Lecture `ode2`: Explicit RK methods, Butcher tables, error control

December 7, 2022

Summary: Description of explicit Runge-Kutta methods (in particular, classical RK4), their encoding via Butcher tables, and embedded methods for error control (in particular, the Runge-Kutta-Fehlberg method.

References: T. Sauer's *Numerical Analysis*, Sections 6.4 and 6.5, second edition, pages 314–332. See also John H. Mathews and Kurtis K. Fink, *Numerical Methods Using Matlab*, 4th Edition, 2004.

Forward/backward Euler as an example of an explicit/implicit method

Recall the generic initial value problem for a system of ordinary differential equations (ODE),

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}); \qquad \mathbf{y}(t_0) = \mathbf{y}_0. \tag{1}$$

Two methods we have seen for numerically solving an IVP are the forward and backward Euler methods:

$$\overbrace{\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k)}^{\text{forward Euler}}, \qquad \overbrace{\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})}^{\text{backward Euler}}, \tag{2}$$

where $h = \Delta t$ is the time-step size. Forward Euler is an *explicit* method, whereas backward Euler is an *implicit* method. To appreciate the distinction, consider the ODE

$$\frac{d^2\theta}{dt} + \gamma\frac{d\theta}{dt} + (g/\ell + A\cos 2\pi t)\sin\theta = 0, \tag{3}$$

describing the motion of a damped pendulum with an oscillating pivot. Here θ is the angle of deflection, g is the acceleration due to the Earth's gravity, ℓ is the length of the pendulum arm, A is the amplitude of the pivot forcing, and γ is a damping parameter. The above ODE is second-order in time, but we may write it as a first-order system via the introduction of an auxiliary variable $\omega = d\theta/dt$. Indeed, let $y_1 = \theta$ and $y_2 = \omega = d\theta/dt$, so that the above equation becomes a first order system

$$\frac{d}{dt}\begin{pmatrix} \theta \\ \omega \end{pmatrix} = \begin{pmatrix} \omega \\ -\gamma\omega - (g/\ell + A\cos 2\pi t)\sin\theta \end{pmatrix}. \tag{4}$$

For an IVP we would need to specify initial values θ_0, ω_0 for the angle and angular velocity. Notice that the first equation in this system $d\theta/dt = \omega$ is really just the definition of ω, whereas the second equation is the one governing the motion of the pendulum.

The forward Euler method applied to (4) is clearly

$$\begin{pmatrix} \theta_{k+1} \\ \omega_{k+1} \end{pmatrix} = \begin{pmatrix} \theta_k \\ \omega_k \end{pmatrix} + h\begin{pmatrix} \omega_k \\ -\gamma\omega_k - (g/\ell + A\cos 2\pi t_k)\sin\theta_k \end{pmatrix}, \tag{5}$$

or equivalently the following updates:

$$\theta_{k+1} = \theta_k + h\omega_k, \qquad \omega_{k+1} = (1 - h\gamma)\omega_k - h(g/\ell + A\cos 2\pi t_k)\sin\theta_k. \tag{6}$$

This is an explicit or "marching" scheme. Given t_k, θ_k, ω_k, we substitute these values into the right-hand sides above and obtain $\theta_{k+1}, \omega_{k+1}$. Of course, $t_{k+1} = a + (k+1)h$ is already known.

However, the backward Euler method applied to (4) is

$$\begin{pmatrix} \theta_{k+1} \\ \omega_{k+1} \end{pmatrix} = \begin{pmatrix} \theta_k \\ \omega_k \end{pmatrix} + h \begin{pmatrix} \omega_{k+1} \\ -\gamma\omega_{k+1} - (g/\ell + A\cos 2\pi t_{k+1})\sin\theta_{k+1} \end{pmatrix}, \tag{7}$$

or equivalently the equations

$$\theta_{k+1} - h\omega_{k+1} = \theta_k, \qquad (1 + h\gamma)\omega_{k+1} + h(g/\ell + A\cos 2\pi t_{k+1})\sin\theta_{k+1} = \omega_k. \tag{8}$$

The point now is that this is a nonlinear system of equations. To find $\theta_{k+1}, \omega_{k+1}$ (to "take a time step") requires a root-finding algorithm to solve this system.

An explicit method gives a direct way to perform a time step, whereas an implicit method typically requires root-finding. Why would we ever want to consider implicit methods? It turns out that these methods have better stability properties. Roughly, they allow for larger time-step size h without compromising the quality of the numerical solutions. For some scenarios implicit methods, with large time-step size h, are the way to go. But we will not consider them further here. We focus on a class of relatively simple explicit methods: explicit Runge-Kutta methods. (There are also implicit Runge-Kutta methods, backward Euler being an example).

