Chapter 9

The Initial Value Problem

§9.1 Basic Concepts

§9.2 The Runge-Kutta Methods

§9.3 The Adams Methods

The goal in the *initial value problem* (IVP) is to find a function $y(t)$ given its value at some initial time t_0 and a recipe $f(t,y)$ for its slope:

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0.$$

In applications we may want to plot an approximation to $y(t)$ over a designated interval of interest $[t_0, t_{max}]$ in an effort to discover qualitative properties of the solution. Or we may require a highly accurate estimate of $y(t)$ at some single, prescribed value $t = T$.

The methods we develop produce a sequence of solution snapshots $(t_1, y_1), (t_2, y_2), \ldots$ that are regarded as approximations to $(t_1, y(t_1), (t_2, y(t_2))$, etc. All we have at our disposal is the "slope function" $f(t,y)$, best thought of as a Matlab function `f(t,y)`, that can be called whenever we need information about where $y(t)$ is "headed." IVP solvers differ in how they use the slope function.

In §9.1 we use the Euler methods to introduce the basic ideas associated with approximate IVP solving: discretization, local error, global error, stability, etc. In practice the IVP usually involves a vector of unknown functions, and the treatment of such problems is also covered in §9.1. In this setting the given slope function $f(t,y)$ is a vector of scalar slope functions, and its evaluation tells us how each component in the unknown $y(t)$ vector is changing with t.

The Runge-Kutta and Adams methods are then presented in §9.2 and §9.3 together with the built-in Matlab IVP solvers `ode23` and `ode45`. We also discuss stepsize control, a topic of great practical importance and another occasion to show off the role of calculus-based heuristics in scientific computing.

Quality software for the IVP is very complex. Years of research and development stand behind codes like `ode23` and `ode45`. The implementations that we develop in this chapter are designed to build intuition and, if anything, are just the first step in the long journey from textbook formula to production software.

9.1 Basic Concepts

A "family" of functions generally satisfies a differential equation of the form $y'(t) = f(t,y)$. The initial condition $y(t_0) = y_0$ singles out one of these family members for the solution to the IVP. For example, functions of the form $y(t) = ce^{-5t}$ satisfy $y'(t) = -5y(t)$. If we stipulate that $y(0) = 1$, then $y(t) = e^{-5t}$ is the unique solution to the IVP. (See Figure 9.1.) Our goal is to produce a sequence of points (t_i, y_i) that reasonably track the solution curve as time evolves. The Euler methods that we develop in this section organize this tracking process around a linear model.

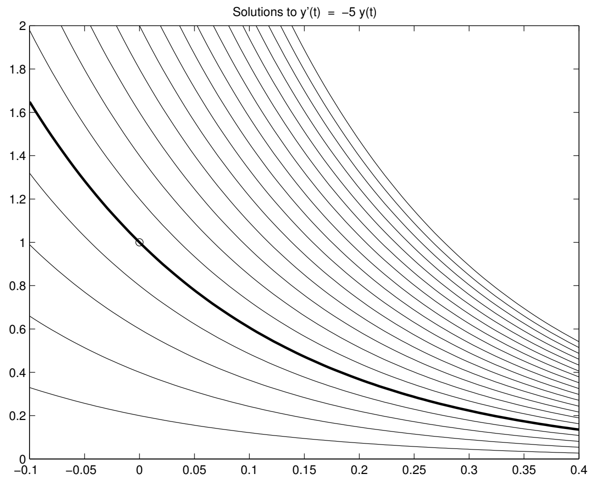

FIGURE 9.1 *Solution curves*

9.1.1 Derivation of the Euler Method

From the initial condition, we know that (t_0, y_0) is on the solution curve. At this point the slope of the solution is computable via the function f:

$$f_0 = f(t_0, y_0).$$

To estimate $y(t)$ at some future time $t_1 = t_0 + h_0$ we consider the following Taylor expansion:

$$y(t_0 + h_0) \approx y(t_0) + h_0 y'(t_0) = y_0 + h_0 f(t_0, y_0).$$

This suggests that we use

$$y_1 = y_0 + h_0 f(t_0, y_0)$$

as our approximation to the solution at time t_1. The parameter $h_0 > 0$ is the *step*, and it can be said that with the production of y_1 we have "integrated the IVP forward" to $t = t_1$.

With $y_1 \approx y(t_1)$ in hand, we try to push our knowledge of the solution one step further into the future. Let h_1 be the next step. A Taylor expansion about $t = t_1$ says that

$$y(t_1 + h_1) \approx y(t_1) + h_1 y'(t_1) = y(t_1) + h_1 f(t_1, y(t_1)).$$

Note that in this case the right-hand side is not computable because we do not know the exact solution at $t = t_1$. However, if we are willing to use the approximations

$$y_1 \approx y(t_1)$$

and

$$f_1 = f(t_1, y_1) \approx f(t_1, y(t_1)),$$

then at time $t_2 = t_1 + h_1$ we have

$$y(t_2) \approx y_2 = y_1 + h_1 f_1.$$

The pattern is now clear. At each step we evaluate f at the current approximate solution point (t_n, y_n) and then use that slope information to get y_{n+1}. The key equation is

$$y_{n+1} = y_n + h_n f(t_n, y_n),$$

and its repeated application defines the *Euler method*:

```
$n = 0$
Repeat:
    $f_n = f(t_n, y_n)$
    Determine the step $h_n > 0$ and set $t_{n+1} = t_n + h_n$.
    $y_{n+1} = y_n + h_n f_n$.
    $n = n + 1$
```

The script file `ShowEuler` solicits the time steps interactively and applies the Euler method to the problem $y' = -5y,\ y(0) = 1$. (See Figure 9.2.) The determination of the step size is crucial.

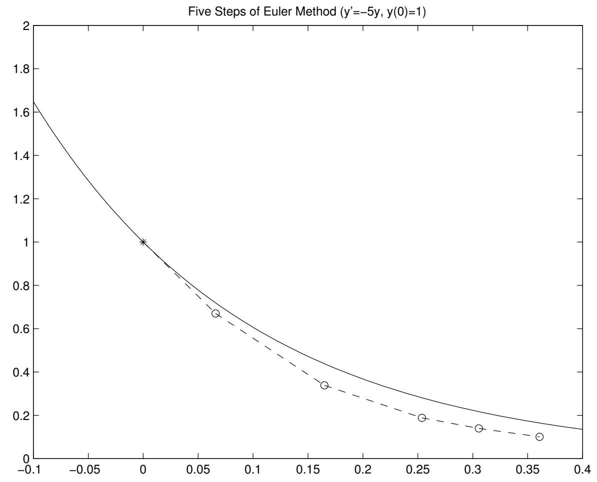

FIGURE 9.2 *Five steps of Euler's method*

Our intuition says that we can control error by choosing h_n appropriately. Accuracy should increase with shorter steps. On the other hand, shorter steps mean more f-evaluations as we integrate across the interval of interest. As in the quadrature problem and the nonlinear equation-solving problem, the number of f-evaluations usually determines execution time, and the efficiency analysis of any IVP method must include a tabulation of this statistic. The basic game to be played is to get the required snapshots of $y(t)$ with sufficient accuracy, evaluating $f(t, y)$ as infrequently as possible. To see what we are up against, we need to understand how the errors in the local model compound as we integrate across the time interval of interest.

9.1.2 Local Error, Global Error, and Stability

Assume in the Euler method that y_{n-1} is exact and let $h = h_{n-1}$. By subtracting $y_n = y_{n-1} + hf_{n-1}$ from the Taylor expansion

$$y(t_n) = y_{n-1} + hy'(t_{n-1}) + \frac{h^2}{2}y^{(2)}(\eta), \qquad \eta \in [t_{n-1}, t_n],$$

we find that

$$y(t_n) - y_n = \frac{h^2}{2}y^{(2)}(\eta).$$

This is called the *local truncation error* (LTE) In general, the LTE for an IVP method is the error that results when a single step is performed with exact "input data." It is a key attribute of any IVP solver, and the *order* of the method is used to designate its form. A method has order k if its LTE goes to zero like h^{k+1}. Thus, the Euler method has order 1. The error in an individual Euler step depends on the square of the step and the behavior of the second derivative. Higher-order methods are pursued in the next two sections.

A good way to visualize the LTE is to recognize that at each step, (t_n, y_n) sits on some solution curve $y_n(t)$ that satisfies the differential equation $y'(t) = f(t, y(t))$. With each step we jump to a new solution curve, and the size of the jump is the LTE. (See Figure 9.3.)

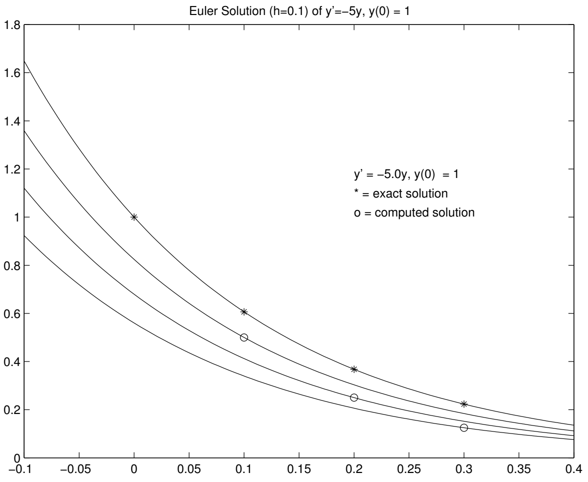

FIGURE 9.3 *Jumping trajectories*

Distinct from the local truncation error is the *global error.* The global error g_n is the actual difference between the $t = t_n$ solution y_n produced by the IVP solver and the *true* IVP solution $y(t_n)$:

$$g_n = y(t_n) - y_n.$$

As we have mentioned, the local truncation error in getting y_n is defined by

$$LTE_n = y_{n-1}(t_n) - y_n,$$

where $y_{n-1}(t)$ satisfies the IVP

$$y'(t) = f(t, y(t)), \qquad y(t_{n-1}) = y_{n-1}.$$

LTE is tractable analytically and, as we shall see, it can be estimated in practice. However, in applications it is the global error that is usually of interest. It turns out that it is possible to control global error by controlling the individual LTEs if the underlying IVP is *stable.* We discuss this after we prove the following result.

