CHAPTER 6

Ordinary Differential Equations

FRONT NOTES———————

[Photos: Tacoma narrows bridge]

On the morning of November 7, 1940, the Tacoma Narrows Bridge, the third longest suspension bridge in the world at the time, fell into Puget Sound. During its short life, it had already become famous for its pronounced vertical oscillations during high winds.

But the motion which preceded the collapse at 11 a.m. on that morning was primarily torsional, twisting from side-to-side. This motion had not been seen prior to that day, and continued for 45 minutes before the collapse. The twisting motion eventually became large enough to snap a support cable, and the bridge disintegrated rapidly.

The debate among architects and engineers about the reason for collapse has continued unabated since that time. The winds were unusually high, even for the Puget Sound. It was known that high winds caused vertical oscillation for aerodynamic reasons, with the bridge acting like an airplane wing. In fact, the bridge's integrity was not in danger from strictly vertical movements. The mystery is how the torsional oscillation arose on that day. Reality Check 6 on page 296 investigates a possible mechanism.

END FRONT NOTES—————

A differential equation is an equation involving derivatives. Differential equations models are the primary means of representing, understanding, and predicting systems that are changing with time. In the form

$$y'(t) = f(t, y(t)),$$

the first-order differential equation expresses the rate of change of a quantity in terms of the present time and the current value of the quantity.

A wide majority of interesting equations have no closed-form solution, and so approximations are the only recourse. This chapter covers the approximate solution of ordinary differential equations by computational methods. After introductory ideas on differential equations, Euler's method is described and analyzed in detail. Although too simple to be heavily used in simulations, Euler's method is crucial, since most of the important issues in the subject can be easily understood in its very simple context.

More sophisticated methods follow, and interesting examples of systems of differential equations are explored. Variable step-size protocols are important for efficient solution, and special methods are necessary for stiff problems. The chapter ends with an introduction to implicit and multistep methods.

6.1 Initial value problems

MANY physical laws that have been successful in modeling nature are expressed in the form of differential equations. Sir Isaac Newton wrote his laws of motion in this form: $F = ma$ is an equation connecting the composite force acting on an object and the object's acceleration, which is the second derivative of the position. In fact, Newton's postulation of his laws together with development of the infrastructure needed to write them down (calculus) comprised one of the most important revolutions in the history of science.

A simple model known as the **logistic equation** models the rate of change of a population as

$$y' = ay(1 - y) \tag{1}$$

where the independent variable t represents time and y' denotes the derivative with respect to t. If we think of y as representing the population as a proportion of the carrying capacity of the animal's habitat, then we expect y to grow to near that capacity and then taper off. The differential equation (1) shows the rate of change y' as being proportional to the product of the current population y and the "remaining capacity" $1 - y$. Therefore the rate of change is small both when the population is small (y near 0) and also when the population nears capacity (y near 1).

The ordinary differential equation (1) is typical in that it has infinitely many solutions $y(t)$. By specifying an initial condition we can identify which of the infinite family we are interested in. (We will get more precise about existence and uniqueness in the next section.) An **initial value problem** for a first order ordinary differential equation is to solve the equation together with an initial condition on a specific interval $a \leq t \leq b$:

$$\begin{cases} y' = f(t, y) \\ y(a) = y_a \\ t \in [a, b]. \end{cases} \tag{2}$$

It will be very helpful for us to think of differential equations as field of slopes, as in Figure 1(a). The equation (1) can be viewed as specifying a slope for any current values of (t, y). If we plot the slope at each point in the plane with an arrow we get the **slope field**, or **direction field**, of the differential equation. When in addition an initial condition is specified, then one out of the infinite family of solutions can be identified. In Figure 1(b), two different solutions are plotted starting at two different initial values, $y(0) = 0.2$ and $y(0) = 1.4$, respectively.

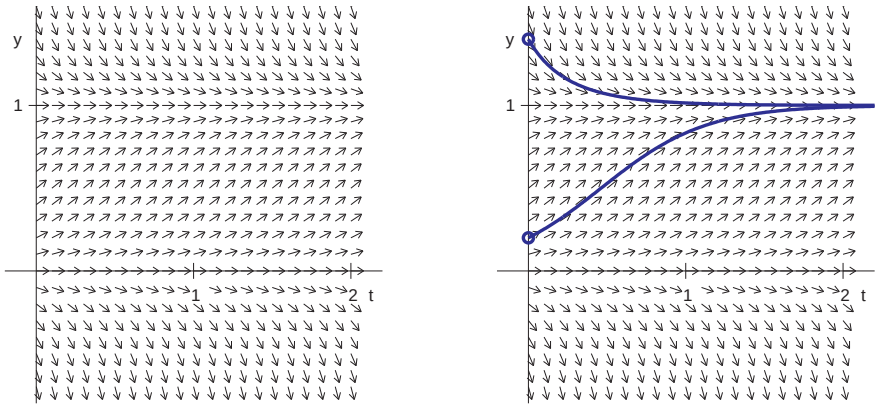
Equation (1) has a solution that can be written in terms of elementary functions. One checks by differentiating and substituting that as long as the initial condition $y_0 \neq 0$,

$$y(t) = 1 - \frac{1}{1 + \frac{y_0}{1-y_0} e^{at}} \tag{3}$$

is the solution of the initial value problem

$$\begin{cases} y' = ay(1 - y) \\ y(0) = y_0 \\ t \in [0, T]. \end{cases} \tag{4}$$

The solution follows the arrows in Figure 1(b). If $y_0 = 0$, the solution is $y(t) = 0$, which is checked the same way.



(a) (b)

Figure 1: The logistic differential equation. (a) The slope field varies in the y-direction but is constant for all t, the definition of an autonomous equation. (b) Two solutions of the differential equation.

6.1.1 Euler's method.

The logistic equation had an explicit, fairly simple solution. A much more common scenario is a differential equation with no explicit solution formula. The geometry of Figure 1 suggests an alternate approach: to computationally "solve" the differential equation by following arrows. Start at the initial condition (t_0, y_0), and follow the direction specified there. After moving a short distance, re-evaluate the slope at the new point (t_1, y_1), move further according to the new slope, and repeat the process. There will be some error associated with the process, since in between evaluations of the slope, we will not be moving along a completely accurate slope. But if the slopes change slowly, we may get a fairly good approximation to the solution of the initial value problem.

Example 6.1 Draw the slope field of the initial value problem

$$\begin{cases} y' = ty + t^3 \\ y(0) = y_0 \end{cases} \tag{5}$$

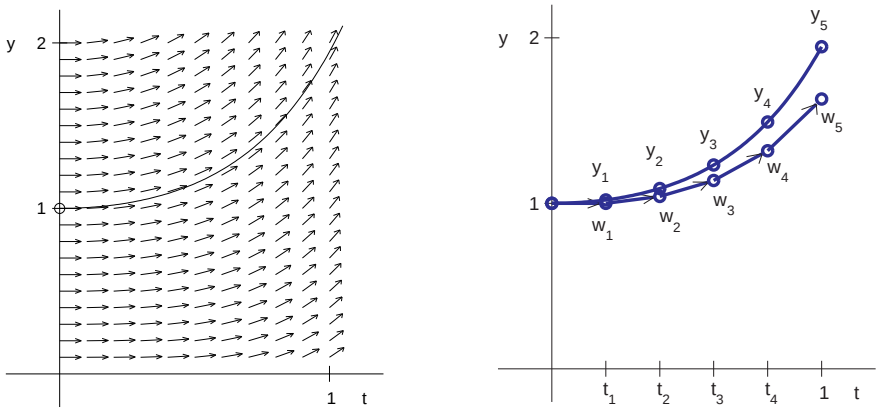
For each point (t, y) in the plane, an arrow with slope equal to $ty + y^3$ was plotted in Figure 2(a). This IVP is called nonautonomous because t appears explicitly in the right-hand-side of the equation. It is also clear from the slope field, which varies according to both t and y. An exact solution $y(t) = 3e^{t^2/2} - t^2 - 2$ is plotted for initial condition $y(0) = 1$, but let us assume that we don't know it.

Figure 2(b) shows an implementation of the method of computationally following the slope field, which is known as Euler's method. We begin with a grid of points

$$t_0 < t_1 < t_2 < \ldots < t_n \tag{6}$$

along the t-axis and assign approximate y values

$$w_0 < w_1 < w_2 < \ldots < w_n \tag{7}$$



(a) (b)

Figure 2: Solution of the initial value problem (5). (a) Slope field for a nonautonomous equation varies in both y and t. Solution is shown. (b) Application of Euler's method to the equation, with stepsize $h = 0.2$.

at the respective t points. In Figure 2(b), the t points were selected to be

$$t_0 = 0.0 < t_1 = 0.2 < t_2 = 0.4 < t_3 = 0.6 < t_4 = 0.8 < t_5 = 1.0, \tag{8}$$

the $y_i = y(t_i)$ correspond to the values on the exact solution curve, and each w_i is an approximation to the solution at t_i. The t_i points are equally spaced with stepsize $h = 0.2$.

■

Since the change in y is the horizontal distance h multiplied by the slope, the formula for each step can be expressed as follows:

Euler's Method

$$\begin{aligned} w_0 &= y_0 \\ w_{i+1} &= w_i + hf(t_i, w_i). \end{aligned} \tag{9}$$

Example 6.2 *Apply Euler's method to initial value problem (5), with initial condition* $y_0 = 1$.

The right-hand-side of the differential equation is $f(t, y) = ty + t^3$. Therefore Euler's method will be the iteration

$$\begin{aligned} w_0 &= 1 \\ w_{i+1} &= w_i + h(t_i w_i + t_i^3). \end{aligned} \tag{10}$$

Using the grid (8) with step size $h = 0.2$, we calculate the approximate solution iteratively from (10). The values w_i given by Euler's method are compared to the true values y_i in the following table, and plotted in Figure 2(b).

step	t_i	w_i	y_i	e_i
0	0.0	1.0000	1.0000	0.0000
1	0.2	1.0000	1.0206	0.0206
2	0.4	1.0416	1.0899	0.0483
3	0.6	1.1377	1.2317	0.0939
4	0.8	1.3175	1.4914	0.1739
5	1.0	1.6306	1.9462	0.3155

The table also shows the error $e_i = y_i - w_i$ at each step. The error tends to grow, from zero at the initial condition to its largest value at the end of the interval, although the maximum error will not always be found at the end.

Applying Euler's method with step size $h = 0.1$ causes the error to decrease. Again using (10) we calculate

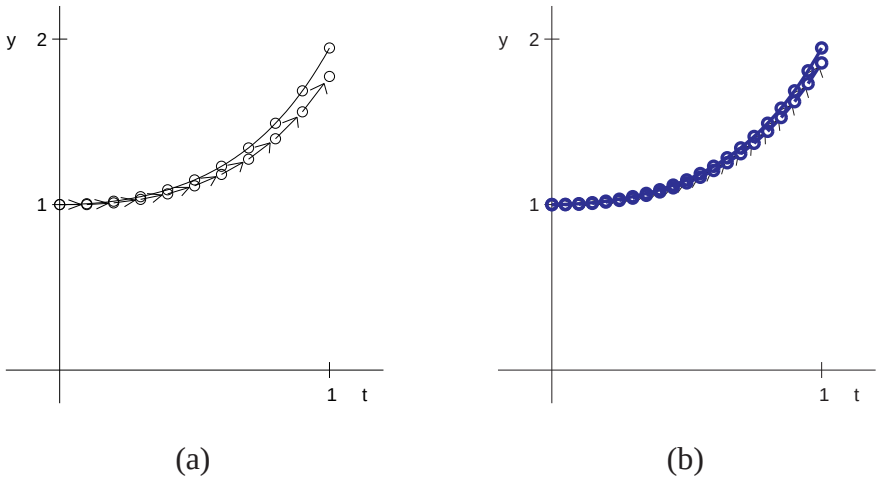
step	t_i	w_i	y_i	e_i
0	0.0	1.0000	1.0000	0.0000
1	0.1	1.0000	1.0050	0.0050
2	0.2	1.0101	1.0206	0.0105
3	0.3	1.0311	1.0481	0.0170
4	0.4	1.0647	1.0899	0.0251
5	0.5	1.1137	1.1494	0.0357
6	0.6	1.1819	1.2317	0.0497
7	0.7	1.2744	1.3429	0.0684
8	0.8	1.3979	1.4914	0.0934
9	0.9	1.5610	1.6879	0.1269
10	1.0	1.7744	1.9462	0.1718

Compare the final error e_{10} for the $h = 0.1$ calculation to the final error e_5 for the $h = 0.2$ calculation. Note that cutting the step size h in half results in cutting the final error approximately in half.

■

Euler's method is implemented in the following Matlab code, which has been written in somewhat modular form, to highlight the three individual components. The plotting program calls a subprogram to execute each single Euler step, which in turn calls the function containing the right-hand-side of the differential equation. In this form, it will be easy later to change out both the right-hand-side, for another differential equation, and the Euler method, for another more sophisticated method.

```
%Program 6.1 Euler's Method for Solving Initial Value Problems
%Use with ydot.m to evaluate rhs of differential equation
function euler(int,y0,h)
% input interval [a,b], initial value y0, step size h
```



(a) (b)

Figure 3: Euler's method applied to IVP (5). The arrows show the Euler steps, exactly as in Figure 2 except for the step size. (a) Ten steps of size h = 0.1 (b) Twenty steps of size h = 0.05

```
% Example usage: euler([0 1],1,0.1);
a=int(1);b=int(2);
t(1)=0; y(1)=y0;
n=round((b-a)/h);
for i=1:n
  t(i+1)=t(i)+h;
  y(i+1)=eulerstep(t(i),y(i),h);
end
plot(t,y)

function y=eulerstep(t,x,h)
%one step of the Euler method
%Input: t is current time, x is current value, h is stepsize
%Output: the approximate solution value at time t+h
y=x+h*ydot(t,x);

function ydot=ydot(t,y)
ydot = t*y + t^3
```

Comparing the Euler method approximation for (5) with the exact solution at $t = 1$ gives us the following table.

steps n	step size h	error at $t = 1$
5	0.20000	0.3155
10	0.10000	0.1718
20	0.05000	0.0899
40	0.02500	0.0460
80	0.01250	0.0233
160	0.00625	0.0117
320	0.00312	0.0059
640	0.00157	0.0029

Two facts are evident from the table and Figures 3 and 4. First, the error is nonzero. Since Euler's method takes non-infinitesimal steps, the slope changes along the step, and the approximation does not lie exactly on the solution curve. Second, the error decreases as the stepsize is decreased, as can be also seen in Figure 3. It appears from the table that the error is proportional to h; we will investigate this further in the next section.

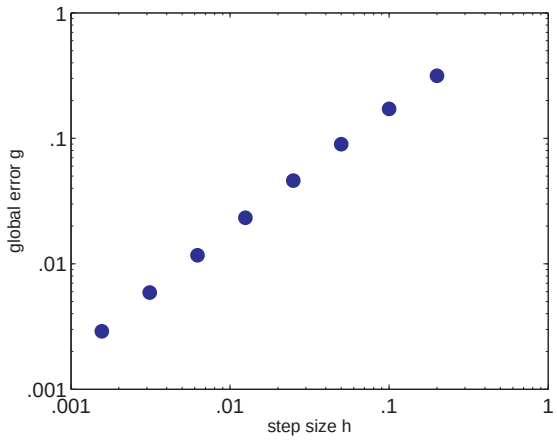


Figure 4: Error as a function of stepsize for Euler's method. The difference between the approximate solution of (5) and the correct solution at $t = 1$ has slope 1 on a loglog plot, so is proportional to the stepsize h, for small h.

Example 6.3 Find the Euler's method formula for the initial value problem

$$\begin{cases} y' = cy \\ y(0) = y_0 \\ t \in [0, 1] \end{cases} \tag{11}$$

Here c is an arbitrary constant. The true solution is $y(t) = y_0 e^{ct}$. Euler's method formula gives

$$\begin{aligned} w_0 &= y_0 \\ w_{i+1} &= w_i + hcw_i = (1 + hc)w_i. \end{aligned}$$

From this we conclude

$$w_i = (1 + hc)w_{i-1} = (1 + hc)^2 w_{i-2} = \ldots = (1 + hc)^i w_0. \tag{12}$$

Setting $h = 1/n$ for an integer n, the value at $t = 1$ is

$$\begin{aligned} w_n &= (1 + hc)^n y_0 \\ &= (1 + \frac{c}{n})^n y_0 \end{aligned}$$

The classical formula says that

$$\lim_{n\to\infty} (1 + \frac{c}{n})^n = e^c \tag{13}$$

which shows that as $n \to \infty$, Euler's method will converge to the correct value.

■

6.1.2 Existence, uniqueness, and continuity for solutions.

Before starting a computational method to find a solution to a problem, it is helpful to know that the solution exists. Further, it is helpful to know that there is only one solution, so that the solution algorithm is not confused about which one to calculate. Under the right circumstances, initial value problems have exactly one solution.

Definition 6.1 A function $f(t, y)$ is **Lipschitz** in the variable y on a convex set C if there exists a constant L (called the **Lipschitz constant**) satisfying

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|$$

for each t and for each $y_1, y_2 \in C$.

Notice that Lipschitz in a variable implies that the function is continuous in that variable, but not necessarily differentiable.

Example 6.4 *Find the Lipschitz constant for the differential equation (5)*

The right-hand-side $f(t, y) = ty + t^3$ is Lipschitz in the variable y on the set $0 \leq t \leq 1, -\infty < y < \infty$. Check that

$$|f(t, y_1) - f(t, y_2)| = |ty_1 - ty_2| \leq |t||y_1 - y_2| \leq |y_1 - y_2| \tag{14}$$

on the set. The Lipschitz constant is 1.

■

Note that if the function f is continuously differentiable in the variable y, the maximum absolute value of the partial derivative $\frac{\partial f}{\partial y}$ is a Lipschitz constant. According to the Mean Value Theorem, for each fixed t there is a c between y_1 and y_2 such that

$$\frac{f(t, y_1) - f(t, y_2)}{y_1 - y_2} = \frac{\partial f}{\partial y}(t, c).$$

Therefore L can be taken to be the maximum of

$$\left|\frac{\partial f}{\partial y}(c)\right|$$

on the set.

The Lipschitz hypotheses guarantees the existence and uniqueness of solutions of initial value problems. We refer to [3] for a proof of the following theorem.

Theorem 6.2 If $f(t, y)$ is Lipschitz in the variable y on an interval $[a, b] \times [y_1, y_2]$ where $y_1 < y_a < y_2$, then for some c between a and b the initial value problem (2)

$$\begin{cases} y' = f(t, y) \\ y(a) = y_a \\ t \in [a, c]. \end{cases} \tag{15}$$

has exactly one solution $y(t)$.

The fine print of Theorem 6.2 is important to understand, especially if your goal is to calculate the solution numerically. Just because the initial value problem satisfies a Lipschitz condition on $[a, b] \times [y_1, y_2]$ containing the initial condition doesn't guarantee a solution for t in *the entire interval* $[a, b]$. The simple reason is that the solution may wander outside the y range $[y_1, y_2]$ for which the Lipschitz constant is valid. The best that can be said is that the solution exists on some shorter interval $[a, b]$. This point is illustrated by the following example.

Example 6.5 *Discuss existence and uniqueness for the initial value problem*

$$\begin{cases} y' = y^2 \\ y(0) = 1 \\ t \in [0, 2]. \end{cases} \tag{16}$$

The partial derivative of f with respect to y is $2y$. If we apply Theorem 6.2 on the set $0 \leq t \leq 2, -10 \leq y \leq 10$, for example, the Lipschitz constant $\max |2y| = 20$ is valid on the entire set. The theorem guarantees a solution starting at $t = 0$ and existing out as far as some $c > 0$, but we are not guaranteed a solution on the entire interval $[0, 2]$.

In fact, the solution of the differential equation (16) guaranteed by the theorem is $y(t) = 1/(1 - t)$, which can be easily checked. This solution goes to infinity as t approaches 1. In other words, the solution exists on the interval $0 \leq t \leq c$ for any $0 < c < 1$, but not for $c = 2$. As mentioned above, the problem is easy to notice - the Lipschitz constant 20 is valid for $|y| \leq 10$, but y along the solution exceeds 10 long before t reaches 2.

■

Theorem 6.3 is the basic fact about stability (error amplification) for ordinary differential equations. If a Lipschitz constant exists for the right-hand-side of the differential equation, then the solution at a later time is a Lipschitz function of the initial value, with a new Lipschitz constant which is exponential in the original one.

Theorem 6.3 Assume that $f(t, y)$ is Lipschitz in the variable y on the set $S = [a, b] \times [y_1, y_2]$. If $Y(t)$ and $Z(t)$ are solutions in S of the differential equation

$$y' = f(t, y)$$

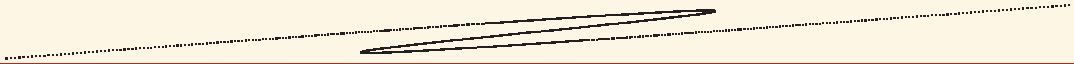
with initial conditions $Y(a)$ and $Z(a)$ respectively, then

$$|Y(t) - Z(t)| \leq e^{L(t-a)} |Y(a) - Z(a)| \tag{17}$$

for all $t \in [a, b]$.

SPOTLIGHT ON: ***Conditioning***

Error magnification was discussed in Chapters 1 and 2 as a way to quantify the effects on a problem's solution due to small changes in the input data. The analogue of that question for initial value problems is given a precise answer by Theorem 6.3. When initial condition (input data) $Y(a)$ is changed to $Z(a)$, the greatest possible change in output t time units later, $Y(t) - Z(t)$, is exponential in t. Alternatively, for fixed time t in the future, the output change is linear in the initial condition difference, which means we can talk of a "condition number" whose role is played by $e^{L(t-a)}$.



Proof. First, if $Y(a) = Z(a)$, then by uniqueness of solutions $Y(t) = Z(t)$ and (17) is trivially satisfied. We may assume $Y(a) \neq Z(a)$, in which case $Y(t) \neq Z(t)$ for all t in the interval to avoid contradicting uniqueness.

