Lectures on Numerical Analysis

Dennis Deturck and Herbert S. Wilf
Department of Mathematics
University of Pennsylvania
Philadelphia, PA 19104-6395

April 30, 2002

Contents

Chapter 1

Differential and Difference Equations

1.1 Introduction

In this chapter we are going to study differential equations, with particular emphasis on how to solve them with computers. We assume that the reader has previously met differential equations, so we're going to review the most basic facts about them rather quickly.

A *differential equation* is an equation in an unknown function, say $y(x)$, where the equation contains various derivatives of y and various known functions of x. The problem is to "find" the unknown function. The *order* of a differential equation is the order of the highest derivative that appears in it.

Here's an easy equation of first order:

$$y'(x) = 0. \tag{1.1.1}$$

The unknown function is $y(x) =$ constant, so we have solved the given equation (1.1.1).

The next one is a little harder:

$$y'(x) = 2y(x). \tag{1.1.2}$$

A solution will, now doubt, arrive after a bit of thought, namely $y(x) = e^{2x}$. But, if $y(x)$ is a solution of (1.1.2), then so is $10y(x)$, or $49.6y(x)$, or in fact $cy(x)$ for any constant c. Hence $y = ce^{2x}$ is a solution of (1.1.2). Are there any other solutions? No there aren't, because if y is any function that satisfies (1.1.2) then

$$(ye^{-2x})' = e^{-2x}(y' - 2y) = 0, \tag{1.1.3}$$

and so ye^{-2x} must be a constant, C.

In general, we can expect that if a differential equation is of the first order, then the most general solution will involve one arbitrary constant C. This is not always the case,

since we can write down differential equations that have no solutions at all. We would have, for instance, a fairly hard time (why?) finding a real function $y(x)$ for which

$$(y')^2 = -y^2 - 2. \tag{1.1.4}$$

There are certain special kinds of differential equations that can always be solved, and it's often important to be able to recognize them. Among there are the "first-order linear" equations

$$y'(x) + a(x)y(x) = 0, \tag{1.1.5}$$

where $a(x)$ is a given function of x.

Before we describe the solution of these equations, let's discuss the word *linear*. To say that an equation is linear is to say that if we have any two solutions $y_1(x)$ and $y_2(x)$ of the equation, then $c_1y_1(x) + c_2y_2(x)$ is also a solution of the equation, where c_1 and c_2 are any two constants (in other words, the set of solutions forms a vector space).

Equation (1.1.1) is linear, in fact, $y_1(x) = 7$ and $y_2(x) = 23$ are both solutions, and so is $7c_1 + 23c_2$. Less trivially, the equation

$$y''(x) + y(x) = 0 \tag{1.1.6}$$

is linear. The linearity of (1.1.6) can be checked right from the equation itself, without knowing what the solutions are (do it!). For an example, though, we might note that $y = \sin x$ is a solution of (1.1.6), that $y = \cos x$ is another solution of (1.1.6), and finally, by linearity, that the function $y = c_1 \sin x + c_2 \cos x$ is a solution, whatever the constants c_1 and c_2. Now let's consider an instance of the first order linear equation (1.1.5):

$$y'(x) + xy(x) = 0. \tag{1.1.7}$$

So we're looking for a function whose derivative is $-x$ times the function. Evidently $y = e^{-x^2/2}$ will do, and the general solution is $y(x) = ce^{-x^2/2}$.

If instead of (1.1.7) we had

$$y'(x) + x^2y(x) = 0,$$

then we would have found the general solution $ce^{-x^3/3}$.

As a last example, take

$$y'(x) - (\cos x)\, y(x) = 0. \tag{1.1.8}$$

The right medicine is now $y(x) = e^{\sin x}$. In the next paragraph we'll give the general rule of which the above are three examples. The reader might like to put down the book at this point and try to formulate the rule for solving (1.1.5) before going on to read about it.

Ready? What we need is to choose some antiderivative $A(x)$ of $a(x)$, and then the solution is $y(x) = ce^{-A(x)}$.

Since that was so easy, next let's put a more interesting right hand side into (1.1.5), by considering the equation

$$y'(x) + a(x)y(x) = b(x) \tag{1.1.9}$$

where now $b(x)$ is also a given function of x (Is (1.1.9) a linear equation? Are you sure?).

To solve (1.1.9), once again choose some antiderivative $A(x)$ of $a(x)$, and then note that we can rewrite (1.1.9) in the equivalent form

$$e^{-A(x)}\,\frac{d}{dx}\left(e^{A(x)}y(x)\right) = b(x).$$

Now if we multiply through by $e^{A(x)}$ we see that

$$\frac{d}{dx}\left(e^{A(x)}y(x)\right) = b(x)e^{A(x)} \tag{1.1.10}$$

so , if we integrate both sides,

$$e^{A(x)}y(x) = \int^x b(t)e^{A(t)}\,dt + \text{const.}\,, \tag{1.1.11}$$

where on the right side, we mean any antiderivative of the function under the integral sign. Consequently

$$y(x) = e^{-A(x)}\left(\int^x b(t)e^{A(t)}\,dt + \text{const.}\right). \tag{1.1.12}$$

As an example, consider the equation

$$y' + \frac{y}{x} = x + 1\,. \tag{1.1.13}$$

We find that $A(x) = \log x$, then from (1.1.12) we get

$$\begin{aligned} y(x) &= \frac{1}{x}\left(\int^x (t+1)t\,dt + C\right) \\ &= \frac{x^2}{3} + \frac{x}{2} + \frac{C}{x}. \end{aligned} \tag{1.1.14}$$

We may be doing a disservice to the reader by beginning with this discussion of certain types of differential equations that can be solved analytically, because it would be erroneous to assume that most, or even many, such equations can be dealt with by these techniques. Indeed, the reason for the importance of the numerical methods that are the main subject of this chapter is precisely that most equations that arise in "real" problems are quite intractable by analytical means, so the computer is the only hope.

Despite the above disclaimer, in the next section we will study yet another important family of differential equations that can be handled analytically, namely linear equations with constant coefficients.

Exercises 1.1

1. Find the general solution of each of the following equations:

   (a) $y' = 2\cos x$

(b) $y' + \dfrac{2}{x}\, y = 0$

(c) $y' + xy = 3$

(d) $y' + \dfrac{1}{x} y = x + 5$

(e) $2yy' = x + 1$

2. Show that the equation (1.1.4) has no real solutions.

3. Go to your computer or terminal and familiarize yourself with the equipment, the operating system, and the specific software you will be using. Then write a program that will calculate and print the sum of the squares of the integers $1, 2, \ldots, 100$. Run this program.

4. For each part of problem 1, find the solution for which $y(1) = 1$.

1.2 Linear equations with constant coefficients

One particularly pleasant, and important, type of linear differential equation is the variety with constant coefficients, such as

$$y'' + 3y' + 2y = 0\,. \tag{1.2.1}$$

It turns out that what we have to do to solve these equations is to try a solution of a certain form, and we will then find that all of the solutions indeed are of that form.

Let's see if the function $y(x) = e^{\alpha x}$ is a solution of (1.2.1). If we substitute in (1.2.1), and then cancel the common factor $e^{\alpha x}$, we are left with the quadratic equation

$$\alpha^2 + 3\alpha + 2 = 0$$

whose solutions are $\alpha = -2$ and $\alpha = -1$. Hence for those two values of α our trial function $y(x) = e^{\alpha x}$ is indeed a solution of (1.2.1). In other words, e^{-2x} is a solution, e^{-x} is a solution, and since the equation is linear,

$$y(x) = c_1 e^{-2x} + c_2 e^{-x} \tag{1.2.2}$$

is also a solution, where c_1 and c_2 are arbitrary constants. Finally, (1.2.2) must be the most general solution since it has the "right" number of arbitrary constants, namely two.

Trying a solution in the form of an exponential is always the correct first step in solving linear equations with constant coefficients. Various complications can develop, however, as illustrated by the equation

$$y'' + 4y' + 4y = 0\,. \tag{1.2.3}$$

Again, let's see if there is a solution of the form $y = e^{\alpha x}$. This time, substitution into (1.2.3) and cancellation of the factor $e^{\alpha x}$ leads to the quadratic equation

$$\alpha^2 + 4\alpha + 4 = 0, \tag{1.2.4}$$

whose two roots are identical, both being -2. Hence e^{-2x} is a solution, and of course so is c_1e^{-2x}, but we don't yet have the general solution because there is, so far, only one arbitrary constant. The difficulty, of course, is caused by the fact that the roots of (1.2.4) are not distinct.

In this case, it turns out that xe^{-2x} is another solution of the differential equation (1.2.3) (verify this), so the general solution is $(c_1 + c_2x)e^{-2x}$.

Suppose that we begin with an equation of third order, and that all three roots turn out to be the same. For instance, to solve the equation

$$y''' + 3y'' + 3y' + y = 0 \tag{1.2.5}$$

we would try $y = e^{\alpha x}$, and we would then be facing the cubic equation

$$\alpha^3 + 3\alpha^2 + 3\alpha + 1 = 0\,, \tag{1.2.6}$$

whose "three" roots are all equal to -1. Now, not only is e^{-x} a solution, but so are xe^{-x} and x^2e^{-x}.

To see why this procedure works in general, suppose we have a linear differential equation with constant coeficcients, say

$$y^{(n)} + a_1y^{(n-1)} + a_2y^{(n-2)} + \cdots + a_ny = 0 \tag{1.2.7}$$

If we try to find a solution of the usual exponential form $y = e^{\alpha x}$, then after substitution into (1.2.7) and cancellation of the common factor $e^{\alpha x}$, we would find the polynomial equation

$$\alpha^n + a_1\alpha^{n-1} + a_2\alpha^{n-2} + \cdots + a_n = 0\,. \tag{1.2.8}$$

The polynomial on the left side is called the *characteristic polynomial* of the given differential equation. Suppose now that a certain number $\alpha = \alpha^*$ is a root of (1.2.8) of multiplicity p. To say that α^* is a root of multiplicity p of the equation is to say that $(\alpha - \alpha^*)^p$ is a factor of the characteristic polynomial. Now look at the left side of the given differential equation (1.2.7). We can write it in the form

$$(D^n + a_1D^{n-1} + a_2D^{n-2} + \cdots + a_n)y = 0\,, \tag{1.2.9}$$

in which D is the differential operator d/dx. In the parentheses in (1.2.9) we see the polynomial $\varphi(D)$, where φ is exactly the characteristic polynomial in (1.2.8).

Since $\varphi(\alpha)$ has the factor $(\alpha - \alpha^*)^p$, it follows that $\varphi(D)$ has the factor $(D - \alpha^*)^p$, so the left side of (1.2.9) can be written in the form

$$g(D)(D - \alpha^*)^p y = 0\,, \tag{1.2.10}$$

where g is a polynomial of degree $n - p$. Now it's quite easy to see that $y = x^k e^{\alpha^* x}$ satisfies (1.2.10) (and therefore (1.2.7) also) for each $k = 0, 1, \ldots, p-1$. Indeed, if we substitute this function y into (1.2.10), we see that it is enough to show that

$$(D - \alpha^*)^p(x^k e^{\alpha^* x}) = 0 \qquad k = 0, 1, \ldots, p-1\,. \tag{1.2.11}$$

However, $(D-\alpha^*)(x^k e^{-\alpha^* x}) = kx^{k-1}e^{\alpha^* x}$, and if we apply $(D-\alpha^*)$ again,

$$(D-\alpha^*)^2(x^k e^{-\alpha^* x}) = k(k-1)x^{k-2}e^{\alpha^* x},$$

etc. Now since $k < p$ it is clear that $(D-\alpha^*)^p(x^k e^{-\alpha^* x}) = 0$, as claimed.

To summarize, then, if we encounter a root α^* of the characteristic equation, of multiplicity p, then corresponding to α^* we can find exactly p linearly independent solutions of the differential equation, namely

$$e^{\alpha^* x},\ xe^{\alpha^* x},\ x^2e^{\alpha^* x},\ldots,\ x^{p-1}e^{\alpha^* x}. \tag{1.2.12}$$

Another way to state it is to say that the portion of the general solution of the given differential equation that corresponds to a root α^* of the characteristic polynomial equation is $Q(x)e^{\alpha^* x}$, where $Q(x)$ is an arbitrary polynomial whose degree is one less than the multiplicity of the root α^*.

One last mild complication may arise from roots of the characteristic equation that are not real numbers. These don't really require any special attention, but they do present a few options. For instance, to solve $y'' + 4y = 0$, we find the characteristic equation $\alpha^2 + 4 = 0$, and the complex roots $\pm 2i$. Hence the general solution is obtained by the usual rule as

$$y(x) = c_1 e^{2ix} + c_2 e^{-2ix}. \tag{1.2.13}$$

This is a perfectly acceptable form of the solution, but we could make it look a bit prettier by using deMoivre's theorem, which says that

$$\begin{aligned} e^{2ix} &= \cos 2x + i\sin 2x \\ e^{-2ix} &= \cos 2x - i\sin 2x. \end{aligned} \tag{1.2.14}$$

Then our general solution would look like

$$y(x) = (c_1 + c_2)\cos 2x + (ic_1 - ic_2)\sin 2x. \tag{1.2.15}$$

But c_1 and c_2 are just arbitrary constants, hence so are $c_1 + c_2$ and $ic_1 - ic_2$, so we might as well rename them c_1 and c_2, in which case the solution would take the form

$$y(x) = c_1\cos 2x + c_2\sin 2x. \tag{1.2.16}$$

Here's an example that shows the various possibilities:

$$y^{(8)} - 5y^{(7)} + 17y^{(6)} - 997y^{(5)} + 110y^{(4)} - 531y^{(3)} + 765y^{(2)} - 567y' + 162y = 0. \tag{1.2.17}$$

The equation was cooked up to have a characteristic polynomial that can be factored as

$$(\alpha - 2)(\alpha^2 + 9)^2(\alpha - 1)^3. \tag{1.2.18}$$

Hence the roots of the characteristic equation are 2 (simple), $3i$ (multiplicity 2), $-3i$ (multiplicity 2), and 1 (multiplicity 3).

Corresponding to the root 2, the general solution will contain the term c_1e^{2x}. Corresponding to the double root at $3i$ we have terms $(c_2 + c_3x)e^{3ix}$ in the solution. From the double root at $-3i$ we get a contribution $(c_4 + c_5x)e^{-3ix}$, and finally from the triple root at 1 we get $(c_6 + c_7x + c_8x^2)e^x$. The general solution is the sum of these eight terms. Alternatively, we might have taken the four terms that come from $3i$ in the form

$$(c_2 + c_3x)\cos 3x + (c_4 + c_5x)\sin 3x. \tag{1.2.19}$$

Exercises 1.2

1. Obtain the general solutions of each of the following differential equations:
   (a) $y'' + 5y' + 6y = 0$
   (b) $y'' - 8y' + 7y = 0$
   (c) $(D + 3)^2y = 0$
   (d) $(D^2 + 16)^2y = 0$
   (e) $(D + 3)^3(D^2 - 25)^2(D + 2)^3y = 0$
2. Find a curve $y = f(x)$ that passes through the origin with unit slope, and which satisfies $(D + 4)(D - 1)y = 0$.

1.3 Difference equations

Whereas a differential equation is an equation in an unknown function, a *difference equation* is an equation in an unknown *sequence*. For example, suppose we know that a certain sequence of numbers $y_0, y_1, y_2, \ldots$ satisfies the following conditions:

$$y_{n+2} + 5y_{n+1} + 6y_n = 0 \qquad n = 0, 1, 2, \ldots \tag{1.3.1}$$

and furthermore that $y_0 = 1$ and $y_1 = 3$.

Evidently, we can compute as many of the y_n's as we need from (1.3.1), thus we would get $y_2 = -21$, $y_3 = 87$, $y_4 = -309$ so forth. The entire sequence of y_n's is determined by the difference equation (1.3.1) together with the two starting values.

Such equations are encountered when differential equations are solved on computers. Naturally, the computer can provide the values of the unknown function only at a discrete set of points. These values are computed by replacing the given differential equations by a difference equation that approximates it, and then calculating successive approximate values of the desired function from the difference equation.

Can we somehow "solve" a difference equation by obtaining a formula for the values of the solution sequence? The answer is that we can, as long as the difference equation is linear and has constant coefficients, as in (1.3.1). Just as in the case of differential equations with constant coefficients, the correct strategy for solving them is to try a solution of the

right form. In the previous section, the right form to try was $y(x) = e^{\alpha x}$. Now the winning combination is $y = \alpha^n$, where α is a constant.

In fact, let's substitute α^n for y_n in (1.3.1) to see what happens. The left side becomes

$$\alpha^{n+2} + 5\alpha^{n+1} + 6\alpha^n = \alpha^n(\alpha^2 + 5\alpha + 6) = 0. \tag{1.3.2}$$

Just as we were able to cancel the common factor $e^{\alpha x}$ in the differential equation case, so here we can cancel the α^n, and we're left with the quadratic equation

$$\alpha^2 + 5\alpha + 6 = 0. \tag{1.3.3}$$

The two roots of this *characteristic equation* are $\alpha = -2$ and $\alpha = -3$. Therefore the sequence $(-2)^n$ satisfies (1.3.1) and so does $(-3)^n$. Since the difference equation is linear, it follows that

$$y_n = c_1(-2)^n + c_2(-3)^n \tag{1.3.4}$$

is also a solution, whatever the values of the constants c_1 and c_2.

Now it is evident from (1.3.1) itself that the numbers y_n are uniquely determined if we prescribe the values of just two of them. Hence, it is very clear that when we have a solution that contains two arbitrary constants we have the most general solution.

When we take account of the given data $y_0 = 1$ and $y_1 = 3$, we get the two equations

$$\begin{cases} 1 &= c_1 + c_2 \\ 3 &= (-2)c_1 + (-3)c_2 \end{cases} \tag{1.3.5}$$

from which $c_1 = 6$ and $c_2 = -5$. Finally, we use these values of c_1 and c_2 in (1.3.4) to get

$$y_n = 6(-2)^n - 5(-3)^n \qquad n = 0, 1, 2, \ldots. \tag{1.3.6}$$

Equation (1.3.6) is the desired formula that represents the unique solution of the given difference equation together with the prescribed starting values.

Let's step back a few paces to get a better view of the solution. Notice that the formula (1.3.6) expresses the solution as a linear combination of nth powers of the roots of the associated characteristic equation (1.3.3). When n is very large, is the number y_n a large number or a small one? Evidently the powers of -3 overwhelm those of -2, so the sequence will behave roughly like a constant times powers of -3. This means that we should expect the members of the sequence to alternate in sign and to grow rapidly in magnitude.

So much for the equation (1.3.1). Now let's look at the general case, in the form of a linear difference equation of order p:

$$y_{n+p} + a_1 y_{n+p-1} + a_2 y_{n+p-2} + \cdots + a_p y_n = 0. \tag{1.3.7}$$

We try a solution of the form $y_n = \alpha^n$, and after substituting and canceling, we get the characteristic equation

$$\alpha^p + a_1\alpha^{p-1} + a_2\alpha^{p-2} + \cdots + a_p = 0. \tag{1.3.8}$$

This is a polynomial equation of degree p, so it has p roots, counting multiplicities, somewhere in the complex plane.

Let α^* be one of these p roots. If α^* is simple (*i.e.*, has multiplicity 1) then the part of the general solution that corresponds to α^* is $c(\alpha^*)^n$. If, however, α^* is a root of multiplicity $k > 1$ then we must multiply the solution $c(\alpha^*)^n$ by an arbitrary polynomial in n, of degree $k-1$, just as in the corresponding case for differential equations we used an arbitrary polynomial in x of degree $k-1$.

We illustrate this, as well as the case of complex roots, by considering the following difference equation of order five:

$$y_{n+5} - 5y_{n+4} + 9y_{n+3} - 9y_{n+2} + 8y_{n+1} - 4y_n = 0. \tag{1.3.9}$$

This example is rigged so that the characteristic equation can be factored as

$$(\alpha^2 + 1)(\alpha - 2)^2(\alpha - 1) = 0 \tag{1.3.10}$$

from which the roots are obviously i, $-i$, 2 (multiplicity 2), 1.

Corresponding to the roots i, $-i$, the portion of the general solution is $c_1 i^n + c_2(-i)^n$. Since

$$i^n = e^{in\pi/2} = \cos\left(\frac{n\pi}{2}\right) + i\sin\left(\frac{n\pi}{2}\right) \tag{1.3.11}$$

and similarly for $(-i)^n$, we can also take this part of the general solution in the form

$$c_1 \cos\left(\frac{n\pi}{2}\right) + c_2 \sin\left(\frac{n\pi}{2}\right). \tag{1.3.12}$$

The double root $\alpha = 2$ contributes $(c_3 + c_4 n)2^n$, and the simple root $\alpha = 1$ adds c_5 to the general solution, which in its full glory is

$$y_n = c_1 \cos\left(\frac{n\pi}{2}\right) + c_2 \sin\left(\frac{n\pi}{2}\right) + (c_3 + c_4 n)2^n + c_5. \tag{1.3.13}$$

The five constants would be determined by prescribing five initial values, say y_0, y_1, y_2, y_3 and y_4, as we would expect for the equation (1.3.9).

EXERCISES 1.3

1. Obtain the general solution of each of the following difference equations:

   (a) $y_{n+1} = 3y_n$

   (b) $y_{n+1} = 3y_n + 2$

   (c) $y_{n+2} - 2y_{n+1} + y_n = 0$

   (d) $y_{n+2} - 8y_{n+1} + 12y_n = 0$

   (e) $y_{n+2} - 6y_{n+1} + 9y_n = 1$

   (f) $y_{n+2} + y_n = 0$

2. Find the solution of the given difference equation that takes the prescribed initial values:
   (a) $y_{n+2} = 2y_{n+1} + y_n$; $y_0 = 0$; $y_1 = 1$
   (b) $y_{n+1} = \alpha y_n + \beta$; $y_0 = 1$
   (c) $y_{n+4} + y_n = 0$; $y_0 = 1$; $y_1 = -1$; $y_2 = 1$; $y_3 = -1$
   (d) $y_{n+2} - 5y_{n+1} + 6y_n = 0$; $y_0 = 1$; $y_1 = 2$
3. (a) For each of the difference equations in problems 1 and 2, evaluate
$$\lim_{n\to\infty} \frac{y_{n+1}}{y_n} \tag{1.3.14}$$
   if it exists.
   (b) Formulate and prove a general theorem about the existence of, and value of the limit in part (a) for a linear difference equation with constant coefficients.
   (c) Reverse the process: given a polynomial equation, find its root of largest absolute value by computing from a certain difference equation and evaluating the ratios of consecutive terms.
   (d) Write a computer program to implement the method in part (c). Use it to calculate the largest root of the equation
$$x^8 = x^7 + x^6 + x^5 + \cdots + 1. \tag{1.3.15}$$

1.4 Computing with difference equations

This is, after all, a book about computing, so let's begin with computing from difference equations since they will give us a chance to discuss some important questions that concern the design of computer programs. For a sample difference equation we'll use

$$y_{n+3} = y_{n+2} + 5y_{n+1} + 3y_n \tag{1.4.1}$$

together with the starting values $y_0 = y_1 = y_2 = 1$. The reader might want, just for practice, to find an explicit formula for this sequence by the methods of the previous section.

Suppose we want to compute a large number of these y's in order to verify some property that they have, for instance to check that

$$\lim_{n\to\infty} \frac{y_{n+1}}{y_n} = 3 \tag{1.4.2}$$

which must be true since 3 is the root of largest absolute value of the characteristic equation.

As a first approach, we might declare y to be a linear array of some size large enough to accommodate the expected length of the calculation. Then the rest is easy. For each n, we would calculate the next y_{n+1} from (1.4.1), we would divide it by its predecessor y_n to get a new ratio. If the new ratio agrees sufficiently well with the previous ratio we announce that the computation has terminated and print the new ratio as our answer. Otherwise, we move the new ratio to the location of the old ratio, increase n and try again.

If we were to write this out as formal procedure (algorithm) it might look like:

$y_0 := 1$; $y_1 := 1$; $y_2 := 1$; $n := 2$;
newrat $:= -10$; oldrat $:= 1$;
while |newrat − oldrat| ≥ 0.000001 **do**
  oldrat := newrat; $n := n + 1$;
  $y_n := y_{n-1} + 5 * y_{n-2} + 3 * y_{n-3}$;
  newrat $:= y_n / y_{n-1}$
endwhile
print newrat**; Halt.**

We'll use the symbol ':=' to mean that we are to compute the quantity on the right, if necessary, and then store it in the place named on the left. It can be read as 'is replaced by' or 'is assigned.' Also, the block that begins with '**while**' and ends with '**endwhile**' represents a group of instructions that are to be executed repeatedly until the condition that follows '**while**' becomes false, at which point the line following '**endwhile**' is executed.

The procedure just described is fast, but it uses lots of storage. If, for instance, such a program needed to calculate 79 y's before convergence occurred, then it would have used 79 locations of array storage. In fact, the problem above doesn't need that many locations because convergence happens a lot sooner. Suppose you wanted to find out how much sooner, given only a programmable hand calculator with ten or twenty memory locations. Then you might appreciate a calculation procedure that needs just four locations to hold all necessary y's.

That's fairly easy to accomplish, though. At any given moment in the program, what we need to find the next y are just the previous three y's. So why not save only those three? We'll use the previous three to calculate the next one, and stow it for a moment in a fourth location. Then we'll compute the new ratio and compare it with the old. If they're not close enough, we move each one of the three newest y's back one step into the places where we store the latest three y's and repeat the process. Formally, it might be:

$y := 1$; ym1 $:= 1$; ym2 $:= 1$;
newrat $:= -10$; oldrat $:= 1$;
while |newrat − oldrat| ≥ 0.000001 **do**
  ym3 := ym2; ym2 := ym1; ym1 := y;
  oldrat := newrat;
  $y :=$ ym1 $+ 5 *$ ym2 $+ 3 *$ ym3;
  newrat := y/ym1 **endwhile**;
print newrat; **Halt**.

The calculation can now be done in exactly six memory locations (y, ym1, ym2, ym3, oldrat, newrat) no matter how many y's have to be calculated, so you can undertake it on your hand calculator with complete confidence. The price that we pay for the memory saving is that we must move the data around a bit more.

One should not think that such programming methods are only for hand calculators. As we progress through the numerical solution of differential equations we will see situations in which each of the quantities that appears in the difference equation will itself be an

array (!), and that very large numbers, perhaps thousands, of these arrays will need to be computed. Even large computers might quake at the thought of using the first method above, rather than the second, for doing the calculation. Fortunately, it will almost never be necessary to save in memory all of the computed values simultaneously. Normally, they will be computed, and then printed or plotted, and never needed except in the calculation of their immediate successors.

EXERCISES 1.4

1. The Fibonacci numbers $F_0, F_1, F_2, \ldots$ are defined by the recurrence formula $F_{n+2} = F_{n+1} + F_n$ for $n = 0, 1, 2, \ldots$ together with the starting values $F_0 = 0$, $F_1 = 1$.

   (a) Write out the first ten Fibonacci numbers.

   (b) Derive an explicit formula for the nth Fibonacci number F_n.

   (c) Evaluate your formula for $n = 0, 1, 2, 3, 4$.

   (d) Prove directly from your formula that the Fibonacci numbers are integers (This is perfectly obvious from their definition, but is not so obvious from the formula!).

   (e) Evaluate
   $$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} \tag{1.4.3}$$

   (f) Write a computer program that will compute Fibonacci numbers and print out the limit in part (e) above, correct to six decimal places.

   (g) Write a computer program that will compute the first 40 members of the modified Fibonacci sequence in which $F_0 = 1$ and $F_1 = (1 - \sqrt{5})/2$. Do these computed numbers seem to be approaching zero? Explain carefully what you see and why it happens.

   (h) Modify the program of part (h) to run in higher (or double) precision arithmetic. Does it change any of your answers?

2. Find the most general solution of each of the following difference equations:

   (a) $y_{n+1} - 2y_n + y_{n-1} = 0$

   (b) $y_{n+1} = 2y_n$

   (c) $y_{n+2} + y_n = 0$

   (d) $y_{n+2} + 3y_{n+1} + 3y_n + y_{n-1} = 0$

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, might still be too complicated to solve exactly on a computer.

For instance, may 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

$$y'' - y' - 2y = 0 \tag{1.5.1}$$

together with the initial data

$$y(0) = 1; \qquad y'(0) = -1. \tag{1.5.2}$$

We are thinking of the independent variable t as the time, 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

$$y(t) = (0.999666\ldots)e^{-t} + (0.000333\ldots)e^{2t} \tag{1.5.3}$$

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_1e^{-t}, and a rising exponential term c_2e^{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 α is positive, that equation is unstable.

Let's restrict attention now to linear differential equations with constant coefficients. We know from section 1.2 that the general solution of such an equation is a sum of terms of the form

$$(\text{polynomial in } t)e^{\alpha t}. \tag{1.5.4}$$

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

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

This takes care of all of the possibilities except the case where α is zero, or more generally, the complex number α has zero real part (is purely imaginary). In that case the question of whether (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 α 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 λ 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

$$y_{n+1} = \frac{5}{2}y_n - y_{n-1} \qquad (n \geq 1) \tag{1.6.1}$$

along with the initial equations

$$y_0 = 1; \quad y_1 = 0.5\,. \tag{1.6.2}$$

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

$$y_0 = 1; \quad y_1 = 0.50000001 \tag{1.6.3}$$

instead of (1.6.2).

The solution of the difference equation with these new data is

$$y = (0.0000000066\ldots)2^n + (0.9999999933\ldots)2^{-n}. \tag{1.6.4}$$

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 $y_{30} = 0.0000000009$ of the solution with the original initial data (1.6.2). A change of 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 α 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 α 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 λ 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

$$a_0 y_{n+p} + a_1 y_{n+p-1} + a_2 y_{n+p-2} + \cdots + a_p y_n = 0. \tag{1.6.6}$$

   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.