Theorem 9 *consec Assume that, for $n = 0{:}N$, a function $y_n(t)$ exists that solves the IVP*

$$y'(t) = f(t, y(t)), \qquad y(t_n) = y_n,$$

where $(t_0, y_0), \ldots, (t_N, y_N)$ are given and $t_0 < t_1 < \cdots < t_N$. Define the global error by

$$g_n \;=\; y_0(t_n) - y_n$$

and the local truncation error by

$$LTE_n \;=\; y_{n-1}(t_n) - y_n.$$

If

$$f_y \;=\; \frac{\partial f(t, y)}{\partial y} \leq 0$$

for all $t \in [t_0, t_N]$ and none of the trajectories

$$\{(t, y_n(t)) : t_0 \leq t \leq t_N\}, \qquad n = 0{:}N$$

intersect, then for $n = 1{:}N$

$$|g_n| \;\le\; \sum_{k=1}^{n} |LTE_k|.$$

Proof If $y_0(t_n) > y_{n-1}(t_n)$, then because f_y is negative we have

$$\int_{t_{n-1}}^{t_n} \left(f(t, y_0(t)) - f(t, y_{n-1}(t)\right) dt < 0.$$

It follows that

$$\begin{aligned} 0 \;<\; y_0(t_n) - y_{n-1}(t_n) \;&=\; (y_0(t_{n-1}) - y_{n-1}(t_{n-1})) + \int_{t_{n-1}}^{t_n} \left(f(t, y_0(t)) - f(t, y_{n-1}(t)\right) dt \\ &<\; (y_0(t_{n-1}) - y_{n-1}(t_{n-1}))\,, \end{aligned}$$

and so

$$|y_0(t_n) - y_{n-1}(t_n)| \;\le\; |y_0(t_{n-1}) - y_{n-1}(t_{n-1})|. \tag{9.1}$$

Likewise, if $y_0(t_n) < y_{n-1}(t_n)$, then

$$\int_{t_{n-1}}^{t_n} \left(f(t, y_{n-1}(t)) - f(t, y_0(t)\right) dt < 0,$$

and so

$$\begin{aligned} 0 \;<\; y_{n-1}(t_n) - y_0(t_n) \;&=\; (y_{n-1}(t_{n-1}) - y_0(t_{n-1})) + \int_{t_{n-1}}^{t_n} \left(f(t, y_{n-1}(t)) - f(t, y_0(t)\right) dt \\ &<\; y_{n-1}(t_{n-1}) - y_0(t_{n-1}). \end{aligned}$$

Thus, in either case (9.1) holds and so

$$\begin{aligned} |g_n| \;&=\; |y_0(t_n) - y_n| \\ &\le\; |y_0(t_n) - y_{n-1}(t_n)| \;+\; |y_{n-1}(t_n) - y_n| \\ &<\; |y_0(t_{n-1}) - y_{n-1}(t_{n-1})| \;+\; |y_{n-1}(t_n) - y_n| \\ &=\; |g_{n-1}| \;+\; |LTE_n|. \end{aligned}$$

The theorem follows by induction since $g_1 = LTE_1$. $\square$

The theorem essentially says that if $\partial f/\partial y$ is negative across the interval of interest, then global error at $t = t_n$ is less than the sum of the local errors made by the IVP solver in reaching t_n. The sign of this partial derivative is tied up with the stability of the IVP. Roughly speaking, if small changes in the initial value induce correspondingly small changes in the IVP solution, then we say that the IVP is *stable*. The concept is much more involved than the condition/stability issues that we talked about in connection with the $Ax = b$ problem. The mathematics is deep and interesting but beyond what we can do here.

So instead we look at the model problem $y'(t) = ay(t), y(0) = c$ and deduce some of the key ideas. In this example, $\partial f/\partial y \;=\; a$ and so Theorem 9 applies if $a < 0$. We know that the solution $y(t) = ce^{at}$ decays if and only if a is negative. If $\tilde{y}(t)$ solves the same differential equation with initial value $y(0) = \tilde{c}$, then

$$|\tilde{y}(t) - y(t)| \;=\; |\tilde{c} - c|e^{at},$$

showing how "earlier error" is damped out as t increases.

To illustrate how global error might be controlled in practice, consider the problem of computing $y(t_{max})$ to within a tolerance *tol*, where $y(t)$ solves a stable IVP $y'(t) = f(t, y(t))$, $y(t_0) = y_0$. Assume that a fixed-step Euler method is to be used and that we have a bound M_2 for $|y^{(2)}(t)|$ on the interval $[t_0, t_{max}]$. If $h = (t_{max} - t_0)/N$ is the step size, then from what we know about the local truncation error of the method,

$$|LTE_n| \le M_2 \frac{h^2}{2}.$$

Assuming that Theorem 9 applies,

$$|y(t_{max}) - y_N| \;\le\; \sum_{n=1}^{N} |LTE_n| \;=\; M_2 N \frac{h^2}{2} \;=\; \frac{t_{max} - t_0}{2} M_2 h.$$

Thus, to make this upper bound less than a prescribed $tol > 0$, we merely set N to be the smallest integer that satisfies

$$\frac{(t_{max} - t_0)^2}{2N} M_2 \le tol.$$

Here is an implementation of the overall process:

```
  function [tvals,yvals] = FixedEuler(f,y0,t0,tmax,M2,tol)
% Fixed step Euler method.
%
% f is a handle that references a function of the form f(t,y).
% M2 a bound on the second derivative of the solution to
%                y' = f(t,y),   y(t0) = y0
% on the interval [t0,tmax].

% Determine positive n so that if tvals = linspace(t0,tmax,n), then
% y(i) is within tol of the true solution y(tvals(i)) for i=1:n.
n = ceil(((tmax-t0)^2*M2)/(2*tol))+1;
h = (tmax-t0)/(n-1);
yvals = zeros(n,1);
tvals = linspace(t0,tmax,n)';
yvals(1) = y0;
for k=1:n-1
   fval = f(tvals(k),yvals(k));
   yvals(k+1) = yvals(k)+h*fval;
end
```

Figure 9.4 shows the error when this solution framework is applied to the model problem $y' = -y$ across the interval $[0, 5]$. The trouble with this approach to global error control is that (1) we rarely have good bound information about $|y^{(2)}|$ and (2) it would be better to determine h adaptively so that longer step sizes can be taken in regions where the solution is smooth. This matter is pursued in §9.3.5.

Rounding errors are also an issue in IVP solving, especially when lots of very short steps are taken. In Figure 9.5 we plot the errors sustained when we solve $y' = -y,\; y(0) = 1$ across $[0, 1]$ with Euler's method in a three-digit floating point environment. The results for steps h =1/140, 1/160, and 1/180 are reported. Note that the error gets worse as h gets smaller because the step sizes are in the neighborhood of unit roundoff. However, for the kind of problems that we are looking at, it is the discretization errors that dominate the discussion of accuracy.

Another issue that colors the performance of an IVP solver is the stability of *the method* itself. This is quite distinct from the notion of problem stability discussed earlier. It is possible for a method with a particular h to be unstable when it is applied to a stable IVP. For example, if we apply the Euler method to $y'(t) = -10y(t)$, then the iteration takes the form

$$y_{n+1} = (1 - 10h)y_n.$$

To ensure that the errors are not magnified as the iteration progresses, we must insist that

$$|1 - 10h| < 1$$

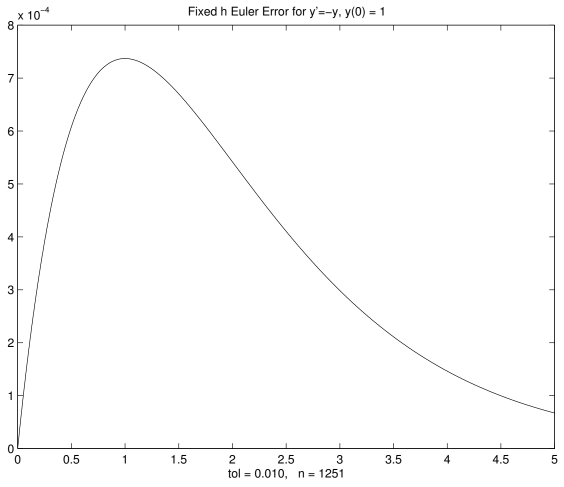

FIGURE 9.4 *Error in fixed-step Euler*

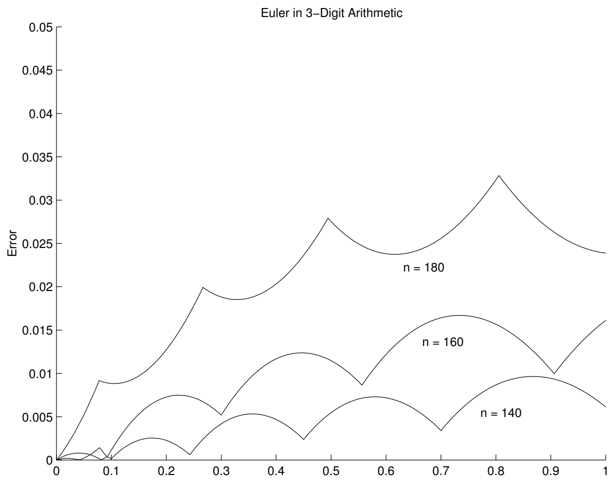

FIGURE 9.5 *Roundoff error in fixed-step Euler*

(i.e., $h < 1/5$). For all h that satisfy this criterion, the method is *stable*. If $h > 1/5$, then any error δ in the initial condition will result in a $(1 - 10h)^n\delta$ contamination of the nth iterate. With this kind of error magnification, we say that the method is *unstable*. Different methods have different h restrictions in order to guarantee stability, and sometimes these restrictions force us to choose h much smaller than we would like.

