Numerics for First-Order ODEs


Analytical solutions to differential equations are sometimes hard to find. The effort can be worth it if the solution is easy. If the solution is complicated it can still be worth it, if the solution will be used over and over again, or if the symbolic form of the solution is self-revealing (and perhaps could be published). On the other hand, if the solution is to be used once, or only a few times, if only numerical values of the solution are sought, or if the analytical solution is too complex or even impossible, a numerical approach is appropriate.

Consider the IV problem

\[ y' = f(x, y) \quad y(x_0) = y_0 \quad (1) \]

where \( x_0 \) and \( y_0 \) are given, and the solution is required and exists on some finite-solution, closed-interval \([a, b]\). Depending on the numerical method used to solve (1), the solution is obtained at a finite number of points \( x_n, n = 0, 1, 2, 3, \ldots N \) in the closed interval \([a, b]\). For an initial value problem \( x_0 = a \) represents the location of the IC, and \( x_N = b \) represents a stopping point. If the solution is asymptotic to some limit, \( b \) is simply selected to be large enough such that \( y \) closely approaches that limit. The solution consists of values \( y_n \), for \( n = 0, 1, 2, 3, \ldots \) obtained at each of the points \( x_n \), where the first value \( y_0 \) is the prescribed IC. The solution then consists of the \( N+1 \) pairs \((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots (x_N, y_N)\).

Looking ahead for motivation, we’ll see that a system of ODEs consists of solving the simultaneous system of equations

\[ y' = f(x, y) \quad y(x_0) = y_0 \quad (2) \]

where \( y \) is a column vector of unknowns, say \( y = [y_a, y_b, y_c, y_d, \ldots]' \), where the subscripts \( a, b, c, \ldots \) represent the different unknowns at each scalar point \( x \) and the prime indicates a vector transform
(to turn the row vector into a column vector). \textit{We use the same methods to solve vector ode (2) that we use to solve scalar ode (1)}. The equations in vector equation (2) may represent a number of different strongly coupled processes, where the value of one unknown depends on the value of another. An example is strongly coupled fluid heat and mass transfer, where the heat transfer coefficient depends on the flow rate, and the flow rate depends on the heat transfer (say, through fluid viscosity or density). The other application of (2) is in the numerics that are used to solve PDEs. In this case the independent variable \( x \) represents time (i.e., \( x = t \)) and the unknowns in vector \( y \) represent spatially distributed values of a state variable (e.g., solute concentration) taken at a finite number of discretized points (node points) in space. An example is a groundwater flow model, such as that programmed into the applied code MODFLOW, where in (2) \( y \) is a vector of nodal values representing spatially distributed hydraulic heads. The nodal solutions are calculated at times \( t_1, t_2, t_3, t_4, \ldots \) by solving (2) (with time, \( t \), replacing \( x \) and hydraulic head, \( h \), replacing \( y \)).

\textbf{Approaches to initial value problems (JLW)}

There are a variety of issues to consider when setting up and interpreting a numerical solution to (1). How big do you make the step size, \( \Delta x \), between the points \( x_{n+1} \) and \( x_n \)? Given a value of the unknown, \( y_n \), at point \( x_n \), how do you estimate the value \( y_{n+1} \) at the next point, \( x_{n+1} \)? What is the error in this estimation? Is the estimation (numerically) stable?

\textbf{Taylor Series and numerical approximation.} Most of the methods we’ll consider are based on the Taylor Series or something similar. We use the notation of Figure 1. The Taylor Series for a function \( y \) at location \( x_{n+1} \), given the value and derivatives of \( y \) at location \( x_n \), is

\[
y'_{n+1} = y_n + \Delta x \, y'(x_n) + \frac{\Delta x^2}{2} \, y''(x_n) + \frac{\Delta x^3}{3!} \, y'''(x_n) + \ldots
\]

where \( \Delta x = x_{n+1} - x_n \). This is exact. If, however, we neglect the higher order terms we get the second-order approximation

\[
y'_{n+1} \approx y_n + \Delta x \, y'(x_n) + \frac{\Delta x^2}{2} \, y''(\xi)
\]

where \( x_n \leq \xi \leq x_n + \Delta x = x_{n+1} \). The last term on the RHS represents the second-order local truncation error (see below) for this approximation, which we encounter by dropping the higher order terms.

We are interested in solving the ODE \( y' = f(x, y) \), thus we seek a numerical approximation to the derivative on the LHS of the ODE. The Taylor Series provides such an approximation, if we solve (4) for the first derivative.

\[
y'(x_n) = \text{slope} \approx \frac{y_{n+1} - y_n}{\Delta x} + \frac{\Delta x}{2} \, y''(\xi)
\]

We now insert this approximation into the ODE to get the finite difference model

\[
\frac{y_{n+1} - y_n}{\Delta x} + \frac{\Delta x}{2} \, y''(\xi) \approx f(x_n, y_n)
\]  

where we take the RHS of \( f \) at the same point \( x_n \) around which we write the Taylor Series. Solving for \( y_{n+1} \) gives
\[ y_{n+1} \approx y_n + \Delta x \, f(x_n, y_n) + \frac{\Delta x^2}{2} \, y''(\xi) \]  

which we usually write as

\[ y_{n+1} \approx y_n + \Delta x \, f(x_n, y_n) + O(\Delta x^2) \]  

to indicate the order of the approximation. This particular approximation is called a \textit{forward or explicit Euler approximation}\textsuperscript{5} (see pp. 886-888 of text). Note that the explicit method handles non-linearity, by explicitly evaluating \( f \) at the previous location, \( x_n \), and previous value of the unknown, \( y_n \).

\[ F(x_n, x_{n+1}, y_n, y_{n+1}) + O(\Delta x^k) \]

where \( k \) is the order of the approximation. The right side function \( F \) can include a contribution from the current unknown, \( y_{n+1} \). It also implicitly includes some combination of \( f_n \) and \( f_{n+1} \). For example, in the explicit Euler method (8) there is a potential contribution from \( x_n, x_{n+1}, y_n, \) and \( f_n \) to the right side, but there is no possible contribution from \( y_{n+1}, \) or \( f_{n+1} \). For a given ODE different numerical methods yield different functions \( F \) and different approximation orders \( k \).

**Errors.** The numerical methods used to make the estimation are subject to both roundoff errors and truncation errors. Roundoff error is due to the limited word length of a computer, while truncation error is due to the numerical approximation nature of the method used to calculate the solution (as in (9)). That is, roundoff error is due to the computer, while truncation error is within your numerical algorithm. You can select better numerical methods (see below) that reduce truncation error (increase the value of \( k \) in (9)), but only at the expense of additional computational effort. Or you can reduce the step size \( \Delta x \) to reduce truncation error, and keep a
simpler method. However, there is a limit to the effectiveness of reducing step size. However, when the step gets too small roundoff error dominates and the solution is worthless.

**Global v. local truncation error.** Truncation error consists of two parts. Local truncation error is due to the application of the numerical method to a single step. See (7) above to observe that the explicit Euler method has a local error of order \( \Delta x^2 \), when measured as to how well the unknown \( y \) is approximated. Propagated or accumulated truncation error is due to the accumulation of error from previous steps. Together they produce global (total) truncation error in the solution. The explicit Euler method has a global error of order \( \Delta x \) (the number of steps is proportional to \( 1/\Delta x \), which you multiply by the local error \( \Delta x^2 \) in each step). The order of approximation for global error is (almost) always lower order than the local error.

