide how to fix the problem, we must first understand what is going wrong. The first problem we noticed happened for a solution with $\theta(t_i) \approx 1.44$, $v(t_i) \approx 0.09$, using a stepsize $h = 0.1$. Since Maple is using a Runge-Kutta method as described in §4.2, let's calculate what is happening.

Recall that in Runge-Kutta, we evaluate the vector field at four points, and average them together to get the next point. We denoted these vectors as $\vec{k}_\ell$, $\vec{k}_c$, $\vec{k}_m$, and $\vec{k}_r$. Since we have to calculate the vector field at four points, we'll write a little function to do this for us, and use the fact that Maple lets us do aritmetic on lists as though they are actual vectors; that is, we can add them and multiply by scalars.[7]

```
> h:=0.1:
  VF:=p->[(p[2]^2 - cos(p[1]))/p[2], -sin(p[1])-R*p[2]^2]:

  p:=[1.44, 0.9];
  k[l]:=VF(p);
  k[c]:=VF(p+h/2*k[l]);
  k[m]:=VF(p+h/2*k[c]);
  k[r]:=VF(p+h*k[m]);
  step:=h/6*(k[l] + 2*k[c] + 2*k[m] + k[r]);
```

$$\begin{aligned} p &:= [1.44, 0.09] \\ k_l &:= [-1.359152319, -.9938883482] \\ k_c &:= [-4.858468860, -.9808007323] \\ k_m &:= [-8.872157510, -.9314790880] \\ k_r &:= [270.3559960, -.5250618133] \\ step &:= [4.025593182, -.08905849670] \end{aligned}$$

The problem here is the vector $\vec{k}_r$, which has a huge $\dot{\theta}$ component, and points in a different direction from the others. This happens because k_r is the value of the vector field calculated at $\theta \approx 1.35$, $v \approx -0.00315$. Not only is the value of v very small here (giving a huge $\dot{\theta}$ component), it is outside the range of allowed values, and the vector field points the wrong way. Similar problems will occur for any solution which comes too close to the $v = 0$ axis with $v(t)$ decreasing.

Now the question is how to avoid this? One not very satisfying answer is just to use a very small step size. But the problem with this idea is that we then must do a huge amount of computation, even when the vector field is behaving nicely.

A better solution is to use an *adaptive stepsize*: when the vector field is "nice", we can use a large step, and when the vector field is being troublesome, we use a small step. Maple has such methods built in to it, such as the Runge-Kutta-Fehlberg method, which we can use by appending `method=rkf45` to the list of options to `DEplot`.

Sometimes, it is not always possible to change the numerical method. In the case of the phugoid, we can accomplish the same goal (in fact, gain something extra) by adjusting the

[7] This ability is not present in releases prior to Maple 6, making this computation a little more cumbersome.

length of the vectors that make up the vector field.

If we think of our solution curves as being parametric curves with tangent vectors which coincide with the vector field, then changing the lengths of the vectors (but not their direction) will not change the solution *curves*, only the parameterization. That is, we can rescale time so that the vectors no longer blow up as solutions approach the $v = 0$ line.

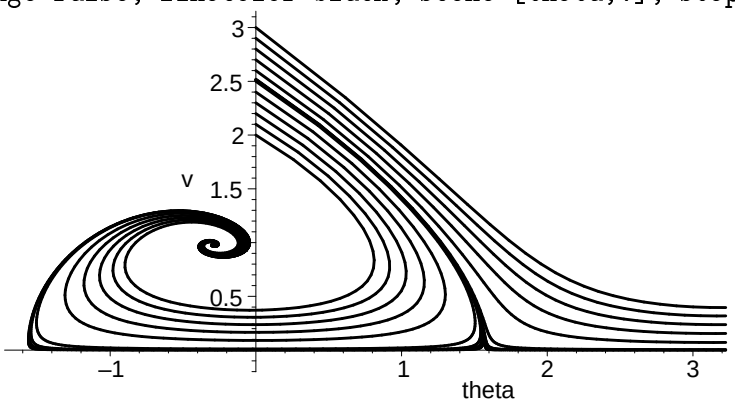
Multiplying each vector by $v(t)$ cancels out the problems in $\dot{\theta}$, and gives a system which has the same solution curves as long as $v > 0$. Thus, we get the new system

$$\dot{\theta} = v^2 - \cos\theta \qquad \dot{v} = -v\sin\theta - Rv^3 \qquad \dot{x} = v^2\cos\theta \qquad \dot{y} = v^2\sin\theta$$

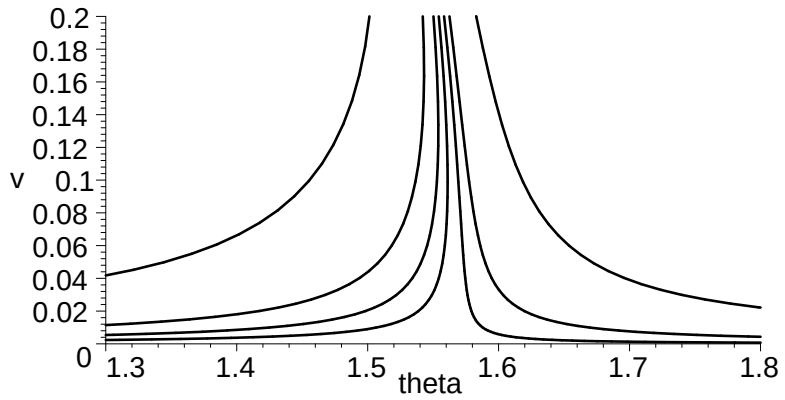
This new system of equations is defined and continuous for all values of v and θ.

Furthermore, if we want to keep track of the original time variable t, we can add a new variable T. In the original, non-rescaled equations, the time could be described as $T(t) = t$, or equivalently, $\dot{T} = 1$. After rescaling the vector field by the factor of v, this gives $\dot{T} = v$.

```
> R:='R';
  vphug:= [ diff(theta(t),t) =  v(t)^2 - cos(theta(t)),
            diff(v(t), t)    = -v(t)*sin(theta(t)) - R*v(t)^3,
            diff(x(t), t)    =  v(t)^2 *cos(theta(t)),
            diff(y(t), t)    =  v(t)^2 *sin(theta(t)),
            diff(T(t), t)    =  v(t) ]:

  R:=0.3:

  DEplot(vphug, [theta, v, x, y, T], t=0..20,
         [seq([theta(0)=0,  v(0)=2+i/10, x(0)=0, y(0)=5, T(0)=0],i=0..10),
              [theta(0)=0,  v(0)=2.51, x(0)=0, y(0)=5, T(0)=0],
              [theta(0)=0,  v(0)=2.52, x(0)=0, y(0)=5, T(0)=0],
           seq([theta(0)=0,  v(0)=2.51+i/1000, x(0)=0, y(0)=5, T(0)=0],i=2..5) ],
         theta=-1.5..3, v=-0.05..3, x=0..10, y=0..5,  T=0..20,
         obsrange=false, linecolor=black, scene=[theta,v], stepsize=0.1);
```




```
> zoom(%, 1.3..1.8, 0..0.2);
```



As you can see, using the desingularized equations solves the problems for small values of v.

In addition, we can see additional structure not apparent in the original equations. There are new fixed points at $v = 0$, $\theta = \pm\pi/2$. These fixed points are saddles, as we can see by looking at the Jacobian. In the desingularized equations, the Jacobian is

$$\begin{pmatrix} \sin\theta & 2v \\ -v\cos\theta & -\sin\theta - 3Rv^2 \end{pmatrix}$$

so the linearization at $(\frac{\pi}{2}, 0)$ is $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$, and at $(-\frac{\pi}{2}, 0)$ we have $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$. These fixed points correspond to the solutions which stall out, and as we surmised earlier, they divide up the qualitative behaviors. The eigenvectors for both of them are parallel to the θ and v axes, which says that solutions tending to a stall (or leaving one) do so by flying nearly straight up (resp. down) with an angle of $\pi/2$ (resp. $-\pi/2$). Solutions with very small velocity and an angle slightly less than $\pi/2$ flip over rapidly to an angle slightly more than $-\pi/2$ and then begin to increase their velocity.

While it is possible to deduce this behavior from the original equations, it is much more obvious one the singularity is filled in. Similar techniques are used in other branches of mathematics (blowing up a singularity in algebraic geometry, construction of a collision manifold in celestial mechanics) with great success.

References

[BDH] P. Blanchard, R. Devaney, G. R. Hall, *Differential Equations*. Brooks/Cole, 1998.

[EP] C. H. Edwards, D. E. Penney, *Differential Equations & Linear Algebra*. Prentice-Hall, 2001.

[HW] J. H. Hubbard, B. H. West, *Differential Equations: A Dynamical Systems Approach. Part 1: Ordinary Differential Equations*, Texts in Applied Mathematics **5**. Springer-Verlag, New York. 1991. *Part 2: Higher Dimensional Systems*, Texts in Applied Mathematics **18**. Springer-Verlag, New York. 1995.

[Lan] F. W. Lanchester, *Aerial Flight Vol II: Aerodonetics*. Colchester & Co., London. 1908.

Chapter 4

fsqFsHn sGGousG

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 name "Cryptography" comes from the Greek words *kryptos* ($\kappa\rho\upsilon\pi\tau o\sigma$), meaning "hidden" or secret, and *graphia* ($\gamma\rho\alpha\phi\iota\alpha$), meaning writing.

Since ancient times, cryptography has been a part of military and governmental communications. More recently it has become part of nearly everyone's life because of the Internet, electronic banking, and so on.

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 deciphering (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

[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 [Kahn, p. 521]),

> 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.

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! It is perfectly all right to have the same message encode to two different ciphertexts, as long as we can get the original when deciphering.)

2 Simple Ciphers

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.

Such substitutions are very simple to implement in Maple. We will use the `CharacterMap` function from the `StringTools` package[2] which does exactly what we want. First, we define two "alphabets"– one for our plain text, and one for our encrypted text. The second is a permutation of the first.

```
> with(StringTools):
  Alphabet :="abcdefghijklmnopqrstuvwxyz":
  Cryptabet:="THEQUICKBROWNFXJMPSVLAZYDG":
```

Now we define a pair of functions[3] to encrypt and decrypt our text.

```
> Scramble  := msg  -> CharacterMap(Alphabet, Cryptabet, msg):
  Unscramble:= code -> CharacterMap(Cryptabet,Alphabet, code):

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

"JWUTSU QX FXV PUTQ VKBS"

[2]The `StringTools` package was introduced in Maple 7, and the string data type was introduced in Maple V release 5. Everything we do in this chapter can easily be implemented without using `StringTools`– in fact, the original version of this chapter was written for Maple Vr4 using names rather than strings. Since Maple 6 is still in widespread use (Spring 2002), we'll try to include alternatives to using this package when practical.

[3]In Maple Vr5 and Maple 6, we could define `CharacterMap` as

```
CharacterMap := (old,new,txt)->cat(seq(new[SearchText(txt[i],old)],i=1..length(txt)));
```

This will only work if all characters in the `txt` correspond to ones in `old`, so a space would have to be added to both `Alphabet` and `Cryptabet` to get this example to work.

```
> Unscramble(%);
```

"please do not read this"

Note that our message contains a spaces which are preserved in the encryption process, because the `CharacterMap` function only modifies those characters which are found in the first string. If a character isn't found, it is left alone. Thus, if we try to scramble a message containing punctuation or upper-case letters, they aren't modified by `Scramble`. Since our `Cryptabet` is all upper-case, a message which began with mixed case will be distorted.

```
> Scramble("He said I need direction; I head for the door.");
  Unscramble(%);
```

"HU STBQ I FUUQ QBPUEVBXF; I KUTQ IXP VKU QXXP."

"be said f need direction; f head for the door."

It is traditional in pen-and-paper cryptography to use a 26-letter alphabet (that is, to remove spaces and punctuation and ignore any distinction between upper and lower case letters). If you wish to do this, it could be done[4] using `LowerCase` (which translates all letters to lower case), and a combination of `Select` and `IsLower` to remove anything that isn't a letter.

