### 1.9 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

\[
y_{n+1} = y_n + hf(x_n, y_n). \tag{1.9.1}
\]

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

\[
y_i'(x) = f_i(x, y_1(x), y_2(x), \ldots, y_N(x)) \quad i = 1, \ldots, N. \tag{1.9.2}
\]

Equation (1.9.2) 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 \( \mathbf{y}(x) \) of unknown functions

\[
\mathbf{y}(x) = [y_1(x), y_2(x), y_3(x), \ldots, y_N(x)] \tag{1.9.3}
\]

and the vector \( \mathbf{f} = \mathbf{f}(x, \mathbf{y}(x)) \) of right-hand sides

\[
\mathbf{f} = [f_1(x, \mathbf{y}), f_2(x, \mathbf{y}), \ldots, f_N(x, \mathbf{y})]. \tag{1.9.4}
\]

In terms of these vectors, equation (1.9.2) can be rewritten as

\[
\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)). \tag{1.9.5}
\]

We observe that equation (1.9.5) looks just like our standard form (1.7.1) for a single equation in a single unknown function, except for the bold face type, i.e., except for the fact that \( \mathbf{y} \) and \( \mathbf{f} \) 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

\[
y_{n+1} = y_n + hf(x, y_n) \tag{1.9.6}
\]

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

\[
\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(x_n, \mathbf{y}_n). \tag{1.9.7}
\]

This means that if we know the entire vector \( \mathbf{y} \) 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 (1.9.7).

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

\[
\mathbf{y}_i(x_{n+1}) = \mathbf{y}_i(x_n) + h f_i(x_n, \mathbf{y}_1(x_n), \mathbf{y}_2(x_n), \ldots, \mathbf{y}_N(x_n)) \tag{1.9.8}
\]
for each \( i = 1, 2, \ldots, N \).

As an example, take the pair of differential equations

\[
\begin{align*}
    y'_1 &= x + y_1 + y_2 \\
    y'_2 &= y_1 y_2 + 1
\end{align*}
\]

(1.9.9)

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

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

\[
\begin{bmatrix}
    0.05 \\
    1.05
\end{bmatrix} =
\begin{bmatrix}
    0 & 0.05 \\
    1 & 1.05
\end{bmatrix}
\]

(1.9.10)

and

\[
\begin{bmatrix}
    0.1075 \\
    1.102625
\end{bmatrix} =
\begin{bmatrix}
    0.05 & 0.05 + 0.05 + 1.05 \\
    1.05 & 1.05 + 1.05 + 1
\end{bmatrix}
\]

(1.9.11)

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 (1.9.2), which we will rewrite just slightly, in the form

\[
y'_i = f_i(x, \vec{y}) \quad i = 1, \ldots, N.
\]

(1.9.12)

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 \( \vec{Y} \) of length \( N \) to hold the values of the \( N \) unknown functions at the current point \( x \). Suppose we have computed the array \( \vec{Y} \) at a point \( x \), and we want to get the new array \( \vec{Y} \) 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 (1.9.8) or (1.9.12) in a certain order. For instance, we might calculate \( \vec{y}_1(x + h) \), then \( \vec{y}_2(x + h) \), then \( \vec{y}_3(x + h) \), \ldots, then \( \vec{y}_N(x + h) \).

The question is this: when we compute \( \vec{y}_i(x + h) \), where shall we put it? If we put it into \( \vec{Y}[1] \), the first position of the \( \vec{Y} \) array in storage, then the previous contents of \( \vec{Y}[1] \) are lost, \( i.e. \), the value of \( \vec{y}_1(x) \) is lost. But we aren’t finished with \( \vec{y}_1(x) \) yet; it’s still needed to compute \( \vec{y}_2(x + h) \), \( \vec{y}_3(x + h) \), etc. This is because the new value \( \vec{y}_i(x + h) \) depends (or might depend), according to (1.9.8), 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 \( \vec{y}_i(x + h) \).

If the point still is murky, go back to (1.9.11) and notice how, in the calculation of \( \vec{y}_2(0.10) \) we needed to know \( \vec{y}_1(0.05) \) even though \( \vec{y}_1(0.10) \) had already been computed. Hence if we had put \( \vec{y}_1(0.10) \) into an array to replace the old value \( \vec{y}_1(0.05) \) we would not have been able to obtain \( \vec{y}_2(0.10) \).