**Balance error.** The error measures above look at how well the function \( y \) is approximated. But we are hydrologists and we are particularly interested in how well the balance of mass, momentum or energy is represented by a numerical approximation. Consider again the ODE, \( y' = f(x, y) \). Rewrite is as a balance equation, \( y' - f(x, y) = 0 \). If we replace the true solution by a numerical approximation, then it can’t possibly balance, at least not perfectly. We can only hope to get close. Rewrite the balance as

\[
y' - f(x, y) = \varepsilon
\]

If we have the exact solution for \( y \) then the out-of-balance error, \( \varepsilon \), will be zero. But when we replace \( y \) by its numerical approximation, \( \varepsilon \) will be non-zero. How much different from zero will it be? For the explicit Euler method you can infer from (6) above that its local balance error is only of order \( \Delta x \), and that the global balance error is even worse.

Aside: when solving non-linear versions of \( y' - f(x, y) = 0 \), especially when involved with simultaneous solutions as in (2), we often resort to Newton’s method (p. 790 of text), searching for the value of \( y \) that takes \( \varepsilon \) in (10) close to zero. That is, we search for the root of \( y' - f(x, y) = 0 \).

**Numerical stability.** Many of the numerical methods used to solve (1) are unconditionally stable. Some methods we like to teach, which are used by beginners, and which lend themselves to parallel computation are conditionally stable (an example is Euler’s forward or explicit scheme, presented above). Make the \( \Delta x \) too large and they blow up. (Of course, some ODEs are themselves unstable; we are referring here only to instabilities introduced by the numerical method.) There are also intrinsically unstable numerical methods (for a given ODE) that we will avoid.

**Step size.** If uniform step sizes \( \Delta x \) are used, mathematicians usually refer to the step with symbol \( h \), or \( h = \Delta x = \text{constant} \). In many applications we allow the step size to vary, either prescriptively or adaptively. In prescriptive declarations of step sizes the user decides where finer prescriptively or adaptively. In prescriptive declarations of step sizes the user decides where finer resolution is needed. In adaptive methods an algorithm is supplied to make this decision as part of the solution. In any event, the step must be small enough to minimize truncation error to an acceptable value, but not so small as to encounter unacceptable roundoff (see above).

**Prescription of variable step size.** In many cases you can anticipate the general functional nature of the solution, that is, you know something about when/where \( y \) is going to vary rapidly with \( x \) ("where the action is") and when/where it is not. In this case you can prescribe a variable
\( \Delta x_n = x_{n+1} - x_n \), increasing the density of the points \( x_n \) where the action is, in order to reduce truncation error there, and reducing it elsewhere. Two examples are groundwater flow and heat transport, both of which respond exponentially to change. When a new forcing is added, you can anticipate that you’ll need smaller \( \Delta x \)’s after the change, but can enlarge them latter. For example, many groundwater codes prescriptively reduce the time step after a well is turned on, and then let the time step increase logarithmically afterwards.

**Adaptive step size.** In other cases you don’t know where the smaller time steps are needed, after all, they are a function of the as-yet uncalculated solution. Suppose that you want the numerical method to figure this out and automatically reduce \( \Delta x \), where/when the action is, and then increase it away from the action. A variety of adaptation schemes are available. See p. 889 of text for a general overview, which is illustrated with the explicit Euler method. These adaptation schemes are usually based on finding the rate of change of the ODE, \( y' = f(x, y) \), with respect to (wrt) the independent variable, \( x \). Using the chain rule:

\[
y'' = f'' = f_x + f_y \frac{dy}{dx}
\]

They then select the next step size \( \Delta x_0 = x_{n+1} - x_n \) such that the truncation error does not exceed a certain tolerance.

**Explicit v. implicit methods.**

The difference between explicit and implicit methods is in how the solution is calculated at each time step. *Explicit methods* use an explicit or forward formula for calculating \( y_{n+1} \) at point \( x_{n+1} \), given \( y_n \) and \( f_n \) at point \( x_n \). In an explicit formula the right side of the numerical equation has only known quantities. That is, the ODE \( y' = f(x, y) \) is replaced through the general numerical approximation in (9) by the particular approximation

\[
y_{n+1} \approx F(x_n, x_{n+1}, y_n) + O(\Delta x^k)
\]

The formula is explicit in that the variables in the right side function, \( (x_n, x_{n+1}, y_n) \) are all known.

The unknown, \( y_{n+1} \), does not appear on the RHS. For this reason explicit methods handle non-linearities by simply evaluating their contributions to \( F \) using information already available at \( x_n \). The explicit Euler approximation in (8) is the simplist example of this.

With *implicit methods* the unknown \( y_{n+1} \) appears on both sides of the equation, just as in the general model (9).

\[
y_{n+1} \approx F(x_n, x_{n+1}, y_n, y_{n+1}) + O(\Delta x^k)
\]

For example, suppose that in the Euler method we took the Taylor Series around \( x_{n+1} \), instead of around \( x_n \). Then (4) would become

\[
y_n \approx y_{n+1} - \Delta x \ y'(x_{n+1}) + \frac{\Delta x^2}{2} y''(\xi)
\]
where, again, \( x_n \leq \xi \leq x_{n+1} \) and we still have \( \Delta x = x_{n+1} - x_n \). Note the change in sign on the first derivative (odd) term, since we are now going “backwards”. The finite difference model becomes

\[
\frac{y_{n+1} - y_n}{\Delta x} + \frac{\Delta x}{2} y''(\xi) \approx f(x_{n+1}, y_{n+1}) \quad (14)
\]

Here we take the RHS value of \( f \) at the same point \( x_{n+1} \) around which we write the Taylor Series. And that point is now the same as the point \( x_{n+1} \) at which we are solving for \( y_{n+1} \), or

\[
y_{n+1} \approx y_n + \Delta x f(x_{n+1}, y_{n+1}) + O(\Delta x^2) \quad (15)
\]

This is an implicit model because \( y_{n+1} \) is now on both sides of the equation. We call (15) a \textit{backward or fully implicit Euler method} (see p. 896 in text). Notice its truncation error is similar to the explicit scheme. Also note that because \( y_{n+1} \) is also on the right sides, non-linearities must be handled implicitly.

Aside: most IV problems in hydrology have time as the independent variable. Both implicit and explicit methods are used to solve these problems. Most spatially distributed problems are \textit{boundary value problems} (BVP) involving second order ODEs in space. BVPs problems must be solved implicitly.

**Single-Step Methods.**

The Euler schemes are examples of a single-step scheme. Most hydrologic models use single-step schemes, so it is a good idea to master them. In a \textit{single-step scheme} the numerical solution \( y_{n+1} \) at a point \( x_{n+1} \) is calculated from the known solution \( y_n \) at the previous point \( x_n \). The general algorithm to solve (1) is

\[
\begin{align*}
x_{n+1} & \approx x_n + \Delta x \\
y_{n+1} & \approx y_n + \Delta x \cdot \text{slope}
\end{align*} \quad (16a,b)
\]

where \( \Delta x = x_{n+1} - x_n \) is the step size and \textit{slope} is an estimate of the value of \( dy/dx \) in the interval \( x_n \) to \( x_{n+1} \). The solution starts at the point \( x_0 \) where the solution is known, and marches forward recursively, step by step, \( n=0, 1, 2, \ldots N \) (see Figures 1 and 4).

![Figure 4 Single-step explicit methods](image-url)
The difference between single-step methods is in the value used for slope. Methods become more sophisticated by improving the calculation of slope, for example, by subdividing the interval between \( x_n \) to \( x_{n+1} \) into smaller pieces. The order \( k \) of the approximation gets higher (better) with increased sophistication, but the computation becomes more expensive (because there are more arithmetic operations). In explicit schemes the slope depends only on known values \( y_n \) and \( f_n \), while in implicit schemes it also depends on unknown values \( y_{n+1} \) and \( f_{n+1} \).