```
> Select(IsLower,LowerCase("I don't mind suggestions. Don't give me any more."));
```

"idontmindsuggestionsdontgivemeanymore"

```
> Scramble(%);
  Unscramble(%);
```

"BXFVCNBFQSLCCUSVBXFSQXFVCBAUNUTFDNXPU"

"idontmindsuggestionsdontgivemeanymore"

In general we won't adhere to this convention in this chapter. Rather, we will treat spaces and punctuation like any other character. This is not a common approach, but one we happen to like.

There are two big drawbacks to such a substitution cipher. One primary problem is that the key for enciphering and deciphering is the permutation: you need to remember an ordering of the entire alphabet.

[4] How to accomplish this without `StringTools` is left to the seriously motivated reader.

A second problem, shared with all such monoalphabetic substitutions (i.e., those that substitute one letter for another), is that they aren't that hard to break. If you have a long enough message, you can use the fact that certain letters occur much more frequently than others to help tease out the original text. In English text the letter "e" is the occurs most frequently, followed by "t", "a", "n", "i", and so on. Spaces occur about twice as often as the letter "e", assuming they are encrypted, rather than just omitted. Once you guess a few letters, common words like "the" and "and" fall into place, giving more clues.

We will examine some polyalphabetic substitution ciphers in §5.

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 (legend has it that Caesar used an offset of 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.[5]

While we could easily implement the Caesar cipher without converting our text to numbers, doing so makes it easier to understand and sets us up for more complicated ciphers we will need later.

Treating characters as numbers

Computers already have a built-in numeric representation of the alphabet— the ASCII encoding.[6] 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 dig**it**. A group of 8 bits is called a byte, and is a single "character" of information.[7] A byte can have $2^8 = 256$ different states:

00000000, 00000001, 00000010, 00000011, ..., 11111111

[5]For example, the title of this chapter (fsqFsHn sGGousG) is actually the result of using a Caesar cipher on its original title. A 53-character alphabet consisting of all the upper-case letters, a space, and all the lower-case letters was used, so the space in the middle may not correspond to a space in the original. The three Gs that occur should be a good hint to decoding it.

[6]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.

[7]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.

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.) 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 using the characters A–F for the integers 10–15. Thus, the byte 01011111 would be written as 5F in hexadecimal, since 0101 means $2^2 + 2^0 = 5$, and 1111 is $2^3 + 2^2 + 2^1 + 2^0 = 15$.

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,[8] 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[9]

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. Maple prints a □ when the ASCII code is not a printable character. We make the table for characters up through 127 because the meanings in extended ASCII differ depending on what character set you use. Feel free to make your own table of all 255 characters. (In the interests of making a nicely formatted table, we have used a some-

[8] ASCII is not the only such standard, although it is currently the most commonly used. Prior to the widespread use of personal computers in the 1980s, EBCDIC (Extended Binary Coding for Data Interchange and Communication) tended to be in more common use, because it was what IBM mainframes used. There are many other such assignments.

[9] 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 properly. There are a number of distinct assignments of characters to the upper 128 positions. One common one is Latin-1 (ISO 8859-1), which contains the characters used in the majority of western European languages. Other assignments include characters for Scandinavian languages, Arabic, Cyrillic, Greek, Hebrew, Thai, and so on. Since the late 1990s, a multi-byte character encoding called Unicode (ISO 10646) has been in use; this universal character set allows computers to handle alphabets (such as Chinese) that need more than 255 characters. While use of Unicode is becoming more widespread, single byte character sets like ASCII will likely predominate for quite a long while.

what complicated command. The much simpler `seq([i,convert([i],bytes)],i=0..127);` would have produced the same information, just not as neatly.)

```
> matrix([ [_,seq(i, i=0..9)],
         seq([10*j, seq(convert([i+10*j],bytes), i=0..9)], j=0..12)]);
```

_	0	1	2	3	4	5	6	7	8	9
0	“”	“□”	“□”	“□”	“□”	“□”	“□”	“\a”	“\b”	“\t”
10	“\n”	“\v”	“\f”	“\r”	“□”	“□”	“□”	“□”	“□”	“□”
20	“□”	“□”	“□”	“□”	“□”	“□”	“□”	“\e”	“□”	“□”
30	“□”	“□”	“ ”	“!”	“\“”	“#”	“$”	“%”	“&”	“ ’ ”
40	“(”	“)”	“*”	“+”	“,”	“-”	“.”	“/”	“0”	“1”
50	“2”	“3”	“4”	“5”	“6”	“7”	“8”	“9”	“:”	“;”
60	“<”	“=”	“>”	“?”	“@”	“A”	“B”	“C”	“D”	“E”
70	“F”	“G”	“H”	“I”	“J”	“K”	“L”	“M”	“N”	“O”
80	“P”	“Q”	“R”	“S”	“T”	“U”	“V”	“W”	“X”	“Y”
90	“Z”	“[”	“\\”	“]”	“^”	“_”	“ ‘ ”	“a”	“b”	“c”
100	“d”	“e”	“f”	“g”	“h”	“i”	“j”	“k”	“l”	“m”
110	“n”	“o”	“p”	“q”	“r”	“s”	“t”	“u”	“v”	“w”
120	“x”	“y”	“z”	“{”	“|”	“}”	“∼”	“□”	“□”	“□”

Note that some of the characters (such as ASCII code 10) print as a backslash and a letter, like \n. These are codes for special control characters: \n indicates a newline, \t is a tab, and so on. Since Maple indicates a string by printing it inside double quotes, it indicates the double quote[10] character (ASCII 34) using \", and the backslash character (ASCII 92) with a pair of backslashes. While it doesn't print out in Maple, ASCII 127 is the control sequence DEL (sometimes printed as ^?). This sometimes is interpreted as “delete the previous character”; if your output device interprets it that way, characters can seem to mysteriously vanish.

Another special character is ASCII code 0: note that it prints as nothing, not even a blank space (ASCII 32 is the space character). This is the null character, and is typically used internally to indicate the end of a string. If a null occurs in the middle of a string, usually what comes after it won't print out unless you take special steps. To see this for yourself, try a command like `convert([73,116,39,115,0,103,111,110,101,33],bytes)`. You'll see that the characters following the 0 don't print out.

The `StringTools` package also contains `Char` which gives the character corresponding to a decimal number, and `Ord` which gives the number corresponding to a character. We'll generally use `convert`, although it makes little difference.

[10] The ASCII code does not have distinct codes for the open-quote (“) and close-quote (”) characters used in typesetting. Neither do most computer keyboards. However, the apostrophe (') and grave accent or backquote (‘) characters have distinct ASCII codes (39 and 96, respectively).

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 a numerical representation, add a pre-specified offset modulo 127, then convert back to characters. Unlike the `Scramble` function we wrote in §2.1, we needn't declare an alphabet explicitly because we will use the ASCII codes. However, in addition to the Maple command `convert( ,bytes)`, we will use another new command, `map`.

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

We *finally* are ready. See if you can understand it before reading the explanation below. Remember to read from the middle outward. Note that we are defining the function we are `map`ping directly as the first argument.

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

Let's give it a try:

```
> Caesar("Et tu, Brute?",3);
```

"Hw#wx/#EuxwhB"

If you don't see what is going on in `Caesar`, try to think of it as a composition of several functions, each of which you DO understand. That is, `Caesar` works as follows

- First, we convert the input string (called `plain`) into a list of numbers, using the call `convert(plain,bytes)`. Note that this is done on the third line, the *innermost* function.

- Next, we apply the function `x ->(x+shift-1) mod 127 + 1` to each of numbers in the resulting list using `map`. This call to `map` has two arguments: the definition of the shift, and what to apply it to (the innermost function `convert(plain,bytes)`).

- Finally, we convert the list back to characters again. This is the outermost function: it begins on the second line (immediately after the `->` and ends with the final parenthesis.

Viewing it as a composition, we have

$$\texttt{Caesar} := \texttt{convert}^{-1} \circ \texttt{map} \circ \texttt{convert}.$$

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 afterward 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.

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 □.[11] 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. The actual character values are different, they just print out the same.

How can we get around this? We can restrict ourselves to printing characters using a variation of the idea in `Scramble` of §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.

[11] 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.

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);
```

$$5\,I$$

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("The square root of ",x,"is not a real number");
    fi;
```

```
    end;
```

$sqr3 := \mathbf{proc}(x)$
$\quad \mathbf{if} 0 \le x \mathbf{then} \operatorname{sqrt}(x) \mathbf{else} \operatorname{print}(\text{" The square root of "}, x, \text{" is not a real number"}) \mathbf{fi}$
end

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

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

```
> 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. [12]

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

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

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

What does that warning mean? It means that Maple has assumed that the variable `y` used in `mesa` 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 `mesa(19)` (which sets *its* copy of `y` to 361), then check what the value of `y` is.

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

$$y := 35$$

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

```
> mesa(19);
```

$$25$$

```
> y;
```

$$35$$

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.

```
> mesa2:= proc(x)
    global y;

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

$$mesa2 := \mathbf{proc}(x)\,\mathbf{global}\,y;\ y := x^2\ ;\ \mathbf{if}\,25 < y\,\mathbf{then}\,25\,\mathbf{else}\,y\,\mathbf{fi}\,\mathbf{end}$$

```
> y;
```

$$35$$

```
> mesa2(19);
```

$$25$$

```
> y;
```

$$361$$

This time, of course, `y` was indeed changed by invoking `mesa2`.

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. Using such statements is called declaring the scope of the variables. 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. Some computer languages require that you declare all variables you use.

Another good habit to get into is to indicate what type of arguments the function will accept. This is optional in Maple, and not always desirable (you may not always know what they will be). This is done by specifying the type after two colons in the argument list. For example,

```
> ith:=proc(l::list, i::posint)
    return(l[i]);
  end:
```

is a function that insists its first argument be a list, and the second must be a positive integer. If you call it with something else, you will get an error message.

```
> ith("Henry",8);

Error, invalid input: ith expects its 1st argument, l, to be of type list, but
received Henry
```

4 Caesar cipher redux

Now that we know about `proc`, let's implement the Caesar cipher again, this time with an alphabet of our choosing.

First, we define a global variable `Alphabet` that lists all the characters we intend to encode. We would like to be able to distinguish all characters which print out, as well as blanks (that is, those which you have keys for on your keyboard). To avoid typing in the full list of characters we want[13], we generate a list of all the integers less than 256 and convert it to a character string with `convert(,bytes)`. Then we use `Select` and `IsPrintable` to pick out the usual printable characters. Finally, since we would like to have a space and a newline as characters in our alphabet, we add them to the resulting string with `cat`.

```
> Alphabet := cat("\n\t", Select(IsPrintable,convert([seq(i,i=1..255)],bytes)));
```

"\n\t !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWX\

YZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~"

We will use this particular alphabet regularly in the next few sections.

Next, we define two functions which perform the analogue of `convert(,bytes)`, but with our special alphabet. `ListToString` will take a list of integers and replace it with the character from that position in `Alphabet`, and `StringToList` does the reverse process. For each character in the input string, `StringToList` uses `SearchText` to find its position in `Alphabet`. We need

[13] For versions prior to Maple 7, just type in the desired alphabet directly

to subtract one from this position because we would like our characters to be numbered from 0 rather than 1. Thus, in the `Alphabet` above, \t is character 0, and the numeric code for ~ is 96. `Alphabet` is declared as a global variable.

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

> ListToString := proc(numlist::list(nonnegint))
    local i;
    global Alphabet;
    cat(seq(Alphabet[numlist[i]+1], i=1..nops(numlist)));
  end:
```

Now the Caesar cipher is written in terms of these functions. Note that since 0 corresponds to a printable character, we don't have to add and subtract 1 as before.

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

    p       := length(Alphabet);
    textnum := StringToList(plaintext);
    codenum := [seq( modp(textnum[i]+shift, p), i=1..length(plaintext)) ];
    ListToString(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 `StringToList` 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` (§2.2, p. **4**:7). 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 with `ListToString`.

Let's try it out.

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

"Yhql/#Ylgl/#Ylfl"

To decode, we can just use the negative of the shift amount.

```
> Caesar2(%,-3);
```

"Veni, Vidi, Vici"

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

text := "I have heard the mermaids singing, each to each.\nI do not think \
that they will sing to me."