Explicit Runge-Kutta methods

We could continue with the generic system (1), but for simplicity switch to a scalar IVP

$$y' = f(t, y); \qquad y(t_0) = y_0. \tag{9}$$

We describe a family of methods for numerically solving the ODE IVP known as *Runge-Kutta methods*. Given only data (t_k, y_k) they determine data (t_{k+1}, y_{k+1}) at the next time-step using (typically nested) function evaluations involving the righthand side $f(t, y)$. Other methods certainly exist, for example *multistep methods* which compute (t_{k+1}, y_{k+1}) from multiple previous states (t_k, y_k), (t_{k-1}, y_{k-1}), $\cdots$, (t_{k-r}, y_{k-r}).

A Runge-Kutta method is determined by a Butcher table (or tableau, the later expression being better for a Google search)

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b} \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s, \end{array}$$

where $\mathbf{c}$ are the *stage times* or *nodes*, $\mathbf{b}$ are the *stage expansions* or *weights*, and A is the Runge-Kutta matrix. The integer s is the number of stages. This is not quite the order of the method, but larger s typically corresponds to larger order. For an explicit RK method, A is a strictly lower triangular matrix and we have

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b} \end{array} = \begin{array}{c|cccc} c_1 & 0 & 0 & \cdots & 0 \\ c_2 & a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s, \end{array}$$

To see how to use a Butcher table

Forward Euler:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \begin{aligned} k_1 &= f(t_k, y_k) \\ y_{k+1} &= y_k + hk_1 \end{aligned}$$

Explicit midpoint:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array} \qquad \begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + \tfrac{1}{2}h, y_k + \tfrac{1}{2}hk_1) \\ y_{k+1} &= y_k + hk_2 \end{aligned}$$

Explicit trapezoid:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + h, y_k + hk_1) \\ y_{k+1} &= y_k + \tfrac{1}{2}h(k_1 + k_2) \end{aligned}$$

Here is a four-stage method.

3/8 rule fourth-order method:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array} \qquad \begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + \tfrac{1}{3}h, y_k + \tfrac{1}{3}hk_1) \\ k_3 &= f(t_k + \tfrac{2}{3}h, y_k - \tfrac{1}{3}hk_1 + hk_2) \\ k_4 &= f(t_k + h, y_k + hk_1 - hk_2 + hk_3) \\ y_{k+1} &= y_k + \tfrac{1}{8}h(k_1 + 3k_2 + 3k_3 + k_4) \end{aligned}$$

While the previous example is a four-stage fourth order method, the most common one is the following.

Classical RK4:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \qquad \begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + \tfrac{1}{2}h, y_k + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_k + \tfrac{1}{2}h, y_k + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_k + h, y_k + hk_3) \\ y_{k+1} &= y_k + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

Here is some MATLABcode implemented a general explicit Runge-Kutta method through an input Butcher table

```
%  UNM CS/Math 375, Spring Semester 2021
%  ODE integration by an explicit Runge-Kutta method read in as Butcher table.
%  function [y] = ExplicitRK(f,t0,MAXTIME,y0,nsteps)
   function [y] = ExplicitRK(f,t0,MAXTIME,y0,nsteps)
h = (MAXTIME-t0)/nsteps; y = y0; t = t0;
[erk_c,erk_b,erk_A]=getButcherTable();         % Get Butcher table you prefer.

n_erk = length(erk_b);                         % n_erk is the number of stages
q = length(y);                                 % q is system size
k=zeros(q,n_erk);                              % q x n_erk storage for function evaluations
for n = 1:nsteps
    for i = 1:n_erk
        t_erk = t0 + h*(n-1+erk_c(i));         % stage time
        y_erk = y;
        for j=1:i-1
            if erk_A(i,j) ~= 0.0
               y_erk = y_erk + erk_A(i,j)*h*k(j);
            end
        end
        k(i) = f(t_erk,y_erk);                 % get k_i
    end
    for i = 1:n_erk                            % stage expansion
        y = y + h*erk_b(i)*k(i);
    end
    t = t0 + n*h;
end
```

This function uses the tableau

```
%
%  Hardcoded Butcher table for 3/8 method as an example. Here erk stands for
%  "explicit Runge-Kutta". Can switch out this table for another if preferred.
%  function [erk_c,erk_b,erk_A] = getButcherTable()
   function [erk_c,erk_b,erk_A] = getButcherTable()
erk_c = [  0; 1/3; 2/3;   1];
erk_b = [1/8; 3/8; 3/8; 1/8];
erk_A = [  0    0    0    0 ;
          1/3    0    0    0 ;
         -1/3    1    0    0 ;
            1   -1    1    0 ];
```

which can easily switch out to use a different ERK method.