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

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

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 \texttt{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 \texttt{seq} command. The brackets \texttt{[]} and \texttt{]} convert the list into a vector. The \texttt{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 (1.9.9). In that case we have $f_1(x, y) = x + y_1 + y_2$ and $f_2(x, y) = y_1y_2 + 1$. This translates into the following:

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

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 \texttt{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

\[ y'' + xy' + (x + 1) \cos y = 2. \]  (1.9.13)

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 (1.9.13), and the function $v$ is to be the derivative of $u$. Then $u$ and $v$ satisfy two simultaneous first-order differential equations:

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

and these are exactly of the form (1.9.5) that we have already discussed!

The same trick works on a general differential equation of $N$th order

$$y^{(N)} + G(x, y, y', y'', \ldots, y^{(N-1)}) = 0.\tag{1.9.15}$$

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

$$
\begin{align*}
u_0' &= u_1 \\
u_1' &= u_2 \\
&\quad \vdots \\
u_{N-2}' &= u_{N-1} \\
u_{N-1}' &= -G(x, u_0, u_1, \ldots, u_{N-2}).
\end{align*}
$$

The system can now be dealt with as before.

**Exercises 1.9**

1. Write each of the following as a system of simultaneous first-order initial-value problems in the standard form (1.9.2):

   (a) $y'' + x^2 y = 0; \quad y(0) = 1; \quad y'(0) = 0$
   (b) $u' + xv = 2; \quad v' + e^{uv} = 0; \quad u(0) = 0; \quad v(0) = 0$
   (c) $u' + xv' = 0; \quad v' + x^2 u = 1; \quad u(1) = 1; \quad v(1) = 0$
   (d) $y''' + 3xy'' + x^2 y'' + 2y' + y = 0; \quad y(0) = y'(0) = y''(0) = y'''(0) = 1$
   (e) $x''(t) + x^3 x(t) + y(t) = 0; \quad y''(t) + x'(t)^2 = t^3; \quad x(0) = 2; \quad x'(0) = 0; \quad x''(0) = 0; \quad y(0) = 1$

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 and 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.
1.10 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 usable by other people. The act of communication that must take place before a program can be used by persons other than its author is a difficult one to carry out, and we will return several times to the principles that 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 1.8. 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 1.8, 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 1.8. 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 in to the vector \( \mathbf{B} \), put the coefficients into the matrix \( \mathbf{A} \) and call the routine. The answer will be returned in \( \mathbf{X} \).”

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

“This program solves the equations \( \mathbf{AX}=\mathbf{B} \), where \( \mathbf{A} \) is an \( N \times N \) matrix and \( \mathbf{B} \) 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 H.S. Wilf, 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 how 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 \( TRUE \) unless 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
numerical analysis

\begin{verbatim}
solver:=proc(A,X,n,ndim,b);

in which the communicating variables have the following types:

\begin{align*}
A & \text{ ndim-by-ndim array of floating point numbers} \\
X & \text{ vector of floating point numbers of length } n \\
n & \text{ integer} \\
\text{ndim} & \text{ integer} \\
b & \text{ vector of floating point numbers of length } n
\end{align*}

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, and 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 \texttt{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 \texttt{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 I'C'O', or a variable \texttt{why} might be of role I'C'O', 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 1.8, that searches for the largest element in a row of a matrix. Here is the table of information about its global variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>floating point matrix</td>
<td>IC'O'</td>
<td>The input matrix</td>
</tr>
<tr>
<td>i</td>
<td>integer</td>
<td>IC'O'</td>
<td>Which row to search</td>
</tr>
<tr>
<td>jwin</td>
<td>integer</td>
<td>I'C0</td>
<td>Column containing largest element</td>
</tr>
</tbody>
</table>

**Exercises 1.10**

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!).