Define $u(t) = Y(t) - Z(t)$. Since $u(t)$ is either strictly positive or strictly negative, and because (17) depends only on $|u|$, we may assume $u > 0$. Then $u(a) = Y(a) - Z(a)$ and the derivative $u' = Y' - Z' = f(t, y) - f(t, z)$. The Lipschitz condition

$$u' = |f(t, Y) - f(t, Z)| \leq L|Y(t) - Z(t)| = L|u(t)| = Lu(t)$$

implies that $(\ln u)' = \frac{u'}{u} \leq L$. By the Mean Value Theorem,

$$\frac{\ln u(t) - \ln u(a)}{t - a} \leq L,$$

which simplifies to

$$\ln \frac{u(t)}{u(a)} \leq L(t - a)$$
$$u(t) \leq u(a)e^{L(t-a)}$$

which is the desired result. □

Returning to Example 6.4, Theorem 6.3 implies that solutions $Y(t)$ and $Z(t)$, starting at different initial values, must not grow apart any faster than a multiplicative factor of e^t for $0 \leq t \leq 1$. In fact, the solution at initial value Y_0 is $Y(t) = (2 + Y_0)e^{t^2/2} - t^2 - 2$, and so the difference between two solutions is

$$|Y(t) - Z(t)| \leq |(2 + Y_0)e^{t^2/2} - t^2 - 2 - ((2 + Z_0)e^{t^2/2} - t^2 - 2)| \leq |Y_0 - Z_0|e^{t^2/2} \quad (18)$$

which is less than $|Y_0 - Z_0|e^t$ for $0 \leq t \leq 1$, as prescribed by Theorem 6.3.

6.1.3 First-order linear equations

A special class of ordinary differential equations that can be readily solved provides a handy set of illustrative examples. They are the first-order equations whose right-hand sides are linear in the y variable. Consider the initial value problem

$$\begin{cases} y' = g(t)y + h(t) \\ y(a) = y_a \\ t \in [a, b]. \end{cases} \tag{19}$$

First note that if $g(t)$ is continuous on $[a, b]$, a unique solution exists by Theorem 6.2, using $L = \max_{[a,b]} g(t)$ as the Lipschitz constant. The solution is found by a trick, multiplying the equation through by an "integrating factor".

The integrating factor is $\exp(\int g(t)\, dt)$, and multiplying both sides by it yields

$$\begin{aligned} (y' - g(t)y)e^{-\int g(t)\, dt} &= e^{-\int g(t)\, dt} h(t) \\ \left(ye^{-\int g(t)\, dt}\right)' &= e^{-\int g(t)\, dt} h(t) \\ ye^{-\int g(t)\, dt} &= \int e^{-\int g(t)\, dt} h(t)\, dt \end{aligned}$$

which can be solved as

$$y(t) = e^{\int g(t)\, dt} \int e^{-\int g(t)\, dt} h(t)\, dt \tag{20}$$

If the integrating factor can be expressed simply, this method allows an explicit solution of the first-order linear equation (19).

Example 6.6 Solve the first-order linear differential equation

$$\begin{cases} y' = ty + y^3 \\ y(0) = 1 \end{cases} \tag{21}$$

The integrating factor is

$$e^{-\int g(t)\, dt} = e^{-\frac{t^2}{2}}.$$

According to (20), the solution is

$$\begin{aligned} y(t) &= e^{\frac{t^2}{2}} \int e^{-\frac{t^2}{2}} t^3\, dt \\ &= e^{\frac{t^2}{2}} \int e^{-u}(2u)\, du \\ &= 2e^{\frac{t^2}{2}} [-\frac{t^2}{2} e^{-\frac{t^2}{2}} - e^{-\frac{t^2}{2}} + C] \\ &= -t^2 - 2 + 2Ce^{\frac{t^2}{2}} \end{aligned}$$

where the substitution $u = t^2/2$ was made. Solving for the integration constant C yields $1 = -2 + 2C$, so $C = 3/2$. The solution is

$$y(t) = 3e^{\frac{t^2}{2}} - t^2 - 2.$$

■

Exercises 6.1

6.1.1. Show that the function $y(t) = t\sin t$ is a solution of the differential equations (a) $y + t^2\cos t = ty'$ (b) $y'' = 2\cos t - y$ (c) $t(y'' + y) = 2y' - 2\sin t$.

6.1.2. Show that the function $y(t) = e^{\sin t}$ is a solution of the initial value problems (a) $y' = y\cos t, y(0) = 1$ (b) $y'' = (\cos t)y' - (\sin t)y, y(0) = 1, y'(0) = 1$ (c)$y'' = y(1 - \ln y - (\ln y)^2), y(\pi) = 1, y'(\pi) = -1$.

6.1.3. (a) Show that if $a \neq 0$, the solution of the initial value problem $y' = ay + b, y(0) = y_0$ is $y(t) = \frac{b}{a}(e^{at} - 1) + y_0e^{at}$. (b) Verify the inequality of Theorem 6.3 for solutions $y(t), z(t)$ with initial values y_0 and z_0, respectively.

6.1.4. Use separation of variables to find solutions of the IVP given by $y(0) = 1$ and the following differential equations.

(a) $y' = t$ (b) $y' = t^2y$ (c) $y' = 2(t + 1)y$

(d) $y' = 5t^4y$ (e) $y' = 1/y^2$ (f) $y' = t^3/y^2$

[Ans.: (a) $y(t) = e^t$ (b) $y(t) = e^{t^3/3}$ (c) $y(t) = e^{t^2-t}$ (d) $y = e^{t^5}$ (e) $y(t) = (3t + 1)^{1/3}$ (f) $y(t) = (3t^4/4 + 1)^{1/3}$]

6.1.5. Find the solutions of the IVP given by $y(0) = 0$ and the following first-order linear differential equations

(a) $y' = t + y$ (b) $y' = t - y$ (c) $y' = 4t - 2y$

[Ans.: (a) $y(t) = e^t - t - 1$ (b) $y(t) = e^{-t} + t - 1$. (c) $y(t) = e^{-2t} + 2t - 1$]

6.1.6. Which of these differential equations have unique solutions for initial value problems on $[0, 1]$, as guaranteed by Theorem 6.2? Find the Lipschitz constants. (a) $y' = t$ (b) $y' = y$ (c) $y' = -y$ (d) $y' = -y^3$.

6.1.7. Sketch the slope field of the differential equations in Exercise 6.1.6, and draw rough approximations to the solutions starting at the initial conditions $y(0) = 1, y(0) = 0$, and $y(0) = -1$.

6.1.8. Find the solutions of the initial value problems in Exercise 6.1.7. For each equation, use the Lipschitz constants from Exercise 6.1.6, and verify the inequality of Theorem 6.3 for the pair of solutions with initial conditions $y(0) = 0$ and $y(0) = 1$.

6.1.9. Find the solution of the initial value problem $y' = ty^2$ with $y(0) = 1$. What is the largest interval $[0, b]$ for which the solution exists? [Ans.: $y(t) = 2/(2 - t^2)$, the interval $[0, \sqrt{2}]$

6.1.10. (a) Write out the Euler's method formula for the IVP in Exercise 6.1.3. (b) Set $a = b = 1, h = 0.5, y_0 = 1$ and carry out two steps to approximate $y(1)$. (c) Change the step size to $h = 0.25$ and carry out 4 steps to approximate $y(1)$. (d) Using the correct solution from Exercise- 6.1.3, compare the errors of (b) and (c) at $t = 1$.

6.1.11. Write out Euler's method for the IVPs in Exercise 6.1.4. Using stepsize $h = 1/4$, calculate the Euler's method approximation on the interval $[0, 1]$. Compare to the correct solution found above, and find the total error at each step.

6.1.12. Repeat Exercise 6.1.11 for the IVPs in Exercise 6.1.5.

Computer Problems 6.1

6.1.1. Print the values of the Euler's method solution with step size $h = 0.1$ in $[0, 1]$ for the initial value problems in Exercise 6.1.4.

6.1.2. Plot the Euler's method approximate solutions for the IVPs in Exercise 6.1.4 on $[0, 1]$ for stepsizes $h = 0.1, 0.05$, and 0.025 along with the true solution.

6.1.3. Plot the Euler's method approximate solutions for the IVPs in Exercise 6.1.5 on $[0, 1]$ for stepsizes $h = 0.1, 0.05$, and 0.025 along with the true solution.

6.1.4. For the IVP's in Exercise 6.1.4, plot the global error of Euler's method at $t = 1$ as a function of $h = 0.1 \times 2^k$ for $0 \leq k \leq 5$. Use semilog plot as in Figure 4.

6.1.5. For the IVP's in Exercise 6.1.5, plot the global error of Euler's method at $t = 1$ as a function of $h = 0.1 \times 2^k$ for $0 \leq k \leq 5$.

6.2 Analysis of IVP solvers.

IN this section we try to explain Figure 4. In that example, the error in the Euler's method approximation seems decrease as stepsize is decreased. Is this generally true? Can we make the error as small as we want, just by decreasing the step size? A careful investigation of error in Euler's method will illustrate the issues for IVP solvers in general.

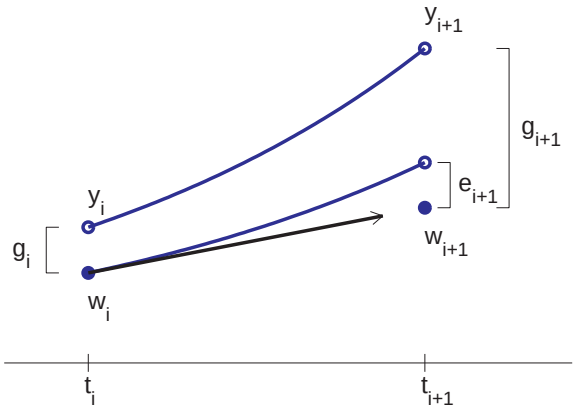


Figure 5: One step of an ODE solver. The Euler method follows a line segment with the slope of the vector field at the current point to the next point (t_{i+1}, w_{i+1}). The upper curve represents the true solution to the differential equation. The global error g_{i+1} is the sum of the local error e_{i+1} and accumulated error from previous steps.

6.2.1 Local and global truncation error

Figure 5 shows a schematic picture for one step of a solver like Euler's method when solving an IVP of form

$$\begin{cases} y' = f(t, y) \\ y(a) = y_a \\ t \in [a, b]. \end{cases} \tag{22}$$

At step i, the accumulated error from the previous steps is carried along and perhaps amplified, while new error from the Euler approximation is added. To be precise let us define the **global truncation error**

$$g_i = |w_i - y_i|,$$

to be the difference between the Euler's method approximation and the correct solution of the IVP. Also, define the **local truncation error**, or one-step error, to be

$$e_{i+1} = |w_{i+1} - z(t_{i+1})|, \tag{23}$$

the difference between the value of the Euler step on that interval and the correct solution of the "one-step initial value problem"

$$\begin{cases} y' = f(t, y) \\ y(t_i) = w_i \\ t \in [t_i, t_{i+1}]. \end{cases} \tag{24}$$

The local truncation error is the error occurring just from a single step, taking the previous solution approximation w_i as the starting point. The global truncation error is the accumulated error from the first i steps. The local and global truncation errors are illustrated in Figure 5. At each step, the new global error is the sum of the amplified global error from the previous step and the new local error. Because of the amplification, the global error is not simply the sum of the local truncation errors.

Example 6.7 *Find the local truncation error for Euler's method.*

According to the definition, this is the new error made on a single step of Euler's method. Assume the previous step w_i is correct, solve the initial value problem (24) exactly, and compare the exact solution $y(t_{i+1})$ to the Euler method approximation.

Assuming y'' is continuous, the exact solution at $t_{i+1} = t_i + h$ is

$$y(t_i + h) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(c)$$

according to Taylor's Theorem, for some (unknown) c satisfying $t_i < c < t_{i+1}$. Since $y(t_i) = w_i$ and $y'(t_i) = f(t_i, w_i)$, this can be written as

$$y(t_{i+1}) = w_i + hf(t_i, w_i) + \frac{h^2}{2}y''(c).$$

Meanwhile, Euler's method says

$$w_{i+1} = w_i + hf(t_i, w_i).$$

Subtracting the two expressions yields

$$e_{i+1} = |w_{i+1} - y(t_{i+1})| = \frac{h^2}{2}|y''(c)|$$

for some c in the interval. This is the local truncation error e_i for Euler's method. If M is an upper bound for y'' on $[a, b]$, then $e_i \leq Mh^2/2$.

■

Now let's investigate how the local errors add up to global errors. At the initial condition $y(a) = y_a$, the global error is $g_0 = |w_0 - y_0| = |y_a - y_a| = 0$. After one step, there is no accumulated error from previous steps and the global error is equal to the first local error, $g_1 = e_1 = |w_1 - y_1|$. After two steps, let's break down g_2 as in Figure 5, into the local truncation error plus the accumulated error from the earlier step. Define $z(t)$ to be the solution of the initial value problem

$$\begin{cases} y' = f(t, y) \\ y(t_1) = w_1 \\ t \in [t_1, t_2]. \end{cases} \tag{25}$$

(We give the solution the name z because y is already being used for the solution to the same IVP starting at the exact initial condition $y(t_0) = y_0$.) Thus $z(t_2)$ is the exact value of the solution starting at initial condition (t_1, w_1). Note that if we use the initial condition (t_1, y_1) we would get

SPOTLIGHT ON: ***Convergence***

Theorem 6.4 is the main theorem on convergence of one-step ode solvers. The dependence of global error on h shows that we can expect error to decrease as h is decreased, so that (at least in exact arithmetic) error can be made as small as desired. Which brings us to the other important point - the exponential dependence of global error on b. As time increases, the global error bound may grow extremely large. For large t_i, the step size h required to keep global error small may be so tiny as to be impractical.

y_2, which is on the actual solution curve, unlike $z(t_2)$. Then $e_2 = |w_2 - z(t_2)|$ is the local truncation error of step 2. The other difference, $z(t_2) - y_2$, is covered by Theorem 6.3, since it is the difference between two solutions of the same equation with different initial conditions w_1 and y_1. Therefore

$$\begin{aligned} g_2 &= |w_2 - y_2| = |w_2 - z(t_2) + z(t_2) - y_2| \\ &\leq |w_2 - z(t_2)| + |z(t_2) - y_2| \\ &\leq e_2 + e^{Lh} g_1 \\ &= e_2 + e^{Lh} e_1. \end{aligned}$$

The argument is the same for step 3, which yields

$$g_3 = |w_3 - y_3| \leq e_3 + e^{Lh} g_2 \leq e_3 + e^{Lh} e_2 + e^{2Lh} e_1, \tag{26}$$

and likewise the general step i satisfies

$$g_i = |w_i - y_i| \leq e_i + e^{Lh} e_{i-1} + e^{2Lh} e_{i-2} + \ldots + e^{(i-1)Lh} e_1. \tag{27}$$

Now we bring in the local truncation error. Assume that it satisfies

$$e_i \leq Ch^{k+1}$$

for a constant $C > 0$. Then

$$\begin{aligned} g_i &\leq Ch^{k+1}(1 + e^{Lh} + \ldots + e^{(i-1)Lh}) \\ &= Ch^{k+1} \frac{e^{iLh} - 1}{e^{Lh} - 1} \\ &\leq Ch^{k+1} \frac{e^{L(t_i - a)} - 1}{Lh} \\ &= \frac{Ch^k}{L}(e^{L(t_i - a)} - 1) \end{aligned} \tag{28}$$

Note how the local truncation error is related to the global truncation error. The local truncation error is proportional to h^k for some K. Roughly speaking, the global truncation error "adds up" the local truncation errors over a number of steps proportional to h^1, the reciprocal of the step size. Thus the global error turns out to be proportional to h^k. This is the major finding of the above calculation, and we state it in the following theorem.

Theorem 6.4 Assume that $f(t,y)$ has a Lipschitz constant L for the variable y and that the value y_i of the solution of the initial value problem (2) at t_i is approximated by w_i from a one-step IVP solver with local truncation error $e_i \leq Ch^{k+1}$, for some constant C and $k \geq 0$. Then for each $a < t_i < b$, the IVP solver has global truncation error

$$g_i = |w_i - y_i| \leq \frac{C}{L}(e^{L(t_i-a)} - 1)h^k. \tag{29}$$

If a method satisfies (29) as $h \to 0$, we say that the method has **order** k. Example 6.7 shows that the local truncation error of Euler's method is of size bounded by $Mh^2/2$, so the order of Euler's method is 1. Restating the theorem in the Euler's method case gives the following corollary.

Corollary 6.5 *(Euler's method convergence.) Assume that $f(t,y)$ has a Lipschitz constant L for the variable y and that the solution y_i of the initial value problem (2) at t_i is approximated by w_i using Euler's method. Let M be an upper bound for $|y''(t)|$ on $[a,b]$. Then*

$$|w_i - y_i| \leq \frac{Mh}{2L}(e^{L(t_i-a)} - 1). \tag{30}$$

Example 6.8 Find an error bound for Euler's method in Example 6.2.

The Lipschitz constant on $[0,1]$ is $L = 1$. Now that the solution $y(t) = 3e^{t^2/2} - t^2 - 2$ is known, the second derivative is determined to be $y''(t) = (t^2+2)e^{t^2/2} - 2$, whose absolute value is bounded above on $[0,1]$ by $M = 3\sqrt{e} - 2$. Corollary 6.5 implies that the global truncation error at $t = b = 1$ must be smaller than

$$\frac{Mh}{2L}e^L(1-0) = \frac{(3\sqrt{e}-2)}{2}eh \approx 4.004h. \tag{31}$$

This upper bound is confirmed by the actual global truncation errors, shown in Figure 4, which are roughly 2 times h for small h.

■

So far, Euler's method seems to be foolproof. It is intuitive in construction, and the errors it makes get smaller when the step size decreases, according to Corollary 6.5. However, for more difficult IVP's, Euler's method is rarely used. There exist more sophisticated methods whose order, or power of h in (29), is greater than one. This leads to vastly reduced global error, as we shall see. We close this section with an innocent-looking example where such a reduction in error is needed.

Example 6.9 *Apply Euler's method to the IVP*

$$\begin{cases} y' = -4t^3y^2 \\ y(-10) = 1/10001 \\ t \in [-10,0]. \end{cases} \tag{32}$$

It is easy to check by substitution that the exact solution is $y(t) = 1/(t^4+1)$. The solution is very well behaved on the interval of interest. We will assess the ability of Euler's method to approximate the solution at $t = 0$.

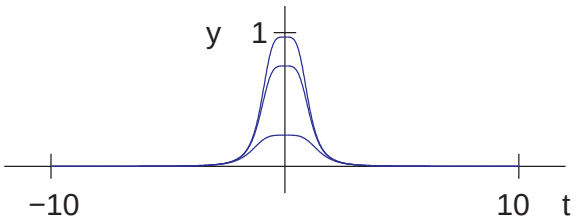


Figure 6: Approximation of Example 6.9 by Euler's method. From bottom to top, step sizes are $h = 10^{-3}, h = 10^{-4}$ and $h = 10^{-5}$. The correct solution has $y(0) = 1$. Extremely small steps are needed to get a reasonable approximation.

Figure 6 shows Euler's method approximations to the solution, with step sizes $h = 10^{-3}, 10^{-4}$ and 10^{-5}, from bottom to top. The value of the correct solution at $t = 0$ is $y(0) = 1$. Even the best approximation, which uses one million steps to reach $t = 0$ from the initial condition, is noticeably incorrect.

■

This example shows that more accurate methods are needed to achieve accuracy in a reasonable amount of computation. The remainder of the chapter is devoted to developing more sophisticated methods that require fewer steps to get the same or better accuracy.

6.2.2 The explicit trapezoid method

A small adjustment in the Euler's method formula makes a great improvement in accuracy. Consider the following geometrically motivated method:

Explicit trapezoid method

$$\begin{aligned} w_0 &= y_0 \\ w_{i+1} &= w_i + \frac{h}{2}(f(t_i, w_i) + f(t_i + h, w_i + hf(t_i, w_i))) \end{aligned} \tag{33}$$

For Euler's method, the slope $y'(t_i)$ governing the discrete step is taken from the slope field at the left-hand end of the interval $[t_i, t_{i+1}]$. In the trapezoid method, this slope is replaced by the average between the contribution $y'(t_i)$ from the left-hand endpoint and the corresponding slope $f(t_i + h, w_i + hf(t_i, w_i))$ from the right-hand side. (See Figure 14(a).) Note that we are using the Euler's method "prediction" as the w-value to evaluate the slope function f at $t_{i+1} = t_i + h$. The Euler's method prediction is corrected by the trapezoid method, which is more accurate, as we will show.

The method is called **explicit** because the new approximation w_{i+1} can be determined by an explicit formula in terms of previous w_i. The reason for the name "trapezoid method" is that in the special case where $f(t, y)$ is independent of y, the method

$$w_{i+1} = w_i + \frac{h}{2}[f(t_i) + f(t_i + h)]$$

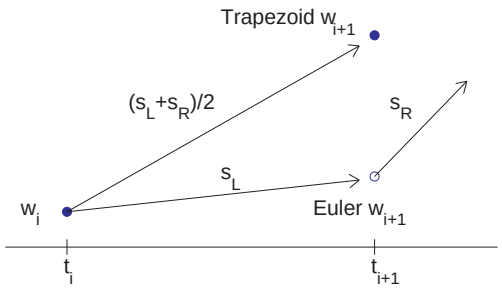


Figure 7: Schematic view of single step of the explicit trapezoid method. The slopes $s_L = f(t_i, w_i)$ and $s_R = f(t_i + h, w_i + hf(t_i, w_i))$ are averaged to define the slope used to advance the solution to t_{i+1}.

can be viewed as adding a trapezoid rule approximation (Chapter 5) of the integral $\int_{t_i}^{t_i+h} f(t)\,dt$ to the current w_i. Since

$$\int_{t_i}^{t_i+h} f(t)\,dt = \int_{t_i}^{t_i+h} y'(t)\,dt = y(t_i + h) - y(t_i),$$

this corresponds to solving the differential equation $y' = f(t)$ by integrating both sides using the trapezoid rule. The explicit trapezoid method is also called the improved Euler method and the Heun method in the literature, but we will use the more descriptive and more easily remembered title.

Let's test the new method on an old example.

Example 6.10 Apply the explicit trapezoid method to the IVP (5) with initial condition $y(0) = 1$.