Chapter 2

The Numerical Solution of Differential Equations

2.1 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

$$y' = f(x, y); \qquad y(x_0) = y_0. \tag{2.1.1}$$

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

$$y(x_n + h) = y(x_n) + hy'(x_n) + h^2 \frac{y''(X)}{2}\,, \tag{2.1.2}$$

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 (2.1.2) 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

$$y(x_n + h) \approx y(x_n) + hy'(x_n). \tag{2.1.3}$$

Next define y_{n+1} to be the approximate value of $y(x_{n+1})$ that we obtain by using the right side of (2.1.3) instead of (2.1.2). Then we get

$$y_{n+1} = y_n + hy'_n\,. \tag{2.1.4}$$

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 (2.1.1) by writing

$$y'_n = f(x_x, y_n), \tag{2.1.5}$$

and if we do so then (2.1.4) takes the form

$$y_{n+1} = y_n + hf(x_n, y_n). \tag{2.1.6}$$

This is Euler's method, in a very explicit form, so that the computational procedure is clear. Equation (2.1.6) 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

$$y' = 0.5y \tag{2.1.7}$$

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 (2.1.7) with (2.1.1), $f(x, y) = 0.5y$, so

$$\begin{aligned} y_{n+1} &= y_n + h\frac{y_n}{2} \\ &= \left(1 + \frac{h}{2}\right) y_n\,. \end{aligned} \tag{2.1.8}$$

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 (2.1.8) and the exact value $y(x_n) = e^{x_n/2}$:

x	Euler(x)	Exact(x)
0.00	1.00000	1.00000
0.05	1.02500	1.02532
0.10	1.05063	1.05127
0.15	1.07689	1.07788
0.20	1.10381	1.10517
0.25	1.13141	1.13315
⋮	⋮	⋮
1.00	1.63862	1.64872
2.00	2.68506	2.71828
3.00	4.39979	4.48169
⋮	⋮	⋮
5.00	11.81372	12.18249
10.00	139.56389	148.41316

TABLE 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 (2.1.8) by multiplying its predecessor y_n by $1 + \frac{h}{2}$. Since $y_0 = 1$, it is clear tha 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

$$\text{Euler}(x) = (1.638616\ldots)^x. \tag{2.1.9}$$

The approximate values can now easily be compared with the true solution, since

$$\begin{aligned} \text{Exact}(x) = e^{\frac{x}{2}} &= \left(e^{\frac{1}{2}}\right)^x \\ &= (1.648721\ldots)^x. \end{aligned} \tag{2.1.10}$$

Therefore both the exact solution of this differential equation and its computed solution have the form (const.)^x. The correct value of "const." is $e^{1/2}$, and the value that is, in effect, used by Euler's method is $(1 + \frac{h}{2})^{1/h}$. 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 Euler(x) will converge to Exact(x), because

$$\lim_{h\to 0}\left(1 + \frac{h}{2}\right)^{1/h} = e^{\frac{1}{2}}. \tag{2.1.11}$$

EXERCISES 2.1

1. Verify the limit (2.1.11).

2. Use a calculator or a computer 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) $y'(x) = \dfrac{y(x)}{1+x}$; $y(0) = 1$
   (d) $y'(x) = -2xy(x)^2$; $y(0) = 10$

2.2 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 useable 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 an 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:

```
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 certain 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 into 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 subroutines 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 equations in Chapter 3, has been divided into six modules, and they are described in section 3.3. 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.

2.3 Systems and equations of higher order

We have already remarked that the methods of numerical integration for a single first-order differential equation carry over with very little change to systems of simultaneous differential equations of first order. In this section we'll discuss exactly how this is done, and furthermore, how the same idea can be applied to equations of higher order than the first. Euler's method will be used as the example, but the same transformations will apply to all of the methods that we will study.

In Euler's method for one equation, the approximate value of the unknown function at the next point $x_{n+1} = x_n + h$ is calculated from

$$y_{n+1} = y_n + hf(x_n, y_n). \tag{2.3.1}$$

Now suppose that we are trying to solve not just a single equation, but a system of N simultaneous equations, say

$$y_i'(x) = f_i(x, y_1, y_2, \ldots y_N) \qquad i = 1, \ldots, N. \tag{2.3.2}$$

Equation (2.3.2) represents N equations, in each of which just one derivative appears, and whose right-hand side may depend on x, and on all of the unknown functions, but not on their derivatives. The "f_i " indicates that, of course, each equation can have a different right-hand side.

Now introduce the vector $y(x)$ of unknown functions

$$y(x) = [y_1(x), y_2(x), y_3(x), \ldots, y_N(x)] \tag{2.3.3}$$

and the vector $f = f(x, y(x))$ of right-hand sides

$$f = [f_1(x, y), f_2(x, y), \ldots, f_N(x, y)]. \tag{2.3.4}$$

In terms of these vectors, equation (2.3.2) can be rewritten as

$$y'(x) = f(x, y(x)). \tag{2.3.5}$$

We observe that equation (2.3.5) looks just like our standard form (2.1.1) for a single equation in a single unknown function, except for the bold face type, i.e., except for the fact that y and f now represent vector quantities.

To apply a numerical method such as that of Euler, then, all we need to do is to take the statement of the method for a single differential equation in a single unknown function, and replace $y(x)$ and $f(x, y(x))$ by vector quantities as above. We will then have obtained the generalization of the numerical method to systems.

To be specific, Euler's method for a single equation is

$$y_{n+1} = y_n + hf(x, y_n) \tag{2.3.6}$$

so Euler's method for a system of differential equations will be

$$y_{n+1} = y_n + hf(x_n, y_n). \tag{2.3.7}$$

This means that if we know the entire vector y of unknown functions at the point $x = x_n$, then we can find the entire vector of unknown functions at the next point $x_{n+1} = x_n + h$ by means of (2.3.7).

In detail, if $\overline{y}_i(x_n)$ denotes the computed approximate value of the unknown function y_i at the point x_n, then what we must calculate are

$$\overline{y}_i(x_{n+1}) = \overline{y}_i(x_n) + hf_i(x_n, \overline{y}_1(x_n), \overline{y}_2(x_n), \ldots, \overline{y}_N(x_n)) \tag{2.3.8}$$

for each $i = 1, 2, \ldots, N$.

As an example, take the pair of differential equations

$$\left\{ \begin{array}{lcl} y_1' & = & x + y_1 + y_2 \\ y_2' & = & y_1 y_2 + 1 \end{array} \right. \tag{2.3.9}$$

together with the initial values $y_1(0) = 0$, $y_2(0) = 1$.

Now the vector of unknown functions is $y = [y_1, y_2]$, and the vector of right-hand sides is $f = [x + y_1 + y_2,\ y_1 y_2 + 1]$. Initially, the vector of unknowns is $y = [0, 1]$. Let's choose a step size $h = 0.05$. Then we calculate

$$\begin{bmatrix} \overline{y}_1(0.05) \\ \overline{y}_2(0.05) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.05 \begin{bmatrix} 0+0+1 \\ 0*1+1 \end{bmatrix} = \begin{bmatrix} 0.05 \\ 1.05 \end{bmatrix} \tag{2.3.10}$$

and

$$\begin{bmatrix} \overline{y}_1(0.10) \\ \overline{y}_2(0.10) \end{bmatrix} = \begin{bmatrix} 0.05 \\ 1.05 \end{bmatrix} + 0.05 \begin{bmatrix} 0.05+0.05+1.05 \\ 0.05*1.05+1 \end{bmatrix} = \begin{bmatrix} 0.1075 \\ 1.102625 \end{bmatrix} \tag{2.3.11}$$

and so forth. At each step we compute the vector of approximate values of the two unknown functions from the corresponding vector at the immediately preceding step.

Let's consider the preparation of a computer program that will carry out the solution, by Euler's method, of a system of N simultaneous equations of the form (2.3.2), which we will rewrite just slightly, in the form

$$y_i' = f_i(x, y) \qquad i = 1, \ldots, N. \tag{2.3.12}$$

Note that on the left is just one of the unknown functions, and on the right there may appear all N of them in each equation.

Evidently we will need an array `Y` of length N to hold the values of the N unknown functions at the current point x. Suppose we have computed the array `Y` at a point x, and we want to get the new array `Y` at the point $x + h$. Exactly what do we do?

Some care is necessary in answering this question because there is a bit of a snare in the underbrush. The new values of the N unknown functions are calculated from (2.3.8) or (2.3.12) in a certain order. For instance, we might calculate $\overline{y}_1(x+h)$, then $\overline{y}_2(x+h)$, then $\overline{y}_3(x+h)$,..., then $\overline{y}_N(x+h)$.

The question is this: when we compute $\overline{y}_1(x+h)$, where shall we put it? If we put it into `Y[1]`, the first position of the `Y` array in storage, then the previous contents of `Y[1]` are lost, *i.e.*, the value of $\overline{y}_1(x)$ is lost. But we aren't finished with $\overline{y}_1(x)$ yet; it's still needed to compute $\overline{y}_2(x+h)$, $\overline{y}_3(x+h)$, etc. This is because the new value $\overline{y}_i(x+h)$ depends (or might depend), according to (2.3.8), on the old values of *all* of the unknown functions, including those whose new values have already been computed before we begin the computation of $\overline{y}_i(x+h)$.

If the point still is murky, go back to (2.3.11) and notice how, in the calculation of $\overline{y}_2(0.10)$ we needed to know $\overline{y}_1(0.05)$ even though $\overline{y}_1(0.10)$ had already been computed. Hence if we had put $\overline{y}_1(0.10)$ into an array to replace the old value $\overline{y}_1(0.05)$ we would not have been able to obtain $\overline{y}_2(0.10)$.

The conclusion is that we need at least two arrays, say `YIN` and `YOUT`, each of length N. The array `YIN` holds the unknown functions evaluated at x, and `YOUT` will hold their values at $x+h$. Initially `YIN` holds the given data at x_0. Then we compute all of the unknowns at $x_0 + h$, and store them in `YOUT` as we find them. When all have been done, we print them if desired, move all entries of `YOUT` back to `YIN`, increase x by h and repeat.

The principal building block in this structure would be a subroutine that would advance the solution exactly one step. The main program would initialize the arrays, call this subroutine, increase x, move date from the output array `YOUT` back to the input array `YIN`, print, etc. The single-step subroutine is shown below. We will use it later on to help us get a solution started when we use methods that need information at more than one point before they can start the integration.

```
Eulerstep:=proc(xin,yin,h,n) local i,yout;
# This program numerically integrates the system
# y'=f(x,y) one step forward by Euler's method using step size
# h.  Enter with values at xin in yin.  Exit with values at xin+h
# in yout.  Supply f as a function subprogram.
yout:=[seq(evalf(yin[i]+h*f(xin,yin,i)),i=1..n)];
return(yout);
end;
```

A few remarks about the program are in order. One structured data type in Maple is a list of things, in this case, a list of floating point numbers. The `seq` command (for "sequence") creates such a list, in this case a list of length n since i goes from 1 to n in the `seq` command. The brackets `[` and `]` convert the list into a vector. The `evalf` command ensures that the results of the computation of the components of `yout` are floating point numbers.

Our next remark about the program concerns the function subprogram `f`, which calculates the right hand sides of the differential equation. This subprogram, of course, must be supplied by the user. Here is a sample of such a program, namely the one that describes the system (2.3.9). In that case we have $f_1(x, y) = x + y_1 + y_2$ and $f_2(x, y) = y_1 y_2 + 1$. This translates into the following:

```
f:=proc(x,y,i);
# Calculates the right-hand sides of the system of differential
# equations.
if i=1 then return(x+y[1]+y[2]) else return(y[1]*y[2]+1) fi;
end;
```

Our last comment about the program to solve systems is that it is perfectly possible to use it in such a way that we would not have to move the contents of the vector `YOUT` back to the vector `YIN` at each step. In other words, we could save N move operations, where N is the number of equations. Such savings might be significant in an extended calculation.

To achieve this saving, we write two blocks of programming in the main program. One block takes the contents of `YIN` as input, advances the solution one step by Euler's method and prints, leaving the new vector in `YOUT`. Then, without moving anything, another block of programming takes the contents of `YOUT` as input, advances the solution one step, leaves the new vector in `YIN`, and prints. The two blocks call `Euler` alternately as the integration proceeds to the right. The reader might enjoy writing this program, and thinking about how to generalize the idea to the situation where the new value of y is computed from two

previously computed values, rather than from just one (then three blocks of programming would be needed).

Now we've discussed the numerical solution of a single differential equation of first order, and of a system of simultaneous differential equations of first order, and there remains the treatment of equations of higher order than the first. Fortunately, this case is very easily reduced to the varieties that we have already studied.

For example, suppose we want to solve a single equation of the second order, say

$$y'' + xy' + (x+1)\cos y = 2. \tag{2.3.13}$$

The strategy is to transform the single second-order equation into a pair of simultaneous first order equations that can then be handled as before. To do this, choose two unknown functions u and v. The function u is to be the unknown function y in (2.3.13), and the function v is to be the derivative of u. Then u and v satisfy two simultaneous first-order differential equations:

$$\begin{cases} u' &= v \\ v' &= -xv - (x+1)\cos u + 2 \end{cases} \tag{2.3.14}$$

and these are exactly of the form (2.3.5) that we have already discussed!

The same trick works on a general differential equation of Nth order

$$y^{(N)} + G(x, y, y', y'', \ldots, y^{(N-1)}) = 0. \tag{2.3.15}$$

We introduce N unknown functions $u_0, u_1, \ldots, u_{N-1}$, and let them be the solutions of the system of N simultaneous first order equations

$$\begin{aligned} u_0' &= u_1 \\ u_1' &= u_2 \\ &\ldots \\ u_{N-2}' &= u_{N-1} \\ u_{N-1}' &= -G(x, u_0, u_1, \ldots, u_{N-2}). \end{aligned} \tag{2.3.16}$$

The system can now be dealt with as before.

Exercises 2.3

1. Write each of the following as a system of simultaneous first-order initial-value problems in the standard form (2.3.2):

   (a) $y'' + x^2 y = 0$; $y(0) = 1$; $y'(0) = 0$

   (b) $u' + xv = 2$; $v' + e^{uv} = 0$; $u(0) = 0$; $v(0) = 0$

   (c) $u' + xv' = 0$; $v' + x^2 u = 1$; $u(1) = 1$; $v(1) = 0$

   (d) $y^{iv} + 3xy''' + x^2 y'' + 2y' + y = 0$; $y(0) = y'(0) = y''(0) = y'''(0) = 1$

   (e) $x'''(t) + t^3 x(t) + y(t) = 0$; $y''(t) + x(t)^2 = t^3$; $x(0) = 2$; $x'(0) = 1$; $x''(0) = 0$; $y(0) = 1$; $y'(0) = 0$

2. For each of the parts of problem 1, write the function subprogram that will compute the right-hand sides, as required by the `Eulerstep` subroutine.

3. For each of the parts of problem 1, assemble nad run on the computer the `Euler` program, together with the relevant function subprogram of problem 2, to print out the solutions for fifty steps of integration, each of size $h = 0.03$. Begin with $x = x_0$, the point at which the initial data was given.

4. Reprogram the `Eulerstep` subroutine, as discussed in the text, to avoid the movement of `YOUT` back to `YIN`.

5. Modify your program as necessary (in Maple, take advantage of the `plot` command) to produce graphical output (graph all of the unknown functions on the same axes). Test your program by running it with `Euler` as it solves $y'' + y = 0$ with $y(0) = 0$, $y'(0) = 1$, and $h = \pi/60$ for 150 steps.

6. Write a program that will compute successive values $y_p,\ y_{p+1}, \ldots$ from a difference equation of order p. Do this by storing the y's as they are computed in a circular list, so that it is never necessary to move back the last p computed values before finding the next one. Write your program so that it will work with vectors, so you can solve systems of difference equations as well as single ones.

2.4 How to document a program

One of the main themes of our study will be the preparation of programs that not only work, but also are easily readable and useable 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 serve as guides to the preparation of readable software.

In this section we discuss further the all-important question of program documentation, already touched upon in section 2.2. Some very nontrivial skills are called for in the creation of good user-oriented program descriptions. One of these is the ability to enter the head of another person, the user, and to relate to the program that you have just written through the user's eyes.

It's hard to enter someone else's head. One of the skills that make one person a better teacher than another person is of the same kind: the ability to see the subject matter that is being taught through the eyes of another person, the student. If one can do that, or even make a good try at it, then obviously one will be able much better to deal with the questions that are really concerning the audience. Relatively few actually do this to any great extent not, I think, because it's an ability that one either has or doesn't have, but because few efforts are made to train this skill and to develop it.

We'll try to make our little contribution here.

(A) What does it do?

The first task should be to describe the precise purpose of the program. Put yourself in the place of a potential user who is looking at a particular numerical instance of the problem that needs solving. That user is now thumbing through a book full of program descriptions in the library of a computer center 10,000 miles away in order to find a program that will do the problem in question. Your description must answer that question.

Let's now assume that your program has been written in the form of a subroutine or procedure, rather than as a main program. Then the list of global, or communicating variables is plainly in view, in the opening statement of the subroutine.

As we noted in section 2.2, you should state the purpose of your program using the global variables of the subroutine in the same sentence. For one example of the difference that makes, see section 2.2. For another, a linear equation solver might be described by saying

> "This program solves a system of simultaneous equations. To use it, put the right-hand sides into the vector `B`, put the coefficients into the matrix `A` and call the routine. The answer will be returned in `X`."

We are, however, urging the reader to do it this way:

> "This program solves the equations `AX=B`, where `A` is an N-by-N matrix and `B` is an N-vector."

Observe that the second description is shorter, only about half as long, and yet more informative. We have found out not only what the program does, but how that function relates to the global variables of the subroutine. This was done by using a judicious sprinkling of symbols in the documentation, along with the words. Don't use only symbols, or only words, but weave them together for maximum information.

Notice also that the ambiguous term "right-hand side" that appeared in the first form has been done away with in the second form. The phrase was ambiguous because exactly what ends up on the right-hand side and what on the left is an accident of how we happen to write the equations, and your audience may not do it the same way you do.

(B) How is it done?

This is usually the easy part of program documentation because it is not the purpose of this documentation to give a course in mathematics or algorithms or anything else. Hence most of the time a reference to the literature is enough, or perhaps if the method is a standard one, just give its name. Often though, variations on the standard method have been chosen, and the user must be informed about those:

> "... is solved by Gaussian elimination, using complete positioning for size..."

> "... the input array `A` is sorted by the Quicksort method (see D.E. Knuth, The Art of Computer Programming, volume 3)..."

"...the eigenvalues and vectors are found by the Jacobi method, using Corbató's method of avoiding the search for the largest off-diagonal element (see, for instance, the description in D.R. Wilson, A First Course in Mathematical Software)."

"...is found by the Simplex method, except that Charnes' selection rule (see F.A. Ficken, The Simplex Method...) is not programmed, and so..."

(C) Describe the global variables

Now it gets hard again. The global variables are the ones through which the subroutine communicates with the user. Generally speaking, the user doesn't care about variables that are entirely local to your subroutine, but is vitally concerned with the communicating variables.

First the user has to know exactly ho each of the global variables is related to the problem that is being solved. This calls for a brief verbal description of the variable, and what it has to do with the functioning of the program.

"`A[i]` is the `i`th element of the input list that is to be sorted, `i=1..N`"

"`WHY` is set by the subroutine to `TRUE` unles the return is because of overflow, and then it will be set to `FALSE`."

"`B[i,j]` is the coefficient of `X[j]` in the `i`th one of the input equations `BX=C`."

"`option` is set by the calling program on input. Set it to 0 if the output is to be rounded to the nearest integer, else set it to m if the output is to be rounded to m decimal places ($m \leq 12$)."

It is extremely important that each and every global variable of the subroutine should get such a description. Just march through the parentheses in the subroutine or procedure heading, and describe each variable in turn.

Next, the user will need more information about each of the global variables than just its description as above. Also required is the "type" of the variable. Some computer languages force each program to declare the types of their variables right in the opening statement. Others declare types by observing various default rules with exceptions stated. In any case, a little redundancy never hurts, and the program documentation should declare the type of each and every global variable.

It's easy to declare along with the types of the variables, their dimensions if they are array variables. For instance we may have a

