

5.

Numerical Methods for ODEs

Numerical methods for solving ordinary differential equations are discussed in many textbooks. Here we will discuss how to use some of them in MATLAB. In particular, we will examine how a reduction in the “step size” used by a particular algorithm reduces the error of the numerical solution, but only at a cost of increased computation time. In line with the philosophy that we are not emphasizing programming in this manual, MATLAB routines for these numerical methods are made available.

Our discussions of numerical methods will be very brief and incomplete. The assumption is that the reader is using a textbook in which these methods are described in more detail.

Euler’s Method

We want to find an approximate solution to the initial value problem $y' = f(t, y)$, with $y(a) = y_0$, on the interval $[a, b]$. In Euler’s very geometric method, we go along the tangent line to the graph of the solution to find the next approximation. We start by setting $t_0 = a$ and choosing a step size h . Then we inductively define

$$\begin{aligned}y_{k+1} &= y_k + hf(t_k, y_k), \\t_{k+1} &= t_k + h,\end{aligned}\tag{5.1}$$

for $k = 0, 1, 2, \dots, N$, where N is large enough so that $t_N \geq b$. This algorithm is available in the MATLAB command `eul`¹.

Example 1. Use Euler’s method to plot the solution of the initial value problem

$$y' = y + t, \quad y(0) = 1,\tag{5.2}$$

on the interval $[0, 3]$.

Note that the differential equation in (5.2) is in normal form $y' = f(t, y)$, where $f(t, y) = y + t$. Before invoking the `eul` routine, you must first write a function M-file

```
function yprime = yplust(t,y)
yprime = y + t;
```

to compute the right-hand side $f(t, y) = y + t$. Save the function M-file with the name `yplust.m`. Before continuing, it is always wise to test that your function M-file is returning the proper output. For example, $f(1, 2) = 2 + 1$, or 3, and

```
>> yplust(1,2)
ans = 3
```

¹ The function `eul`, as well as `rk2` and `rk4` described later in this chapter, are function M-files which are not distributed with MATLAB. For example, type `help eul` to see if `eul` is installed correctly on your computer. If not, see the Appendix to Chapter 3 for instructions on how to obtain them. The M-files defining these commands are printed in the appendix to this chapter for the illumination of the reader.

If you don't receive this output, check your function `yplust` for coding errors. If all looks well, and you still aren't getting the proper output, refer to the section headed **The MATLAB Path** in Chapter 4.

Now that your function M-file is operational, it's time to invoke the `eul` routine. The general syntax is

```
[t,y]=eul(@yplust,tspan,y0,stepsize)
```

where `yplust` is the name of the function M-file, `tspan` is the vector `[t0,tfinal]` containing the initial and final time conditions, `y0` is the `y`-value of the initial condition, and `stepsize` is the step size to be used in Euler's method.² Actually, the step size is an optional parameter. If you enter

```
[t,y]=eul(@yplust,tspan,y0)
```

the program will choose a default step size equal to $(t_{\text{final}}-t_0)/100$.

It is also possible to invoke `eul` with the command `[t,y]=eul('yplust',tspan,y0,stepsize)`. The only difference is the use of `'yplust'` instead of `@yplust`. If you are using a version of MATLAB prior to version 6, you have no choice. You must use `'yplust'`. With version 6 you are given the choice. Using the *function handle* `@yplust` will result in speedier solution. This speedup is not at all important for the simple equations we solve in this chapter. However, it becomes very important when you are solving more complicated systems of ODEs. The use of function handles also provides us with more flexibility of use, as we will explain in Chapter 7. We will systematically use the function handle notation, leaving it to you to make the replacement, if necessary.

The following commands will produce an approximate solution of the initial value problem (5.2) similar to that shown in Figure 5.1.

```
>> [t,y] = eul(@yplust,[0,3],1,1);  
>> plot(t,y,'.-')
```

We can arrange the output in two columns with the command

```
>> [t,y]  
ans =  
    0     1  
    1     2  
    2     5  
    3    12
```

Notice that the t_k 's given in the first column differ by 1, the stepsize. The y_k 's computed inductively by the algorithm in (5.1) are in the second column. In Figure 5.1 these four points are plotted and connected with straight lines.

² You can use an inline function defined with the command `f = inline('y+t','t','y');` instead of the function M-file. Then the syntax for using `eul` is `[t,y] = eul(f,tspan,y0,stepsize)`.