The explicit or forward Euler scheme (8) is the simplest single-step method, but there are other Euler schemes that are commonly used. These include the backward scheme (15) and a mid-point Euler scheme (not in your text) with theoretically less truncation error. In any event Euler schemes are commonly encountered in most heat transfer, solute transport, groundwater, and vadose zone codes. The next most sophisticated single-step class of methods are called Runge-Kutta methods (RK methods). In these methods the slope is calculated from a weighted average of estimates of the slope at several points along the interval \( x_n \) to \( x_{n+1} \). Higher order RK methods use more points. MATLAB has M-files that use RK methods to solve ODEs, and has several orders of RK methods available to choose from. RK methods are typically used in pairs. By comparing the solution at one order to that at another, lower order, you can estimate error, and if needed decrease \( \Delta x \). RK methods are commonly used in hydrology for particle tracking, in which the pathlines and travel/residence times in various hydrologic systems are calculated.

**Multistep Methods.** In multistep methods the solution \( y_{n+1} \) at a point \( x_{n+1} \) is calculated from the known solutions at several previous points, e.g., at three points \( y_n \) at \( x_n \) AND \( y_{n-1} \) at \( x_{n-1} \) AND \( y_{n-2} \) at \( x_{n-2} \). The idea is that by using more points you can get a better idea of the trend of the solution. These methods have not been widely applied in hydrology in the past, but that is changing. Also, they are becoming common in sophisticated, commercially available multiphysics and domain specific codes, like COMSOL Multiphysics and the CFD (computational fluid dynamics) code Fluent, which are finding more frequent use in hydrology. Multistep methods are introduced in Section 21.2 of the text.

**Predictor-Corrector Methods.** Predictor-corrector methods use two formulas to solve an ODE at each step. The predictor uses an explicit formula that is used to get an estimate of the solution \( y^*_{n+1} \). Since this is an explicit calculation it only uses the known solution \( y_n \) at the previous step (single step methods) or several previous steps (multistep methods). Once the estimator \( y^*_{n+1} \) is found, a correction is applied.

**Stiff ODEs**

Error terms involve the higher derivatives. What happens to error when \( \Delta x \) increases? If the error grows “faster” than the solution we have a problem, the solution is unstable and can become nonsensical. Such a system for which \( \Delta x \) must be restricted to smaller values, and the physical systems they model, are called stiff. Some solution methods can handle stiff problems (like the backward Euler method), others (like the forward Euler method) can only handle them by restricting \( \Delta x \).

Later we’ll examine multiple simultaneous odes. Let’s say the independent variable is time and the equations are linear, for simplicity. We’ll discover another version of stiffness when the various odes have considerably different time constants. Such stiff problems are harder to solve.
**Euler’s Methods**

Used extensively to solve a large number of hydrology problems, including heat transfer, fluid flow in channels and overland, groundwater flow, vadose zone flow, solute mass diffusion and advection/dispersion, etc.

**Euler’s Forward or Explicit Method**

See p. 89-90 above for derivation, and pp. 887-889 in text, for background and example. Works on linear or non-linear problems, but is only conditionally stable.

\[ y_{n+1} \approx y_n + \Delta x \cdot f(x_n, y_n) + O(\Delta x^2) \]  

(8)

Figure 2 Euler’s explicit method numerical and true solution

**Example from text:**

Apply the Euler method to the following initial value problem, choosing \( h = 0.2 \) and computing \( y_1, \cdots, y_5 \):

\[ y' = x + y, \quad y(0) = 0. \]

**Solution.** Here \( f(x, y) = x + y \); hence \( f(x_n, y_n) = x_n + y_n \), and we see that (3) becomes

\[ y_{n+1} = y_n + 0.2(x_n + y_n). \]

**Euler’s Backward or Fully Implicit Method**

See pp. 92-93 above for derivation and p. 896 of text for discussion and example. The method is unconditionally stable.
Evaluating the right-hand side is no problem for a linear problem, but can require something like a fixed-point or Newton iteration for a non-linear problem. The method is also particularly useful for stiff problems (text, p. 896).

**Euler’s Midpoint** (or the Crank-Nicholson) Method

This method is not mentioned in the text. We review it below, as a special case of the so-called general Euler \( \theta \)-method. The midpoint method is higher order (second order) and stable, and often used in hydrology.

**Stability Issues**

We’ll review these in discussing the general Euler \( \theta \)-method, below.

**Modified Euler’s Method – a Predictor-Corrector Approach**

This is called an improved Euler method or Huen’s method in the text, pp. 889-891, which includes pseudocode (see below) and an example. Note that the first equation is the prediction step and the second is the corrector step. The prediction step is a simple forward, explicit prediction.

\[
y_{n+1} \approx y_n + \Delta x f\left(x_n, y_n\right)
\]

\[
y_{n+1} \approx y_n + \frac{\Delta x}{2} \left[f\left(x_n, y_n\right) + f\left(x_{n+1}, y^*_{n+1}\right)\right]
\]

As one would expect this has higher order local and global error (see p. 891 for proof). The local error is of order \( \Delta x^3 \) and the global error is of order \( \Delta x^2 \). In other words, this method is a **second order method**. Also, note how this method handles non-linearities.

The pseudo-code is:

<table>
<thead>
<tr>
<th>Table 21.2 Improved Euler Method (Heun’s Method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGORITHM EULER ( f, x_0, y_0, h, N )</td>
</tr>
<tr>
<td>This algorithm computes the solution of the initial value problem ( y' = f(x, y), y(x_0) = y_0 ) at equidistant points ( x_1 = x_0 + h, x_2 = x_0 + 2h, \ldots, x_N = x_0 + Nh ); here ( f ) is such that this problem has a unique solution on the interval ( [x_0, x_N] ) (see Sec. 1.7).</td>
</tr>
<tr>
<td>INPUT: Initial values ( x_0, y_0 ), step size ( h ), number of steps ( N )</td>
</tr>
<tr>
<td>OUTPUT: Approximation ( y_{n+1} ) to the solution ( y(x_{n+1}) ) at ( x_{n+1} = x_0 + (n + 1)h ), where ( n = 0, \ldots, N - 1 )</td>
</tr>
<tr>
<td>For ( n = 0, 1, \ldots, N - 1 ) do:</td>
</tr>
<tr>
<td>( x_{n+1} = x_n + h )</td>
</tr>
<tr>
<td>( k_1 = hf(x_n, y_n) )</td>
</tr>
<tr>
<td>( k_2 = hf(x_{n+1}, y_n + k_1) )</td>
</tr>
<tr>
<td>( y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2) )</td>
</tr>
<tr>
<td>OUTPUT: ( x_{n+1}, y_{n+1} )</td>
</tr>
<tr>
<td>End</td>
</tr>
<tr>
<td>Stop</td>
</tr>
<tr>
<td>End EULER</td>
</tr>
</tbody>
</table>
Exact solutions to an ODE
In order to better understand truncation error in the Euler method we need to examine exact solutions, written in a similar way, and compare to the various Euler approximations. In this section we look at exact solutions. In the next we reexamine approximate solutions and truncation error. Keep in mind that in this section everything is exact, even though we are representing solutions by (infinite) series. In the next section we truncate those series to get the approximate solutions.

Consider a first order ODE in the interval $x_0 \leq x \leq x_N$. It can be linear or non-linear. The equation is

$$y'(x) = f(x, y) \quad (1)$$

where $y$ is the unknown, $x$ is the independent variable, $f$ is a function of $x$ and $y$, $y'(x)$ is the derivative of $y$ at location $x$. This equation applies anywhere on domain along the $x$-axis where $f$ is defined. Now consider $n = 0, 1, 2, 3, \ldots N-1, N$ specific points, $x_n$, on the $x$-axis, each separated from its $n+1$st neighbor to the right by some distance $\Delta x_n = x_{n+1} - x_n$. Note that unlike our discussion above, we are now allowing the point spacing to vary. Equation (1) applies everywhere in the interval $x_0 \leq x \leq x_N$, and therefore it applies at each point $x_n$ and its neighbor to the right, $x_{n+1}$. Assume that (1) has exact solutions at these two points, respectively $y(x_n)$ and $y(x_{n+1})$, where (1) is written for each point as

$$y'(x_n) = f(x_n, y(x_n)) = f(x_n) \quad (2)$$

$$y'(x_{n+1}) = f(x_{n+1}, y(x_{n+1})) = f(x_{n+1}) \quad (3)$$

On the right, I’ve introduced short-hand notation for the function $f$ to indicate its $x$ location, not its dependence. It is understood that it can depend on the value of both $x$ and $y$ at that location, as explicitly shown in (1).

