1.5 Stability theory

In the study of natural phenomena it is most often true that a small change in conditions will produce just a small change in the state of the system being studied. If, for example, a very slight increase in atmospheric pollution could produce dramatically large changes in populations of flora and fauna, or if tiny variations in the period of the earth’s rotation produced huge changes in climatic conditions, the world would be a very different place to live in, or to try to live in. In brief, we may say that most aspects of nature are stable.

When physical scientists attempt to understand some facet of nature, they often will make a mathematical model. This model will usually not faithfully reproduce all of the structure of the original phenomenon, but one hopes that the important features of the system will be preserved in the model, so that predictions will be possible. One of the most important features to preserve is that of stability.

For instance, the example of atmospheric pollution and its effect on living things referred to above is important and very complex. Therefore considerable effort has gone into the construction of mathematical models that will allow computer studies of the effects of atmospheric changes. One of the first tests to which such a model should be subjected is that of stability: does it faithfully reproduce the observed fact that small changes produce small changes? What is true in nature need not be true in a man-made model that is a simplification or idealization of the real world.

Now suppose that we have gotten ourselves over this hurdle, and we have constructed a model that is indeed stable. The next step might be to go to the computer and do calculations from the model, and to use these calculations for predicting the effects of various proposed actions. Unfortunately, yet another layer of approximation is usually introduced at this stage, because the model, even though it is a simplification of the real world, is still too complicated to solve exactly on a computer.

For instance, many models use differential equations. Models of the weather, of the motion of fluids, of the movement of astronomical objects, of spacecraft, of population growth, of predator-prey relationships, of electric circuit transients, and so forth, all involve differential equations. Digital computers solve differential equations by approximating them by difference equations, and then solving the difference equations. Even though the differential equation that represents our model is indeed stable, it may be that the difference equation that we use on the computer is no longer stable, and that small changes in initial data on the computer, or small roundoff errors, will produce not small but very large changes in the computed solution.

An important job of the numerical analyst is to make sure that this does not happen, and we will find that this theme of stability recurs throughout our study of computer approximations.

As an example of instability in differential equations, suppose that some model of a system led us to the equation

\[ y'' - y' - 2y = 0 \]  \hspace{1cm} (1.5.1)

together with the initial data

\[ y(0) = 1; \quad y'(0) = -1. \]  \hspace{1cm} (1.5.2)

We are thinking of the independent variable \( t \) as the time, and so we will be interested in the solution as \( t \) becomes large and positive.

The general solution of (1.5.1) is \( y(t) = c_1 e^{-t} + c_2 e^{2t} \). The initial conditions tell us that \( c_1 = 1 \) and \( c_2 = 0 \), hence the solution of our problem is \( y(t) = e^{-t} \), and it represents a function that decays rapidly to zero with increasing \( t \). In fact, when \( t = 10 \), the solution has the value 0.000045.

Now let’s change the initial data (1.5.2) just a bit, by asking for a solution with \( y'(0) = -0.999 \). It’s easy to check that the solution is now

\[ y(t) = (0.999666 \ldots) e^{-t} + (0.000333 \ldots) e^{2t} \]  \hspace{1cm} (1.5.3)
instead of just \( y(t) = e^{-t} \). If we want the value of the solution at \( t = 10 \), we would find that it has changed from 0.000045 to about 7.34.

At \( t = 20 \) the change is even more impressive, from 0.00000002 to 161,720+, just from changing the initial value of \( y' \) from \(-1\) to \(-0.999\). Let’s hope that there are no phenomena in nature that behave in this way, or our lives hang by a slender thread indeed.

Now exactly what is the reason for the observed instability of the equation (1.5.1)? The general solution of the equation contains a falling exponential term \( c_1 e^{-t} \), and a rising exponential term \( c_2 e^{2t} \). By prescribing the initial data (1.5.2) we suppressed the growing term, and picked out only the decreasing one. A small change in the initial data, however, results in the presence of both terms in the solution.

Now it’s time for a formal

**Definition:** A differential equation is said to be **stable** if for every set of initial data (at \( t = 0 \)) the solution of the differential equation remains bounded as \( t \) approaches infinity.

A differential equation is called **strongly stable** if, for every set of initial data (at \( t = 0 \)) the solution not only remains bounded, but approaches zero as \( t \) approaches infinity.