Formula (33) for $f(t, y) = ty + t^3$ is

$$\begin{aligned} w_0 &= y_0 = 1 \\ w_{i+1} &= w_i + \frac{h}{2}(f(t_i, w_i) + f(t_i + h, w_i + hf(t_i, w_i))) \\ &= w_i + \frac{h}{2}(t_i y_i + t_i^3 + (t_i + h)(w_i + h(t_i y_i + t_i^3)) + (t_i + h)^3) \end{aligned}$$

Using step size $h = 0.1$, the iteration yields the following table.

step	t_i	w_i	y_i	e_i
0	0.0	1.0000	1.0000	0.0000
1	0.1	1.0051	1.0050	0.0001
2	0.2	1.0207	1.0206	0.0001
3	0.3	1.0483	1.0481	0.0002
4	0.4	1.0902	1.0899	0.0003
5	0.5	1.1499	1.1494	0.0005
6	0.6	1.2323	1.2317	0.0006
7	0.7	1.3437	1.3429	0.0008
8	0.8	1.4924	1.4914	0.0010
9	0.9	1.6890	1.6879	0.0011
10	1.0	1.9471	1.9462	0.0009

■

The comparison of Example 6.10 with the results of Euler's method on the same problem in Example 6.2 is striking. In order to quantify the improvement that the trapezoid method brings toward solving IVP's, we need to calculate its local truncation error (23).

The local truncation error is the error made on a single step. Starting at an assumed correct solution point (t_i, y_i), the correct extension of the solution at t_{i+1} can be given by the Taylor expansion

$$y_{i+1} = y(t_i + h) = y_i + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{6}y'''(c), \tag{34}$$

for some number c between t_i and t_{i+1}, assuming y''' is continuous. In order to compare these terms with the trapezoid method, we will write them a little differently. From the differential equation $y'(t) = f(t, y)$, differentiate both sides with respect to t, using the chain rule:

$$\begin{aligned} y''(t) &= \frac{\partial f}{\partial t}(t, y) + \frac{\partial f}{\partial y}(t, y)y'(t) \\ &= \frac{\partial f}{\partial t}(t, y) + \frac{\partial f}{\partial y}(t, y)f(t, y) \end{aligned}$$

The new version of (34) is

$$y_{i+1} = y_i + hf(t_i, y_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t}(t_i, y_i) + \frac{\partial f}{\partial y}(t_i, y_i)f(t_i, y_i)\right) + \frac{h^3}{6}y'''(c), \tag{35}$$

We want to compare this expression to the explicit trapezoid method, using the two-dimensional Taylor theorem to expand the term

$$\begin{aligned} f(t_i + h, y_i + hf(t_i, y_i)) &= f(t_i, y_i) \\ &+ h\frac{\partial f}{\partial t}(t_i, y_i) + hf(t_i, y_i)\frac{\partial f}{\partial y}(t_i, y_i) \\ &+ O(h^2). \end{aligned}$$

The trapezoid method can be written

$$\begin{aligned} w_{i+1} &= y_i + \frac{h}{2}(f(t_i,y_i) + f(t_i+h, y_i + hf(t_i,y_i))) \\ &= y_i + \frac{h}{2}f(t_i,y_i) + \frac{h}{2}\left(f(t_i,y_i) + h\left(\frac{\partial f}{\partial t}(t_i,y_i) + f(t_i,y_i)\frac{\partial f}{\partial y}(t_i,y_i) + O(h^2)\right)\right) \\ &= y_i + hf(t_i,y_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t}(t_i,y_i) + f(t_i,y_i)\frac{\partial f}{\partial y}(t_i,y_i)\right) + O(h^3). \end{aligned} \tag{36}$$

Subtracting (36) from (35) gives the local truncation error as

$$y_{i+1} - w_{i+1} = O(h^3)$$

Theorem 6.4 shows that the global error of the trapezoid method is proportional to h^2, meaning that the method is of order two, compared with order one for Euler's method. For small h this is a significant difference, as shown by returning to Example 6.9.

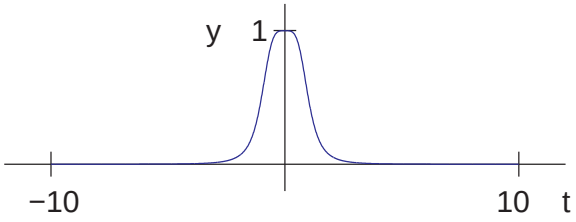


Figure 8: Approximation of Example 6.9 by the trapezoid method. Step size is $h = 10^{-3}$. Note the significant improvement in accuracy as compared to Euler's method in Figure 6.

Example 6.11 Apply the trapezoid method to the Example 6.9:

$$\begin{cases} y' = -4t^3y^2 \\ y(-10) = 1/10001 \\ t \in [-10, 0]. \end{cases}$$

SPOTLIGHT ON: ***Complexity***

Is a second-order method more efficient or less efficient than a first-order method? On each step, the error is smaller, but the computational work is greater, since ordinarily two function evaluations (of $f(t,y)$) are required instead of one. A rough comparison goes like this: Suppose an approximation has been run with step size h, and one wants to double the amount of computation to improve the approximation. For the same number of function evaluations, one can (a) halve the step size of the first order method, multiplying the global error by $1/2$, or (b) keep the same step size, but use a second order method, replacing the h in Theorem 6.4 by h^2, essentially multiplying the global error by h. For small h, (b) wins.

Revisiting Example 6.9 with a more powerful method shows a great improvement in approximating the solution, for example at $x = 0$. The correct value $y(0) = 1$ is attained within $.0015$ with a step size of $h = 10^{-3}$ with the trapezoid method, as shown in Figure 8. This is already better than Euler with a step size $h = 10^{-5}$. Using the trapezoid method with $h = 10^{-5}$ yields an error on the order of 10^{-7} for this relatively difficult initial value problem.

■

6.2.3 Taylor methods

So far we have learned two methods for approximating solutions of ordinary differential equations. The Euler method has order one, and the apparently superior trapezoid method has order two. In this section we show that methods of all orders exist. For each positive integer k, there is a Taylor method of order k, which we will describe next.

The basic idea is a straightforward exploitation of the Taylor expansion. Assume the solution $y(t)$ is $k + 1$ times continuously differentiable. Given the current point $(t, y(t))$ on the solution curve, the goal is to express $y(t + h)$ in terms of $y(t)$ for some stepsize h, using information about the differential equation. The Taylor expansion of $y(t)$ about t is

$$y(t+h) = y(t) + hy'(t) + \frac{1}{2}h^2y''(t) + \ldots + \frac{1}{k!}h^k y^{(k)}(t) + \frac{1}{(k+1)!}h^{k+1}y^{(k+1)}(c)$$

where c lies between t and $t + h$. The last term is the Taylor remainder term. For $k = 1$ the expansion is

$$\begin{aligned} y(t+h) &= y(t) + hy'(t) + O(h^2) \\ &= y(t) + hf(t,y) + O(h^2) \end{aligned}$$

where we have used the differential equation to replace $y'(t)$ with f. Setting $w_i = y(t)$ to be the current y position, we see that the method gives

$$w_{i+1} = w_i + hf(t_i, w_i),$$

with a local truncation error of order h^2. Therefore we recognize the first-order Taylor method as Euler's method.

For $k = 2$ we find a new method. The Taylor expansion is

$$\begin{aligned} y(t+h) &= y(t) + hy'(t) + \frac{1}{2}h^2y''(t) + \ldots + \frac{1}{3!}h^3y'''(c) \\ &= y(t) + hf(t,y) + \frac{1}{2}h^2f'(t,y) + O(h^3) \end{aligned}$$

where we have denoted the full t-derivative

$$f'(t,y) = f'(t,y(t)) = f_t(t,y) + f_y(t,y)y'(t).$$

We will use the notation f_t to denote the partial derivative of f with respect to t, and similarly for f_y. Setting the current y position to be $w_i = y(t)$, the method is

$$w_{i+1} = w_i + hf(t_i, w_i) + \frac{1}{2}h^2(f_t(t_i, w_i) + f_y(t_i, w_i)f(t_i, w_i)).$$

Example 6.12 Determine the second-order Taylor method for

$$\begin{cases} y' = ty + t^3 \\ y(0) = y_0 \end{cases} \tag{37}$$

Since $f(t, y) = ty + t^3$,

$$\begin{aligned} f'(t, y) &= f_t + f_y f \\ &= y + 3t^2 + t(ty + t^3), \end{aligned}$$

and the method gives

$$w_{i+1} = w_i + h(t_i w_i + t_i^3) + \frac{1}{2}h^2(w_i + 3t_i^2 + t_i(t_i w_i + t_i^3)).$$

■

The second-order Taylor method gives a second-order method, but notice that manual labor on the user's part was required, to determine the partial derivatives. Compare this to the other second-order method we have learned, where (33) requires only calls to a routine that computes values of $f(t, y)$ itself.

Conceptually, the lesson represented by Taylor methods is that ODE methods of arbitrary order exist. They can be derived by following the scheme shown above. However, they suffer from the problem that extra work is needed to compute the partial derivatives of f that show up in the formula. Since formulas of the same orders can be developed that don't require these partial derivatives, the Taylor methods are used only for specialized purposes.

Exercises 6.2

6.2.1. Compute the Euler's method error bound from Corollary 6.5 for the solution at $t = 1$, for the initial value problems of Exercise 6.1.5.

6.2.2. Write out the explicit trapezoid method for the IVPs in Exercise 6.1.4. Using stepsize $h = 1/4$, calculate the trapezoid method approximation on the interval $[0, 1]$. Compare to the correct solution found in Exercise 6.1.4, and find the total error at each step.

6.2.3. Carry out Exercise 6.2.2 for the IVPs in Exercise 6.1.5.

6.2.4. Find a general formula, similar to (37), for the third-order Taylor method.

6.2.5. Find the formula for the second-order Taylor method for the following differential equations. (a) $y' = ty$ (b) $y' = ty^2 + y^3$ (c) $y' = y \sin y$ (d) $y' = e^{yt^2}$
[Ans. (a) $w_{i+1} = w_i + ht_i w_i + \frac{1}{2}h^2(w_i + t_i^2 w_i)$ (b) $w_{i+1} = w_i + h(t_i w_i^2 + w_i^3) + \frac{1}{2}h^2(w_i^2 + (2t_i w_i + 3w_i^2)(t_i w_i^2 + w_i^3))$ (c) $w_{i+1} = w_i + hw_i \sin w_i + \frac{1}{2}h^2(\sin w_i + w_i \cos w_i)w_i \sin w_i$ (d) $w_{i+1} = w_i + he^{w_i t_i^2} + \frac{1}{2}h^2 e^{w_i t_i^2}(2t_i + t_i^2 e^{w_i t_i^2})$]

6.2.6. Same as Exercise 6.2.5, but determine the third-order Taylor method.

6.2.7. Find the formula for the second-order Taylor method applied to the initial value problems in Exercise 6.1.4. Using step size $h = 1/4$, calculate the second-order Taylor method approximation on the interval $[0, 1]$. Compare to the correct solution found in Exercise 6.1.4, and find the total error at each step.

6.2.8. (a) Prove (26). (b) Prove (27)

Computer Problems 6.2

6.2.1. Print the values of the explicit trapezoid method solution on a grid of step size $h = 0.1$ in $[0, 1]$ for the initial value problems in Exercise 6.1.4.

6.2.2. Plot the approximate solutions for the IVPs in Exercise 6.1.4 on $[0, 1]$ for step sizes $h = 0.1, 0.05$, and 0.025 along with the true solution.

6.2.3. For the IVP's in Exercise 6.1.4, plot the global error of the explicit trapezoid method at $t = 1$ as a function of $h = 0.1 \times 2^k$ for $0 \leq k \leq 5$. Use a semilog plot as in Figure 4.

6.2.4. Print the values of the second-order Taylor method solution on a grid of step size $h = 0.1$ in $[0, 1]$ for the initial value problems in Exercise 6.1.4.

6.3 Systems of ordinary differential equations.

APPROXIMATION of systems of differential equations can be done as a simple extension of the methodology for a single differential equation. Treating systems of equations greatly extends our ability to model interesting dynamical behavior. In addition, we will discuss the handling of higher-order equations. The **order** of a differential equation refers to the highest order derivative appearing in the equation.

A first-order system has form

$$\begin{aligned} y_1' &= f_1(t, y_1, \ldots, y_n) \\ y_2' &= f_2(t, y_1, \ldots, y_n) \\ &\vdots \\ y_n' &= f_n(t, y_1, \ldots, y_n) \end{aligned}$$

Example 6.13 *Apply Euler's method to the first-order system of two equations*

$$\begin{aligned} y_1' &= y_2^2 - 2y_1 \\ y_2' &= y_1 - y_2 - ty_2^2 \\ y_1(0) &= 0 \\ y_2(0) &= 1 \end{aligned} \tag{38}$$

First, check that the solution of the system (38) is the vector-valued function

$$\begin{aligned} y_1(t) &= te^{-2t} \\ y_2(t) &= e^{-t} \end{aligned}$$

For the moment, forget that we know the solution and apply Euler's method. The Euler's method formula is applied to each component in turn.

$$\begin{aligned} w_{i+1,1} &= w_{i,1} + h(w_{i,2}^2 - 2w_{i,1}) \\ w_{i+1,2} &= w_{i,2} + h(w_{i,1} - w_{i,2} - tw_{i,2}^2) \end{aligned}$$

Figure 9 shows the Euler method approximations of y_1 and y_2 along with the correct solution. Matlab code that carries this out is very similar to Program 6.1.

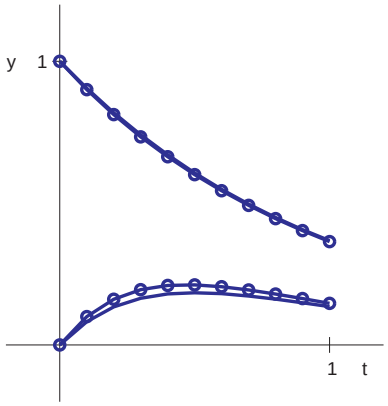


Figure 9: Equation (38) approximated by Euler method. Step size $h = 0.1$. The upper curve is $y_1(t)$, along with its approximate solution $w_{i,1}$ (circles), while the lower curve is $y_2(t)$ and $w_{i,2}$.

```
% Program 6.2  Vector version of Euler method
function euler2(int,y0,h)
% input interval [a,b], initial vector y0, step size h
% Example usage: euler2([0 1],[0 1],0.1);
a=int(1);b=int(2);
t(1)=0; y(1,:)=y0;
n=round((b-a)/h);
for i=1:n
  t(i+1)=t(i)+h;
  y(i+1,:)=eulerstep(t(i),y(i,:),h);
end
plot(t,y(:,1),t,y(:,2));


function y=eulerstep(t,x,h)
%one step of the Euler method
%Input: t is current time, x is current vector, h is stepsize
%Output: the approximate solution vector at time t+h
y=x+h*ydot(t,x);

function ydot=ydot(t,y)
ydot(1) = y(2)^2-2*y(1);
ydot(2) = y(1)-y(2)-t*y(2)^2;
```

■

6.3.1 Higher order equations

A single differential equation of higher order can be converted to a system. Let

$$y^{(n)} = f(t, y, y', y'', \ldots, y^{(n-1)}$$

be an nth-order ordinary differential equation. Define new variables

$$\begin{aligned} y_1 &= y \\ y_2 &= y' \\ y_3 &= y'' \\ &\vdots \\ y_n &= y^{(n-1)} \end{aligned}$$

and notice that the original differential equation can be written

$$y_n' = f(t, y_1, y_2, \ldots, y_n).$$

Together with the equations

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= y_4 \\ &\vdots \\ y_{n-1}' &= y_n, \end{aligned}$$

the nth-order differential equation can be converted to a system of first-order equations, which we can solve using methods like the Euler or trapezoid methods.

Example 6.14 Convert the third-order differential equation

$$y''' = a(y'')^2 - y' + yy'' + \sin t \tag{39}$$

to a system.

Set $y_1 = y$ and define the new variables

$$\begin{aligned} y_2 &= y' \\ y_3 &= y''. \end{aligned}$$

Then in terms of first derivatives

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= ay_3^2 - y_2 + y_1 y_3 + \sin t, \end{aligned} \tag{40}$$

and the solution $y(t)$ of the third-order equation (39) can be found by solving the system (40).

■

Because of the possibility of converting higher-order equations to systems, we will restrict our attention to systems of first-order equations. Note also that a system of higher-order equations can be converted in the same way to a system of first-order equations.

6.3.2 The pendulum

Figure 10 shows a pendulum swinging under the influence of gravity. Assume that the pendulum is hanging from a rigid rod that is free to swing through 360 degrees. Denote by y the angle of the pendulum with respect to the vertical, so that $y = 0$ corresponds to straight down. Therefore y and $y + 2\pi$ should be considered the same angle.

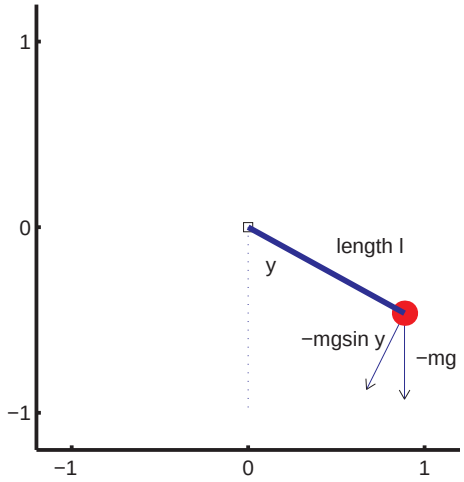


Figure 10: The pendulum. Component of force in the tangential direction is $F = -mg \sin y$, where y is the angle the pendulum bob makes with the vertical.

We will use Newton's law of motion $F = ma$ to find the pendulum equation. The motion of the pendulum bob is constrained to be along a circle of radius l, where l is the length of the pendulum rod. If y is measured in radians, then the component of acceleration tangent to the circle is ly'', because the component of position tangent to the circle is ly. The component of force along the direction of motion is $mg \sin y$. It is a restoring force, meaning that it is directed in the opposite direction from the displacement of the variable y. The differential equation governing the frictionless pendulum is therefore

$$ml y'' = F = -mg \sin y, \tag{41}$$

according to Newton's law of motion. This is a second-order differential equation for the angle y of the pendulum. The initial conditions are given by the initial angle $y(0)$ and angular velocity $y'(0)$.

By setting $y_1 = y$ and introducing the new variable $y_2 = y'$, the second-order equation is converted to a first-order system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\frac{g}{l} \sin y_1. \end{aligned} \tag{42}$$

If the pendulum is started from a position straight out to the right, the initial conditions are $y_1(0) = \pi/2$ and $y_2(0) = 0$. Using MKS units, the gravitational acceleration at the earth's surface is about 9.8m/sec^2. We assume below that the pendulum rod is 1 meter long. Using these parameters, we can test the ability of the Euler method as a solver for this system.

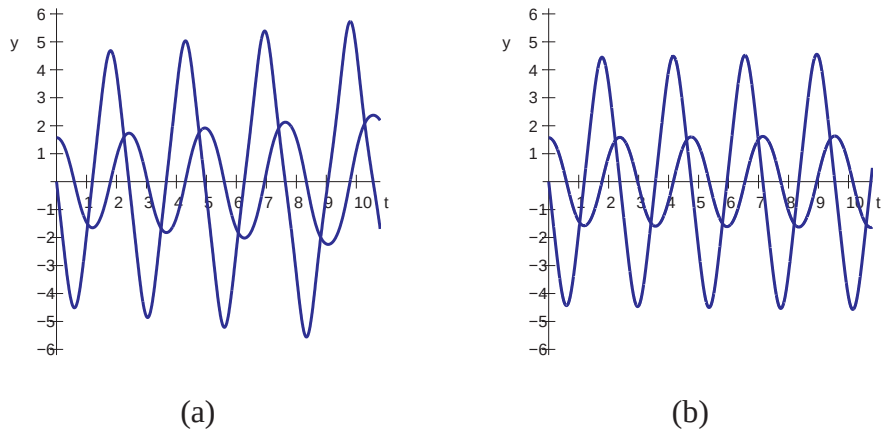


Figure 11: Euler method applied to the pendulum equations. The curve of smaller amplitude is the angle y_1 in radians; the curve of larger amplitude is the angular velocity y_2. (a) Stepsize $h = 0.01$ is too large; energy is growing. (b) Stepsize $h = 0.001$ shows more reasonable trajectories.

Figure 11 shows Euler's method approximations to the pendulum equations with two different step sizes. The smaller curve represents the angle y as a function of time, and the larger amplitude curve is the instantaneous angular velocity. Note that the zeros of the angle, representing the vertical position of the pendulum, correspond to the largest angular velocity, positive or negative. The pendulum is travelling fastest as it swings through the lowest point. When the pendulum is extended to the far right, the peak of the smaller curve, the velocity is zero as it turns from positive to negative.

The inadequacy of Euler's method is apparent in Figure 11. The step size $h = 0.01$ is clearly too large to get even the qualitative parts correct. An undamped pendulum started with zero velocity should swing back and forth forever, returning to its starting position with a regular periodicity. The amplitude of the angle in Figure 11(a) is growing, which cannot be correct. Using 10 times more steps, as in Figure 11(b), improves the situation at least visually, but a total of 10^4 steps are needed, an extreme number for the routine dynamical behavior shown by the pendulum.

A second-order ODE solver like the trapezoid method can help. We will rewrite the Matlab code to use the trapezoid method, and take the opportunity to illustrate the ability of Matlab to do simple animations.