9.1.3 The Backward Euler Method

To clarify this point about method stability, we examine the *backward Euler method.* The (forward) Euler method is derived from a Taylor expansion of the solution $y(t)$ about $t = t_n$. If instead we work with the approximation

$$y(t_{n+1} + h) \approx y(t_{n+1}) + y'(t_{n+1})h = y(t_{n+1}) + f(t_{n+1}, y(t_{n+1}))h$$

and set $h = -h_n = (t_n - t_{n+1})$, then we get

$$y(t_n) \approx y(t_{n+1}) - h_n f(t_{n+1}, y(t_{n+1})).$$

Substituting y_n for $y(t_n)$ and y_{n+1} for $y(t_{n+1})$, we are led to

$$y_{n+1} = y_n + h_n f(t_{n+1}, y_{n+1})$$

and, with repetition, the *backward Euler framework*:

```
$n = 0$
Repeat:
    Determine the step $h_n > 0$.
    $t_{n+1} = t_n + h_n$.
    Let $y_{n+1}$ solve $F(z) = z - h_n f(t_{n+1}, z) - y_n = 0$.
    $n = n + 1$
```

Like the Euler method, the backward Euler method is first order. However, the two techniques differ in a very important aspect. Backward Euler is an *implicit method* because it defines y_{n+1} implicitly. For a simple problem like $y' = ay$ this poses no difficulty:

$$y_{n+1} = y_n + h_n a y_{n+1} = \frac{1}{1 - h_n a} y_n.$$

Observe that if $a < 0$, then the method is stable for *all* choices of positive step size. This should be contrasted with the situation in the Euler setting, where $|1 + ah| < 1$ is required for stability.

Euler's is an example of an *explicit method*, because y_{n+1} is defined explicity in terms of quantities already computed. [e.g., y_n, $f(t_n, y_n)$]. Implicit methods tend to have better stability properties than their explicit counterparts. But there is an implementation penalty to be paid, because y_{n+1} is defined as a zero of a nonlinear function. In backward Euler, y_{n+1} is a zero of $F(z) = z - h_n f(t_{n+1}, z)$. Fortunately, this does not necessarily require the application of the Chapter 8 root finders. A simpler, more effective approach is presented in §9.3.

9.1.4 Systems

We complete the discussion of IVP solving basics with comments about systems of differential equations. In this case the unknown function $y(t)$ is a vector of unknown functions:

$$y(t) = \begin{bmatrix} z_1(t) \\ \vdots \\ z_d(t) \end{bmatrix}.$$

(We name the component functions with a z instead of a y to avoid confusion with earlier notation.) In this case, we are given an initial value for each component function and a recipe for its slope. This recipe generally involves the value of all the component functions:

$$\begin{bmatrix} z_1'(t) \\ \vdots \\ z_d'(t) \end{bmatrix} = \begin{bmatrix} f_1(t, z_1(t), \ldots, z_d(t)) \\ \vdots \\ f_m(t, z_1(t), \ldots, z_d(t)) \end{bmatrix} \qquad \begin{matrix} z_1(t_0) = z_{10} \\ \vdots \\ z_d(t_0) = z_{d0} \end{matrix}.$$

In vector language, $y'(t) = f(t, y(t)), y(t_0) = y_0$, where the y's are now column d-vectors. Here is a $d = 2$ example:

$$\begin{array}{rcl} u'(t) & = & 2u(t) - .01u(t)v(t) \\ v'(t) & = & -v(t) + .01u(t)v(t) \end{array}, \qquad u(0) = u_0,\ v(0) = v_0.$$

It describes the density of rabbit and fox populations in a classical predator-prey model. The rate of change of the rabbit density $u(t)$ and the fox density $v(t)$ depend on the current rabbit/fox densities.

Let's see how the derivation of Euler's method proceeds for a systems problem like this. We start with a pair of time-honored Taylor expansions:

$$\begin{array}{rcccl} u(t_{n+1}) & \approx & u(t_n) + u'(t_n)h_n & = & u(t_n) + h_n(2u(t_n) - .01u(t_n)v(t_n)) \\ v(t_{n+1}) & \approx & v(t_n) + v'(t_n)h_n & = & v(t_n) + h_n(-v(t_n) + .01u(t_n)v(t_n)) \end{array}$$

Here (as usual), $t_{n+1} = t_n + h_n$. With the definitions

$$y_n = \left[\begin{array}{c} u_n \\ v_n \end{array}\right] \approx \left[\begin{array}{c} u(t_n) \\ v(t_n) \end{array}\right] = y(t_n)$$

and

$$f_n = f(t_n, y_n) = \left[\begin{array}{c} 2u_n - .01u_nv_n \\ -v_n + .01u_nv_n \end{array}\right] \approx \left[\begin{array}{c} 2u(t_n) - .01u(t_n)v(t_n) \\ -v(t_n) + .01u(t_n)v(t_n) \end{array}\right] = f(t_n, y(t_n)),$$

we obtain he following vector implementation of the Euler method:

$$\left[\begin{array}{c} u_{n+1} \\ v_{n+1} \end{array}\right] = \left[\begin{array}{c} u_n \\ v_n \end{array}\right] + h_n \left[\begin{array}{c} 2u_n - .01u_nv_n \\ -v_n + .01u_nv_n \end{array}\right].$$

In full vector notation, this can be written as

$$y_{n+1} = y_n + h_n f_n,$$

which is exactly the same formula that we developed in the scalar case.

As we go through the next two sections presenting more sophisticated IVP solvers, we shall do so for scalar ($d = 1$) problems, being mindful that all method-defining equations apply at the system level with no modification.

Systems can arise in practice from the conversion of *higher-order* IVPs. In a kth order IVP, we seek a function $y(t)$ that satisfies

$$y^{(k)}(t) = f(t, y(t), y^{(1)}(t), \ldots, y^{(k-1)}(t)) \qquad \text{where} \qquad \left\{\begin{array}{lcl} y(t_0) & = & y_0 \\ y^{(1)}(t_0) & = & y_0^{(1)} \\ & \vdots & \\ y^{(k-1)}(t_0) & = & y_0^{(k-1)} \end{array}\right.$$

and $y_0, y_0^{(1)}, \ldots, y_0^{(k-1)}$ are given initial values. Higher order IVPs can be solved through conversion to a system of first-order IVPs. For example, to solve

$$v''(t) = 2v(t) + v'(t)\sin(t), \qquad v(0) = \alpha,\ v'(0) = \beta,$$

we define $z_1(t) = v(t)$ and $z_2(t) = v'(t)$. The problem then transforms to

$$\begin{array}{l} z_1'(t) = z_2(t) \\ z_2'(t) = 2z_1(t) + z_2(t)\sin(t) \end{array}, \qquad z_1(0) = \alpha,\ z_2(0) = \beta.$$

Problems

P9.1.1 Produce a plot of the solution to

$$y'(t) = -ty + \frac{1}{y^2}, \qquad y(1) = 1$$

across the interval $[1, 2]$. Use the Euler method.

P9.1.2 Compute an approximation to $y(1)$ where

$$\begin{array}{lcl} x''(t) & = & (3 - \sin(t))x'(t) + x(t)/(1 + [y(t)]^2), \\ y'(t) & = & -\cos(t)y(t) - x'(t)/(1 + t^2), \end{array}$$

$x(0) = 3$, $x'(0) = -1$, and $y(0) = 4$. Use the forward Euler method with fixed step determined so that three significant digits of accuracy are obtained. Hint: Define $z(t) = x'(t)$ and rewrite the recipe for x'' as a function of x, y, and z. This yields a $d = 3$ system.

P9.1.3 Plot the solutions to

$$y'(t) = \left[\begin{array}{cc} -1 & 4 \\ -4 & -1 \end{array}\right] y(t), \qquad y(0) = \left[\begin{array}{c} 2 \\ -1 \end{array}\right]$$

across the interval $[0, 3]$.

P9.1.4 Consider the initial value problem

$$Ay'(t) = By(t), \qquad y(0) = y_0,$$

where A and B are given n-by-n matrices with A nonsingular. For fixed step size h, explain how the backwards Euler method can be used to compute approximate solutions at $t = kh$, $k = 1{:}100$.

9.2 The Runge-Kutta Methods

In an Euler step, we "extrapolate into the future" with only a single sampling of the slope function $f(t, y)$. The method has order 1 because its LTE goes to zero as h^2. Just as we moved beyond the trapezoidal rule in Chapter 4, so we must now move beyond the Euler framework with more involved models of the slope function. In the Runge-Kutta framework, we sample f at several judiciously chosen spots and use the information to obtain y_{n+1} from y_n with the highest possible order of accuracy.