In this sample, we have used a text which contains a newline character. Maple represents this as \n inside a text string. Since we will treat the newline as a regular character, it too will encode to something. If you look carefully at the output below, you can probably pick it out.

```
> Caesar2(text,64);
```

"('G@UD'GD@QC'SGD'LDQL@HCR'RHMFHMFl'D@BG'SN'D@BGn^\
('CN'MNS'SGHMJ'SG@S'SGDX'VHKK'RHMF'SN'LDn"

```
> Caesar2(%,-64);
```

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

If we would like to see the decrypted text with the newlines expanded, we need to use `printf`.

```
> printf(%);
```

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

One thing that we should check is what happens if our message should contain a character that is not in our alphabet. Rather than changing our text, we'll dramatically shorten the alphabet.

```
> Alphabet:="abcdefghijklmnop_";
```

Alphabet := "abcdefghijklmnop_"

Now let us go then, you and I, and re-encode the test message using the new alphabet, but 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(text,0);
```

"_ha_e_hea_d__he_me_maid__inging_each__o_each___do_no__hink__ha__he__ill_ing__o_me_"

Can you explain why this happens?

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.

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[14], which was first used around 1600, and was popularly believed to be unbreakable.[15] This cipher is called a polyalphabetic substitution cipher, because several different substitutions are made depending on the position of the character within the text.

In our first 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.

[14] 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. More details van be found in [Kahn].

[15] 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.

Now we implement this in Maple. This is very similar to the Caesar cipher, just with the extra complication of multiple shifts, and letting our key be a string.

First, we set our Alphabet to the usual one. We also use the conversion functions from §4.

```
> Alphabet := cat("\n\t", Select(IsPrintable,convert([seq(i,i=1..255)],bytes))):
> Vignere:= proc(plaintext::string, key::string)
    local textnum,codenum,i,p,offsets,keylen;
    global Alphabet;

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

To try it out, we'll the same text as in the previous section. Notice how much harder it is to pick out the word boundaries in the resulting ciphertext.

```
> coded:=Vignere(text,"Prufrock");
```

coded := "{tˆHiUeT6ThKtdLQR‘[Y‘QMPDtiPaWMZ8\twLTSLmEbwLTSL \
{Rr?hW_eZ@gw[[YRWRgˆHgqXT6lwˆ\\\\PmD\\dNtdSm>X$"

We can make the decoding function from the original (let's call it `unVignere`) by changing exactly one + sign to a −.[16] 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.

```
> printf(unVignere(coded,"Prufrock"));

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

Even though this scheme looks quite daunting, it is not so very hard to crack if you use a computer or have a very large supply of perseverance. 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 key 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.

5.2 One-time pads

Note that the longer the key is in the Vignère cipher, the harder it is to break. If the key were as longer than the text, then it might seem at first that analyzing the frequency of letters in the encrypted text would be of no help, since each letter would be shifted by a different

[16] 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 "`M+(7+.:2`": the numeric code of `P` plus the numeric code of `M` is 97 (the length of the alphabet), similarly for `r` and `+`, `u` and (, and so on.

amount. This is *almost* true. If the key is an passage of text in English, then the shifts will occur with a predictable frequency. Of course, the problem gets very difficult, but cryptanalysts are persistent people.

But what if there were no predictability within the key, having the shifts come at random? The result (a Vignère cipher with an infinitely long, completely random key) is an cryptosystem that *cannot be broken.* Since the shifts are random, if you manage to decipher part of the message, this gives you no clue about the rest. Furthermore, any plaintext of a given length can encrypt to any ciphertext (with a different key, of course). For example, the ciphertext "=5nwhn KDNO?uWTBC-XA" might have come from the phrase "Let's have dinner." (with the key "pOyntmbbXYtrjSTGe1"), or it might be the encryption of "Attack at midnight" with the key "{@y4#!Jbz>&moSYEoL". Since any message could encrypt to any other, there is no way to break such a code unless you know the key.

But that is the problem: the key is infinitely long. Infinitely long, truly random sequences of numbers tend to be somewhat unwieldy. And to decode the message, you must know what random sequence the message was encoded with.

Such a system is called a one-time pad, and was used regularly by spies and military personnel. 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 could be given his own codebook (so that if he is captured, the whole system is not compromised), and he uses one page per message. Central command has on file the books for each agent.

A variation on this theme is the Augustus cipher,[17] 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.

Another issue is that to be truly unbreakable, the random sequence must be truly random, with no correlation among the characters. This is harder than it sounds— real randomness is hard to come by. If the random sequence has some predictability, the resulting stream can be attacked. A number of attacks on cryptosystems have been made not by breaking the encryption scheme directly, but because the underlying random-number generator was predictable.

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.[18] You might think that generating a

[17] Sometimes a Caesar cipher with a shift of +1 is also called an "Augustus Cipher", even though these are very different ciphers.

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

random sequence of numbers would be inherently unreproducible, so the message would be indecipherable unless we record the stream of random numbers. 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.[19] 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 that can be called later 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`.[20]

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

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

In most descriptions of one-time pad systems, one takes 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 truly 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

[19] 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. For example, Maple's `rand` function (and that of most computer languages, such as C, Fortran, and Java) gives the result of a affine sequence of numbers, reduced to some modular base. That is, $x_i = ax_{i-1} + b$ and $s_i = x_i \bmod n$ for some fixed choices of a, b, and n. In this setting, the seed is x_0. We shall ignore the problem that this sequence is guessable, but if you want real security, you cannot.

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

for the version given above; to make a decryption procedure, we would need to modify the above by changing the + to a −.

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. We will use 99-graphs on a 256-letter alphabet (the ASCII code) when we implement the RSA cryptosystem in §11.2. While frequency analysis is still possible (charts of digraph frequencies are readily available, trigraphs less so), the analysis is much more complex.

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 $7 \times 26 + 8 = 190$, assuming the correspondence of H=7, I=8. To convert back, we look at the quotient and remainder when dividing by 26. For example, $300 = 26 \times 11 + 14$, yielding "LO".

Either we can do this arithmetic ourselves directly, or we can use the `convert(,base)` command. This command takes a list of numbers in one base and converts it to another. One slightly confusing fact is that in this conversion, the least significant figure comes first in the list,[21] instead of the usual method of writing numbers with the most significant digit first.

For example, $128 = 1 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$ would be written as `[8,2,1]`. To convert this to base 16, we would note that $128 = 8 \times 16^1 + 0 \times 16^0$, so in base 16, it is written as 80.

Doing this calculation in Maple, we have

```
> convert([8,2,1], base, 10, 16);
```

$$[0, 8]$$

Below is one way to implement the conversion of text to numeric values for k-graphs. We assume our usual functions `StringToList` and `ListToString` are defined (see §4), 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. A block of `k` characters $c_1c_2c_3\ldots c_k$ is assigned the numeric value $\sum_{i=0}^{k-1} x_i p^k$, where x_i is the numeric equivalent of c_{i+1} assigned by `StringToList`.

[21] Such a representation is called little-endian, as opposed to a big-endian one. Some computers represent numbers internally in big-endian format (Sun SPARC, Power Macintosh), and others use a little-endian representation (those using Intel processors, which is what MS-Windows and most versions of Linux run on). This name comes from Gulliver's Travels, where in the land of *Blefuscu* there is war between the big-endians (who eat their eggs big end first), and the little-endians (who eat eggs starting from the little end).

```
> StringToKgraph := proc(text::string, k::posint)
    local p;
    global Alphabet;
    p:= length(Alphabet);
    convert(StringToList(text), base, p, p^k);
  end:
> KgraphToString := proc(numlist::list(nonnegint), k::posint)
    local p;
    global Alphabet;
    p:=length(Alphabet);
    ListToString( convert(numlist, base, p^k, p));
  end:
```

In the examples below, we are using our usual 97-character alphabet. Of course, this will work on any alphabet, although the specific numbers will differ.

```
> StringToKgraph("two by two",2);
```

$$[8719, 275, 8895, 8344, 7946]$$

In our alphabet, "t" is character number 86 and "w" is number 89, so the digraph "tw" encodes as $86+89\times97 = 8719$ (remember we are using a little-endian representation). Similarly, "o " gives $2 \times 97 + 81$, and so on. Notice that the two occurrences of "two" give different numbers, because one begins on an even character and the other starts on an odd one. The first corresponds to the digraphs "tw" and "o ", while the second is " t" and "wo".

We can also encode with 4-graphs, if we like.

```
> StringToKgraph("two by two",4);
```

One advantage (for us) of Maple's use of the little-endian order is that we needn't worry whether the length of our text is divisible by k. In the above example, the last 4-graph is "wo", which is encoded as though it had two more characters on the end with numeric code of 0. The disadvantage of this is that if our text ends with the character with code 0 (in our standard 97-character alphabet, this is a newline), that character will be lost.

Another way to treat multiple characters together is to think of them as vectors. For example, the digraph "by" might correspond to the vector [68, 91]. We will treat this approach in §8.

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 §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.[22]. 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.[23] Below is a small example of a procedure which encrypts a file one line at a time using a procedure `EncryptLine`. We have never given such a procedure, but it should be clear how to modify it to use your favorite encryption routine instead.

```
> 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:
```

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 §5 can 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 $x \mapsto$

[22] 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`

[23] On several operating systems, including MS-Windows and MacOS (but not Unix or Linux), 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, text and binary files are the same, so no translation is necessary, but there is no harm in specifying the file type.

$ax+b \pmod p$ 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.

7.1 When do affine encodings fail?

You might be wondering if this will work at all. 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 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 97 is prime and large enough so that we can include all the usual printable ASCII characters (see §2.2, especially the table on page **4**:6). 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 (since the length of the alphabet will be either 128 or 256, both or which are powers of 2).

7.2 Implementing and using an affine encoding

Implementing this scheme is a trivial modification to the `Caesar2` procedure we wrote in §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.

In light of the Prop. 7.1, we will also add some error checking to ensure that `a` and the length of the alphabet are relatively prime. Recall that if they are not, the encryption will produce a message that cannot be decrypted. We use the `error` command[24] if this is the case. While we typically use an alphabet of prime length, we may not always do so.

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

    p       := length(Alphabet);
    if (gcd(a,p)>1) then
       error("The %-1 parameter %2 must be prime to the length of Alphabet %3",
             2,a,p);
    fi;
    textnum := StringToList(plaintext);
    codenum := [seq( modp(a*textnum[i]+b, p), i=1..length(plaintext)) ];
    ListToString(codenum);
  end:
> mess:=AffineCode("A fine mess, Stanley!",10,5);
```

$$mess := \text{“\^{}7Lj;B71BmmN7Pw\{;'BHA”}$$

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. We'll take the latter approach.

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 $\{0, 1, \ldots, p-1\}$. Such an element exists exactly when a and p are relatively prime, as a consequence of Prop. 7.1. Since Maple knows about division modulo p, it allows us to use this notation, which is quite convenient.

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


In our definition of `AffineCode` we insisted that `a` and `b` be integers. This means that using a call like `AffineCode(mess,1/10,-5/10)` will give an error, since `1/10` is not an integer.

[24] Versions prior to Maple 6 need to use `ERROR` instead, which has slightly different behavior.

We can fix that by changing the definition of `AffineCode` to be `proc(plaintext::string, a::rational, b::rational)` or we can explicitly calculate these modular expressions before passing them to `AffineCode`.

```
> AffineDecode:= proc(ciphertext::string, a::integer, b::integer)
    local A, B, p;
    global Alphabet;
    p := length(Alphabet);
    A :=  1/a  mod p;
    B := -b/a  mod p;
    AffineCode(ciphertext, A, B);
  end:
```

```
> AffineDecode(mess,10,5);
```

"A fine mess, Stanley!"

7.3 Breaking an affine cipher

Suppose we have a message that was encrypted with an affine cipher, and we also know the `Alphabet` in use. 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 the message was enciphered with $x \mapsto y = (ax + b) \bmod p$, and we have 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 have Maple do this for us using `msolve`.

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

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

Thus, we see that the message was encoded by $x \mapsto 47x + 25 \pmod{53}$, and can be decrypted using $a = 1/47 \pmod{53}$ and $b = -25/47 \pmod{53}$. Of course, it would have been more efficient to solve for the *deciphering* transformation directly.