The following code `pend.m` contains the same differential equation information, but `eulerstep` is replaced by `trapstep`. In addition, the variables `rod` and `bob` are introduced to represent the rod and pendulum bob, respectively. The Matlab `set` command assigns attributes to variables. The `drawnow` command plots the `rod` and `bob` variables. Note that the erase mode of both variables is set to `xor`, meaning that when the plotted variable is redrawn somewhere else, the previous position is erased. Figure 10 is a screen shot of the animation.

```
% Program 6.3 Animation program for pendulum using IVP solver
function pend(int,ic,h,p)
% Inputs: int = [a b] time interval,
% ic = [y(1,1) y(1,2)], initialize
% h = stepsize, p = steps per point plotted
% Calls a one-step method such as trapstep.m
```

```
% Example usage: pend([0 10],[pi/2 0],.05,1)
clf                       % clear figure window
a=int(1);b=int(2);n=ceil((b-a)/(h*p)); % plot n points in total
y(1,:)=ic;                % enter initial conds in y
t(1)=a;
set(gca,'XLim',[-1.2 1.2],'YLim',[-1.2 1.2], ...
  'XTick',[-1 0 1],'YTick',[-1 0 1], ...
  'Drawmode','fast','Visible','on','NextPlot','add');
cla;            % clear screen
plot(0,0,'ks')  % pivot where rod attached
axis square     % make aspect ratio 1 - 1
bob = line('color','r','Marker','.','markersize',40,'erase','xor',...
  'xdata',[],'ydata',[]);
rod = line('color','b','LineStyle','-','LineWidth',3,'erase','xor',...
  'xdata',[],'ydata',[]);
for k=1:n
  for i=1:p
    t(i+1) = t(i)+h;
    y(i+1,:) = trapstep(t(i),y(i,:),h);
  end
  y(1,:) = y(p+1,:);t(1)=t(p+1);
  xbob = cos(y(1,1)-pi/2); ybob = sin(y(1,1)-pi/2);
  xrod = [0 xbob]; yrod = [0 ybob];
  set(rod,'xdata',xrod,'ydata',yrod)
  set(bob,'xdata',xbob,'ydata',ybob)
  drawnow; pause(h)
end

function y = trapstep(t,x,h)
%one step of the trapezoid method
z1=ydot(t,x);
g=x+h*z1;
z2=ydot(t+h,g);
y=x+h*(z1+z2)/2;

function ydot=ydot(t,y)
g=9.8;length=1;
ydot(1) = y(2);
ydot(2) = -(g/length)*sin(y(1));
```

Example 6.15 The damped pendulum.

The force of damping, such as air resistance or friction, is often modelled as being proportional and in the opposite direction to velocity. The pendulum equation becomes

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\frac{g}{l}\sin y_1 - d y_2, \end{aligned} \tag{43}$$

where $d > 0$ is the damping coefficient. Unlike the undamped pendulum above, this one will lose energy through damping and with time approach the limiting equilibrium solution $y_1 = y_2 = 0$, from any initial condition. Computer Problem 6.3.3 asks you to run a damped version of `pend.m`.

■

Example 6.16 The forced damped pendulum.

Adding a time-dependent term to (43) represents outside forcing on the damped pendulum. Consider adding the sinusoidal term $f\sin(t)$ to the y_2' right-hand-side, yielding

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\frac{g}{l}\sin y_1 - dy_2 + f\sin t, \end{aligned} \tag{44}$$

This can be considered as a model of a pendulum that is affected by an oscillating magnetic field, for example.

A host of new dynamical behavior becomes possible when forcing is added. For a two-dimensional autonomous system of differential equations, the Poincare-Bendixson Theorem from the theory of differential equations says that trajectories can tend toward only two types of limiting behavior: stable equilibria like the down position of the pendulum, or stable periodic cycles like the pendulum swinging back and forth forever. The forcing makes the system non-autonomous (it can be rewritten as a three-dimensional autonomous system, but not two-dimensional) so that a third type of trajectories are allowed: chaotic trajectories.

Setting the damping coefficient to $d = 1$ and the forcing coefficient to $f = 10$ results in interesting periodic behavior, explored in Computer Problem 6.3.4. Moving the parameter to $f = 15$ introduces chaotic trajectories.

■

Example 6.17 The double pendulum.

The program `pend.m` can be adapted to make an animation of the double pendulum, which is a single pendulum with another pendulum hanging from the bob of the first pendulum. If y_1 and y_3 are the angles of the two bobs with respect to the vertical, the system of differential equations is

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{-3g\sin y_1 - g\sin(y_1 - 2y_3) - 2\sin(y_1 - y_3)(y_4^2 - y_2^2\cos(y_1 - y_3))}{3 - \cos(2y_1 - 2y_3)} - dy_2 \\ y_3' &= y_4 \\ y_4' &= \frac{2\sin(y_1 - y_3)[2y_2^2 + 2g\cos y_1 + y_4^2\cos(y_1 - y_3)]}{3 - \cos(2y_1 - 2y_3)} \end{aligned}$$

The parameter d represents friction at the pivot. If $d > 0$, the pendulum will eventually move toward the down position. The double pendulum is believed to be chaotic for $d = 0$.

```
%Program 6.? Animation program for double pendulum
function pend2(int,ic,h,p)
%Inputs: int = [a b] time interval,
%ic = [y(1,1) y(1,2) y(1,3) y(1,4)], initialize
%h = stepsize, p = steps per point plotted
%Calls a one-step method such as trapstep.m
%Example usage: pend2([0 100],[pi/2 0 pi/2 0],.01,5)
clf                       % clear figure window
a=int(1);b=int(2);n=ceil((b-a)/(h*p)); % plot n points in total
```

```
y(1,:)=ic;                % enter initial conds in y
t(1)=a;
set(gca,'XLim',[-2.2 2.2],'YLim',[-2.2 2.2], ...
  'XTick',[-2 0 2],'YTick',[-2 0 2], ...
  'Drawmode','fast','Visible','on','NextPlot','add');
cla;            % clear screen
axis square     % make aspect ratio 1 - 1
plot(0,0,'ks')  % pivot where rod attached
bob1 =line('color','r','Marker','.','markersize',40,'erase','xor',...
  'xdata',[],'ydata',[]);
rod1 =line('color','b','LineStyle','-','LineWidth',3,'erase','xor',...
  'xdata',[],'ydata',[]);
bob2 =line('color','g','Marker','.','markersize',40,'erase','xor',...
  'xdata',[],'ydata',[]);
rod2 =line('color','b','LineStyle','-','LineWidth',3,'erase','xor',...
  'xdata',[],'ydata',[]);
for k=1:n
  for i=1:p
    t(i+1) = t(i)+h;
    y(i+1,:) = trapstep(t(i),y(i,:),h);
  end
  y(1,:) = y(p+1,:);t(1)=t(p+1);
  xbob1 = cos(y(1,1)-pi/2); ybob1 = sin(y(1,1)-pi/2);
  xbob2 = xbob1+cos(y(1,3)-pi/2); ybob2 = ybob1+sin(y(1,3)-pi/2);
  xrod1 = [0 xbob1]; yrod1 = [0 ybob1];
  xrod2 = [xbob1 xbob2]; yrod2 = [ybob1 ybob2];
  set(rod1,'xdata',xrod1,'ydata',yrod1)
  set(bob1,'xdata',xbob1,'ydata',ybob1)
  set(rod2,'xdata',xrod2,'ydata',yrod2)
  set(bob2,'xdata',xbob2,'ydata',ybob2)
  drawnow; pause(h)
end

function y = trapstep(t,x,h)
%one step of the trapezoid method
z1=ydot(t,x);
g=x+h*z1;
z2=ydot(t+h,g);
y=x+h*(z1+z2)/2;

function ydot=ydot(t,y)
g=9.8;length=1;
a=y(1)-y(3);
ydot(1) = y(2);
ydot(2)=-3*g*sin(y(1))-g*sin(y(1)-2*y(3))-2*sin(a)*(y(4)^2-y(2)^2*cos(a));
ydot(2) = ydot(2)/(3-cos(2*a)) - .003*y(2);
ydot(3) = y(4);
ydot(4) = (2*sin(a)*(2*y(2)^2+2*g*cos(y(1))+y(4)^2*cos(a)))/(3-cos(2*a));
```

■

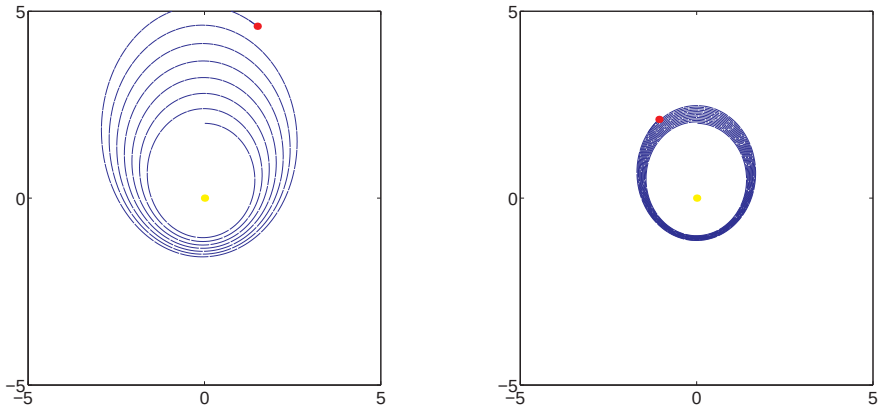
6.3.3 Orbital mechanics

As a first example we discuss the one-body problem of an orbiting satellite. Newton's second law of motion says that the acceleration a of the satellite is related to the force F applied to the satellite

as $F = ma$, where m is the mass. The law of gravitation expresses the force on a body of mass m_1 by a body of mass m_2 by an inverse-square law

$$F = \frac{gm_1m_2}{r^2}$$

where r is the distance separating the masses. In the one-body problem, one of the masses is considered negligible compared to the other, as in the case of a small satellite orbiting a large planet. This simplification allows us to neglect the force of the satellite on the planet, so that the planet may be regarded as fixed.



(a) (b)

Figure 12: Euler on one-body problem. (a) $h = 0.01$ and (b) $h = 0.001$

Place the large mass at the origin and denote by (x, y) the position of the satellite. The distance between the masses is $r = \sqrt{x^2 + y^2}$, and the force on the satellite is central, meaning in the direction of the large mass. The direction vector, a unit vector in this direction, is

$$\left(-\frac{x}{\sqrt{x^2 + y^2}}, -\frac{y}{\sqrt{x^2 + y^2}} \right).$$

Therefore the force on the satellite in terms of components is

$$(F_x, F_y) = \left(\frac{gm_1m_2}{x^2 + y^2} \frac{-x}{\sqrt{x^2 + y^2}}, \frac{gm_1m_2}{x^2 + y^2} \frac{-y}{\sqrt{x^2 + y^2}} \right) \tag{45}$$

Inserting these forces into Newton's law of motion yields the two second-order equations

$$\begin{aligned} m_1x'' &= -\frac{gm_1m_2x}{(x^2 + y^2)^{3/2}} \\ m_1y'' &= -\frac{gm_1m_2y}{(x^2 + y^2)^{3/2}} \end{aligned}$$

Introducing the variables $v_x = x'$ and $v_y = y'$ allows reduction of the two second-order equations to a system of four first-order equations

$$\begin{aligned} x' &= v_x \\ {v_x}' &= -\frac{gm_2 x}{(x^2+y^2)^{3/2}} \\ y' &= v_y \\ v_y' &= -\frac{gm_2 y}{(x^2+y^2)^{3/2}} \end{aligned} \tag{46}$$

Applying Euler's method to a system of first-order equations is done componentwise: by using Euler separately on each component. The Matlab code for one Euler step of (46) looks exactly like the code for the scalar version, except that the input x is a vector.

We have written Euler's method as a function to be called from a driver program, so that we can substitute more sophisticated one-step methods later. The following driver program called `orbit.m` calls `eulerstep.m` and sequentially plot the results in the Matlab plotting window.

```
%Program 6.4 Plotting program for one-body problem by IVP solver
function z=orbit(int,ic,h,p)
%Inputs: int = [a b] time interval,
% ic = [x0 vx0 y0 vy0], initialize x position, x velocity, y pos, y vel
% h = stepsize, p = steps per point plotted
%Calls a one-step method such as eulerstep.m
%Example usage: orbit([0 100],[0 1 2 0],.01,50)
a=int(1);b=int(2);n=ceil((b-a)/(h*p)); % plot n points in total
x0=ic(1);vx0=ic(2);y0=ic(3);vy0=ic(4); % grab initial conditions
y(1,:)=[x0 vx0 y0 vy0];t(1)=a;         % enter initial conds in y
set(gca, ...
  'XLim',[-5 5],'YLim',[-5 5], ...
  'XTick',[-5 0 5],'YTick',[-5 0 5], ...
  'Drawmode','fast', ...
  'Visible','on', ...
  'NextPlot','add');
cla;
sun=line('color','y', ...
  'Marker','.', ...
  'markersize',25,...
  'xdata',0,'ydata',0);
drawnow;
head = line( ...
  'color','r', ...
  'Marker','.', ...
  'markersize',25, ...
  'erase','xor', ...
  'xdata',[],'ydata',[]);
tail=line( ...
  'color','b', ...
  'LineStyle','-', ...
  'erase','none', ...
  'xdata',[],'ydata',[]);
%[px,py,button]=ginput(1);       % include these three lines
%[px1,py1,button]=ginput(1);     % to enable two mouse clicks
%y(1,:)=[px px1-px py py1-py];   % for setting initial conditions
```

```
for k=1:n
  for i=1:p
    t(i+1)=t(i)+h;
    y(i+1,:)=eulerstep(t(i),y(i,:),h);
  end
  y(1,:)=y(p+1,:);t(1)=t(p+1);
  set(head,'xdata',y(1,1),'ydata',y(1,3))
  set(tail,'xdata',y(2:p,1),'ydata',y(2:p,3))
  drawnow;
end

function y=eulerstep(t,x,h)
%one step of the Euler method
y=x+h*ydot(t,x);

function ydot = ydot(t,x)
m2=3;g=1;mg2=m2*g;px2=0;py2=0;
px1=x(1);py1=x(3);vx1=x(2);vy1=x(4);
dist=sqrt((px2-px1)^2+(py2-py1)^2);
ydot=zeros(1,4);
ydot(1)=vx1;
ydot(2)=(mg2*(px2-px1))/(dist^3);
ydot(3)=vy1;
ydot(4)=(mg2*(py2-py1))/(dist^3);
```

Running the Matlab script `orbit.m` immediately shows the limitations of Euler's method for approximating interesting problems. Figure 12(a) shows the outcome of running `orbit([0 100],[0 1 2 0],.01,50)`. That means we follow the orbit over the time interval $[a, b] = [0, 100]$, the initial position is $(x_0, y_0) = (0, 2)$, the initial velocity is $(v_x, v_y) = (1, 0)$, the stepsize is $h = 0.01$, and the current position is plotted once every $p = 50$ steps.

Solutions to the one-body problem must be conic sections, either ellipses, parabolas, or hyperbolas. The spiral seen in Figure 12(a) is a numerical artifact, meaning a misrepresentation caused by errors of computation. In this case, it is the truncation error of Euler's method that leads to the failure of the orbit to close up into an ellipse. If the stepsize is cut by a factor of ten to $h = 0.001$, the result is improved as shown in Figure 12(b). Several orbits are shown, and it is clear that even with the greatly decreased stepsize, the accumulated error is noticeable.

Corollary 6.5 says that the Euler method, in principle, can approximate a solution with as much accuracy as desired, if the step size h is sufficiently small. However, results like Figures 6 and 12 show that the method is seriously limited in practice.

Figure 13 shows the clear improvement in the one-body problem resulting from the replacement of the Euler step with the trapezoid step. The plot was made by replacing the function `eulerstep.m` by `trapstep.m` in the above code.

Exercises 6.3

6.3.1. Apply four steps of the Euler method with $h = 1/4$ to the initial value problem.

(a) $\begin{cases} y_1' = y_1 + y_2 \\ y_2' = -y_1 + y_2 \\ y_1(0) = 1 \\ y_2(0) = 0 \end{cases}$ (b) $\begin{cases} y_1' = -y_1 - y_2 \\ y_2' = -y_1 + y_2 \\ y_1(0) = 1 \\ y_2(0) = 0 \end{cases}$ (c) $\begin{cases} y_1' = -y_2 \\ y_2' = y_1 \\ y_1(0) = 1 \\ y_2(0) = 0 \end{cases}$ (d) $\begin{cases} y_1' = y_1 + 3y_2 \\ y_2' = 2y_1 + 2y_2 \\ y_1(0) = 5 \\ y_2(0) = 0 \end{cases}$

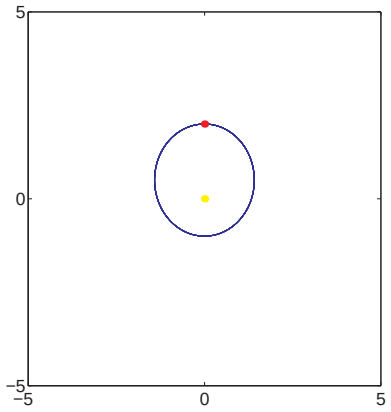


Figure 13: One-body problem approximated by the trapezoid method. Step size $h = 0.01$. The orbit appears to close, at least to the resolution visible in the plot.

Compare your answer at $t = 1$ with the value of the correct solutions (a) $y_1(t) = e^t \cos t, y_2(t) = -e^t \sin t$ (b) $y_1(t) = e^{-t} \cos t, y_2(t) = e^{-t} \sin t$ (c) $y_1(t) = \cos t, y_2(t) = \sin t$ (d) $y_1(t) = 3e^{-t} + 2e^{4t}, y_2(t) = -2e^{-t} + 2e^{4t}$.

6.3.2. Apply four steps of the trapezoid method with $h = 1/4$ to the initial value problems in Exercise 6.3.1. Compare answers with the correct solutions.

6.3.3. Convert the higher-order ordinary differential equation to a first-order system of equations.

(a) $y'' - ty = 0$ (Airy's equation) (b) $y'' - 2ty' + 2y = 0$ (Hermite's equation) (c) $y'' - ty' - y = 0$

6.3.4. Apply four steps of the Euler method with $h = 1/4$ to the initial value problems in 6.3.3, using $y(0) = y'(0) = 1$.

6.3.5. (a) Show that $y(t) = (e^t + e^{-t} - t^2)/2 - 1$ is the solution of the initial value problem $y''' - y' = t$ with $y(0) = y'(0) = y''(0) = 0$. (b) Convert the differential equation to a system of three first-order equations. (c) Use Euler's method with step size $h = 1/4$ to approximate the solution on $[0, 1]$. (d) Compare your approximate solution at $t = 1$ with the correct solution.

Computer Problems 6.3

6.3.1. Apply Euler's method with step sizes $h = 0.1$ and $h = 0.01$ to the initial value problems in Exercise 6.3.1. Find the solution on $[0, 1]$, and compare the solution vector at $t = 1$ with the correct value to find the total error. How much better is the error for the smaller step size? Does it correspond to the difference predicted by the order of Euler's method?

6.3.2. Repeat Computer Problem 6.3.1, but use the trapezoid method.

6.3.3. Adapt `pend.m` to accept damping coefficients. Run the resulting code with $d = 0.1$. Except for the initial condition $y_1(0) = \pi, y_2(0) = 0$, all trajectories move toward the straight down position as time progresses. Check the exceptional initial condition. What does the theory say, and what does the program do? Explain any differences.

6.3.4. Adapt `pend.m` to build a forced, damped version of the pendulum. Run the code with $d = 1$ in the following. (a) Set the forcing parameter $f = 10$. After moving through some temporary, transient behavior, the pendulum will settle into a periodic trajectory. Describe this trajectory qualitatively. Try different initial conditions. Do all solutions end up at the same "attracting" periodic trajectory? (b) Set $f = 12$. There are now two periodic attractors, that are mirror images of one another. Describe the two attracting trajectories, and find two initial conditions $(y_1, y_2) = (a, 0)$ and $(b, 0)$ where $|a - b| < 0.1$ that are attracted to different periodic trajectories. (c) Set $f = 15$ to view chaotic motion of the forced damped pendulum.

6.3.5. Adapt `pend.m` to build a damped pendulum with oscillating pivot. The goal is to investigate the phenomenon of parametric resonance, by which the inverted (rigid) pendulum becomes stable! The equation is

$$y'' + (\frac{g}{l} + dy' + a\cos 2\pi t)\sin y = 0,$$

where a is the forcing strength. Set $d = 0.1$ and the length of the pendulum to be 2.5 meters. In the absence of forcing $a = 0$, the downward pendulum $y = 0$ is a stable equilibrium and the inverted pendulum $y = \pi$ is an unstable equilibrium. Find as accurately as possible the range of parameter a for which the inverted pendulum becomes stable. (Of course, $a = 0$ is too small; it turns out that $a = 30$ is too large.) Use the initial condition $y = 3.1$ for your test, and call the inverted position "stable" if the pendulum does not pass through the downward position.

6.3.6. Use the parameter settings of Computer Problem 6.3.5 to demonstrate the other effect of parametric resonance: the stable equilibrium can become unstable with an oscillating pivot. Find the smallest (positive) value of the forcing strength a for which this happens. Classify the downward position as unstable if the pendulum eventually travels to the inverted position.

6.4 Runge-Kutta methods and applications.

THE Runge-Kutta methods are a family of methods that include the Euler and trapezoid methods, and also more sophisticated methods of higher order. We have seen that Euler has order one and the trapezoid method has order two. Another order two method of the Runge-Kutta type is the

Midpoint method

$$\begin{aligned} w_0 &= y_0 \\ w_{i+1} &= w_i + hf(t_i + \frac{h}{2}, w_i + \frac{h}{2}f(t_i, w_i)). \end{aligned} \tag{47}$$

To verify the order of the Midpoint method we must compute its local truncation error. When we did this for the trapezoid method, we found the expression (35) useful:

$$y_{i+1} = y_i + hf(t_i, y_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t}(t_i, y_i) + \frac{\partial f}{\partial y}(t_i, y_i)f(t_i, y_i)\right) + \frac{h^3}{6}y'''(c), \tag{48}$$

To compute the local truncation error at step i, we assume $w_i = y_i$ and calculate $y_{i+1} - w_{i+1}$. Repeating the use of the Taylor series expansion as for the trapezoid method, we can write

$$\begin{aligned} w_{i+1} &= y_i + hf(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i)) \\ &= y_i + h\left(f(t_i, y_i) + \frac{h}{2}\frac{\partial f}{\partial t}(t_i, y_i) + \frac{h}{2}f(t_i, y_i)\frac{\partial f}{\partial y}(t_i, y_i)\right). \end{aligned} \tag{49}$$

Comparing (48) and (49) yields

$$y_{i+1} - w_{i+1} = O(h^3)$$

and so the Midpoint method is of order two by Theorem 6.4.

Each function evaluation of the right-hand-side of the differential equations, corresponding to an s_i above, is called a **stage** of the method. Both the Trapezoid and Midpoint methods are two-stage, second-order Runge-Kutta methods.