**1.11 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 \( x_n \):

\[
y(x_n + h) = y(x_n) + hy'(x_n) + \frac{h^2}{2} y''(x_n) + \frac{h^3}{6} y'''(x_n) + \cdots
\]  

(1.11.1)
Now we rewrite equation (1.11.1) with \( h \) replaced by \(-h\) to get
\[
y(x_n - h) = y(x_n) - hy'(x_n) + h^2 \frac{y''(x_n)}{2} - h^3 \frac{y'''(x_n)}{6} \ldots
\]
and then subtract these equations, obtaining
\[
y(x_n + h) - y(x_n - h) = 2hy'(x_n) + 2h^3 \frac{y'''(x_n)}{6} + \ldots. \tag{1.11.3}
\]

Now, just as we did in the derivation of Euler’s method, we will truncate the right side of (1.11.3) 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
\[
y_{n+1} - y_{n-1} = 2hy'_n. \tag{1.11.4}
\]
If, as usual, we are solving the differential equation \( y' = f(x,y) \), then finally (1.11.4) takes the form
\[
y_{n+1} = y_{n-1} + 2hf(x_n, y_n) \tag{1.11.5}
\]
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 (1.11.5) 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, and 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 (1.7.6) 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 1.7.1). It’s easy to continue the calculation now from (1.11.5).

For instance
\[
y_2 = y_0 + 2h(0.5y_1) \\
= 1 + 0.1(0.5 \times 1.025) \\
= 1.05125
\]
and
\[
y_3 = y_1 + 2h(0.5y_2) \\
= 1.025 + 0.1(0.5 \times 1.05125) \\
= 1.0775625.
\]
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.

<table>
<thead>
<tr>
<th>$x$</th>
<th>Midpoint($x$)</th>
<th>Euler($x$)</th>
<th>Exact($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>0.05</td>
<td>1.025000</td>
<td>1.025000</td>
<td>1.025320</td>
</tr>
<tr>
<td>0.10</td>
<td>1.051250</td>
<td>1.050630</td>
<td>1.051270</td>
</tr>
<tr>
<td>0.15</td>
<td>1.077560</td>
<td>1.076890</td>
<td>1.077880</td>
</tr>
<tr>
<td>0.20</td>
<td>1.105130</td>
<td>1.103810</td>
<td>1.105170</td>
</tr>
<tr>
<td>0.25</td>
<td>1.132820</td>
<td>1.131410</td>
<td>1.133150</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1.00</td>
<td>1.648470</td>
<td>1.638620</td>
<td>1.648720</td>
</tr>
<tr>
<td>2.00</td>
<td>2.717630</td>
<td>2.685060</td>
<td>2.718280</td>
</tr>
<tr>
<td>3.00</td>
<td>4.480320</td>
<td>4.399790</td>
<td>4.481690</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5.00</td>
<td>12.177430</td>
<td>11.813720</td>
<td>12.182490</td>
</tr>
<tr>
<td>10.00</td>
<td>148.31274</td>
<td>139.56389</td>
<td>148.41316</td>
</tr>
</tbody>
</table>

**Table 1.11.1**

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

$$y(x + h) = y(x) + \int_{x}^{x+h} f(t, y(t)) \, dt. \quad (1.11.6)$$

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

$$\int_{a}^{b} f(t) \, dt$$

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 1.11.1. That area is $\frac{1}{2}(f(a) + f(b))(b - a)$.

**Figure 1.11.1**
If we apply the trapezoidal rule to the integral that appears in (1.11.6), we obtain

\[ y(x_n + h) \approx y(x_n) + \frac{h}{2} \left( f(x_n, y(x_n)) + f(x_n + h, y(x_n + h)) \right) \]  

(1.11.7)

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 (1.11.7) becomes

\[ y_{n+1} = y_n + \frac{h}{2} \left( f(x_n, y_n) + f(x_{n+1}, y_{n+1}) \right). \]  

(1.11.8)

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

At first sight, (1.11.8) looks like a recurrence formula from which the next approximate value, \( y_{n+1} \), of the unknown function, can 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 (1.11.8) 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 1.13. 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

\[ y' = 2xe^y + 1 \]

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

\[ y_1 = 1 + 0.025(2 + 0.1e^{y_1}) \]  

(1.11.9)

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 (1.11.9), 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 1.11.2 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(\( x \)).
1.12 Comparison of the methods

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

\[ y_{n+1} = y_n + hy'_n, \]  
(1.12.1)

the trapezoidal rule

\[ y_{n+1} = y_n + \frac{h}{2}(y'_n + y'_{n+1}), \]  
(1.12.2)

and the midpoint rule

\[ y_{n+1} = y_{n-1} + 2hy'_n. \]  
(1.12.3)

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

\[ y' = -\frac{y}{L}; \quad y(0) = 1; \quad L > 0 \]  
(1.12.4)

where \( L \) is a constant. The most interesting and revealing case is where the true solution is a decaying exponential, and 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

\[ \text{Exact}(x) = e^{-x/L}. \]  
(1.12.5)

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 (1.12.1)–(1.12.3) handles the problem (1.12.4).
Suppose first that we ask Euler’s method to solve the problem. If we substitute $y' = f(x, y) = -y/L$ into (1.12.1), we get

$$y_{n+1} = y_n + h \cdot \left(-\frac{y_n}{L}\right)$$

$$= \left(1 - \frac{h}{L}\right)y_n.$$  \hspace{1cm} (1.12.6)

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 $\tau$.