```
solver:=proc(A,X,n,ndim,b);
```

in which the communicating variables have the following types:

`A`	`ndim`-by-`ndim` array of floating point numbers
`X`	vector of floating point numbers of length `n`
`n`	integer
`ndim`	integer
`b`	vector of floating point numbers of length `n`

The best way to announce all of these types and dimensions of global variables to the user is simply to list them, as above, in a table.

Now surely we've finished taking the pulse, blood pressure, etc. of the global variables, haven't we? Well, no, we haven't. There's still more vital data that a user will need to know about these variables. There isn't any standard name like "type" to apply to this information, so we'll call it the "role" of the variable.

First, for some of the global variables of the subroutine, it may be true that their values at the time the subroutine is called are quite irrelevant to the operation of the subroutine. This would be the case for the output variables, and in certain other situations. For some other variables, the values at input time would be crucial. The user needs to know which are which. Just for one example, if the value at input time is irrelevant, then the user can feel free to use the same storage for other temporary purposes between calls to the subroutine.

Second, it may happen that certain variables are returned by the subroutine with their values unchanged. This is particularly true for "implicitly passed" global variables, i.e., variables whose values are used by the subroutine but which do not appear explicitly in the argument list. In such cases, the user may be delighted to hear the good news. In other cases, the action of a subroutine may change an input variable, so if the user needs to use those quantities again it will be necessary to save them somewhere else before calling the subroutine. In either case, the user needs to be informed.

Third, it may be that the computation of the value of a certain variable is one of the main purposes of the subroutine. Such variables are the outputs of the program, and the user needs to know which these are (whether they are explicit in heading or the `return` statement, or are "implicit").

Although some high-level computer languages require type declarations immediately in the opening instruction of a subroutine, none require the descriptions of the roles of the variables (well, Pascal requires the `VAR` declaration, and Maple separates the input variables from the output ones, but both languages allow implicit passing and changing of global variables). These are, however, important for the user to know, so let's invent a shorthand for describing them in the documentation of the programs that occur in this book.

First, if the value at input time is important, let's say that the role of the variable is `I`, otherwise it is `I'`.

Second, if the value of the variable is changed by the action of the subroutine, we'll say that its role is `C`, else `C'`.

Finally, if the computation of this variable is one of the main purposes of the subroutine,

it's role is `O` (as in output), else `O'`.

In the description of each communicating variable, all three of these should be specified,. Thus, a variable `X` might have role `IC'O'`, or a variable `why` might be of role `I'CO`, etc.

To sum up, the essential features of program documentation are a description of that the program does, phrased in terms of the global variables, a statement of how it gets the job done, and a list of all of the global variables, showing for each one its name, type, dimension (or structure) if any, its role, and a brief verbal description.

Refer back to the short program in section 2.2, that searches for the largest element in a row of a matrix. Here is the table of information about its global variables:

Name	Type	Role	Description
`A`	floating point matrix	`IC'O'`	The input matrix
`i`	integer	`IC'O'`	Which row to search
`jwin`	integer	`I'CO`	Column containing largest element

Exercises 2.4

Write programs that perform each of the jobs stated below. In each case, after testing the program, document it with comments. Give a complete table of information about the global variables in each case.

(a) Find and print all of the prime numbers between M and N.

(b) Find the elements of largest and smallest absolute values in a given linear array (vector), and their positions in the array.

(c) Sort the elements of a given linear array into ascending order of size.

(d) Deal out four bridge hands (13 cards each from a standard 52-card deck – This one is not so easy!).

(e) Solve a quadratic equation (any quadratic equation!).

2.5 The midpoint and trapezoidal rules

Euler's formula is doubtless the simplest numerical integration procedure for differential equations, but the accuracy that can be obtained with it is insufficient for most applications. In this section and those that follow, we want to introduce a while family of methods for the solution of differential equations, called the *linear multistep methods*, in which the user can choose the degree of precision that will suffice for the job, and then select a member of the family that will achieve it.

Before describing the family in all of its generality, we will produce two more of its members, which illustrate the different sorts of creatures that inhabit the family in question.

Recall that we derived Euler's method by chopping off the Taylor series expansion of the solution after the linear term. To get a more accurate method we could, of course, keep the quadratic term, too. However, that term involves a second derivative, and we want to avoid the calculation of higher derivatives because our differential equations will always be written as first-order systems, so that only the first derivative will be conveniently computable.

We can have greater accuracy without having to calculate higher derivatives if we're willing to allow our numerical integration procedure to involve values of the unknown function and its derivative at more than one point. In other words, in Euler's method, the next value of the unknown function, at $x + h$, is gotten from the values of y and y' at just one backwards point x. In the more accurate formulas that we will discuss next, the new value of y depends in y and y' at more than one point, for instance, at x and $x - h$, or at several points.

As a primitive example of this kind, we will now discuss the *midpoint rule*. We begin once again with the Taylor expansion of the unknown function $y(x)$ about the point x_n:

$$y(x_n + h) = y(x_n) + hy'(x_n) + h^2 \frac{y''(x_n)}{2} + h^3 \frac{y'''(x_n)}{6} + \cdots . \tag{2.5.1}$$

Now we rewrite equation (2.5.1) with h replaced by $-h$ to get

$$y(x_n - h) = y(x_n) - hy'(x_n) + h^2 \frac{y''(x_n)}{2} - h^3 \frac{y'''(x_n)}{6} + \cdots \tag{2.5.2}$$

and then subtract these equations, obtaining

$$y(x_n + h) - y(x_n - h) = 2hy'(x_n) + 2h^3 \frac{y'''(x_n)}{6} + \cdots . \tag{2.5.3}$$

Now, just as we did in the derivation of Euler's method, we will truncate the right side of (2.5.3) after the first term, ignoring the terms that involve h^3, h^5, etc. Further, let's use y_n to denote the computed approximate value of $y(x_n)$ (and y_{n+1} for the approximate $y(x_{n+1})$, etc.). Then we have

$$y_{n+1} - y_{n-1} = 2hy'_n . \tag{2.5.4}$$

If, as usual, we are solving the differential equation $y' = f(x, y)$, then finally (2.5.4) takes the form

$$y_{n+1} = y_{n-1} + 2hf(x_n, y_n) \tag{2.5.5}$$

and this is the midpoint rule. The name arises from the fact that the first derivative y'_n is being approximated by the slope of the chord that joins the two points (x_{n-1}, y_{n-1}) and (x_{n+1}, y_{n+1}), instead of the chord joining (x_n, y_n) and (x_{n+1}, y_{n+1}) as in Euler's method.

At first sight it seems that (2.5.5) can be used just like Euler's method, because it is a recurrence formula in which we compute the next value y_{n+1} from the two previous values y_n and y_{n-1}. Indeed the rules are quite similar, except for the fact that we can't get started with the midpoint rule until we know *two* consecutive values y_0, y_1 of the unknown function at two consecutive points x_0, x_1. Normally a differential equation is given together with just one value of the unknown function, so if we are to use the midpoint rule we'll need to manufacture one more value of $y(x)$ by some other means.

This kind of situation will come up again and again as we look at more accurate methods, because to obtain greater precision without computing higher derivatives we will get the next approximate value of y from a recurrence formula that may involve not just one or two, but several of its predecessors. To get such a formula started we will have to find several starting values in addition to the one that is given in the statement of the initial-value problem.

To get back to the midpoint rule, we can get it started most easily by calculating y_1, the approximation to $y(x_0 + h)$, from Euler's method, and then switching to the midpoint rule to carry out the rest of the calculation.

Let's do this, for example with the same differential equation (2.1.7) that we used to illustrate Euler's rule, so we can compare the two methods. The problem consists of the equation $y' = 0.5y$ and the initial value $y(0) = 1$. We'll use the same step size $h = 0.05$ as before.

Now to start the midpoint rule we need two consecutive values of y, in this case at $x = 0$ and $x = 0.05$. At 0.05 we use the value that Euler's method gives us, namely $y_1 = 1.025$ (see Table 1). It's easy to continue the calculation now from (2.5.5).

For instance

$$\begin{aligned} y_2 &= y_0 + 2h(0.5y_1) \\ &= 1 + 0.1(0.5 * 1.025) \\ &= 1.05125 \end{aligned} \tag{2.5.6}$$

and

$$\begin{aligned} y_3 &= y_1 + 2h(0.5y_2) \\ &= 1.025 + 0.1(0.5 * 1.05125) \\ &= 1.0775625\,. \end{aligned} \tag{2.5.7}$$

In the table below we show for each x the value computed from the midpoint rule, from Euler's method, and from the exact solution $y(x) = e^{x/2}$. The superior accuracy of the midpoint rule is apparent.

x	Midpoint(x)	Euler(x)	Exact(x)
0.00	1.00000	1.00000	1.00000
0.05	1.02500	1.02500	1.02532
0.10	1.05125	1.05063	1.05127
0.15	1.07756	1.07689	1.07788
0.20	1.10513	1.10381	1.10517
0.25	1.13282	1.13141	1.13315
⋮	⋮	⋮	⋮
1.00	1.64847	1.63862	1.64872
2.00	2.71763	2.68506	2.71828
3.00	4.48032	4.39979	4.48169
⋮	⋮	⋮	⋮
5.00	12.17743	11.81372	12.18249
10.00	148.31274	139.56389	148.41316

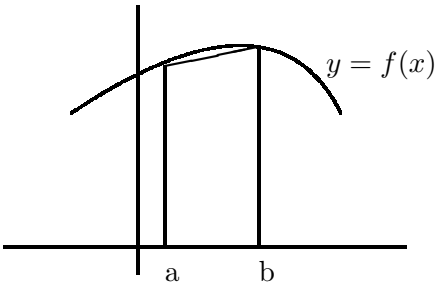

Figure 2.1: The trapezoidal rule

TABLE 2

Next, we introduce a third method of numerical integration, the trapezoidal rule. The best way to obtain it is to convert the differential equation that we're trying to solve into an integral equation, and then use the trapezoidal approximation for the integral.

We begin with the differential equation $y' = f(x, y(x))$, and we integrate both sides from x to $x + h$, getting

$$y(x + h) = y(x) + \int_x^{x+h} f(t, y(t))\, dt. \tag{2.5.8}$$

Now if we approximate the right-hand side in any way by a weighted sum of values of the integrand at various points we will have found an approximate method for solving our differential equation.

The trapezoidal rule states that for an approximate value of an integral

$$\int_a^b f(t)\, dt \tag{2.5.9}$$

we can use, instead of the area under the curve between $x = a$ and $x = b$, the area of the trapezoid whose sides are the x axis, the lines $x = a$ and $x = b$, and the line through the points $(a, f(a))$ and $(b, f(b))$, as shown in Figure 2.1. That area is $\frac{1}{2}(f(a) + f(b))(b - a)$.

If we apply the trapezoidal rule to the integral that appears in (2.5.8), we obtain

$$y(x_n + h) \approx y(x_n) + \frac{h}{2}(f(x_n, y(x_n)) + f(x_n + h, y(x_n + h))) \tag{2.5.10}$$

in which we have used the "≈" sign rather than the "=" because the right hand side is not exactly equal to the integral that really belongs there, but is only approximately so.

If we use our usual abbreviation y_n for the computed approximate value of $y(x_n)$, then (2.5.10) becomes

$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1})). \tag{2.5.11}$$

This is the trapezoidal rule in the form that is useful for differential equations.

At first sight, (2.5.11) looks like a recurrence formula from which the next approximate value, y_{n+1}, of the unknown function, can immediately be computed from the previous value, y_n. However, this is not the case.

Upon closer examination one observes that the next value y_{n+1} appears not only on the left-hand side, but also on the right (it's hiding in the second f on the right side).

In order to find the value y_{n+1} it appears that we need to carry out an iterative process. First we would guess y_{n+1} (guessing y_{n+1} to be equal to y_n wouldn't be all that bad, but we can do better). If we use this guess value on the right side of (2.5.11) then we would be able to calculate the entire right-hand side, and then we could use that value as a new "improved" value of y_{n+1}.

Now if the new value agrees with the old sufficiently well the iteration would halt, and we would have found the desired value of y_{n+1}. Otherwise we would use the improved value on the right side just as we previously used the first guess. Then we would have a "more improved" guess, etc.

Fortunately, in actual use, it turns out that one does not actually have to iterate to convergence. If a good enough guess is available for the unknown value, then just one refinement by a single application of the trapezoidal formula is sufficient. This is not the case if a high quality guess is unavailable. We will discuss this point in more detail in section 2.9. The pair of formulas, one of which supplies a very good guess to the next value of y, and the other of which refines it to a better guess, is called a *predictor-corrector pair*, and such pairs form the basis of many of the highly accurate schemes that are used in practice.

As a numerical example, take the differential equation

$$y' = 2xe^y + 1 \tag{2.5.12}$$

with the initial value $y(0) = 1$. If we use $h = 0.05$, then our first task is to calculate y_1, the approximate value of $y(0.05)$. The trapezoidal rule asserts that

$$y_1 = 1 + 0.025(2 + 0.1e^{y_1}) \tag{2.5.13}$$

and sure enough, the unknown number y_1 appears on both sides.

Let's guess $y_1 = 1$. Since this is not a very inspired guess, we will have to iterate the trapezoidal rule to convergence. Hence, we use this guess on the right side of (2.5.13), compute the right side, and obtain $y_1 = 1.056796$. If we use this new guess the same way, the result is $y_1 = 1.057193$. Then we get 1.057196, and since this is in sufficiently close agreement with the previous result, we declare that the iteration has converged. Then we take $y_1 = 1.057196$ for the computed value of the unknown function at $x = 0.05$, and we go next to $x = 0.1$ to repeat the same sort of thing to get y_2, the computed approximation to $y(0.1)$.

In Table 3 we show the results of using the trapezoidal rule (where we have iterated until two successive guesses are within 10^{-4}) on our test equation $y' = 0.5y,\ y(0) = 1$ as the column Trap(x). For comparison, we show Midpoint(x) and Exact(x).

x	Trap(x)	Midpoint(x)	Exact(x)
0.00	1.00000	1.00000	1.00000
0.05	1.02532	1.02500	1.02532
0.10	1.05127	1.05125	1.05127
0.15	1.07789	1.07756	1.07788
0.20	1.10518	1.10513	1.10517
0.25	1.13316	1.13282	1.13315
⋮	⋮	⋮	⋮
1.00	1.64876	1.64847	1.64872
2.00	2.71842	2.71763	2.71828
3.00	4.48203	4.48032	4.48169
⋮	⋮	⋮	⋮
5.00	12.18402	12.17743	12.18249
10.00	148.45089	148.31274	148.41316

TABLE 3

2.6 Comparison of the methods

We are now in possession of three methods for the numerical solution of differential equations. They are Euler's method

$$y_{n+1} = y_n + hy'_n\,, \tag{2.6.1}$$

the trapezoidal rule

$$y_{n+1} = y_n + \frac{h}{2}(y'_n + y'_{n+1}), \tag{2.6.2}$$

and the midpoint rule

$$y_{n+1} = y_{n-1} + 2hy'_n\,. \tag{2.6.3}$$

In order to compare the performance of the three techniques it will be helpful to have a standard differential equation on which to test them. The most natural candidate for such an equation is $y' = Ay$, where A is constant. The reasons for this choice are first that the equation is easy to solve exactly, second that the difference approximations are also relatively easy to solve exactly, so comparison is readily done, third that by varying the sign of A we can study behavior of either growing or shrinking (stable or unstable) solutions, and finally that many problems in nature have solutions that are indeed exponential, at least over the short term, so this is an important class of differential equations.

We will, however, write the test equation in a slightly different form for expository reasons, namely as

$$y' = -\frac{y}{L}\,; \qquad y(0) = 1\,; \qquad L > 0 \tag{2.6.4}$$

where L is a constant. The most interesting and revealing case is where the true solution is a decaying exponential, so we will assume that $L > 0$. Further, we will assume that $y(0) = 1$ is the given initial value.

The exact solution is of course

$$\text{Exact}(x) = e^{-x/L}. \tag{2.6.5}$$

Notice that if x increases by L, the solution changes by a factor of e. Hence L, called the *relaxation length* of the problem, can be conveniently visualized as the distance over which the solution falls by a factor of e.

Now we would like to know how well each of the methods (2.6.1)–(2.6.3) handles the problem (2.6.4).

Suppose first that we ask Euler's method to solve the problem. If we substitute $y' = f(x,y) = -y/L$ into (2.6.1), we get

$$\begin{aligned} y_{n+1} &= y_n + h * \left(-\frac{y_n}{L}\right) \\ &= \left(1 - \frac{h}{L}\right) y_n \,. \end{aligned} \tag{2.6.6}$$

Before we solve this recurrence, let's comment on the ratio h/L that appears in it. Now L is the distance over which the solution changes by a factor of e, and h is the step size that we are going to use in the numerical integration. Instinctively, one feels that if the solution is changing rapidly in a certain region, then h will have to be kept small there if good accuracy is to be retained, while if the solution changes only slowly, then h can be larger without sacrificing too much accuracy. The ratio h/L measures the step size of the integration in relation to the distance over which the solution changes appreciably. Hence, h/L is exactly the thing that one feels should be kept small for a successful numerical solution.

Since h/L occurs frequently below, we will denote it with the symbol τ.

Now the solution of the recurrence equation (2.6.6), with the starting value $y_0 = 1$, is obviously

$$y_n = (1-\tau)^n \qquad n = 0, 1, 2, \ldots . \tag{2.6.7}$$

Next we study the trapezoidal approximation to the same equation (2.6.4). We substitute $y' = f(x,y) = -y/L$ in (2.6.2) and get

$$y_{n+1} = y_n + \frac{h}{2}\left(-\frac{y_n}{L} - \frac{y_{n+1}}{L}\right). \tag{2.6.8}$$

The unknown y_{n+1} appears, as usual with this method, on both sides of the equation. However, for the particularly simple equation that we are now studying, there is no difficulty in solving (2.6.8) for y_{n+1} (without any need for an iterative process) and obtaining

$$y_{n+1} = \frac{1 - \frac{\tau}{2}}{1 + \frac{\tau}{2}}\, y_n \,. \tag{2.6.9}$$

Together with the initial value $y_0 = 1$, this implies that

$$y_n = \left(\frac{1 - \frac{\tau}{2}}{1 + \frac{\tau}{2}}\right)^n \qquad n = 0, 1, 2, \ldots . \tag{2.6.10}$$

Before we deal with the midpoint rule, let's pause to examine the two methods whose solutions we have just found. Note that for a given value of h, all three of (a) the exact solution, (b) Euler's solution and (c) the trapezoidal solution are of the form $y_n = (\text{constant})^n$, in which the three values of "constant" are

$$\begin{array}{ll} \text{(a)} & e^{-\tau} \\ \text{(b)} & 1-\tau \\ \text{(c)} & \frac{1-\frac{\tau}{2}}{1+\frac{\tau}{2}}. \end{array} \tag{2.6.11}$$

It follows that to compare the two approximate methods with the "truth," all we have to do is see how close the constants (b) and (c) above are to the true constant (a). If we remember that τ is being thought of as small compared to 1, then we have the power series expansion of $e^{-\tau}$

$$e^{-\tau} = 1 - \tau + \frac{\tau^2}{2} - \frac{\tau^3}{6} + \cdots \tag{2.6.12}$$

to compare with $1-\tau$ and with the power series expansion of

$$\frac{1-\frac{\tau}{2}}{1+\frac{\tau}{2}} = 1 - \tau + \frac{\tau^2}{2} - \frac{\tau^3}{4} + \cdots. \tag{2.6.13}$$

The comparison is now clear. Both the Euler and the trapezoidal methods yield approximate solutions of the form $(\text{constant})^n$, where "constant" is near $e^{-\tau}$. The trapezoidal rule does a better job of being near $e^{-\tau}$ because its constant agrees with the power series expansion of $e^{-\tau}$ through the quadratic term, whereas that of the Euler method agrees only up to the linear term.

Finally we study the nature of the approximation that is provided by the midpoint rule. We will find that a new and important phenomenon rears its head in this case. The analysis begins just as it did in the previous two cases: We substitute the right-hand side $f(x,y) = -y/L$ for y' in (2.6.3) to get

$$y_{n+1} = y_{n-1} + 2h * \left(-\frac{y_n}{L}\right). \tag{2.6.14}$$

One important feature is already apparent. Instead of facing a first-order difference equation as we did in (2.6.6) for Euler's method and in (2.6.9) for the trapezoidal rule, we have now to contend with a second-order difference equation.

Since the equation is linear with constant coefficients, we know to try a solution of the form $y_n = r^n$. This leads to the quadratic equation

$$r^2 + 2\tau r - 1 = 0. \tag{2.6.15}$$

Evidently the discriminant of this equation is positive, so its roots are distinct. If we denote these two roots by $r_+(\tau)$ and $r_-(\tau)$, then the general solution of the difference equation (2.6.14) is

$$y_n = c\,(r_+(\tau))^n + d\,(r_-(\tau))^n\ , \tag{2.6.16}$$

where c and d are constants whose values are determined by the initial data y_0 and y_1.

The Euler and trapezoidal approximations were each of the form (constant)^n. This one is a sum of two terms of that kind. We will see that $r_+(\tau)$ is a very good approximation to $e^{-\tau}$. The other term, $(r_-(\tau))^n$ is, so to speak, the price that we pay for getting such a good approximation in $r_+(\tau)$. We hope that the other term will stay small relative to the first term, so as not to disturb the closeness of the approximation. We will see, however, that it need not be so obliging, and that in fact it might do whatever it can to spoil things.

The two roots of the quadratic equation are

$$\begin{aligned} r_+(\tau) &= -\tau + \sqrt{1+\tau^2} \\ r_-(\tau) &= -\tau - \sqrt{1+\tau^2}. \end{aligned} \tag{2.6.17}$$

When $\tau = 0$ the first of these is $+1$, so when τ is small $r_+(\tau)$ is near $+1$, and it is the root that is trying to approximate the exact constant $e^{-\tau}$ as well as possible. In fact it does pretty well, because the power series expansion of $r_+(\tau)$ is

$$r_+(\tau) = 1 - \tau + \frac{\tau^2}{2} - \frac{\tau^4}{8} + \cdots \tag{2.6.18}$$

so it agrees with $e^{-\tau}$ through the quadratic terms.

What about $r_-(\tau)$? Its Taylor series is

$$r_-(\tau) = -1 - \tau - \frac{\tau^2}{2} + \cdots. \tag{2.6.19}$$

The bad news is now before us: When τ is a small positive number, the root $r_-(\tau)$ is larger than 1 in absolute value. This means that the stability criterion of Theorem 1.6.1 is violated, so we say that the midpoint rule is unstable.

In practical terms, we observe that $r_+(\tau)$ is close to $e^{-\tau}$, so the first term on the right of (2.6.16) is very close to the exact solution. The second term of (2.6.16), the so-called *parasitic solution*, is small compared to the first term when n is small, because the constant d will be small compared with c. However, as we move to the right, n increases, and the second term will eventually dominate the first, because the first term is shrinking to zero as n increases, because that's what the exact solution does, while the second term increases steadily in size. In fact, since $r_-(\tau)$ is negative and larger than 1 in absolute value, the second term will alternate in sign as n increases, and grow without bound in magnitude.

In Table 4 below we show the result of integrating the problem $y' = -y$, $y(0) = 1$ with each of the three methods that we have discussed, using a step size of $h = 0.05$. To get the midpoint method started, we used the exact value of $y(0.05)$ (*i.e.*, we cheated), and in the trapezoidal rule we iterated to convergence with $\epsilon = 10^{-4}$. The instability of the midpoint rule is quite apparent.

x	Euler(x)	Trap(x)	Midpoint(x)	Exact(x)
0.0	1.00000	1.00000	1.00000	1.00000
1.0	0.35849	0.36780	0.36806	0.36788
2.0	0.12851	0.13527	0.13552	0.13534
3.0	0.04607	0.04975	0.05005	0.04979
4.0	0.01652	0.01830	0.01888	0.01832
5.0	0.00592	0.00673	0.00822	0.00674
10.0	0.00004	0.00005	0.21688	0.00005
14.55	3.3×10^{-7}	4.8×10^{-7}	-20.48	4.8×10^{-7}
15.8	9.1×10^{-8}	1.4×10^{-7}	71.45	1.4×10^{-7}

TABLE 4

In addition to the above discussion of accuracy, we summarize here there additional properties of integration methods as they relate to the examples that we have already studied.

First, a numerical integration method might be *iterative* or *noniterative*. A method is noniterative if it expresses the next value of the unknown function quite explicitly in terms of values of the function and its derivatives at preceding points. In an iterative method, at each step of the solution process the next value of the unknown is defined implicitly by an equation, which must be solved to obtain the next value of the unknown function. In practice, we may either solve this equation completely by an iteration or do just one step of the iteration, depending on the quality of available estimates for the unknown value.

Second, a method is *self-starting* if the next value of the unknown function is obtained from the values of the function and its derivatives at exactly one previous point. It is not self-starting if values at more than one backward point are needed to get the next one. In the latter case some other method will have to be used to get the first few computed values.

Third, we can define a numerical method to be *stable* if when it is applied to the equation $y' = -y/L$, where $L > 0$, then for all sufficiently small positive values of the step size h a stable difference equation results, *i.e.*, the computed solution (neglecting roundoff) remains bounded as $n \to \infty$.

We summarize below the properties of the three methods that we have been studying.

	Euler	Midpoint	Trapezoidal
Iterative	No	No	Yes
Self-starting	Yes	No	Yes
Stable	Yes	No	Yes

Of the three methods, the trapezoidal rule is clearly the best, though for efficient use it needs the help of some other formula to predict the next value of y and thereby avoid lengthy iterations.

2.7 Predictor-corrector methods

The trapezoidal rule differs from the other two that we've looked at in that it does not explicitly tell us what the next value of the unknown function is, but instead gives us an equation that must be solved in order to find it. At first sight this seems like a nuisance, but in fact it is a boon, because it enables us to regulate the step size during the course of a calculation, as we will discuss in section 2.9.

Let's take a look at the process by which we refine a guessed value of y_{n+1} to an improved value, using the trapezoidal formula

$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1})). \tag{2.7.1}$$

Suppose we let $y_{n+1}^{(k)}$ represent some guess to the value of y_{n+1} that satisfies (2.7.1). Then the improved value $y_{n+1}^{(k+1)}$ is computed from

$$y_{n+1}^{(k+1)} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1}^{(k)})). \tag{2.7.2}$$

We want to find out about how rapidly the successive values $y_{n+1}^{(k)}$, $k = 1, 2, \ldots$ approach a limit, if at all. To do this, we rewrite equation (2.7.2), this time replacing k by $k-1$ to obtain

$$y_{n+1}^{(k)} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1}^{(k-1)})) \tag{2.7.3}$$

and then subtract (2.7.3) from (2.7.2) to get

$$y_{n+1}^{(k+1)} - y_{n+1}^{(k)} = \frac{h}{2}(f(x_{n+1}, y_{n+1}^{(k)}) - f(x_{n+1}, y_{n+1}^{(k-1)})). \tag{2.7.4}$$

Next we use the mean-value theorem on the difference of f values on the right-hand side, yielding

$$y_{n+1}^{(k+1)} - y_{n+1}^{(k)} = \frac{h}{2}\left.\frac{\partial f}{\partial y}\right|_{(x_{n+1},\eta)} \left(y_{n+1}^{(k)} - y_{n+1}^{(k-1)}\right), \tag{2.7.5}$$

where η lies between $y_{n+1}^{(k)}$ and $y_{n+1}^{(k-1)}$.

From the above we see at once that the difference between two consecutive iterated values of y_{n+1} will be $\frac{h}{2}\frac{\partial f}{\partial y}$ times the difference between the previous two iterated values.

It follws that the iterative process will converge if h is kept small enough so that $\frac{h}{2}\frac{\partial f}{\partial y}$ is less than 1 in absolute value. We refer to $\frac{h}{2}\frac{\partial f}{\partial y}$ as the *local convergence factor* of the trapezoidal rule.

If the factor is a lot less than 1 (and this can be assured by keeping h small enough), then the convergence will be extremely rapid.

In actual practice, one uses an iterative formula together with another formula (the predictor) whose mission is to provide an intelligent first guess for the iterative method

to use. The predictor formula will be explicit, or noniterative. If the predictor formula is clever enough, then it will happen that just a single application of the iterative refinement (corrector formula) will be sufficient, and we won't have to get involved in a long convergence process.

If we use the trapezoidal rule for a corrector, for instance, then a clever predictor would be the midpoint rule. The reason for this will become clear if we look at both formulas together with their error terms. We will see in the next section that the error terms are as follows:

$$y_{n+1} = y_{n-1} + 2hy'_n + \frac{h^3}{3}y'''(X_m) \tag{2.7.6}$$

$$y_{n+1} = y_n + \frac{h}{2}(y'_n + y'_{n+1}) - \frac{h^3}{12}y'''(X_t). \tag{2.7.7}$$

Now the exact locations of the points X_m and X_t are unknown, but we will assume here that h is small enough that we can regard the two values of y''' that appear as being the same.

As far as the powers of h that appear in the error terms go, we see that the third power occurs in both formulas. We say then, that the midpoint predictor and the trapezoidal corrector constitute a *matched pair*. The error in the trapezoidal rule is about one fourth as large as, and of opposite sign from, the error in the midpoint method.

The midpont guess is therefore quite "intelligent". The subsequent iterative refinement of that guess needs to reduce the error only by a factor of four. Now let y_P denote the midpoint predicted value, $y^{(1)}_{n+1}$ denote the first refined value, and y_{n+1} be the final converged value given by the trapezoidal rule. Then we have

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{2}\left(y'_n + f(x_{n+1}, y_{n+1})\right) \\ y^{(1)}_{n+1} &= y_n + \frac{h}{2}\left(y'_n + f(x_{n+1}, y_P)\right) \end{aligned} \tag{2.7.8}$$

and by subtraction

$$y^{(1)}_{n+1} - y_{n+1} = \frac{h}{2}\frac{\partial f}{\partial y}\left(y_P - y_{n+1}\right). \tag{2.7.9}$$

This shows that, however far from the converged value the first guess was, the refined value is $\frac{h}{2}\frac{\partial f}{\partial y}$ times closer. Hence if we can keep $\frac{h}{2}\frac{\partial f}{\partial y}$ no bigger than about 1/4, then the distance from the first refined value to the converged value will be no larger than the size of the error term in the method, so there would be little point in gilding the iteration any further.

The conclusion is that when we are dealing with a matched predictor-corrector pair, we need do only a single refinement of the corrector if the step size is kept moderately small. Furthermore, "moderately small" means that the step size times the local value of $\frac{\partial f}{\partial y}$ should be small compared to 1. For this reason, iteration to full convergence is rarely done in practice.

2.8 Truncation error and step size

We have so far regarded the step size h as a silent partner, more often than not choosing it to be equal to 0.05, for no particular reason. It is evident, however, that the accuracy of the calculation is strongly affected by the step size. If h is chosen too large, the computed solution may be quite far from the true solution of the differential equation, if too small then the calculation will become unnecessarily time-consuming, and roundoff errors may build up excessively because of the numerous arithmetic operations that are being carried out.

Speaking in quite general terms, if the true solution of the differential equation is rapidly changing, then we will need a small values of h, that is, small compared to the local relaxation length (see p. 44), and if the solution changes slowly, then a larger value of h will do.

Frequently in practice we deal with equations whose solutions change very rapidly over part of the range of integration and slowly over another part. Examples of this are provided by the study of the switching on of a complicated process, such as beginning a multi-stage chemical reaction, turning on a piece of electronic equipment, starting a power reactor, etc. In such cases there usually are rapid and ephemeral or "transient" phenomena that occur soon after startup, and that disappear quickly. If we want to follow these transients accurately, we may need to choose a very tiny step size. After the transients die out, however, the steady-state solution may be a very quiet, slowly varying or nearly constant function, and then a much larger value of h will be adequate.

If we are going to develop software that will be satisfactory for such problems, then the program will obviously have to choose, and re-choose its own step size as the calculation proceeds. While following a rapid transient it should use a small mesh size, then it should gradually increase h as the transient fades, use a large step while the solution is steady, decreae it again if further quick changes appear, and so forth, all without operator intervention.

Before we go ahead to discuss methods for achieving this step size control, let's observe that one technique is already available in the material of the previous section. Recall that if we want to, we can implement the trapezoidal rule by first guessing, or predicting, the unknown at the next point by using Euler's formula, and then correcting the guess to complete convergence by iteration.

The first guess will be relatively far away from the final converged value if the solution is rapidly varying, but if the solution is slowly varying, then the guess will be rather good. It follows that the number of iterations required to produce convergence is one measure of the appropriateness of the current value of the step size: if many iterations are needed, then the step size is too big. Hence one way to get some control on h is to follow a policy of cutting the step size in half whenever more than, say, one or two iterations are necessary.

This suggestion is not sufficiently sensitive to allow doubling the stepsize when only one iteration is needed, however, and somewhat more delicacy is called for in that situation. Furthermore this is a very time-consuming approach since it involves a complete iteration to convergence, when in fact a single turn of the crank is enough if the step size is kept

small enough.

The discussion does, however, point to the fundamental idea that underlies the automatic control of step size during the integration. That basic idea is precisely that we can estimate the correctness of the step size by whatching how well the first guess in our iterative process agrees with the corrected value. The correction process itself, when viewed this way, is seen to be a powerful ally of the software user, rather than the "pain in the neck" it seemed to be when we first met it.

Indeed, why would anyone use the cumbersome procedure of guessing and refining (*i.e.*, prediction and correction) as we do in the trapezoidal rule, when many other methods are available that give the next value of the unknown immediately and explicitly? No doubt the question crossed the reader's mind, and the answer is now emerging. It will appear that not only does the disparity between the first prediction and the corrected value help us to control the step size, it actually can give us a quantitative estimate of the local error in the integration, so that if we want to, we can print out the approximate size of the error along with the solution.

Our next job will be to make these rather qualitative remarks into quantitative tools, so we must discuss the estimation of the error that we commit by using a particular difference approximation to a differential equation, instead of that equation itself, on one step of the integration process. This is the *single-step truncation error*. It does not tell us how far our computed solution is from the true solution, but only how much error is committed in a single step.

The easiest example, as usual, is Euler's method. In fact, in equation (2.1.2) we have already seen the single-step error of this metnod. That equation was

$$y(x_n + h) = y(x_n) + hy'(x_n) + h^2 \frac{y''(X)}{2} \tag{2.8.1}$$

where X lies between x_n and $x_n + h$. In Euler's procedure, we drop the third term on the right, the "remainder term," and compute the solution from the rest of the equation. In doing this we commit a single-step trunction error that is equal to

$$E = h^2 \frac{y''(X)}{2} \qquad x_n < X < x_n + h. \tag{2.8.2}$$

Thus, Euler's method is exact ($E = 0$) if the solution is a polynomial of degree 1 or less ($y'' = 0$). Otherwise, the single-step error is proportional to h^2, so if we cut the step size in half, the local error is reduced to 1/4 of its former value, approximately, and if we double h the error is multiplied by about 4.

We could use (2.8.2) to estimate the error by somehow computing an estimate of y''. For instance, we might differentiate the differential equation $y' = f(x, y)$ once more, and compute y'' directly from the resulting formula. This is usually more trouble than it is worth, though, and we will prefer to estimate E by more indirect methods.

Next we derive the local error of the trapezoidal rule. There are various special methods that might be used to do this, but instead we are going to use a very general method that

is capable of dealing with the error terms of almost every integration rule that we intend to study.

First, let's look a little more carefully at the capability of the trapezoidal rule, in the form

$$y_{n+1} - y_n - \frac{h}{2}(y'_n + y'_{n+1}) = 0. \tag{2.8.3}$$

Of course, this is a recurrence by meand of which we propagate the approximate solution to the right. It certainly is not exactly true if y_n denotes the value of the true solution at the point x_n unless that true solution is very special. How special?

Suppose the true solution is $y(x) = 1$ for all x. Then (2.8.3) would be exactly valid. Suppose $y(x) = x$. Then (2.8.3) is again exactly satisfied, as the reader should check. Furthermore, if $y(x) = x^2$, then a brief calculation reveals that (2.8.3) holds once more. How long does this continue? Our run of good luck has just expired, because if $y(x) = x^3$ then (check this) the left side of (2.8.3) is not 0, but is instead $-h^3/2$.

We might say that the trapezoidal rule is exact on 1, x, and x^2, but not x^3, *i.e.*, that it is an integration rule of *order two* ("order" is an overworked word in differential equations). It follows by linearity that the rule is exact on any quadratic polynomial.

By way of contrast, it is easy to verify that Euler's method is exact for a linear function, but fails on x^2. Since the error term for Euler's method in (2.8.2) is of the form $\text{const} * h^2 * y''(X)$, it is perhaps reasonable to expect the error term for the trapezoidal rule to look like $\text{const} * h^3 * y'''(X)$.

Now we have to questions to handle, and they are respectively easy and hard:

(a) If the error term in the trapezoidal rule really is $\text{const} * h^3 * y'''(X)$, then what is "const"?

(b) Is it true that the error term is $\text{const} * h^3 * y'''(X)$?

We'll do the easy one first, anticipating that the answer to (b) is affirmative so the effort won't be wasted. If the error term is of the form stated, then the trapezoidal rule can be written as

$$y(x_h) - y(x) - \frac{h}{2}\left(y'(x+h) + y'(x)\right) = c * h^3 * y'''(X), \tag{2.8.4}$$

where X lies between x and $x + h$. To find c all we have to do is substitute $y(x) = x^3$ into (2.8.4) and we find at once that $c = -1/12$. The single-step truncation error of the trapezoidal rule would therefore be

$$E = -h^3 \frac{y'''(X)}{12} \qquad x < X < x + h. \tag{2.8.5}$$

Now let's see that question (b) has the answer "yes" so that (2.8.5) is really right.

To do this we start with a truncated Taylor series with the integral form of the remainder, rather than with the differential form of the remainder. In general the series is

$$y(x) = y(0) + xy'(0) + x^2 \frac{y''(0)}{2!} + \cdots + x^n \frac{y^{(n)}(0)}{n!} + R_n(x) \tag{2.8.6}$$

where

$$R_n(x) = \frac{1}{n!}\int_0^x (x-s)^n y^{(n-1)}(s)\,ds. \tag{2.8.7}$$

Indeed, one of the nice ways to prove Taylor's theorem begins with the right-hand side of (2.8.7), plucked from the blue, and then repeatedly integrates by parts, lowereing the order of the derivative of y and the power of $(x-s)$ until both reach zero.

In (2.8.6) we choose $n = 2$, because the trapezoidal rule is exact on polynomials of degree 2, and we write it in the form

$$y(x) = P_2(x) + R_2(x) \tag{2.8.8}$$

where $P_2(x)$ is the quadratic (in x) polynomial $P_2(x) = y(0) + xy'(0) + x^2y''(0)/2$.

Next we define a certain operation that transforms a function $y(x)$ into a new function, namely into the left-hand side of equation (2.8.4). We call the operator L so it is defined by

$$\mathrm{L}y(x) = y(x+h) - y(x) - \frac{h}{2}\left(y'(x) + y'(x+h)\right). \tag{2.8.9}$$

Now we apply the operator L to both sides of equation (2.8.8), and we notice immediately that $\mathrm{L}P_2(x) = 0$, because the rule is exact on quadratic polynomials (this is why we chose $n = 2$ in (2.8.6)). Hence we have

$$\mathrm{L}y(x) = \mathrm{L}R_2(x). \tag{2.8.10}$$

Notice that we have here a remainder formula for the trapezoidal rule. It isn't in a very satisfactory form yet, so we will now make it a bit more explicit by computing $\mathrm{L}R_2(x)$. First, in the integral expression (2.8.7) for $R_2(x)$ we want to replace the upper limit of the integral by $+\infty$. We can do this by writing

$$R_2(x) = \frac{1}{2!}\int_0^\infty H_2(x-s)y'''(s)\,ds \tag{2.8.11}$$

where $H_2(t) = t^2$ if $t > 0$ and $H_2(t) = 0$ if $t < 0$.

Now if we bear in mind the fact that the operator L acts only on x, and that s is a dummy variable of integration, we find that

$$\mathrm{L}R_2(x) = \frac{1}{2!}\int_0^\infty \mathrm{L}H_2(x-s)y'''(s)\,ds. \tag{2.8.12}$$

Choose $x = h$. Then if s lies between 0 and h we find

$$\begin{aligned}\mathrm{L}H_2(x-s) &= (h-s)^2 - \frac{h}{2}\left(2(h-s)\right)\\ &= -s(h-s\end{aligned} \tag{2.8.13}$$

(*Caution:* Do not read past the right-hand side of the first equals sign unless you can verify the correctness of what you see there!), whereas if $s > h$ then $\mathrm{L}H_2(x-s) = 0$.

Then (2.8.12) becomes

$$\mathrm{L}R_2(h) = -\frac{1}{2}\int_0^h s(h-s)y'''(s)\,ds. \tag{2.8.14}$$

This is a much better form for the remainder, but we still do not have the "hard" question (b). To finish it off we need a form of the mean-value theorem of integral calculus, namely

Theorem 2.8.1 *If $p(x)$ is nonnegative, and $g(x)$ is continuous, then*

$$\int_a^b p(x)g(x)\,dx = g(X)\int_a^b p(x)\,dx \tag{2.8.15}$$

where X lies between a and b.

The theorem asserts that a weighted average of the values of a continuous function is itself one of the values of that function. The vital hypothesis is that the "weight" $p(x)$ does not change sign.

Now in (2.8.14), the function $s(h-s)$ does not change sign on the s-interval $(0,h)$, so

$$\mathrm{L}R_2(h) = -\frac{1}{2}y'''(X)\int_0^h s(h-s)\,ds \tag{2.8.16}$$

and if we do the integral we obtain, finally,

Theorem 2.8.2 *The trapezoidal rule with remainder term is given by the formula*

$$y(x_{n+1}) - y(x_n) = \frac{h}{2}\left(y'(x_n) + y'(x_{n+1})\right) - \frac{h^3}{12}y'''(X), \tag{2.8.17}$$

where X lies between x_n and x_{n+1}.

The proof of this theorem involved some ideas tha carry over almost unchanged to very general kinds of integration rules. Therefore it is important to make sure that you completely understand its derivation.

2.9 Controlling the step size

In equation (2.8.5) we saw that if we can estimate the size of the third derivative during the calculation, then we can estimate the error in the trapezoidal rule as we go along, and modify the step size h if necessary, to keep that error within preassigned bounds.

To see how this can be done, we will quote, without proof, the result of a similar derivation for the midpoint rule. It says that

$$y(x_{n+1}) - y(x_{n-1}) = 2hy'(x_n) + \frac{h^3}{3}y'''(X), \tag{2.9.1}$$

where X is between x_{n-1} and x_{n+1}. Thus the midpoint rule is also of second order. If the step size were halved, the local error would be cut to one eighth of its former value. The error in the midpoint rule is, however, about four times as large as that in the trapezoidal formula, and of opposite sign.

Now suppose we adopt an overall strategy of predicting the value y_{n+1} of the unknown by means of the midpoint rule, and then refining the prediction to convergence with the trapezoidal corrector. We want to estiamte the size of the single-step truncation error, using only the following data, both of which are available during the calculation: (a) the initial guess, from the midpoint method, and (b) the converged corrected value, from the trapezoidal rule.

We begin by defining three different kinds of values of the unknown function at the "next" point x_{n+1}. They are

(i) the quantity p_{n+1} is defined as the predicted value of $y(x_{n+1})$ obtained from using the midpoint rule, except that backwards values are not the computed ones, but are the exact ones instead. In symbols,

$$p_{n+1} = y(x_{n+1}) + 2hy'(x_n). \tag{2.9.2}$$

Of course, p_{n+1} is not available during an actual computation.

(ii) the quantity q_{n+1} is the value that we would compute from the trapezoidal corrector if for the backward value we use the exact solution $y(x_n)$ instead of the calculated solution y_n. Thus q_{n+1} satisfies the equation

$$q_{n+1} = y(x_n) + \frac{h}{2}\left(f(x_n, y(x_n)) + f(x_{n+1}, q_{n+1})\right). \tag{2.9.3}$$

Again, q_{n+1} is not available to us during calculation.

(iii) the quantity $y(x_{n+1})$, which is the exact solution itself. It staisfies two different equations, one of which is

$$y(x_{n+1}) = y(x_n) + \frac{h}{2}\left(f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))\right) - \frac{h^3}{12}y'''(X) \tag{2.9.4}$$

and the other of which is (2.9.1). Note that the two X's may be different.

Now, from (2.9.1) and (2.9.2) we have at once that

$$y(x_{n+1}) = p_{n+1} + \frac{h^3}{3}y'''(X). \tag{2.9.5}$$

Next, from (2.9.3) and (2.9.4) we get

$$\begin{aligned} y(x_{n+1}) &= \frac{h}{2}\left(f(x_{n+1}, y(x_{n+1})) - f(x_{n+1}, q_{n+1})\right) - \frac{h^3}{12}y'''(X) \\ &= q_{n+1}\frac{h}{2}(y(x_{n+1}) - q_{n+1})\frac{\partial f}{\partial y}(x_{n+1}, Y) - \frac{h^3}{12}y'''(X \end{aligned} \tag{2.9.6}$$

where we have used the mean-value theorem, and Y lies between $y(x_{n+1})$ and q_{n+1}. Now if we subtract q_{n+1} from both sides, we will observe that $y(x_{n+1}) - q_{n+1}$ will then appear on both sides of the equation. Hence we will be able to solve for it, with the result that

$$y(x_{n+1}) = q_{n+1} - \frac{h^3}{12} y'''(X) + \text{ terms involving } h^4. \tag{2.9.7}$$

Now let's make the working hypothesis that y''' is constant over the range of values of x considered, namely from $x_n - h$ to $x_n + h$. The $y'''(X)$ in (2.9.7) is thereby decreed to be equal to the $y'''(X)$ in (2.9.5), even though the X's are different. Under this assumption, we can eliminate $y(x_{n+1})$ between (2.9.5) and (2.9.7) and obtain

$$q_{n+1} - p_{n+1} = \frac{5}{12} h^3 y''' + \text{ terms involving } h^4. \tag{2.9.8}$$

We see that this expresses the unknown, but assumed constant, value of y''' in terms of the difference between the initial prediction and the final converged value of $y(x_{n+1})$. Now we ignore the "terms involving h^4" in (2.9.8), solve for y''', and then for the estimated single-step truncation error we have

$$\begin{aligned} \text{Error} &= -\frac{h^3}{12} y''' \\ &\approx -\frac{1}{12}\frac{12}{5}\left(q_{n+1} - p_{n+1}\right) \\ &= -\frac{1}{5}(q_{n+1} - p_{n+1}). \end{aligned} \tag{2.9.9}$$

The quantity $q_{n+1} - p_{n+1}$ is not available during the calculation, but as an estimator we can use the compted predicted value and the compted converged value, because these differ only in that they use computed, rather than exact backwards values of the unknown function.

Hence, we have here an estimate of the single-step trunction error that we can conveniently compute, print out, or use to control the step size.

The derivation of this formula was of course dependent on the fact that we used the midpoint metnod for the predoctor and the trapezoidal rule for the corrector. If we had used a different pair, however, the same argument would have worked, provided only that the error terms of the predictor and corrector formulas both involved the same derivative of y, *i.e.*, both formulas were of the same order.

Hence, "matched pairs"" of predictor and corrector formulas, *i.e.*, pairs in which both are of the same order, are most useful for carrying out extended calculations in which the local errors are continually monitored and the step size is regulated accordingly.

Let's pause to see how this error estimator would have turned out in the case of a general matched pair of predictor-corrector formulas, insted of just for the midpoint and trapezoidal rule combination. Suppose the predictor formula has an error term

$$y_{exact} - y_{predicted} = \lambda h^q y^{(q)}(X) \tag{2.9.10}$$

and suppose that the error in the corrector formula is given by

$$y_{exact} - y_{corrected} = \mu h^q y^{(q)}(X). \tag{2.9.11}$$

Then a derivation similar to the one that we have just done will show that the estimator for the single-step error that is available during the progress of the computation is

$$\text{Error} \approx \frac{\mu}{\lambda - \mu}(y_{predicted} - y_{corrected}). \tag{2.9.12}$$

In the table below we show the result of integrating the differential equation $y' = -y$ with $y(0) = 1$ using the midpoint and trapezoidal formulas with $h = 0.05$ as the predictor and corrector, as described above. The successive columns show x, the predicted value at x, the converged corrected value at x, the single-step error estimated from the approximation (2.9.9), and the actual single-step error obtained by computing

$$y(x_{n+1}) - y(x_n) - \frac{h}{2}(y'(x_n) + y'(x_{n+1})) \tag{2.9.13}$$

using the true solution $y(x) = e^{-x}$. The calculation was started by (cheating and) using the exact solution at 0.00 and 0.05.

x	Pred(x)	Corr(x)	Errest(x)	Error(x)
0.00	——-	——-	——-	——-
0.05	——-	——-	——-	——-
0.10	0.904877	0.904828	98×10^{-7}	94×10^{-7}
0.15	0.860747	0.860690	113×10^{-7}	85×10^{-7}
0.20	0.818759	0.818705	108×10^{-7}	77×10^{-7}
0.25	0.778820	0.778768	102×10^{-7}	69×10^{-7}
0.30	0.740828	0.740780	97×10^{-7}	61×10^{-7}
0.35	0.704690	0.704644	93×10^{-7}	55×10^{-7}
0.40	0.670315	0.670271	88×10^{-7}	48×10^{-7}
0.45	0.637617	0.637575	84×10^{-7}	43×10^{-7}
0.50	0.606514	0.606474	80×10^{-7}	37×10^{-7}
⋮	⋮	⋮	⋮	⋮
0.95	0.386694	0.386669	51×10^{-7}	5×10^{-7}
1.00	0.367831	0.367807	48×10^{-7}	3×10^{-7}

TABLE 5

Now that we have a simple device for estimating the single-step truncation error, namely by using one fifth of the distance between the first guess and the corrected value, we can regulate the step size so as to keep the error between preset limits. Suppose we would like to keep the single-step error in the neighborhood of 10^{-8}. We might then adopt, say 5×10^{-8} as the upper limit of tolerable error and, for instance, 10^{-9} as the lower limit.

Why should we have a lower limit? If the calculation is being done with more precision than necessary, the step size will be smaller than needed, and we will be wasting computer time as well as possibly building up roundoff error.

Now that we have fixed these limits we should be under no delusion that our computed solution will depart from the true solution by no more than 5×10^{-8}, or whatever. What we are controlling is the one-step truncation error, a lot bettern than nothing, but certainly not the same as the total accumulated error.

With the upper and lower tolerances set, we embark on the computation. First, since the midpoint method needs two backward values before it can be used, something special will have to be done to get the procedure moving at the start. This is typical of numerical solution of differential equations. Special procedures are needed at the beginning to build up enough computed values so that the predictor-corrector formulas can be used thereafter, or at least until the step size is changed.

In the present case, since we're going to use the trapezoidal corrector anyway, we might as well use the trapezoidal rule, unassisted by midpoint, with complete iteration to convergence, to get the value of y at the first point $x_0 + h$ beyond the initial point x_0.

Now we have two consecutive values of y, and the generic calculation can begin. From any two consecutive values, the midpoint rule is used to predict the next value of y. This predicted value is also saved for future use in error estimation. The predicted value is then refined by the trapezoidal rule.

With the trapezoidal value in hand, the local error is then estimated by calculating one-fifth of the difference between that value and the midpoint guess.

If the absolute value of the local error lies between the preset limits 10^{-9} and 5×10^{-8}, then we just go on to the next step. This means that we augment x by h, and move back the newer values of the unknown function to the locations that hold older values (we remember, at any moment, just two past values).

Otherwise, suppose the local error was too large. Then we must reduce the step size h, say by cutting it in half. When all this is done, some message should be printed out that announces the change, and then we should restart the procedure, with the new value of h, from the "farthest backward" value of x for which we still have the corresponding y in memory. One reason for this is that we may find out right at the outset that our very first choice of h is too large, and perhaps it may need to be halved, say, three times before the errors are tolerable. Then we would like to restart each time from the same originally given data point x_0, rather than let the computation creep forward a little bit with step sizes that are too large for comfort.

Finally, suppose the local error was too small. Then we double the step size, print a message to that effect, and restart, again from the smallest possible value of x.

Now let's apply the philosophy of structured programming to see how the whole thing should be organized. We ask first for the major logical blocks into which the computation is divided. In this case we see

(i) a procedure `midpt`. Input to this procedure will be $x,\ h,\ y_{n-1},\ y_n$. Output from it will be y_{n+1} computed from the midpoint formula. No arrays are involved. The three values of y in question occupy just three memory locations. The leading statement in this routine might be