In fact, the Trapezoid and Midpoint methods are just members of the family of two-stage, second-order Runge-Kutta methods, having form

$$w_{i+1} = w_i + h(1 - \frac{1}{2\alpha})f(t_i, w_i) + \frac{h}{2\alpha}f(t_i + \alpha h, w_i + \alpha h f(t_i, w_i)). \tag{50}$$

for some $\alpha \neq 0$. Setting $\alpha = 1$ corresponds to the explicit trapezoid method, and $\alpha = 1/2$ to the midpoint method. Exercise 6.4.5 asks you to verify the order of methods in this family.

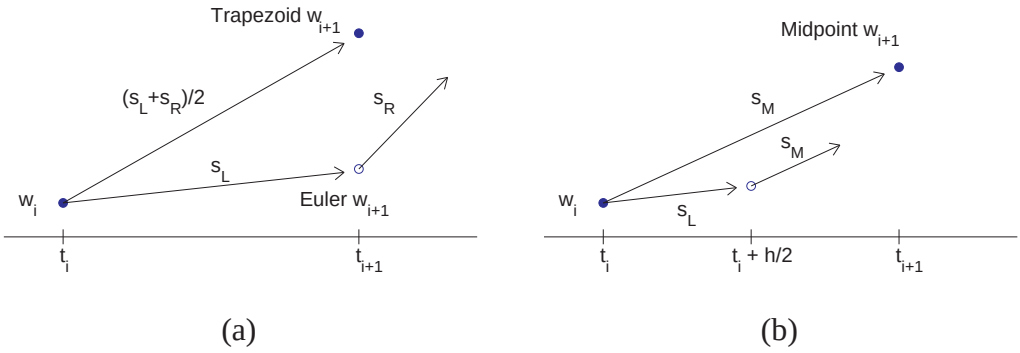


Figure 14: Schematic view of two members of the RK2 family. (a) The Trapezoid method uses an average from the left and right endpoints to traverse the interval. (b) The Midpoint method uses a slope from the interval midpoint

Figure 14 illustrates the intuition behind the Trapezoid and Midpoint methods. The Trapezoid method uses an Euler step to the right endpoint of the interval, evaluates the slope there, and then averages with the slope from the left endpoint. The Midpoint method uses an Euler step to the midpoint of the interval, evaluates the slope there from $f(t_i + h/2, w_i + (h/2)f(t_i, w_i))$, and uses that slope to move from w_i to the new approximation w_{i+1}. The methods use different approaches to solving the same problem: acquiring a slope that represents the entire interval better than the Euler method, which uses only the slope estimate from the left end of the interval.

There are Runge-Kutta methods of all orders. A particularly popular method is the

Runge-Kutta method of order four (RK4)

$$w_{i+1} = w_i + \frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4) \tag{51}$$

where

$$\begin{aligned} s_1 &= f(t_i, w_i) \\ s_2 &= f(t_i + \frac{h}{2}, w_i + \frac{h}{2}s_1) \\ s_3 &= f(t_i + \frac{h}{2}, w_i + \frac{h}{2}s_2) \\ s_4 &= f(t_i + h, w_i + hs_3) \end{aligned}$$

This method is the workhorse of quick and dirty ODE solvers because of its simplicity and ease of

SPOTLIGHT ON: ***Convergence***

The convergence properties of a fourth-order method, like RK4, are far superior to those of the order 1 and 2 methods we have discussed so far. Convergence here means how fast the (global) error of the ODE approximation at some fixed time t goes to zero as the stepsize h goes to zero. Fourth order means that for every halving of the stepsize, the error drops by approximately a factor of $2^4 = 16$.

programming. It is a one-step method, so that it requires only an initial condition to get started, yet as an order four method is much more accurate that either the Euler or trapezoid method.

The quantity $\frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4)$ in the fourth-order Runge-Kutta method takes the place of slope in the Euler method. This quantity can be considered as an improved guess for the slope of the solution in the interval $[t_i, t_i + h]$. Note that s_1 is the slope at the left end of the interval, s_2 is the slope used in the midpoint method, s_3 is an improved slope at the midpoint, and s_4 is an approximate slope at the right-hand endpoint $t_i + h$. The algebra needed to prove that this method is order four is similar to our derivation of the trapezoid and midpoint methods, but a bit lengthy, and can be found for example in [Henrici].

6.4.1 Classical examples

In this subsection we present two examples of both historical and ongoing interest. Computers were in their early development stages in the middle of the twentieth century. Some of the first applications were to help solve hitherto intractable systems of differential equations. In so doing, Hodgkin and Huxley essentially began the field of computational neuroscience, and Edward Lorenz first glimpsed in meteorological models what later became known as chaos.

A landmark in the history of neuroscience was the development of a realistic firing model for nerve cells, or neurons. The originators of the model, Hodgkin and Huxley, won the Nobel Prize in Biology in 1963. The model is a system of four coupled differential equations, one of which models the voltage difference between the interior and exterior of the cell. The three other equations model activation levels of ion channels, which do the work of exchanging sodium and potassium ions between the inside and outside. The **Hodgkin-Huxley equations** are:

$$\begin{aligned} Cv' &= -g_1 m^3 h(v - E_1) - g_2 n^4 (v - E_2) - g_3(v - E_3) + I_{\text{in}} \\ m' &= \alpha_m(v - E_0)(1 - m)\beta_m(v - E_0)m \\ n' &= \alpha_n(v - E_0)(1 - m)\beta_n(v - E_0)n \\ h' &= \alpha_h(v - E_0)(1 - m)\beta_h(v - E_0)h, \end{aligned} \tag{52}$$

where

$$\alpha_m(v) = \frac{2.5 - 0.1v}{e^{2.5-0.1v} - 1}, \beta_m(v) = 4e^{\frac{v}{18}},$$

$$\alpha_n(v) = \frac{0.1 - 0.01v}{e^{1-0.1v} - 1}, \beta_n(v) = \frac{1}{8}e^{\frac{v}{80}},$$

and where

$$\alpha_h(v) = 0.07e^{-\frac{v}{20}}, \beta_h(v) = \frac{1}{e^{3-0.1v}+1}.$$

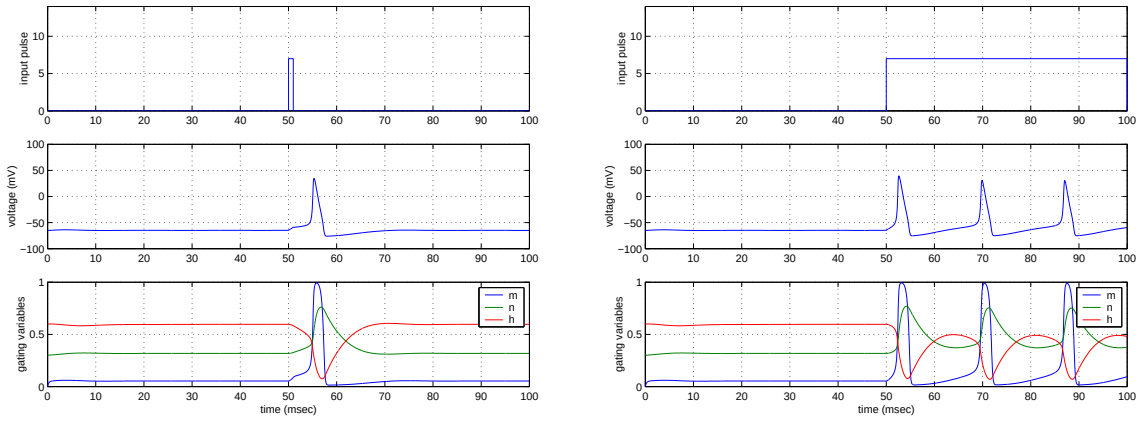
The coefficient C denotes the capacitance of the cell, and I_{in} denotes the input current from other cells. Typical coefficient values are $C = 1$ (capacitance, in microFarads), $g_1 = 120, g_2 = 36, g_3 = 0.3$ (conductances), and $E_0 = -65, E_1 = 50, E_2 = -77, E_3 = -54.4$ (voltages, in millivolts).

The v' equation is an equation of current per unit area, in units of milliamperes/cm^2, while the three other activations m, n, and h are unitless. The coefficient C is the capacitance of the neuron membrane, g_1, g_2, g_3 are conductances, and E_1, E_2 and E_3 are the "reversal potentials", which are the voltage levels that form the boundary between current flowing inward and outward.

Hodgkin and Huxley carefully chose the form of the equations to match experimental data, which was acquired from the squid giant axon. They also fit parameters to the model. Although the particulars of the squid axon differ from mammal neurons, the model has held up in general terms as a realistic depiction of neural dynamics. More generally, it is useful as an example of excitable media that translates continuous input into an all-or-nothing response.

```
% Program 6.5 Plotting program for Hodgkin-Huxley by IVP solver
function hh
% [a b] time interval,
% ic = initial voltage v, gating variables m, n, h
% h = stepsize
% Calls a one-step method such as rk4step.m
% Example usage: hh
global pa pb pulse
inp=input('square pulse start, square pulse end, muamps in [ ], e.g. [50 51 7]: ');
pa=inp(1);pb=inp(2);pulse=inp(3);
ic=[-65 0 .3 .6];
h=.05;p=10; %p steps per point plotted
a=0;b=100;n=ceil((b-a)/h); % plot n points in total
y(1,:)=ic;                % enter initial conds in y
t(1)=a;

for i=1:n
  t(i+1)=t(i)+h;
  y(i+1,:)=rk4step(t(i),y(i,:),h);
end
subplot(3,1,1);
plot([a pa pa pb pb b],[0 0 pulse pulse 0 0]);
grid;axis([0 100 0 2*pulse])
ylabel('input pulse')
subplot(3,1,2);
plot(t,y(:,1));grid;axis([0 100 -100 100])
ylabel('voltage (mV)')
subplot(3,1,3);
plot(t,y(:,2),t,y(:,3),t,y(:,4));grid;axis([0 100 0 1])
ylabel('gating variables')
legend('m','n','h')
xlabel('time (msec)')


function y=rk4step(t,w,h)
%one step of the Runge-Kutta order 4 method
s1=ydot(t,w);
```

```
s2=ydot(t+h/2,w+h*s1/2);
s3=ydot(t+h/2,w+h*s2/2);
s4=ydot(t+h,w+h*s3);
y=w+h*(s1+2*s2+2*s3+s4)/6;

function ydot = ydot(t,w)
global pa pb pulse
c=1;g1=120;g2=36;g3=0.3;T=(pa+pb)/2;len=pb-pa;
e0=-65;e1=50;e2=-77;e3=-54.4;
in=pulse*(1-sign(abs(t-T)-len/2))/2;
% square pulse input on interval [pa,pb] of pulse muamps
v=w(1);m=w(2);n=w(3);h=w(4);
ydot=zeros(1,4);
ydot(1)=(in - g1*m*m*m*h*(v-e1) - g2*n*n*n*n*(v-e2) - g3*(v-e3))/c;
v = v-e0; %modern convention
ydot(2)=(1-m)*(2.5-0.1*v)/(exp(2.5-0.1*v)-1) - m*4*exp(-v/18);
ydot(3)=(1-n)*(0.1-0.01*v)/(exp(1-0.1*v)-1) - n*0.125*exp(-v/80);
ydot(4)=(1-h)*0.07*exp(-v/20) - h/(exp(3-0.1*v)+1);
```



(a) (b)

Figure 15: Screenshots of Hodgkin-Huxley program. (a) Square wave input of size $I_{\text{in}} = 7\ \mu$A at time 50 msecs, 1 msec duration, causes the model neuron to fire once. (b) Sustained square wave, with $I_{\text{in}} = 7\ \mu$A causes the model neuron to fire periodically.

Without input, the Hodgkin-Huxley neuron stays quiescent, at a voltage of approximately E_0. Setting I_{in} to be a square current pulse of length 1 msec and strength 7 microamps is sufficient to cause a spike, a large depolarizing deflection of the voltage. This is illustrated in Figure 15. Run the program to check that $6.9\ \mu$A is not sufficient to cause a full spike. It is this property of greatly magnifying the effect of small differences in input that may explain the neuron's success at information processing. Figure 15(b) shows that if the input current is sustained, the neuron will fire a periodic volley of spikes.

The Lorenz equations are a simplification of a miniature atmosphere model that was designed to study Rayleigh-Bénard convection, the movement of heat in a fluid like air from a lower warm medium (such as the ground) to a higher cool medium (like the upper atmosphere). In this model of

a two-dimensional atmosphere, a circulation of air develops that can be described by the system of three equations

$$\begin{aligned} x' &= -sx + sy \\ y' &= -xz + rx - y \\ z' &= xy - bz, \end{aligned} \tag{53}$$

called the **Lorenz equations**. The variable x denotes the clockwise circulation velocity, y measures the temperature difference between the ascending and descending columns of air, and z measures the deviation from a strictly linear temperature profile in the vertical direction. The Prandtl number s, the Reynolds number r, and b are parameters of the system. The most common setting for the parameters is $s = 10, r = 28$, and $b = 8/3$, which results in the trajectory shown in Figure 16.

```
function ydot=ydot(t,y)
%Lorenz equations
s=10; r=28; b=8/3;
ydot(1) = -s*y(1)+s*y(2);
ydot(2) = -y(1)*y(3)+r*y(1)-y(2)
ydot(3) = y(1)*y(2) - b*y(3)
```

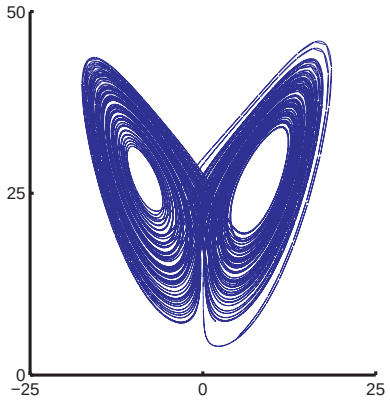


Figure 16: One trajectory of the Lorenz equations (53), projected to the xz-plane. Parameters are set to $s = 10, r = 28$, and $b = 8/3$.

The Lorenz equations are an important example because the trajectories show great complexity, despite the fact that the equations are deterministic, and fairly simple (almost linear). The explanation for the complexity is often called **sensitive dependence on initial conditions**, which, in our language, is just another way of saying that the problem that takes an initial condition as input and gives the trajectory location as output has a high condition number.

► REALITY CHECK 6: THE TACOMA NARROWS BRIDGE DISASTER ◄

A mathematical model that attempts to model the Tacoma Narrows bridge incident was proposed recently by McKenna and Tuama [MT]. The goal is to explain how torsional, or twisting, oscillations can be magnified by forcing that is strictly vertical.

Consider a roadway of width $2l$ hanging between two suspended cables, as in Figure 17(a). We will consider a two-dimensional slice of the bridge, ignoring the dimension of the bridge's length for this model, since we are only interested in the side-to-side motion. At rest, the roadway hangs at a certain equilibrium height due to gravity; let y denote the current distance the center of the roadway hangs below this equilibrium.

Hooke's Law postulates a linear response, meaning that the restoring force the cables apply will be proportional to the deviation. Let θ be the angle the roadway makes with the horizontal. There are two suspension cables, stretched $y - l\sin\theta$ and $y + l\sin\theta$ from equilibrium, respectively. Assume a viscous damping term that is proportional to the velocity. Using Newton's law $F = ma$ and denoting Hooke's constant by K, the equations of motion for y and θ are

$$\begin{aligned} y'' &= -dy' - \left[\frac{K}{m}(y - l\sin\theta) + \frac{K}{m}(y + l\sin\theta)\right] \\ \theta'' &= -d\theta' + \frac{3\cos\theta}{l}\left[\frac{K}{m}(y - l\sin\theta) - \frac{K}{m}(y + l\sin\theta)\right] \end{aligned}$$

However, Hooke's law is designed for springs, where the restoring force is more or less equal whether the springs is compressed or stretched. McKenna and Tuama hypothesize that cables pull back with more force when stretched than they push back when compressed. (Think of a string as an extreme example.) They replace the linear Hooke's Law restoring force $f(y) = Ky$ with a nonlinear force $f(y) = (K/a)(e^{ay} - 1)$, as shown in Figure 17(b). Both functions have the same slope K at $y = 0$, but for the nonlinear force, a positive y (stretched cable) causes a stronger restoring force than the corresponding negative y (slackened cable). Making this replacement in the above equations yields

$$\begin{aligned} y'' &= -dy' - \frac{K}{ma}\left[e^{y - l\sin\theta} - 1 + e^{y + l\sin\theta} - 1\right] \\ \theta'' &= -d\theta' + \frac{3\cos\theta}{l}\frac{K}{ma}\left[e^{y - l\sin\theta} - e^{y + l\sin\theta}\right]. \end{aligned} \tag{54}$$

As the equations stand, the point $(y, \theta) = (0, 0)$ is an equilibrium. Now turn on the wind. Add the forcing term $A\sin\omega t$ to the right-hand-side of the y equation. This adds a strictly vertical oscillation to the bridge.

Useful estimates for the physical constants can be made. The mass of a one foot length of roadway was about 2500 kg, and the spring constant K has been estimated at 1000 Newtons. The roadway was about 12 meters wide. For this simulation, the damping coefficient was set at $d = 0.01$ and the Hooke's nonlinearity coefficient $a = 0.1$. The vertical forcing supplied by the wind on the final day caused the bridge to oscillate vertically about once every two seconds, so estimate $\omega = 2\pi/2 \approx 3$. These coefficients are only guesses, but they suffice to show ranges of motion that tend to match photographic evidence of the bridge's final oscillations. Matlab code that runs this model follows:

```
%Program 6.? Animation program for bridge using IVP solver
function tacoma(int,ic,h,p)
%Inputs: int = [a b] time interval,
```

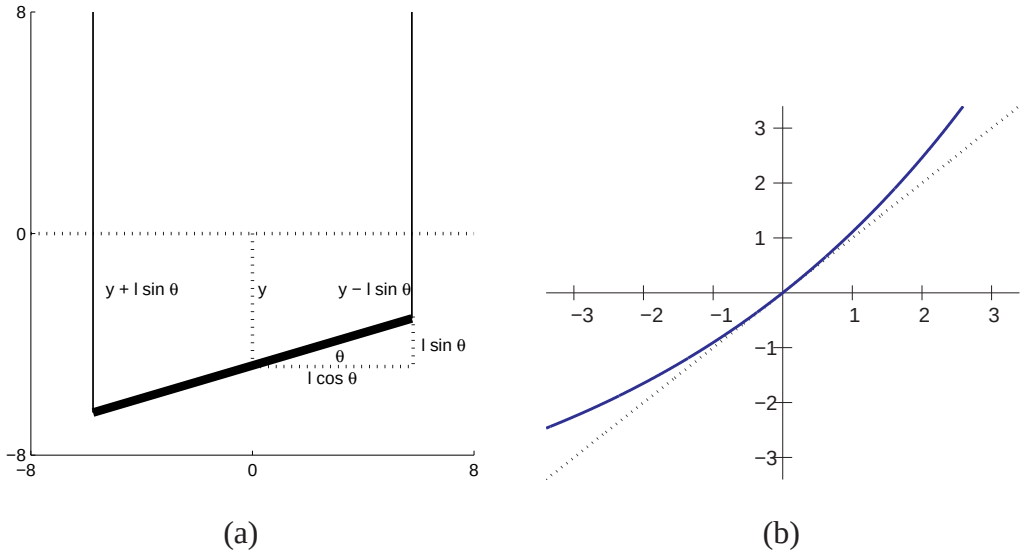


Figure 17: Schematics for the McKenna-Tuama model of the Tacoma Narrows bridge. (a) Denote the distance from the roadway center of mass to its equilibrium position by y, and the angle of the roadway with the horizontal by θ. (b) Exponential Hooke's Law curve $f(y) = (K/a)(e^{ay} - 1)$.

```
%ic = [y(1,1) y(1,2) y(1,3) y(1,4)], initialize
%h = stepsize, p = steps per point plotted
%Calls a one-step method such as trapstep.m
%Example usage: tacoma([0 500],[1 0 0.001 0],.04,3)
clf                       % clear figure window
a=int(1);b=int(2);n=ceil((b-a)/(h*p)); % plot n points in total
y(1,:)=ic;                % enter initial conds in y
t(1)=a;len=6;
set(gca,'XLim',[-8 8],'YLim',[-8 8], ...
   'XTick',[-8 0 8],'YTick',[-8 0 8], ...
   'Drawmode','fast','Visible','on','NextPlot','add');
cla;              % clear screen
axis square       % make aspect ratio 1 - 1
road=line('color','b','LineStyle','-','LineWidth',5,'erase','xor',...
   'xdata',[],'ydata',[]);
lcable=line('color','r','LineStyle','-','LineWidth',1,'erase','xor',...
   'xdata',[],'ydata',[]);
rcable=line('color','r','LineStyle','-','LineWidth',1,'erase','xor',...
   'xdata',[],'ydata',[]);
for k=1:n
  for i=1:p
    t(i+1) = t(i)+h;
    y(i+1,:) = trapstep(t(i),y(i,:),h);
  end
  y(1,:) = y(p+1,:);t(1)=t(p+1);
z1(k)=y(1,1);z3(k)=y(1,3);
c=len*cos(y(1,3));s=len*sin(y(1,3));
```

```
set(road,'xdata',[-c c],'ydata',[-s-y(1,1) -s-y(1,1)])
set(lcable,'xdata',[-c -c],'ydata',[-s-y(1,1) 8])
set(rcable,'xdata',[c c],'ydata',[s-y(1,1) 8])
drawnow; pause(h)
end

function y = trapstep(t,x,h)
%one step of the trapezoid method
z1=ydot(t,x);
g=x+h*z1;
z2=ydot(t+h,g);
y=x+h*(z1+z2)/2;

function ydot=ydot(t,y)
len=6;a=0.1;
a1=exp(a*(y(1)-len*sin(y(3))));
a2=exp(a*(y(1)+len*sin(y(3))));
ydot(1) = y(2);
ydot(2) = -0.01*y(2)-0.4*(a1+a2-2)/a+11*sin(3*t);
ydot(3) = y(4);
ydot(4) = -0.01*y(4)+1.2*cos(y(3))*(a1-a2)/(len*a);
```

Run `tacoma.m` with the default parameter values, to see the phenomenon postulated earlier. If the angle θ of the roadway is set to any small nonzero value, vertical forcing causes θ to eventually grow to a macroscopic value, leading to significant torsion of the roadway. The interesting point is that there is no torsional forcing applied to the equation; the "torsional mode" is excited completely by vertical forcing.