What makes the equation (1.5.1) unstable, then, is the presence of a rising exponential in its general solution. In other words, if we have a differential equation whose general solution contains a term \( e^{\alpha t} \) in which \( \alpha \) is positive, that equation is unstable.

Let’s restrict attention now to linear differential equations with constant coefficients. We know from section 1.3 that the general solution of such an equation is a sum of terms of the form

\[
\text{(polynomial in } t) e^{\alpha t}.
\]

Under what circumstances does such a term remain bounded as \( t \) becomes large and positive?

Certainly if \( \alpha \) is negative then the term stays bounded. Likewise, if \( \alpha \) is a complex number and its real part is negative, then the term remains bounded. If \( \alpha \) has positive real part the term is unbounded.

This takes care of all of the possibilities except the case where \( \alpha \) is zero, or more generally, the complex number \( \alpha \) has zero real part (is purely imaginary). In that case the question of whether \( (\text{polynomial in } t) e^{\alpha t} \) remains bounded depend on whether the “polynomial in \( t \)” is of degree zero (a constant polynomial) or of higher degree. If the polynomial is constant then the term does indeed remain bounded for large positive \( t \), whereas otherwise the term will grow as \( t \) gets large, for some values of the initial conditions, thereby violating the definition of stability.

Now recall that the “polynomial in \( t \)” is in fact a constant if the root \( \alpha \) is a simple root of the characteristic equation of the differential equation, and otherwise it is of higher degree. This observation completes the proof of the following:

**Theorem 1.5.1:** A linear differential equation with constant coefficients is stable if and only if all of the roots of its characteristic equation lie in the left half plane, and those that lie on the imaginary axis, if any, are simple. Such an equation is strongly stable if and only if all of the roots of its characteristic equation lie in the left half plane, and none lie on the imaginary axis.

**Exercises 1.5**

1. Determine, for each of the following differential equations whether it is strongly stable, stable, or unstable.

   (a) \( y'' - 5y' + 6y = 0 \)
(b) \( y'' + 5y' + 6y = 0 \)
(c) \( y'' + 3y = 0 \)
(d) \((D + 3)^3(D + 1)y = 0\)
(e) \((D + 1)^2(D^2 + 1)^2y = 0\)
(f) \((D^4 + 1)y = 0\)

2. Make a list of some natural phenomena that you think are unstable. Discuss.

3. The differential equation \( y'' - y = 0 \) is to be solved with the initial conditions \( y(0) = 1 \), \( y'(0) = -1 \), and then solved again with \( y(0) = 1 \), \( y'(0) = -0.99 \). Compare the two solutions when \( x = 20 \).

4. For exactly which real values of the parameter \( \lambda \) is each of the following differential equations stable? ... strongly stable?
   (a) \( y'' + (2 + \lambda)y' + y = 0 \)
   (b) \( y'' + \lambda y' + y = 0 \)
   (c) \( y' + \lambda y = 1 \)

### 1.6 Stability theory of difference equations

In the previous section we discussed the stability of differential equations. The key ideas were that such an equation is stable if every one of its solutions remains bounded as \( t \) approaches infinity, and strongly stable if the solutions actually approach zero.

Similar considerations apply to difference equations, and for similar reasons. As an example, take the equation

\[
y_{n+1} = \frac{5}{2}y_n - y_{n-1} \quad (n \geq 1)
\]

along with the initial equations

\[
y_0 = 1; \quad y_1 = 0.5.
\]

It’s easy to see that the solution is \( y_n = 2^{-n} \), and of course, this is a function that rapidly approaches zero with increasing \( n \).

Now let’s change the initial data (1.6.2), say to

\[
y_0 = 1; \quad y_1 = 0.50000001
\]

instead of (1.6.2).

The solution of the difference equation with these new data is

\[
y = (0.0000000066 \ldots)2^n + (0.9999999933 \ldots)2^{-n}.
\]

The point is that the coefficient of the growing term \( 2^n \) is small, but \( 2^n \) grows so fast that after a while the first term in (1.6.4) will be dominant. For example, when \( n = 30 \), the solution is \( y_{30} = 7.16 \), compared to the value \( y_{30} = 0.000000009 \) of the solution with the original initial data (1.6.2). A change in one part in fifty million in the initial condition produced, thirty steps later, an answer one billion times as large.