```
midpt:=proc(x,h,y0,ym1,n);
```

and its `return` statement would be `return(yp1);`. One might think that it is scarcely necessary to have a separate subroutine for such a simple calculation. The spirit of structured programming dictates otherwise. Someday one might want to change from the midpoint predictor to some other predictor. If organized as a subrouting, then it's quite easy to disentangle it from the program and replace it. This is the "modular" arpproach.

(ii) a procedure `trapez`. This routine will be called from two or three different places in the main routine: when starting, when restarting with a new value of h, and in a generic step, wher eit is the corrector formula. Operation of thie routine is different, too, depending on the circumstances. When starting or restarting, there is no externally supplied guess to help it. It must find its own way to convergence. On a generic step of the integration, however, we want it to use the prediction supplied by `midpt`, and then just do a single correction.

One way to handle this is to use a logical variable `start`. If `trapez` is called with `start = TRUE`, then the subroutine would supply a final converged value without looking for any input guess. Suppose the first line of `trapez` is

```
trapez:=proc(x,h,yin,yguess,n,start,eps);
```

and its `return` statement is `return(yout);`. When called with `start = TRUE`, then the routine might use `yin` as its first guess to *yout*, and iterate to convergence from there, using `eps` as its convergence criterion. If `start = FALSE`, it will take `yguess` as an estimate of `yout`, then use the trapezoidal rule just once, to refine the value as `yout`.

The combination of these two modules plus a small main program that calls them as needed, constitutes the whole package. Each of the subroutines and the main routine should be heavily documented in a self-contained way. That is, the descriptions of `trapez`, and of `midpt`, should precisely explain their operation as separate modules, with their own inputs and outputs, quite independently of the way they happen to be used by the main program in this problem. It should be possible to unplug a module from this application and use it without change in another. The documentation of a procedure should never make reference to how it is used by a certain calling program, but should describe only how it transforms its own inputs into its own outputs.

In the next section we are going to take an intermission from the study of integration rules in order to discuss an actual physical problem, the flight of a spacecraft to the moon. This problem will need the best methods that we can find for a successful, accurate solution. Then in section 2.12, we'll return to the modules discussed above, and will display complete computer programs that carry them out. In the meantine, the reader might enjoy trying to write such a complete program now, and comparing it with the one in that section.

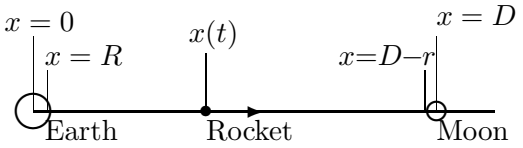

Figure 2.2: 1D Moon Rocket

2.10 Case study: Rocket to the moon

Now we have a reasonably powerful apparatus for integration of initial-value problems and systems, including the automatic regulation of step size and built-in error estimation. In order to try out this software on a problem that will use all of its capability, in this section we are going to derive the differential equations that govern the flight of a rocket to the moon. We do this first in a one-dimensional model, and then in two dimensions. It will be very useful to have these equations available for testing various proposed integration methods. Great accuracy will be needed, and the ability to chenge the step size, both to increase it and the devrease it, will be essential, or else the computation will become intolerably long. The variety of solutions that can be obtained is quite remarkable.

First, in the one-dimensional simplified model, we place the center of the earth at the origin of the x-axis, and let R denote the earth's radius. At the point $x = D$ we place the moon, and we let its radius be r. Finally, at a position $x = x(t)$ is our rocket, making its way towards the moon.

We will use Newton's law of gravitation to find the net gravitational force on the rocket,and equate it to the mass of the rocket times its acceleration (Newton's second law of motion). According to Newton's law of gravitation, the gravitational force exerted by one body on another is proportional to the product of their masses and inversely proportional to the square of the distance between them. If we use K for the constant of proportionality, then the force on the rocket due to the earth is

$$-K\frac{M_E m}{x^2}, \tag{2.10.1}$$

whereas the force on the rocket due to the moon's gravity is

$$K\frac{M_M m}{(D-x)^2} \tag{2.10.2}$$

where M_E, M_M and m are, respectively, the masses of the earth, the moon and the rocket.

The acceleration of the rocket is of course $x''(t)$, and so the assertion that the net force is equal to mass times acceleration takes the form:

$$mx'' = -K\frac{M_E m}{x^2} + K\frac{M_M m}{(D-x)^2}. \tag{2.10.3}$$

This is a (nasty) differential equation of second order in the unknown function $x(t)$, the position of the rocket at time t. Note the nonlinear way in which this unknown function appears on the right-hand side.

A second-order differential equations deserves two initial values, and we will oblige. First. let's agree that at time $t = 0$ the rocket was on the surface of the earth, and second, that the rocket was fired at the moon with a certain initial velocity V. Hence, the initial conditions that go with (2.10.3) are

$$x(0) = R\,; \qquad x'(0) = V. \tag{2.10.4}$$

Now, just a quick glance at (2.10.3) shows that m cancels out, so let's remove it, but not before pointing out the immense significance of that fact. It implies that the motion of the rocket is independent of its mass. For performing a now-legendary experiment with rocks of different sizes dropping from the Tower of Pisa, Galileo demonstrated that fact to an incredulous world.

At any rate, (2.10.3) now reads as

$$x'' = -\frac{KM_E}{x} + \frac{KM_M}{(D-x)^2}. \tag{2.10.5}$$

We can make this equation a good bit prettier by changing the units of distance and time from miles and seconds (or whatever) to a set of more natural units for the problem.

For our unit of distance we choose R, the radius of the earth. If we divide (2.10.5) through by R, we can write the result as

$$\left(\frac{x}{R}\right)'' = -\frac{\frac{KM_E}{R^3}}{\left(\frac{X}{R}\right)^2} + \frac{\frac{KM_M}{R^3}}{\left(\frac{D}{R} - \frac{x}{R}\right)^2}. \tag{2.10.6}$$

Now instead of the unknown function $x(t)$, we define $y(t) = x(t)/R$. Then $y(t)$ is the position of the rocket, expressed in earth radii, at time t. Further, the ratio D/R that occurs in (2.10.6) is a dimensionless quantity, whose numerical value is about 60. Hence (2.10.6) has now been transformed to

$$y'' = -\frac{\frac{KM_E}{R^3}}{y^2} + \frac{\frac{KM_M}{R^3}}{(60-y)^2}. \tag{2.10.7}$$

Next we tackle the new time units. Since y is now dimensionless, the dimension of the left side of the equation is the reciprocal of the square of a time. If we look next at the first term on the right, which of course has the same dimension, we see that the quantity R^3/KM_E is the square of a time, so

$$T_0 = \sqrt{\frac{R^3}{KM_E}} \tag{2.10.8}$$

is a time. Its numerical value is easier to calculate if we change the formula first, as follows.

Consider a body of mass m on the surface of the earth. Its weight is the magnitude of the force exerted on it by the earth's gravity, namely KM_Em/R^2. Its weight is also equal to m times the acceleration of the body, namely the acceleration due to gravity, usually denoted by g, and having the value 32.2 feet/sec^2.

It follows that

$$\frac{KM_e m}{R^2} = mg, \tag{2.10.9}$$

and if we substitute into (2.10.8) we find that our time unit is

$$T_0 = \sqrt{\frac{R}{g}}\,. \tag{2.10.10}$$

We take $R = 4000$ miles, and find T_0 is about 13 minutes and 30 seconds. We propose to measure time in units of T_0. To that end, we multiply through equation (2.10.7) by T_0 and get

$$T_0^2 y'' = -\frac{1}{y^2} + \frac{\frac{M_M}{M_E}}{(60-y)^2}. \tag{2.10.11}$$

The ratio of the mass M_M of the moon to the mass M_E of the earth is about 0.012. Furthermore, we will now introduce a new independent variable τ and a new dependent variable $u = u(\tau)$ by the relations

$$u(\tau) = y(\tau T_0)\,; \qquad t = \tau T_0\,. \tag{2.10.12}$$

Thus, $u(\tau)$ represents the position of the rocket, measured in units of the radius of the earth, at a time τ that is measured in units of T_0, *i.e.*, in units of 13.5 minutes.

The substitution of (2.10.12) into (2.10.11) yields the differential equation for the scaled distance $u(\tau)$ as a function of the scaled time τ in the form

$$u'' = -\frac{1}{u^2} + \frac{0.012}{(60-u)^2}. \tag{2.10.13}$$

Finally we must translate the initial conditions (2.10.4) into conditions on the new variables. The first condition is easy: $u(0) = 1$. Next, if we differentiate (2.10.12) and set $\tau = 0$ we get

$$u'(0) = \frac{T_0 V}{R} = \frac{V}{R/T_0}. \tag{2.10.14}$$

This is a ratio of two velocities. In the numerator is the velocity with which the rocket is launched. What is the significance of the velocity R/T_0?

We claim that it is, aside from a numerical factor, the escape velocity from the earth, if there were no moon. Perhaps the quickest way to see this is to go back to equation (2.10.11) and drop the second term on the right-hand side (the one that comes from the moon). Then we will be looking at the differential equation that would govern the motion if the moon were absent. This equation can be solved. Multiply bth sides by $2y'$, and it becomes

$$T_0^2 \left((y')^2\right)' = \left(\frac{2}{y}\right)'\,, \tag{2.10.15}$$

and integration yields

$$T_0^2 (y')^2 = \frac{2}{y} + C. \tag{2.10.16}$$

Now let $t = 0$ and find that $C = T_0^2 V^2/R^2 - 2$, so

$$T_0^2 (y')^2 = \frac{2}{y} - \left(\frac{T_0^2 V^2}{R^2} - 2 \right). \tag{2.10.17}$$

Suppose the rocket is launched with sufficient initial velocity to escape from the earth. Then the function $y(t)$ will grow without bound. Hence let $y \to \infty$ on the right side of (2.10.17). For all values of y, the left side is a square, and therefore a nonnegative quantity. Hence the right side, which approaches the constant C, must also be nonnegative. Thus $C \geq 0$ or, equivalently

$$V \geq \sqrt{2} \frac{R}{T_0}. \tag{2.10.18}$$

Thus, if the rocket escapes, then (2.10.18) is true, and the converse is easy to show also. Hence the quantity $\sqrt{2}\, R/T_0$ is the *escape velocity* from the earth. We shall denote it by V_{esc}. Its numerical value is approximately $25,145$ miles per hour.

Now we can return to (2.10.12) to translate the initial conditions on $x'(t)$ into initial conditions on $u'(\tau)$. In terms of the escape velocity, it becomes $u'(0) = \sqrt{2}\, V/V_{esc}$. We might say that if we choose to measure distance in units of earth radii, and time in units of T_0, then velocities turn out to be measured in units of escape velocity, aside from the $\sqrt{2}$.

In summary, the differential equation and the initial conditions have the final form

$$\begin{aligned} u'' &= -\frac{1}{u^2} + \frac{0.012}{(60-u)^2} \\ u(0) &= 1 \\ u'(0) &= \sqrt{2} \frac{V}{V_{esc}} \end{aligned} \tag{2.10.19}$$

Since that was all so easy, let's try the two-dimensional case next. Here, the earth is centered at the origin of the xy-plane, and the moon is moving. Let the coordinates of the moon at time t be $(x_m(t), y_m(t))$. For example, if we take the orbit of the moon to be a circle of radius D, then we would have $x_m = D\cos(\omega t)$ and $y_m(t) = D\sin(\omega t)$.

If we put the rocket at a generic position $(x(t), y(t))$ on the way to the moon, then we have the configuration shown in figure 1.16.2.

Consider the net force on the rocket in the x direction. It is given by

$$F_x = -\frac{KM_E m \cos\theta}{x^2 + y^2} + \frac{KM_M m \cos\psi}{(x - x_m)^2 + (y - y_m)^2}, \tag{2.10.20}$$

where the angles θ and ψ are shown in figure 1.16.2. From that figure, we see that

$$\cos\theta = x\sqrt{x^2 + y^2} \tag{2.10.21}$$

and

$$\cos\psi = \frac{x_m - x}{\sqrt{(x_m - x)^2 + (y_m - y)^2}}. \tag{2.10.22}$$

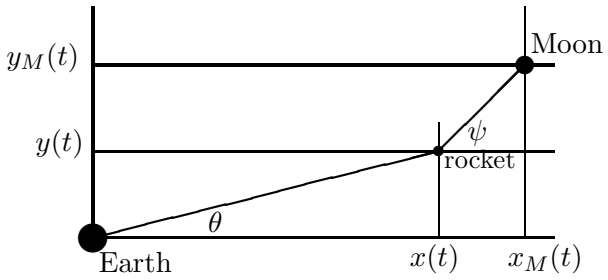

Figure 2.3: The 2D Moon Rocket

Now we substitute into (2.10.20), and equate the force in the x direction to $mx''(t)$, to obtain the differential equation

$$mx''(t) = -\frac{KM_E mx}{(x^2+y^2)^{3/2}} + \frac{KM_M m(x_m - x)}{((x_m - x)^2 + (y_m - y)^2)^{3/2}}. \tag{2.10.23}$$

If we carry out a similar analysis for the y-component of the force on the rocket, we get

$$my''(t) = -\frac{KM_E my}{(x^2+y^2)^{3/2}} + \frac{KM_M m(y_m - y)}{((x_m - x)^2+)y_m - y)^2)^{3/2}}. \tag{2.10.24}$$

We are now looking at two (even nastier) simultaneous differential equations of the second order in the two unknown functions $x(t)$, $y(t)$ that describe the position of the rocket. To go with these equations, we need four initial conditions. We will suppose that at time $t = 0$, the rocket is on the earth's surface, at the point $(R, 0)$. Further, at time $t = 0$, it will be fired with an initial speed of V, in a direction that makes an angle α with the positive x-axis. Thus, our initial conditions are

$$\begin{cases} x(0) = R\,; & y(0) = 0 \\ x'(0) = V\cos\alpha\,; & y'(0) = V\sin\alpha \end{cases}. \tag{2.10.25}$$

The problem has now been completely defined. It remains to change the units into the same natural dimensions of distance and time that were used in the one-dimensional problem. This time we leave the details to the reader, and give only the results. If $u(\tau)$ and $v(\tau)$ denote the x and y coordinates of the rocket, measured in units of earth radii, at a time τ measured in units of T_0 (see (2.10.10)), then it turns out the u and v satisfy the differential equations

$$\begin{aligned} u'' &= -\frac{u}{(u^2+v^2)^{3/2}} + \frac{0.012(u_m - u)}{((u_m - u)^2 + (v_m - v)^2)^{3/2}} \\ v'' &= -\frac{v}{(u^2+v^2)^{3/2}} + \frac{0.012(v_m - v)}{((u_m - u)^2 + (v_m - v)^2)^{3/2}}. \end{aligned} \tag{2.10.26}$$

Furthermore, the initial data (2.10.25) take the form

$$\begin{cases} u(0) = 1\,; & v(0) = 0 \\ u'(0) = \sqrt{2}\dfrac{V\cos\alpha}{V_{esc}}\,; & v'(0) = \sqrt{2}\dfrac{V\sin\alpha}{V_{esc}}. \end{cases} \tag{2.10.27}$$

In these equations, the functions $u_m(\tau)$ and $v_m(\tau)$ are the x and y coordinates of the moon, in units of R, at the time τ. Just to be specific, let's decree that the moon is in a circular orbit of radius $60R$, and that it completes a revolution every twenty eight days. Then, after a brief session with a hand calculator or a computer, we discover that the equations

$$\begin{aligned} u_m(\tau) &= 60\cos(0.002103745\tau) \\ v_m(\tau) &= 60\sin(0.002103745\tau) \end{aligned} \tag{2.10.28}$$

represent the position of the moon.

2.11 Maple programs for the trapezoidal rule

In this section we will first display a complete Maple program that can numerically solve a system of ordinary differential equations of the first order together with given initial values. After discussing those programs, we will illustrate their operation by doing the numerical solution of the one dimensional moon rocket problem.

We will employ Euler's method to predict the values of the unknowns at the next point $x + h$ from their values at x, and then we will apply the trapezoidal rule to correct these predicted values until sufficient convergence has occurred.

First, here is the program that does the Euler method prediction.

```
> eulermethod:=proc(yin,x,h,f)
> local yout,ll,i:
> # Given the array yin of unknowns at x, uses Euler method to return
> # the array of values of the unknowns at x+h. The function f(x,y) is
> # the array-valued right hand side of the given system of ODE's.
> ll:=nops(yin):
> yout:=[]:
> for i from 1 to ll do
>   yout:=[op(yout),yin[i]+h*f(x,yin,i)];
>                    od:
> RETURN(yout):
> end:
```

Next, here is the program that takes as input an array of guessed values of the unknowns at $x + h$ and refines the guess to convergence using the trapezoidal rule.

```
> traprule:=proc(yin,x,h,eps,f)
> local ynew,yfirst,ll,toofar,yguess,i,allnear,dist;
> # Input is the array yin of values of the unknowns at x. The program
> # first calls eulermethod to obtain the array ynew of guessed values
> # of y at x+h. It then refines the guess repeatedly, using the trapezoidal
> # rule, until the previous guess, yguess, and the refined guess, ynew, agree
> # within a tolerance of eps in all components. Program then computes dist,
> # which is the largest deviation of any component of the final converged
> # solution from the initial Euler method guess. If dist is too large
```

```
> # the mesh size h should be decreased; if too small, h should be increased.
> ynew:=eulermethod(yin,x,h,f);
> yfirst:=ynew;
> ll:=nops(yin);
> allnear:=false;
> while(not allnear) do
> yguess:=ynew;
> ynew:=[];
> for i from 1 to ll do
>   ynew:=[op(ynew),yin[i]+(h/2)*(f(x,yin,i)+f(x+h,yguess,i))];
>                   od;
> allnear:=true;
> for i from 1 to ll do allnear:=allnear and abs(ynew[i]-yguess[i])<eps od:
>               od; #end while
> dist:=max(seq(abs(ynew[i]-yfirst[i]),i=1..ll));
> RETURN([dist,ynew]):
> end:
```

The two programs above each operate at a single point x and seek to compute the unknowns at the next point $x + h$. Now we need a global view, that is a program that will call the above repeatedly and increment the value of x until the end of the desired range of x. The global routine also needs to check whether or not the mesh size h needs to be changed at each point and to do so when necessary.

```
> trapglobal:=proc(f,y0,h0,xinit,xfinal,eps,nprint)
> local x,y,y1,h,j,arr,dst,cnt:
> # Finds solution of the ODE system y'=f(x,y), where y is an array
> # and f is array-valued. y0 is initial data array at x=xinit.
> # Halts when x>xfinal. eps is convergence criterion for
> # trapezoidal rule; Prints every nprint-th value that is computed.
> x:=xinit:y:=y0:arr:=[[x,y[1]]]:h:=h0:cnt:=0:
> while x<=xfinal do
>   y1:=traprule(y,x,h,eps,f):
>   y:=y1[2]:dst:=y1[1];
> # Is dst too large? If so, halve the mesh size h and repeat.
>   while dst>3*eps do
>     h:=h/2; lprint(`At x=`,x,`h was reduced to`,h);
>     y1:=traprule(y,x,h,eps,f):
>     y:=y1[2]:dst:=y1[1];
>                   od:
> # Is dst too small? If so, double the mesh size h and repeat.
>   while dst<.0001*eps do
>       h:=2*h; lprint(`At x=`,x,`h was increased to`,h);
>       y1:=traprule(y,x,h,eps,f):
>       y:=y1[2]:dst:=y1[1];
>                   od:
> # Adjoin newly computed values to the output array arr.
>  x:=x+h; arr:=[op(arr),[x,y[2]]]:
> # Decide if we should print this line of output or not.
>   cnt:=cnt+1: if cnt mod nprint =0 or x>=xfinal then print(x,y) fi;
```

```
>                od:
> RETURN(arr);
> end:
```

The above three programs comprise a general package that can numerically solve systems of ordinary differential equations. The applicability of the package is limited mainly by the fact the Euler's method and the Trapezoidal Rule are fairly primitive approximations to the truth, and therefore one should not expect dazzling accuracy when these routines are used over long intervals of integration.

2.11.1 Example: Computing the cosine function

We will now give two examples of the operation of the above programs. First let's compute the cosine function. We will numerically integrate the equation $y'' + y = 0$ with initial conditions $y(0) = 1$ and $y'(0) = 0$ over the range from $x = 0$ to $x = 2\pi$. To do this we use two unknown functions y_1, y_2 which are subject to the equations $y_1' = y_2$ and $y_2' = -y_1$, together with initial values $y_1(0) = 1$, $y_2(0) = 0$. Then the function $y_1(x)$ will be the cosine function.

To use the routines above, we need to program the function $f = f(x, y)$ that gives the right hand sides of the input ODE's. This is done as follows:

```
> f:=proc(x,u,j)
> if j=1 then RETURN(u[2]) else RETURN(-u[1]) fi:
> end:
```

That's all. To run the programs we type the line

```
> trapglobal(f,[1,0],.031415927,0,6.3,.0001,50):
```

This means that we want to solve the system whose right hand sides are as given by f, with initial condition array $[1, 0]$. The initial choice of the mesh size h is $\pi/100$, in order to facilitate the comparison of the values that will be output with those of the cosine function at the same points. The range of x over which we are asking for the solution is from $x = 0$ to $x = 6.3$. Our convergence criterion `eps` is set to .0001, and we are asking the program to print every 50th line of putput that it computes. Maple responds to this call with the following output.

```
At x=   0   h was reduced to   .1570796350e-1
               .7853981750, [ .6845520546,  -.7289635418]
              1.570796368,  [-.0313962454,  -.9995064533]
              2.356194568,  [-.7289550591,  -.6845604434]
              3.141592768,  [-.9995063863,   .0313843153]
              3.926990968,  [-.6845688318,   .7289465759]
              4.712389168,  [ .0313723853,   .9995063195]
              5.497787368,  [ .7289380928,   .6845772205]
              6.283185568,  [ .9995062524,  -.0313604551]
```

We observe that the program first decided that the value of h that we gave it was too big and it was cut in half. Next we see that the accuracy is pretty good. At $x = \pi$ we have 3 or 4 correct digits, and we still have them at $x = 2\pi$. The values of the negative of the sine function, which are displayed above as the second column of unknowns, are less accurate at those points. To improve the accuracy we might try reducing the error tolerance `eps`, but realistically we will have to confess that the major source of imprecision lies in the Euler and Trapezoidal combination itself, which, although it provides a good introduction to the philosophy of these methods, is too crude to yield results of great accuracy over a long range of integration.

2.11.2 Example: The moon rocket in one dimension

As a second example we will run the moon rocket in one dimension. The equations that we're solving now are given by (2.10.19). So all we need to do is to program the right hand sides f, which we do as follows.

```
> f:=proc(x,u,j)
> if j=1 then RETURN(u[2]) else RETURN(-1/u[1]^2+.012/(60-u[1])^2) fi:
> end:
```

Then we invoke our routines by the statement

```
 trapglobal(f,[1,1.4142],.02,0,250,.0001,1000):
```

This means that we are solving our system with initial data $(1, \sqrt{2})$, with an initial mesh size of $h = .02$, integrating over the range of time from $t = 0$ until $t = 250$, with a convergence tolerance `eps` of .0001, and printing the output every 1000 lines. We inserted an extra line of program also, so as to halt the calculation as soon as $y[1]$, the distance from earth, reached 59.75, which is the surface of the moon.

Maple responded with the following output.

```
At x=   0   h was reduced to   .1000000000e-1
               10.00000000, [7.911695068, .5027323920]
               20.00000000, [12.36222569, .4022135344]
               30.00000000, [16.11311118, .3523608113]
               40.00000000, [19.46812426, .3206445364]
               50.00000000, [22.55572225, .2979916199]
               60.00000000, [25.44558364, .2806820525]
               70.00000000, [28.18089771, .2668577906]
               80.00000000, [30.79081775, .2554698522]
               90.00000000, [33.29624630, .2458746369]
               100.0000000, [35.71287575, .2376534139]
               110.0000000, [38.05293965, .2305225677]
               120.0000000, [40.32630042, .2242857188]
               130.0000000, [42.54117736, .2188074327]
               140.0000000, [44.70467985, .2139997868]
```