This project is an example of experimental mathematics. The equations are too difficult to derive closed-form solutions, and even too difficult to prove qualitative results about. Equipped with reliable ode solvers, we can generate numerical trajectories for various parameter settings, to illustrate the types of phenomena available to this model. Used in this way, differential equations models can predict behavior and shed light on mechanisms in scientific and engineering problems.

Questions to consider:

1. What happens if with the default system if the initial angle and angular velocity θ, θ' are set to zero?

2. What ranges of forcing amplitude A between 0 and 20 cause the torsional mode to be excited? (Use the default $\omega = 3$.) What ranges of forcing frequency ω between 0 and 4? (Use the default $A = 11$.) If the torsional mode is not excited, what happens to the angle θ? Try some large initial θ, say ≈ 0.1.

3. Print the time series of $y(t)$ and $\theta(t)$ from the code. Swap in an RK4 solver, and compare accuracy of the series before and after. How large a stepsize can be used in `trapstep` before accuracy is lost?

► ◄

Exercises 6.4

6.4.1. Write out the midpoint method for the IVPs in Exercise 6.1.4. Using stepsize $h = 1/4$, calculate the midpoint method approximation on the interval $[0, 1]$. Compare to the correct solution found in Exercise 6.1.4, and find the total error at each step.

6.4.2. Repeat Exercise 6.4.1 for the IVPs in Exercise 6.1.5.

6.4.3. Write out the order four Runge-Kutta method for the IVPs in Exercise 6.1.4. Using stepsize $h = 1/4$, calculate the RK4 method approximation on the interval $[0, 1]$. Compare to the correct solution found in Exercise 6.1.4, and find the total error at each step.

6.4.4. Repeat Exercise 6.4.3 for the IVPs in Exercise 6.1.5.

6.4.5. Prove that for any $\alpha \neq 0$, the method (50) is second order.

6.4.6. Consider the IVP $y' = \lambda y, y(0) = 1$. The solution is $y(t) = e^{\lambda t}$.

(a) Calculate w_1 for RK4 in terms of w_0 for this differential equation.

(b) Calculate the local truncation error by setting $w_0 = y_0 = 1$ and determining $y_1 - w_1$. Show that the local truncation error is of size $O(h^5)$, as expected for a fourth-order method.

6.4.7. Assume that the right-hand side $f(t, y) = f(t)$ doesn't depend on y. Show that $s_2 = s_3$ in fourth-order Runge-Kutta and that RK4 is equivalent to Simpson's rule for the integral $\int_{t_i}^{t_i+h} f(s)\, ds$.

Computer Problems 6.4

6.4.1. Write and test a Matlab m-file called `rkstep.m` that can be substituted for `eulerstep.m` or `trapstep.m` in the programs of the previous section.

6.4.2. Print the values of the midpoint method solution on a grid of step size $h = 0.1$ in $[0, 1]$ for the initial value problems in Exercise 6.1.4.

6.4.3. Print the values of the fourth-order Runge-Kutta method solution on a grid of step size $h = 0.1$ in $[0, 1]$ for the initial value problems in Exercise 6.1.4.

6.4.4. Repeat Computer Problem 6.4.3 but plot the approximate solutions on $[0, 1]$ for step sizes $h = 0.1, 0.05$, and 0.025 along with the true solution.

6.4.5. Repeat Computer Problem 6.4.3 for the equations of Exercise 6.1.5.

6.4.6. For the IVP's in Computer Problem 6.4.3, plot the global error of the RK4 method at $t = 1$ as a function of h, as in Figure 4.

6.5 Variable step-size methods.

UP to this point the step-size h has been treated as a constant in the implementation of the ODE solver. However, there is no reason that h cannot be changed during the solution process. A good reason to want to change the step size is for a solution that moves between periods of small change and periods of fast change. To make the fixed step-size small enough to track the fast changes accurately may mean that the rest of the solution is solved intolerably slowly.

The key idea of a variable step-size method is to monitor the error produced by the current step. The user sets an error tolerance that must be met by the current step. Then the method is designed to (1) reject the step and cut the step-size if the error tolerance is exceeded, or (2) if the error tolerance is met, to accept the step and then choose a step-size h that should be appropriate for the next step. The key need then is for some way to approximate the error made on each step. First let's assume we have found such a way, and explain how to change the step size.

The simplest way to vary step size is to double or halve the step size, depending on the current error. Compare the error estimate e_i, or relative error estimate $e_i/|w_i|$, with the error tolerance. (Here, as in the rest of this section, we will assume the ODE system being solved consists of one equation. It is fairly easy to generalize the ideas of this section to higher dimensions.) If the tolerance is not met, the step is repeated with new step size equal to $h_i/2$. If the tolerance is met too

well, say if the error is less than $1/10$ the tolerance, after accepting the step, the step size is doubled for the next step.

In this way, the step size will be adjusted automatically to a size that maintains the (relative) local truncation error near the user-requested level. Whether the absolute or relative error is used depends on the context; a good general purpose technique is to use the hybrid $e_i/\max(|w_i|, \theta)$ to compare with the error tolerance, where $\theta > 0$ protects against very small values of w_i.

A more sophisticated way to choose the appropriate step-size follows from knowledge of the order of the ODE solver. Assume the solver has order p, so that the local truncation error $e_i = O(h^{p+1})$. Let T be the relative error tolerance allowed by the user for each step. That means the goal is to ensure $e_i/|w_i| < T$.

If the goal $e_i/|w_i| < T$ is met, then the step is accepted and a new step size for the next step is needed. Assuming that

$$e_i \approx ch_i^{p+1} \tag{55}$$

for some constant c, the step size h that best meets the tolerance satisfies

$$T|w_i| = ch^{p+1}. \tag{56}$$

Solving the equations (55) and (56) for h yields

$$h_* = 0.8 * \left(\frac{T|w_i|}{e_i}\right)^{1/p+1} h_i \tag{57}$$

where we have added a safety factor of 0.8 to be conservative. Thus the next step size will be set to $h_{i+1} = h_*$.

On the other hand, if the goal $e_i/|w_i| < T$ is not met by the relative error, then h_i is set to h_* for a second try. This should suffice, because of the safety factor. However, if the second try also fails to meet the goal, then the step size is simply cut in half. This continues until the goal is achieved. As above, for general purposes, once should replace the relative error by $e_i/\max(|w_i|, \theta)$.

Both the simple and sophisticated methods described above depend heavily on some way to estimate the error of the current step of the ODE solver $e_i = |w_{i+1} - y_{i+1}|$. An important constraint is to gain the estimate without requiring a large amount of extra computation.

The most widely-used way for obtaining such an error estimate is to run a higher order ODE solver in parallel with the ODE solver of interest. The higher order method's estimate for w_{i+1}, call it z_{i+1}, will be significantly more accurate than the original w_{i+1}, so that the difference

$$e_i \approx |z_{i+1} - w_{i+1}| \tag{58}$$

is used as an error estimate for the current step from t_i to t_{i+1}.

Following this idea, several "pairs" of Runge-Kutta methods, one of order p and another of order $p + 1$, have been developed that share much of the needed computations. In this way the extra cost of step size control is kept low. Such a pair is often called an **embedded Runge-Kutta pair**.

Example 6.18 *RK2/3, An example of a Runge-Kutta order2/order 3 embedded pair.*

The explicit trapezoid method can be paired with a third-order RK method to make an embedded pair suitable for step size control. Set

$$\begin{aligned} w_{i+1} &= w_i + h\frac{s_1 + s_2}{2} \\ z_{i+1} &= w_i + h\frac{s_1 + 4s_3 + s_2}{6} \end{aligned}$$

where

$$\begin{aligned} s_1 &= f(t_i, w_i) \\ s_2 &= f(t_i + h, w_i + hs_1) \\ s_3 &= f(t_i + \frac{1}{2}h, w_i + \frac{1}{2}h\frac{s_1 + s_2}{2}) \end{aligned}$$

In the above, w_{i+1} is the trapezoid step, and z_{i+1} represents a third-order method, which requires the three Runge-Kutta stages shown. The third-order method is just an application of Simpson's rule for numerical integration to the context of differential equations. From the two ODE solvers, an estimate for the error can be found by subtracting the two approximations:

$$e_i \approx |w_{i+1} - z_{i+1}| = \left| h\frac{s_1 - 2s_3 + s_2}{3} \right|. \tag{59}$$

Using this estimate for the local truncation error allows the implementation of either of the step size control protocols described above.

■

Although the step size protocol has been worked out for w_{i+1}, it makes even better sense to use the higher order approximation z_{i+1} to advance the step, since it is available. This is called **local extrapolation**.

Example 6.19 *The Bogacki-Shampine order2/order 3 embedded pair.*

Matlab uses a different embedded pair in its `ode23` command. Let

$$\begin{aligned} s_1 &= f(t_i, w_i) \\ s_2 &= f(t_i + \frac{1}{2}h, w_i + \frac{1}{2}hs_1) \\ s_3 &= f(t_i + \frac{3}{4}h, w_i + \frac{3}{4}hs_2) \\ z_{i+1} &= w_i + \frac{h}{9}(2s_1 + 3s_2 + 4s_3) \\ s_4 &= f(t + h, z_{i+1}) \\ w_{i+1} &= w_i + \frac{h}{24}(7s_1 + 6s_2 + 8s_3 + 3s_4) \end{aligned} \tag{60}$$

It can be checked that z_{i+1} is an order 3 approximation, and w_{i+1}, despite having four stages, is order 2. The error estimate needed for step size control is

$$e_i = |z_{i+1} - w_{i+1}| = \frac{h}{72}| - 5s_1 + 6s_2 + 8s_3 - 9s_4|. \tag{61}$$

Note that s_4 becomes s_1 on the next step, if it is accepted, so there are no wasted stages - at least 3 stages are needed anyway for a third-order Runge-Kutta method. This design of the second order method is called FSAL, for First Same As Last.

■

Example 6.20 *The Runge-Kutta-Fehlberg order 4/order 5 embedded pair.*

$$
\begin{aligned}
s_1 &= f(t_i, w_i) \\
s_2 &= f(t_i + \frac{1}{4}h, w_i + \frac{1}{4}hs_1) \\
s_3 &= f(t_i + \frac{3}{8}h, w_i + \frac{3}{32}hs_1 + \frac{9}{32}hs_2) \\
s_4 &= f(t_i + \frac{12}{13}h, w_i + \frac{1932}{2197}hs_1 - \frac{7200}{2197}hs_2 + \frac{7296}{2197}hs_3) \\
s_5 &= f(t_i + h, w_i + \frac{439}{216}hs_1 - 8hs_2 + \frac{3680}{513}hs_3 - \frac{845}{4104}hs_4) \\
s_6 &= f(t_i + \frac{1}{2}h, w_i - \frac{8}{27}hs_1 + 2hs_2 - \frac{3544}{2565}hs_3 + \frac{1859}{4104}hs_4 - \frac{11}{40}hs_5) \\
w_{i+1} &= w_i + h(\frac{25}{216}s_1 + \frac{1408}{2565}s_3 + \frac{2197}{4104}s_4 - \frac{1}{5}s_5) \\
z_{i+1} &= w_i + h(\frac{16}{135}s_1 + \frac{6656}{12825}s_3 + \frac{28561}{56430}s_4 - \frac{9}{50}s_5 + \frac{2}{55}s_6)
\end{aligned}
\tag{62}
$$

It can be checked that z_{i+1} is an order 5 approximation, and that w_{i+1} is order 4. The error estimate needed for step size control is

$$
e_i = |z_{i+1} - w_{i+1}| = h\left|\frac{1}{360}s_1 - \frac{128}{4275}s_3 - \frac{2197}{75240}s_4 + \frac{1}{50}s_5 + \frac{2}{55}s_6\right| \tag{63}
$$

■

The Runge-Kutta-Fehlberg method (RKF45) is currently the most well-known variable step-size one-step method. Implementation is simple, given the above formulas. The user must set a relative error tolerance T and an initial step size h. After computing w_1, z_1 and e_0, the relative error test

$$
\frac{e_i}{|w_i|} < T \tag{64}
$$

is checked for $i = 0$. If successful, the new w_1 is replaced with the locally extrapolated version z_1 and the program moves on to the next step. On the other hand, if the relative error test (64) fails, the step is tried again with step size h given by (57) with $p = 4$, the order of the method producing w_i. (A repeated failure, which is unlikely, is treated by cutting step size in half until success is reached.) In either case, the step size h_1 for the next step should be calculated from (57).

Example 6.21 *The Dormand-Prince order 4/order 5 embedded pair.*

$$
\begin{aligned}
s_1 &= f(t_i, w_i) \\
s_2 &= f(t_i + \frac{1}{5}h, w_i + \frac{1}{5}hs_1) \\
s_3 &= f(t_i + \frac{3}{10}h, w_i + \frac{3}{40}hs_1 + \frac{9}{40}hs_2) \\
s_4 &= f(t_i + \frac{4}{5}h, w_i + \frac{44}{45}hs_1 - \frac{56}{15}hs_2 + \frac{32}{9}hs_3) \\
s_5 &= f(t_i + \frac{8}{9}h, w_i + h(\frac{19372}{6561}s_1 - \frac{25360}{2187}s_2 + \frac{64448}{6561}s_3 - \frac{212}{729}s_4)) \\
s_6 &= f(t_i + h, w_i + h(\frac{9017}{3168}s_1 - \frac{355}{33}s_2 + \frac{46732}{5247}s_3 + \frac{49}{176}s_4) - \frac{5103}{18656}s_5) \\
z_{i+1} &= w_i + h(\frac{35}{384}s_1 + \frac{500}{1113}s_3 + \frac{125}{192}s_4 - \frac{2187}{6784}s_5 + \frac{11}{84}s_6) \\
s_7 &= f(t_i + h, z_{i+1}) \\
w_{i+1} &= w_i + h(\frac{5179}{57600}s_1 + \frac{7571}{16695}s_3 + \frac{393}{640}s_4 - \frac{92097}{339200}s_5 + \frac{187}{2100}s_6 + \frac{1}{40}s_7) \qquad (65)
\end{aligned}
$$

It can be checked that z_{i+1} is an order 5 approximation, and that w_{i+1} is order 4. The error estimate needed for step size control is

$$
e_i = |z_{i+1} - w_{i+1}| = h\left|\frac{71}{57600}s_1 - \frac{71}{16695}s_3 + \frac{71}{1920}s_4 - \frac{17253}{339200}s_5 + \frac{22}{525}s_6 - \frac{1}{40}s_7\right| \qquad (66)
$$

Again local extrapolation is used, meaning that the step is advanced with z_{i+1} instead of w_{i+1}. Note that in fact, w_{i+1} need not be computed – only e_i is necessary for error control. This is a FSAL method, like the Bogacki-Shampine method, since s_7 becomes s_1 on the next step, if it is accepted. There are no wasted stages - it can be shown that at least 6 stages are needed for a fifth-order Runge-Kutta method.

■

The Matlab command `ode45` uses the Dormand-Prince embedded pair along with step size control roughly as described above. The user can set the relative tolerance T as desired. The right-hand side of the differential equation can be specified in a function file, for example

```
function y=f(t,y)
y = t*y+t^3;
```

Then the command

```
>> opts=odeset('RelTol',1e-4,'Refine',1,'MaxStep',1);
>> [t,y]=ode45('f',[0 1],1,opts);
```

will solve the initial value problem of Example 6.1 with initial condition $y_0 = 1$, and relative error tolerance $T = 0.0001$. If the parameter `RelTol` is not set, the default of 0.001 is used. Note that the function f input to `ode45` must be a function of two variables, in this case t and y, even if one of them is absent in the definition of the function. The command can be run without an accompanying function file by defining the function f "inline", as

```
>> [t,y]=ode45(inline('t*y+t^3','t','y'),[0 1],1,opts);
```

The output from `ode45` using the above parameter settings for this problem is

step	t_i	w_i	y_i	e_i
0	0.00000000	1.00000000	1.00000000	0.00000000
1	0.54021287	1.17946818	1.17946345	0.00000473
2	1.00000000	1.94617812	1.94616381	0.00001431

and if a relative tolerance of 10^{-6} is used,

step	t_i	w_i	y_i	e_i
0	0.00000000	1.00000000	1.00000000	0.00000000
1	0.21506262	1.02393440	1.02393440	0.00000000
2	0.43012524	1.10574441	1.10574440	0.00000001
3	0.68607729	1.32535658	1.32535653	0.00000005
4	0.91192246	1.71515156	1.71515144	0.00000012
5	1.00000000	1.94616394	1.94616381	0.00000013

The approximate solutions more than meet the relative error tolerance because of local extrapolation, meaning that the z_{i+1} is being used instead of w_{i+1}, even though the step size is designed to be sufficient for w_{i+1}. This is the best we can do - if we had an error estimate for z_{i+1}, we could use it to tune the step size even better, but we don't have one. Note also that the solutions stop exactly at the end of the interval $[0, 1]$, since `ode45` detects the end of the interval and truncates the step as necessary.

In order to see `ode45` do its step size selection, we had to turn off some basic default settings using the `odeset` command. The `Refine` parameter normally increases the number of solution values reported beyond what is computed by the method, to make a more beautiful graph, if and when the output is used for that purpose. The default value is 4, which causes four times the necessary number of points to be provided as output. The `MaxStep` parameter puts an upper limit on the step size h, and defaults to one-tenth the interval length. Using the default values for both of these parameters would mean a step size of $h = 0.1$ would be used, and after refining by a factor of 4, the solution would be shown with a step size of 0.025. In fact, running the command without an output variable specified

```
>> opts=odeset('RelTol',1e-6);
>> ode45(inline('t*y+t^3','t','y'),[0 1],1,opts);
```

will cause Matlab to automatically plot the solution on a grid of constant step size 0.025, as shown in Figure 18.

While it is tempting to crown variable step-size Runge-Kutta methods as the champion ODE solvers, there are a few types of equations that they do not handle very well. Here is a particularly simple example. See if you can decide where the problem lies.

Example 6.22 *Use* `ode45` *to solve the initial value problem within a relative tolerance of* 10^{-4}*:*

$$\begin{cases} y' = 10(1-y) \\ y(0) = 1/2 \\ t \in [0, 100]. \end{cases} \tag{67}$$

This can be accomplished in three lines of Matlab code:

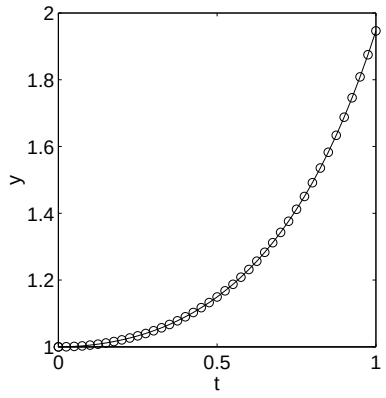


Figure 18: Matlab's `ode45` command. Solution of the initial value problem of Example 6.1 is computed, correct to within 10^{-6}.

```
>> opts=odeset('RelTol',1e-4);
>> [t,y]=ode45(inline('10*(1-y)','t','y'),[0 100],.5,opts);
>> length(t)

ans =

        1241
>>
```

We have printed the number of steps because it seems excessive. The solution to the initial value problem is easy to determine: $y(t) = 1 - e^{-10t}/2$. For $t > 1$ the solution has already reached its equilibrium 1 within 4 decimal places, and it never moves any farther away from 1. Yet `ode45` moves at a snail's pace, using an average step size of less than 0.1. Why such a conservative step size selection for such a tame solution?

Part of the answer is clear by viewing the output from `ode45` in Figure 19. Although the solution is very close to 1, the solver overshoots continually in trying to approximate closely. The differential equation is "stiff", a term we will formally define in the next section. For stiff equations, a different strategy in numerical solution increases solving efficiency greatly. For example, note the difference in steps needed when one of Matlab's stiff solvers are used:

```
>> opts=odeset('RelTol',1e-4);
>> [t,y]=ode23s(inline('10*(1-y)','t','y'),[0 100],.5,opts);
>> length(t)

ans =

        39
>>
```

Figure 19(b) plots the solution points from the solver `ode23s`. Relatively few points are needed to keep the numerical solution within the tolerance. We will investigate how to build methods that handle this type of difficulty in the next section.

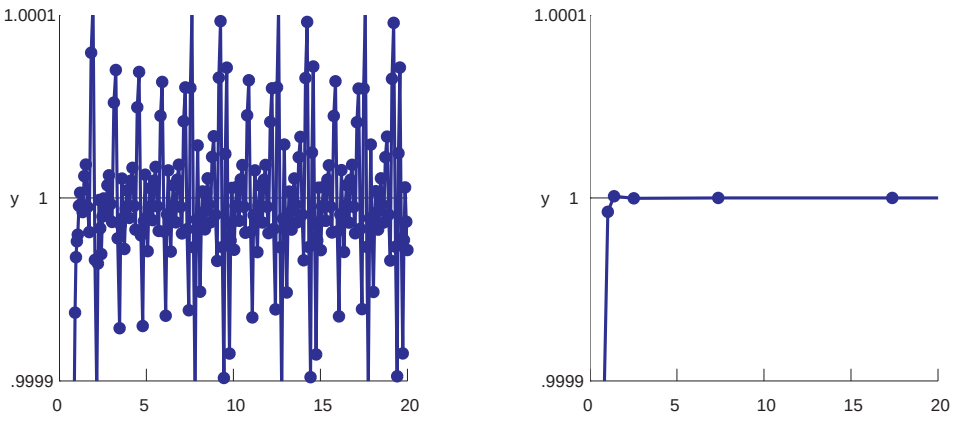


Figure 19: Numerical solution of the initial value problem of Example 6.22. (a) Using `ode45` requires over 10 steps per unit time to stay within relative tolerance 10^{-4}. (b) With `ode23s`, far fewer steps are needed.

■

Computer Problems 6.5

6.5.1. Write a Matlab implementation of RK23 (Example 6.18) and apply to approximating the solutions of the IVPs in Exercise 6.1.4 with a relative tolerance of 10^{-8} on $[0, 1]$. Ask the program to stop exactly at the endpoint $t = 1$. Report the maximum step size used and the number of steps.