Forward Taylor Series Expansion
The solution at $x_{n+1}$ can be written as a Taylor series expansion around the solution at $x_n$. As long as all terms in the expansion are retained, the expansion is exact.

$$y(x_{n+1}) = y(x_n) + (x_{n+1} - x_n) y'(x_n) + \frac{(x_{n+1} - x_n)^2}{2!} y''(x_n) + \frac{(x_{n+1} - x_n)^3}{3!} y'''(x_n) + \ldots \quad (4)$$

This is called a forward expansion. It projects forward in $x$, from $x_n$ to $x_{n+1}$. Recalling the definition of the increment or spacing, $\Delta x_n$, this forward expansion becomes
\[ y(x_{n+1}) = y(x_n) + \Delta x_n \cdot y'(x_n) + \frac{\Delta x_n^2}{2!} \cdot y''(x_n) + \frac{\Delta x_n^3}{3!} \cdot y'''(x_n) + \ldots \] (5)

We can now solve (5) for \( y'(x_n) \), which we’ll eventually substitute into (2).

\[ y'(x_n) = \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} + \frac{\Delta x_n}{2!} \cdot y''(x_n) - \frac{\Delta x_n^2}{3!} \cdot y'''(x_n) - \ldots \] (6)

This expansion for the derivative at \( x_n \), is also called a forward expansion. Once again, it is exact.

When (6) is substituted into (2) we get

\[ \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} - \frac{\Delta x_n}{2!} \cdot y''(x_n) + \frac{\Delta x_n^2}{3!} \cdot y'''(x_n) + \ldots = f(x_n) \] (7a)

or

\[ \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} = f(x_n) + \frac{\Delta x_n}{2!} \cdot y''(x_n) + \frac{\Delta x_n^2}{3!} \cdot y'''(x_n) + \ldots \] (7b)

which is a new but still exact version of (2), written as a forward difference and series expansion around \( x_n \). Expansions (5) and (7) will later become building blocks for finite difference numerical approximations, when the higher order terms are dropped.

Since by (2) \( y'(x_n) = f(x_n) \), these expansions can also be written in terms of derivatives of the function \( f \). For example (5) becomes

\[ y(x_{n+1}) = y(x_n) + \Delta x_n \cdot f(x_n) + \frac{\Delta x_n^2}{2!} f'(x_n) + \frac{\Delta x_n^3}{3!} f''(x_n) + \ldots \] (8)

where, e.g., \( f' \) is the (partial) derivative of \( f \) wrt \( x \). And (6) becomes

\[ y'(x_n) = \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} - \frac{\Delta x_n}{2!} f'(x_n) + \frac{\Delta x_n^2}{3!} f''(x_n) - \ldots \] (9)

When (9) is substituted into (2), we get a new version of (7),

\[ \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} - \frac{\Delta x_n}{2!} f'(x_n) + \frac{\Delta x_n^2}{3!} f''(x_n) - \ldots = f(x_n) \] (10a)

or

\[ \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} = f(x_n) + \frac{\Delta x_n}{2!} f'(x_n) + \frac{\Delta x_n^2}{3!} f''(x_n) + \ldots \] (10b)

The higher order terms of these series are non-zero only if the respective derivative wrt \( x \) of \( f \) is non-zero. For example, if \( f \) is a constant then only the first term on the RHS of (10b) is non-zero.

If \( f \) is a linear function of \( x \) (and therefore the ODE (2) is linear), then only the first two terms are non-zero. Etc.

**Backward Taylor Series Expansion**

The solution at \( x_n \) can be written as a backward Taylor series expansion around the solution at \( x_{n+1} \).
\[ y(x_n) = y(x_{n+1}) + (x_n - x_{n+1}) y'(x_{n+1}) \]
\[ + \frac{(x_n - x_{n+1})^2}{2!} y''(x_{n+1}) + \frac{(x_n - x_{n+1})^3}{3!} y'''(x_{n+1}) + \ldots \] (11)

Substituting step size, \( \Delta x_n = x_{n+1} - x_n \), this backward expansion becomes

\[ y(x_n) = y(x_{n+1}) - \Delta x_n y'(x_{n+1}) + \frac{\Delta x_n^2}{2!} y''(x_{n+1}) - \frac{\Delta x_n^3}{3!} y'''(x_{n+1}) + \ldots \] (12)

which resembles (5), but with alternating signs (since \( x \) is going “backwards”) and with the derivatives of \( y \) taken at a different point. We can rewrite in terms of derivatives of the function \( f \).

\[ y(x_n) = y(x_{n+1}) - \Delta x_n f(x_{n+1}) + \frac{\Delta x_n^2}{2!} f''(x_{n+1}) - \frac{\Delta x_n^3}{3!} f'''(x_{n+1}) + \ldots \] (13)

and then rewrite it for \( y(x_{n+1}) \).

\[ y(x_{n+1}) = y(x_n) + \Delta x_n f(x_{n+1}) - \frac{\Delta x_n^2}{2!} f''(x_{n+1}) + \frac{\Delta x_n^3}{3!} f'''(x_{n+1}) + \ldots \] (14)

We can also get a backward expansion for the ode, solving (12) for \( y'(x_{n+1}) \), which we then substitute into (3). The backward expansion is

\[ y'(x_{n+1}) = \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} + \frac{\Delta x_n}{2!} y''(x_{n+1}) - \frac{\Delta x_n^2}{3!} y'''(x_{n+1}) + \ldots \] (15)

and the ode becomes

\[ \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} = f(x_{n+1}) - \frac{\Delta x_n}{2!} y''(x_{n+1}) + \frac{\Delta x_n^2}{3!} y'''(x_{n+1}) + \ldots \] (16)

or

\[ \frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} = f(x_{n+1}) - \frac{\Delta x_n}{2!} f''(x_{n+1}) + \frac{\Delta x_n^2}{3!} f'''(x_{n+1}) + \ldots \] (17)

which has been written in terms of derivatives of the function \( f \). These expansions at \( x_{n+1} \) for the derivative and the backward difference are also called backward expansions.

**Euler’s Forward and Backward Methods**

We now develop approximate models for \( y \) and \( y' \), at \( x_n \) and \( x_{n+1} \), by dropping the higher order terms in the Taylor Series expansions. We approximate (2) by a truncated forward expansion and (3) by a truncated backward expansion. Notationally, where \( y(x_{n+1}) \) is the exact solution for \( y \) at \( x_{n+1} \), we write \( y_{n+1} \) to represent the approximate numerical solution at the same point. That is, subscript notation is used to denote a discrete approximation. Note that we hope that \( y_{n+1} \approx y(x_{n+1}) \), but how well we meet this goals depends on the truncation error (and the stability of the method).