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

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

```
> Alphabet:="ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz":
  AffineCode("nrIbQyxEbkrI bwr EUbJaPybL abpyJJy UbUlrrxU", 44, 13);
```

"You bend your words like Uri Gellers spoons"

8 Enciphering matrices

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 §5.3, we could think of it as a vector of characters. Thus, if we were using, say, 2-vectors, then the name "Dougal" would become the three vectors [68, 111], [117, 103] and [97, 108] (assuming we are using the ASCII codes). If the length of the text we were converting was not divisible by 2, we would have to extend it by adding a character to the end. Typically, this would be a space, or perhaps a null character (ASCII 0). Thus the name "Ian" occurring at the end of a message would become "Ian ", and might encode as the pair of vectors [73, 97] and [110, 32].

Below is an implementation of this conversion, `StringToVects`, which takes as input the text to convert to n-vectors and the length of the vectors, n. It uses the `StringToList` procedure from §4, as well as the usual Maple commands.

```
> StringToVects:= proc(text::string,n::posint)
    local i,j,textnums,vectors;
    textnums := StringToList(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.

Strictly speaking, the result is a list of lists, rather than a list of vectors. Maple is happy to accept a list in contexts where a vector is wanted.

Converting back from a list of n-vectors to a character string is easy, because we already have `ListToString` to convert a list of numbers to a character string for us. All that is needed is remove the inner structure from each of our vectors, leaving only the sequence of numbers. 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 for Maple, there is a difference 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.

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

$$[1..2, [1 = 36, 2 = 79], 1..2, [1 = 85, 2 = 71], 1..2, [1 = 65, 2 = 76]]$$

In order to be sure that our function will work with both vectors and lists, we need to convert the vectors to lists first. This is easily done with the Swiss army knife `convert`. The unusual-looking type declaration in the function says that we will be happy to accept a list of lists, a list of vectors, or a list of Vectors,[25] provided their elements are integers.

```
> VectsToString:=
    proc(vects::list({list(integer), vector(integer), Vector(integer)}))
      local l;
      l:=map(convert, vects, list);
      ListToString(map(op, l)):
  end:
```

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 (§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.

[25] Maple has two different linear algebra packages, `linalg` and the newer `LinearAlgebra` introduced in Maple 6. The `vector` type is from the former, `Vector` from the latter.

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 after Lester Hill, who studied such cryptosystems in the late 1920s and early 1930s [Hil1], [Hil2].

In Maple, one does matrix arithmetic pretty much as you would expect, except that we need to use `&*` to indicate matrix multiplication. This is because matrix multiplication is not commutative, while `*` is. Using `&*` tells Maple not to attempt the multiplication right away, and so it doesn't carry out the operations until you execute `evalm`. When the `evalm` function is executed, it does the matrix multiplication properly. Thus, to compute $Av + b$, we would write `evalm(A &* v + b)`. Because we want to reduce everything modulo `p`, we `map modp` onto the matrix. Since we might want to specify an element as the inverse of another, we allow our matrix and vector to contain rationals. As before, we accept several types of inputs.

```
> with(linalg):
  AffineMatEncode :=
   proc(text::string,
        A::{matrix(rational),Matrix(rational),list(list(rational))},
        B::{vector(rational),Vector(rational),list(rational)})
   local   vtext,vcrypt,i,n,p;
   global  Alphabet;

   p:=length(Alphabet);
   n:=nops(convert(B,list));  # look at B to get the size

   if ( gcd(linalg[det](A),p)>1) then
       error("Determinant of %1 is not prime to the length of Alphabet %2",
              A,p);
   fi;

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

Note that matrices 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 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$:

We should point out that all the examples in this section are using the 53-character alphabet of the previous section.

```
> AffineMatEncode("I am so mixed up", [[0,1],[1,0]], [0,0]);
```

" Imas oimex dpu"

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([[12, 27], [46, 44]]); B:=[25, 50];
```

$$A := \begin{bmatrix} 12 & 27 \\ 46 & 44 \end{bmatrix}$$

$$B := [25, 50]$$

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

xMPKFUGRxtaa

```
> AffineMatEncode(%, inverse(A), evalm(-inverse(A) &* B));
```

"Lester Hill "

8.3 A Known-plaintext attack on an affine matrix cipher

As in §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 $n+1$ different 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 ciphertext "cilbup". This is sufficient information to decipher the message, because we know the encodings of three different 2-vectors.

```
> Alphabet := cat("\n\t", Select(IsPrintable,convert([seq(i,i=1..255)],bytes))):
  plain:=StringToVects("secret",2);
```

$$plain := [[85, 71], [69, 84], [71, 86]]$$

```
> crypt:=StringToVects("cilbup",2);
```

$$crypt := [[69,\ 75],\ [78,\ 68],\ [87,\ 82]]$$

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.[26]

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];
```

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

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

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]);
```

$$[69\,a + 75\,b - 85 + e,\ 69\,c + 75\,d - 71 + f]$$

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)};
```

$$eqns := \{87\,c + 82\,d - 86 + f,\ 87\,a + 82\,b - 71 + e,\ 78\,c + 68\,d - 84 + f,$$
$$78\,a + 68\,b - 69 + e,\ 69\,c + 75\,d - 71 + f,\ 69\,a + 75\,b - 85 + e\}$$

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

[26] Do you see why this is so? Hint: think about the relationship between matrices and linear functions.

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

$$Answer := [\begin{bmatrix} 42 & 84 \\ 19 & 78 \end{bmatrix}, [5,\ 88]]$$

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.

```
> AffineMatEncode("cilbup",Answer[1],Answer[2]);
```

"secret"

It is worth noting that combining an affine matrix cipher with a large multibyte alphabet (see §5.3) greatly increases its security, at least if one restricts to short messages, each using different keys. While the matrix cipher is still susceptible to a known-plaintext attack, getting the plaintext becomes much harder.

9 Modern cryptography

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. This idea (that the potential cryptanalyst knows everything but the key) is called Kerckhoff's Law or sometimes Shannon's Maxim.

Kerckhoff's Law at first seems unreasonably strong rule; you may be thinking that, given a mass of encrypted data, how is the cryptanalyst to know *by what means* it was encrypted? If you are encrypting your own files for personal use and writing your own software which you keep under lock and key, this assumption is not unreasonable. 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 alone as a good measure of security. For example, when you make a purchase over the internet, the encryption method that is used between your browser and the seller's web server is public knowledge. Indeed, it must be public: if not, only those privy to the secret could create web browsers or web servers. The security of the transaction depends on the security of the keys.

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[27]).

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.[28] 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.

DES was originally expected to be used for "only a few years" when it was first designed. However, due to its surprising resistance to attack it was generally unassailable for nearly 25 years. The powerful methods of linear and differential cryptanalysis were developed to attack block ciphers like DES.

In the January of 1997, the National Institute for Standards and Technology (NIST) issued a call for a new encryption standard to be developed, called AES (the Advanced Encryption Standard). The requirements were that they operate on 128-bit blocks and support key sizes of 128, 192, and 256 bits. There were five ciphers which advanced to the second round: MARS, RC6, Rijndael, Serpent, and Twofish. All five were stated to have "adequate security"– Rijndael was adopted as the standard in October of 2000. Rijndael was created by the Belgian cryptographers Joan Daemen and Vincent Rijmen; the underlying mathematics is the algebra of polynomials in the finite field $GF(2)$.

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 in early 1995 in about 8 days using 112 networked computers (see Damien Doligez's web page about cracking SSL[29] for more details); later in the same year a second session was broken in under 32 hours. Given the speed increases in computing since then, it is reasonable to believe that a 40-bit key can be cracked in a few hours. Notice, however, that both of these attacks were essentially brute-force, trying a large fraction of the 2^{40} possible keys. Increasing the key size resolves that problem. Nearly all browsers these days use at least 128-bit keys.

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

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

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 different message with the same hash.

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).

It is worth remarking here that computer systems can use a cryptographic hash function as a way to store a password (for example, Linux allows the use of MD5 for passwords). A computer password never need be decrypted; it only needs to be verified. Instead of storing the password itself, or an encrypted version of it, the computer stores the corresponding hash. Then, when a user signs in, the password is hashed and compared to the stored value. If they match, the user typed the right password (or managed to find another password with the same hash, which is computationally impossible).

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 $x \mapsto ax + b \bmod p$ (see §7), if we know the values of a and b used to encode the message, we can quickly compute the deciphering key $1/a \bmod p$, $-b/a \bmod p$. By the same token, the ability to decrypt the message usually also yields the key used for encryption.[30]

[30] A story related to this appears in Casanova's memoirs (1757) (quoted in [Kahn]). He was given an encrypted manuscript by a Madame d'Urfé, which he deciphered. From deciphering, he determined the key used to encipher the manuscript. Later, upon learning this, Madame d'Urfé was incredulous, believing he must have learned the keyword in order to decode the message.

> 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

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 Whitfield Diffie and Martin Hellman invented public key cryptography [DH76]. 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). Such a function is sometimes called a one-way function. However, to be usable for public key cryptography, it must be possible to find the inverse of the function provided you have some extra information. This extra information is referred to as a trapdoor (like a trapdoor that a magician would use to escape from a "sealed" box), resulting in a trapdoor one-way function.

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.[31] However, if the merchant provides a public key, 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.[32]

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 §9.2) computed on its contents. This feature is commonly used to digitally sign email by

---

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.

[31] 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 or 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 that can be quite a lot. This same problem occurs with some cable-modem providers: appropriate software can detect the network traffic of one's neighbors. 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 receiver.

[32] 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.

software such as PGP[33].

Public-key cryptography is very computationally intensive, so typically its use is limited to allow for the secure transmission of a secret key; this secret key is then used to encrypt the rest of the message using a symmetric encryption method such as AES, triple-DES or IDEA.

A common public key system in current use is RSA (named for its inventors: Ronald Rivest, Adi Shamir and Leonard Adelman), which we will look at in detail in §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.

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 (e.g. [NZM], [Ros]) for more information.

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,

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

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 10 minutes on a 600 Mhz pentium computer) 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 $\gcd(138, 126)$,

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

$$\begin{array}{rcrrcrcl}
 & & a & & = & 138 & & \\
 & & & b & = & 126 & & \\
 & & a & -b & = & 12 & = & r_1 \\
b - 10r_1 & = & -10a & +11b & = & 6 & = & r_2 \\
r_1 - 2r_2 & = & 21a & -23b & = & 0 & &
\end{array}$$

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

Theorem 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:*

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

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

Why does the Euclidean Algorithm work? It follows from the observation that if $a = bk + r$, then $\gcd(a, b) = \gcd(b, r)$. For example, $\gcd(138, 126) = \gcd(126, 12) = \gcd(12, 6) = 6$. We write this as a lemma.

Lemma 10.3 *Let b be a positive integer, and a be a nonnegative integer. If*

$$a = bk + r, \qquad \textit{with } k \textit{ and } r \textit{ positive integers}$$

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

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 $a - bk$. Since $r = a - bk$, we have d a divisor of r. So it must divide $\gcd(b, r)$.

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

Thus, $\gcd(b, r) = \gcd(a, b)$. □

Now we are ready to prove that the Euclidean Algorithm (Thm. 10.1) always yields the greatest common divisor.

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 can be written as $k_{n+1}r_n = r_{n-1}$, and so r_n is a divisor of r_{n-1}. Then certainly $r_n = \gcd(r_n, r_{n-1})$. Applying the lemma to the equation above it, we obtain $\gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2})$. Repeating in this way, we obtain $r_n = \gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2}) = \ldots = \gcd(r_1, b) = \gcd(a, b)$. □

Recall that if r is such that $ar \equiv 1 \pmod{n}$, then r is called the multiplicative inverse of a modulo n. 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. 7.1 slightly reworded.

Theorem 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

$$ak \equiv 1 \pmod{n}.$$

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 $\gcd(a, n) = 1$ (by Cor. 10.2).

Now suppose that $\gcd(a, n) = 1$. Then again using Cor. 10.2, there are integers r and s with $ar + ns = 1$. This means $ar - 1$ is divisible by n, so

$$ar \equiv 1 \pmod{n}.$$

But this says that r is the inverse of a modulo n. □

Notation. It is common (and quite convenient) 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 already know, this set is closed under addition and multiplication, and the usual associative, distributive, and commutativity laws hold. This set is an example of an algebraic structure called a ring.