```
               150.0000000, [46.82325670, .2098188816]
               160.0000000, [48.90316649, .2062733168]
               170.0000000, [50.95113244, .2034559376]
At x=   174.2300000   h was increased to   .2000000000e-1
At x=   183.4500000   h was increased to   .4000000000e-1
               188.0900000, [54.61091112, .2014073390]
At x=   211.7300000   h was reduced to   .2000000000e-1
               211.8100000, [59.75560109, .3622060968]
```

So the trip was somewhat eventful. The mesh size was reduced to .01 right away. It was increased again after 174 time units because at that time the rocket was moving quite slowly, and again after 183 time units for the same reason. Impact on the surface of the moon occurred at time 211.81. Since each time unit corresponds to 13.5 minutes, this means that the whole trip took $211.8 \cdot 13.5$ minutes, or 47.65 hours – nearly two days.

The reader might enjoy experimenting with this situation a little bit. For instance, if we reduce the initial velocity from 1.4142 by a small amount, then the rocket will reach some maximum distance from Earth and will fall back to the ground without ever having reached the moon.

2.12 The big leagues

The three integration rules that we have studied are able to handle small-to-medium sized problems in reasonable time, and with good accuracy. Some problems, however, are more demanding than that. Our two-dimensional moon shot is an example of such a situation, where even a good method like the trapezoidal rule is unable to give the pinpoint accuracy that is needed. In this section we discuss a general family of methods, of which all three of the rules that we have studied are examples, that includes methods of arbitrarily high accuracy. These are the *linear multistep methods*.

A general linear multistep method is of the form

$$y_{n+1} = \sum_{i=0}^{p} a_{-i} y_{n-i} + h \sum_{i=-1}^{p} b_{-i} y'_{n-i} \,. \tag{2.12.1}$$

In order to compute the next value of y, namely y_{n+1}, we need to store $p+1$ backwards values of y and $p+1$ backwards values of y'. A total of $p+2$ points are involved in the formula, and so we can call (2.12.1) a $p+2$-*point formula*.

The trapezoidal rule, for example, arises by taking $p = 0$, $a_0 = 1$, $b_0 = 1/2$, $b_{-1} = 1/2$. Euler's method has $p = 0$, $a_0 = 1$, $b_{-1} = 0$, $b_0 = 1$, whereas for the midpoint rule we have $p = 1$, $a_0 = 0$, $a_{-1} = 1$, $b_1 = 0$, $b_0 = 2$, $b_{-1} = 0$.

We can recognize an explicit, or noniterative, formula by looking at b_1. If b_1 is nonzero, then y_{n+1} appears implicitly on the right side of (2.12.1) as well as on the left, and the formula does not explicitly tell us the value of y_{n+1}. Otherwise, if $b_1 = 0$, the formula is explicit. In either case, if $p > 0$ the formula will need help getting started or restarted, whereas if $p = 0$ it is self-starting.

The general linear multistep formula contains $2p + 3$ constants

$$a_0, \ldots, a_{-p} \quad \text{and} \quad b_1, b_0, b_{-1}, \ldots, b_{-p}.$$

These constants are chosen to give the method various properties that we may deem to be desirable in a particular application. For instance, if we value explicit formulas, then we may set $b_1 = 0$ immediately, thereby using one of the $2p + 3$ free parameters, and leaving $2p + 2$ others.

One might think that the remaining parameters should be chosen so as to give the highest possible accuracy, in some sense. However, for a fixed p, the more accuracy we demand, the more we come into conflict with stability, another highly desirable feature. Indeed, if we demand "too much accuracy" for a fixed p, it will turn out that no stable formulas exist. An important theorem of the subject, due to Dahlquist, states roughly that we cannot use more than about half of the $2p + 3$ "degrees of freedom" to achieve high accuracy if we want to have a stable formula (and we do!).

First let's discuss the conditions of accuracy. These are usually handled by asking that the equation (2.12.1) should be exactly true if the unknown function y happens to be a polynomial of low enough degree. For instance, suppose $y(x) = 1$ for all x. Substitute $y_k = 1$ and $y'_k = 0$ for all k into (2.12.1), and there follows the condition

$$\sum_{i=0}^{p} a_{-i} = 1. \tag{2.12.2}$$

Notice that in all three of the methods we have been studying, the sum of the a's is indeed equal to 1.

Now suppose we want our multistep formula to be exact not only when $y(x) = 1$, but also when $y(x) = x$. We substitute $y_k = kh$, $y'_k = 1$ for all k into (2.12.1) and obtain, after some simplification, and use of (2.12.2),

$$-\sum_{i=0}^{p} i a_{-i} + \sum_{i=-1}^{p} b_{-i} = 1. \tag{2.12.3}$$

The reader should check that this condition is also satisfied by all three of the methods we have studied.

In general let's find the condition for the formula to integrate the function $y(x) = x^r$ and all lower powers of x exactly for some fixed value of r. Hence, in (2.12.1), we substitute $y_k = (kh)^r$ and $y'_k = r(kh)^{r-1}$. After cancelling the factor h^r, we get

$$(n+1)^r = \sum_{i=0}^{p} a_{-i}(n-i)^r + r \sum_{i=-1}^{p} b_{-i}(n-i)^{r-1}. \tag{2.12.4}$$

Now clearly we do x^r and all lower powers of x exactly if and only if we do $(x + c)^r$ and all lower powers of x exactly. The conditions are therefore translation invariant, so we can choose one special value of n in (2.12.4) if we want.

Let's choose $n = 0$, because the result simplifies then to

$$(-1)^r = \sum_{i=0}^{p} i^r a_{-i} - r \sum_{i=-1}^{p} i^{r-1} b_{-i} \qquad r = 0, 1, 2, \ldots . \tag{2.12.5}$$

A small technical remark here is that when $r = 1$ and $i = 0$ we see a 0^0 in the second sum on the right. This should be interpreted as 1, in accordance with (2.12.3).

By the *order* (of accuracy) of a linear multistep method we mean the highest power of x that the formula integrates exactly, or equivalently, the largest number r for which (2.12.5) is true (together with its analogues for all numbers between 0 and r).

The accuracy conditions enable us to take the role of designer, and to construct accurate formlas with desirable characteristics. The reader should verify that the trapezoidal rule is the most accurate of all possible two-point formulas, and should seach for the most accurate of all possible three-point formulas (ignoring the stability question altogether).

Now we must discuss the question of stability. Just as in section 2.6, stability will be judged with respect to the performance of our multistep formula on a particular differential equation, namely

$$y' = -\frac{y}{L} \qquad (L > 0) \tag{2.12.6}$$

with $y(0) = 1$.

To see how well the general formula (2.12.1) does with this differential equation, whose solution is a falling exponential, substitute $y'_k = -y_k/L$ for all k in (2.12.1), to obtain

$$(1 + \tau b_1) y_{n+1} - \sum_{i=0}^{p} (a_{-i} - \tau b_{-i}) y_{n-i} = 0 \tag{2.12.7}$$

where as in section 2.6, we have written $\tau = h/L$, the ratio of the step size to the relaxation length of the problem.

Equation (2.12.7) is a linear difference equation with constant coefficients of order $p+1$. If as usual with such equations, we look for solutions of the form α^n, then after substitution and cancellation we obtain the characteristic equation

$$(1 + \tau b_1) \alpha^{p+1} - \sum_{i=0}^{p} (a_{-i} - \tau b_{-i}) \alpha^{p-i} = 0. \tag{2.12.8}$$

This is a polynomial equation of degree $p + 1$, so it has $p + 1$ roots somewhere in the complex plane. These roots depend on the value of τ, that is, on the step size that we use to do the integration.

We can't expect that these roots will have absolute value less than 1 however large we choose the step size h. All we can hope for is that it should be possible, by choosing h small enough, to get all of these roots to lie inside the unit disk in the complex plane.

Just for openers, let's see where the roots are when $h = 0$. In fact, if when $h = 0$ some root has absolute value strictly greater than 1, then there is no hope at all for the formula,

because for all sufficiently small h there will be a root outside the unit disk, and the method will be unstable.

Now when $h = 0$, $\tau = 0$ also, so the polynomial equation (2.12.8) becomes simply

$$\alpha^{p+1} - \sum_{i=0}^{p} a_{-i}\alpha^{p-i} = 0. \tag{2.12.9}$$

This is also a polynomial equation of degree $p + 1$. However, its coefficients don't depend on the step size, or even on the values of the various b's in the formula. The coefficients, and therefore the roots, depend only on the a's that we use.

Hence, our first necessary condition for the stability of the formula (2.12.1) is that the polynomial equation (2.12.9) should have no roots of absolute value greater than 1. The reader should now check the three methods that we have been using to see if this condition is satisfied.

Next we have to study what happens to the roots when h is small and positive. Qualitatively, here's what happens. One of the roots of (2.12.9) is always $\alpha = 1$, because condition (2.12.2) is always satisfied in practice, and since all it asks is that we correctly integrate the equation $y' = 0$, which is surely not an excessive request.

The one root of (2.12.9) that is $= 1$ is our good friend, and we'll call it the *principal root*. As h increases slightly away from 0 the principal root moves slightly away from 1. Let $\alpha_1(h)$ denote the value of the principal root at some small value of h. Then $\alpha_1(0) = 1$. Now for small positive h, it turns out that the principal root tries as hard as it can to be a good approximation to $e^{-\tau}$. This means that the portion of the solution of the difference equation (2.12.7) that comes from the one root $\alpha_1(h)$ is

$$\begin{aligned} \alpha_1(h)^n &= (\text{“nearly” } e^{-\tau})^n \\ &= \text{“nearly” } e^{-n\tau} \\ &= \text{“nearly” } e^{-nh/L}. \end{aligned} \tag{2.12.10}$$

But $e^{-nh/L}$ is the exact solution of the differential equation (2.12.6) that we're trying to solve. Therefore the principal root is trying to give us a very accurate approximation to the exact solution.

Well then, what are all of the other p roots of the difference equation (2.12.7) doing for us? The answer is that they are trying as hard as they can to make our lives difficult. The high quality of our approximate solution derives from the nearness of the principal root to $e^{-\tau}$. This high quality is bought at the price of using a $p + 2$-point multistep formula, and this forces the characteristic equation to be of degree $p + 1$, and hence the remining roots have to be there, too.

People have invented various names for these other, non-principal, roots of the difference equations. One that is printable is “parasitic,” so we'll call them the *parasitic roots*. We would be happiest if they would all be zero, but failing that, we would like them to be as small as possible in absolute value.

A picture of a good multistep method in action, then, with some small value of h, shows one root of the characteristic equation near 1, more precisely, very near $e^{-\tau}$, and all of the other roots near the origin.

Now let's try to make this picture a bit more quantitative. We return to the polynomial equation (2.12.8), and attempt to find a power series expansion for the principal root $\alpha_1(\tau)$ in powers of τ. The expansion will be in the form

$$\alpha_1(\tau) = 1 + q_1\tau + q_2\tau^2 + \cdots \tag{2.12.11}$$

where the q's are to be determined. If we substitue (2.12.11) into (2.12.8), we get

$$(1 - \tau b_1)(1 + q_1\tau + q_2\tau^2 + \cdots)^{p+1} - \sum_{i=0}^{p}(a_{-i} - \tau b_{-i})(1 + q_1\tau + q_2\tau^2 + \cdots)^{p-i} = 0. \tag{2.12.12}$$

Now we equate the coefficient of each power of τ to zero. First, the coefficient of the zeroth power of τ is

$$1 - \sum_{i=0}^{p} a_{-i} \tag{2.12.13}$$

and, according to (2.12.2), this is indeed zero if our multistep formula correctly integrates a constant function.

Second, the coefficient of τ is

$$b_1 + (p+1)q_1 - \sum_{i=0}^{p}[a_{-i}(p-i)q_1 + b_{-i}] = 0. \tag{2.12.14}$$

However, if we use (2.12.2) and (2.12.3), this simplifies instantly to the simple statement that $q_1 = -1$.

So far, we have shown by direct calculation that if our multistep formula is of order at least 1 (*i.e.*, if (2.12.4) holds for $r = 0$ and $r = 1$), then the expansion (2.12.11) of the principal root agrees with the expansion of $e^{-\tau}$ through terms of first degree in τ.

Much more is true, although we will not prove it here: the expansion of the principal root agrees with the expansion of $e^{-\tau}$ through terms of degree equal to the order of the multistep method under consideration. The proof can be done just by continuing the argument that we began above, but we omit the details.

Thus the careful determination of the a's and the b's so as to make the formula as accurate as possible all result in the principal root being "nearly" $e^{-\tau}$. Equal care must be taken to assure that the parasitic roots stay small.

We illustrate these ideas with a new multistep formula,

$$y_{n+1} = y_{n-1} + \frac{h}{3}(y'_{n+1} + 4y'_n + y'_{n-1}). \tag{2.12.15}$$

This, like the midpoint rule, is a three-point method ($p = 1$). It is iterative, because $b_1 = 1/3 \neq 0$, and it happens to be very accurate. Indeed, we can quickly check that

the accuracy conditions (2.12.5) are satisfied for $r = 0,\ 1,\ 2,\ 3,\ 4$. The method is of fourth order, and in fact its error term can be found by the method of section 2.8 to be $-h^5 y^{(v)}(X)/90$, where X lies between x_{n-1} and x_{n+1}.

Now we examine the stability of this method. First, when $h = 0$ the equation (2.12.9) that determines the roots is just $\alpha^2 - 1 = 0$, so the roots are $+1$ and -1. The root at $+1$ is the friendly one. As h increases slightly to small positive values, that root will follow the power series expansion of $e^{-\tau}$ very accurately, in fact, through the first four powers of τ.

The root at -1 is to be regarded with apprehension, because it is poised on the brink of causing trouble. If as h grows to a small positive value, this root grows in absolute value, then its powers will dwarf the powers of the principal root in the numerical solution, and all accuracy will eventually be lost.

To see if this happens, let's substitute a power series

$$\alpha_2(h) = -1 + q_1\tau + q_2\tau^2 + \cdots \tag{2.12.16}$$

into the characteristic equation (2.12.8), which in the present case is just the quadratic equation

$$\left(1 + \frac{\tau}{3}\right)\alpha^2 + \frac{4\tau}{3}\alpha - \left(1 - \frac{\tau}{3}\right) = 0. \tag{2.12.17}$$

After substituting, we quickly find that $q_1 = -1/3$, and our apprehension was fully warrented, because for small τ the root acts like $-1 - \tau/3$, so for all small positive values of τ this lies outside of the unit disk, so the method will be unstable.

In the next section, we are going to describe a family of multistep methods, called the *Adams methods*, that are stable, and that offer whatever accuracy one might want, if one is willing to save enough backwards values of the y's. First we will develop a very general tool, the Lagrange interpolation formula, that we'll need in several parts of the sequel, and following that we discuss the Adams formulas. The secret weapon of the Adams formulas is that when $h = 0$, one of the roots (the friendly one) is as usual sitting at 1, ready to develop into the exponential series, and all of the unfriendly roots are huddled together at the origin, just as far out of trouble as they can get.

2.13 Lagrange and Adams formulas

Our next job is to develop formulas that can give us as much accuracy as we want in a numerical solution of a differential equation. This means that we want methods in which the formulas span a number of points, *i.e.*, in which the next value of y is obtained from several backward values, instead of from just one or two as in the methods that we have already studied. Furthermore, these methods will need some assistance in getting started, so we will have to develop matched formulas that will provide them with starting values of the right accuracy.

All of these jobs can be done with the aid of a formula, due to Lagrange, whose mission in life is to fit a polynomial to a given set of data points, so let's begin with a little example.

PROBLEM: Through the three points $(1, 17),\ (3, 10),\ (7, -5)$ there passes a unique quadratic polynomial. Find it.

SOLUTION:

$$P(x) = 17\left(\frac{(x-3)(x-7)}{(1-3)(1-7)}\right) + 10\left(\frac{(x-1)(x-7)}{(3-1)(3-7)}\right) + (-5)\left(\frac{(x-1)(x-3)}{(7-1)(7-3)}\right). \quad (2.13.1)$$

Let's check that this is really the right answer. First of all, (2.13.1) is a quadratic polynomial, since each term is. Does it pass through the three given points? When $x = 1$, the second and third terms vanish, because of the factor $x - 1$, and the first term has the vlaue 17, as is obvious from the cancellation that takes place. Similarly when $x = 3$, the first and third terms are zero and the second is 10, etc.

Now we can jump from this little example all the way to the general formula. If we want to find the polynomial of degree $n - 1$ that passes through n given data points

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

then all we have to do is to write it out:

$$P(x) = \sum_{i=1}^{n} y_i \left(\prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \right). \quad (2.13.2)$$

This is the Lagrange interpolation formula. In the ith term of the sum above is the product of $n - 1$ factors $(x - x_j)/(x_i - x_j)$, namely of all those factors except for the one in which $j = i$.

Next, consider the special case of the above formula in which the points $x_1,\ x_2, \ldots,\ x_n$ are chosen to be equally spaced. To be specific, we might as well suppose that $x_j = jh$ for $j = 1, 2, \ldots, n$.

If we substitute in (2.13.2), we get

$$P(x) = \sum_{i=1}^{n} y_i \left(\prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - jh}{(i-j)h} \right). \quad (2.13.3)$$

Consider the product of the denominators above. It is

$$(i-1)(i-2)\cdots(1)(-1)(-2)\cdots(i-n)h^{n-1} = (-1)^{n-i}(i-1)!(n-i)!h^{n-1}. \quad (2.13.4)$$

Finally we replace x by xh and substitute in (2.13.3) to obtain

$$P(xh) = \sum_{i=1}^{n} \frac{(-1)^{n-i}}{(i-1)!(n-i)!} y_i \left(\prod_{\substack{j=1 \\ j \neq i}}^{n} (x - j) \right) \quad (2.13.5)$$

and that is about as simple as we can get it.

Now we'll use this result to obtain a collection of excellent methods for solving differential equations, the so-called Adams formulas. Begin with the obvious fact that

$$y((p+1)h) = y(ph) + h\int_p^{p+1} y'(ht)\,dt. \tag{2.13.6}$$

Instead of integrating the exact function y' in this formula, we will integrate the polynomial that agrees with y' at a number of given data points. First, we replace y' by the Lagrange interpolating polynomial that agrees with it at the $p+1$ points $h,\ 2h, \ldots, (p+1)h$. This transforms (2.13.6) into

$$y_{p+1} = y_p + h\sum_{i=1}^{p+1} \frac{(-1)^{p-i+1}}{(i-1)!(p-i+1)!} y'(ih) j \int_p^{p+1} \left(\prod_{\substack{j=1 \\ j\neq i}}^{p+1} (x-j) \right) dx. \tag{2.13.7}$$

We can rewrite this in the more customary form of a linear multistep method:

$$y_{n+1} = y_n + h\sum_{i=-1}^{p-1} b_{-i} y'_{n-i}\,. \tag{2.13.8}$$

This involves replacing i by $p-i$ in (2.13.7), so the numbers b_{-i} are given by

$$b_{-i} = \frac{(-1)^{i+1}}{(p-1-i)!(i+1)!} \int_p^{p+1} \left(\prod_{\substack{j=1 \\ j\neq p-i}}^{p+1} (x-j) \right) dx \qquad i = -1, 0, \ldots, p-1. \tag{2.13.9}$$

Now to choose a formula in this family, all we have to do is specify p. For instance, let's take $p = 2$. Then we find

$$\begin{aligned} b_1 = &\ \frac{1}{2}\int_2^3 (x-1)(x-2)dx &= \tfrac{5}{12} \\ b_0 = &\ -\int_2^3 (x-1)(x-3)dx &= \tfrac{2}{3} \\ b_{-1} = &\ \frac{1}{2}\int_2^3 (x-2)(x-3)dx &= -\tfrac{1}{12} \end{aligned} \tag{2.13.10}$$

so we have the third-order implicit Adams method

$$y_{n+1} = y_n + \frac{h}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}). \tag{2.13.11}$$

If this process is repeated for various values of p we get the whole collection of these formulas. For reference, we show the first four of them below, complete with their error terms.

$$y_{n+1} \ = \ y_n + hy'_n - \frac{h^2}{2}y'' \tag{2.13.12}$$

$$y_{n+1} = y_n + \frac{h}{2}(y'_{n+1} + y'_n) - \frac{h^3}{12}y''' \tag{2.13.13}$$

$$y_{n+1} = y_n + \frac{h}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}) - \frac{h^4}{24}y^{(iv} \tag{2.13.14}$$

$$y_{n+1} = y_n + \frac{h}{24}(9y'_{n+1} + 19y'_n - 5y'_{n-1} + y'_{n-2}) - \frac{19h^5}{720}y^{(v} \tag{2.13.15}$$

If we compare these formulas with the general linear multistep method (2.12.1) then we quickly find the reason for the stability of these formulas. Notice that only one backwards value of y itself is used, namely y_n. The other backwards values are all of the derivatives. Now look at equation (2.12.9) again. It is the polynomial equation that determines the roots of the characteristic equation of the method, in the limiting case where the step size is zero.

If we use a formula with all the $a_i = 0$, except that $a_0 = 1$, then that polynomial equation becomes simply

$$\alpha^{p+1} - \alpha^p = 0. \tag{2.13.16}$$

The roots of this are 1, 0, 0,. . . ,0. The root at 1, as h grows to a small positive value, is the one that gives us the accuracy of the computed solution. The other roots all start at the origin, so when h is small they too will be small, instead of trying to cause trouble for us. All of the Adams formulas are stable for this reason.

In Table 6 we show the behavior of the three roots of the characteristic equation (2.12.8) as it applies to the fourth-order method (2.13.15). It happens that the roots are all real in this case. Notice that the friendly root is the one of largest absolute value for all $\tau \leq 0.9$.

τ	$e^{-\tau}$	Friendly(τ)	Root2(τ)	Root3(τ)
0.0	1.000000	1.000000	0.000000	0.000000
0.1	0.904837	0.904837	0.058536	-0.075823
0.2	0.818731	0.818723	0.081048	-0.116824
0.3	0.740818	0.740758	0.098550	-0.153914
0.4	0.670320	0.670068	0.113948	-0.189813
0.5	0.606531	0.605769	0.128457	-0.225454
0.6	0.548812	0.546918	0.142857	-0.261204
0.7	0.496585	0.492437	0.157869	-0.297171
0.8	0.449329	0.440927	0.174458	-0.333333
0.9	0.406570	0.390073	0.194475	-0.369595
1.0	0.367879	0.333333	0.224009	-0.405827

TABLE 6

Now we have found the implicit Adams formulas. In each of them the unknown y_{n+1} appears on both sides of the equation. They are, therefore, useful as corrector formulas.

To find matching predictor formulas is also straightforward. We return to (2.13.6), and for a predictor method of order $p + 1$ we replace y' by the interpolating polynomial that

agrees with it at the data points $0,\ h,\ 2h,\ 3h, \ldots,\ ph$ (but not at $(p+1)h$ as before). Then, in place of (2.13.7) we get

$$y_{p+1} = y_p + h\sum_{i=0}^{p} \frac{(-1)^{p-i}}{i!(p-i)!} y'(ih) \int_p^{p+1} \left(\prod_{\substack{j=0 \\ j\neq i}}^{p} (x-j) \right) dx. \tag{2.13.17}$$

As before, we can write this in the more familiar form

$$y_{n+1} = y_n + h\sum_{i=0}^{p} b_{-i} y'_{n-i} \tag{2.13.18}$$

where the numbers b_{-i} are now given by

$$b_{-i} = \frac{(-1)^i}{(p-i)!i!} \int_p^{p+1} \left(\prod_{\substack{j=0 \\ j\neq p-i}}^{p} (x-j) \right) dx \qquad i = 0, 1, \ldots, p. \tag{2.13.19}$$

We tabulate below these formulas in the cases $p = 1, 2, 3$, together with their error terms:

$$y_{n+1} = y_n + \frac{h}{2}(3y'_n - y'_{n-1}) + \frac{5h^3}{12} y''' \tag{2.13.20}$$

$$y_{n+1} = y_n + \frac{h}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + \frac{3h^4}{8} y^{(iv} \tag{2.13.21}$$

$$y_{n+1} = y_n + \frac{h}{24}(55y'_n - 59y'_{n-1} + 37y'_{n-2} - 9y'_{n-3}) + \frac{251h^5}{720} y^{(v} \tag{2.13.22}$$

Notice, for example, that the explicit fourth-order formula (2.13.22) has about 13 times as large an error term as the implicit fourth-order formula (2.13.15). This is typical for matched pairs of predictor-corrector formulas. As we have noted previously, a single application of a corrector formula reduces error by about $h\frac{\partial f}{\partial y}$, so if we keep h small enough so that $h\frac{\partial f}{\partial y}$ is less than $1/13$ or thereabouts, then a single application of the corrector formula will produce an estimate of the next value of y with full attainable accuracy.

We are now fully equipped with Adams formulas for prediction and correction, in matched pairs, of whatever accuracy is needed, all of them stable. The use of these pairs requires special starting formulas, since multistep methods cannot get themselves started or restarted without assistance. Once again the Lagrange interpolation formula comes to the rescue.

This time we begin with a slight variation of (2.13.6),

$$y(mh) = y(0) + \int_0^{mh} y'(t)\, dt. \tag{2.13.23}$$

Next, replace $y'(t)$ by the Lagrange polynomial that agrees with it at $0,\ h,\ 2h, \ldots, ph$ (for $p \geq m$). We then obtain

$$y_m = y_0 + h\sum_{i=0}^{p} \frac{(-1)^{p-i}}{i!(p-i)!} y'_i \int_0^m \left(\prod_{\substack{j=0 \\ j\neq i}}^{p} (t-j) \right) dt \qquad m = 1, 2, \ldots, p. \tag{2.13.24}$$

We can rewrite these equations in the form

$$y_m = y_0 + h\sum_{j=0}^{p} \lambda_j y'_j \qquad m = 1, 2, \ldots, p \tag{2.13.25}$$

where the coefficients λ_i are given by

$$\lambda_i = \frac{(-1)^{p-i}}{i!(p-i)!} \int_0^m \left(\prod_{\substack{j=0 \\ j\neq i}}^{p} (t-j) \right) dt \qquad i = 0, \ldots, p. \tag{2.13.26}$$

Of course, when these formulas are used on a differential equation $y' = f(x, y)$, each of the values of y' on the right side of (2.13.25) is replaced by an $f(x, y)$ value. Therefore equations (2.13.25) are a set of p simultaneous equations in p unknowns $y_1,\ y_2, \ldots, y_p$ (y_0 is, of course, known). We can solve them with an iteration in which we first guess all p of the unknowns, and then refine the guesses all at once by using (2.13.25) to give us the new guesses from the old, until sufficient convergence has occurred.

For example, if we take $p = 3$, then the starting formulas are

$$\begin{aligned} y_1 &= y_0 + \frac{h}{24}(9y'_0 + 19y'_1 - 5y'_2 + y'_3) - \frac{19h^5}{720}y^{(v)} \\ y_2 &= y_0 + \frac{h}{3}(y'_0 + 4y'_1 + y'_2) - \frac{h^5}{90}y^{(v)} \\ y_3 &= y_0 + \frac{3h}{8}(y'_0 + 3y'_1 + 3y'_2 + y'_3) - \frac{3h^5}{80}y^{(v)}. \end{aligned} \tag{2.13.27}$$

The philosophy of using a matched pair of Adams formulas for propagation of the solution, together with the starter formulas shown above has the potential of being implemented in a computer program in which the user could specifiy the desired precision by giving the value of p. The program could then calculate from the formulas above the coefficients in the predictor-corrector pair and the starting method, and then proceed with the integration. This would make a very versatile code.

Chapter 3

Numerical linear algebra

3.1 Vector spaces and linear mappings

In this chapter we will study numerical methods for the solution of problems in linear algebra, that is to say, of problems that involve matrices or systems of linear equations. Among these we mention the following:

(a) Given an $n \times n$ matrix, calculate its determinant.

(b) Given m linear algebraic equations in n unknowns, find the most general solution of the system, or discover that it has none.

(c) Invert an $n \times n$ matrix, if possible.

(d) Find the eigenvalues and eigenvectors of an $n \times n$ matrix.

As usual, we will be very much concerned with the development of efficient software that will accomplish the above purposes.

We assume that the reader is familiar with the basic constructs of linear algebra: vector space, linear dependence and independence of vectors, Euclidean n-dimensional space, spanning sets of vectors, basis vectors. We will quickly review some additional concepts that will be helpful in our work. For a more complete discussion of linear algebra, see any of the references cited at the end of this chapter.

And now, to business. The first major concept we need is that of a linear mapping.

Let V and W be two vector spaces over the real numbers (we'll stick to the real numbers unless otherwise specified). We say that T is a *linear mapping* from V to W if T associates with every vector $\mathbf{v}$ of V a vector $T\mathbf{v}$ of W (so T is a mapping) in such a way that

$$T(\alpha\mathbf{v}' + \beta\mathbf{v}'') = \alpha T\mathbf{v}' + \beta T\mathbf{v}'' \tag{3.1.1}$$

for all vectors $\mathbf{v}'$, $\mathbf{v}''$ of V and real numbers (or *scalars*) α and β (*i.e.*, T is *linear*). Notice that the "+" signs are different on the two sides of (3.1.1). On the left we add two vectors of V, on the right we add two vectors of W.

Here are a few examples of linear mappings.

First, let V and W both be the same, namely the space of all polynomials of some given degree n. Consider the mapping that associates with a polynomial f of V its derivative $Tf = f'$ in W. It's easy to check that this mapping is linear.

Second, suppose V is Euclidean two-dimensional space (the plane) and W is Euclidean three-dimensional space. Let T be the mapping that carries the vector (x, y) of V to the vector $(3x + 2y, x - y, 4x + 5y)$ of W. For instance, $T(2, -1) = (4, 3, 3)$. Then T is a linear mapping.

More generally, let A be a given $m \times n$ matrix of real numbers, let V be Euclidean n-dimensional space and let W be Euclidean m-space. The mapping T that carries a vector $\mathbf{x}$ of V into $A\mathbf{x}$ of W is a linear mapping. That is, any matrix generates a linear mapping between two appropriately chosen (to match the dimensions of the matrix) vector spaces.

The importance of studying linear mappings in general, and not just matrices, comes from the fact that a particular mapping can be represented by many different matrices. Further, it often happens that problems in linear algebra that seem to be questions about matrices, are in fact questions about linear mappings. This means that we can change to a simpler matrix that represents the same linear mapping before answering the question, secure in the knowledge that the answer will be the same. For example, if we are given a square matrix and we want its determinant, we seem to confront a problem about matrices. In fact, any of the matrices that represent the same mapping will have the same determinant as the given one, and making this kind of observation and identifying simple representatives of the class of relevant matrices can be quite helpful.

To get back to the matter at hand, suppose the vector spaces V and W are of dimensions m and n, respectively. Then we can choose in V a basis of m vectors, say $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \ldots, \mathbf{e}_m$, and in W there is a basis of n vectors $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n$. Let T be a linear mapping from V to W. Then we have the situation that is sketched in figure 3.1 below.

We claim now that the action of T on every vector of V is known if we know only its effect on the m basis vectors of V. Indeed, suppose we know $T\mathbf{e}_1,\ T\mathbf{e}_2, \ldots, T\mathbf{e}_m$. Then let $\mathbf{x}$ be any vector in V. Express $\mathbf{x}$ in terms of the basis of V,

$$\mathbf{x} = \alpha_1\mathbf{e}_1 + \alpha_2\mathbf{e}_2 + \cdots + \alpha_m\mathbf{e}_m. \tag{3.1.2}$$

Now apply T to both sides and use the linearity of T (extended, by induction, to linear combinations of more than two vectors) to obtain

$$T\mathbf{x} = \alpha_1(T\mathbf{e}_1) + \alpha_2(T\mathbf{e}_2) + \cdots + \alpha_m(T\mathbf{e}_m). \tag{3.1.3}$$

The right side is known, and the claim is established.

So, to describe a linear mapping, "all we have to do" is describe its action on a set of basis vectors of V. If $\mathbf{e}_i$ is one of these, then $T\mathbf{e}_i$ is a vector in W. As such, $T\mathbf{e}_i$ can be written as a linear combination of the basis vectors of W. The coefficients of this linear combination will evidently depend on i, so we write

$$T\mathbf{e}_i = \sum_{j=1}^{n} t_{ji}\mathbf{f}_j \qquad i = 1, \ldots, m. \tag{3.1.4}$$

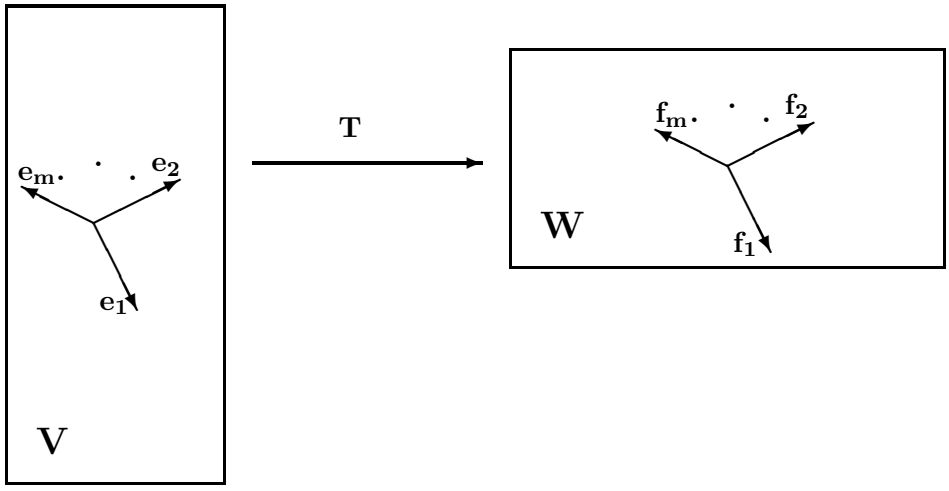

Figure 3.1: The action of a linear mapping

Now the mn numbers t_{ji}, $i = 1, \ldots, m$, $j = 1, \ldots, n$, together with the given sets of basis vectors for V and W, are enough to describe the linear operator T completely. Indeed, if we know all of those numbers, then by (3.1.4) we know what T does to every basis vector of V, and then by (3.1.3) we know the action of T on every vector of V.

To summarize, an $n \times m$ matrix t_{ji} represents a linear mapping T from a vector space V with a distinguished basis $E = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m\}$, to a vector space W with a distinguished basis $F = \{\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n\}$, in the sense that from a knowledge of (t, E, F) we know the full mapping T.

Next, suppose once more that T is a linear mapping from V to W. Since T is linear, it is easy to see that T carries the $\mathbf{0}$ vector of V into the $\mathbf{0}$ vector of W. Consider the set of *all* vectors of V that are mapped by T into the zero vector of W. This set is called the *kernel* of T, and is written $\ker(T)$. Thus

$$\ker(T) = \{\mathbf{x} \in V \,|\, T\mathbf{x} = \mathbf{0}_W\}. \tag{3.1.5}$$

Now $\ker(T)$ is not just a set of vectors, it is itself a vector space, that is a vector *subspace* of V. Indeed, one sees immediately that if $\mathbf{x}$ and $\mathbf{y}$ belong to $\ker(T)$ and α and β are scalars, then

$$T(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha T(\mathbf{x}) + \beta T(\mathbf{y}) = \mathbf{0}, \tag{3.1.6}$$

so $\alpha\mathbf{x} + \beta\mathbf{y}$ belongs to $\ker(T)$ also.

Since $\ker(T)$ is a vector space, we can speak of its dimension. If $\nu = \dim\ker(T)$, then ν is called the *nullity* of the mapping T.

Consider also the set of vectors $\mathbf{w}$ of W that are of the form $\mathbf{w} = T\mathbf{v}$, for some vector $\mathbf{v} \in V$ (possibly many such $\mathbf{v}$'s exist). This set is called the *image* of T, and is written

$$\mathrm{im}(T) = \{\mathbf{w} \in W \,|\, \mathbf{w} = T\mathbf{v},\ \mathbf{v} \in V\}. \tag{3.1.7}$$

Once again, we remark that $\mathrm{im}(T)$ is more than just a set of vectors, it is in fact a vector subspace of W, since if $\mathbf{w}'$ and $\mathbf{w}''$ are both in $\mathrm{im}(T)$, and if α and β are scalars, then we have $\mathbf{w}' = T\mathbf{v}'$ and $\mathbf{w}'' = T\mathbf{v}''$ for some $\mathbf{v}', \mathbf{v}''$ in V. Hence

$$\alpha\mathbf{w}' + \beta\mathbf{w}'' = \alpha T\mathbf{v}' + \beta\mathbf{v}'' = T(\alpha\mathbf{v}' + \beta\mathbf{v}'') \tag{3.1.8}$$

so $\alpha\mathbf{w}' + \beta\mathbf{w}''$ lies in $\mathrm{im}(T)$, too.

The dimension of the vector (sub)space $\mathrm{im}(T)$ is called the *rank* of the mapping T.

A celebrated theorem of Sylvester asserts that

$$\mathrm{rank}(T) + \mathrm{nullity}(T) = \dim(V). \tag{3.1.9}$$

By the *rank of a matrix* A we mean any of the following:

(a) the maximum number of linearly independent rows that we can find in A

(b) same for columns

(c) the largest number r for which there is an $r \times r$ nonzero sub-determinant in A (*i.e.*, a set of r rows and r columns, not necessarily consecutive, such that the $r \times r$ submatrix that they determine has a nonzero determinant.

It is true that the rank of a linear mapping T from V to W is equal to the rank of any matrix A that represents T with respect to some pair of bases of V and W.

It is also true that the rank of a matrix is not computable unless infinite-precision arithmetic is used. In fact, the 2×2 matrix

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 + 10^{-20} \end{bmatrix} \tag{3.1.10}$$

has rank 2, but if the [2,2]-entry is changed to 1, the rank becomes 1. No computer program will be able to tell the difference between these two situations unless it is doing arithmetic to at least 21 digits of precision. Therefore, unless our programs do exact arithmetic on rational numbers, or do finite field arithmetic, or whatever, the rank will be uncomputable. What we are saying, really, is just that the rank of a matrix is not a continuous function of the matrix entries.

Now we are ready to consider one of the most important problems of numerical linear algebra: the solution of simultaneous linear equations.

Let A be a given $m \times n$ matrix, let $\mathbf{b}$ be a given column vector of length m, and consider the system

$$A\mathbf{x} = \mathbf{b} \tag{3.1.11}$$

of m linear simultaneous equations in n unknowns $x_1,\ x_2, \ldots, x_n$.

Consider the set of all solution vectors $\mathbf{x}$ of (3.1.11). Is it a vector space? That's right, it isn't, unless $\mathbf{b}$ happens to be the zero vector (why?).

Suppose then that we intend to write a computer program that will in some sense present as output all solutions $\mathbf{x}$ of (3.1.11). What might the output look like?

If $\mathbf{b} = \mathbf{0}$, *i.e.*, if the system is *homogeneous*, there is no difficulty, for in that case the solution set is $\ker(A)$, a vector space, and we can describe it by printing out a set of basis vectors of $\ker(A)$.

If the right-hand side vector $\mathbf{b}$ is not $\mathbf{0}$, then consider any two solutions $\mathbf{x}'$ and $\mathbf{x}''$ of (3.1.11). Then $A\mathbf{x}' = \mathbf{b}$, $A\mathbf{x}'' = \mathbf{b}$, and by subtraction, $A(\mathbf{x}'-\mathbf{x}'') = \mathbf{0}$. Hence $\mathbf{x}'-\mathbf{x}''$ belongs to $\ker(A)$, so if $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_\nu$ are a basis for $\ker(A)$, then $\mathbf{x}' - \mathbf{x}'' = \alpha_1\mathbf{e}_1 + \alpha_2\mathbf{e}_2 + \cdots + \alpha_\nu\mathbf{e}_\nu$.

If $\mathbf{b} \neq \mathbf{0}$ then, we can describe all possible solutions of (3.1.11) by printing out one particular solution $\mathbf{x}''$, and a list of the basis vectors of $\ker(A)$, because then all solutions are of the form

$$\mathbf{x} = \mathbf{x}'' + \alpha_1\mathbf{e}_1 + \alpha_2\mathbf{e}_2 + \cdots + \alpha_\nu\mathbf{e}_\nu. \tag{3.1.12}$$

Therefore, a computer program that alleges that it solves (3.1.11) should print out a basis for the kernel of A together with, in case $\mathbf{b} \neq \mathbf{0}$, any one particular solution of the system. We will see how to accomplish this in the next section.

EXERCISES 3.1

1. Show by examples that for every n, the rank of a given $n \times n$ matrix A is a discontinuous function of the entries of A.

2. Consider the vector space V of all polynomials in x of degree at most 2. Let T be the linear mapping that sends each polynomial to its derivative.

   (a) What is the rank of T?

   (b) What is the image of T?

   (c) For the basis $\{1, x, x^2\}$ of V, find the 3×3 matrix that represents T.

   (d) Regard T as a mapping from V to the space W of polynomials of degree 1. Use the basis given in part (c) for V, and the basis $\{1, x-1\}$ for W, and find the 2×3 matrix that represents T with respect to these bases.

   (e) Check that the ranks of the matrices in (c) and (d) are equal.

3. Let T be a linear mapping of Euclidean 3-dimensional space to itself. Suppose T takes the vector $(1, 1, 1)$ to $(1, 2, 3)$, and T takes $(1, 0, -1)$ to $(2, 0, 1)$ and T takes $(3, -1, 0)$ to $(1, 1, 2)$. Find $T(1, 2, 4)$.

4. Let A be an $n \times n$ matrix with integer entries having absolute values at most M. What is the maximum number of binary digits that could be needed to represent all of the elements of A?

5. If T acts on the vector space of polynomials of degree at most n according to $T(f) = f' - 3f$, find $\ker(T)$, $\mathrm{im}(T)$, and $\mathrm{rank}(T)$.

6. Why isn't the solution set of (3.1.11) a vector space if $\mathbf{b} \neq 0$?

7. Let $a_{ij} = r_i s_j$ for $i, j = 1, \ldots, n$. Show that the rank of A is at most 1.

8. Suppose that the matrix
$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \\ -2 & 1 & 1 \end{bmatrix} \tag{3.1.13}$$
represents a certain linear mapping from V to V with respect to the basis
$$\{(1,0,0), (0,1,0), (1,1,1)\} \tag{3.1.14}$$
of V. Find the matrix that represents the same mapping with respect to the basis
$$\{(0,0,1), (0,1,1), (1,1,1)\}. \tag{3.1.15}$$
Check that the determinant is unchanged.

9. Find a system of linear equations whose solution set consists of the vector $(1,2,0)$ plus any linear combination of $(-1,0,1)$ and $(0,1,0)$.

10. Construct two sets of two equations in two unknowns such that
    (a) their coefficient matrices differ by at most 10^{-12} in each entry, and
    (b) their solutions differ by at least 10^{+12} in each entry.

3.2 Linear systems

The method that we will use for the computer solution of m linear equations in n unknowns will be a natural extension of the familiar process of Gaussian elimination. Let's begin with a little example, say of the following set of two equations in three unknowns:

$$\begin{aligned} x + y + z &= 2 \\ x - y - z &= 5. \end{aligned} \tag{3.2.1}$$

If we subtract the second equation from the first, then the two equations can be written

$$\begin{aligned} x \;+\; y \;+\; z &= 2 \\ 2y \;+\; 2z &= -3. \end{aligned} \tag{3.2.2}$$

We divide the second equation by 2, and then subtract it from the first, getting

$$\begin{aligned} x \qquad\quad &= \tfrac{7}{2} \\ y \;+\; z &= -\tfrac{3}{2}. \end{aligned} \tag{3.2.3}$$

The value of z can now be chosen arbitrarily, and then x and y will be determined. To make this more explicit, we can rewrite the solution in the form

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{7}{2} \\ -\frac{3}{2} \\ 0 \end{bmatrix} + z \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}. \tag{3.2.4}$$

In (3.2.4) we see clearly that the general solution is the sum of a particular solution $(7/2, -3/2, 0)$, plus any multiple of the basis vector $(0, -1, 1)$ for the kernel of the coefficient matrix of (3.2.1).

The calculation can be compactified by writing the numbers in a matrix and omitting the names of the unknowns. A vertical line in the matrix will separate the left sides and the right sides of the equations. Thus, the original system (3.2.1) is

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 2 \\ 1 & -1 & -1 & 5 \end{array}\right]. \tag{3.2.5}$$

Now we do $(\overrightarrow{\text{row}}\, 2) := (\overrightarrow{\text{row}}\, 1) - (\overrightarrow{\text{row}}\, 2)$ and we have

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 2 \\ 0 & 2 & 2 & -3 \end{array}\right]. \tag{3.2.6}$$

Next $(\overrightarrow{\text{row}}\, 2) := (\overrightarrow{\text{row}}\, 2)/2$, and then $(\overrightarrow{\text{row}}\, 1) := (\overrightarrow{\text{row}}\, 1) - (\overrightarrow{\text{row}}\, 2)$ bring us to the final form

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & \frac{7}{2} \\ 0 & 1 & 1 & -\frac{3}{2} \end{array}\right] \tag{3.2.7}$$

which is the matrix equivalent of (3.2.3).

Now we will step up to a slightly larger example, to see some of the situations that can arise. We won't actually write the numerical values of the coefficients, but we'll use asterisks instead. So consider the three equations in five unknowns shown below.

$$\left[\begin{array}{ccccc|c} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{array}\right]. \tag{3.2.8}$$

The first step is to create a 1 in the extreme upper left corner, by dividing the first row through by the [1,1] element. We will assume for the moment that the various numbers that we want to divide by are not zero. Later on we will take extensive measures to assure this.

After we divide the first row by the [1,1] element, we use the 1 in the upper left-hand corner to zero out the entries below it in column 1. That is, we let $t = a_{21}$ and then do

$$\overrightarrow{\text{row}}(2) := \overrightarrow{\text{row}}(2) - t * \overrightarrow{\text{row}}(1). \tag{3.2.9}$$

Then let $t = a_{31}$ and do

$$\overrightarrow{\text{row}}(3) := \overrightarrow{\text{row}}(3) - t * \overrightarrow{\text{row}}(1). \tag{3.2.10}$$

The result is that we now have the matrix

$$\left[\begin{array}{ccccc|c} 1 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \end{array}\right]. \tag{3.2.11}$$

We pause for a moment to consider what we've done, in terms of the original set of simultaneous equations. First, to divide a row of the matrix by a number corresponds to dividing an equation through by the same number. Evidently this does not change the solutions of the system of equations. Next, to add a constant multiple of one row to another in the matrix corresponds to adding a multiple of one equation to another, and this also doesn't affect the solutions. Finally, in terms of the linear mapping that the matrix represents, what we are doing is changing the sets of basis vectors, keeping the mapping fixed, in such a way that the matrix that represents the mapping becomes a bit more acceptable to our taste.

Now in (3.2.11) we divide through the second row by a_{22} (again blissfully assuming that a_{22} is not zero) to create a 1 in the [2,2] position. Then we use that 1 to create zeroes (just one zero in this case) below it in the second column by letting $t = a_{32}$ and doing

$$\overrightarrow{\text{row}}(3) := \overrightarrow{\text{row}}(3) - t * \overrightarrow{\text{row}}(2). \tag{3.2.12}$$

Finally, we divide the third row by the [3,3] element to obtain

$$\left[\begin{array}{ccccc|c} 1 & * & * & * & * & * \\ 0 & 1 & * & * & * & * \\ 0 & 0 & 1 & * & * & * \end{array}\right]. \tag{3.2.13}$$

This is the end of the so-called *forward solution*, the first phase of the process of obtaining the general solution.

Again, let's think about the system of equations that is represented here. What is special about them is that the first unknown does not appear in the second equation, and neither the first nor the second unknown appears in the third equation.

To finish the solution of such a system of equations we would use the third equation to express x_3 in terms of x_4 and x_5, then the second equation would give us x_2 in terms of x_4 and x_5, and finally the first equation would yield x_1, also expressed in terms of x_4 and x_5. Hence, we would say that x_4 and x_5 are *free*, and that the others are determined by them. More precisely, we should say that the kernel of A has a two-dimensional basis.

Let's see how all of this will look if we were to operate directly on the matrix (3.2.13). The second phase of the solution, that we are now beginning, is called the *backwards substitution*, because we start with the last equation and work backwards.

First we use the 1 in the [3,3] position to create zeros in the third column above that 1. To do this we let $t = a_{23}$ and then we do

$$\overrightarrow{\text{row}}(2) := \overrightarrow{\text{row}}(2) - t * \overrightarrow{\text{row}}(3). \tag{3.2.14}$$

Then we let $t = a_{13}$ and set

$$\overrightarrow{\text{row}}\,1 := \overrightarrow{\text{row}}(1) - t * \overrightarrow{\text{row}}(3) \tag{3.2.15}$$

resulting in

$$\left[\begin{array}{ccccc|c} 1 & * & 0 & * & * & * \\ 0 & 1 & 1 & * & * & * \\ 0 & 0 & 1 & * & * & * \end{array}\right]. \tag{3.2.16}$$

Observe that now x_3 does not appear in the equations before it. Next we use the 1 in the [2,2] position to create a zero in column 2 above that 1 by letting $t = a_{12}$ and

$$\overrightarrow{\text{row}}(1) := \overrightarrow{\text{row}}(1) - t * \overrightarrow{\text{row}}(2). \tag{3.2.17}$$

Of course, none of our previously constructed zeros gets wrecked by this process, and we have arrived at the *reduced echelon form* of the original system of equations

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & * & * & * \\ 0 & 1 & 0 & * & * & * \\ 0 & 0 & 1 & * & * & * \end{array}\right]. \tag{3.2.18}$$

We need to be more careful about the next step, so it's time to use numbers instead of asterisks. For instance, suppose that we have now arrived at

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & a_{14} & a_{15} & a_{16} \\ 0 & 1 & 0 & a_{24} & a_{25} & a_{26} \\ 0 & 0 & 1 & a_{34} & a_{35} & a_{36} \end{array}\right]. \tag{3.2.19}$$

Each of the unknowns is expressible in terms of x_4 and x_5:

$$\begin{array}{rrcr} x_1 = & a_{16} - a_{14}x_4 & - & a_{15}x_5 \\ x_2 = & a_{26} - a_{24}x_4 & - & a_{25}x_5 \\ x_3 = & a_{36} - a_{34}x_4 & - & a_{35}x_5 \\ x_4 = & x_4 & & \\ x_5 = & & & x_5. \end{array} \tag{3.2.20}$$

This means that we have found the general solution of the given system by finding a particular solution and a pair of basis vectors for the kernel of A. They are, respectively, the vector and the two columns of the matrix shown below:

$$\left[\begin{array}{c} a_{16} \\ a_{26} \\ a_{36} \\ 0 \\ 0 \end{array}\right], \qquad \left[\begin{array}{cc} a_{14} & a_{15} \\ a_{24} & a_{25} \\ a_{34} & a_{35} \\ -1 & 0 \\ 0 & -1 \end{array}\right]. \tag{3.2.21}$$

As this shows, we find a particular solution by filling in extra zeros in the last column until its length matches the number (five in this case) of unknowns. We find a basis matrix (*i.e.*, a matrix whose columns are a basis for the kernel of A) by extending the fourth and fifth columns of the reduced row echelon form of A with $-I$, where I is the identity matrix whose size is equal to the nullity of the system, in this case 2.

It's time to deal with the case where one of the $*$'s that we divide by is actually a zero. In fact we will have to discuss rather carefully what we mean by zero. In numerical work on computers, in the presence of rounding errors, it is unreasonable to expect a 0 to be exactly zero. Instead we will set a certain threshold level, and numbers that are smaller than that will be declared to be zero. The hard part will be the determination of the right threshold,

but let's postpone that question for a while, and make the convention that in this and the following sections, when we speak of matrix entries being zero, we will mean that their size is below our current tolerance level.

With that understanding, suppose we are carrying out the row reduction of a certain system, and we've arrived at a stage like this:

$$\left[\begin{array}{cccccc|c} 1 & * & * & * & \cdots & * & * \\ 0 & 1 & * & * & \cdots & * & * \\ 0 & 0 & 0 & * & \cdots & * & * \\ 0 & 0 & * & * & \cdots & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array}\right] . \tag{3.2.22}$$

Normally, the next step would be to divide by a_{33}, but it is zero. This means that x_3 happens not to appear in the third equation. However, x_3 might appear in some later equation.

If so, we can renumber the equations so that later equation becomes the third equation, and continue the process. In the matrix, this means that we would exchange two rows, so as to bring a nonzero entry into the [3,3] position, and continue.

It is possible, though, that x_3 does not appear in *any* later equation. Then all entries $a_{i3} = 0$ for $i \geq 3$. Then we could ask for some other unknown x_j for $j > 3$ that does appear in some equation later than the third. In the matrix, this amounts to searching through the whole rectangle that lies "southeast" of the [3,3] position, extending over to, but not beyond, the vertical line, to find a nonzero entry, if there is one.

If a_{ij} is such a nonzero entry, then we want next to bring a_{ij} into the *pivot* position [3,3]. We can do this in two steps. First we exchange rows 3 and i (interchange two equations). Second, exchange columns 3 and j (interchange the numbers of the unknowns, so that x_3 becomes x'_j and x_j becomes x'_3). We must remember somehow that we renumbered the unknowns, so we'll be able to recognize the answers when we see them. The calculation can now proceed from the rejuvenated pivot element in the [3,3] position.

Else, it may happen that the rectangle southeast of [3,3] consists entirely of zeros, like this:

$$\left[\begin{array}{cccccc|c} 1 & * & * & * & * & \cdots & * \\ 0 & 1 & * & * & * & \cdots & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & * \\ 0 & 0 & 0 & 0 & 0 & \cdots & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array}\right] . \tag{3.2.23}$$

What then? We're finished with the forward solution. The equations from the third onwards have only zeros on their left-hand sides. If any solutions at all exist, then those equations had better have only zeros on their right-hand sides also. The last three $*$'s in the last column (and all the entries below them) must all be zeros, or the calculation halts with the announcement that the input system was *inconsistent* (*i.e*, has no solutions). If the system is consistent, then of course we ignore the final rows of zeros, and we do the backwards solution just as in the preceding case.

It follows that in all cases, whether there are more equations than unknowns, fewer, or the same number, the backwards solution always begins with a matrix that has a diagonal of 1's stretching from top to bottom (the bottom may have moved up, though!), with only zero entries below the 1's on the diagonal.

Speaking in theoretical, rather than practical terms for a moment, the number of nonzero rows in the coefficient matrix at the end of the forward solution phase is the rank of the matrix. Speaking practically again, this number simply represents a number of rows beyond which we cannot do any more reductions because the matrix entries are all indistinguishable from zero. Perhaps a good name for it is *pseudorank*. The pseudorank should be thought of then, not as some interesting property of the matrix that we have just computed, but as the number of rows we were able to reduce before roundoff errors became overwhelming.

Exercises 3.2

In problems 1–5, solve each system of equations by transforming the coefficient matrix to reduced echelon form. To "solve" a system means either to show that no solution exists or to find all possible solutions. In the latter case, exhibit a particular solution and a basis for the kernel of the coefficient matrix.

1.
$$\begin{aligned} 2x &- y + z &&= 6 \\ 3x &+ y + 2z &&= 3 \\ 7x &- y + 4z &&= 15 \\ 8x &+ y + 5z &&= 12 \end{aligned} \tag{3.2.24}$$

2.
$$\begin{aligned} x + y + z + q &= 0 \\ x - y - z - q &= 0 \end{aligned} \tag{3.2.25}$$

3.
$$\begin{aligned} x + y + z &= 3 \\ 3x - y - 2z &= -1 \\ 5x + y &= 7 \end{aligned} \tag{3.2.26}$$

4.
$$\begin{aligned} 3x + u + v + w + t &= 1 \\ x - u + 2v + w - 3t &= 2 \end{aligned} \tag{3.2.27}$$

5.
$$\begin{aligned} x + 3y - z &= 4 \\ 2x - y + 2z &= 6 \\ x + y + z &= 6 \\ 3x - y - z &= -2 \end{aligned} \tag{3.2.28}$$

6. Construct a set of four equations in four unknowns, of rank two.

7. Construct a set of four equations in three unknowns, with a unique solution.

8. Construct a system of homogeneous equations that has a three-dimensional vector space of solutions.

9. Under precisely what conditions is the set of all solutions of the system $A\mathbf{x} = \mathbf{b}$ a vector space?

10. Given a set of m vectors in n space, describe an algorithm that will extract a maximal subset of linearly independent vectors.

11. Given a set of m vectors, and one more vector $\mathbf{w}$, describe an algorithm that will decide whether or not $\mathbf{w}$ is in the vector subspace spanned by the given set.

3.3 Building blocks for the linear equation solver

Let's now try to amplify some of the points raised by the informal discussion of the procedure for solving linear equations, with a view towards the development of a formal algorithm.

First, let's deal with the fact that a diagonal element might be zero (in the fuzzy sense defined in the previous section) at the time when we want to divide by it.

Consider the moment when we are carrying out the forward solution, we have made $i-1$ 1's down the diagonal, all the entries below the 1's are zeros, and next we want to put a 1 into the $[i, i]$ position and use that 1 as a pivot to reduce the entries below it to zeros.

Previously we had said that this could be done by dividing through the ith row by the $[i, i]$ element, unless that element is zero, in which case we carry out a search for some nonzero element in the rectangle that lies southeast of $[i, i]$ in the matrix. After careful analysis, it turns out that an even more conservative approach is better: the best procedure consists in searching through the entire southeast rectangle, *whether or not* the $[i, i]$ element is zero, to find the largest matrix element in absolute value.

If this complete search is done every time, whether or not the $[i, i]$ element is zero, then it develops that the sizes of the matrix elements do not grow very much as the algorithm unfolds, and the growth of the numerical errors is also kept to a minimum.

If that largest element found in the rectangle is, say, a_{uv}, then we bring a_{uv} into the pivot position $[i, i]$ by interchanging rows u and i (renumbering the equations) and interchanging columns v and i (renumbering the unknowns). Then we proceed as before.

It may seem wasteful to make a policy of carrying out a complete search of the rectangle whenever we are ready to find the next pivot element, and especially even if a nonzero element already occupies the pivot position anyway, without a search, but it turns out that the extra labor is well rewarded with optimum numerical accuracy and stability.

If we are solving m equations in n unknowns, then we need to carry along an extra array of length n. Let's call it τ_j, $j = 1, \ldots, n$. This array will keep a record of the column interchanges that we do as we do them, so that in the end we will be able to identify the output. Initially, we put $\tau_j = j$ for $j = 1, \ldots, n$. If at a certain moment we are about to interchange, say, the pth column and the qth column, then we will also interchange the entries τ_p and τ_q. At all times then, τ_j will hold the number of the column where the current jth column really belongs.

It must be noted that there is a fundamental difference between the interchange of rows and the interchange of columns. An interchange of two rows corresponds simply to listing the equations that we are trying to solve in a slightly different sequence, but has no effect on the solutions. On the other hand, an interchange of two columns amounts to renumbering two of the unknowns. Hence we must keep track of the column interchanges while we are doing them, so we'll be able to tell which unknown is which at output time, but we don't need to record row interchanges.

At the end of the calculation then, the output arrays will have to be shuffled. The reader might want to think about how do carry out that rearrangement, and we will return to it in section 3.6 under the heading of "to unscramble the eggs".

The next item to consider is that we would like our program to be able to solve not just one system $A\mathbf{x} = \mathbf{b}$, but several systems of simultaneous equations, each of the form $A\mathbf{x} = \mathbf{b}$, where the left-hand sides are all the same, but the right-hand sides are different. The data for our program will therefore be an m by $n+p$ matrix whose first n columns will contain the coefficient matrix A and whose last p columns will be the p different right-hand side vectors $\mathbf{b}$.

Why are we allowing several different right sides? Some of the main customers for our program will be matrices A whose inverses we want to calculate. To find, say, the first column of the inverse of A we want to solve $A\mathbf{x} = \mathbf{b}$, where $\mathbf{b}$ is the first column of the identity matrix. For the second column of the inverse, $\mathbf{b}$ would be the second column of the identity matrix, and so on. Hence, to find A^{-1}, if A is an $n \times n$ matrix, we must solve n systems of simultaneous equations each having the same left-hand side A, but with n different right-hand side vectors.

It is convenient to solve all n of these systems at once because the reduction that we apply to A itself to bring it into reduced echelon form is useful in solving all n of these systems, and we avoid having to repeat that part of the job n times. Thus, for matrix inversion, and for other purposes too, it is very handy to have the capability of solving several systems with a common left-hand side at once.

The next point concerns the linear array τ that we are going to use to keep track of the column interchanges. Instead of storing it in its own private array, it's easier to adjoin it to the matrix A that we're working on, as an extra row, for then when we interchange two columns we will automatically interchange the corresponding elements of τ, and thereby avoid separate programming.

This means that the full matrix that we will be working with in our program will be $(m+1) \times (n+p)$ if we are solving p systems of m equations in n unknowns with p right-hand sides. In the program itself, let's call this matrix C. So C will be thought of as being partitioned into blocks of sizes as shown below:

$$C = \left[\begin{array}{c|c} A:\ m \times n & RHS:\ m \times p \\ \hline \tau:\ 1 \times n & 0:\ 1 \times p \end{array}\right]. \tag{3.3.1}$$

Now a good way to begin the writing of a program such as the general-purpose matrix

analysis program that we now have in mind is to consider the different procedures, or modules, into which it may be broken up. We suggest that the individual blocks that we are about to discuss should be written as separate subroutines, each with its own clearly defined input and output, each with its own documentation, and each with its own local variable names. They should then be tested one at a time, by giving them small, suitable test problems. If this is done, then the main routine won't be much more than a string of calls to the various blocks.

1. Procedure `searchmat(C,r,s,i1,j1,i2,j2)`

This routine is given an $r \times s$ array C, and two positions in the matrix, say $[i_1, j_1]$ and $[i_2, j_2]$. It then carries out a search of the rectangular submatrix of C whose northwest corner is at $[i_1, j_1]$ and whose southeast corner is at $[i_2, j_2]$, inclusive, in order to find an element of largest absolute value that lives in that rectangle. The subroutine returns this element of largest magnitude, as `big`, and the row and column in which it lives, as `iloc, jloc`.

Subroutine `searchmat` will be called in at least two different places in the main routine. First, it will do the search for the next pivot element in the southeast rectangle. Second, it can be used to determine if the equations are consistent by searching the right-hand sides of equations $r+1, \ldots, m$ (r is the pseudorank) to see if they are all zero (*i.e.*, below our tolerance level).

2. Procedure `switchrow(C,r,s,i,j,k,l)`

The program is given an $r \times s$ matrix C, and four integers i, j, k and l. The subroutine interchanges rows i and j of C, between columns k and l inclusive, and returns a variable called `sign` with a value of -1, unless $i = j$, in which case it does nothing to C and returns a $+1$ in `sign`.

3. Procedure `switchcol(C,r,s,i,j,k,l)`

This subroutine is like the previous one, except it interchanges columns i and j of C, between rows k and l inclusive. It also returns a variable called `sign` with a value of -1, unless $i = j$, in which case it does nothing to C and returns a $+1$ in `sign`.

The subroutines `switchrow` and `switchcol` are used during the forward solution in the obvious way, and again after the back solution has been done, to unscramble the output (see procedure 5 below).

4. Procedure `pivot(C,r,s,i,k,u)`

Given the $r \times s$ matrix C, and three integers i, k and u, the subroutine assumes that $C_{ii} = 1$. It then stores C_{ki} in the local variable tm sets C_{ki} to zero, and reduces row k of C, in columns u to s, by doing the operation $C_{kq} := C_{kq} - t * C_{iq}$ for $q = u, \ldots, s$.

The use of the parameter u in this subroutine allows the flexibility for economical operation in both the forward and back solution. In the forward solution, we take $u = i + 1$ and it reduces the whole row k. In the back solution we use $u = n + 1$, because the rest of row k will have already been reduced.

5. Procedure `scale(C,r,s,i,u)`

Given an $r \times s$ matrix C and integers i and u, the routine stores C_{ii} in a variable called piv. It then does $C_{ij} := C_{ij}/piv$ for $j = u, \ldots, s$ and returns the value of piv.

6. Procedure `scramb(C,r,s,n)`

This procedure permutes the first n rows of the $r \times s$ matrix C according to the permutation that occupies the positions $C_{r1}, C_{r2}, \ldots, C_{rn}$ on input.

The use of this subroutine is explained in detail in section 3.6 (*q.v.*). Its purpose is to rearrange the rows of the output matrix that holds a basis for the kernel, and also the rows of the output matrix that holds particular solutions of the give system(s). After rearrangement the rows will correspond to the original numbering of the unknowns, thereby compensating for the renumbering that was induced by column interchanges during the forward solution. This subroutine poses some interesting questions if we require that it should not use any additional array space beyond the input matrix itself.

7. Procedure `ident(C,r,s,i,j,n,q)`

This procedure inserts q times the $n \times n$ identity matrix into the $n \times n$ submatrix whose Northwest corner is at position $[i, j]$ of the $r \times s$ matrix C.

Now let's look at the assembly of these building blocks into a complete matrix analysis procedure called `matalg(C,r,s,m,n,p,opt,eps)`. Input items to it are:

- An $r \times s$ matrix C (as well as the values of r and s), whose Northwest $m \times n$ submatrix contains the matrix of coefficients of the system(s) of equations that are about to be solved. The values of m and n must also be provided to the procedure. It is assumed that $r = 1 + \max(m, n)$. Unless the inverse of the coefficient matrix is wanted, the Northeast $m \times p$ submatrix of C holds p different right-hand side vectors for which we want solutions.
- The numbers r, s, m, n and p.
- A parameter `option` that is equal to 1 if we want an inverse, equal to 2 if we want to see the determinant of the coefficient matrix (if square) as well as a basis for the kernel (if it is nontrivial) and a set of p particular solution vectors.
- A real parameter `eps` that is used to bound roundoff error.

Output items from the procedure `matalg` are:

- The pseudorank r
- The determinant `det` if $m = n$
- An $n \times r$ matrix `basis` , whose columns are a basis for the kernel of the coefficient matrix.
- An $n \times p$ matrix `partic`, whose columns are particular solution vectors for the given systems.

In case `opt` $= 1$ is chosen, the procedure will fill the last m columns and rows of C with an $m \times m$ identity matrix, set $p = n = m$, and proceed as before, leaving the inverse matrix in the same place, on output.

Let's remark on how the determinant is calculated. The reduction of the input matrix to echelon form in the forward solution phase entails the use of three kinds of operations. First we divide a row by a pivot element. Second, we multiply a row by a number and add it to another row. Third, we exchange a pair of rows or columns.

The first operation divides the determinant by that same pivot element. The second has no effect on the determinant. The third changes the sign of the determinant, at any rate if the rows or columns are distinct. At the end of the forward solution the matrix is upper triangular, with 1's on the diagonal, hence its determinant is clearly 1.

What must have been the value of the determinant of the input matrix? Clearly it must have been equal to the product of all the pivot elements that were used during the reduction, together with a plus or minus sign from the row or column interchanges.

Hence, to compute the determinant, we begin by setting `det` to 1. Then, each time a new pivot element is selected, we multiply `det` by it. Finally, whenever a pair of different rows or columns are interchanged we reverse the sign of `det`. Then `det` holds the determinant of the input matrix when the forward solution phase has ended.

Now we have described the basic modules out of which a general purpose program for linear equations can be constructed. In the next section we are going to discuss the vexing question of roundoff error and how to set the tolerance level below which entries are declared to be zero. A complete formal algorithm that ties together all of these modules, with control of rounding error, is given at the end of the next section.

EXERCISES 3.3

1. Make a test problem for the major program that you're writing by tracing through a solution the way the computer would:

   Take one of the systems that appears at the end of section 3.2. Transform it to reduced row echelon form step by step, being sure to carry out a complete search of the Southeast rectangle each time, and to interchange rows and columns to bring the largest element found into the pivot position. Record the column interchanges in τ, as described above. Record the status of the matrix C after each major loop so you'll be able to test your program thoroughly and easily.

2. Repeat problem 1 on a system of each major type: inconsistent, unique solution, many solutions.

3. Construct a formal algorithm that will invert a matrix, using no more array space than the matrix itself. The idea is that the input matrix is transformed, a column at a time, into the identity matrix, and the identity matrix is transformed, a column at a time, into the inverse. Why store all of the extra columns of the identity matrix? (Good luck!)

4. Show that a matrix A is of rank one if and only if its entries are of the form $A_{ij} = f_i g_j$ for all I and j.

5. Show that the operation $\overrightarrow{\text{row}}(i) := c * \overrightarrow{\text{row}}(j) + \overrightarrow{\text{row}}(i)$ applied to a matrix A has the same effect as first applying that same operation to the identity matrix I to get a certain matrix E, and then computing EA.

6. Show that the operation of scaling $\overrightarrow{\text{row}}(i)$:

$$a_{ik} := \frac{a_{ik}}{a_{ii}} \qquad k = 1, \ldots, n \tag{3.3.2}$$

   has the same effect as first dividing the ith row of the identity matrix by a_{ii} to get a certain matrix E, and then computing EA.

7. Suppose we do a complete forward solution without ever searching or interchanging rows or columns. Show that the forward solution amounts to discovering a lower triangular matrix L and an upper triangular matrix U such that $LA = U$ (think of L as a product of several matrices E such as you found in the preceding two problems).

3.4 How big is zero?

The story of the linear algebra subroutine has just two pieces untold: the first concerns how small we will allow a number to be without calling it zero, and the second concerns the rearrangement of the output to compensate for interchanges of rows and columns that are done during the row-echelon reduction.

The main reduction loop begins with a search of the rectangle that lies Southeast of the pivot position $[i, i]$, in order to locate the largest element that lives there and to use it for the next pivot. If that element is zero, the forward solution halts because the remaining pivot candidates are all zero.

But "how zero" do they have to be? Certainly it would be to much to insist, when working with sixteen decimal digits, that a number should be exactly equal to zero. A little more natural would be to declare that any number that is no larger than the size of the accumulated roundoff error in the calculation should be declared to be zero, since our microscope lens would then be too clouded to tell the difference.

It is important that we should know how large roundoff error is, or might be. Indeed, if we set too small a threshold, then numbers that "really are" zero will slip through, the calculation will continue after it should have terminated because of unreliability of the computed entries, and so forth. If the threshold is too large, we will declare numbers to be zero that aren't, and our numerical solution will terminate too quickly because the computed matrix elements will be declared to be unreliable when really they are perfectly OK.

The phenomenon of roundoff error occurs because of the finite size of a computer word. If a word consists of d binary digits, then when two d-digit binary numbers are multiplied

together, the answer that should be $2d$ bits long gets rounded off to d bits when it is stored. By doing so we incur a rounding error whose size is at most 1 unit in the $(d+1)$st place.

Then we proceed to add that answer to other numbers with errors in them, and to multiply, divide, and so forth, some large number of times. The accumulation of all of this rounding error can be quite significant in an extended computation, particularly when a good deal of cancellation occurs from subtraction of nearly equal quantities.

The question is to determine the level of rounding error that is present, while the calculation is proceeding. Then, when we arrive at a stage where the numbers of interest are about the same size as the rounding errors that may be present in them, we had better halt the calculation.

How can we estimate, during the course of a calculation, the size of the accumulated roundoff error? There are a number of theoretical *a priori* estimates for this error, but in any given computation these would tend to be overly conservative, and we would usually terminate the calculation too soon, thinking that the errors were worse than they actually were.

We prefer to let the computer estimate the error for us while it's doing the calculation. True, it will have to do more work, but we would rather have it work a little harder if the result will be that we get more reliable answers.

Here is a proposal for estimating the accumulated rounding error during the progress of a computation. This method was suggested by Professors Nijenhuis and Wilf. We carry along an additional matrix of the same size as the matrix C, the one that has the coefficients and right-hand sides of the equations that we are solving. In this extra matrix we are going to keep estimates of the roundoff error in each of the elements of the matrix C.

In other words, we are going to keep two whole matrices, one of which will contain the coefficients and the right-hand sides of the equations, and the other of which will contain estimates of the roundoff error that is present in the elements of the first one.

At any time during the calculation that we want to know how reliable a certain matrix entry is, we'll need only to look at the corresponding entry of the error matrix to find out.

Let's call this auxiliary matrix R (as in roundoff). Initially an element R_{ij} might be as large as $2^{-d}|C_{ij}|$ in magnitude, and of either sign. Therefore, to initialize the R matrix we choose a number uniformly at random in the interval $[-|2^{-d}C_{ij}|, |2^{-d}C_{ij}|]$, and store it in R_{ij} for each i and j. Hence, to begin with, the matrix R is set to randomly chosen values in the range in which the actual roundoff errors lie.

Then, as the calculation unfolds, we do arithmetic on the matrix C of two kinds. We either scale a row by dividing it through by the pivot element, or we pivot a row against another row. In each case let's look at the effect that the operation has on the corresponding roundoff error estimator in the R matrix.

In the first case, consider a scaling operation, in which a certain row is divided by the pivot element. Specifically, suppose we are dividing row i through by the element C_{ii}, and

let R_{ii} be the corresponding entry of the error matrix. Then, in view of the fact that

$$\frac{C_{ij}+R_{ij}}{C_{ii}+R_{ii}} = \frac{C_{ij}}{C_{ii}} + \frac{R_{ij}}{C_{ii}} - \frac{R_{ii}C_{ij}}{C_{ii}^2} + \text{ terms involving products of two or more errors} \quad (3.4.1)$$

we see that the error entries R_{ij} in the row that is being divided through by C_{ii} should be computed as

$$R_{ij} := \frac{R_{ij}}{C_{ii}} - \frac{R_{ii}C_{ij}}{C_{ii}^2}. \quad (3.4.2)$$

In the second case, suppose we are doing a pivoting operation on the kth row. Then for each column q we do the operation $C_{kq} := C_{kq} - t * C_{iq}$, where $t = C_{ki}$. Now let's replace C_{kq} by $C_{kq} + R_{kq}$, replace C_{iq} by $C_{iq} + R_{iq}$ and replace t by $t + t'$ (where $t' = R_{ki}$). Then substitute these expressions into the pivot operation above, and keep terms that are of first order in the errors (*i.e.*, that do not involve products of two of the errors).

Then $C_{kq} + R_{kq}$ is replaced by

$$\begin{aligned} C_{kq} + R_{kq} - (t+t') * (C_{iq} + R_{iq}) &= (C_{kq} - t * C_{iq}) + (R_{kq} - t * R_{iq} - t' * C_{iq}) \\ &= (\text{new } C_{kq}) + (\text{new error } R_{kq}). \end{aligned} \quad (3.4.3)$$

It follows that as a result of the pivoting, the error estimator is updated as follows:

$$R_{kq} := R_{kq} - C_{ki} * R_{iq} - R_{ki} * C_{iq}. \quad (3.4.4)$$

Equations (3.4.2) and (3.4.4) completely describe the evolution of the R matrix. It begins life as random roundoff error; it gets modified along with the matrix elements whose errors are being estimated, and in return, we are supplied with good error estimates of each entry while the calculation proceeds.

Before each scaling and pivoting sequence we will need to update the R matrix as described above. Then, when we search the now-famous Southeast rectangle for the new pivot element we accept it if it is larger in absolute value that its corresponding roundoff estimator, and otherwise we declare the rectangle to be identically zero and halt the forward solution.

The R matrix is also used to check the consistency of the input system. At the end of the forward solution all rows of the coefficient matrix from a certain row onwards are filled with zeros, in the sense that the entries are below the level of their corresponding roundoff estimator. Then the corresponding right-hand side vector entries should also be zero in the same sense, else as far as the algorithm can tell, the input system was inconsistent. With typical ambiguity of course, this means either that the input system was "really" inconsistent, or just that rounding errors have built up so severely that we cannot decide on consistency, and continuation of the "solution" would be meaningless.

Algorithm `matalg(C,r,s,m,n,p,opt,eps)`. The algorithm operates on the matrix C, which is of dimension $r \times s$, where $r = \max(m,n)+1$. It solves p systems of m equations in n unknowns, unless `opt`$= 1$, in which case it will calculate the inverse of the matrix in the first $m = n$ rows and columns of C.