**Euler’s forward method**

The (exact) forward expansion at \( x_n \) for the ODE (2), is given in (7b). It can be rewritten as
\[
\frac{y(x_{n+1}) - y(x_n)}{\Delta x_n} = f(x_n) + \frac{\Delta x_n^2}{2!} y''(\xi)
\]  
(18)

where the last term represents the rest of the series, and \(\xi\) is somewhere in the interval \(x_n\) to \(x_{n+1}\). That is, this one term is exactly equivalent to the rest of the series (if you know how to pick \(\xi\)). If we neglect higher order terms then the no longer have an exact equation, but a numerical approximation. We write this as

\[
\frac{y_{n+1} - y_n}{\Delta x_n} \approx f_n + O(\Delta x_n)
\]  
(19)

By truncating the higher order terms in (18), the solution to (19) is not exact solution of (2). The second term on the RHS represents that truncation error. Note the important notational change. Where \(y(x_{n+1})\) is the exact solution for \(y\) at \(x_{n+1}\) and satisfies (18) exactly, the variable \(y_{n+1}\) is the approximate numerical solution at the same point, and (19) is only an approximation. The second term on the RHS of (19) represents that truncation error, and is used to indicate that the order of local ODE error is \(\Delta x_n\). That is, it depends on the step size taken to the first power. For example, if the ODE is one of mass balance then this represents the local mass balance error. Often we don’t write the truncation error term, instead just writing the difference equation,

\[
\frac{y_{n+1} - y_n}{\Delta x_n} \approx f(x_n)
\]  
(20)

The forward expansion for \(y\), from \(x_n\) to \(x_{n+1}\), is given by (5) and (8). We approximate it by truncating the higher order terms to yield an approximate numerical solution for \(y(x_{n+1})\),

\[
y_{n+1} \approx y_n + \Delta x_n f(x_n) + \frac{\Delta x_n^2}{2!} y''(x_n)
\]  
(21)

or

\[
y_{n+1} \approx y_n + \Delta x_n f(x_n) + \frac{\Delta x_n^2}{2!} f'(x_n)
\]  
(22)

We write this as

\[
y_{n+1} \approx y_n + \Delta x_n f(x_n) + O(\Delta x_n^2)
\]  
(23)

to indicate that the order of local truncation error for \(y_{n+1}\) is \(\Delta x_n^2\), and then without the truncation error just to indicate the solution to the forward difference (20).

\[
\left[ y_{n+1} \approx y_n + \Delta x_n f(x_n) \right]
\]  
(24)

The function \(f\) in the second term on the RHS is evaluated the old location, \(x_n\).

**Euler’s backward method**

We use equations (12)-(17), truncating higher order terms, to find the differences used in Euler’s backward difference method. From (16) or (17) we get the backward difference approximation to ODE (3)

\[
\frac{y_{n+1} - y_n}{\Delta x_n} \approx f(x_{n+1}) + O(\Delta x_n)
\]  
(25)

or writing it without the truncation error,
The backward expansion for $y_n$, from $x_n+1$ to $x_n$ is given by (12) and (13). Approximating (13) we have

$$y_n \approx y_{n+1} - \Delta x_n f(x_{n+1}) + O(\Delta x_n^2)$$

(27)

However, we are interested in finding $y_{n+1}$ given $y_n$. The expansion for this is (14), which we approximate as

$$y_{n+1} \approx y_n + \Delta x_n f(x_{n+1}) + O(\Delta x_n^2)$$

(28)

We can also get this from (27). It is usually written without the order of local truncation error,

$$y_{n+1} \approx y_n + \Delta x_n f(x_{n+1})$$

(29)

Comparing the backward and forward methods we see that they have the same order of truncation error, but that the function $f$ in the second term on the RHS is different. For the backward method it is evaluated at the new location, $x_{n+1}$. Later, we’ll see that the choice of a backward or forward method turns on stability and computational effort.

**General Euler $\theta$-Method**

We can approximate the derivative $y'$ at any point $x_{n+\theta}$, $0 \leq \theta \leq 1$, within the interval $x_n$ to $x_{n+1}$, by a linear combination of the derivatives at the end points.

$$y'(x_{n+\theta}) \approx (1-\theta)y'(x_n) + \theta y'(x_{n+1})$$

(30)

where the derivatives on the right side are exact and are given by (2) and (3). $\theta$ is a weighting coefficient. We later replace the derivatives with their numerical approximations. We also have the following special values of $\theta$ and their associated numerical approximation:

- $\theta = 0$, forward or “explicit” Euler Method
- $\theta = 1$, backward or “fully implicit” Euler Method
- $\theta = 0.5$, midpoint Euler Method
  ($\theta = 0.5$ also called “Crank-Nicholson” when used to solve time dependent PDEs.)

We’ve already seen the results for $\theta = 0$ and 1.

For $0 < \theta \leq 1$ the method is implicit. For $\theta = 1$ it is **fully implicit**. Your textbook refers to the fully implicit Euler method simply as the implicit method. It is the only implicit Euler method discussed in there. The general method, and in particular the midpoint method, are not discussed. However, they are commonly used to solve the time derivative aspect of transient PDEs in hydrology, so we discuss them here.

The exact ODE (1) at point $x_{n+\theta}$ is given by
\[ y'(x_{n+\theta}) = f(x_{n+\theta}) \]  \hspace{1cm} (31)

We can write the LHS in terms of the weighted derivatives at the end points given in (30). We can also write the RHS in terms of \( f \) weighted at the end points.

\[ f(x_{n+\theta}) \equiv (1-\theta)f(x_n) + \theta f(x_{n+1}) \]  \hspace{1cm} (32)

Note if we substitute (30) and (32) into (31) we get

\[ (1-\theta)y'(x_n) + \theta y'(x_{n+1}) = (1-\theta)f(x_n) + \theta f(x_{n+1}) \]

which satisfies (1) exactly (if we ignore the fact that (30) and (32) individually are not exact). To proceed we need the numerical approximations to the derivatives in (30) and the values of \( f \) in (32). For example, \( f(x_n) = f(x_n, y(x_n)) \), where \( y(x_n) \) is approximated by \( y_n \) in the numerical solution.

Truncate the series in (6) to develop a forward difference to approximate \( y'(x_n) \), and truncate the series in (15) to get a backward difference to approximate \( y'(x_{n+1}) \), or

\[ y'(x_n) \approx \frac{y_{n+1} - y_n}{\Delta x_n} \quad \text{and} \quad y'(x_{n+1}) \approx \frac{y_{n+1} - y_n}{\Delta x_n} \]  \hspace{1cm} (33a,b)

Both differences are of order \( \Delta x_n \) and, while the derivatives are at different locations \( x \), their differences (RHS of (33)) are the same. Combine (30)-(33) to get

\[ \frac{y_{n+1} - y_n}{\Delta x_n} \approx (1-\theta)f_n + \theta f_{n+1} \]  \hspace{1cm} (34)

where \( f_n \) represents the numerical approximation \( f_n \) to \( f(x_n, y(x_n)) \), etc. The numerical solution \( y_{n+1} \), approximating \( y(x_{n+1}) \), is given by

\[ y_{n+1} \approx y_n + \Delta x_n [(1-\theta)f_n + \theta f_{n+1}] \]  \hspace{1cm} (35)

As you can see this becomes the forward difference solution (24) when \( \theta = 0 \) and the backward difference solution (29) when \( \theta = 1 \). Its easy to shown that its local truncation error in (35) is of order \( \Delta x_n^2 \), as we already found for the explicit and fully implicit cases, with the exception of a special case, when \( \theta = 0.5 \), the midpoint method, for which the local truncation error is of order \( \Delta x_n \).

For an ODE it is possible to write a general code representing (35), with \( \theta \) simply an input parameter. In this way one can leave it to the user to choose which Euler method to use.

Performance of the method varies with the equation (function \( f \)) being solved, the step size \( \Delta x_n \), and the choice of weight \( \theta \). For example, we’ll see later that for \( 0.5 \leq \theta \leq 1 \) the Euler method is
unconditionally stable, but that for $0 \leq \theta < 0.5$ stability is conditional (constrains $\Delta x_n < \text{some critical value in order not to “blow up”}$).