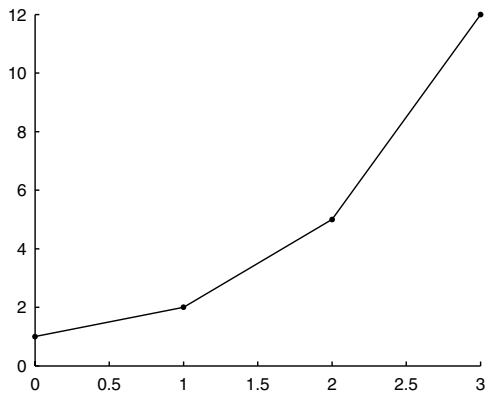


Figure 5.1. Euler's solution of initial value problem (5.1).

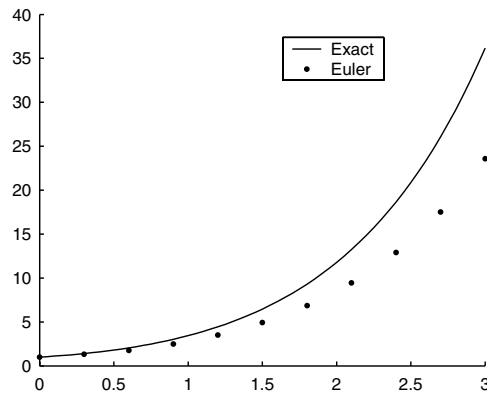


Figure 5.2. Euler versus exact solution (Step size $h = 0.3$).

Euler's Method Versus the Exact Solution

In this section we want to examine visually the error in the numerical solution obtained from Euler's method.

Example 2. *Re-examine the initial value problem*

$$y' = y + t, \quad y(0) = 1. \quad (5.3)$$

Compare the exact solution and the numerical solution obtained from Euler's method on the interval $[0, 3]$. Use a step size of $h = 0.3$.

The initial value problem in (5.3) is linear and easily solved:

$$y = -t - 1 + 2e^t. \quad (5.4)$$

The Euler's method solution of the initial value problem in (5.3) with step size 0.3 is found by

```
>> h = 0.3;
>> [teuler,yeuler] = eul(@yplust,[0,3],1,h);
```

We compute the exact solution using equation (5.4), but we want to choose a finer time increment on the interval $[0, 3]$ so that the exact solution will have the appearance of a smooth curve.

```
>> t = 0:.05:3;
>> y = -t - 1 + 2*exp(t);
```

The command

```
>> plot(t,y,teuler,yeuler,'.')
>> legend('Exact','Euler')
```

produces Figure 5.2, a visual image of the accuracy of Euler's method using the step size $h = 0.3$. The accuracy is not too good in this case.

Changing the Step Size — Using Script M-Files

We would like to repeat the process in Example 2 with different step sizes to analyze graphically how step size affects the accuracy of the approximation.

Example 3. Examine the numerical solutions provided by Euler's method for the initial value problem

$$y' = y + t, \quad y(0) = 1,$$

on the interval $[0, 3]$. Use step sizes $h = 0.2, 0.1, 0.05,$ and 0.025 .

It will quickly get tedious to type the required commands into the Command Window. This is a good place to use a script M-file, as discussed in Chapter 2. We collect all of the commands and type them once into an editor. To be precise, this file should contain the lines

```
[teuler,yeuler] = eul(@yplust,[0,3],1,h);  
t = 0:.05:3;  
y = -t - 1 + 2*exp(t);  
plot(t,y,teuler,yeuler,'.')
```

```
legend('Exact','Euler')  
shg
```

We save the file as `batch1.m` (or whatever else we want to call it as long as it meets the requirements for function names). Then executing `>> h = 0.2; batch1` produces Figure 5.3, while the command `>> h = 0.05; batch1` produces Figure 5.5.

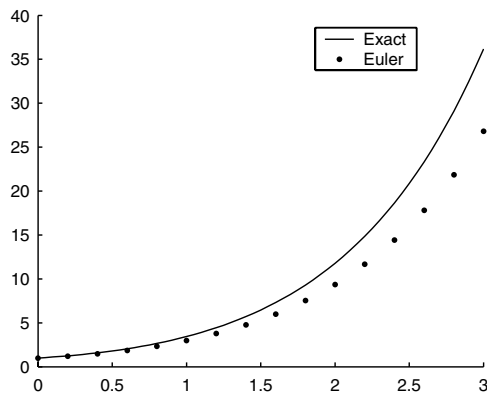


Figure 5.3. Euler versus exact solution (Step size $h = 0.2$).