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

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

For example,

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\} \qquad \mathbb{Z}_8^* = \{1, 3, 5, 7\} \qquad \mathbb{Z}_{12}^* = \{1, 5, 7, 11\}$$

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 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 $mn - 1$.

As an example of this, suppose five kids pool all their money and buy a bag of candy. They divide the candy up evenly, and find there are 3 pieces left over. Suddenly, one kid's mom comes out, tells him "no candy before dinner", and hauls him off. The other four kids divide up the candy and there is one piece left over. While they are squabbling over it, a dog runs up and eats the disputed piece.

The Chinese Remainder Theorem tells us that we can always determine how many pieces of candy were in the bag, at least up to multiples of 20. In the above example, there were 13 pieces of candy (or 33, or 53, or ...). If we want to solve this problem with Maple, the command `chrem([3,1],[5,4])` does the trick. We could also use `numtheory[mcombine](5,3,4,1)`.

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:

$$c \equiv ant \equiv a(1 - mk) \equiv a \pmod{m}$$

and

$$c \equiv bkm \equiv b(1 - nt) \equiv b \pmod{n}$$

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

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 four numbers $3, 9, 7, 1$ and then it repeats. What about 4? We have

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

This is also periodic, but notice that we never get back to 1. Similarly, let's look at the powers of 6:

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

This gets stuck at 16. Finally, consider the powers of 11:

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

This sequence is periodic of period 2, and contains 1.

Definition 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 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 $\left[\!\left[a^{k-1}\right]\!\right]_n$. By Thm. 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

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

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 $\left[\!\left[a^k\right]\!\right]_n$ and $\left[\!\left[a^t\right]\!\right]_n$, with $1 \leq t < k$. Since

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

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 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.

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 modulo n.

Definition 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,[34] *or sometimes Euler's totient function.*

[34] Although the function was invented by Euler, the notation $\varphi(n)$ to represent it is actually due to Gauss.

We can compute a few values of $\varphi(n)$ just by counting.

$$\begin{array}{rcrlcl}
\varphi(2) &=& 1 & \text{since} & \mathbb{Z}_2^* &= \{1\} \\
\varphi(3) &=& 2 & \text{since} & \mathbb{Z}_3^* &= \{1,2\} \\
\varphi(4) &=& 2 & \text{since} & \mathbb{Z}_4^* &= \{1,3\} \\
\varphi(5) &=& 4 & \text{since} & \mathbb{Z}_5^* &= \{1,2,3,4\} \\
\varphi(6) &=& 2 & \text{since} & \mathbb{Z}_6^* &= \{1,5\} \\
\varphi(7) &=& 6 & \text{since} & \mathbb{Z}_7^* &= \{1,2,3,4,5,6\} \\
\varphi(8) &=& 4 & \text{since} & \mathbb{Z}_8^* &= \{1,3,5,7\} \\
\varphi(9) &=& 6 & \text{since} & \mathbb{Z}_9^* &= \{1,2,4,5,7,8\} \\
\varphi(10) &=& 4 & \text{since} & \mathbb{Z}_{10}^* &= \{1,3,7,9\} \\
\varphi(11) &=& 10 & \text{since} & \mathbb{Z}_{11}^* &= \{1,2,3,4,5,6,7,8,9,10\} \\
\varphi(12) &=& 4 & \text{since} & \mathbb{Z}_{12}^* &= \{1,5,7,11\}
\end{array}$$

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 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. 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 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 $r = \left[\!\left[t\right]\!\right]_a$. 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 $s = \left[\!\left[t\right]\!\right]_b$, 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 10.11 (Euler, 1750) *Let a and n be relatively prime integers, with $n \geq 2$. Then*

$$\left[\!\left[a^{\varphi(n)}\right]\!\right]_n = 1.$$

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,

$$\left[\!\left[a^{\varphi(n)}\right]\!\right]_n = 1.$$

□

11 The RSA Public key cryptosystem

In 1978, R. Rivest, A. Shamir, and L. Adelman published the description of the RSA public key system[RSA], which we will describe here. As discussed in §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.

11.1 Details of the RSA algorithm

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 number e, called the exponent, which is relatively prime to $\varphi(n)$. We make the numbers (n, e) public— these form the key needed for encoding. We also use the Euclidean algorithm to compute x which satisfies $ex - y\varphi(n) = 1$; that is, so that $\llbracket ex \rrbracket_{\varphi(n)} = 1$. The number x is the part of the key we keep private; our decrypting key is the pair (n, x).

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 §5.3, with $0 \leq \beta < n$. For each block of plaintext, the sender transmits $\llbracket \beta^e \rrbracket_n = \mu$.

To decode the message: For each unit μ of the crypttext received, the recipient computes $\llbracket \mu^x \rrbracket_n$, which is the integer representing of a k-graph of the original plaintext message. We can see that this is so, because

$$\llbracket \mu^x \rrbracket_n = \llbracket \beta^{ex} \rrbracket_n = \llbracket \beta^{1+\varphi(n)y} \rrbracket_n = \llbracket \beta \rrbracket_n \llbracket \beta^{y\varphi(n)} \rrbracket_n = \llbracket \beta \rrbracket_n \left(\llbracket \beta^{\varphi(n)} \rrbracket_n \right)^y = \llbracket \beta \rrbracket_n \cdot 1^y = \beta.$$

This relies on Euler's theorem (Thm. 10.11), so that $\llbracket \beta^{\varphi(n)} \rrbracket_n = 1$. Euler's theorem only guarantees that this will work for $\gcd(\beta, n) = 1$. However, we can easily see that decryption works for all $\beta < n$, whether or not it is relatively prime to n.

Proposition 11.1 *Let $n = pq$, with p and q prime, and let $0 \leq \beta < n$. Then for any integer y, $\llbracket \beta^{1+y\varphi(n)} \rrbracket_n = \beta$.*

Proof. If $\gcd(\beta, n) = 1$, the result follows from Euler's theorem.

Otherwise, since p and q are prime, we have β a multiple of either p or q, but not both (since $\beta < pq$ by assumption). Suppose $\beta \equiv 0 \pmod{p}$, and so

$$\llbracket \beta^{1+y\varphi(n)} \rrbracket_p = 0 = \llbracket \beta \rrbracket_p.$$

Since β is a multiple of p less than pq, and both p and q are prime, we must have $\gcd(\beta, q) = 1$. We can apply Euler's theorem using modulus q to conclude $\llbracket \beta^{\varphi(q)} \rrbracket_q = 1$, so

$$\llbracket \beta^{1+y\varphi(n)} \rrbracket_q = \llbracket \beta \rrbracket_q \left(\llbracket \beta^{\varphi(q)} \rrbracket_q \right)^{y(p-1)} = \llbracket \beta \rrbracket_q.$$

We have two equations modulo p and modulo q, and can apply the Chinese Remainder Theorem (Thm. 10.5) to obtain one modulo pq. Hence

$$\llbracket \beta^{1+y\varphi(n)} \rrbracket_n = \llbracket \beta \rrbracket_n = \beta$$

as desired. In the case where $\llbracket \beta \rrbracket_q = 0$, the argument is the same after exchanging p and q. □

Remarks.

1. The symmetry between e and x means that we can encrypt with either the public or the private key, using the other for decryption. To use RSA works for authentication, the private key (n, x) is used encrypt a message which can be decrypted with the public key (n, e). Only one who possesses the private key could have encrypted such a message, which serves as a proof of identity.

2. If e is small, there is a possibility of having some message elements β for which $\beta^e < n$; for such a β, decrypting without knowledge of $\varphi(n)$ is easy: just compute the e-th root of β, since n is not a consideration. With care, one can either ensure that such β do not occur, or one can encrypt such messages in a slightly different manner.

3. For efficiency reasons, many implementations always use 3 for the encoding exponent e. In general, computing $[\![\beta^e]\!]_n$ when e is of the form $2^k + 1$ is quite efficient. Other common choices for e are 17 and 65537 $(2^{16} + 1)$, which are only slightly less efficient than 3.

4. Instead of using the Euler φ-function, one can use Carmichael's λ-function, which is the largest order of any element of $\mathbb{Z}_n^*$. In the case where n is a product of two primes p and q, we have $\lambda(n) = \operatorname{lcm}(p-1, q-1)$. Since $p-1$ and $q-1$ are always even, the resulting private exponent x will be smaller.

5. There are a number of considerations in the choice of primes p and q that we will mostly ignore. For example, p and q should not be too close together, and both $p-1$ and $q-1$ should both have at least one large prime factor.

11.2 Implementing RSA in Maple

Now that we have covered how the RSA system works in theory, it is fairly straightforward to implement it.

Implementing the basics of RSA

First, we write a Maple procedure which, given a size for n (our public base), will choose n as the product of two large primes p and q, as well as an exponent e which is relatively prime to $\varphi(n)$, and compute the decrypting exponent x.

For real security, n should be at least 200 decimal digits, meaning that the primes p and q will be about 100 digits each. Such large numbers are not easily factored (which is how we get our security), so how do we expect to be able to find 100-digit prime numbers? It turns out that there are a number of very efficient ways to test whether a number is prime without actually factoring it. Maple's `isprime` function uses such methods. So all we really need to do is generate a random number of around 100 digits, then use `nextprime` to get the next prime bigger than that number. We should use `randomize()` to ensure that we get different numbers each time (See the discussion of pseudo-random numbers in §5.2, page **4**:17).

In `RSAsetup` below, the approximate number of digits for n is given, and p is chosen as a prime with one less than half that many digits, while q has one more than half the number of digits of n. This means n must have at least 5 digits (or else p would have to be 2, 3, 5 or 7). After calculating n and $\varphi(n)$, a random number is chosen for the public exponent e. If this number is not relatively prime to $\varphi(n)$, another random choice is taken until such a number is found. Finally, x is computed as the inverse of e modulo $\varphi(n)$. The result of the function is the public key (n, e) and the private key (n, x).

```
> randomize():

  RSAsetup:=proc(numdigits::posint)
    local p,q,phi,e,n,  x, pm, qm;

    if (numdigits <= 4) then
       error("Too few digits for this to work");
    fi;

    pm:=floor((numdigits-1)/2);
    qm:=numdigits-pm;
    p:=nextprime(rand(10^(pm-1)..10^(pm))());
    q:=nextprime(rand(10^(qm-1)..10^(qm))());

    n:=p*q;
    phi:=(p-1)*(q-1);

    e:=rand(3..phi/2)();
    while(gcd(e,phi) <> 1) do
      e:=rand(3..phi/2)();
    od;
    x := modp(1/e, phi);
    return([[n,e],[n,x]]);
  end:
```

We should remark that in certain cases, both p and q might be chosen at the extreme ends of their ranges, giving n with one fewer or one more digit than requested. If this were important, it could be repaired with a slightly more complicated program.

Once the keys are chosen, writing the heart of the RSA encoding is quite easy. We just need a function that eats a number β and a key (n, e), and computes $\left[\!\left[\beta^e\right]\!\right]_n$.

```
> RSAEncodeNum := proc(num::nonnegint, key::list(posint))
    return( modp( num &^ key[2], key[1] ));
  end:
```

As a brief example, we'll generate a random pair of keys with a 5-digit modulus, and then encrypt and decrypt the number 47.

```
> keys:=RSAsetup(5):
  public := keys[1];
  private:= keys[2];
```

$$public := [10793, 2371]$$

$$private := [10793, 31]$$

```
> RSAEncodeNum(47,public);
  RSAEncodeNum(%,private);
```

$$3991$$

$$47$$

Of course, we want to encrypt more than a single number, but this is the heart of the whole thing.

Making it Useful

In order to encrypt a string of text, we have to convert our text to a list of numbers. It is probably best to use k-graphs to represent our text, as in §5.3, where k is the largest integer power of the size of our alphabet that is smaller than n. While it will work to use a single character alphabet, this can compromise security significantly— see the remarks at the end of §11.1.

We can use `StringToKgraph` and `KgraphToString` from §5.3 to represent our strings as k-graphs. In order to determine the appropriate choice of k, we find the largest integer k so that

$$\texttt{Alen}^k < n$$

where `Alen` is the length of our alphabet.