Consider the linear first-order ODE (p. 26 of text), $y'(x) + p(x)y = r(x)$, where $p$ and $r$ are functions depending on the independent variable, $x$. Then $f = r - py$ and equation (35) becomes

$$y_{n+1} \simeq y_n + \Delta x_n[ (1 - \theta)(r_n - p_n y_n) + \theta (r_{n+1} - p_{n+1} y_{n+1}) ]$$

(36)

Collecting terms involving $y_{n+1}$, which is implicit in (36),

$$[(1 + \Delta x_n \theta p_{n+1}) y_{n+1} \simeq [1 - \Delta x_n (1 - \theta) p_n ] y_n + \Delta x_n [(1 - \theta) r_n + \theta r_{n+1}]$$

(37)

The first term on the RHS represents the response to the IC in an initial value problem, while the second term represents the effects of the forcing $r$ (mimicking the exact solution; see (5) on p. 78 of these notes or p. 28 of the text). Continuing to simplify for $y_{n+1}$ we get the algebraic equation

$$y_{n+1} \simeq \frac{[1 - \Delta x_n (1 - \theta)p_n]}{(1 + \Delta x_n \theta p_{n+1})} y_n + \Delta x_n \frac{(1 - \theta) r_n + \theta r_{n+1}}{(1 + \Delta x_n \theta p_{n+1})}$$

(38)

### What does this have to do with solving partial differential equations (PDEs)?

This general Euler approach is commonly used to solve the time derivative in PDE solutions of groundwater flow, heat transport, and solute mass transport. In those cases the space dimensions are usually handled by finite difference or finite element methods, to be introduced later. In the Euler approach to ODEs we divide by the coefficient, $(1 + \Delta x_n \theta p_{n+1})$, on the left-hand side of (37), in order to get (38). In the Euler approach to the time derivative of PDEs, the left-hand-side coefficient is actually an $m \times m$ matrix, where $m$ is the number of spatial node points. If $\theta = 0$, this coefficient is the identity matrix, $I$. Since $I^{-1} = I$ this division operation is trivial for the forward method. For $\theta > 0$ the coefficient is more complex and simultaneous equation solvers, like the Gauss Elimination, are needed. We’ll introduce these solvers a little later, when we start to deal with 2nd order ODEs and boundary value problems (BVPs). The choice of $\theta$ then effects how you solve the spatial part of the PDE. Many PDE codes allow fully implicit and midpoint solutions for the time derivative. Some allow any value of $\theta \geq 0.5$, or even $\theta > 0$. In these codes, programming both the option for an explicit solution ($\theta = 0$) and the option for implicit solutions ($\theta > 0$) is cumbersome, since only the implicit solutions require simultaneous equation solvers for space. Consequently, that combination is unusual. In some hydrology codes, different solvers are used in different parts of the spatial domain. So called mixed implicit-explicit time derivatives are used. These codes determine where (in space) the explicit solution ($\theta = 0$) is stable, and use that scheme in that part of the domain; an implicit solution (say, $\theta = 1$) is used elsewhere. This allocation is calculated dynamically.

### Midpoint Method

In the midpoint method $\theta = 0.5$, and (34) and (35) become
\[ \frac{y_{n+1} - y_n}{\Delta x_n} \approx \frac{1}{2} \left( f_n + f_{n+1} \right) \]  

(39)

and

\[ y_{n+1} \approx y_n + \frac{\Delta x_n}{2} \left( f_n + f_{n+1} \right) \]  

(40)

This method is said to have less local truncation error than Euler methods with other choices of \( \theta \), and the same as the improved or modified Euler method (Huen’s method). Why?

The exact ODE at the midpoint is

\[ y'(x_{n+1/2}) = f(x_{n+1/2}) \]  

(41)

Approximate \( y'(x_{n+1/2}) \), by (30) with \( \theta = 0.5 \).

\[ y'(x_{n+1/2}) \approx \frac{1}{2} \left[ y'(x_n) + y'(x_{n+1}) \right] \]  

(42)

and replace the derivatives by their series expansions (6) and (15), keeping all the terms.

\[ y'(x_{n+1/2}) \approx \frac{1}{2} \left[ \frac{y'(x_{n+1}) - y'(x_n)}{\Delta x_n} + \frac{\Delta x_n}{2!} y''(x_n) + \frac{\Delta x_n^2}{3!} y'''(x_n) - \cdots \right] + \]

\[ \frac{1}{2} \left[ \frac{y'(x_{n+1}) - y'(x_n)}{\Delta x_n} + \frac{\Delta x_n}{2!} y''(x_{n+1}) - \frac{\Delta x_n^2}{3!} y'''(x_{n+1}) \pm \cdots \right] \]  

(43)

Notice that the first terms of each expression on the RHS is the same. The second terms are very similar, but with a change in sign. They would cancel out if the second derivatives were the same at the two endpoints. Assume that they are approximately equal, and neglect these terms. The third terms have the same sign and will be additive. And it continues to alternate like this with every other term approximately zero. The result is

\[ y'(x_{n+1/2}) \approx \frac{y'(x_{n+1}) - y'(x_n)}{\Delta x_n} - \frac{\Delta x_n^2}{3!} y'''(\xi) \]  

(44)

When we truncate this exact expression, and substitute it and (32) into the original equation (41), the numerical difference and the solution for \( y_{n+1} \) are higher order.

\[ \frac{y_{n+1} - y_n}{\Delta x_n} \approx \frac{1}{2} \left( f_n + f_{n+1} \right) + O(\Delta x_n^2) \]  

(45)

\[ y_{n+1} \approx y_n + \frac{\Delta x_n}{2} \left( f_n + f_{n+1} \right) + O(\Delta x_n^3) \]  

(47)

That is, the midpoint-method local truncation order for \( y_{n+1} \) is \( \Delta x_n^3 \), while it is of order \( \Delta x_n^2 \) for all other Euler methods. For this reason the midpoint method is often programmed into code, and
is preferred by many users. While theoretically it might have a lower truncation error, actual performance is usually not so obviously better.

**Stability of Euler Methods**

Let’s examine stability by focusing on the linear ODE (p. 27 of text), \( y' + p(x) y = r(x) \), where \( p \) and \( r \) are independent variables. It will be easier to focus if we set the forcing \( r \) to zero, and if the coefficient \( p \) is a constant. That is, consider the linear ODE

\[
y' + p y = 0
\]  

This is a version of (1) with \( f = -py \). The exact solution of (48) is \( y = Ce^{-px} \), an exponential response. If we take the initial condition (IC) as \( y(x_n) \), and predict \( y(x_{n+1}) \), we can evaluate the constant \( C \), and write the particular solution

\[
y(x_{n+1}) = y(x_n) e^{-px_n \Delta x_n}
\]  

where \( \Delta x_n = x_{n+1} - x_n \). The dependent variable decays exponentially in value from the IC at \( x_n \) to the final condition at \( x_{n+1} \).

The general Euler method approximate solution for the same situation is given in (38) with \( r=0 \) and \( p \) = constant, or

\[
y_{n+1} \equiv y_n \left( 1 - \left( \frac{p}{1 + p \Delta x_n} \right) \right) = y_n A
\]  

where \( A \) is a so-called *amplification factor* that is attempting to mimic or approximate the exponential behavior in (49).

\[
A = \frac{1 - (1 - \theta) p \Delta x_n}{1 + \theta p \Delta x_n}
\]

How well does \( A \) mimic \( e^{-p\Delta x} \)?

First, note that \( 0 \leq e^{-p\Delta x} \leq 1 \), while these bounds don’t apply to \( A \). Then compare plots of \( A \) and \( e^{-p\Delta x} \) for various values of \( \theta \), say \( \theta = 0, 0.25, 0.5, 0.75, 1.0 \) using Matlab. For which values of \( \theta \) and \( p \Delta x \) is \( |A| > 1 \)? For which \( \theta \) does the error grow faster than the solution decays (stiffness\(^6\)) and becomes unstable (small perturbations grow)? Under what conditions is the solution always (unconditionally) stable? Here is a sketch.