9.2.1 Derivation

The Euler methods evaluate f once per step and have order 1. Let's sample f twice per step and see if we can obtain a second-order method. We arrange it so that the second evaluation depends on the first:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \alpha h, y_n + \beta k_1) \\ y_{n+1} &= y_n + ak_1 + bk_2 \,. \end{aligned}$$

Our goal is to choose the parameters α, β, a, and b so that the LTE is $O(h^3)$. From the Taylor series we have

$$y(t_{n+1}) \;=\; y(t_n) + y^{(1)}(t_n)h + y^{(2)}(t_n)\frac{h^2}{2} + O(h^3).$$

Since

$$\begin{aligned} y^{(1)}(t_n) &= f \\ y^{(2)}(t_n) &= f_t + f_y f \end{aligned}$$

where

$$\begin{aligned} f &= f(t_n, y_n) \\ f_t &= \frac{\partial f(t_n, y_n)}{\partial t} \\ f_y &= \frac{\partial f(t_n, y_n)}{\partial y}, \end{aligned}$$

it follows that

$$y(t_{n+1}) \;=\; y(t_n) + fh \;+\; (f_t + f_y f)\,\frac{h^2}{2} \;+\; O(h^3). \qquad (9.2)$$

On the other hand,

$$k_2 = hf(t_n + \alpha h, y_n + \beta k_1) \;=\; h\left(f + \alpha h f_t + \beta k_1 f_y + O(h^2)\right)$$

and so

$$y_{n+1} = y_n + ak_1 + bk_2 = y_n + (a+b)\,fh + b\,(\alpha f_t + \beta f f_y)\,h^2 + O(h^3). \qquad (9.3)$$

For the LTE to be $O(h^3)$, the equation

$$y(t_{n+1}) - y_{n+1} = O(h^3)$$

must hold. To accomplish this, we compare terms in (9.2) and (9.3) and require

$$\begin{aligned} a + b &= 1 \\ 2b\alpha &= 1 \\ 2b\beta &= 1\,. \end{aligned}$$

There are an infinite number of solutions to this system, the canonical one being $a = b = 1/2$ and $\alpha = \beta = 1$. With this choice the LTE is $O(h^3)$, and we obtain a second-order Runge-Kutta method:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h, y_n + k_1) \\ y_{n+1} &= y_n + (k_1 + k_2)/2\,. \end{aligned}$$

The actual expression for the LTE is given by

$$\text{LTE}(RK2) = \frac{h^3}{12}(f_{tt} + 2ff_{ty} + f^2 f_{yy} - 2f_t f_y - 2ff_y^2),$$

where the partials on the right are evaluated at some point in $[t_n, t_n + h]$. Notice that two f-evaluations are required per step.

The most famous Runge-Kutta method is the classical fourth order method:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(t_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\,. \end{aligned}$$

This can be derived using the same Taylor expansion technique illustrated previously. It requires four f-evaluations per step.

The function `RKStep` can be used to carry out a Runge-Kutta step of prescribed order. Here is its specification along with an abbreviated portion of the implementation:

```
  function [tnew,ynew,fnew] = RKstep(f,tc,yc,fc,h,k)
% f is a handle that references a function of the form f(t,y)
% where t is a scalar and y is a column d-vector.
% yc is an approximate solution to y'(t) = f(t,y(t)) at t=tc.
% fc = f(tc,yc).
% h is the time step.
% k is the order of the Runge-Kutta method used, 1<=k<=5.
% tnew=tc+h, ynew is an approximate solution at t=tnew, and
% fnew = f(tnew,ynew).

   if k==1
      k1 = h*fc;
      ynew = yc + k1;
   elseif k==2
      k1 = h*fc;
      k2 = h*f(tc+(h),yc+(k1));
      ynew  = yc + (k1 + k2)/2;
   elseif k==3
      k1 = h*fc;
      k2 = h*f(tc+(h/2),yc+(k1/2));
      k3 = h*f(tc+h,yc-k1+2*k2);
      ynew  = yc + (k1 + 4*k2 + k3)/6;
   elseif k==4
        :
   end
   tnew = tc+h;
   fnew = f(tnew,ynew);
```

As can be imagined, symbolic algebra tools are useful in the derivation of such an involved sampling and combination of f-values.

Problems

P9.2.1 The RKF45 method produces both a fourth order estimate and a fifth order estimate using six function evaluations:

$$\begin{aligned}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \tfrac{h}{4}, y_n + \tfrac{1}{4}k_1) \\
k_3 &= hf(t_n + \tfrac{3h}{8}, y_n + \tfrac{3}{32}k_1 + \tfrac{9}{32}k_2) \\
k_4 &= hf(t_n + \tfrac{12h}{13}, y_n + \tfrac{1932}{2197}k_1 - \tfrac{7200}{2197}k_2 + \tfrac{7296}{2197}k_3) \\
k_5 &= hf(t_n + h, y_n + \tfrac{439}{216}k_1 - 8k_2 + \tfrac{3680}{513}k_3 - \tfrac{845}{4104}k_4) \\
k_6 &= hf(t_n + \tfrac{h}{2}, y_n - \tfrac{8}{27}k_1 + 2k_2 - \tfrac{3544}{2565}k_3 + \tfrac{1859}{4104}k_4 - \tfrac{11}{40}k_5) \\
y_{n+1} &= y_n + \tfrac{25}{216}k_1 + \tfrac{1408}{2565}k_3 + \tfrac{2197}{4104}k_4 - \tfrac{1}{5}k_5 \\
z_{n+1} &= y_n + \tfrac{16}{135}k_1 + \tfrac{6656}{12825}k_3 + \tfrac{28561}{56430}k_4 - \tfrac{9}{50}k_5 + \tfrac{2}{55}k_6 \,.
\end{aligned}$$

Write a script that discovers which of y_{n+1} and z_{n+1} is the fourth order estimate and which is the fifth order estimate.

9.2.2 Implementation

Runge-Kutta steps can obviously be repeated, and if we keep the step size fixed, then we obtain the following implementation:

```
  function [tvals,yvals] = FixedRK(f,t0,y0,h,k,n)
% [tvals,yvals] = FixedRK(fname,t0,y0,h,k,n)
% Produces approximate solution to the initial value problem
%
%      y'(t) = f(t,y(t))     y(t0) = y0
%
% using a strategy that is based upon a k-th order Runge-Kutta method. Stepsize
% is fixed. f is a handle that references the function f, t0 is the initial time,
% y0 is the initial condition vector, h is the stepsize, k is the order of
% method (1<=k<=5), and n is the number of steps to be taken,

% tvals(j) = t0 + (j-1)h, j=1:n+1
% yvals(j,:) = approximate solution at t = tvals(j), j=1:n+1

tc = t0; tvals = tc;
yc = y0; yvals = yc';
fc = f(tc,yc);
for j=1:n
   [tc,yc,fc] = RKstep(f,tc,yc,fc,h,k);
   yvals = [yvals; yc'];
   tvals = [tvals tc];
end
```

The function file ShowRK can be used to illustrate the performance of the Runge-Kutta methods on the IVP $y' = -y$, $y(0) = 1$. The results are reported in Figure 9.6. All the derivatives of f are "nice," which means that if we increase the order and keep the step size fixed, then the errors should diminish by a factor of h. Thus for $n = 500$, $h = 1/100$ and we find that the error in the kth order method is about 100^{-k}.

Do not conclude from the example that higher-order methods are necessarily more accurate. If the higher derivatives of the solution are badly behaved, then it may well be the case that a lower-order method gives more accurate results. One must also be mindful of the number of f-evaluations that are required to purchase a given level of accuracy. The situation is analogous to what we found in the quadrature unit. Of course, the best situation is for the IVP software to handle the selection of method and step.

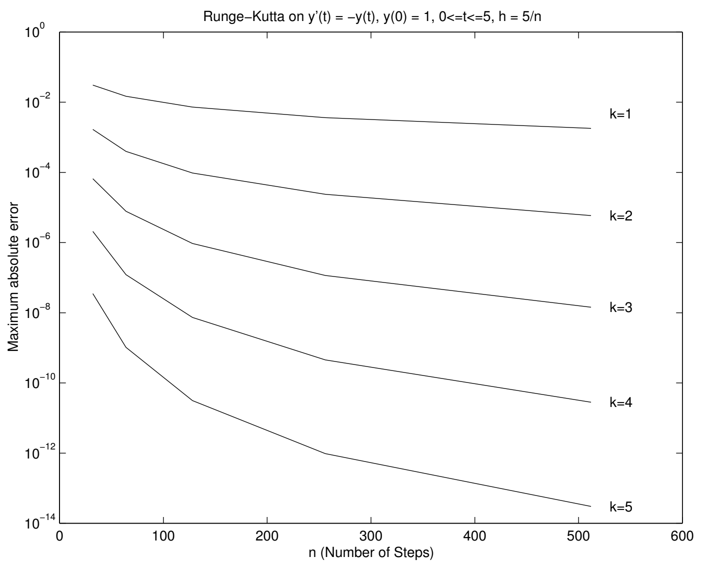

FIGURE 9.6 *Runge-Kutta error*

Problems

P9.2.2 For $k = 1{:}5$, how many f-evaluations does the kth-order Runge-Kutta method require to solve $y'(t) = -y(t),\ y(0) = 1$ with error $\leq 10^{-6}$ across $[0, 1]$?

9.2.3 The MATLAB IVP Solving Tools

MATLAB supplies a number of techniques for solving initial value problems. We start with `ode23`, which is based on a pair of second- and third-order Runge-Kutta methods. With two methods for predicting y_{n+1}, it uses the discrepancy of the predictions to determine heuristically whether the current step size is "safe" with respect to the given tolerances.