```
matalg:=proc(C,r,s,m,n,p,opt,eps)
local R,i,j,Det,Done,ii,jj,Z,k,psrank;
# if opt = 1 that means inverse is expected
if opt=1 then ident(C,r,s,1,n+1,n,1) fi;
# initialize random error matrix
R:=matrix(r,s,(i,j)->0.000000000001*(rand()-500000000000)*eps*C[i,j]);
# set row permutation to the identity
for j from 1 to n do C[r,j]:=j od;
# begin forward solution
Det:=1; Done:=false; i:=0;
while ((i<m) and not(Done))do
# find largest in SE rectangle
  Z:=searchmat(C,r,s,i+1,i+1,m,n);ii:=Z[1][1]; jj:=Z[1][2];
  if abs(Z[2])>abs(R[ii,jj]) then
    i:=i+1;
# switch rows
    Det:=Det*switchrow(C,r,s,i,ii,i,s);
    Z:=switchrow(R,r,s,i,ii,i,s);
# switch columns
    Det:=Det*switchcol(C,r,s,i,jj,1,r);
    Z:=switchcol(R,r,s,i,jj,1,r);
# divide by pivot element
    Z:=scaler(C,R,r,s,i,i);
    Det:=Det*scale(C,r,s,i,i);
    for k from i+1 to m do
# reduce row k against row i
      Z:=pivotr(C,R,r,s,i,k,i+1);
      Z:=pivot(C,r,s,i,k,i+1);
    od;
  else Done:=true fi;
od;
psrank:=i;
# end forward solution; begin consistency check
if psrank<m then
  Det:=0;
  for j from 1 to p do
# check that right hand sides are 0 for i>psrank
    Z:=searchmat(C,r,s,psrank+1,n+j,m,n+j);
    if abs(Z[2])>abs(R[Z[1][1],Z[1][2]]) then
       printf("Right hand side %d is inconsistent",j);
       return;
    fi;
  od;
fi;
# equations are consistent, do back solution
```