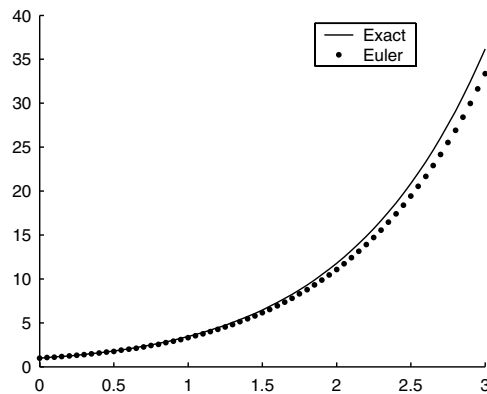


Figure 5.4. Euler versus exact solution (Step size $h = 0.05$).

Repeating this command with various values of the step size h allows us to visually examine the effect of choosing smaller and smaller step sizes. Note how the error decreases, but the computation time increases as you reduce the step size.

Numerical Error Analysis

In this section we analyze the error inherent in Euler's method numerically as well as graphically by computing the *absolute error* at each step, defined as the absolute value of the difference between the actual solution and the numerical solution at that step. We will continue to look at the error graphically.

Example 4. Consider again the initial value problem

$$y' = y + t, \quad y(0) = 1,$$

on the interval $[0, 3]$. Record the maximum error in the Euler's solution for the step sizes $h = 0.2, 0.1, 0.05,$ and 0.025 .

We record the error at each step by adjusting the script file `batch1.m` from Example 3. We need to evaluate the exact solution at the same points at which we have the approximate values calculated, i.e. at the points in the vector `teuler`. This is easy. Simply enter `z = -teuler - 1 + 2*exp(teuler)`. Then to compare with the approximate values, we look at the difference of the two vectors `z` and `yeuler`. We are only interested in the magnitude of the error and not the sign. The command `abs(z-yeuler)` yields a vector containing the absolute value of the error made at each step of the Euler computation. Finally, we compute the largest of these errors with the MATLAB command `maxerror = max(abs(z-yeuler))`. We enter this command without an ending semi-colon, because we want to see the result in the command window. Therefore, to effect these changes we alter the file `batch1.m` to

```
[teuler,yeuler] = eul(@yplust,[0,3],1,h);
t = 0:.05:3;
y = -t - 1 + 2*exp(t);
plot(t,y,teuler,yeuler, '.')
legend('Exact','Euler')
shg
z = -teuler - 1 + 2*exp(teuler);
maxerror = max(abs(z - yeuler))
```

Save this file as `batch2.m`. A typical session might show the following in the command window:

```
>> h = 0.2; batch2
maxerror =
    9.3570
>> h = 0.1; batch2
maxerror =
    5.2723
>> h = 0.05; batch2
maxerror =
    2.8127
```

Of course, each of these commands will result in a visual display of the error in the figure window as well. Thus, we can very easily examine the error in Euler's method both graphically and quantitatively.

If you have executed all of the commands up to now, you will have noticed that decreasing the step size has the effect of reducing the error. But at what expense? If you continue to execute `batch2`, halving

the step size each time, you will notice that the routine takes more and more time to complete. In fact, halving the step size doubles the number of computations that must take place and therefore doubles the time needed. Therefore, decreasing the step size used in Euler's method is expensive, in terms of computer time. However, notice that halving the step size results in a maximum error almost exactly half the previous one.

At this point it would be illuminating to see a plot of how the maximum error changes versus the step size.

Example 5. Consider again the initial value problem

$$y' = y + t, \quad y(0) = 1.$$

Sketch a graph of the maximum error in Euler's method versus the step size.

To do this using MATLAB, we have to devise a way to capture the various step sizes and the associated errors into vectors and then use an appropriate plotting command. We will call the vector of step sizes `h_vect` and the vector of errors `err_vect`. The following modification of `batch2.m` will do the job. Save the file as `batch3.m`

```
h_vect = []; err_vect = []; h = 1;
t = 0:.05:3;
y = -t - 1 + 2*exp(t);
for k = 1:8
    [teuler,yeuler] = eul(@yplust,[0,3],1,h);
    plot(t,y,teuler,yeuler,'.')
    legend('Exact','Euler',2)
    shg
    z = -teuler - 1 + 2*exp(teuler);
    maxerror = max(abs(z - yeuler));
    h_vect = [h_vect,h]
    err_vect = [err_vect,maxerror]
    pause
    h = h/2;
end
```