Both codes can be used to solve systems, and to illustrate how they are typically used, we apply them to the following initial value problem:

$$\begin{aligned} \ddot{x}(t) &= -\frac{x(t)}{(x(t)^2 + y(t)^2)^{3/2}} \qquad & x(0) = .4 \quad \dot{x}(0) = 0 \\ \ddot{y}(t) &= -\frac{y(t)}{(x(t)^2 + y(t)^2)^{3/2}} \qquad & y(0) = 0 \quad \dot{y}(0) = 2\,. \end{aligned}$$

These are Newton's equations of motion for the two-body problem. As t ranges from 0 to 2π, $(x(t), y(t))$ defines an ellipse.

Both `ode23` and `ode45` require that we put this problem in the standard $y' = f(t, y)$ form. To that end, we define $u_1(t) = x(t)$, $u_2(t) = \dot{x}(t)$, $u_3(t) = y(t)$, $u_4(t) = \dot{y}(t)$. The given IVP problem transforms to

$$\begin{aligned} \dot{u}_1(t) &= u_2(t) & u_1(0) = .4 \\ \dot{u}_2(t) &= -u_1(t)/(u_1(t)^2 + u_3(t)^2)^{3/2} & u_2(0) = 0 \\ \dot{u}_3(t) &= u_4(t) & u_3(0) = 0 \\ \dot{u}_4(t) &= -u_3(t)/(u_1(t)^2 + u_3(t)^2)^{3/2} & u_4(0) = 2\,. \end{aligned}$$

We then write the following function, which returns the derivative of the u vector:

```
  function up = Kepler(t,u)
% up = Kepler(t,u)
% t (time) is a scalar and u is a 4-vector whose components satisfy
%
%           u(1) = x(t)      u(2) = (d/dt)x(t)
%           u(3) = y(t)      u(4) = (d/dt)y(t)
%
% where (x(t),y(t)) are the equations of motion in the 2-body problem.
%
% up is a 4-vector that is the derivative of u at time t.

r3 = (u(1)^2 + u(3)^2)^1.5;
up = [  u(2)     ;...
       -u(1)/r3  ;...
        u(4)     ;...
       -u(3)/r3] ;
```

With this function available, we can call `ode23` and plot various results:

```
tInitial = 0;
tFinal   =  2*pi;
uInitial = [ .4; 0 ; 0 ; 2];
tSpan = [tInitial tFinal];
[t, u] = ode23(@Kepler, tSpan, uInitial);
```

`ode23` requires that we pass the name of the "slope function", the span of integration, and the initial condition vector. The slope function must be of the form `f(t,y)` where `t` is a scalar and `y` is a vector. It must return a column vector. In this call the `tSpan` vector simply specifies the initial and final times. The output produced is a column vector of times `t` and a matrix `u` of solution snapshots. If `n = length(t)` then (a) `t(0) = tInitial`, `t(n) = tFinal`, and `u(k,:)` is an approximation to the solution at time `t(k)`. The time step lengths and (therefore their number) is determined by the default error tolerance: `Reltol` $= 10^{-3}$ and `AbsTol` $= 10^{-6}$. Basically, `ode23` integrates from `tInitial` to `tFinal` "as quick as possible" subject to these two tolerances. We can display the orbit via

```
plot(u(:,1),u(:,3))
```

i.e., by plotting the computed y-values against the computed x-values. (See Figure 9.7.) From the output we display in Figure 9.8 the step lengths with

```
plot(t(2:length(t)),diff(t))
```

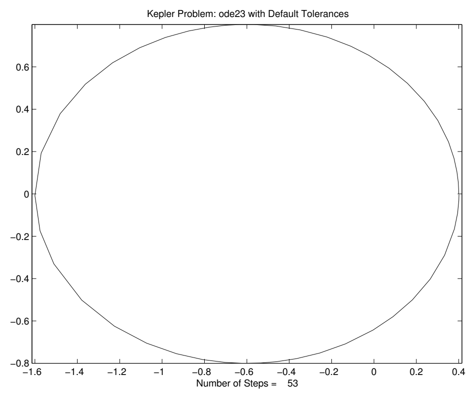

FIGURE 9.7 *Generation of the Orbit via* `ode23`

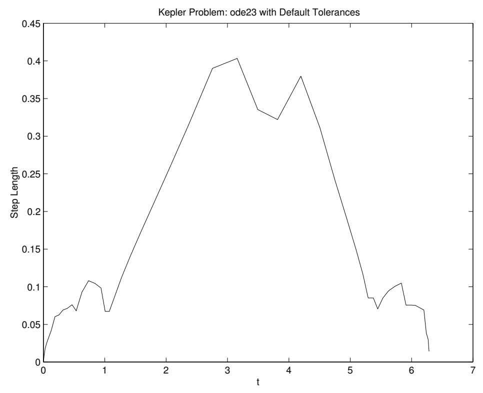

FIGURE 9.8 *Step length with* `ode23` *default tolerances*

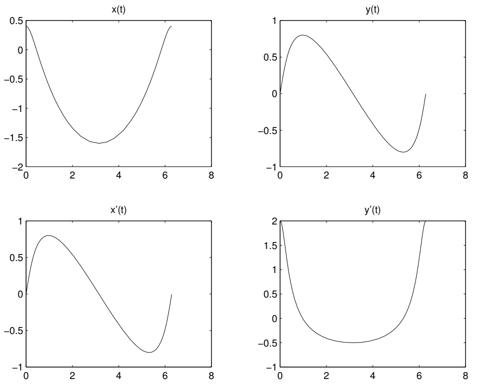

FIGURE 9.9 *Component solutions*

and in Figure 9.9 the component solutions via

```
subplot(2,2,1), plot(t,u(:,1)), title('x(t)')
subplot(2,2,2), plot(t,u(:,2)), title('y(t)')
subplot(2,2,3), plot(t,u(:,3)), title('x''(t)')
subplot(2,2,4), plot(t,u(:,4)), title('y''(t)')
```

The function `ode23` can also be asked to return the solution at specified times. Here is a script that generates 20 solution snapshots and does a spline fit of the output

```
tSpan = linspace(tInitial,tFinal,20);
[t, u] = ode23('Kepler', tSpan, uInitial);
xvals = spline(t,u(:,1),linspace(0,2*pi));
yvals = spline(t,u(:,3),linspace(0,2*pi));
plot(xvals,yvals,u(:,1),u(:,3),'o')
```

(See Figure 9.10.)

Using the function `odeset` it is possible to specify various parameters that are used by `ode23`. For example,

```
tSpan = [tInitial tFinal];
options = odeset('AbsTol',.00000001,'RelTol',.000001,'stats','on');
disp(sprintf('\n Stats for ode23 Call:\n'))
[t, u] = ode23('Kepler', tSpan, uInitial,options);
```

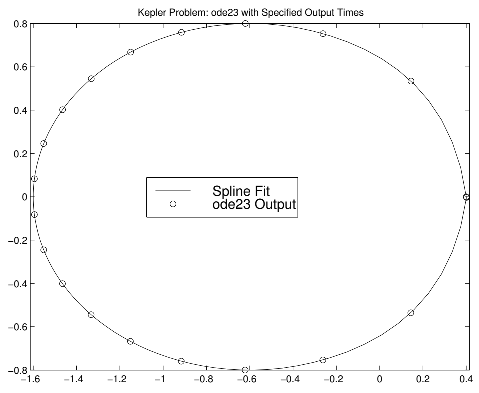

FIGURE 9.10 *Specified output times and spline fit*

overrides the default tolerance for relative error and absolute error and activates the 'stats' option. As expected, the time steps are now shorter as shown in Figure 9.11.

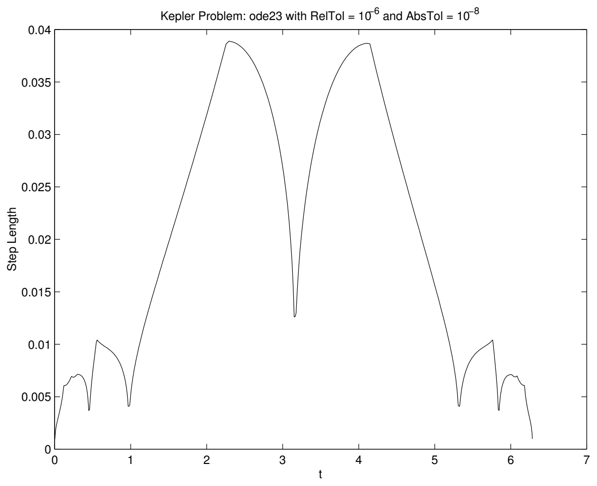

FIGURE 9.11 `ode23` *Timesteps with more stringent tolerances*

The "cost" statistics associated with the call are displayed in the command window:

```
517 successful steps
0 failed attempts
1552 function evaluations
0 partial derivatives
0 LU decompositions
0 solutions of linear systems
```

Sometimes a higher order method can achieve the same accuracy with fewer function evaluations. To illustrate this we apply `ode45` to the same problem:

```
tSpan = [tInitial tFinal];
options = odeset('AbsTol',.00000001,'RelTol',.000001,'stats','on');
disp(sprintf('\n Stats for ode45 Call:\n'))
[t, u] = ode45('Kepler', tSpan, uInitial,options);
```

`ode45` is just like `ode23` except that it uses a mix of 4th and 5th order Runge-Kutta methods. For this problem `ode45` can take decidedly longer time steps in the "high curvature" regions of the orbit. (Compare Figure 9.11 and Figure 9.12.) The output statistics reveal that just 337 function evaluations are required.