The fault lies with the difference equation, because it has both rising and falling components to its general solution. It should be clear that it is hopeless to do extended computation with an unstable difference equation, since a small roundoff error may alter the solution beyond recognition several steps later.
As in the case of differential equations, we’ll say that a difference equation is *stable* if every solution remains bounded as \( n \) grows large, and that it is *strongly stable* if every solution approaches zero as \( n \) grows large. Again, we emphasize that *every* solution must be well-behaved, not just the solution that is picked out by a certain set of initial data. In other words, the stability, or lack of it, is a property of the equation and not of the starting values.

Now consider the case where the difference equation is linear with constant coefficients. The we know that the general solution is a sum of terms of the form

\[
\text{(polynomial in } n) \alpha^n. \tag{1.6.5}
\]

Under what circumstances will such a term remain bounded or approach zero?

Suppose \(|\alpha| < 1\). Then the powers of \( \alpha \) approach zero, and multiplication by a polynomial in \( n \) does not alter that conclusion. Suppose \(|\alpha| > 1\). Then the sequence of powers grows unboundedly, and multiplication by a nonzero polynomial only speeds the parting guest.

Finally suppose the complex number \( \alpha \) has absolute value 1. Then the sequence of its powers remains bounded (in fact they all have absolute value 1), but if we multiply by a nonconstant polynomial, the resulting expression would grow without bound.

To summarize then, the term (1.6.5), if the polynomial is not identically zero, approaches zero with increasing \( n \) if and only if \(|\alpha| < 1\). It remains bounded as \( n \) increases if and only if either (a) \(|\alpha| < 1\) or (b) \(|\alpha| = 1\) and the polynomial is of degree zero (a constant). Now we have proved:

**Theorem 1.6.1:** A linear difference equation with constant coefficients is stable if and only if all of the roots of its characteristic equation have absolute value at most 1, and those of absolute value 1 are simple. The equation is strongly stable if and only if all of the roots have absolute value strictly less than 1.

### Exercises 1.6

1. Determine, for each of the following difference equations whether it is strongly stable, stable, or unstable.

   (a) \( y_{n+2} - 5y_{n+1} + 6y_n = 0 \)
   (b) \( 8y_{n+2} + 2y_{n+1} - 3y_n = 0 \)
   (c) \( 3y_{n+2} + y_n = 0 \)
   (d) \( 3y_{n+3} + 9y_{n+2} - y_{n+1} - 3y_n = 0 \)
   (e) \( 4y_{n+4} + 5y_{n+2} + y_n = 0 \)

2. The difference equation \( 2y_{n+2} + 3y_{n+1} - 2y_n = 0 \) is to be solved with the initial conditions \( y_0 = 2, \ y_1 = 1 \), and then solved again with \( y_0 = 2, \ y_1 = 0.99 \). Compare \( y_{20} \) for the two solutions.

3. For exactly which real values of the parameter \( \lambda \) is each of the following difference equations stable? . . . strongly stable?

   (a) \( y_{n+2} + \lambda y_{n+1} + y_n = 0 \)
   (b) \( y_{n+1} + \lambda y_n = 1 \)
   (c) \( y_{n+2} + y_{n+1} + \lambda y_n = 0 \)

4. (a) Consider the (constant-coefficient) difference equation

\[
\alpha_0 y_{n+p} + \alpha_1 y_{n+p-1} + \alpha_2 y_{n+p-2} + \cdots + \alpha_p y_n = 0.
\]
Show that this difference equation cannot be stable if $|a_p/a_0| > 1$.

(b) Give an example to show that the converse of the statement in part (a) is false. Namely, exhibit a difference equation for which $|a_p/a_0| < 1$ but the equation is unstable anyway.

1.7 Euler’s method

Our study of numerical methods will begin with a very simple procedure, due to Euler. We will state it as a method for solving a single differential equation of first order. One of the nice features of the subject of numerical integration of differential equations is that the techniques that are developed for just one first order differential equation will apply, with very little change, both to systems of simultaneous first order equations and to equations of higher order. Hence the consideration of a single equation of first order, seemingly a very special case, turns out to be quite general.

By an initial-value problem we mean a differential equation together with enough given values of the unknown function and its derivatives at an initial point $x_0$ to determine the solution uniquely.

Let’s suppose that we are given an initial-value problem of the form

$$y' = f(x, y); \quad y(x_0) = y_0. \quad (1.7.1)$$

Our job is to find numerical approximate values of the unknown function $y$ at points $x$ to the right of (larger than) $x_0$.