Once we have our plaintext represented as a list of integers, we merely `map` the function `RSAEncodeNum` onto that list.

```
> Alphabet := cat("\n\t",Select(IsPrintable,convert([seq(i,i=1..255)],bytes))):
  Alen := length(Alphabet);
  k:=floor(log[Alen](public[1]));
```

$$Alen := 97$$

$$k := 2$$

```
> text := "Once upon a midnight dreary, while I pondered, weak and weary":
```

```
> StringToKgraph(text,k);
```

[7809, 6956, 8441, 7939, 274, 261, 7354, 7830, 7156, 8416, 6792, 6971, 8215, 1449, 8635, 7349, 6965, 4173, 7956, 7841, 6957, 6971, 1428, 8635, 6570, 271, 7827, 264, 6976, 8215, 91]

```
> crypt:=map(RSAEncodeNum,%,public);
```

crypt := [23972, 14519, 16728, 21226, 15537, 16168, 23547, 21311, 657, 15151, 20048, 7796, 961, 8740, 4392, 5615, 18965, 18341, 18628, 14070, 9286, 7796, 23471, 4392, 17418, 12708, 3247, 11788, 23561, 961, 9606]

We can reverse the process, using the private key, to decrypt.

```
> KgraphToString(
      map(RSAEncodeNum,crypt,private),
      k);
```

"Once upon a midnight dreary, while I pondered, weak and weary"

It is important to note that if we want to represent the result of the encryption as a string instead of a list of numbers, we must use $(k+1)$-graphs rather than k-graphs. This is because $\texttt{Alen}^k < n$, so there will often be β so that $[\![\beta^e]\!]_n > \texttt{Alen}^k$. In the example above, note that $97^2 = 9409$, and there are several elements of the list `crypt` larger than 9409.

We can put it all together in two simple procedures.

```
> RSAEncodeString := proc(text::string, key::list(posint))
    local Alen, k;
    global Alphabet;

    Alen := length(Alphabet);
    k:=floor(log[Alen](key[1]));
    KgraphToString(
        map(RSAEncodeNum,
            StringToKgraph(text, k),
            key),
        k+1);
end:

RSADecodeString := proc(text::string, key::list(posint))
    local Alen, k;
    global Alphabet;

    Alen := length(Alphabet);
    k:=floor(log[Alen](key[1]));
    KgraphToString(
        map(RSAEncodeNum,
            StringToKgraph(text, k+1),
```

```
            key),
        k);
end:
```

Now we try it out, with a 150-digit choice of n. Note that this means we'll be using 75-graphs, since $97^{75} < 10^{150} < 97^{76}$.

```
> keys:=RSAsetup(150):
  public:=keys[1]; private:=keys[2];
```

$$public := [582757988242592241527296499231639188385179508302715855017\backslash$$
$$144484847421609272778490767597650118443597311849831459240406 7\backslash$$
$$384025550370142971709602674385 21, 184998221862523554916292823 87\backslash$$
$$306972419512598470315009879478782685649065787593705898397921 6\backslash$$
$$656484126931022139703964622198669253149707372291289384981 3]$$

$$private := [582757988242592241527296499231639188385179508302715855017\backslash$$
$$144484847421609272778490767597650118443597311849831459240406 7\backslash$$
$$384025550370142971709602674385 21, 331162906961939060283538890 77\backslash$$
$$810953835226409028594751957894266894819508519136812230877143 08\backslash$$
$$970811249627796735482001360890541195019272423373550779335 65]$$

```
> crypt:=RSAEncodeString(text,public);
```

$$crypt := \text{“?!2<*h\&l9![</*(=X5'oMT*Mnc /T\%j*r\~{}\“_=?b00‘UP_\backslash$$
$$\text{gQ\#O¡oyVQdx3r—. N=s)¡!oidl?+\$”}$$

```
> RSADecodeString(crypt,private);
```

“Once upon a midnight dreary, while I pondered, weak and weary”

12 RSA encoding a file

Now we will write a procedure that uses RSA to encrypt a file, rather than a string. Of course, we could just read in the string and use `RSAEncryptString`, but in some cases, it makes sense to use a different alphabet for the crypttext. For example, we might want to accept all possible ASCII codes 0-255 as plaintext input, but restrict our encrypted output to use a limited collection of characters (so that it can be printed or emailed without transcription problems, say). For example, in the example below, we use the character set of RFC 2045[35], which specifies how to encode base 64 MIME attachments for email.

[35] http://www.faqs.org/rfcs/rfc2045.html

Here is our procedure which reads the contents of a file, encrypts it with RSA, and writes out the result to another file.

```
> Alphabet:=cat(Select(IsAlphaNumeric,convert([seq(i,i=1..255)],bytes)),"+/");
> RSAEncodeFile:=proc(plainfile::string, cryptfile::string, key::list(posint))
    local  k, pf, cf, chunk, chunklen, codenum, crypt, Alen;
    global Alphabet;

    Alen    := length(Alphabet);
    chunklen:= ceil(log[Alen](key[1]));
    k       := floor(log[256](key[1]));

    try
      pf      := fopen(plainfile, READ, BINARY);
      cf      := fopen(cryptfile, WRITE, TEXT);
      chunk   := readbytes(pf, k);
      while (chunk <> 0) do
          codenum := map(RSAEncodeNum,
                         convert( chunk, base, 256, 256^k),
                         key);
          crypt:=KgraphToString(codenum, chunklen);
          writeline(cf, crypt);
          chunk := readbytes(pf, k);
      od;

    finally
       fclose(cryptfile);
       fclose(plainfile);
    end try;
    RETURN();
  end:
> RSADecodeFile:=proc(cryptfile::string, plainfile::string, key::list(posint))
    local  k, pf, cf, chunk, chunklen, plain, crypt, Alen;
    global Alphabet;

    Alen    := length(Alphabet);
    chunklen:= ceil(log[Alen](key[1]));
    k:=        floor(log[256](key[1]));

    try
      pf      := fopen(plainfile, WRITE, BINARY);
      cf      := fopen(cryptfile, READ, TEXT);
      chunk   := readline(cf);
      while (chunk <> 0) do
          plain:= convert(
                      map(RSAEncodeNum,
                          StringToKgraph(chunk, chunklen),
                          key),
                      base, 256^k, 256);
          writebytes(pf, plain);
```

```
            chunk := readline(cf);
        od;

      finally
         fclose(cryptfile);
         fclose(plainfile);
      end try;

      RETURN();
    end:
```

References

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory* **IT-22** no.6 (1976), p. 644–654.

[Hil1] L. Hill, "Cryptology in an Algebraic Alphabet", *American Mathematical Monthly* **36** (1929), p. 306–320.

[Hil2] L. Hill, "Concerning the Linear Transformation Apparatus in Cryptography", *American Mathematical Monthly* **38** (1931), p. 135–154.

[HP] J. H. Humphreys & M. Y. Prest, *Numbers, Groups, and Codes*. Cambridge University Press, Cambridge. 1989.

[Kahn] D. Kahn, *The Codebreakers; The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Charles Scribner's Sons, New York. 1996.

[Kob] N. Koblitz, *A Course in Number Theory and Cryptography* (Graduate Texts in Mathematics, No 114). Springer-Verlag, New York. 1994.

[NZM] I. Niven, H. Zuckerman, H. Montgomery, *An Introduction to the Theory of Numbers*, John Wiley & Sons, New York. 1991.

[RSA] R. L. Rivest, A. Shamir and L. Adelman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM* **21** (1978), p. 120–126.

[Ros] K. Rosen, *Elementary Number Theory and its Applications*, 4th edition. Addison-Wesley, Boston. 2000.

[Sch] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd edition. John Wiley & Sons, New York. 1995.

Chapter 5

A turtle in a fractal garden

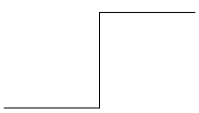
1 Turtle Graphics

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`);
```



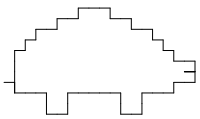
We shall begin assuming the turtle knows the following commands:

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.

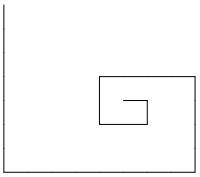
We will point out some more commands later, although this set is sufficient to do quite a lot.[2] For example, we can ask our turtle to make a self portrait:

```
> 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‘);
```

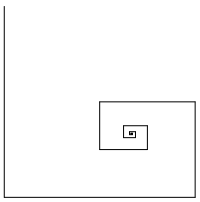


This last can be represented more succinctly if we add a pair of new commands, namely

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)));
```

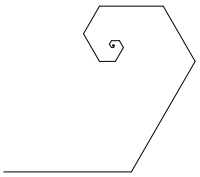


You may have noticed that the command `TurtleCmd` always resizes the resulting graphic to be of constant width. While this may seem a minor point, it is not. In any single string of turtle commands, “FF” will always be twice as long as “F”; however, `TurtleCmd(‘F‘)` and `TurtleCmd(‘FFFFFFFFFF‘)` produce 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.

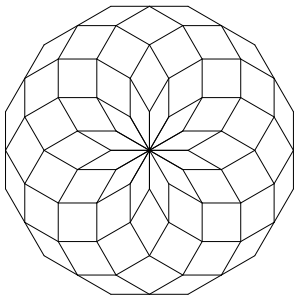
[2] In fact, this set of commands is already redundant. We could replace L with RRR and B with RRFLL.

```
> ResetTurtle(); SetTurtleAngle(60);
  TurtleCmd(cat(seq(`FRFRG`,i=1..20)));
```



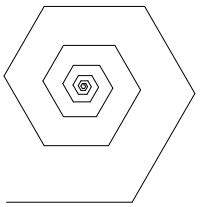
Here is a somewhat prettier example. Can you figure out what is happening?

```
> ResetTurtle(); SetTurtleAngle(30);
  TurtleCmd(cat(seq(cat(`L`,
                        seq(`FL`,i=1..12)),
                    j=1..12)));
```



`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)));
```



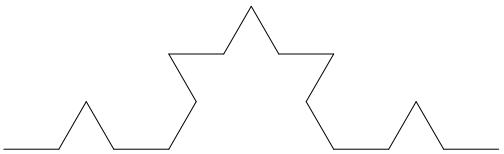
2 A fractal

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.

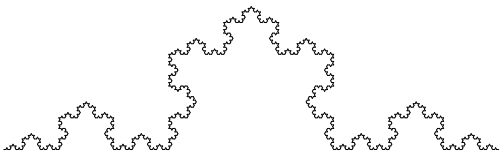
```
> SetTurtleAngle(60);
  plots[display](array([TurtleCmd(‘F‘), TurtleCmd(‘FLFRRFLF‘)]));
```

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

$$d(\mathcal{K}_N, \mathcal{K}_n) < \sum_{j=N}^{n} 1/3^j < \frac{1}{2 \cdot 3^{N-1}}.$$

This means that the sets $\mathcal{K}_n$ form a Cauchy sequence, and so there is a well-defined limit set $\mathcal{K}_\infty$.[3]

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.[4] 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.

[3] In order to completely finish this argument, we need to know that the collection of all compact sets forms a complete metric space if we use the Hausdorff distance as our metric. Without this fact, we don't know that the limiting object $\mathcal{K}_\infty$ is also a compact set in the plane. Although this isn't difficult to prove (and is nearly self-evident), we will skip over this detail.

[4] 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 one of the most well-known examples 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 5).

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.

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 $\mathcal{K}_\infty$.

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

$$K_n = \begin{cases} K_{n-1}\mathrm{L}K_{n-1}\mathrm{RR}K_{n-1}\mathrm{L}K_{n-1} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} .$$

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.[5] To see this, let's implement the factorial function in both ways:

[5]Most programming languages these days do allow recursive function calls, although this was not the case until 1985 or so.

```
> 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.[6] However, there is a price to pay: a recursively defined function has a bit more overhead than a non-recursive one. However, the savings in simplicity often overwhelms the minor loss in speed.[7]

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!