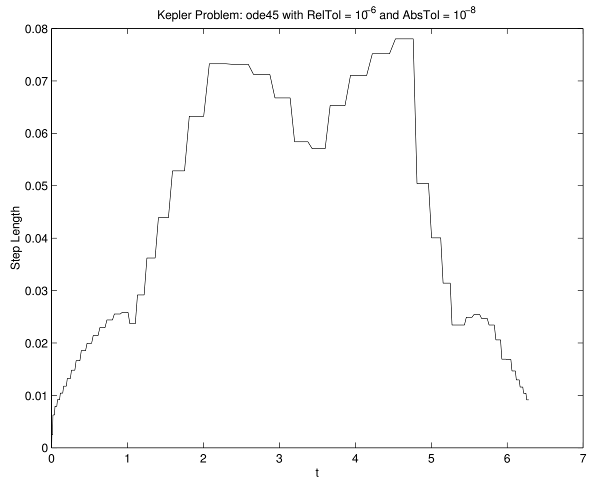

FIGURE 9.12 *Stepsize with ode45*

Problems

P9.2.3 This is about `ode23` vs `ode45`. Suppose it takes T_1 seconds to execute `[t,y] = ode23(@MyF,[0,4],y0)` and T_2 seconds to execute `[t,y] = ode45(@MyF,[0,8],y0)` What factors determine T_2/T_1?

P9.2.4 Our goal is to produce a plot of an approximate solution to the *boundary value problem*

$$y''(t) = f(t, y(t), y'(t)), \qquad y(a) = \alpha,\ y(b) = \beta.$$

Assume the availability of a MATLAB function `f(t,y,yp)`. **(a)** Write a function `g(mu)` that returns an estimate of $y(b)$ where $y(t)$ solves

$$y''(t) = f(t, y(t), y'(t)), \qquad y(a) = \alpha,\ y'(a) = \mu.$$

Use `ode23` and define the function that must be passed to it. **(b)** How could μ_* be computed so that $g(\mu_*) = \beta$? **(c)** Finally, how could a plot of the boundary value problem solution across $[a, b]$ be obtained?

P9.2.5 Consider the following initial value problem

$$A\dot{y} = By + u(t), \qquad y(0) = y_0,$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular, $B \in \mathbb{R}^{n\times n}$, and $u(t) \in \mathbb{R}^n$. Making effective use of `ode45` with default tolerances, write a MATLAB fragment that assigns to `yFinal` an estimate of $y(t_{final})$. Write out completely the function that your script passes to `ode45`. Assume that `A`, `B`, `y0`, and `tfinal` are given and that `u.m` implements $u(t)$.

P9.2.6 Consider the following IVP:

$$\begin{aligned}
\ddot{x}(t) &= 2\dot{y}(t) + x(t) - \frac{\mu_*(x(t)+\mu)}{r_1^3} - \frac{\mu_(x(t)-\mu_*)}{r_2^3}, & x(0) = 1.2 \quad \dot{x}(0) = 0,\\
\ddot{y}(t) &= -2\dot{x}(t) + y(t) - \frac{\mu_* y(t)}{r_1^3} - \frac{\mu y(t)}{r_2^3}, & y(0) = 0 \qquad \dot{y}(0) = -1.0493575,
\end{aligned}$$

where $\mu = 1/82.45$, $\mu_* = 1 - \mu$, and

$$r_1 = \sqrt{((x(t)+\mu)^2 + y(t)^2}$$
$$r_2 = \sqrt{((x(t)-\mu_*)^2 + y(t)^2}$$

It describes the orbit of a spacecraft that starts behind the Moon (located at $(1-\mu, 0)$), swings by the Earth (located at $(-\mu, 0)$), does a large loop, and returns to the vicinity of the Earth before returning to its initial position behind the Moon at time $T_0 = 6.19216933$. Here, $\mu = 1/82.45$.

(a) Apply `ode45` with $t_{initial} = 0$, $t_{final} = T_0$, and $tol = 10^{-6}$. Plot the orbit twice, once with the default "pen" and once with `'.'` so that you can see how the time step varies. **(b)** Using the output from the `ode45` call in part (a), plot the distance of the spacecraft to Earth as a function of time across $[0, T_0]$. Use `spline` to fit the distance "snapshots." To within a mile, how close does the spacecraft get to the Earth's surface? Assume that the Earth is a sphere of radius 4000 miles and that the Earth-Moon separation is 238,000 miles. Use `fmin` with an appropriate spline for the objective function. Note that the IVP is scaled so that one unit of distance is 238,000 miles. **(c)** Repeat Part **(a)** with `ode23`. **(d)** Apply `ode45` with $t_{initial} = 0$, $t_{final} = 2T_0$, and $tol = 10^{-6}$, but change $\dot{y}(0)$ to and $\dot{y}(0) = -.8$. Plot the orbit. For a little more insight into what happens, repeat with $t_{final} = 8 * T_0$. **(e)** To the nearest minute, compute how long the spacecraft is hidden to an observer on earth as it swings behind the Moon during its orbit. Assume that the observer is at $(-\mu, 0)$ and that the Moon has diameter 2160 miles. Make intelligent use of `fzero`. **(f)** Find t_* in the interval $[0, T_0/2]$ so that at time t_*, the spacecraft is equidistant from the Moon and the Earth.

9.3 The Adams Methods

From the fundamental theorem of calculus, we have

$$y(t_{n+1}) \;=\; y(t_n) + \int_{t_n}^{t_{n+1}} y'(t)dt,$$

and so

$$y(t_{n+1}) \;=\; y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t))dt.$$

The Adams methods are based on the idea of replacing the integrand with a polynomial that interpolates $f(t, y)$ at selected solution points (t_j, y_j). The kth order Adams-Bashforth method is explicit and uses the current point (t_n, y_n) and $k-1$ "historical" points. The kth order Adams-Moulton method is implicit and uses the future point (t_{n+1}, y_{n+1}), the current point, and $k-2$ historical points. The implementation and properties of these two IVP solution frameworks are presented in this section.

9.3.1 Derivation of the Adams-Bashforth Methods

In the kth order *Adams-Bashforth* (AB) method, we set

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p_{k-1}(t)dt, \tag{9.4}$$

where $p_{k-1}(t)$ interpolates $f(t, y)$ at (t_{n-j}, y_{n-j}), $j = 0{:}k-1$. We are concerned with the first five members of this family:

Order	Interpolant	AB Interpolation Points
1st	constant	(t_n, f_n)
2nd	linear	$(t_n, f_n), (t_{n-1}, f_{n-1})$
3rd	quadratic	$(t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2})$
4th	cubic	$(t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2}), (t_{n-3}, f_{n-3})$
5th	quartic	$(t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2}), (t_{n-3}, f_{n-3}), (t_{n-3}, f_{n-3})$

If $k = 1$, then the one-point Newton-Cotes rule is applied and we get

$$y_{n+1} = y_n + h_n f(t_n, y_n) \qquad h_n = t_{n+1} - t_n.$$

Thus the first-order AB method is the Euler method.

In the second-order Adams-Bashforth method, we set

$$p_{k-1}(t) = f_{n-1} + \frac{f_n - f_{n-1}}{h_{n-1}}(t - t_{n-1})$$

in (9.4). This is the linear interpolant of (t_{n-1}, f_{n-1}) and (t_n, f_n), and we obtain

$$\begin{aligned} \int_{t_n}^{t_{n+1}} f(t, y(t))dt \;&\approx\; \int_{t_n}^{t_{n+1}} \left(f_{n-1} + \frac{f_n - f_{n-1}}{h_{n-1}}(t - t_{n-1}) \right) dt \\ &=\; \frac{h_n}{2}\left(\frac{h_n + 2h_{n-1}}{h_{n-1}} f_n - \frac{h_n}{h_{n-1}} f_{n-1} \right). \end{aligned}$$

If $h_n = h_{n-1} = h$, then from (9.4)

$$y_{n+1} = y_n + \frac{h}{2}\left(3f_n - f_{n-1}\right).$$

The derivation of higher-order AB methods is analogous. A table of the first five Adams-Bashforth methods along with their respective local truncation errors is given in Figure 9.13. The derivation of the

Order	Step	LTE
1	$y_{n+1} = y_n + hf_n$	$\frac{h^2}{2}y^{(2)}(\eta)$
2	$y_{n+1} = y_n + \frac{h}{2}\left(3f_n - f_{n-1}\right)$	$\frac{5h^3}{12}y^{(3)}(\eta)$
3	$y_{n+1} = y_n + \frac{h}{12}\left(23f_n - 16f_{n-1} + 5f_{n-2}\right)$	$\frac{3h^4}{8}y^{(4)}(\eta)$
4	$y_{n+1} = y_n + \frac{h}{24}\left(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}\right)$	$\frac{251h^5}{720}y^{(5)}(\eta)$
5	$y_{n+1} = y_n + \frac{h}{720}\left(1901f_n - 2774f_{n-1} + 2616f_{n-2} - 1274f_{n-3} + 251f_{n-4}\right)$	$\frac{95h^6}{288}y^{(6)}(\eta)$

FIGURE 9.13 *Adams-Bashforth family*

LTEs for the AB methods is a straightforward computation that involves the Newton-Cotes error:

$$y(t_{n+1}) - y_n \;=\; \int_{t_n}^{t_{n+1}} \left(f(t, y_n(t)) - p_{k-1}(t)\right)dt.$$

9.3.2 Implementation

To facilitate experimentation with the AB method, here is a function that can carry out any of the methods specified in Figure 9.13:

```
  function [tnew,ynew,fnew] = ABstep(f,tc,yc,fvals,h,k)
% f is a handle that references a function of the form f(t,y)
%   where t is a scalar and y is a column d-vector.
%
% yc is an approximate solution to y'(t) = f(t,y(t)) at t=tc.
%
% fvals is an d-by-k matrix where fvals(:,i) is an approximation
%   to f(t,y) at t = tc +(1-i)h, i=1:k
%
% h    = the time step.
% k    = the order of the AB method used, 1<=k<=5.
% tnew = tc+h.
% ynew = an approximate solution at t=tnew.
% fnew = f(tnew,ynew).

if k==1,     ynew = yc + h*fvals;
elseif k==2, ynew = yc + (h/2)*(fvals*[3;-1]);
elseif k==3, ynew = yc + (h/12)*(fvals*[23;-16;5]);
elseif k==4, ynew = yc + (h/24)*(fvals*[55;-59;37;-9]);
elseif k==5, ynew = yc + (h/720)*(fvals*[1901;-2774;2616;-1274;251]);
end
tnew = tc+h;
fnew = f(tnew,ynew);
```

In the systems case, fval is a matrix and ynew is yc plus a matrix-vector product.