What we actually will find will be approximate values of the unknown function at a discrete set of points $x_0, x_1 = x_0 + h, x_2 = x_0 + 2h, x_3 = x_0 + 3h$, etc. At each of these points $x_n$ we will compute $y_n$, our approximation to $y(x_n)$.

Hence, suppose that the spacing $h$ between consecutive points has been chosen. We propose to start at the point $x_0$ where the initial data are given, and move to the right, obtaining $y_1$ from $y_0$, then $y_2$ from $y_1$ and so forth until sufficiently many values have been found.

Next we need to derive a method by which each value of $y$ can be obtained from its immediate predecessor. Consider the Taylor series expansion of the unknown function $y(x)$ about the point $x_n$

$$y(x_n + h) = y(x_n) + hy'_{n}(x_n) + h^2 y''_{n}(X)/2, \quad (1.7.2)$$

where we have halted the expansion after the first power of $h$ and in the remainder term, the point $X$ lies between $x_n$ and $x_n + h$.

Now equation (1.7.2) is exact, but of course it cannot be used for computation because the point $X$ is unknown. On the other hand, if we simply “forget” the error term, we’ll have only an approximate relation instead of an exact one, with the consolation that we will be able to compute from it. The approximate relation is

$$y(x_n + h) \approx y(x_n) + hy'_{n}(x_n). \quad (1.7.3)$$

Next define $y_{n+1}$ to be the approximate value of $y(x_{n+1})$ that we obtain by using the right side of (1.7.3) instead of (1.7.2). Then we get

$$y_{n+1} = y_n + hy'_{n}. \quad (1.7.4)$$

Now we have a computable formula for the approximate values of the unknown function, because the quantity $y'_{n}$ can be found from the differential equation (1.7.1) by writing

$$y'_{n} = f(x_n, y_n).$$
and if we do so then (1.7.4) takes the form

\[ y_{n+1} = y_n + hf(x_n, y_n). \] (1.7.5)

This is Euler’s method, in a very explicit form, so that the computational procedure is clear. Equation (1.7.5) is in fact a recurrence relation, or difference equation, whereby each value of \( y_n \) is computed from its immediate predecessor.

Let’s use Euler’s method to obtain a numerical solution of the differential equation

\[ y' = 0.5y \] (1.7.6)

together with the starting value \( y(0) = 1 \). The exact solution of this initial-value problem is obviously \( y(x) = e^{x/2} \).

Concerning the approximate solution by Euler’s method, we have, by comparing (1.7.6) with (1.7.1), \( f(x, y) = 0.5y \), and so

\[ y_{n+1} = y_n + \frac{h}{2} y_n \]

\[ = \left(1 + \frac{h}{2}\right) y_n. \] (1.7.7)

Therefore, in this example, each \( y_n \) will be obtained from its predecessor by multiplication by \( 1 + \frac{h}{2} \). To be quite specific, let’s take \( h \) to be 0.05. Then we show below, for each value of \( x = 0, 0.05, 0.10, 0.15, 0.20, \ldots \), the approximate value of \( y \) computed from (1.7.7) and the exact value \( y(x_n) = e^{x_n/2} \):

<table>
<thead>
<tr>
<th>( x )</th>
<th>Euler(( x ))</th>
<th>Exact(( x ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.05</td>
<td>1.02500</td>
<td>1.02532</td>
</tr>
<tr>
<td>0.10</td>
<td>1.05063</td>
<td>1.05127</td>
</tr>
<tr>
<td>0.15</td>
<td>1.07689</td>
<td>1.07788</td>
</tr>
<tr>
<td>0.20</td>
<td>1.10381</td>
<td>1.10517</td>
</tr>
<tr>
<td>0.25</td>
<td>1.13141</td>
<td>1.13315</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>1.00</td>
<td>1.63862</td>
<td>1.64872</td>
</tr>
<tr>
<td>2.00</td>
<td>2.68506</td>
<td>2.71828</td>
</tr>
<tr>
<td>3.00</td>
<td>4.39979</td>
<td>4.48169</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>5.00</td>
<td>11.81372</td>
<td>12.18249</td>
</tr>
<tr>
<td>10.00</td>
<td>139.56389</td>
<td>148.41316</td>
</tr>
</tbody>
</table>