6.5.2. Compare the results of Computer Problem 6.5.1 with the application of Matlab's `ode23` to the same problem.

6.5.3. Repeat Computer Problem 6.5.1 for the Runge-Kutta-Fehlberg method RKF45.

6.5.4. Compare the results of Computer Problem 6.5.3 with the application of Matlab's `ode45` to the same problem.

6.5.5. Apply a Matlab implementation of RKF45 to approximating the solutions of the systems in Exercise 6.3.1 with a relative tolerance of 10^{-6} on $[0, 1]$. Report the maximum step size used and the number of steps.

6.6 Implicit methods and stiff equations.

The difficulty that occurs when using variable step-size Runge-Kutta in Example 6.22 is that there is a particular step-size that cannot be exceeded for this solver applied to this differential equation. This phenomenon can be best understood in a much simpler context.

Example 6.23 *Apply Euler's method to Example 6.22.*

Euler's method for the right-hand side $f(t, y) = 10(1 - y)$ with step-size h is

$$\begin{aligned} w_{i+1} &= w_i + hf(t_i, w_i) \\ &= w_i + h(10)(1 - w_i) \\ &= w_i(1 - 10h) + 10h. \end{aligned} \tag{68}$$

Since the solution is $y(t) = 1 - e^{-10t}/2$, the approximate solution must approach 1 in the long run. Here we get some help from Chapter 1. Notice that (68) can be viewed as a fixed point iteration with $g(x) = x(1 - 10h) + 10h$. This iteration has a fixed point at $x = 1$, and it will be converged to as long as $|g'(1)| = |1 - 10h| < 1$. Solving this inequality yields $0 < h < 0.2$. For any larger h, the fixed point 1 will repel nearby guesses, and the solution will have no hope of being accurate.

■

Figure 20 shows this effect for Example 6.23. The solution is very tame - an attracting equilibrium at $y = 1$. An Euler step of size $h = 0.3$ has difficulty finding the equilibrium because the slope of the nearby solution changes so much between the beginning and the end of the h interval. This causes overshoot in the numerical solution.

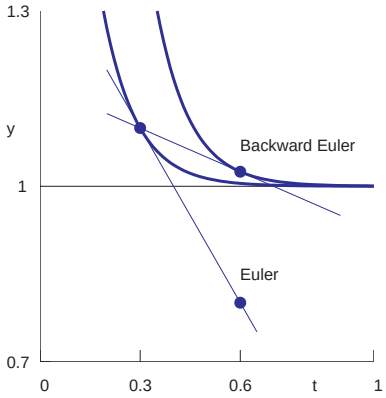


Figure 20: Comparison of Euler and Backward Euler steps. The differential equation in Example 6.22 is stiff. The equilibrium solution $y = 1$ is surrounded by other solutions with large curvature (fast changing slope). The Euler step overshoots, while the Backward Euler step is more consistent with the system dynamics.

Differential equations with this property, that attracting solutions are surrounded with fast-changing nearby solutions, are called **stiff**. This is often a sign of multiple time scales in the system. Quantitatively, it corresponds to the linear part of the right-hand-side f of the differential equation, in the variable y, being large and negative. (For a system of equations, this corresponds to an eigenvalue of the linear part being large and negative.) This definition is a bit relative, but that is the nature of stiffness - the more large and negative, the smaller the step-size must be to avoid overshoot. For Example 6.23, stiffness is measured by evaluating $\partial f/\partial y = -10$ at the equilibrium solution $y = 1$.

One way to solve the problem depicted in Figure 20 is to somehow bring in information from the right side of the interval $[t_i, t_i + h]$, instead of relying solely on information from the left side. That is the motivation behind the

Backward Euler method

$$w_{i+1} = w_i + hf(t_{i+1}, w_{i+1}). \tag{69}$$

Note the difference - while the Euler method uses the left-end slope to step across the interval, Backward Euler would like to somehow cross the interval so that the slope is correct at the right end.

A price must be paid for this improvement. Backward Euler is our first example of an **implicit** method, meaning that the method does not give directly a formula for the new approximation w_{i+1}. Instead, we must work a little to get it. For the example $y' = 10(1-y)$, the Backward Euler method gives

$$w_{i+1} = w_i + 10h(1 - w_{i+1}),$$

which, after a little algebra, can be expressed as

$$w_{i+1} = \frac{w_i + 10h}{1 + 10h}.$$

Setting $h = 0.3$ for example, the Backward Euler method gives $w_{i+1} = (w_i + 3)/4$. We can again evaluate the behavior as a fixed point iteration $w \to g(w) = (w + 3)/4$. There is a fixed point at 1, and $g'(1) = 1/4 < 1$, verifying convergence to the true equilibrium solution $y = 1$. Unlike the Euler method with $h = 0.3$, at least the correct qualitative behavior is followed by the numerical solution. In fact, note that the Backward Euler method's solution converges to $y = 1$ no matter how large the step size h (Exercise 6.6.1).

Because of the better behavior of implicit methods like Backward Euler in the presence of stiff equations, it is worthwhile performing extra work required to evaluate the next step, even though it is not explicitly available. Example 6.23 was not challenging to solve for w_{i+1}, due to the fact that the differential equation is linear, and it was possible to change the original implicit formula to an explicit one for evaluation. In general, however, this is not possible, and we need to use more indirect means.

If the implicit method leaves a nonlinear equation to solve, we must refer to Chapter 1. Both fixed point iteration and Newton's method are often used to solve for w_{i+1}. This means there is an equation-solving loop within the loop advancing the differential equation. The next Example shows how this can be done.

Example 6.24 *Apply the Backward Euler method to the initial value problem*

$$\begin{cases} y' = y + 8y^2 - 9y^3 \\ y(0) = 1/2 \\ t \in [0, 3]. \end{cases}$$

This equation, like the previous example, has an equilibrium solution $y = 1$. The partial derivative $\partial f/\partial y = 1 + 16y - 27y^2$ evaluates to -10 at $y = 1$, identifying this equation as moderately stiff. There will be a upper bound, similar to that of the previous example, for h such that Euler's method is successful. Thus we are motivated to try the Backward Euler method

$$\begin{aligned} w_{i+1} &= w_i + hf(t_{i+1}, w_{i+1}) \\ &= w_i + h(w_{i+1} + 8w_{i+1}^2 - 9w_{i+1}^3). \end{aligned}$$

This is a nonlinear equation in w_{i+1}, which we need to solve in order to advance the numerical solution. Renaming $z = w_{i+1}$, we must solve the equation $z = w_i + h(z + 8z^2 - 9z^3)$, or

$$9hz^3 - 8hz^2 + (1 - h)z - w_i = 0 \tag{70}$$

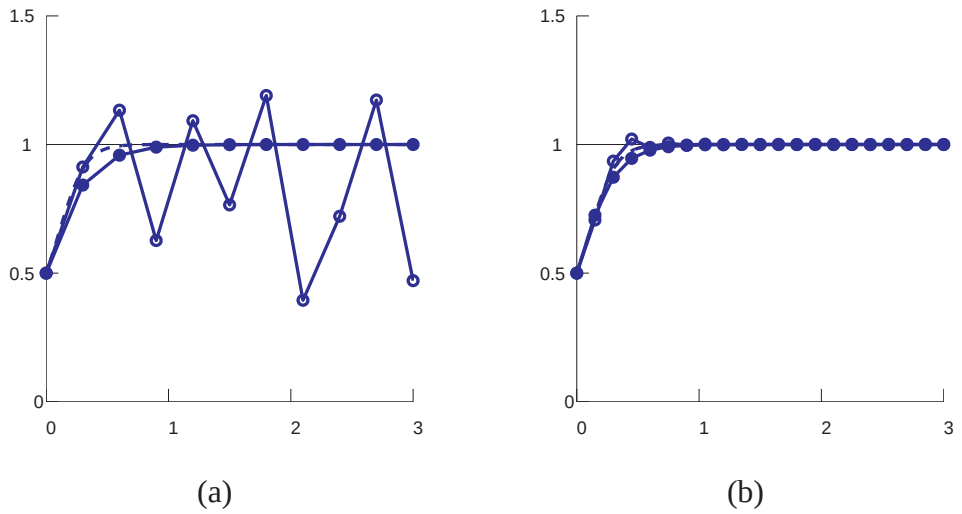


Figure 21: Numerical solution of the initial value problem of Example 6.24. True solution is the dashed curve. The open circles denote the Euler method approximation; the closed circles denote Backward Euler. (a) $h = 0.3$ (b) $h = 0.15$.

for the unknown z. We will demonstrate with Newton's method. To start Newton's method, we need an initial guess. Many choices could work - two choices that come to mind are the previous approximation w_i, and the Euler's method approximation for w_{i+1}. Although the latter is accessible since Euler is explicit, it may not be the best choice for stiff problems, as shown in Figure 20. In this case we will use w_i as the starting guess.

Assembling Newton's method for (70) yields

$$z_{\text{new}} = z - \frac{9hz^3 - 8hz^2 + (1-h)z - w_i}{27hz^2 - 16hz + 1 - h} \tag{71}$$

After evaluating (71), replace z with z_{new} and repeat. For each Backward Euler step, Newton's method is run until $z_{\text{new}} - z$ is smaller than a preset tolerance (smaller than the errors that are being made in approximating the differential equation solution).

Figure 21 shows the results for two different step sizes. In addition, numerical solutions from Euler's method are shown. Clearly $h = 0.3$ is too large for Euler on this stiff problem. On the other hand, when h is cut to 0.15, both methods perform at about the same level. So-called stiff solvers like Backward Euler allow sufficient error control with comparatively large step size, increasing efficiency.

■

Exercises 6.6

6.6.1. Show that for every step size h, the Backward Euler approximate solution converges to the equilibrium solution $y = 1$ as $t_i \to \infty$ for Example 6.23.

6.6.2. Find all equilibrium solutions and the value of the Jacobian at the equilibria. Is the equation stiff? (a) $y' = y - y^2$ (b) $y' = 10y - 10y^2$ (c) $y' = -10 \sin y$

[Ans.: (a) $y = 0, f_y(0) = 1$ No. $y = 1, f_y(1) = -1$ No. (b) $y = 0, f_y(0) = 10$ No. $y = 1, f_y(1) = -10$ Yes. (c) $y = n\pi$ for all integers n, $f_y(\text{odd}\pi) = 10$ No, $f_y(\text{even}\pi) = -10$ Yes.]

6.6.3. Consider the linear differential equation $y' = ay + b$ for $a < 0$. (a) Find the equilibrium. (b) Write down the Backward Euler method for the equation. (c) View Backward Euler as a Fixed Point Iteration to prove that the method's approximate solution will converge to the equilibrium as $t \to \infty$.

Computer Problems 6.6

6.6.1. Compare Euler's method to Backward Euler for the IVP

$$\text{(a)} \begin{cases} y' = y^2 - y^3 \\ y(0) = 1/2 \\ t \in [0, 100]. \end{cases} \quad \text{(b)} \begin{cases} y' = 6y - 6y^2 \\ y(0) = 1/2 \\ t \in [0, 100]. \end{cases} \quad \text{(c)} \begin{cases} y' = 6y - 3y^2 \\ y(0) = 1/2 \\ t \in [0, 100]. \end{cases}$$

Which of the equilibrium solutions are approached by the approximate solution? For what range of h can Euler be used successfully? How large can the backward Euler step size be made, while achieving equivalent accuracy?

6.7 Multistep methods.

THE Runge-Kutta family that we have studied consists of one-step methods, meaning that the newest step w_{i+1} is produced on the basis of the differential equation and the value of the previous step w_i. This is in the spirit of well-defined initial value problems, for which a unique solution exists starting at an arbitrary w_i.

The multistep methods suggest a different approach - using the knowledge of more than one of the previous w_i to help produce the next step. This will lead to ODE solvers that have order as high as the one-step methods, but much of the computation necessary will be replaced by essentially interpolating from past values on the solution path. For example, while the second-order Midpoint method

$$w_{i+1} = w_i + hf(t_i + \frac{h}{2}, w_i + \frac{h}{2} f(t_i, w_i))$$

needs two function evaluations of the ODE right-hand side f per step, the

Adams-Bashforth Two-Step method

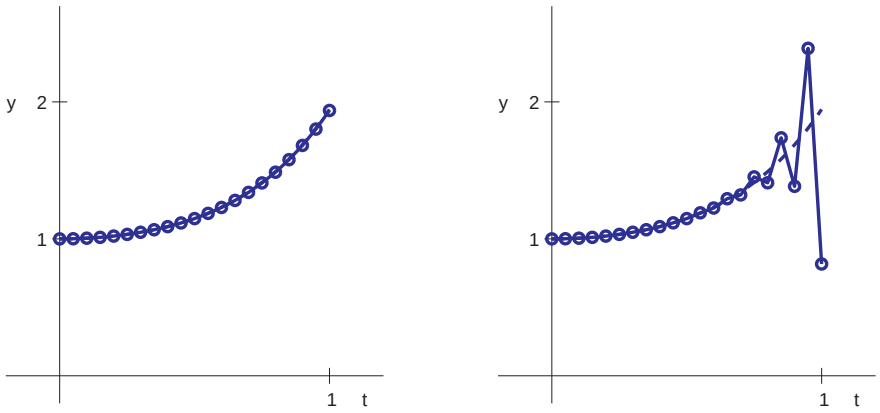
$$w_{i+1} = w_i + h\left[\frac{3}{2} f(t_i, w_i) - \frac{1}{2} f(t_{i-1}, w_{i-1})\right] \tag{72}$$

requires only one new evaluation per step (one is stored from the previous step). We will see below that (72) is also an second-order method. Therefore multistep methods can achieve the same order with less computational effort - usually just one function evaluation per step.

Since multistep methods use more than one previous w values, they need help getting started. The start-up phase for an s-step method typically consists of a one-step method which uses w_0 to produce $s - 1$ values $w_1, w_2, \ldots, w_{s-1}$, before the multistep method can be used. The Adams-Bashforth two-step method (72) needs w_1 along with the given initial condition w_0 in order to begin. The following Matlab code uses the trapezoid method to provide the start-up value w_1.

```
% Program 6.6 Multistep method plotting program
function y=exmultistep(int,ic,h,s)
% Inputs: int= [a,b] time interval
% ic = [y0]  initial condition
% h = stepsize, s = number of (multi)steps
% Calls a multistep method such as ab2step.m
% Example usage: exmultistep
a=int(1);b=int(2);n=ceil((b-a)/h);
% Start-up phase
y(1,:)=ic;t(1)=a;
for i=1:s-1               % start-up phase, using one-step method
  t(i+1)=t(i)+h;
  y(i+1,:)=trapstep(t(i),y(i,:),h);
  f(i,:)=ydot(t(i),y(i,:));
end
for i=s:n                 % multistep method loop
  t(i+1)=t(i)+h;
  f(i,:)=ydot(t(i),y(i,:));
  y(i+1,:)=ab2step(t(i),i,y,f,h);
end

function y=trapstep(t,x,h)
%one step of the trapezoid method from section 6.2
z1=ydot(t,x);
g=x+h*z1;
z2=ydot(t+h,g);
y=x+h*(z1+z2)/2;

function z=ab2step(t,i,y,f,h)
%one step of the Adams-Bashforth 2-step method
z=y(i,:)+h*(3*f(i,:)/2-f(i-1,:)/2);

function z=another2step(t,i,y,f,h)
%one step of another 2-step method
z=y(i,:)/2+y(i-1,:)/2+h*(7*f(i,:)/4-f(i-1,:)/4);

function z=unstable2step(t,i,y,f,h)
%one step of an unstable 2-step method
z=-y(i,:)+2*y(i-1,:)+h*(5*f(i,:)/2+f(i-1,:)/2);

function z=weaklystable2step(t,i,y,f,h)
%one step of a  weakly-stable 2-step method
z=y(i-1,:)+h*2*f(i,:);

function z=weaklystable2step1(t,i,y,f,h)
%one step of another weakly-stable 2-step method
z=2*y(i,:)-y(i-1,:)+h*(f(i,:)-f(i-1,:));

function ydot = ydot(t,y)  % IVP from section 6.1
ydot = t*y+t*t*t;
```

Figure 22(a) shows the result of applying Adams-Bashforth two-step method to the initial value problem (5) from earlier in the chapter, using step size $h = 0.05$, using the trapezoid method for start-up. Part (b) of the figure shows the use of a different two-step method. Its instability will be the subject of our look at stability analysis in the next sections.



(a) (b)

Figure 22: Two-step methods applied to IVP (5). Correct solution is the dashed curve. Step size $h = 0.05$. (a) Adams-Bashforth two-step method plotted as circles (b) Unstable method (81) in circles.

6.7.1 Generating multistep methods.

A general s-step method has the form

$$w_{i+1} = a_1 w_i + a_2 w_{i-1} + \ldots + a_s w_{i-s+1} + h[b_0 f_{i+1} + b_1 f_i + b_2 f_{i-1} + \ldots + b_s f_{i-s+1}]. \quad (73)$$

The step size is h and we use the notational convenience

$$f_i \equiv f(t_i, w_i).$$

If $b_0 = 0$, the method is explicit. If $b_0 \neq 0$, the method is implicit. We will discuss how to use implicit methods below.

First, we want to show how multistep methods are derived, and how to decide which ones will work best. The main issues that arise with multistep methods can be introduced in the relatively simple case of two-step methods, so we begin there. A general two-step method (setting $s = 2$ in (73)) has the form

$$w_{i+1} = a_1 w_i + a_2 w_{i-1} + h[b_0 f_{i+1} + b_1 f_i + b_2 f_{i-1}]. \quad (74)$$

To develop a multistep method, we need to resort to Taylor's Theorem once more, since the game remains to match as many terms of the solution's Taylor expansion as possible with the terms of the method. What remains will be the local truncation error.

We assume that all previous w_i are correct, i.e. $w_i = y_i$ and $w_{i-1} = y_{i-1}$ in (74). The differential equation says that $y_i' = f_i$, so that all terms can be expanded in a Taylor expansion as

follows:

$$\begin{aligned} w_{i+1} &= a_1 w_i + a_2 w_{i-1} + h[b_0 f_{i+1} + b_1 f_i + b_2 f_{i-1}] \\ &= a_1[y_i] \\ &\quad + a_2[y_i - hy_i' + \tfrac{h^2}{2}y_i'' - \tfrac{h^3}{6}y_i''' + \tfrac{h^4}{24}y_i'''' - \ldots] \\ &\quad + b_0[hy_i' + h^2 y_i'' + \tfrac{h^3}{2}y_i''' + \tfrac{h^4}{6}y_i'''' + \ldots] \\ &\quad + b_1[hy_i'] \\ &\quad + b_2[hy_i' - h^2 y_i'' + \tfrac{h^3}{2}y_i''' - \tfrac{h^4}{6}y_i'''' + \ldots] \end{aligned}$$

Adding up yields

$$\begin{aligned} w_{i+1} &= (a_1 + a_2)y_i + (b_0 + b_1 + b_2 - a_2)hy_i' + (a_2 - 2b_2 + 2b_0)\frac{h^2}{2}y_i'' \\ &\quad + (-a_2 + 3b_0 + 3b_2)\frac{h^3}{6}y_i''' + (a_2 + 4b_0 - 4b_2)\frac{h^4}{24}y_i'''' + \ldots \end{aligned} \tag{75}$$

By choosing the a_i and b_i appropriately, the local truncation error $y_{i+1} - w_{i+1}$, where

$$y_{i+1} = y_i + hy_i' + \frac{h^2}{2}y_i'' + \frac{h^3}{6}y_i''' + \ldots \tag{76}$$

can be made as small as possible, assuming the derivatives involved actually exist. Next, we will investigate the possibilities.

6.7.2 Explicit multistep methods

To look for explicit methods, set $b_0 = 0$. A second-order method can be developed by matching terms in (75) and (76) up to and including the h^2 term, making the local truncation error of size $O(h^3)$. Comparing terms, we find we need to solve the system

$$\begin{aligned} a_1 + a_2 &= 1 \\ -a_2 + b_1 + b_2 &= 1 \\ a_2 - 2b_2 &= 1 \end{aligned} \tag{77}$$

There are three equations in four unknowns, so it will be possible to find infinitely many different explicit order-two methods. (There is also one order-three method that turns out to be not useful. See Exercise 6.7.4.) Note that the equations can be written in terms of a_1 as follows:

$$\begin{aligned} a_2 &= 1 - a_1 \\ b_1 &= 2 - \frac{1}{2}a_1 \\ b_2 &= -\frac{1}{2}a_1 \end{aligned} \tag{78}$$

The local truncation error will be

$$\begin{aligned} y_{i+1} - w_{i+1} &= \frac{1}{6}h^3 y_i''' - \frac{3b_2 - a_2}{6}h^3 y_i''' + O(h^4) \\ &= \frac{1 - 3b_2 + a_2}{6}h^3 y_i''' + O(h^4) \\ &= \frac{4 + a_1}{12}h^3 y_i''' + O(h^4). \end{aligned} \tag{79}$$

SPOTLIGHT ON: ***Complexity***

The advantage of multistep methods to one-step methods is clear. After the first few steps, only one new evalution of the right-hand side function need be made. For one-step methods, it is typical for function evaluations to be needed. Fourth-order Runge-Kutta, for example, needs four evaluations per step, while the fourth-order Adams-Bashforth method needs only one after the startup phase.

We are free to set a_1 arbitrarily - any choice leads to a second-order method, as we have just shown. Setting $a_1 = 1$ yields the second-order Adams-Bashforth method (72). Note that $a_2 = 0$ by the first equation, and $b_2 = -1/2$ and $b_1 = 3/2$. According to (79), the local truncation error is $\frac{5}{12}h^3y'''(t_i) + O(h^4)$.

Alternatively, we could set $a_1 = 1/2$ to get another two-step second-order method with $a_2 = 1/2, b_1 = 7/4$, and $b_2 = -1/4$:

Another second-order two-step method

$$w_{i+1} = \frac{1}{2}w_i + \frac{1}{2}w_{i-1} + h[\frac{7}{4}f_i - \frac{1}{4}f_{i-1}], \tag{80}$$

which has local truncation error $\frac{3}{8}h^3y'''(t_i) + O(h^4)$.