```
for j from psrank to 2 by -1 do
   for i from 1 to j-1 do
      Z:=pivotr(C,R,r,s,j,i,psrank+1);
      Z:=pivot(C,r,s,j,i,psrank+1);
      C[i,j]:=0; R[i,j]:=0;
   od;
od;
# end back solution, insert minus identity in basis
if psrank<n then
# fill bottom of basis matrix with -I
  Z:=ident(C,r,s,psrank+1,psrank+1,n-psrank,-1);
# fill under right-hand sides with zeroes
  for i from psrank+1 to n do for j from n+1 to s do C[i,j]:=0 od od;
# fill under R matrix with zeroes
  for i from psrank+1 to n do for j from n-psrank to s do R[i,j]:=0 od od;
fi;
# permute rows prior to output
Z:=scramb(C,r,s,n);
# copy row r of C to row r of R
for j from 1 to n do R[r,j]:=C[r,j] od;
Z:=scramb(R,r,s,n);
return(Det,psrank,evalm(R));
end;
```

If the procedure terminates successfully, it returns a list containing three items: the first is the determinant (if there is one), the second is the pseudorank of the coefficient matrix, and the third is the matrix of estimated roundoff errors. The matrix C (which is called *by name* in the procedure, which means that the input matrix is altered by the procedure) will contain a basis for the kernel of the coefficient matrix in columns $psrank + 1$ to n, and p particular solution vectors, one for each input right-hand side, in columns $n + 1$ to $n + p$.

Two new sub-procedures are called by this procedure, namely `scaler` and `pivotr`. These are called immediately before the action of `scale` or `pivot`, respectively, and their mission is to update the R matrix in accordance with equations (3.4.2) or (3.4.4) to take account of the impending scaling or pivoting operation .

Exercises 3.4

1. Break off from the complete algorithm above, the forward solution process. State it formally as algorithm `forwd`, list its global variables, and describe precisely its effect on them. Do the same for the backwards solution.

2. When the program runs, it gives the solutions and their roundoff error estimates. Work out an elegant way to print the answers and the error estimates. For instance, there's no point in giving 12 digits of roundoff error estimate. That's too much. Just print the number of digits of the answers that can be trusted. How would you do that? Write subroutine `prnt` that will carry it out.

3. Suppose you want to re-run a problem, with a different set of random numbers in the roundoff matrix initialization. How would you do that? Run one problem three or four times to see how sensitive the roundoff estimates are to the choice of random values that start them off.

4. Show your program to a person who is knowledgeable in programming, but who is not one of your classmates. Ask that person to use your program to solve some set of three simultaneous equations in five unknowns.

   Do not answer any questions verbally about how the program works or how to use it. Refer all such questions to your written documentation that accompanies the program.

   If the other person is able to run your program and understand the answers, award yourself a gold medal in documentation. Otherwise, improve your documentation and let the person try again. When successful, try again on someone else.

5. Select two vectors f, g of length 10 by choosing their elements at random. Form the 10×10 matrix of rank 1 whose elements are $f_i g_j$. Do this three times and add the resulting matrices to get a single 10×10 matrix of rank three.

   Run your program on the coefficient matrix you just constructed, in order to see if the program is smart enough to recognize a matrix of rank three when it sees one, by halting the forward solution with pseudorank = 3.

   Repeat the above experiment 50 times, and tabulate the frequencies with which your program "thought" that the 10×10 matrix had various ranks.

3.5 Operation count

With any numerical algorithm it is important to know how much work is involved in carrying it out. In this section we are going to estimate the labor involved in solving linear systems by the method of the previous sections.

Let's recognize two kinds of labor: arithmetic operations, and other operations, both as applied to elements of the matrix. The arithmetic operations are $+$, $-$, $\times$, $\div$, all lumped together, and by other operations we mean comparisons of size, movement of data, and other operations performed directly on the matrix elements. Of course there are many "other operations," not involving the matrix elements directly, such as augmenting counters, testing for completion of loops, etc., that go on during the reduction of the matrix, but the two categories above represent a good measure of the work done. We're not going to include the management of the roundoff error matrix R in our estimates, because its effect would be simply to double the labor involved. Hence, remember to double all of the estimates of the labor that we are about to derive if you're using the R matrix.

We consider a generic stage in the forward solution where we have been given m equations in n unknowns with p right-hand sides, and during the forward solution phase we have just arrived at the $[i, i]$ element.

The next thing to do is to search the Southeast rectangle for the largest element, The rectangle contains about $(m-i)*(m-i)$ elements. Hence the search requires that many comparisons.

Then we exchange two rows ($n+p-i$ operations), exchange two columns (m operations) and divide a row by the pivot element ($n+p-i$ arithmetic operations).

Next, for each of the $m-i-1$ rows below the ith, and for each of the $n+p-i$ elements of one of those rows, we do two arithmetic operations when we do the elementary row operation that produces a zero in the ith column. This requires, therefore, $2(n+p-i)(m-i-1)$ arithmetic operations.

For the forward phase of the solution, therefore, we count

$$A_f = \sum_{i=1}^{r} \{2(n+p-i)(m-i-1) + (n+p-i)\} \tag{3.5.1}$$

arithmetic operations altogether, where r is the pseudorank of the matrix, because the forward solution halts after the rth row with only zeros below.

The non-arithmetic operations in the forward phase amount to

$$N_f = \sum_{i=1}^{r} \{(m-i)(n-i) + (n+p-i) + m\}. \tag{3.5.2}$$

Let's leave these sums for a while, and go to the backwards phase of the solution. We do the columns in reverse order, from column r back to 1, and when we have arrived at a generic column j, we want to create zeroes in all of the positions above the 1 in the $[j,j]$ position.

To do this we perform the elementary row operation

$$\overrightarrow{\text{row}}(i) := \overrightarrow{\text{row}}(i) - A_{ij} * \overrightarrow{\text{row}}(j) \tag{3.5.3}$$

to each of the $j-1$ rows above row j. Let i be the number of one of these rows. Then, exactly how many elements of $\overrightarrow{\text{row}}(i)$ are acted upon by the elementary row operation above? Certainly the elements in $\overrightarrow{\text{row}}(i)$ that lie in columns 1 through $j-1$ are unaffected, because only zero elements are in $\overrightarrow{\text{row}}(j)$ below them, thanks to the forward reduction process.

Furthermore, the elements in $\overrightarrow{\text{row}}(i)$ that lie in columns $j+1$ through r are unaffected, for a different reason. Indeed, any such entry is already zero, because it lies above an entry of 1 in some diagonal position that has already had its turn in the back solution (remember that we're doing the columns in the sequence $r, r-1, \ldots, 1$). Not only is such an entry zero, but it remains zero, because the entry of $\overrightarrow{\text{row}}(j)$ below it is also zero, having previously been deleted by the action of a diagonal element below it.

Hence in $\overrightarrow{\text{row}}(i)$, the elements that are affected by the elementary row operation (3.5.3) are those that lie in columns j, $n-r, \ldots, n, n+1, \ldots, n+p$ (be sure to write [or modify!] the program so that the row reduction (3.5.3) acts only on those columns!). We have now

shown that exactly $N + p + r - 1$ entries of each row above $\overrightarrow{\text{row}}(j)$ are affected (note that the number is independent of j and i), so

$$A_b = \sum_{j=1}^{r} (n + p - r + 1)(j - 1) \tag{3.5.4}$$

arithmetic operations are done during the back solution, and no other operations.

It remains only to do the various sums, and for this purpose we recall that

$$\begin{aligned} \sum_{i=1}^{N} i &= \frac{N(N+1)}{2} \\ \sum_{i=1}^{N} i^2 &= \frac{N(N+1)(2N+1)}{6} \end{aligned} \tag{3.5.5}$$

Then it is straightforward to find the total number of arithmetic operations from $A_f + A_b$ as

$$\text{Arith}(m, n, p, r) = \frac{r^3}{6} - (2m + n + p - 5)\frac{r^2}{2} + ((n + p)(2m - 5/2) - m + 1/3)r \tag{3.5.6}$$

and the total of the non-arithmetic operations from N_f as

$$\text{NonArith}(m, n, p, r) = \frac{r^3}{3} - (m + n)\frac{r^2}{2} + (6mn + 3m + 3n + 6p - 2)\frac{r}{6}. \tag{3.5.7}$$

Let's look at a few important special cases. First, suppose we are solving one system of n equations in n unknowns that has a unique solution. Then we have $m = n = r$ and $p = 1$. We find that

$$\text{Arith}(n, n, 1, n) = \frac{2}{3}n^3 + \text{O}(n^2) \tag{3.5.8}$$

where $\text{O}(n^2)$ refers to some function of n that is bounded by a constant times n^2 as n grows large. Similarly, for the non-arithmetic operations on matrix elements we find $\frac{1}{3}n^3 + \text{O}(n^2)$ in this case.

It follows that a system of n equations can be solved for about one third of the price, in terms of arithmetic operations, of one matrix multiplication, at least if matrices are multiplied in the usual way (did you know that there is a faster way to multiply two matrices? We will see one later on).

Now what is the price of a matrix inversion by this method? Then we are solving n systems of n equations in n unknowns, all with the same left-hand side. Hence we have $r = m = n = p$, and we find that

$$\text{Arith}(n, n, n, n) = \frac{13}{6}n^3 + \text{O}(n^2). \tag{3.5.9}$$

Hence we can invert a matrix by this method for about the same price as solving 3.25 systems of equations! At first glance, it may seem as if the cost should be n times as great

because we are solving n systems. The great economy results, of course, from the common left-hand sides.

The cost of the non-arithmetic operations remains at $\frac{1}{3}n^3 + \mathrm{O}(n^2)$.

If we want only the determinant of a square matrix A, or want only the rank of A then we need to do only the forward solution, and we can save the cost of the back solution. We leave it to the reader to work out the cost of a determinant, or of finding the rank.

3.6 To unscramble the eggs

Now we have reached the last of the issues that needs to be discussed in order to plan a complete linear equation solving routine, and it concerns the rearrangement of the output so that it ends up in the right order.

During the operation of the forward solution algorithm we found it necessary to interchange rows and columns so that the largest element of the Southeast rectangle was brought into the pivot position. As we mentioned previously, we don't need to keep a record of the row interchanges, because they correspond simply to solving the equations in a different sequence. We must remember the column interchanges that occur along the way though, because each time we do one of them we are, in effect, renumbering the unknowns.

To remember the column interchanges we glue onto our array C an additional row, just for bookkeeping purposes. Its elements are called τ_j, $j = 1, \ldots, n$, and it is kept at the bottomof the matrix. More precisely, the elements of $\{\tau_j\}$ are the first n entries of the new last row of the matrix, where the row contains $n + p$ entries altogether, the last p of which are not used (refer to (3.3.1) to see the complete partitioning of the matrix C).

Now suppose we have arrived at the end of the back solution, and the answers to the original question are before us, except that they are scrambled. Here's an example of the kind of situation that might result:

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & a & b & c \\ 0 & 1 & 0 & d & e & f \\ 0 & 0 & 1 & g & h & k \\ \vdots & \vdots & \cdots & \cdots & \vdots & \vdots \\ 3 & 5 & 2 & 1 & 4 & * \end{array}\right]. \tag{3.6.1}$$

The matrix above represents schematically the reduced row echelon form in a problem where there are five unknowns ($n = 5$), the pseudorank $r = 3$, just one right-hand side vector is given ($p = 1$), and the permuations that were carried out on the columns are recorded in the array $\tau : [3, 5, 2, 1, 4]$ shown in the last row of the matrix as it would be storded in a computation.

The question now is, how do we express the general solution of the given set of equations? To find the answer, let's go back to the set of equations that (3.6.1) stands for. The first of these is

$$x_3 = c - ax_1 - bx_4 \tag{3.6.2}$$

because the numbering of the unknowns is as shown in the τ array. The next two equations are

$$\begin{aligned} x_5 &= f - dx_1 - ex_4 \\ x_2 &= k - gx_1 - hx_4. \end{aligned} \tag{3.6.3}$$

If we add the two trivial equations $x_1 = x_1$ and $x_4 = x_4$, then we get the whole solution vector which, after re-ordering the equations, can be written as

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ k \\ c \\ 0 \\ f \end{bmatrix} + (-x_1) * \begin{bmatrix} -1 \\ g \\ a \\ 0 \\ d \end{bmatrix} + (-x_4) * \begin{bmatrix} 0 \\ h \\ b \\ -1 \\ e \end{bmatrix}. \tag{3.6.4}$$

Now we are looking at a display of the output as we would like our subroutine to give it. The three vectors on the right side of (3.6.4) are, respectively, a particular solution of the given system of equations, and the two vectors of a basis for the kernel of the coefficient matrix.

The question can now be rephrased: exactly what operations must be done to the matrix shown in (3.6.1) that represents the situation at the end of the back solution, in order to obtain the three vectors in (3.6.4)?

The first things to do are, as we have previously noted, to append the negative of a 2×2 identity matrix to the bottom of the fourth and fifth columns of (3.6.1), and to lengthen the last column on the right by appending two more zeros. That brings us to the matrix

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & a & b & c \\ 0 & 1 & 0 & d & e & f \\ 0 & 0 & 1 & g & h & k \\ & & & -1 & 0 & 0 \\ & & & 0 & -1 & 0 \\ \hline [3 & 5 & 2 & 1 & 4] & * \end{array}\right]. \tag{3.6.5}$$

The first two of the three long columns above will be the basis for the kernel, and the last column above will be the particular solution, but only after we do the right rearrangement.

Now here is the punch line: the right rearrangement to do is to permute the rows of those three long columns as described by the permuation τ.

That means that the first row becomes the third, the second row becomes the fifth, the third row becomes the second, the fourth row is the new first, and the old fifth row is the new fourth. The reader is invited to carry out on the rows the interchanges just described, and to compare the result with what we want, namely with (3.6.4). It will be seen that we have gotten the desired result.

The point that is just a little surprising is that to undo the column interchanges that are recorded by τ, we do row interchanges. Just roughly, the reason for this is that we begin by wanting to solve $A\mathbf{x} = \mathbf{b}$, and instead we end up solving $(AE)\mathbf{y} = \mathbf{b}$, where E is

a matrix obtained from the identity by elementary column operations. Evidently, $\mathbf{x} = E\mathbf{y}$, which means that we must perform *row* operations on $\mathbf{y}$ to recover the answers in the right order.

Now we can leave the example above, and state the rule in general. We are given p systems of m simultaneous equations each, all having a commom $m \times n$ coefficient matrix A, in n unknowns. At the end of the back solution we will have before us a matrix of the form

$$\left[\begin{array}{c|c|c} I(r,r) & B(r,n-r) & P(r,p) \end{array}\right] \tag{3.6.6}$$

where $I(r,r)$ is the $r \times r$ identity matrix, r is the pseudorank of A, and B and P are matrices of the sizes shown.

We adjoin under B the negative of the $(n-r) \times (n-r)$ identity matrix, and under P we adjoin an $(n-r) \times p$ block of zeros. Next, we forget the identity matrix on the left, and we consider the entire remaining $n \times (n-r+p)$ matrix as a whole, call it T, say. Now we exchange the rows of T according to the permuation array τ. Preciesly, row 1 of T will be row τ_1 of the new T, row 2 will be row τ_2, ... Conceptually, we should regard the old T and the new T as occupying different areas of storage, so that the new T is just a rearrangement of the rows of the old.

Now the first $n-r$ columns of the new T are a basis for the kernel of A, and should be output as such, and the jth one of the last p columns of the new T is a particular solution of the jth one of the input systems of equations, and should be output as such.

Although conceptually we should think of the old T and the new T as occupying distinct arrays in memory, in fact it is perfectly possible to carry out the whole row interchange procedure described above in just one array, the one that holds T, without ever "stepping on our own toes," so let's consider that problem.

Suppose a linear array $a = [a_1, \ldots, a_n]$ is given, along with a permutation array $\tau = [\tau_1, \ldots, \tau_n]$. We want to rearrange the entries of the array a according to the permuation τ without using any additional array storage. Thus the present array a_1 will end up as the output a_{τ_1}, the initial a_2 will end up as a_{τ_2}, etc.

To do this with no extra array storage, let's first pick up the element a_1 and move it to a_{τ_1}, being careful to store the original a_{τ_1} in a temporary location t so it won't be destroyed. Next we move the contents of t to its destination, and so forth. After a certain number of steps (maybe only 1), we will be back to a_1.

Her's an example to help clarify the situation. Suppose the arrays a and τ at input time were:

$$\begin{aligned} a &= [5, 7, 13, 9, 2, 8] \\ \tau &= [3, 4, 5, 2, 1, 6]. \end{aligned} \tag{3.6.7}$$

So we move the 5 in position a_1 to position a_3 (after putting the 13 into a safe place), and then the 13 goes to position a_5 (after putting the 2 into a safe place) and the 2 is moved into position a_1, and we're back where we started. The a array now has become

$$a = [2, 7, 5, 9, 13, 8]. \tag{3.6.8}$$

The job, however, is not finished. Somehow we have to recognize that the elements a_2, a_4 and a_6 haven't yet been moved, while the others have been moved to their destinations. For this purpose we will flag the array positions. A convenient place to hang a flag is in the sign position of an entry of the array τ, since we're sure that the entries of τ are all supposed to be positive. Therefore, initially we'll change the signs of all of the entries of τ to make them negative. Then as elements are moved around in the a array we will reverse the sign of the corresponding entry of the τ array. In that way we can always begin the next block of entries of a to move by searching τ for a negative entry. When none exist, the job is finished.

Here's a complete algorithm, in Maple:

```
shuffle:=proc(a,tau,n) local i,j,t,q,u,v;
#permutes the entries of a according to the permutation tau
#
# flag entries of tau with negative signs
for i from 1 to n do tau[i]:=-tau[i] od;
for i from 1 to n do
# has entry i been moved?
  if tau[i]<0 then
# move the block of entries beginning at a[i]
    t:=i; q:=-tau[i]; tau[i]:=q;
    u:=a[i]; v:=u;
    while q<>t do
      v:=a[q]; a[q]:=u; tau[q]:=-tau[q];
      u:=v; q:=tau[q]; od;
    a[t]:=v;
  fi;
od;
return(1);
end;
```

The reader should carefully trace through the complete operation of this algorithm on the sample arrays shown above. In order to apply the method to the linear equation solving program, the entries $C[r+1,i]$, $i = 1\ldots,n$ are interpreted as τ_i, and the array a of length n whose entries are going to be moved is one of the columns $r+1,\ldots,n+p$ of the matrix C in rows $1,\ldots,n$.

3.7 Eigenvalues and eigenvectors of matrices

Our next topic in numerical linear algebra concerns the computation of the eigenvalues and eigenvectors of matrices. Until further notice, all matrices will be square. If A is $n \times n$, by an *eigenvector* of A we mean a vector $\mathbf{x} \neq \mathbf{0}$ such that

$$A\mathbf{x} = \lambda\mathbf{x} \tag{3.7.1}$$

where the scalar λ is called an *eigenvalue* of A. We say that the eigenvector $\mathbf{x}$ *corresponds to*, or belongs to, the eigenvalue λ. We will see that in fact the eigenvalues of A are properties of the linear mapping that A represents, rather than of the matrix A, so we can exploit changes of basis in the computation of eigenvalues.

For an example, consider the 2×2 matrix

$$A = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}. \tag{3.7.2}$$

If we write out the vector equation (3.7.1) for this matrix, it becomes the two scalar equations

$$\begin{aligned} 3x_1 - x_2 &= \lambda x_1 \\ -x_1 + 3x_2 &= \lambda x_2. \end{aligned} \tag{3.7.3}$$

These are two homogeneous equations in two unknowns, and therefore they have no solution other than the zero vector unless the determinant

$$\begin{vmatrix} 3-\lambda & -1 \\ -1 & 3-\lambda \end{vmatrix} \tag{3.7.4}$$

is equal to zero. This condition yields a quadratic equation for λ whose two roots are $\lambda = 2$ and $\lambda = 4$. These are the two eigenvalues of the matrix (3.7.2).

For the same 2×2 example, let's now find the eigenvectors (by a method that doesn't bear the slightest resemblance to the numerical method that we will discuss later). First, to find the eigenvector that belongs to the eigenvalue $\lambda = 2$, we go back to (3.7.3) and replace λ by 2 to obtain the two equations

$$\begin{aligned} x_1 - x_2 &= 0 \\ -x_1 + x_2 &= 0. \end{aligned} \tag{3.7.5}$$

These equations are, of course, redundant since λ was chosen to make them so. They are satisfied by any vector $\mathbf{x}$ of the form $c * [1, 1]$, where c is an arbitrary constant. If we refer back to the definition (3.7.1) of eigenvectors we notice that if $\mathbf{x}$ is an eigenvector then so is $c\mathbf{x}$, so eigenvectors are determined only up to constant multiples. The first eigenvector of our 2×2 matrix is therefore any multiple of the vector $[1, 1]$.

To find the eigenvector that belongs to the eigenvalue $\lambda = 4$, we return to (3.7.3), replace λ by 4, and solve the equations. The result is that any scalar multiple of the vector $[1, -1]$ is an eigenvector corresponding to the eigenvalue $\lambda = 4$.

The two statements that $[1, 1]$ is an eigenvector and that $[1, -1]$ is an eigenvector can either be written as two vector equations:

$$\begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = 4 \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{3.7.6}$$

or as a single matrix equation

$$\begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}. \tag{3.7.7}$$

Observe that the matrix equation (3.7.7) states that $AP = P\Lambda$, where A is the given 2×2 matrix, P is a (nonsingular) matrix whose columns are eigenvectors of A, and Λ is the diagonal matrix that carries the eigenvalues of A down the diagonal (in order corresponding to the eigenvectors in the columns of P). This matrix equation $AP = P\Lambda$ leads to one of the many important areas of application of the theory of eigenvalues, namely to the computation of functions of matrices.

Suppose we want to calculate A^{2147}, where A is the 2×2 matrix (3.7.2). A direct calculation, by raising A to higher and higher powers would take quite a while (although not as long as one might think at first sight! Exactly what powers of A would you compute? How many matrix multiplications would be required?).

A better way is to begin with the relation $AP = P\Lambda$ and to observe that in this case the matrix P is nonsingular, and so P has an inverse. Since P has the eigenvectors of A in its columns, the nonsingularity of P is equivalent to the linear independence of the eigenvectors. Hence we can write

$$A = P\Lambda P^{-1}. \tag{3.7.8}$$

This is called the *spectral representation* of A, and the set of eigenvalues is often called the *spectrum* of A.

Equation (3.7.8) is very helpful in computing powers of A. For instance

$$A^2 = (P\Lambda P^{-1})(P\Lambda P^{-1}) = P\Lambda^2 P^{-1},$$

and for every m, $A^m = P\Lambda^m P^{-1}$. It is of course quite easy to find high powers of the diagonal matrix Λ, because we need only raise the entries on the diagonal to that power. Thus for example,

$$A^{2147} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2^{2147} & 0 \\ 0 & 4^{2147} \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & -1/2 \end{bmatrix}. \tag{3.7.9}$$

Not only can we compute powers from the spectral representation (3.7.8), we can equally well obtain any polynomial in the matrix A. For instance,

$$13A^3 + 78A^{19} - 43A^{31} = P(13\Lambda^3 + 78\Lambda^{19} - 43\Lambda^{31})P^{-1}. \tag{3.7.10}$$

Indeed if f is any polynomial, then

$$f(A) = Pf(\Lambda)P^{-1} \tag{3.7.11}$$

and $f(\Lambda)$ is easy to calculate because it just has the numbers $f(\lambda_i)$ down the diagonal and zeros elsewhere.

Finally, it's just a short hop to the conclusion that (3.7.11) remains valid even if f is not a polynomial, but is represented by an everywhere-convergent powers series (we don't even need that much, but this statement suffices for our present purposes). So for instance, if A is the above 2×2 matrix, then

$$e^A = Pe^{\Lambda}P^{-1} \tag{3.7.12}$$

where e^{Λ} has e^2 and e^4 on its diagonal.

We have now arrived at a very important area of application of eigenvalues and eigenvectors, to the solution of systems of differential equations. A system of n linear simultaneous differential equations in n unknown functions can be written simply as $\mathbf{y}' = A\mathbf{y}$, with say $\mathbf{y}(0)$ given as initial data. The solution of this system of differential equations is $\mathbf{y}(t) = e^{At}\mathbf{y}(0)$, where the matrix e^{At} is calculated by writing $A = P\Lambda P^{-1}$ if possible, and then putting $e^{At} = Pe^{\Lambda t}P^{-1}$.

Hence, whenever we can find the spectral representation of a matrix A, we can calculate functions of the matrix and can solve differential equations that involve the matrix.

So, when can we find a spectral representation of a given $n \times n$ matrix A? If we can find a set of n linearly independent eigenvectors for A, then all we need to do is to arrange them in the columns of a new matrix P. Then P will be invertible, and we'll be all finished. Conversely, if we somehow have found a spectral representation of A *à la* (3.7.8), then the columns of P obviously do comprise a set of n independent eigenvectors of A.

That changes the question. What kind of an $n \times n$ matrix A has a set of n linearly independent eigenvectors? This is quite a hard problem, and we won't answer it completely. Instead, we give an example of a matrix that does *not* have as many independent eigenvectors as it "ought to," and then we'll specialize our discussion to a kind of matrix that is guaranteed to have a spectral representation.

For an example we don't have to look any further than

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}. \tag{3.7.13}$$

The reader will have no difficulty in checking that this matrix has just one eigenvalue, $\lambda = 0$, and that corresponding to that eigenvalue there is just one independent eigenvector, and therefore there is no spectral representation of this matrix.

Now first we're going to devote our attention to the *real symmetric* matrices. *i.e.*, to matrices A for which $A_{ij} = A_{ji}$ for all $i, j = 1, \ldots, n$. These matrices occur in many important applications, and they always have a spectral representation. Indeed, much more is true, as is shown by the following fundamental theorem of the subject, whose proof is deferred to section 3.9, where it will emerge (see Theorem 3.9.1) as a corollary of an algorithm.

Theorem 3.7.1 *(The Spectral Theorem) – Let A be an $n \times n$ real symmetric matrix. Then the eigenvalues and eigenvectors of A are real. Furthermore, we can always find a set of n eigenvectors of A that are pairwise orthogonal to each other (so they are surely independent).*

Recall that the eigenvectors of the symmetric 2×2 matrix (3.7.2) were $[1, 1]$ and $[1, -1]$, and these are indeed orthogonal to each other, though we didn't comment on it at the time.

We're going to follow a slightly unusual route now, that will lead us simultaneously to a proof of the fundamental theorem (the "spectral theorem") above, and to a very elegant

computer algorithm, called *the method of Jacobi*, for the computation of eigenvalues and eigenvectors of real symmetric matrices.

In the next section we will introduce a very special family of matrices, first studied by Jacobi, and we will examine their properties in some detail. Once we understand these properties, a proof of the spectral theorem will appear, with almost no additional work.

Following that we will show how the algorithm of Jacobi can be implemented on a computer, as a fast and pretty program in which all of the eigenvalues and eigenvectors of a real symmetric matrix are found simultaneously, and are delivered to your door as an orthogonal set.

Throughout these algorithms certain themes will recur. Specifically, we will see several situations in which we have to compute a certain angle and then carry out a rotation of space through that angle. Since the themes occur so often we are going to abstract from them certain basic modules of algorithms that will be used repeatedly.

This choice will greatly simplify the preparation of programs, but at a price, namely that each module will not always be exactly optimal in terms of machine time for execution in each application, although it will be nearly so. Consequently it was felt that the price was worth the benefit of greater universality. We'll discuss these points further, in context, as they arise.

3.8 The orthogonal matrices of Jacobi

A matrix P is called an *orthogonal* matrix if it is real, square, and if $P^{-1} = P^T$, *i.e.*, if $P^T P = PP^T = I$. If we visualize the way a matrix is multiplied by its transpose, it will be clear that an orthogonal matrix is one in which each of the rows (columns) is a unit vector and any two distinct rows (columns) are orthogonal to each other.

For example, the 2×2 matrix

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{3.8.1}$$

is an orthogonal matrix for every real θ.

We will soon prove that a real symmetric matrix always has a set of n pairwise orthogonal eigenvectors. If we take such as set of vectors, normalize them by dividing each by its length, and arrange them in the consecutive columns of a matrix P, then P will be an orthogonal matrix, and further we will have $AP = P\Lambda$. Since $P^T = P^{-1}$, we can multiply on the right by P^T and obtain

$$A = P\Lambda P^T, \tag{3.8.2}$$

and this is the spectral theorem for a symmetric matrix A.

Conversely, if we can find an orthogonal matrix P such that $P^T AP$ is a diagonal matrix D, then we will have found a complete set of pairwise orthogonal eigenvectors of A (the columns of P), and the eigenvalues of A (on the diagonal of D).

In this section we are going to describe a numerical procedure that will find such an orthogonal matrix, given a real symmetric matrix A. As soon as we prove that the method works, we will have proved the spectral theorem at the same time. Hence the method is of theoretical as well as algorithmic importance. It is important to notice that we will not have to find an eigenvalue, then find a corresponding eigenvector, then find another eigenvalue and another vector, etc. Instead, the whole orthogonal matrix whose columns are the desired vectors will creep up on us at once.

The first thing we have to do is to describe some special orthogonal matrices that will be used in the algorithm. Let n, p and q be given positive integers, with $n \geq 2$ and $p \neq q$, and let θ be a real number. We define the matrix $J_{pq}(\theta)$ by saying that J is just like the $n \times n$ identity matrix except that in the four positions that lie at the intersections of rows and columns p and q we find the entries (3.8.1).

More precisely, $J_{pq}(\theta)$ has in position $[p,p]$ the entry $\cos\theta$, it has $\sin\theta$ in the $[p,q]$ entry, $-\sin\theta$ in the $[q,p]$ entry, $\cos\theta$ in entry $[q,q]$, and otherwise it agrees with the identity matrix, as shown below:

$$\begin{array}{l} \\ \\ \\ \\ \\ \text{row } p \\ \\ \text{row } q \\ \\ \\ \end{array} \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \cdots & \ddots & \cdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \cdots & \cdots & \cdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 & 0 & 1 \end{bmatrix}. \tag{3.8.3}$$

Not only is $J_{pq}(\theta)$ an orthogonal matrix, there is a reasonably pleasant way to picture its action on n-dimensional space. Since the 2×2 matrix of (3.8.1) is the familiar rotation of the plane through an angle θ, we can say that the matrix $J_{pq}(\theta)$ carries out a special kind of rotation of n-dimensional space, namely one in which a certain plane, the plane of the pth and qth coordinate, is rotated through the angle θ, and the remaining coordinates are all left alone. Hence $J_{pq}(\theta)$ carries out a two-dimensional rotation of n-dimensional space.

These matrices of Jacobi turn out to be useful in a host of numerical algorithms for the eigenproblem. The first application that we'll make of them will be to the real symmetric matrices, but later we'll find that the same two-dimensional rotations will play important roles in the solution of non-symmetric problems as well.

First, let's see how they can help us with symmetric matrices. What we propose to do is the following. If a real symmetric matrix A is given, we will determine p, q, and the angle θ in such a way that the matrix JAJ^T is a little bit more diagonal (whatever that means!) than A is. It turns out that this can always be done, at any rate unless A is already diagonal, so we will have the germ of a numerical procedure for computing eigenvalues and eigenvectors.

Indeed, suppose we have found out how to determine such an angle θ, and let's then see what the whole process would look like. Starting with A, we would find p, q, and θ, and then the matrix JAJ^T is somehow a little more diagonal than A was. Now JAJ^T is still a symmetric matrix (try to transpose it and see what happens) so we can do it again. After finding another p, q and θ we will have $J''J'A(J''J')^T$ a bit "more diagonal" and so forth.

Now suppose that after some large number of repetitions of this process we find that the current matrix is very diagonal indeed, so that perhaps aside from roundoff error it is a diagonal matrix D. Then we will know that

$$D = (\text{product of all } J\text{'s used})A(\text{product of all } J\text{'s used})^T. \tag{3.8.4}$$

If we let P denote the product of all J's used, then we have $PAP^T = D$, so the columns of P will be the (approximate) eigenvectors of A and the diagonal elements of D will be its eigenvalues. The matrix P will automatically be an orthogonal matrix, since it is the product of such matrices, and the product of orthogonal matrices is always orthogonal (proof?).

That, at any rate, is the main idea of Jacobi's method (he introduced it in order to study planetary orbits!). Let's now fill in the details.

First we'll define what we mean by "more diagonal". For any square, real matrix A, let $\mathrm{Od}(A)$ denote the sum of the squares of the off-diagonal entries of A. From now on, instead of "B is more diagonal than A," we'll be able to say $\mathrm{Od}(B) < \mathrm{Od}(A)$, which is much more professional.

Now we claim that if A is a real symmetric matrix, and it is not already a diagonal matrix, then we can find p, q and θ such that $\mathrm{Od}(J_{pq}(\theta)AJ_{pq}(\theta)^T) < \mathrm{Od}(A)$. We'll do this by a very direct computation of the elements of JAJ^T (we'll need them anyway for the computer program), and then we will be able to see what the new value of Od is.

So fix p, q and θ. Then by direct multiplication of the matrix in (3.8.3) by A we find that

$$(JA)_{ij} = \begin{cases} (\cos\theta)A_{pj} + (\sin\theta)A_{qj} & \text{if } i = p \\ -(\sin\theta)A_{pj} + (\cos\theta)A_{qj} & \text{if } i = q \\ a_{ij} & \text{otherwise} \end{cases} \tag{3.8.5}$$

Then after one more multiplication, this time on the right by the transpose of the matrix in (3.8.3), we find that

$$(JAJ^T)_{ij} = \begin{cases} CA_{ip} + SA_{iq} & \text{if } i \notin \{p,q\};\ j = p \text{ or } i = p;\ j \notin \{p,q\} \\ -SA_{ip} + CA_{iq} & \text{if } i \notin \{p,q\};\ j = q \text{ or } i = q;\ j \notin \{p,q\} \\ C^2A_{pp} + 2SCA_{pq} + S^2A_{qq} & \text{if } i = j = p \\ S^2A_{pp} - 2SCA_{pq} + C^2A_{qq} & \text{if } i = j = q \\ CS(A_{qq} - A_{pp}) + (C^2 - S^2)A_{pq} & \text{if } i = p,\ j = q \text{ or } i = q\ j = p \\ a_{ij} & \text{otherwise.} \end{cases} \tag{3.8.6}$$

In (3.8.6) we have written C for $\cos\theta$ and S for $\sin\theta$.

Now we are going to choose the angle θ so that the elements A_{pq} and A_{qp} are reduced to zero, assuming that they were not already zero. To do this we refer to the formula in

(3.8.6) for A_{pq}, equate it to zero, and then solve for θ. The result is

$$\tan 2\theta = \frac{2A_{pq}}{A_{pp} - A_{qq}} \tag{3.8.7}$$

and we will choose the value of θ that lies between $-\frac{\pi}{4}$ and $\frac{\pi}{4}$.

With this value of θ, we will have reduced one single off-diagonal element of A to zero in the new symmetric matrix JAJ^T.

The full Jacobi algorithm consists in repeatedly executing these plane rotations, each time choosing the largest off-diagonal element A_{pq} and annihilating it by the choice (3.8.7) of θ. After each rotation, $\mathrm{Od}(A)$ will be a little smaller than it was before. We will prove that $\mathrm{Od}(A)$ converges to zero.

It is important to note that a plane rotation that annihilates A_{pq} may "revive" some other A_{rs} that was set to zero by an earlier plane rotation. Hence we should not think of the zero as "staying put".

Let's now see exactly what happens to $\mathrm{Od}(A)$ after a single rotation. If we sum the squares of all of the off-diagonal elements of JAJ^T using the formulas (3.8.6), but remembering that the new A_{pq}=0, then it's quite easy to check that the new sum of squares is exactly equal to the old sum of squares minus the squares of the two entries A_{pq} and A_{qp} that were reduced to zero. Hence we have

Theorem 3.8.1 *Let A be an $n \times n$ real, symmetric matrix that is not diagonal. If $A_{pq} \neq 0$ for some $p \neq q$, then we can choose θ as in equation (3.8.7) so that if $J = J_{pq}(\theta)$ then*

$$\mathrm{Od}(JAJ^T) = \mathrm{Od}(A) - 2A_{pq}^2 < \mathrm{Od}(A). \tag{3.8.8}$$

3.9 Convergence of the Jacobi method

We have now described the fundamental operation of the Jacobi algorithm, namely the plane rotation in n-dimensional space that sends a real symmetric matrix A into JAJ^T, and we have explicit formulas for the new matrix elements. There are still a number of quite substantive points to discuss before we will be able to assemble an efficient program for carrying out the method. However, in line with the philosophy that it is best to break up large programs into small, manageable chunks, we are now ready to prepare the first module of the Jacobi program.

What we want is to be able to execute the rotation through an angle θ, according to the formulas (3.8.6) of the previous section. This could be accomplished by a single subroutine that would take the symmetric matrix A and the sine and cosine of the rotation angle, and execute the operation (3.8.6).

If we keep later applications in mind, then the best choice for a module will be one that will, on demand, multiply a given not-necessarily-symmetric matrix on the left by J or on the right by J^T, depending on the call. This is one of the situations we referred to earlier

where the most universal choice of subroutine will not be the most economical one in every application, but we will get a lot of mileage out of this routine!

Hence, suppose we are given

1. an $n \times n$ real matrix A
2. the sine S and cosine C of a rotation angle
3. the plane $[p, q]$ of the rotation, and
4. a parameter `option` that will equal 1 if we want to do JA, and 2 if we want AJ^T.

The procedure will be called

Procedure `rotate(s,c,p,q,option);`

What the procedure will do is exactly this. If called with `option = 1`, it will multiply A on the left by J, according to the formulas (3.8.5), and exit. If `option = 2`, it will multiply A on the right by J^T. The Maple procedure is as follows:

```
rotate:=proc(s,c,p,q,opt) local j,temp;
 global  A,n;
 if opt=1 then
    for j from 1 to n do
        temp:=evalf(c*A[p,j]+s*A[q,j]);
      A[q,j]:=-s*A[p,j]+c*A[q,j];
      A[p,j]:=temp;
                      od
 else
 for j from 1 to n do
      temp:=c*A[j,p]+s*A[j,q];
    A[j,q]:=-s*A[j,p]+c*A[j,q];
    A[j,p]:=temp;
                   od
 fi;
 RETURN()
 end:
```

To carry out one iteration of the Jacobi method, we will have to call `rotate` twice, once with `option = 1` and then with `option = 2`.

The amount of computational labor that is done by this module is $\mathrm{O}(N)$ per call, since only two lines of the matrix are affected by its operation.

Next, let's prove that the results of applying one rotation after another do in fact converge to a diagonal matrix.

Theorem 3.9.1 *Let A be a real symmetric matrix. Suppose we follow the strategy of searching for the off-diagonal element of largest absolute value, choosing so as to zero out that element by carrying out a Jacobi rotation on A, and then repeating the whole process on the resulting matrix, etc. Then the sequence of matrices that is thereby obtained approaches a diagonal matrix D.*

Proof. At a certain stage of the iteration, let A_{pq} denote the off-diagonal element of largest absolute value. Since the maximum of any set of numbers is at least as big as the average of that set (proof?), it follows that the maximum of the squares of the off-diagonal elements of A is at least as big as the average square of an off-diagonal element. The average square is equal to the sum of the squares of the off-diagonal elements divided by the number of such elements, *i.e.*, divided by $n(n-1)$. Hence the average square is exactly $\mathrm{Od}(A)/(n(n-1))$, and therefore

$$A_{pq}^2 \geq \frac{\mathrm{Od}(A)}{n(n-1)}. \tag{3.9.1}$$

Now the effect of a single rotation of the matrix is to reduce $\mathrm{Od}(A)$ by $2A_{pq}^2$, so equation (3.8.8) yields

$$\begin{aligned} \mathrm{Od}(JAJ^T) &= \mathrm{Od}(A) - 2A_{pq}^2 \\ &\leq \mathrm{Od}(A) - \frac{2\,\mathrm{Od}(A)}{n(n-1} \\ &= \left(1 - \frac{2}{n(n-1)}\right) \mathrm{Od}(A). \end{aligned} \tag{3.9.2}$$

Hence a single rotation will reduce $\mathrm{Od}(A)$ by a multiplicative factor of $1 - 2/(n(n-1))$ at least. Since this factor is less than 1, it follows that the sum of squares of the off-diagonal entries approaches zero as the number of plane rotations grows without bound, completing the proof.

The proof told us even more since it produced a quantitative estimate of the rate at which $\mathrm{Od}(A)$ approaches zero. Indeed, after r rotations, the sum of squares will have dropped to at most

$$\left(1 - \frac{2}{n(n-1}\right)^r \mathrm{Od}(\text{original } A). \tag{3.9.3}$$

If we put $r = n(n-1)/2$, then we see that $\mathrm{Od}(A)$ has dropped by at least a factor of (approximately) e. Hence, after doing an average of one rotation per off-diagonal element, the function $\mathrm{Od}(A)$ is no more than $1/e$ times its original value. After doing an average of, say, t rotations per off-diagonal element (*i.e.*, $tn(n-1)/2$ rotations), the function $\mathrm{Od}(A)$ will have dropped to about e^{-t} times its original value. If we want it to drop to, say, 10^{-m} times its original value then we can expect to need no more than about $m(\ln 10)n(n-1)/2$ rotations.

To put it in very concrete terms, suppose we're working in double precision (12-digit) arithmetic and we are willing to decree that convergence has taken place if Od has been reduced by 10^{-12}. Then at most $12(\ln 10)n(n-1)/2 < 6(\ln 10)n^2 \approx 13.8n^2$ rotations will have to be done. Of course in practice we will be watching the function $\mathrm{Od}(A)$ as it drops,

so there won't be any need to know in advance how many iterations are needed. We can stop when the actual observed value is small enough. Still, it's comforting to know that at most $O(n^2)$ iterations will be enough to do the job.

Now let's re-direct our thoughts to the grand iterations process itself. At each step we apply a rotation matrix to the current symmetric matrix in order to make it "more diagonal". At the same time, of course, we must keep track of the product of all of the rotation matrices that we have so far used, because that is the matrix that ultimately will be an orthogonal matrix with the eigenvectors of A across its rows.

Let's watch this happen. Begin with A. After one rotation we have $J_1AJ_1^T$, after two iterations we have $J_2J_1AJ_1^TJ_2^T$, after three we have $J_3J_2J_1AJ_1^TJ_2^TJ_3^T$, etc. After all iterations have been done, and we are looking at a matrix that is "diagonal enough" for our purposes, the matrix we see is $PAP^T = D$, where P is obtained by starting with the identity matrix and multiplying successively on the left by the rotational matrices J that are used, and D is (virtually) diagonal.

Since $PAP^T = D$, we have $AP^T = P^TD$, so the columns of P^T, or equivalently the rows of P, are the eigenvectors of A.

Now, we have indeed proved that the repeated rotations will diagonalize A. We have not proved that the matrices P themselves converge to a certain fixed matrix. This is true, but we omit the proof. One thing we do want to do, however, is to prove the spectral theorem, Theorem 3.7.1, itself, since we have long since done all of the work.

Proof (of the Spectral Theorem 3.7.1): Consider the mapping f that associates with every orthogonal matrix P the matrix $f(P) = P^TAP$. The set of orthogonal matrices is compact, and the mapping f is continuous. Hence the image of the set of orthogonal matrices under f is compact. Hence there is a matrix F in that image that minimizes the continuous function $\mathrm{Od}(f(P)) = \mathrm{Od}(P^TAP)$. Suppose D is *not* diagonal. Then we could find a Jacobi rotation that would produce another matrix in the image whose Od would be lower, which is a contradiction (of the fact that $\mathrm{Od}(D)$ was minimal). Hence F is diagonal. So there is an orthogonal matrix P such that $P^TAP = D$, *i.e.*, $AP = PD$. Hence the columns of P are n pairwise orthogonal eigenvectors of A, and the proof of the spectral theorem is complete.

Now let's get on with the implementation of the algorithm.

3.10 Corbató's idea and the implementation of the Jacobi algorithm

It's time to sit down with our accountants and add up the costs of the Jacobi method. First, we have seen that $O(n^2)$ rotations will be sufficient to reduce the off-diagonal sum of squares below some pre-assigned threshold level. Now, what is the price of a single rotation? Here are the steps:

(i) Search for the off-diagonal element having the largest absolute value. The cost seems to be equal to the number of elements that have to be looked at, namely $n(n-1)/2$, which we abbreviate as $O(n^2)$.

(ii) Calculate θ, $\sin\theta$ and $\cos\theta$, and then carry out a rotation on the matrix A. This costs $O(n)$, since only four lines of A are changed.

(iii) Update the matrix P of eigenvectors by multiplying it by the rotation matrix. Since only two rows of P change, this cost is $O(n)$ also.

The longest part of the job is the search for the largest off-diagonal element. The search is n times as expensive in time as either the rotation of A or the update of the eigenvector matrix.

For this reason, in the years since Jacobi first described his algorithm, a number of other strategies for dealing with the eigenvalue problem have been worked out. One of these is called the *cyclic Jacobi method*. In that variation, one does not search, but instead goes marching through the matrix one element at a time. That is to say, first do a rotation that reduces A_{12} to zero. Next do a rotation that reduces A_{13} to zero (of course, A_{12} doesn't stay put, but becomes nonzero again!). Then do A_{14} and so forth, returning to A_{12} again after A_{nn}, cycling as long as necessary. This method avoids the search, but the proof that it converges at all is quite complex, and the exact rate of convergence is unknown.

A variation on the cyclic method is called the *threshold Jacobi method*, in which we go through the entries cyclically as above, but we do not carry out a rotation unless the magnitude of the current matrix entry exceeds a certain threshold ("throw it back, it's too small"). This method also has an uncertain rate of convergence.

At a deeper level, two newer methods due to Givens and Householder have been developed. These methods work not by trying to diagonalize A by rotations, but instead to tri-diagonalize A by rotations. A *tri-diagonal* matrix is one whose entries are all zero except for those on the diagonal, the sub-diagonal and the super-diagonal (*i.e*, $A_{ij} = 0$ unless $|i - j| \leq 1$).

The advantage of tri-diagonalization is that it is a finite process: it can be done in such a way that elements, once reduced to zero, stay zero instead of bouncing back again as they do in the Jacobi method. The disadvantage is that, having arrived at a tri-diagonal matrix, one is not finished, but instead one must then confront the question of obtaining the eigenvalues and eigenvectors of a tri-diagonal matrix, a nontrivial operation.

One of the reasons for the wide use of the Givens and Householder methods has been that they get the answers in just $O(n^3)$ time, instead of the $O(n^4)$ time in which the original Jacobi method operates.

Thanks to a suggestion of Corbató, (F. J. Corbató, On the coding of Jacobi's method for computing eigenvalues and eigenvectors of symmetric matrices, JACM, 10: 123-125, 1963) however, it is now easy to run the original Jacobi method in $O(n^3)$ time also. The suggestion is all the more remarkable because of its simplicity and the fact that it lives at the software level, rather than at the mathematical level. What it does is just this: it allows us to use the largest off-diagonal entry at each stage while paying a price of a mere $O(n)$ for the privilege, instead of the $O(n^2)$ billing mentioned above.

We can do this by carrying along an additional linear array during the calculation. The ith entry of this array, say `loc[i]`, contains the number of a column in which an off-diagonal

element of largest absolute value in row i lives, *i.e.*, $A_{i,loc(i)}$ is an entry in row i of A of largest absolute value in that row.

Now of course if some benefactor is kind enough to hand us this array, then it would be a simple matter to find the biggest off-diagonal element of the entire matrix A. We would just look through the $n-1$ numbers $|A_{i,loc(i)}|$ $(i = 1, 2, \ldots, n-1)$ to find the largest one. If the largest one is the one where $i = p$, say, then the desired matrix element would be $A_{p,loc(p)}$. Hence the cost of using this array is $\mathrm{O}(n)$.

Since there are no such benefactors as described above, we are going to have to pay a price for the care and feeding of this array. How much does it cost to create and to maintain it?

Initially, we just search through the whole matrix and set it up. This clearly costs $\mathrm{O}(n^2)$ operations, but we pay just once. Now let's turn to a typical intermediate stage in the calculation, and see what the price is for updating the `loc` array.

Given an array `loc`, suppose now that we carry out a single rotation on the matrix A. Precisely how do we go about modifying `loc` so it will correspond to the new matrix? The rotated matrix differs from the previous matrix in exactly two rows and two columns. Certainly the two rows, p and q, that have been completely changed will simply have to be searched again in order to find the new values of `loc`. This costs $\mathrm{O}(n)$ operations.

What about the other $n-2$ rows of the matrix? In the ith one of those rows exactly two entries were changed, namely the ones in the pth column and in the qth column. Suppose the largest element that was previously in row i was not in either the pth or the qth column, *i.e.*, suppose `loc[i]`$\notin \{p, q\}$. Then that previous largest element will still be there in the rotated matrix. In that case, in order to discover the new entry of largest absolute value in row i we need only compare at most three numbers: the new $|A_{ip}|$, the new $|A_{iq}|$, and the old $|A_{i,loc(i)}|$. The column in which the largest of these three numbers is found will be the new `loc[i]`. The price paid is at most three comparisons, and this does the updating job in every row except those that happen to have had `loc[i]`$\in \{p, q\}$.

In the latter case we can still salvage something. If we are replacing the entry of previously largest absolute value in the row i, we might after all get lucky and replace it with an even larger number, in which case we would again know the new `loc(i)`. Christmas, however, comes just once an year, and since the general trend of the off-diagonal entries is downwards, and the previous largest entry was uncommonly large, most of the time we'll be replacing the former largest entry with a smaller entry. In that case we'll just have to re-search the entire row to find the new champion.

The number of rows that must be searched in their entireties in order to update the `loc` array is therefore at most two (for rows p and q) plus the number of rows `i` in which `loc[i]` happens to be equal to p or to q. It is reasonable to expect that the probability of the event `loc[i]`$\in \{p, q\}$ is about two chances out of n, since it seems that it ought not to be any more likely that the winner was previously in those two columns than in any other columns. This has never been in any sense proved, but we will assume that it is so. Then the expected number of rows that will have to be completely searched will be about four, on the average (the pth, the qth, and an average of about two others).

It follows that the expected cost of maintaining the `loc` array is $O(n)$ per rotation. The cost of finding the largest off-diagonal element has therefore been reduced to $O(n)$ per rotation, after all bills have been paid. Hence the cost of finding that element is comparable with all of the other operations that go on in the algorithm, and it poses no special problem.

Using Corbató's suggestion, and subject to the equidistribution hypothesis mentioned above, the cost of the complete Jacobi algorithm for eigenvalues and eigenvectors is $O(n^3)$. We show below the complete Maple procedure for updating the array `loc` immediately after a rotation has been done in the plane of p and q.

```
update:=proc(p,q) local i,r;
 global loc,A,n;
 for i from 1 to n-1 do
    if i=p or i=q then searchrow(i)
                  else
      r:=loc[i];
      if r=p or r=q then
           if abs(A[i,p])>=abs(A[i,loc[i]]) then loc[i]:=p; fi;
           if abs(A[i,q])>=abs(A[i,loc[i]]) then loc[i]:=q; fi;
                    else
           if abs(A[i,loc[i]])<=abs(A[i,r]) then loc[i]:=r
                                            else searchrow(i)
 fi;fi;fi;od;
 RETURN();
 end:
```

The above procedure uses a small auxiliary routine:

Procedure `searchrow(i)`

This procedure searches the portion of row i of the $n \times n$ matrix A that lies above the main diagonal and places the index of a column that contains an entry of largest absolute value in `loc[i]`.

```
searchrow:=proc(i)
 local j,bigg;
 global loc,A,n,P;
 bigg:=0;
 for j from i+1 to n do
            if abs(A[i,j])>bigg then
                  bigg:=abs(A[i,j]);loc[i]:=j;fi;
                     od;
 RETURN();
 end:
```

We should mention that the search can be speeded up a little bit more by using a data structure called a *heap*. What we want is to store the locations of the biggest elements in each row in such a way that we can quickly access the biggest of all of them.

If we had stored the set of winners in a heap, or priority queue, then we would have been able to find the overall winner in a single step, and the expense would have been the maintenance of the heap structure, at a cost of a mere $O(\log n)$ operations. This would have reduced the program overhead by an addition twenty percent or thereabouts, although the programming job itself would have gotten harder.

To learn about heaps and how to use them, consult books on data structures, computer science, and discrete mathematics.

3.11 Getting it together

The various pieces of the procedure that will find the eigenvalues and eigenvectors of a real symmetric matrix by the method of Jacobi are now in view. It's time to discuss the assembly of those pieces.

Procedure `jacobi(eps,dgts)`

The input to the `jacobi` procedure is the $n \times n$ matrix A, and a parameter `eps` that we will use as a reduction factor to test whether or not the current matrix is "diagonal enough," so that the procedure can terminate its operation. Further input is `dgts`, which is the number of significant digits that you would like to be carried in the calculation. The input matrix A itself is global, which is to say that its value is set outside of the `jacobi`procedure, and it is available to all procedures that are involved.

The output of procedure `jacobi`will be an orthogonal matrix P that will hold the eigenvectors of A in its rows, and a linear array `eig` that will hold the eigenvalues of A.

The input matrix A will be destroyed by the action of the procedure.

The first step in the operation of the procedure will be to compute the original off-diagonal sum of squares `test`. The Jacobi process will halt when this sum of squares of all off-diagonal elements has been reduced to `eps*test` or less.

Next we set the matrix P to the $n \times n$ identity matrix, and initialize the array `loc` by calling the subr outine `search` for each row of A. This completes the initialization.

The remaining steps all get done while the sum of the squares of the off-diagonal entries of the current matrix A exceeds `eps` times `test`.

We first find the largest off-diagonal element by searching through the numbers $A_{i,loc(i)}$ for the largest in absolute value. If that is $A_{p,loc(p)}$, we will then know p, q, A_{pq}, A_{pp} amd A_{qq}, and it will be time to compute the sine and cosine of the rotation angle θ. This is a fairly ticklish operation, since the Jacobi method is sensitive to small inaccuracies in these quantities. Also, note that we are going to calculate $\sin\theta$ and $\cos\theta$ without actually calculating θ itself. After careful analysis it turns out that the best formulas for the purpose

are:

$$\begin{aligned} x &:= 2A_{pq} \\ y &:= A_{pp} - A_{qq} \\ t &:= \sqrt{x^2 + y^2} \\ \sin\theta &:= \operatorname{sign}(xy)\left(\frac{1 - |y|/t}{2}\right)^{1/2} \\ \cos\theta &:= \left(\frac{1 + |y|/t}{2}\right)^{1/2}. \end{aligned} \tag{3.11.1}$$

Having computed $\sin\theta$ and $\cos\theta$, we can now call `rotate` twice, as discussed in section 3.9, first with `opt=1` and again with `opt=2`. The matrix A has now been transformed into the next stage of its march towards diagonalization.

Next we multiply the matrix P on the left by the same $n \times n$ orthogonal matrix of Jacobi (3.8.3), in order to update the matrix that will hold the eigenvectors of A on output. Notice that this multiplication affects only rows p and q of the matrix P, so only $2n$ elements are changed.

Next we call `update`, as discussed in section 3.10, to modify the `loc` array to correspond to the newly rotated matrix, and we are at the end (or `od`) of the `while` that was started a few paragraphs ago. In other words, we're finished. The Maple program for the Jacobi algorithm follows.

```
jacobi:=proc(eps,dgts)
local test,eig,i,j,iter,big,p,q,x,y,t,s,c,x1;
global loc,n,P,A;
 with(linalg):                                  # initialize
 iter:=0;n:=rowdim(A);Digits:=dgts;
 loc:=array(1..n);
 test:=add(add(2*A[i,j]^2,j=i+1..n),i=1..n-1);  #sum squares o-d entries
 P:=matrix(n,n,(i,j)->if i=j then 1 else 0 fi); #initialize eigenvector matrix
 for i from 1 to n-1 do searchrow(i) od;        #set up initial loc array
 big:=test;
  while big>eps*test do                         #begin next sweep
   x:=0;                                        #find largest o.d. element
   for i from 1 to n-1 do
     if abs(A[i,loc[i]])>x then x:=abs(A[i,loc[i]]);p:=i; fi;od;
   q:=loc[p];
   x:=2*A[p,q]; y:=A[p,p]-A[q,q];               #find sine and cosine of theta
   t:=evalf(sqrt(x^2+y^2));
   s:=sign(x*y)*evalf(sqrt(0.5*(1-abs(y)/t)));
   c:=evalf(sqrt(0.5*(1+abs(y)/t)));
   rotate(s,c,p,q,1);rotate(s,c,p,q,2);         #apply rotations to A
   for j from 1 to n do                         #update matrix of eigenvectors
      t:=c*P[p,j]+s*P[q,j];
```

```
      P[q,j]:=-s*P[p,j]+c*P[q,j];
      P[p,j]:=t;
                    od;
   update(p,q);                                  #update loc array
   big:=big-x^2/2;iter:=iter+1;                  #go do next sweep
                  od;                            #end of while
eig:=[seq(A[i,i],i=1..n)];                       #output eigenvalue array
print(eig,P,iter);                               #print eigenvals, vecs, and
RETURN();                                        # no. of sweeps needed
end:
```

To use the programs one does the following. First, enter the four procedures `jacobi`, `update`, `rotate`, `searchrow` into a Maple worksheet. Next enter the matrix A whose eigenvalues and eigenvectors are wanted. Then choose `dgts`, the number of digits of accuracy to be maintained, and `eps`, the fraction by which the original off diagonal sum of squares must be reduced for convergence.

As an example, a call `jacobi(.00000001,15)` will carry 15 digits along in the computation, and will terminate when the sum of squares of the off diagonal elements is .00000001 times what it was on the input matrix.

3.12 Remarks

For a parting volley in the direction of eigenvalues, let's review some connections with the first section of this chapter, in which we studied linear mappings, albeit sketchily.

It's worth noting that the eigenvalues of a matrix really are the eigenvalues of the linear mapping that the matrix represents with respect to some basis.

In fact, suppose T is a linear mapping of E^n (Euclidean n-dimensional space) to itself. If we choose a basis for E^n then T is represented by an $n \times n$ matrix A with respect to that basis. Now if we change to a different basis, then the same linear mapping is represented by $B = HAH^{-1}$, where H is a nonsingular $n \times n$ matrix. The proof of this fact is by a straightforward calculation, and can be found in standard references on linear algebra.

First question: what happens to the determinant if we change the basis? Answer: nothing, because

$$\begin{aligned} \det(HAH^{-1}) &= \det(H)\det(A)\det(H^{-1}) \\ &= \det(A). \end{aligned} \tag{3.12.1}$$

Hence the value of the determinant is a property of the linear mapping T, and will be the same for every matrix that represents T in some basis. Hence we can speak of $\det(T)$, the determinant of the linear mapping itself.

Next question: what happens to the eigenvalues if we change basis? Suppose $\mathbf{x}$ is an eigenvector of A for the eigenvalue λ. Then $A\mathbf{x} = \lambda\mathbf{x}$. If we change basis, A changes to $B = HAH^{-1}$, or $A = H^{-1}BH$. Hence $H^{-1}BH\mathbf{x} = \lambda\mathbf{x}$, or $B(H\mathbf{x}) = \lambda(H\mathbf{x})$. Therefore $H\mathbf{x}$

is an eigenvector of B with the same eigenvalue λ. The eigenvalues are therefore independent of the basis, and are properties of the linear mapping T itself. Hence we can speak of the eigenvalues of a linear mapping T.

In the Jacobi method we carry out transformations $A \to JAJ^T$, where $J^T = J^{-1}$. Hence this transformation corresponds exactly to looking at the underlying linear mapping in a different basis, in which the matrix that represents it is a little more diagonal than before. Since J is an orthogonal matrix, it preserves lengths and angles because it preserves inner products between vectors:

$$\langle J\mathbf{x}, J\mathbf{y}\rangle = \langle \mathbf{x}, J^T J\mathbf{y}\rangle = \langle \mathbf{x}, \mathbf{y}\rangle. \tag{3.12.2}$$

Therefore the method of Jacobi works by rotating a basis slightly, into a new basis in which the matrix is closer to being diagonal.