Embedded methods, error control and adaptivity

Often we wish to solve an IVP numerically, with the error in the computed solution ensured lower than a chosen tolerance. One strategy is to solve the IVP at least twice, with time-step sizes h and $h/2$, and then compare the two computed numerical solutions at the final time $t_F = b$. If they agree to within the tolerance, then we accept the computed solution (either $y_{2m}^{h/2}$ or y_m^h, where $h = (b-a)/m$). If they do not agree, then we compute $y_{4m}^{h/4}$, comparing it with $y_{2m}^{h/2}$, etc. While straightforward, this procedure is expensive. We seek a way control the time-step size, and so the error, as the numerical integration unfolds. Certain Runge-Kutta schemes have *embedded methods* which allow for this possibility. We consider a workhorse example: the Runge-Kutta-Fehlberg (RKF45) method.

Here is the Butcher table for the RKF45 method.

$$
\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b} \\ & \mathbf{b}' \end{array} =
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0 \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & 0 & 0 & 0 \\
1 & \frac{439}{216} & -8 & -\frac{3680}{513} & -\frac{845}{4104} & 0 & 0 \\
\frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & 0 \\
\hline
 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0 \\
 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55}
\end{array}
$$

Notice that this Butcher table lists two stage-expansion vectors, $\mathbf{b}$ and $\mathbf{b}'$. As before, the stage times $\mathbf{c}$ and Runge-Kutta matrix A determine the quantities

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f\Big(t_n + \tfrac{1}{4}h, y_n + \tfrac{1}{4}hk_1\Big) \\
k_3 &= f\Big(t_n + \tfrac{3}{8}h, y_n + \tfrac{3}{32}hk_1 + \tfrac{9}{32}hk_2 \\
k_4 &= f\Big(t_n + \tfrac{12}{13}h, y_n + \tfrac{1932}{2197}hk_1 - \tfrac{7200}{2197}hk_2 + \tfrac{7296}{2197}hk_3 \\
k_5 &= f\Big(t_n + h, y_n + \tfrac{439}{216}hk_1 - 8hk_2 - \tfrac{3680}{513}hk_3 - \tfrac{845}{4104}hk_4 \\
k_6 &= f\Big(t_n + \tfrac{1}{2}h, y_n - \tfrac{8}{27}hk_1 + 2hk_2 - \tfrac{3544}{2565}hk_3 + \tfrac{1859}{4104}hk_4 - \tfrac{11}{40}hk_5\Big).
\end{aligned}
$$

Clearly, this is an $s = 6$ stage method. However, now we have two choices for updating the solution. The first is a fourth-order method

$$y_{k+1} = y_k + h\Big(\tfrac{25}{216}k_1 + \tfrac{1408}{2565}k_3 + \tfrac{2197}{4104} >$$

determined by the stage-expansion vector $\mathbf{b}$ in the table. The second is a fifth-order method

$$z_{k+1} = y_k + h\Big(\tfrac{16}{135}k_1 + \tfrac{6656}{12285}k_3 + \tfrac{28561}{56430}k_4 - \tfrac{9}{50}k_5 + \tfrac{2}{55}k_6\Big)$$

determined by the stage-expansion vector $\mathbf{b}'$ in the table. One might this of RKF45 as a 6-stage, order-5 method, with the 4-order method *embedded* into it. You might also find the 4-order, 5-order methods together described as an *embedded pair*. Regardless, the key point is that both the fourth and fifth order expansion use the same function evaluations: $k_1, k_2, k_3, k_4, k_5, k_6$. Therefore, both y_{k+1} and z_{k+1} can be computed, and then compared to see if the taken time-step (of size h) is "good". If not, then we throw out the time step, make h *smaller*, and try again. If yes, then we accept y_{k+1} (or z_{k+1}) as the numerical solution at time t_{k+1} and continue (in this case we might also make h *larger* for the next time step).

One possible choice for dynamical adjustment of the step-size h that has been used in practice is the following. Whether or not the time step is accepted or rejected, the time-step size h is readjusted $h \to sh$ to an *optimal step size* sh, where the scalar adjustment is

$$s = \left(\frac{\texttt{tol}\cdot h}{2|z_{k+1} - y_{k+1}|}\right)^{1/4} \simeq 0.84\left(\frac{\texttt{tol}\cdot h}{|z_{k+1} - y_{k+1}|}\right)^{1/4}.$$

Notice that if the error comparison $|z_{k+1} - y_{k+1}|$ is small, then s will be large, and vice versa. The variable `tol` is a user chosen tolerance.