**Table 1.7.1**

Considering the extreme simplicity of the approximation, it seems that we have done pretty well by this equation. Let’s continue with this simple example by asking for a formula for the numbers that are called Euler(\( x \)) in the above table. In other words, exactly what function of \( x \) is Euler(\( x \))? To answer this, we note first that each computed value \( y_{n+1} \) is obtained according to (1.7.7) by multiplying its predecessor \( y_n \) by \( 1 + \frac{h}{2} \). Since \( y_0 = 1 \), it is clear that we will compute \( y_n = (1 + \frac{h}{2})^n \). Now we want to express this in terms of \( x \) rather than \( n \). Since \( x_n = nh \), we have \( n = x/h \), and since \( h = 0.05 \) we have \( n = 20x \). Hence the computed approximation to \( y \) at a particular point \( x \) will be \( 1.025^{20x} \), or equivalently

\[ \text{Euler}(x) = (1.638616 \ldots)^x. \]
The approximate values can now easily be compared with the true solution, since

\[ \text{Exact}(x) = e^x = (e^{1/2})^x = (1.648721\ldots)^x. \]

Therefore both the exact solution of this differential equation and its computed solution have the form \((\text{const.})^x\). The correct value of “const.” is \(e^{1/2}\), and the value that is, in effect, used by Euler’s method is \((1 + \frac{1}{2}h)^{1/h}\). For a fixed value of \(x\), we see that if we use Euler’s method with smaller and smaller values of \(h\) (neglecting the increase in roundoff error that is sure to result), the values \(\text{Euler}(x)\) will converge to \(\text{Exact}(x)\), because

\[ \lim_{h \to 0} \left(1 + \frac{h}{2}\right)^{1/h} = e^{1/2}. \tag{1.7.8} \]

**Exercises 1.7**

1. Verify the limit (1.7.8).

2. Use a hand calculator to integrate each of the following differential equations forward ten steps, using a spacing \(h = 0.05\) with Euler’s method. Also tabulate the exact solution at each value of \(x\) that occurs.

   (a) \(y'(x) = xy(x); \ y(0) = 1\)
   (b) \(y'(x) = xy(x) + 2; \ y(0) = 1\)
   (c) \(y'(x) = \frac{y(x)}{1 + x}; \ y(0) = 1\)
   (d) \(y'(x) = -2xy(x)^2; \ y(0) = 10\)

**1.8 Software notes**

One of the main themes of our study will be the preparation of programs that not only work, but also are easily readable and usable by other people. The act of communication that must take place before a program can be used by persons other than its author is a difficult one to carry out, and we will return several times to the principles that have evolved as guides to the preparation of readable software. Here are some of these guidelines.

1. **Documentation**

   The documentation of a program is the set of written instructions to a user that inform the user about the purpose and operation of the program. At the moment that the job of writing and testing a program has been completed it is only natural to feel and urge to get the whole thing over with and get on to the next job. Besides, one might think, it’s perfectly obvious how to use this program. Some programs may be obscure, but not this one.

   It is amazing how rapidly our knowledge of our very own program fades. If we come back to a program after a lapse of a few months’ time, it often happens that we will have no idea what the program did or how to use it, at least not without making a large investment of time.

   For that reason it is important that when our program has been written and tested it should be documented immediately, while our memory of it is still green. Furthermore, the best place for documentation is in the program itself, in “comment” statements. That way one can be sure that when the comments are needed they will be available.
The first mission of program documentation is to describe the purpose of the program. State clearly the problem that the program solves, or the exact operation that it performs on its input in order to get its output.

Already in this first mission, a good bit of technical skill can be brought to bear that will be very helpful to the use, by intertwining the description of the program purpose with the names of the communicating variables in the program.

Let's see what that means by considering an example. Suppose we have written a subroutine that searches through a specified row of a matrix to find the element of largest absolute value, and outputs a column in which it was found. Such a routine, in Maple for instance, might look like this:

```maple
search:=proc(A,i)
  local j, winner, jwin;
  winner:=-1;
  for j from 1 to coldim(A) do
    if (abs(A[i,j])>winner) then
      winner:=abs(A[i,j]) ; jwin:=j fi
  od;
  return(jwin);
end;
```

Now let's try our hand at documenting this program:

"The purpose of this program is to search a given row of a matrix to find an element of largest absolute value and return the column in which it was found."

That is pretty good documentation, perhaps better than many programs get. But we can make it a lot more useful by doing the intertwining that we referred to above. There we said that the description should be related to the communicating variables. Those variables are the ones that the user can see. They are the input and output variables of the subroutine. In most important computer languages, the communicating variables are announced in the first line of the coding of a procedure or subroutine. Maple follows this convention at least for the input variables, although the output variable is usually specified in the “return” statement.