The first line initializes `h_vect`, `err_vect`, and the starting step size `h = 1`. Since at the beginning they contain no information, `h_vect` and `err_vect` are defined to be empty matrices, as indicated by `[]`. The next two lines compute the exact solution for later plotting. The main work is done in the `for` loop. This comprises all of the indented steps between `for k = 1:8` and `end`. The first four lines compute the approximate solution, and plot it in comparison with the exact solution. Notice the third input to the `legend` command. This results in the legend being placed in the upper left corner, where it does not interfere with the graph. The maximum error is computed next and the new step size and error are added to the vectors `h_vect` and `err_vect`. The command `pause` stops the execution and gives you time to observe the figure and the output in the command window. When you are finished examining the figure and output, hit any key (the space bar will do) to advance to the next step. The final line in the `for` loop halves the step size in preparation for the next step. Notice that the `for` loop is executed 8 times with different values of `k`

Now we are ready to go. Executing `batch3` results in a visual display similar to Figures 5.3 or 5.4 at each step. There is also output on the command window because we have not ended the `h_vect = ...` and `err_vect = ...` commands with semi-colons. By hitting a key twice, we get three steps of the for loop, and the output

```
>> batch3
h_vect =
     1
err_vect =
    24.1711
h_vect =
    1.0000    0.5000
err_vect =
    24.1711    17.3898
h_vect =
    1.0000    0.5000    0.2500
err_vect =
    24.1711    17.3898    11.0672
```

at the command line. Continuing to step through `batch3`, we get more and more data. After the seventh iteration the output is

```
h_vect =
    1.0000    0.5000    0.2500    0.1250    0.0625    0.0313    0.0156
err_vect =
    24.1711    17.3898    11.0672    6.3887    3.4581    1.8030    0.9211
```

Notice that the maximum error is approximately halved on each step, at least on the last few steps.

After all eight steps we are ready to plot the error data. This can be done with any of the commands `plot`, `semilogx`, `semilogy`, or `loglog`. Try them all to see which you like best. However, since the maximum error seems to be halved each time the step size is halved, we expect a power law relationship. Therefore we will use a `loglog` graph. The commands

```
>> loglog(h_vect,err_vect)
>> xlabel('Step size')
>> ylabel('Maximum error')
>> title('Maximum error vs. step size for Euler''s method')
>> grid
>> axis tight
```

produce Figure 5.5. The command `axis tight` sets the axis limits to the range of the data.

Notice that the graph of the logarithm of the maximum error versus the logarithm of the step size is approximately linear. If the graph were truly linear, we would have a linear relationship between the logarithms of the coordinates, like

$$\log_{10}(\text{maximum_error}) = A + B \log_{10}(\text{step_size}),$$

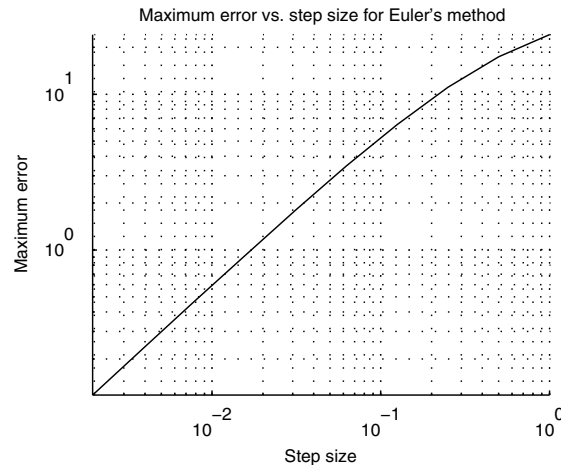


Figure 5.5. Analysis of the error in Euler’s method.

for some constants A and B .³

If we exponentiate this relationship, we get

$$\text{maximum_error} = 10^A \cdot (\text{step_size})^B = C(\text{step_size})^B,$$

where $C = 10^A$. Thus a linear relationship with slope B on a loglog graph means that there is a power law relationship between the coordinates with the exponent being B . Indeed, we will see in Exercise 21 that Euler’s method is a first order method. This means that if y_k is the calculated approximation at t_k , and $y(t_k)$ is the exact solution evaluated at t_k , then $|y(t_k) - y_k|$ is the error being made, and it satisfies an inequality of the form

$$|y(t_k) - y_k| \leq C|h|, \quad \text{for all } k, \tag{5.5}$$

where C is a constant. This inequality satisfied by the error is reflected in the nearly linear relationship we see in Figure 5.5, and it is the reason why we chose a loglog graph.