---

\(^6\) See p. 896 of text.
If \( |A| > 1 \) we have uncontrolled growth and instability. Why? Recall that \( y_{n+1} = y_n A \), thus \( y_1 = y_0 A \), where \( y_0 \) is the IC, \( y_2 \approx y_1 A \approx y_0 A^2 \), \( y_3 \approx y_0 A^3 \), etc., so that \( y_n \approx y_0 A^n \). If \( |A| > 1 \) then \( A^n \) will blow up, and with it \( y_n \). We want to mimic decay, not growth. We need stability. That requires that \( |A| < 1 \). In the diagram note the values of \( A \) as \( p \Delta x \) approaches infinity. Call these \( A_\infty \).

From the diagram, if \( 0 \leq \theta < 0.5 \) stability depends on the value of \( p \Delta x \). The method is conditionally stable (\( |A_\infty| > 1 \)). If \( 0.5 \leq \theta \leq 1 \) the method is unconditionally stable (\( |A_\infty| \leq 1 \)). The midpoint method is borderline conditionally stable, (\( |A_\infty| = 1 \)). In practice it is desirable to choose \( 0.5 \leq \theta \leq 1 \) so solution never blows up. \( \theta = 0.5 \) and 1 are the most popular choices.

**Restricted step size for forward method**

What if you choose to use the explicit or forward method, with \( \theta = 0 \). From the sketch you can see that if \( p \Delta x_n > 2 \), then the method will blow up. This restricts the step size, \( \Delta x_n \), so that

\[
p \Delta x_n \leq 2 \quad \text{or} \quad \Delta x_n \leq 2/p
\]

To avoid instabilities. Recall from our discussion of the analytical solution for this problem that the \( e \)-fold drop decay constant (or “time constant”) is \( 1/p \); i.e., \( e^{-p \Delta x_n} = e^{-\Delta x_n / \text{decay constant}} \) Thus

\[
\Delta x_n \leq 2 \times \text{decay constant}
\]

We want the step size to be smaller than the decay constant anyway, so this doesn’t seem so onerous, despite what our textbook says. However, when solving the time domain for big hydrology PDEs (see box on p. 103 of these notes) the “time constant” is not that of the system, but of the spatial finite difference grid block or finite element used to handle the spatial part of the problem. In that case our \( p \) is equivalent to the eigenvalue of that block or element in the “conductivity” coefficient matrix, and strongly restricts the solution. We’ll visit this later.
Runge-Kutta Methods

**Background** (text, pp. 892-898; §8.5 Gilat and Subramaniam, 2008)

Once again we are solving the IV problem

\[ y' = f(x, y) \quad y(x_0) = y_0 \]  

(1)

without restricting it to linear equations. Recall the general approach for single-step explicit methods,

\[ x_{n+1} \approx x_n + \Delta x_n \]
\[ y_{n+1} \approx y_n + \Delta x_n \cdot \text{slope} \]  

(16a,b)

The explicit Euler method used the *slope* of \( y \) at \( x_n \). The Runge-Kutta (RK) method calculates the *slope* from a weighted average of estimates of the slope of \( y \) at several points within the interval \( x_n \) to \( x_{n+1} \).

Higher order RK methods use more points and have smaller truncation error. For example, the so-called “classical” RK method is of 4\(^{th}\) order (globally) and uses four points. This is the method described in our textbook. The second order RK method uses only two points, and is 2\(^{nd}\) order (globally). (Recall that the backward and forward Euler methods are 1\(^{st}\) order globally. The modified Euler method and the midpoint Euler method are 2\(^{nd}\) order globally, just like the 2\(^{nd}\) order RK method.)

The difference between the methods is in the location of the points, within the interval, that are used to calculate the slope.

**Second-order RK methods** (not covered in the textbook)

The general form is given by

\[ y_{n+1} = y_n + (c_1k_1 + c_2k_2)\Delta x_n \]

with \( k_1 = f(x_n, y_n) \) and \( k_2 = f(x_n + a_2\Delta x_n, y_n + b_{21}k_1\Delta x_n) \), and where \( c_1, c_2, a_2, \) and \( b_{21} \) are constants. The modified Euler method is a special case of the 2\(^{nd}\) order RK scheme (particular values of these constants), with both \( c_1 = \frac{1}{2}, c_2 = \frac{1}{2}, a_2 = 1, \) and \( b_{21} = 0 \).

**Third-order RK methods** (not covered in the textbook)

The general form is given by

\[ y_{n+1} = y_n + (c_1k_1 + c_2k_2 + c_3k_3)\Delta x_n \]

with particular definitions for the \( k \)'s, and eight constants to assign values to. The classical 3\(^{rd}\) order RK method has particular values of the constants, and is related to Simpson’s rule of numerical integration (§8.5 Gilat and Subramaniam, 2008)).
Fourth-order RK methods (pp. 892-894 of textbook)
The general form is given by

$$y_{n+1} = y_n + (c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4)\Delta x$$

with

$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + a_2\Delta x, y_n + b_21k_1\Delta x)$$
$$k_3 = f(x_n + a_3\Delta x, y_n + b_31k_1\Delta x + b_32k_2\Delta x)$$
$$k_4 = f(x_n + a_4\Delta x, y_n + b_41k_1\Delta x + b_42k_2\Delta x + b_43k_3\Delta x)$$

In the classical 4th order RK method

$$c_1 = 1/6, c_2 = 2/6, c_3 = 2/6, c_4 = 1/6,$$
$$a_2 = 1/2, a_3 = 1/2, a_4 = 1, and$$
$$b_{21} = 1/2, b_{31} = 0, b_{32} = 1/2, b_{41} = 0, b_{42} = 0, b_{43} = 1$$

From the text, an explanation and pseudo-code:

A method of great practical importance and much greater accuracy than that of the improved Euler method is the classical Runge–Kutta method of fourth order, which we call briefly the Runge–Kutta method. It is shown in Table 21.4. We see that in each step we first compute four auxiliary quantities $k_1, k_2, k_3, k_4$ and then the new value $y_{n+1}$. The method is well suited to the computer because it needs no special starting procedure, makes light demand on storage, and repeatedly uses the same straightforward computational procedure. It is numerically stable.

Note that if $f$ depends only on $x$, this method reduces to Simpson's rule of integration (Sec. 19.5). Note further that $k_1, \cdots, k_4$ depend on $n$ and generally change from step to step.

Table 21.4 Classical Runge–Kutta Method of Fourth Order

<table>
<thead>
<tr>
<th>ALGORITHM RUNGE–KUTTA $(f, x_0, y_0, h, N)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This algorithm computes the solution of the initial value problem $y' = f(x, y), y(x_0) = y_0$ at equidistant points</td>
</tr>
<tr>
<td>$x_1 = x_0 + h, x_2 = x_0 + 2h, \cdots, x_N = x_0 + Nh$;</td>
</tr>
<tr>
<td>here $f$ is such that this problem has a unique solution on the interval $[x_0, x_N]$ (see Sec. 1.7).</td>
</tr>
<tr>
<td>INPUT: Function $f$, initial values $x_0, y_0$, step size $h$, number of steps $N$</td>
</tr>
<tr>
<td>OUTPUT: Approximation $y_{n+1}$ to the solution $y(x_{n+1})$ at $x_{n+1} = x_0 + (n + 1)h$, where $n = 0, 1, \cdots, N - 1$</td>
</tr>
</tbody>
</table>
For \( n = 0, 1, \ldots, N - 1 \) do:

\[
\begin{align*}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\
k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\
k_4 &= hf(x_n + h, y_n + k_3)
\end{align*}
\]

\[
\begin{align*}
x_{n+1} &= x_n + h \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{align*}
\]

OUTPUT \( x_{n+1}, y_{n+1} \)

End

Stop

End RUNGE-KUTTA

### Classical Runge-Kutta Method

Apply the Runge-Kutta method to the initial value problem (4) in Example 1, choosing \( h = 0.2 \), as before, and computing five steps.