In the first line of the little subroutine above we see the list *(A, i)* of its input variables (or “arguments”). These are the ones that the user has to understand, as opposed to the other “local” variables that live inside the subroutine but don’t communicate with the outside world (like j, winner, jwin, which are listed on the second line of the program).

The best way to help the user to understand these variables is to relate them directly to the description of the purpose of the program.

"The purpose of this program is to search row *i* of a given matrix *A* to find an entry of largest absolute value, and returns the column *jwin* where that entry lives."

We’ll come back to the subject of documentation, but now let’s mention another ingredient of ease of use of programs, and that is:

2. **Modularity**

It is important to divide a long program into a number of smaller modules, each with a clearly stated set of inputs and outputs, and each with its own documentation. That means that we should get into the habit of writing lots of subroutines or procedures, because the subroutine or procedure
mode of expression forces one to be quite explicit about the relationship of the block of coding to
the rest of the world.

When we are writing a large program we would all write a subroutine if we found that a cer-
tain sequence of steps was being called for repeatedly. Beyond this, however, there are numerous
inducements for breaking off subroutines even if the block of coding occurs just once in the main
program.

For one thing it’s easier to check out the program. The testing procedure would consist of
first testing each of the subroutines separately on test problems designed just for them. Once the
subroutines work, it would remain only to test their relationships to the calling program.

For another reason, we might discover a better, faster, more elegant, or what-have-you method
of performing the task that one of these subroutines does. Then we would be able to yank out the
former subroutine and plug in the new one, while being careful only to make sure that the new
subroutine relates to the same inputs and outputs as the old one. If jobs within a large program are
not broken into subroutines it can be much harder to isolate the block of coding that deals with a
particular function and remove it without affecting the whole works.

For another reason, if one be needed, it may well happen that even though the job that is
done by the subroutine occurs only once in the current program, it may recur in other programs
as yet undreamed of. If one is in the habit of writing small independent modules and stringing
them together to make large programs, then it doesn’t take long before one has a library of useful
subroutines, each one tested, working and documented, that will greatly simplify the task of writing
future programs.

Finally, the practice of subdividing the large jobs in to the smaller jobs of which they are
composed is an extremely valuable analytical skill, one that is useful not only in programming, but
in all sorts of organizational activities where smaller efforts are to be pooled in order to produce a
larger effect. It is therefore a quality of mind that provides much of its own justification.

In this book, the major programs that are the objects of study have been broken up into subrou-
tines in the expectation that the reader will be able to start writing and checking out these modules
even before the main ideas of the current subject have been fully explained. This was done in part
because some of these programs are quite complex, and it would be unreasonable to expect the
whole program to be written in a short time. It was also done to give examples of the process of
subdivision that we have been talking about.

For instance, the general linear algebra program for solving systems of linear simultaneous equa-
tions in Part 2, has been divided into six modules, and they are described in section 2.3. The reader
might wish to look ahead at those routines and to verify that even though their relationship to the
whole job of solving equations is by no means clear now, nonetheless, because of the fact that they
are independent and self-contained, they can be programmed and checked out at any time without
waiting for the full explanation.

One more ingredient that is needed for the production of useful software is:

3. Style

We don’t mean style in the sense of “class”, although this is as welcome in programming as it
is elsewhere. There have evolved a number of elements of good programming style, and these will
mainly be discussed as they arise. But two of them (one trivial and one quite deep) are:

(a) Indentation: The instructions that lie within the range of a loop are indented in the program
listing further to the right than the instructions that announce that the loop is about to begin,
or that it has just terminated.
(b) **Top-down structuring:** When we visualize the overall logical structure of a complicated program we see a grand loop, within which there are several other loops and branchings, within which … etc. According to the principles of top-down design the looping and branching structure of the program should be visible at once in the listing. That is, we should see an announcement of the opening of the grand loop, then indented under that perhaps a two-way branch (if-then-else), where, under the “then” one sees all that will happen if the condition is met, and under the “else” one sees what happens if it is not met.

When we say that we see all that will happen, we mean that there are not any “go-to instructions that would take our eye out of the flow of the if-then-else loop to some other page. It all happens right there on the same page, under “then” and under “else”.

These few words can scarcely convey the ideas of structuring, which we leave to the numerous examples in the sequel.