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 §3, namely that

$$K_n = \begin{cases} K_{n-1}\mathrm{L}K_{n-1}\mathrm{RR}K_{n-1}\mathrm{L}K_{n-1} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} .$$

[6] 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.

[7] 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.

```
> 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

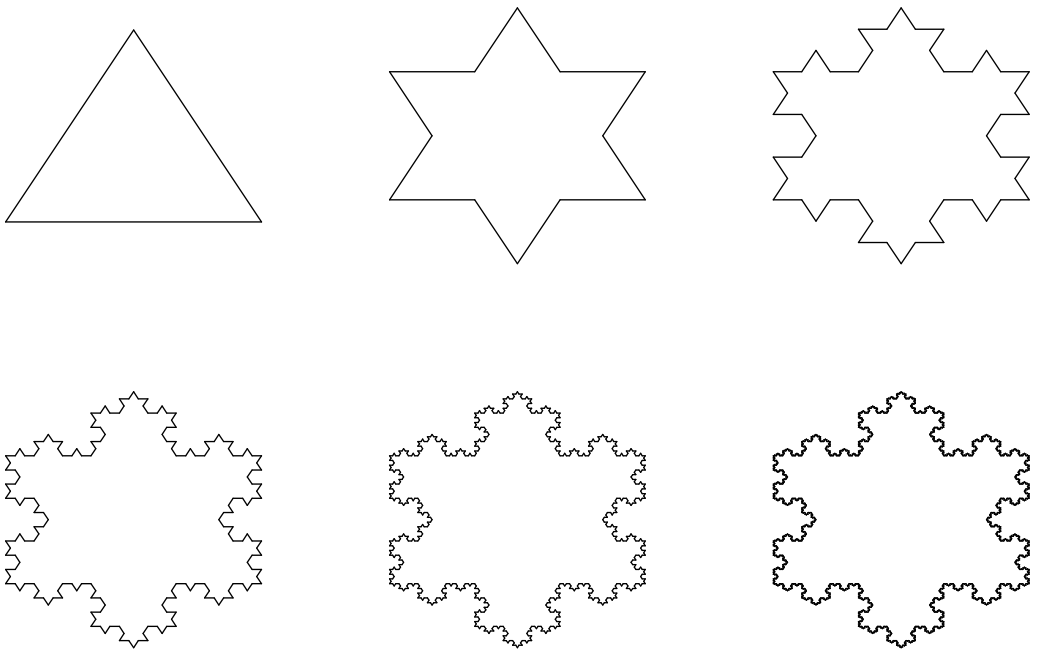
If you think about it for a second, you might notice that what we have just done is to write a little program (`Koch`) whose output is a program in another language (the string of turtle commands which describe K_n). While this may seem odd at first, doing this sort of thing is not at all uncommon.

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:

```
> 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)]]));
```



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.

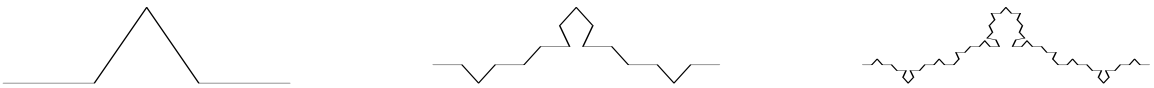
```
> ResetTurtle(); SetTurtleAngle(10);
  TurtleCmd(Koch(6));
```

But an angle of 80° gives a curve which wiggles wildly and almost seems to take up area.

```
> ResetTurtle(); SetTurtleAngle(80);
  TurtleCmd(Koch(6));
```

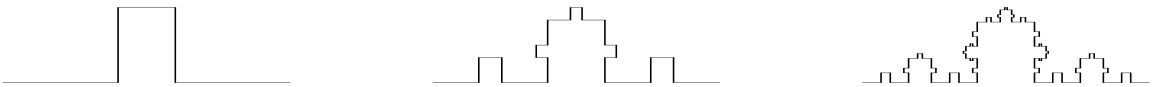
We shall make this observation more precise in section 5. You might try some additional experiments on your own, to gain intuition about how the limit curve depends on the parameters. For example, what would happen if you make the angle be 90°? Do you find the result surprising?

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`?



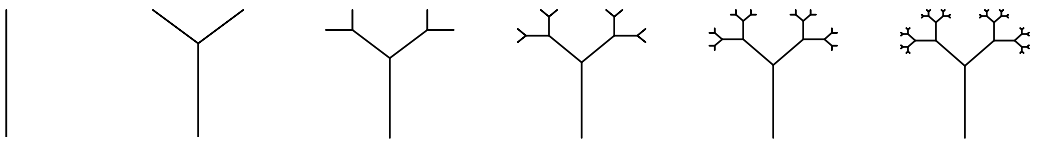
4 Making a tree

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.[8] 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 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,[9] 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 proper direction when we finish the branch, or things won't work out at all.

[8] As in §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.

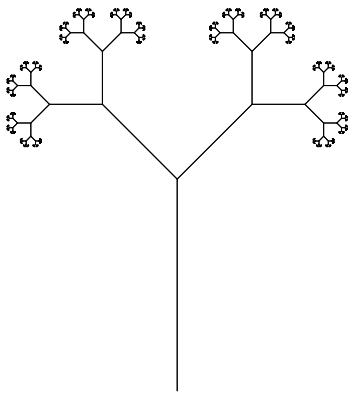
[9] The file turtle.txt does, in fact, implement these commands as D and U— see §8 for more details.

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

$$T_n = \begin{cases} \text{FSL}T_{n-1}\text{RR}K_{n-1}\text{LGB} & \text{if } n > 0 \\ \text{FB} & \text{if } n = 0 \end{cases} .$$

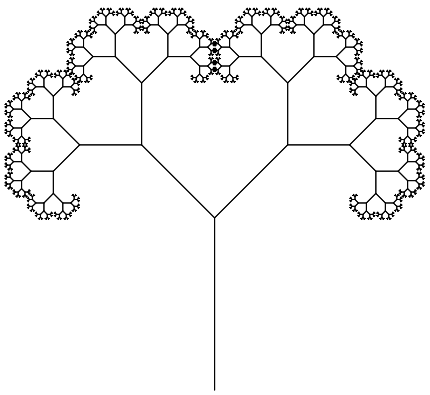
Using this, it is now easy to write the generating procedure.

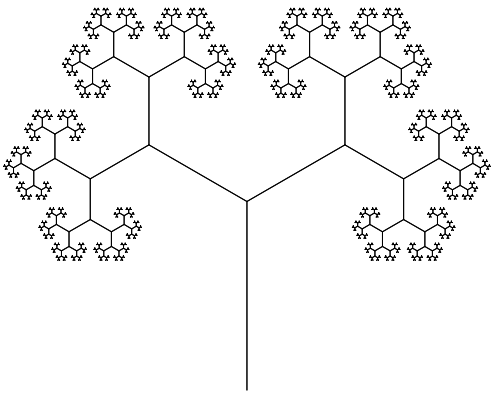
```
> Tree:= proc(n::nonneg)
    options remember;
    if (n=0) then
       `FB`;
    else
       cat(`FSL`, Tree(n-1), `RR`, Tree(n-1), `LGB`);
    fi;
  end:

> TurtleCmd(Tree(10));
```



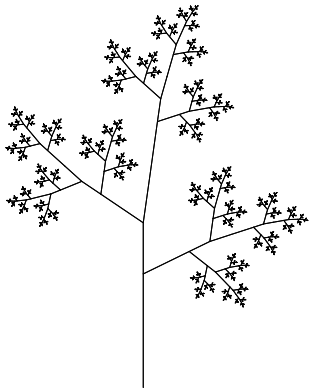
You might want to experiment with varying the angle and the scale factor, and see how the tree changes. For example, if we use `SetTurtleScale(.6)`, the branches of $\mathcal{T}_8$ just touch, and they overlap in $\mathcal{T}_n$ if $n > 8$. However, if we change the angle to 60°, the branches stay away from each other.

```
> 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.



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.[10] 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 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

[10] 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.

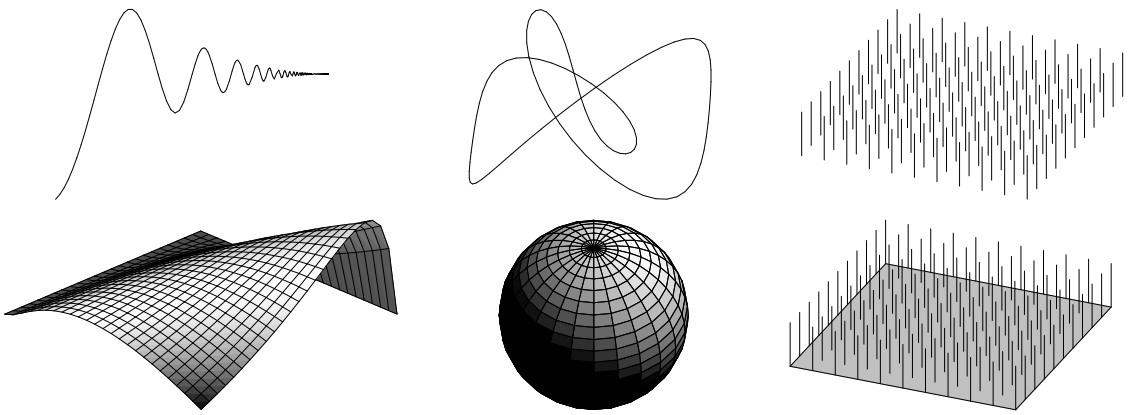


Figure 1: Some one- and two-dimensional sets (the sphere is hollow, not solid).

plane $\mathbb{R}^2$, and in such a small piece, you only need two coordinates to describe the location of a point.[11]

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.

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.

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.

[11] 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 $2n$.

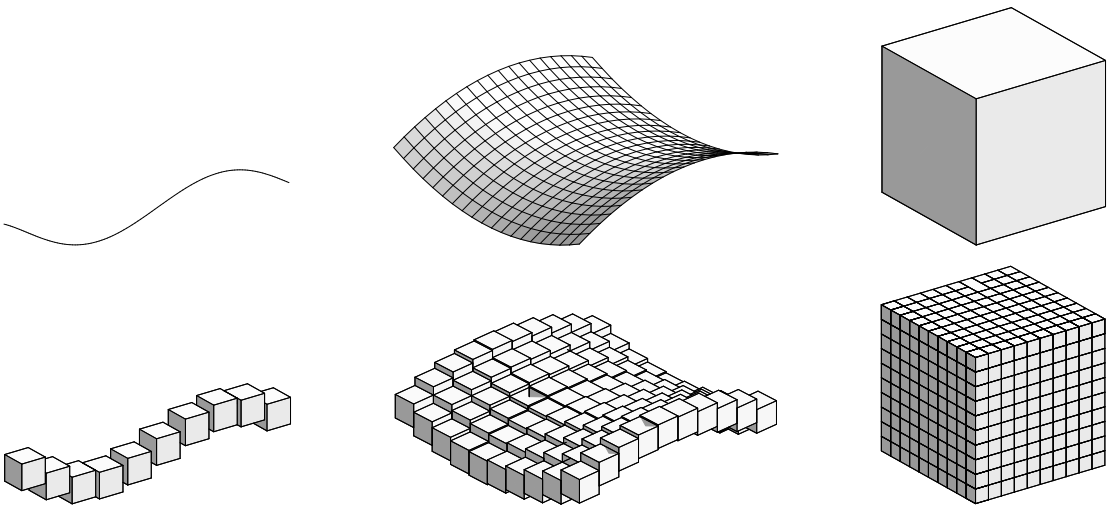


Figure 2: Covering a curve, a surface, and a solid cube with cubes of diameter ϵ.

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

$$d = \lim_{\epsilon \to 0} \frac{\ln k - \ln N_\epsilon(\mathcal{S})}{\ln \epsilon} = -\lim_{\epsilon \to 0} \frac{\ln N_\epsilon(\mathcal{S})}{\ln \epsilon}.$$

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.[12] We will not encounter such examples here, however. Since the box-counting dimension is so often used to calculate the dimensions of fractal

[12]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.

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 $\ln N/\ln r$.

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 the discussion at the beginning of §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,

$$\mathcal{Q} = \left\{ \frac{p}{q} \,\middle|\, p, q \text{ are relatively prime integers with } p \leq q \right\}.$$

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, $1/4 = .25$, $1/11 = .010\overline{10}\ldots$, 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[13] .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.