**Solution.** For the present problem we have \( f(x, y) = x + y \). Hence

\[
\begin{align*}
k_1 &= 0.2(x_n + y_n), & k_2 &= 0.2(x_n + 0.1 + y_n + 0.5k_1), \\
k_3 &= 0.2(x_n + 0.1 + y_n + 0.5k_2), & k_4 &= 0.2(x_n + 0.2 + y_n + k_3).
\end{align*}
\]

Table 21.5 shows the results and their errors, which are smaller by factors \( 10^3 \) and \( 10^4 \) than those for the two Euler methods. See also Table 21.6. We mention in passing that since the present \( k_1, \ldots, k_4 \) are simple, operations were saved by substituting \( k_1 \) into \( k_2 \), then \( k_2 \) into \( k_3 \), etc.; the resulting formula is shown in Column 4 of Table 21.5.

#### Table 21.5 Runge-Kutta Method Applied to (4)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( x_n )</th>
<th>( y_n )</th>
<th>( 0.2214(x_n + y_n) + 0.0214 )</th>
<th>Exact Values (6D)</th>
<th>( 10^6 \times ) Error of ( y_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0</td>
<td>0.021 400</td>
<td>0.000 000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.2</td>
<td>0.021 400</td>
<td>0.070 418</td>
<td>0.021 403</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
<td>0.091 818</td>
<td>0.130 289</td>
<td>0.091 825</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0.6</td>
<td>0.222 107</td>
<td>0.203 414</td>
<td>0.222 119</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>0.8</td>
<td>0.425 521</td>
<td>0.292 730</td>
<td>0.425 541</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>0.718 251</td>
<td>0.718 282</td>
<td>0.718 282</td>
<td>31</td>
</tr>
</tbody>
</table>

### Comparison from the text:

#### Table 21.6 Comparison of the Accuracy of the Three Methods Under Consideration in the Case of the Initial Value Problem (4), with \( h = 0.2 \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y = e^x - x - 1 )</th>
<th>Euler (Table 21.1)</th>
<th>Improved Euler (Table 21.3)</th>
<th>Runge-Kutta (Table 21.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.021 403</td>
<td>0.021</td>
<td>0.0014</td>
<td>0.000 003</td>
</tr>
<tr>
<td>0.4</td>
<td>0.091 825</td>
<td>0.052</td>
<td>0.0034</td>
<td>0.000 007</td>
</tr>
<tr>
<td>0.6</td>
<td>0.222 119</td>
<td>0.094</td>
<td>0.0063</td>
<td>0.000 011</td>
</tr>
<tr>
<td>0.8</td>
<td>0.425 541</td>
<td>0.152</td>
<td>0.0102</td>
<td>0.000 020</td>
</tr>
<tr>
<td>1.0</td>
<td>0.718 282</td>
<td>0.229</td>
<td>0.0156</td>
<td>0.000 031</td>
</tr>
</tbody>
</table>
Runge-Kutta-Felhberg (RKF), Error and Step Size control with RK

The idea of adaptive integration (Sec. 19.5) has analogs for Runge-Kutta (and other) methods. In Table 21.4 for RK (Runge-Kutta), if we compute in each step approximations \( \hat{y} \) and \( \tilde{y} \) with step sizes \( h \) and \( 2h \), respectively, the latter has error per step equal to \( 2^5 = 32 \) times that of the former; however, since we have only half as many steps for \( 2h \), the actual factor is \( 2^5/2 = 16 \), so that, say,

\[
\epsilon^{(2h)} \approx 16\epsilon^{(h)}
\]

and thus

\[
y^{(h)} - y^{(2h)} = \epsilon^{(2h)} - \epsilon^{(h)} \approx (16 - 1)\epsilon^{(h)}.
\]

Hence the error \( \epsilon = \epsilon^{(h)} \) for step size \( h \) is about

\[
(10) \quad \epsilon = \frac{1}{16} (\hat{y} - \tilde{y})
\]

where \( \hat{y} - \tilde{y} = y^{(h)} - y^{(2h)} \), as said before. Table 21.7 illustrates (10) for the initial value problem

\[
y' = (y - x - 1)^2 + 2, \quad y(0) = 1,
\]

the step size \( h = 0.1 \) and \( 0 \leq x \leq 0.4 \). We see that the estimate is close to the actual error. This method of error estimation is simple but may be unstable.

<table>
<thead>
<tr>
<th>( x ) (Step size ( h ))</th>
<th>( \hat{y} ) (Step size ( 2h ))</th>
<th>Error Estimate (10)</th>
<th>Actual Error</th>
<th>Exact Solution (9D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.000 000 000</td>
<td>1.000 000 000</td>
<td>0.000 000 000</td>
<td>0.000 000 000</td>
</tr>
<tr>
<td>0.1</td>
<td>1.200 334 589</td>
<td>1.402 707 408</td>
<td>0.000 000 165</td>
<td>0.000 000 157</td>
</tr>
<tr>
<td>0.2</td>
<td>1.402 709 878</td>
<td>1.822 788 993</td>
<td>0.000 000 267</td>
<td>0.000 000 226</td>
</tr>
<tr>
<td>0.3</td>
<td>1.609 336 039</td>
<td>2.222 222 222</td>
<td>0.000 000 357</td>
<td>0.000 000 308</td>
</tr>
<tr>
<td>0.4</td>
<td>1.822 792 993</td>
<td>2.644 444 444</td>
<td>0.000 000 451</td>
<td>0.000 000 396</td>
</tr>
</tbody>
</table>

One can run RK with these two different step sizes, compare the results, and if the error estimate is too large reduce step size and do it again. In application, this is done adaptively at each step.

The Runge-Kutta-Felberg (RKF) algorithm uses a more sophisticated and efficient approach to step-size control. It uses two RK methods of different orders for each step, and picks those methods to minimize the number of arithmetic operations. If the error estimate is too large the step size is adjusted. The method has a global truncation error of \( 5^{th} \) order. The text covers this (pp. 894-895) because RKF is very popular (e.g., in Maple and Matlab). In hydrology it is the second most popular way to track particles in path line and travel time simulations, just after Euler methods.
Multistep Methods (see text, §21.3)

Single step methods use only one previous step, at $x_n$, in order to predict the value of the dependent variable, $y_{n+1}$, at the next step, $x_{n+1}$. Multistep methods look further backwards, to solutions and information at “older” steps $x_{n-1}$, $x_{n-2}$, etc. Recalling that we have certain smoothness constraints on the dependent variable $y$, this provides more information about how $y$ is behaving and a more accurate prediction of $y_{n+1}$. Different multistep methods use different amounts of “older” information (number of previous steps used) and employ that information differently. Because they use more “past” information each prediction step is more computationally expensive (more arithmetic operations and storage). The text reviews the Adams-Bashforth and Adams-Moulton multistep methods, both shown for their respective 4th order versions.

One of problems with implementing these methods is getting them started. At the beginning of a computation the first, initial value (IV $y_0$) is at $x_0$ with right side $f_0$. There are “old values” of $y$ and $f$ at $x_{-1}$, $x_{-2}$, etc. Consequently, you have to start the solution with another method, say RK or Euler, until you are far enough into the domain to have all the “old values” that you need with the multistep method.

Multistep methods are becoming more common in applied codes, so you should be familiar with the basic concepts, but there is no need to become expert in this introductory course.