A third choice, $a_1 = -1$, gives

Unstable second-order two-step method

$$w_{i+1} = -w_i + 2w_{i-1} + h[\frac{5}{2}f_i + \frac{1}{2}f_{i-1}] \tag{81}$$

Figure 22 showed that the methods (72) and (81) are not equal in effectiveness. The failure of (81) brings out an important condition that must be met by multistep solvers, called stability. Consider the even simpler IVP

$$\begin{cases} y' = 0 \\ y(0) = 0 \\ t \in [0, 1]. \end{cases} \tag{82}$$

Write out method (81) for this example. It is

$$w_{i+1} = -w_i + 2w_{i-1} + h[0] \tag{83}$$

There are many solutions $\{w_i\}$ to (83). One is $w_i \equiv 0$. However, there are others. Substituting the

form $w_i = c\lambda^i$ into (83) yields

$$\begin{aligned} c\lambda^{i+1} + c\lambda^i - 2c\lambda^{i-1} &= 0 \\ c\lambda^{i-1}(\lambda^2 + \lambda - 2) = 0 & \end{aligned} \tag{84}$$

The solutions of the "characteristic polynomial" $\lambda^2 + \lambda - 2 = 0$ of this recurrence relation are 1 and -2. The latter is a problem - it means that solutions of form $(-2)^i c$ are solutions of the method for small c. This allows small rounding and truncation errors to quickly grow to unit size and swamp the computation, as seen in Figure 22. It is important to be sure that the roots of the characteristic polynomial of the method stay bounded by 1 in absolute value, to avoid this possibility. This leads to the following definition.

Definition 6.6 The multistep method (73) is **stable** if the roots of the polynomial $P(x) = x^s + a_1 x^{s-1} + \ldots + a_s$ are bounded by 1 in absolute value, and any roots of absolute value 1 are simple roots. A stable method for which 1 is the only root of absolute value 1 is called **strongly stable**; otherwise it is **weakly stable**.

The Adams-Bashforth method (72) has roots 0 and 1, making it strongly stable, while (81) has roots -2 and 1, making it unstable.

The characteristic polynomial of the general two-step formula is

$$\begin{aligned} \rho(x) &= x^2 - a_1 x - a_2 \\ &= x^2 - a_1 x - 1 + a_1 \\ &= (x-1)(x - a_1 + 1) \end{aligned}$$

whose roots are 1 and $a_1 - 1$. Returning to (78), we can find a weakly stable second-order method by setting $a_1 = 0$. Then the roots are 1 and -1, leading to a

Weakly stable second-order two-step method

$$w_{i+1} = w_{i-1} + 2hf_i \tag{85}$$

Example 6.25 Apply strongly stable method (72), weakly stable method (85), and unstable method (81) to the initial value problem

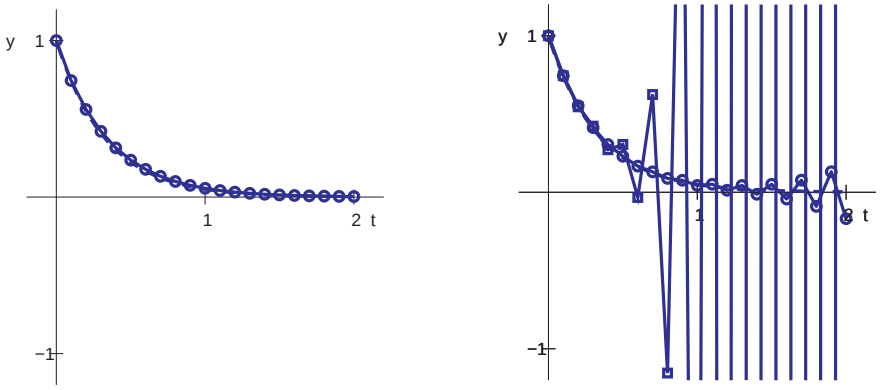
$$\begin{cases} y' = -3y \\ y(0) = 1 \\ t \in [0, 2]. \end{cases} \tag{86}$$

The solution is the curve $y = e^{-3t}$. We will use Program 6.3 to follow the solutions, where `ydot.m` has been changed to

```
function ydot = ydot(t,y)
ydot = -3*y;
```

and `ab2step.m` is replaced by one of the three calls `ab2step`, `weaklystable2step`, or `unstable2step`.

Figure 23 shows the three solution approximations for stepsize $h = 0.1$. The weakly stable and unstable methods seem to follow closely for a while, and then move quickly away from the correct solution. Reducing the step size does not eliminate the problem, although it may delay the onset of instability.



(a) (b)

Figure 23: Comparison of second-order, two-step methods applied to IVP (86). (a) Adams-Bashforth method (b) Weakly stable method (in circles) and unstable method (in squares).

With two more definitions, we can state the fundamental theorem of multistep solvers.

Definition 6.7 A multistep solver is **consistent** if it has order at least 1. A solver is **convergent** if the approximate solutions converge to the exact solution for each t, as $h \to 0$.

Theorem 6.8 (Dahlquist) Assume that the starting values are correct. Then a multistep method (73) is convergent if and only if it is stable and consistent.

One root of the characteristic polynomial must be at 1 (see Exercise 6.7.6). The Adams-Bashforth methods are the ones whose other roots are all at 0. For this reason, the Adams-Bashforth two-step method is considered the ”most stable” of the two-step methods.

The derivation of higher-order methods, using more steps, is precisely analogous to our derivation of two-step methods above. Strongly stable multistep methods include the following.

Adams-Bashforth Three-Step Method (third-order)

$$w_{i+1} = w_i + \frac{h}{12}[23f_i - 16f_{i-1} + 5f_{i-2}] \tag{87}$$

Adams-Bashforth Four-Step Method (fourth-order)

$$w_{i+1} = w_i + \frac{h}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}] \tag{88}$$

6.7.3 Implicit multistep methods

When the coefficient b_0 in (73) is nonzero, the method is implicit. The simplest second-order implicit method (see Exercise 6.7.3) is the

Implicit Trapezoid Method (second-order)

$$w_{i+1} = w_i + \frac{h}{2}[f_{i+1} + f_i] \tag{89}$$

If the f_{i+1} term is replaced by evaluating f at the "prediction" for w_{i+1} made by Euler's method, then this becomes the explicit trapezoid method. The implicit trapezoid method is also called the Adams-Moulton one-step method, by analogy with what follows. An example of a two-step implicit method is the

Adams-Moulton Two-Step Method (third-order)

$$w_{i+1} = w_i + \frac{h}{12}[5f_{i+1} + 8f_i - f_{i-1}] \tag{90}$$

There are significant differences between the implicit and explicit methods. First, it is possible to get a stable third-order implicit method using only two steps, unlike the explicit case. Second, the corresponding local truncation error formula is smaller for implicit methods. On the other hand, the implicit method has the inherent difficulty that it needs some kind of help to evaluate the implicit part.

For these reasons, implicit methods are often used as the corrector in a "predictor-corrector" pair. Implicit and explicit methods of the same order are used together. Each step is the combination of a prediction by the explicit method and a correction by the implicit method, where the implicit method uses the predicted w_{i+1} to calculate f_{i+1}. Predictor-corrector methods use approximately twice the computational effort, since an evaluation of the differential equation right-hand-side f is done on both the prediction and the correction parts of the step. However, the added accuracy and stability often make the price worth paying.

A simple predictor-corrector method pairs the two-step Adams-Bashforth explicit method as

predictor with the one-step Adams-Moulton implicit method as corrector. Both are second-order methods. Matlab code looks similar to the Adams-Bashforth code used earlier, but with a corrector step added.

```
% Program 6.7 Adams-Bashforth-Moulton second-order predictor-corrector
function y=predcorr(int,ic,h,s)
% Inputs: int= [a,b] time interval
% ic = [y0]  initial condition
% h = stepsize, s = number of steps for explicit method
% Calls multistep methods such as ab2step.m and am1step.m
% Example usage: predcorr([0 1],1,.05,2)
a=int(1);b=int(2);n=ceil((b-a)/h);
% Start-up phase
y(1,:)=ic;t(1)=a;
for i=1:s-1               % start-up phase, using one-step method
  t(i+1)=t(i)+h;
  y(i+1,:)=trapstep(t(i),y(i,:),h);
  f(i,:)=ydot(t(i),y(i,:));
end
for i=s:n                 % multistep method loop
  t(i+1)=t(i)+h;
  f(i,:)=ydot(t(i),y(i,:));
  y(i+1,:)=ab2step(t(i),i,y,f,h);  % predict
  f(i+1,:)=ydot(t(i+1),y(i+1,:));
  y(i+1,:)=am1step(t(i),i,y,f,h);  % correct
end
plot(t,y(:,1),t,y(:,1),'o');

function y=trapstep(t,x,h)
%one step of the trapezoid method from section 6.2
z1=ydot(t,x);
g=x+h*z1;
z2=ydot(t+h,g);
y=x+h*(z1+z2)/2;

function z=ab2step(t,i,y,f,h)
%one step of the Adams-Bashforth 2-step method
z=y(i,:)+h*(3*f(i,:)-f(i-1,:))/2;

function z=am1step(t,i,y,f,h)
%one step of the Adams-Moulton 1-step method
z=y(i,:)+h*(f(i+1,:)+f(i,:))/2;

function ydot = ydot(t,y)  % IVP
ydot = t*y+t*t*t;
```

Deriving the Adams-Moulton two-step method is done just as the explicit methods were established. Redo the set of equations (77) but without requiring $b_0 = 0$. Since there is an extra parameter now (b_0) we are able to match up (75) and (76) through the degree 3 terms, putting the

local truncation error in the h^4 term. The analogue to (77) is

$$\begin{aligned} a_1 + a_2 &= 1 \\ -a_2 + b_0 + b_1 + b_2 &= 1 \\ a_2 + 2b_0 - 2b_2 &= 1 \\ -a_2 + 3b_0 + 3b_2 &= 1 \end{aligned} \tag{91}$$

Satisfying these equations results in a third-order two-step implicit method.

The equations can be written in terms of a_1 as follows:

$$\begin{aligned} a_2 &= 1 - a_1 \\ b_0 &= \frac{1}{3} + \frac{1}{12}a_1 \\ b_1 &= \frac{4}{3} - \frac{2}{3}a_1 \\ b_2 &= \frac{1}{3} - \frac{5}{12}a_1 \end{aligned} \tag{92}$$

The local truncation error is

$$\begin{aligned} y_{i+1} - w_{i+1} &= \frac{1}{24}h^4 y_i'''' - \frac{4b_0 - 4b_2 + a_2}{24}h^4 y_i'''' + O(h^5) \\ &= \frac{1 - a_2 - 4b_0 + 4b_2}{24}h^4 y_i'''' + O(h^5) \\ &= -\frac{a_1}{24}h^4 y_i'''' + O(h^5). \end{aligned} \tag{93}$$

The order of the method will be three as long as $a_1 \neq 0$. Again, a_1 is a free parameter, so there are infinitely many third-order two-step implicit methods. The Adams-Moulton two-step method uses the choice $a_1 = 1$. Exercise 6.7.8 asks you to verify that this method is strongly stable. Exercise 6.7.9 explores other choices of a_1.

Note one more special choice, $a_1 = 0$. From the local truncation formula we see that this two-step method

Simpson's Method

$$w_{i+1} = w_{i-1} + \frac{h}{3}[f_{i+1} + 4f_i + f_{i-1}] \tag{94}$$

will be fourth-order. Exercise 6.7.15 asks you to check that it is only weakly-stable. For this reason, it is susceptible to error magnification.

The suggestive terminology of the Trapezoid Method (89) and Simpson's Method (94) should remind the reader of the numerical integration formulas from Chapter 5. In fact, although we have not emphasized this approach, many of the multistep formulas we have presented can be alternately derived by integrating approximating interpolants, in a close analogy to numerical integration schemes.

The basic idea behind this approach is that the differential equation $y' = f(t, y)$ can be integrated on the interval $[t_i, t_{i+1}]$ to give

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y)\, dt. \tag{95}$$

Applying a numerical integration scheme to approximate the integral in (95) results in a multistep ODE method. For example, using the trapezoid rule for numerical integration from Chapter 5 yields

$$y(t_{i+1}) - y(t_i) = \frac{1}{2}(f_{i+1} + f_i) + O(h^2),$$

which is the second-order Trapezoid Method for ODE's. If we approximate the integral by Simpson's Rule, the result is

$$y(t_{i+1}) - y(t_i) = \frac{1}{3}(f_{i+1} + 4f_i + f_{i-1}) + O(h^4),$$

the fourth-order Simpson's Method (94). Essentially, we are approximating the right-hand-side of the ODE by a polynomial and integrating, just as is done in numerical integration. This approach can be extended to recover various of the multistep methods we have already presented, by changing the degree of interpolation and the location of the interpolation points. Although this approach is a more geometric way of deriving some the the multistep methods, it gives no particular insight into the stability of the resulting ODE solver.

By extending the previous methods, the higher-order Adams-Moulton methods can be derived, in each case using $a_1 = 1$:

Adams-Moulton Three-Step Method (fourth-order)

$$w_{i+1} = w_i + \frac{h}{24}[9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2}] \tag{96}$$

Adams-Moulton Four-Step Method (fifth-order)

$$w_{i+1} = w_i + \frac{h}{720}[251f_{i+1} + 646f_i - 264f_{i-1} + 106f_{i-2} - 19f_{i-3}] \tag{97}$$

These methods are heavily used in predictor-corrector methods along with an Adams-Bashforth predictor of the same order. Computer Problems 6.7.5 etc. asks you to write Matlab code to implement this idea.

Exercises 6.7

6.7.1. Write out the Adams-Bashforth method for the IVPs in Exercise 6.1.4. Using stepsize $h = 1/4$, calculate the approximation on the interval $[0, 1]$. Use the trapezoid method to create w_1. Compare to the correct solution, and find the total error at each step.

6.7.2. Repeat Exercise 6.7.1 for the IVPs in Exercise 6.1.5.

6.7.3. Show that the trapezoid rule (89) is a second-order method.

6.7.4. Find a two-step, third-order explicit method. Is the method stable? [Ans. $w_{i+1} = -4w_i + 5w_{i-1} + h[4f_i + 2f_{i-1}]$, NO.]

6.7.5. Find a second-order two-step explicit method whose characteristic polynomial has a double root at 1.

6.7.6. Explain why the characteristic polynomial of an explicit or implicit s-step method must have a root at 1.

6.7.7. (a) For which a_1 does there exist a strongly stable second-order two-step explicit method? (b) Same question for weakly stable. [Ans. (a) $0 < a_1 < 2$ (b) $a_1 = 0, 2$]

6.7.8. Show that the coefficients of the two-step Adams-Moulton implicit method satisfy (92) and that the method is strongly stable.

6.7.9. Find the order and stability type of the following two-step implicit methods.

(a) $w_{i+1} = 3w_i - 2w_{i-1} + \frac{h}{12}[13f_{i+1} - 20f_i - 5f_{i-1}]$

(b) $w_{i+1} = \frac{4}{3}w_i - \frac{1}{3}w_{i-1} + \frac{2}{3}hf_{i+1}$

(c) $w_{i+1} = \frac{4}{3}w_i - \frac{1}{3}w_{i-1} + \frac{h}{9}[4f_{i+1} + 4f_i - 2f_{i-1}]$

(d) $w_{i+1} = 3w_i - 2w_{i-1} + \frac{h}{12}[7f_{i+1} - 8f_i - 11f_{i-1}]$

(e) $w_{i+1} = 2w_i - w_{i-1} + \frac{h}{2}[f_{i+1} - f_{i-1}]$

[Ans. (a) second order unstable (b) second order strongly stable (c) third order strongly stable (d) third order unstable (e) third order weakly stable]

6.7.10. Find a second-order two-step implicit method that is weakly stable. [Ans. For example, $a_1 = 0, a_2 = 1, b_1 = 2 - b_0, b_2 = b_0$, where $b_0 \neq 0$ is arbitrary.]

6.7.11. Find a third-order two-step implicit method that is weakly stable.

6.7.12. (a) Find the conditions (analogous to (77)) on a_i, b_i required for a third-order, three-step explicit method. (b) Show that the Adams-Bashforth three-step method satisfies the conditions. (c) Show that the Adams-Bashforth three-step method is strongly stable. (d) Find a weakly-stable third-order three-step explicit method and verify these properties.

6.7.13. (a) Find the conditions (analogous to (77)) on a_i, b_i required for a fourth-order, four-step explicit method. (b) Show that the Adams-Bashforth four-step method satisfies the conditions. (c) Show that the Adams-Bashforth four-step method is strongly stable.

6.7.14. (a) Find the conditions (analogous to (77)) on a_i, b_i required for a fourth-order, three-step implicit method. (b) Show that the Adams-Moulton three-step method satisfies the conditions. (c) Show that the Adams-Moulton three-step method is strongly stable.

6.7.15. Derive Simpson's method (94) from (92) and show that it is fourth-order and weakly stable.

Computer Problems 6.7

6.7.1. Adapt the `exmultistep.m` program to apply the Adams-Bashforth method to the IVPs in Exercise 6.1.4. Using stepsize $h = 0.1$, calculate the approximation on the interval $[0, 1]$. Compare to the correct solution, and find the total error at each step.

6.7.2. Repeat Exercise 6.7.1 for the IVPs in Exercise 6.1.5.

6.7.3. Repeat Exercise 6.7.1 using the weakly-stable 2-step method, and compare results with those from the Adams-Bashforth 2-step method.

6.7.4. Repeat Exercise 6.7.1 using the Adams-Bashforth 3-step method.

6.7.5. Change Program 6.4 into a third-order predictor-corrector method, using Adams-Bashforth three-step method and the Adams-Moulton two-step method. Apply to the solution of IVP (9) using step size $h = 0.05$.

6.7.6. Same as previous problem, but with a fourth-order predictor-corrector method.

6.8 Software and Further Reading

TRADITIONAL sources for fundamentals on ordinary differential equations are [5, 6, 4, 10, 16]. Many books teach the basics of ODE's along with ample computational and graphical help; we mention ODE Architect [8] as a good example. Polking's Matlab codes and manual [19] are an excellent way to learn and visualize ODE concepts.

To supplement our tour through one-step and multi-step numerical methods for solving ordinary differential equations, there are many intermediate and advanced texts. The texts [14, 11]

are classics. A contemporary Matlab approach is taken by [21]. Other recommended texts are [15, 20, 1, 17, 9, 7] and the comprehensive two volume set [12, 13].

There is a great deal of sophisticated software available for solving ODEs. Details on the solvers used by Matlab can be found in [22, 2]. Variable stepsize explicit methods of the Runge-Kutta type are usually successful for non-stiff or mildly stiff problems. In addition to Runge-Kutta-Fehlberg and Dormand-Prince, the variant Runge-Kutta-Verner, an order 5/6 method, is often used. For stiff problems, backward-difference methods and extrapolation methods are typically used.

The IMSL library includes the double precision routine DIVPRK, based on the Runge-Kutta-Verner method, and DIVPAG for a multistep Adams-type method that can handle stiff problems.

The NAG library provides a driver routine D02BJF that runs standard Runge-Kutta steps. The multistep driver is D02CJF, which includes Adams-style programs with error control. For stiff problems, the D02EJF routine is recommended, where the user has an option to specify the Jacobian for faster computation.

The Netlib repository contains a Fortran routine RKF45 for the Runge-Kutta-Fehlberg method and DVERK for the Runge-Kutta-Verner method. The Netlib package ODE contains several multistep routines. The routine VODE handles stiff problems.

The collection ODEPACK is a public domain set of Fortran code implementing ODE solvers developed at Lawrence Livermore National Laboratory (LLNL). The basic solver LSODE and its variants are suitable for stiff and non-stiff problems. The routines are freely available at the LLNL website `http://ww.llnl.gov/CASC/odepack`.

References

[1] U.M. Ascher, L. Petzold. Computer methods for ordinary differential equations and differential-algebraic equations. SIAM, Philadelphia (1998).

[2] R. Ashino, M. Nagase, R. Vaillancourt, Behind and beyond the Matlab ODE Suite. CRM-2651, Jan. 2000. (and refs therein)

[3] G. Birkhoff, G. Rota, Ordinary differential equations, 4th Edition. J. Wiley & Sons (1989).

[4] P. Blanchard, R. Devaney, and G.R. Hall, Differential Equations, 2nd Ed. Brooks-Cole (2002).

[5] W.E. Boyce, R.C. DiPrima, Elementary differential equations and boundary value problems, 7th Ed. J. Wiley & Sons (2003).

[6] M. Braun, Differential equations and their applications, 4th ed. New York: Springer-Verlag (1993).

[7] J.C. Butcher, Numerical analysis of ordinary differential equations. Wiley, London (1987).

[8] CODEE, ODE architect. J. Wiley & Sons (1999).

[9] J. R. Dormand, Numerical methods for differential equations. CRC Press. Boca Raton, FL (1996).

[10] Edwards and Penny, Differential equations and boundary value problems, 5th Ed. Prentice Hall (2004).

[11] C.W. Gear, Numerical initial value problems in ordinary differential equations. Prentice-Hall (1971).

[12] E. Hairer, S.P. Norsett and G. Wanner, Solving ordinary differential equations I: Nonstiff problems, 2nd ed., Springer Verlag, Berlin (1993).

[13] E. Hairer and G. Wanner, Solving ordinary differential equations II: Stiff and differential-algebraic problems, 2nd ed., Springer Verlag, Berlin (1996).

[14] P. Henrici, Discrete variable methods in ordinary differential equations. J. Wiley & Sons (1962).

[15] A. Iserles, A first course in the numerical analysis of differential equations, Cambridge University Press (1996).

[16] E. Kostelich, D. Armbruster, Introductory differential equations: From linearity to chaos. Addison Wesley (1997).

[17] John Denholm Lambert, Numerical methods for ordinary differential systems, John Wiley & Sons, Chichester (1991).

[18] P.J. McKenna, C. Tuama, Large torsional oscillations in suspension bridges visited again: Vertical forcing creates torsional response. Amer. Math. Monthly **108**, 738-745 (2001).

[19] J. Polking, Ordinary differential equations using Matlab. Prentice Hall (1999).

[20] L.F. Shampine, Numerical solution of ordinary differential equations. Chapman & Hall, New York (1994).

[21] L.F. Shampine, I. Gladwell, S. Thompson. Solving ODEs with Matlab. Cambridge University Press (2003).

[22] L.F. Shampine, M.W. Reichelt. The Matlab ODE suite. SIAM J. Sci. Computing **18**, 1-22 (1997).