Now the solution of the recurrence equation (1.12.6), with the starting value $y_0 = 1$, is obviously

$$y_n = (1 - \tau)^n \quad n = 0, 1, 2, \ldots.$$  \hspace{1cm} (1.12.7)

Next we study the trapezoidal approximation to the same equation (1.12.4). We substitute $y' = f(x, y) = -y/L$ in (1.12.2) and get

$$y_{n+1} = y_n + \frac{h}{2} \left(-\frac{y_n}{L} - \frac{y_{n+1}}{L}\right).$$  \hspace{1cm} (1.12.8)

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 (1.12.8) for $y_{n+1}$ (without any need for an iterative process) and obtaining

$$y_{n+1} = \frac{1 - \frac{\tau}{2}}{1 + \frac{\tau}{2}}y_n.$$  \hspace{1cm} (1.12.9)

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

$$y_n = \left(\frac{1 - \frac{\tau}{2}}{1 + \frac{\tau}{2}}\right)^n \quad n = 0, 1, 2, \ldots.$$  \hspace{1cm} (1.12.10)

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

(a) $e^{-\tau}$

(b) $1 - \tau$

(c) $\frac{1 - \frac{\tau}{2}}{1 + \frac{\tau}{2}}$.  \hspace{1cm} (1.12.11)

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 $\tau$ is being thought of as small compared to 1, then we have the power series expansion of $e^{-\tau}$

$$e^{-\tau} = 1 - \tau + \frac{\tau^2}{2} - \frac{\tau^3}{6} + \cdots.$$  \hspace{1cm} (1.12.12)
to compare with \(1 - \tau\) and with the power series expansion of
\[
\frac{1 - \frac{\tau}{2}}{1 + \frac{\tau}{2}} = 1 - \tau + \frac{\tau^2}{2} - \frac{\tau^3}{4} + \cdots. \tag{1.12.13}
\]

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 (1.12.3) to get
\[
y_{n+1} = y_{n-1} + 2h \ast \left(\frac{-y_n}{L}\right). \tag{1.12.14}
\]

One important feature is already apparent. Instead of facing a first-order difference equation as we did in (1.12.6) for Euler’s method and in (1.12.9) 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
\[
r^2 + 2\tau r - 1 = 0. \tag{1.12.15}
\]
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 (1.12.14) is
\[
y_n = c (r_+(\tau))^n + d (r_-(\tau))^n, \tag{1.12.16}
\]
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 \((\text{constant})^n\). 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 is not so obliging, and that in fact it does whatever it can to spoil things.

The two roots of the quadratic equation are
\[
r_+(\tau) = -\tau + \sqrt{1 + \tau^2} \tag{1.12.17}
\]
\[
r_-(\tau) = -\tau - \sqrt{1 + \tau^2}.
\]
When \(\tau = 0\) the first of these is +1, so when \(\tau\) 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
\[
r_+(\tau) = 1 - \tau + \frac{\tau^2}{2} - \frac{\tau^4}{8} + \cdots \tag{1.12.18}
\]
and so it agrees with \(e^{-\tau}\) through the quadratic terms.

What about \(r_-(\tau)\)? Its Taylor series is
\[
r_-(\tau) = -1 - \tau - \frac{\tau^2}{2} + \cdots. \tag{1.12.19}
\]
The bad news is now before us: When \( \tau \) 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, and so we say that the midpoint rule is unstable.

In practical terms, we observe that \( r_+ (\tau) \) is close to \( e^{-\tau} \), and so the first term on the right of (1.12.16) is very close to the exact solution. The second term of (1.12.16), 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 1.12.1 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.

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

Table 1.12.1

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 non-iterative. A method is non-iterative 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.
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.