Note that k snapshots of $f(t,y)$ are required, and this is why Adams methods are called *multistep* methods. Because of this there is a "start-up" issue with the Adams-Bashforth method: How do we perform the first step when there is no "history"? There are several approaches to this, and care must be taken to ensure that the accuracy of the generated start-up values is consistent with the overall accuracy aims. For a kth order Adams framework we use a kth order Runge-Kutta method, to get $f_j = f(t_j, y_j)$, $j = 1{:}k-1$. See the function ABStart. Using ABStart we are able to formulate a fixed-step Adams-Bashforth solver:

```
  function [tvals,yvals] = FixedAB(f,t0,y0,h,k,n)
% Produces an approximate solution to the initial value problem
% y'(t) = f(t,y(t)), y(t0) = y0 using a strategy that is based upon a k-th order
% Adams-Bashforth method. Stepsize is fixed.
%
% f  = handle that references the function f.
% t0 = initial time.
% y0 = initial condition vector.
% h  = stepsize.
% k  = order of method. (1<=k<=5).
% n  = number of steps to be taken,
%
% tvals(j) = t0 + (j-1)h, j=1:n+1
% yvals(j,:) = approximate solution at t = tvals(j), j=1:n+1

[tvals,yvals,fvals] = ABStart(f,t0,y0,h,k);
tc = tvals(k);
yc = yvals(k,:)';
fc = fvals(:,k);

for j=k:n
   % Take a step and then update.
   [tc,yc,fc] = ABstep(f,tc,yc,fvals,h,k);
   tvals = [tvals tc];
   yvals = [yvals; yc'];
   fvals = [fc fvals(:,1:k-1)];
end
```

If we apply this algorithm to the model problem, $y' = -y, y(0) = 1$. (See Figure 9.14.) Notice that for the kth-order method, the error goes to zero as h^k, where $h = 1/n$.

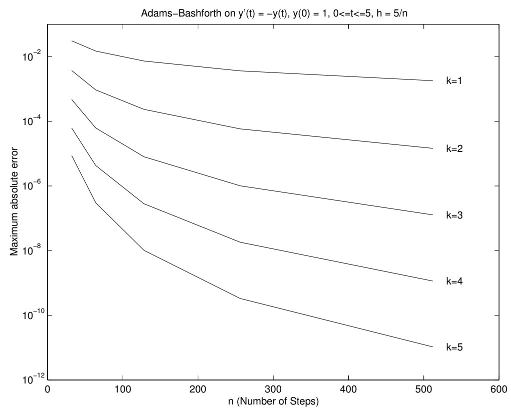

FIGURE 9.14 *kth order Adams-Bashforth error*

9.3.3 The Adams-Moulton Methods

The kth order *Adams-Moulton* (AM) method is just like the kth-order Adams-Bashforth method, but the points at which we interpolate the integrand in

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p_{k-1}(t)dt$$

are "shifted" one time step into the future. In particular, the kth-order Adams-Moulton method uses a degree $k-1$ interpolant of the points (t_{n+1-j}, f_{n+1-j}), $j = 0{:}k-1$:

Order	Interpolant	AM Interpolation Points
1st	constant	(t_{n+1}, f_{n+1})
2nd	linear	$(t_{n+1}, f_{n+1}), (t_n, f_n)$
3rd	quadratic	$(t_{n+1}, f_{n+1}), (t_n, f_n), (t_{n-1}, f_{n-1})$
4th	cubic	$(t_{n+1}, f_{n+1}), (t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2})$
5th	quartic	$(t_{n+1}, f_{n+1}), (t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2}), (t_{n-3}, f_{n-3})$

For example, in the second-order Adams-Moulton method we set

$$p_{k-1}(t) = f_n + \frac{f_{n+1} - f_n}{h_n}(t - t_n),$$

Order	Step	LTE
1	$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$	$-\frac{h^2}{2}y^{(2)}(\eta)$
2	$y_{n+1} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}) + f_n\right)$	$-\frac{h^3}{12}y^{(3)}(\eta)$
3	$y_{n+1} = y_n + \frac{h}{12}\left(5f(t_{n+1}, y_{n+1}) + 8f_n - f_{n-1}\right)$	$-\frac{h^4}{24}y^{(4)}(\eta)$
4	$y_{n+1} = y_n + \frac{h}{24}\left(9f(t_{n+1}, y_{n+1}) + 19f_n - 5f_{n-1} + f_{n-2}\right)$	$-\frac{19h^5}{720}y^{(5)}(\eta)$
5	$y_{n+1} = y_n + \frac{h}{720}\left(251f(t_{n+1}, y_{n+1}) + 646f_n - 264f_{n-1} + 106f_{n-2} - 19f_{n-3}\right)$	$-\frac{3h^6}{160}y^{(6)}(\eta)$

FIGURE 9.15 *The Adams-Moulton methods*

the linear interpolant of (t_n, f_n) and (t_{n+1}, f_{n+1}). We then obtain the approximation

$$\int_{t_n}^{t_{n+1}} f(t, y(t))dt \;\approx\; \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_{n+1} - f_n}{h_n}(t - t_n)\right) \;=\; \frac{h_n}{2}(f_n + f_{n+1}),$$

and thus

$$y_{n+1} = y_n + \frac{h_n}{2}(f(t, y_{n+1}) + f_n).$$

As in the backward Euler method, which is just the first order Adams-Moulton method, y_{n+1} is specified implicitly through a nonlinear equation. The higher-order Adams-Moulton methods are derived similarly, and in Figure 9.15 we specify the first five members in the family.

The LTE coefficient for any AM method is slightly smaller than the LTE coefficients for the corresponding AB method. Analogous to `ABstep`, we have

```
  function [tnew,ynew,fnew] = AMstep(f,tc,yc,fvals,h,k)
% Single step of the kth order Adams-Moulton method.
%
% f is a handle that references a function of the form f(t,y)
% where t is a scalar and y is a column d-vector.
%
% yc is an approximate solution to y'(t) = f(t,y(t)) at t=tc.
%
% fvals is an d-by-k matrix where fvals(:,i) is an approximation
% to f(t,y) at t = tc +(2-i)h, i=1:k.
%
% h is the time step.
%
% k is the order of the AM method used, 1<=k<=5.
%
% tnew=tc+h
% ynew is an approximate solution at t=tnew
% fnew = f(tnew,ynew).

if k==1,     ynew = yc + h*fvals;
elseif k==2, ynew = yc + (h/2)*(fvals*[1;1]);
elseif k==3, ynew = yc + (h/12)*(fvals*[5;8;-1]);
elseif k==4, ynew = yc + (h/24)*(fvals*[9;19;-5;1]);
elseif k==5, ynew = yc + (h/720)*(fvals*[251;646;-264;106;-19]);
end
tnew = tc+h;
fnew = f(tnew,ynew);
```

We could discuss methods for the solution of the nonlinear $F(z) = 0$ that defines y_{n+1}. However, we have other plans for the Adams-Moulton methods that circumvent this problem.

9.3.4 The Predictor-Corrector Idea

A very important framework for solving IVPs results when we couple an Adams-Bashforth method with an Adams-Moulton method of the same order. The idea is to *predict* y_{n+1} using an Adams-Bashforth method and then to *correct* its value using the corresponding Adams-Moulton method. In the second-order case, AB2 gives

$$y_{n+1}^{(P)} = y_n + \frac{h}{2}(3f_n - f_{n-1}),$$

which then is used in the right-hand side of the AM2 recipe to render

$$y_{n+1}^{(C)} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}^{(P)}) + f_n\right).$$

For general order we have developed a function

```
[tnew,yPred,fPred,yCorr,fCorr] = PCstep(f,tc,yc,fvals,h,k)
```

that implements this idea. It involves a simple combination of `ABStep` and `AMStep`:

```
[tnew,yPred,fPred] = ABstep(f,tc,yc,fvals,h,k);
[tnew,yCorr,fCorr] = AMstep(f,tc,yc,[fPred fvals(:,1:k-1)],h,k);
```

The repeated application of this function defines the fixed-step predictor-corrector framework:

```
  function [tvals,yvals] = FixedPC(f,t0,y0,h,k,n)
% Produces an approximate solution to the initial value problem
% y'(t) = f(t,y(t)), y(t0) = y0 using a strategy that is based upon a k-th order
% Adams Predictor-Corrector framework. Stepsize is fixed.
%
% f = handle that references the function f.
% t0 = initial time.
% y0 = initial condition vector.
% h  = stepsize.
% k  = order of method. (1<=k<=5).
% n  = number of steps to be taken,
%
% tvals(j) = t0 + (j-1)h, j=1:n+1
% yvals(j,:) = approximate solution at t = tvals(j), j=1:n+1

[tvals,yvals,fvals] = StartAB(f,t0,y0,h,k);
tc = tvals(k);
yc = yvals(:,k)';
fc = fvals(:,k);

for j=k:n
   % Take a step and then update.
   [tc,yPred,fPred,yc,fc] = PCstep(f,tc,yc,fvals,h,k);
   tvals = [tvals tc];
   yvals = [yvals; yc'];
   fvals = [fc fvals(:,1:k-1)];
end
```

The error associated with this method when applied to the model problem is given in Figure 9.16 on the next page.