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.

[13] $\mathcal{D}$ contains .4 because it can be written as $0.399\overline{9}\ldots$, which contains no 4s or 5s.

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 §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. 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 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 §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, and the base used by just about every living human[14]for everyday numbers). For example, we could express this[15] 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.[16]

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.[17] 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.

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

$$\dim_{\square}(\mathcal{C}) = \lim_{n\to\infty} \frac{\ln 2^n}{\ln 3^n} = \frac{\ln 2}{\ln 3} \approx 0.63093.$$

[14]Some Native American peoples used base 20, and the Babylonians were very fond of base 12 and base 60 (consider how we measure time).

[15]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?

[16]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 (bits), 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 don't break up nicely into groups of 3 bits.

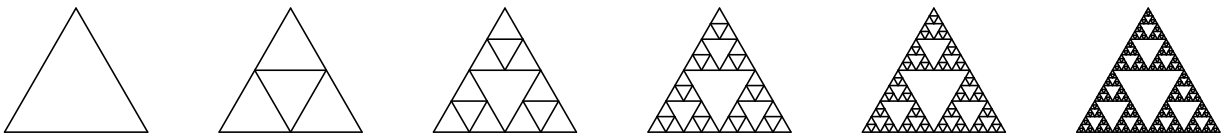
[17]As in decimal expansions, some points have two representations. Thus, the point $1/3$ can be written either as $.1_3$ or as $.0222\overline{2}\ldots$. As in the construction of $\mathcal{D}$ in §5.2, we keep such points in our set.

You may have noticed the strong similarity in the construction of the Koch curve $\mathcal{K}$ of §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.

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[18] 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.

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.

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

[18] We will give another solution which is closer to our original description in §9.

of the gasket, and then there is the final RBLLB which makes the outer "hat" (see figure 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

$$S_n = s_n\, h \qquad s_n = \begin{cases} \mathrm{S}s_{n-1}\mathrm{RBL}s_{n-1}\mathrm{LBR}s_{n-1}\mathrm{G} & \text{if } n > 0 \\ \mathrm{F} & \text{if } n = 0 \end{cases} \qquad h = \mathrm{RBLLB}.$$

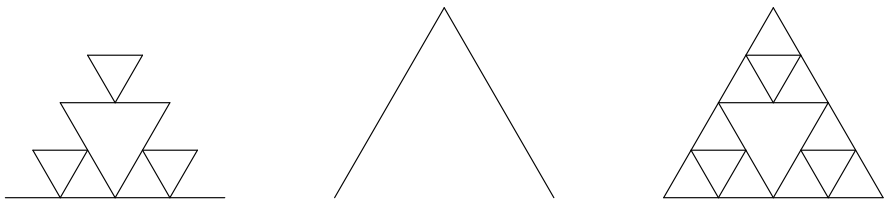


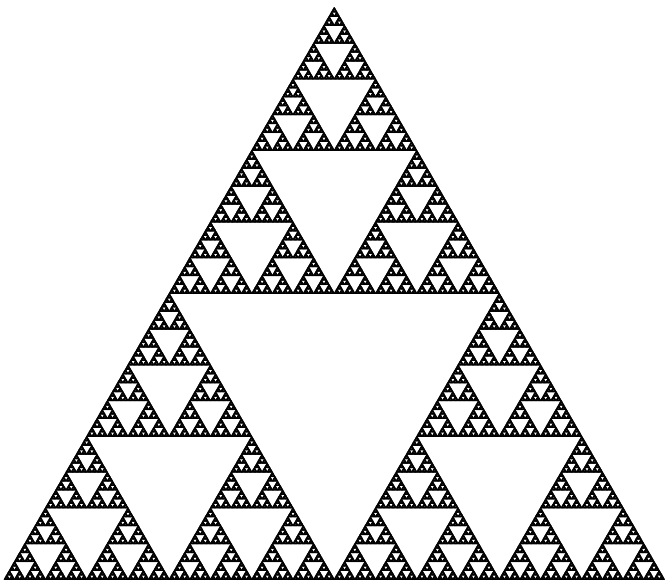
Figure 4: The sets produced by the commands for s_2, h, and S_2 in the construction of the Sierpinski gasket. Here, the set for s_1 is drawn thicker within that of s_2.

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.

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. At the highest or outermost layer is the `TurtleCmd` procedure,[19] which does all the work of interpreting the turtle "language". Much of this procedure is implemented in other "lower level" procedures, such as `Forward` and `Left`, which exist to make the programming of the higher-level routines easier. And these procedures may be implemented using even other lower-level procedures.

Why was it programmed this way? In part, to make it easier to understand and to change. This program is modular, and to write the top-level program we don't need to know all the ins and outs of the lower levels, just what is available and what it does. This sort of thing is very common and useful in programming, and in everyday life as well. For example, to go to the store you might drive a car. But you don't need to know how to *build* a car to drive it, nor do you need to know how to build a road. You just need to know what it does and how to use it. And the man in the body shop who repairs your windshield (after a rock from the construction site you passed on the way to the store cracks it) doesn't need to know how to manufacture a windshield, he only needs to know where to get the right one for your car and how to install it properly.

[19] The command `AnimateTurtleCmd`, which we haven't discussed, is very similar to this, but shows the turtle's path as the curve is traversed.

command	meaning
`TurtleCmd(cmd)`	run the turtle program given in `cmd` and plot the result
`AnimateTurtleCmd(cmd)`	like `TurtleCmd`, but animates the path
`SetTurtleAngle(n)`	set the angle the turtle turns at each `L` or `R` to `n` degrees
`GetTurtleAngle()`	retrieve the current setting of the TurtleAngle
`SetTurtleScale(s)`	Set the factor the length scales by when an `S` or `G` is done
`GetTurtleScale()`	return the current scale factor
`SetInitialTurtleHeading(h)`	set the initial heading of the turtle to `h`
`GetInitialTurtleHeading()`	report the value of the turtle's initial heading
`SetTurtleHeading(h)`	Set the current heading of the turtle to `h`
`GetTurtleHeading()`	return the current heading of the turtle
`SetTurtleStepsize(l)`	Set the current length of a `F` command to be `l`
`GetTurtleStepsize()`	return the current length of a `F` command
`DoCommand(c)`	perform the command indicated by `c`
`DoUserCommand(c)`	"hook" to handle extra turtle commands
`Forward()`	move the turtle forward one unit in its current heading
`Back()`	move the turtle back by one unit
`Left()`	change the heading of the turtle TurtleAngle degrees left
`Right()`	change the heading of the turtle TurtleAngle degrees Right
`Shrink()`	multiply the current length unit by the scale factor
`Grow()`	divide the current length unit by the scale factor
`PenUp`	Stop remembering the turtle's path ("stop drawing")
`PenDown`	Start remembering the turtle's path again
`SetPenColor(r,g,b)`	Set the path to the RGB color given
`PushState()`	remember the state of the turtle
`PopState()`	restore the state of the turtle (except the position history)
`ShowPath()`	Plot the turtle's path
`ClearPage()`	clear the turtle's page, and set the position to $(0, 0)$
`ResetTurtle()`	Clear the page, and set all variables to default.

Table 5.1: Most of the commands implemented by the turtle package.

Without going into the deepest details, let's pop off the shell of `TurtleCmd` and see how it works.

```
> TurtleCmd := proc(cmd::name)
    local c, i;
    ClearPage();
    for i from 1 to length(cmd) do
       c:=substring(cmd,i);
       DoCommand(c);
    od;
    ShowPath();
  end:
```

Not much to it, is there? The procedure uses the command `ClearPage` to wipe the slate clean, setting the turtle's position to $(0, 0)$, aiming in the initial direction, and so on. Then, for each character in its command string, it calls `DoCommand` to perform that command, and finally uses `ShowPath` to actually render the plot. We won't discuss exactly how `ClearPage` and `ShowPath` now (feel free to look at them yourself, of course). But here is `DoCommand`:

```
> DoCommand:=proc(c)
     if   (c=`F`) then
        Forward();
     elif (c=`B`) then
        Back();
     elif (c=`L`) then
        Left();
     elif (c=`R`) then
        Right();
     elif (c=`S`) then
        Shrink();
     elif (c=`G`) then
        Grow();
     elif (c=`U`) then
        PenUp();
     elif (c=`D`) then
        PenDown();
     elif (c=` `) then
        NULL;
     else
        if (not DoUserCommand(c)) then
           WARNING(sprintf("Unknown turtle command %c encountered",c));
        fi;
     fi;
     NULL;
  end:

  DoUserCommand:=c->false:
```

All this does is look at its argument (a single character), and call the corresponding lower-level routine. For example, `DoCommand(`F`)` just calls `Forward()`, which advances the internal position of the turtle ahead one step. Note that there are two commands we haven't used yet, namely `U` and `D`, which raise and lower the pen that the turtle holds. For example, the turtle

command `FUFDF` produces two line segments separated by a gap. The line with `DoUserCommand` will be discussed in the next section; it is a way to allow the user of the turtle package to add his or her own commands.

In the previous paragraph, we refered to the "internal position of the turtle"— what did we mean by this? We think of the turtle as having a current position, direction, step size, and so on. Each of these are stored in variables internal to the turtle package and updated by commands such as `Back`, `Shrink`, or `SetTurtleAngle`. A history of the positions of the turtle since the last `ClearPage` is kept, and when `ShowPage` is called, they are formatted into an appropriate maple `Plot` command.

9 Extending the turtle's commands

Armed with this little bit of information about the inner workings of the turtle, how might we exploit it? We can extend the turtle language. You may recall that the way we made the Sierpinski gasket in §7 did not correspond exactly to our initial description, but relied on finding a fractal curve that traced out most of the gasket. We'll now describe another method, which will be closer to our original description.

Recall that the gasket consists of three copies of itself, two along the base, and one at the top. Consequently, we can construct an approximate gasket $\mathcal{S}_n$ out of three copies of the approximate gasket $\mathcal{S}_{n-1}$. The first approximation $\mathcal{S}_0$ is just an equilateral triangle.

First, lets write a procedure `EqTriangle`, which causes the turtle to trace out an equilateral triangle with a horizontal base.

```
> EqTriangle := proc()
     PushState();
     PenDown();
     SetTurtleHeading(0);
     SetTurtleAngle(120);

     Forward();
     Left();
     Forward();
     Left();
     Forward();

     PopState();
  end:
```

The middle of this procedure is obvious: we send the turtle along the path `FLFLF`. But before that, we save its current state with `PushState`, then make sure that the heading and angle are correct for the triangle (since perhaps they were different), and ensure that the pen is down. After we are finished with the triangle, we put the turtle back the way it was with `PopState`. Note that we have taken care to ensure that the triangle always has the same orientation, no matter what direction the turtle is heading. The procedure also leaves the position and heading

of the turtle unchanged.

Now, we will plug in an additional procedure to handle the new turtle command `T`, which draws a triangle. This is done by replacing `DoUserCommand` with our own version. This procedure is expected to return `true` if it recognized and handled the character, and `false` if not.

```
> DoUserCommand:=proc(c)
    if (c=`T`) then
       EqTriangle();
    else
       RETURN(false);
    fi;
    true;
  end:
```

With this new command in hand, we can produce a new set of commands to produce a Sierpinski gasket, closer to our original description. Assuming that the angle is 60°and the scaling factor is .5, then

$$S_n = \begin{cases} \text{S } S_{n-1} \text{ F} S_{n-1} \text{B LFR} S_{n-1} \text{LBR G} & \text{if } n > 0 \\ \text{T} & \text{if } n = 0 \end{cases}.$$

In words, this means to make S_n, we include a half-size S_{n-1}, advance forward along its base and put in another copy. The we back up to the starting point, go up the left side, put in the third copy, and return. It is important to take care that the turtle is pointing in the proper direction when we finish.

Here is the corresponding maple procedure:

```
> Sierp2 := proc(n::nonnegint)
    if (n=0) then
      `T`;
    else
       cat(`S`, Sierp2(n-1), `F`, Sierp(n-1),
           `BLFR`, Sierp2(n-1), `LBRG`);
    fi;
  end:
```