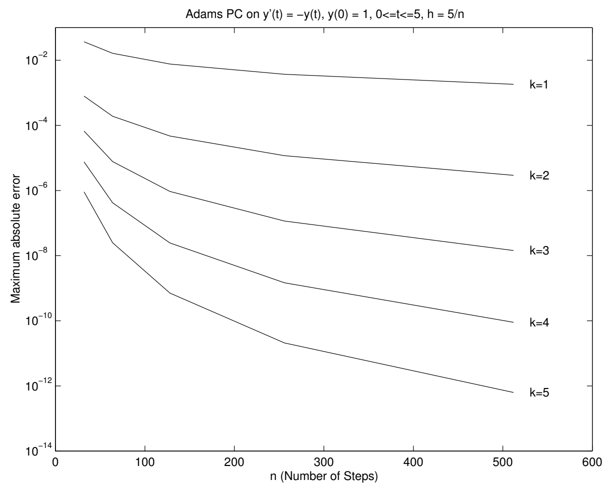

FIGURE 9.16 *kth Order predictor-corrector error*

Problems

P9.3.1 Write functions

```
[tvals,yvals] = AFixedAB(A,t0,y0,h,k,n)
[tvals,yvals] = AFixedAM(A,t0,y0,h,k,n)
```

that can be used to solve the IVP $y'(t) = Ay(t)$, $y(t_0) = y_0$, where A is a d-by-d matrix. In `AFixedAM` a linear system will have to be solved at each step. Get the factorization "out of the loop."

P9.3.2 Use `FixedAB` and `FixedPC` to solve the IVP described in problem P9.2.3. Explore the connections between step size, order, and the number of required function evaluations.

9.3.5 Stepsize Control

The idea behind error estimation in adaptive quadrature is to compute the integral in question in two ways, and then accept or reject the better estimate based on the observed discrepancies. The predictor-corrector framework presents us with a similar opportunity. The quality of $y_{n+1}^{(C)}$ can be estimated from $|y_{n+1}^{(C)} - y_{n+1}^{(P)}|$. If the error is too large, we can reduce the step. If the error is too small, then we can lengthen the step. Properly handled, we can use this mechanism to integrate the IVP across the interval of interest with steps that are as long as possible given a prescribed error tolerance. In this way we can compute the required solution, more or less minimizing the number of f evaluations. The MATLAB IVP solvers `ode23` and `ode45` are Runge-Kutta based and do just that. We develop a second-order adaptive step solver based on the second-order AB and AM methods.

Do we accept $y^{(C)}$ as our chosen y_{n+1}? If $\Delta = |y_{n+1}^{(P)} - y_{n+1}^{(C)}|$ is small, then our intuition tells us that $y_{n+1}^{(C)}$ is probably fairly good and worth accepting as our approximation to $y(t_{n+1})$. If not, there are two possibilities. We could refine $y_{n+1}^{(C)}$ through repeated application of the AM2 formula:

```
$y_{n+1} = y_{n+1}^{(C)}$
Repeat:
    $y_{n+1} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}) + f_n\right)$
```

A reasonable termination criterion might be to quit as soon as two successive iterates differ by a small amount. The goal of the iteration is to produce a solution to the AM2 equation. Alternatively, we could halve h and try another predict/correct step [i.e., produce an estimate y_{n+1} of $y(t_n + h/2)$]. The latter approach is more constructive because it addresses the primary reason for discrepancy between the predicted and corrected value: an overly long step h.

To implement a practical step size control process, we need to develop a heuristic for estimating the error in $y_{n+1}^{(c)}$ based on the discrepancy between it and $y_{n+1}^{(P)}$. The idea is to manipulate the LTE expressions

$$\begin{aligned} y(t_{n+1}) &= y_{n+1}^{(P)} + \frac{5}{12}h^3 y^{(3)}(\eta_1), \qquad \eta_1 \in [t_n, t_n + h] \\ y(t_{n+1}) &= y_{n+1}^{(C)} - \frac{1}{12}h^3 y^{(3)}(\eta_2), \qquad \eta_2 \in [t_n, t_n + h] \end{aligned}$$

We make the assumption that $y^{(3)}$ does *not* vary much across $[t_n, t_n + h]$. Subtracting the first equation from the second leads to approximation

$$|y_{n+1}^{(C)} - y_{n+1}^{(P)}| \approx \frac{1}{2}h^3 |y^{(3)}(\eta)|, \qquad \eta \in [t_n, t_n + h]$$

and so

$$|y_{n+1}^{(C)} - y(t_{n+1})| \approx \frac{1}{6}|y_{n+1}^{(C)} - y_{n+1}^{(P)}|.$$

This leads to the following framework for a second-order predictor-corrector scheme:

```
$y_{n+1}^{(P)} = y_n + \frac{h}{2}(3f_n - f_{n-1})$
$y_{n+1}^{(C)} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}^{(P)} + f_n\right)$
$\epsilon = \frac{1}{6}|y_{n+1}^{(C)} - y_{n+1}^{(P)}|$
```

```
If $\epsilon$ is too big, then
    reduce $h$ and try again.
Else if $\epsilon$ is about right, then
    set $y_{n+1} = y^{(C)}_{n+1}$ and keep $h$.
Else if $\epsilon$ is too small, then
    set $y_{n+1} = y^{(C)}_{n+1}$ and increase $h$.
```

The definitions of "too big," "about right," and "too small" are central. Here is one approach. Suppose we want the global error in the solution snapshots across $[t_0, t_{max}]$ to be less than δ. If it takes n_{max} steps to integrate across $[t_0, t_{max}]$, then we can heuristically guarantee this if

$$\sum_{n=1}^{n_{max}} \text{LTE}_n \leq \delta.$$

Thus if h_n is the length of the nth step, and

$$|\text{LTE}_n| \leq \frac{h_n \delta}{t_{max} - t_0},$$

then

$$\sum_{n=1}^{n_{max}} \text{LTE}_n \;\leq\; \sum_{n=1}^{n_{max}} \frac{h_n \delta}{t_{max} - t_0} \;\leq\; \delta.$$

This tells us when to accept a step. But if the estimated LTE is considerably smaller than the threshold, say

$$\epsilon \leq \frac{1}{10} \frac{\delta h}{t_{max} - t_0},$$

then it might be worth doubling h.

If the ϵ is too big, then our strategy is to halve h. But to carry out the predictor step with this step size, we need $f(t_n - h/2, y_{n-1/2})$ where $y_{n-1/2}$ is an estimate of $y(t_n - h/2)$. "Missing" values in in this setting can be generated by interpolation or by using (for example) an appropriate Runge-Kutta estimate.

We mention that the MATLAB IVP solver `ode113` implements an Adams-Bashforth-Moulton predictor-corrector framework.

Problems

P9.3.3 Derive an estimate for $|y^{(C)}_{n+1} - y(t_{n+1})|$ for the third-, fourth- and fifth-order predictor-corrector pairs.

M-Files and References

Script Files

`ShowTraj`	Shows family of solutions.
`ShowEuler`	Illustrates Euler method.
`ShowFixedEuler`	Plots error in fixed step Euler for y'=y, y(0)=1.
`ShowTrunc`	Shows effect of truncation error.
`EulerRoundoff`	Illustrates Euler in three-digit floating point.
`ShowAB`	Illustrates `FixedAB`.
`ShowPC`	Illustrates `FixedPC`.
`ShowRK`	Illustrates `FixedRK`.
`ShowMatIVPTools`	Illustrates `ode23` and `ode45` on a system.

Function Files

`FixedEuler`	Fixed step Euler method.
`ABStart`	Gets starting values for Adams methods.
`ABStep`	Adams-Bashforth step (order <= 5).
`FixedAB`	Fixed step size Adams-Bashforth.
`AMStep`	Adams-Moulton step (order <= 5).
`PCStep`	AB-AM predictor-corrector Step (order <= 5).
`FixedPC`	Fixed stepsize AB-AM predictor-corrector.
`RKStep`	Runge-Kutta step (order <= 5).
`FixedRK`	Fixed step size Runge-Kutta.
`Kepler`	For solving two-body IVP.
`f1`	The f function for the model problem y'=y.

References

C.W. Gear (1971). *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, Englewood Cliffs, NJ.

J. Lambert (1973). *Computational Methods in Ordinary Differential Equations*, John Wiley, New York.

J. Ortega and W. Poole (1981). *An Introduction to Numerical Methods for Differential Equations*, Pitman, Marshfield, MA.

L. Shampine and M. Gordon (1975). *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, Freeman, San Francisco.