Hopefully, you now understand two key points about Euler’s method. You can increase the accuracy of the solution by decreasing the step size, but you pay for the increase in accuracy with an increase in the number of computational steps and therefore in the computing time.

The Second Order Runge-Kutta Method

What is needed are solution routines that will provide more accuracy without having to drastically reduce the step size and therefore increase the computation time. The Runge-Kutta routines are designed to do just that. The simplest of these, the Runge-Kutta method of order two, is sometimes called the improved Euler’s method. Again, we want to find a approximate solution of the initial value problem

³ We are using the logarithm to base 10 instead of the natural logarithm because it is the basis of the loglog graph. However there is very little difference in this case. Since $\ln x = \log_{10} x \cdot \ln 10$, the inequality is true with \log_{10} replaced by \ln , with the same slope B and A replaced by $A \cdot \ln 10$.

$y' = f(t, y)$, with $y(a) = y_0$, on the interval $[a, b]$. As before, we set $t_0 = a$, choose a step size h , and then inductively define

$$\begin{aligned} s_1 &= f(t_k, y_k), \\ s_2 &= f(t_k + h, y_k + hs_1), \\ y_{k+1} &= y_k + h(s_1 + s_2)/2, \\ t_{k+1} &= t_k + h, \end{aligned}$$

for $k = 0, 1, 2, \dots, N$, where N is large enough so that $t_N \geq b$. This algorithm is available in the M-file `rk2.m`, which is listed in the appendix of this chapter.

The syntax for using `rk2` is exactly the same as the syntax used for `eu1`. As the name indicates, it is a second order method, so, if y_k is the calculated approximation at t_k , and $y(t_k)$ is the exact solution evaluated at t_k , then there is a constant C such that

$$|y(t_k) - y_k| \leq C|h|^2, \quad \text{for all } k. \quad (5.6)$$

Notice that the error is bounded by a constant times the square of the step size. As we did for Euler's method, we have a power inequality, but now the power is 2 instead of 1. We should expect a nearly linear relationship between the maximum error and the step size in a loglog graph, but with a steeper slope than we observed for Euler's method.

The error in the second order Runge-Kutta method can be examined experimentally using the same techniques that we illustrated for Euler's method. Modify the script files, `batch1.m`, `batch2.m`, and `batch3.m`, replacing the command `eu1` with `rk2` everywhere it occurs, and changing all references to `euler` to refer to `rk2` instead. You might also want to change the names of the files to indicate that they now refer to `rk2`. Replay the commands in Examples 3, 4, and 5 using your newly modified routines.

The Fourth Order Runge-Kutta Method

This is the final method we want to consider. We want to find an approximate solution to the initial value problem $y' = f(t, y)$, with $y(a) = y_0$, on the interval $[a, b]$. We set $t_0 = a$ and choose a step size h . Then we inductively define

$$\begin{aligned} s_1 &= f(t_k, y_k), \\ s_2 &= f(t_k + h/2, y_k + hs_1/2), \\ s_3 &= f(t_k + h/2, y_k + hs_2/2), \\ s_4 &= f(t_k + h, y_k + hs_3), \\ y_{k+1} &= y_k + h(s_1 + 2s_2 + 2s_3 + s_4)/6, \\ t_{k+1} &= t_k + h, \end{aligned}$$

for $k = 0, 1, 2, \dots, N$, where N is large enough so that $t_N \geq b$. This algorithm is available in the M-file `rk4.m`, which is listed in the appendix of this chapter.

The syntax for using `rk4` is exactly the same as that required by `eu1` or `rk2`. As the name indicates, it is a fourth order method, so, if y_k is the calculated approximation at t_k , and $y(t_k)$ is the exact solution evaluated at t_k , there is a constant C such that

$$|y(t_k) - y_k| \leq C|h|^4, \quad \text{for all } k. \quad (5.7)$$

The error in the fourth order Runge-Kutta method can be examined experimentally using the same techniques that we illustrated for Euler's and the Runge-Kutta 2 methods. Again, you can easily modify the script files, `batch1.m`, `batch2.m`, and `batch3.m` to create files to evaluate the errors in the Runge-Kutta 4 algorithm. Since the error is bounded by a constant times the fourth power of the step size in (5.7), we again expect a nearly linear relationship between the maximum error and the step size in a loglog graph, but with a much steeper slope than before.

Comparing Euler, RK2, and RK4

It is very interesting to compare the accuracy of these three methods, both for individual step sizes, and for a range of step sizes.

Example 6. Consider again the initial value problem

$$y' = y + t, \quad y(0) = 1.$$

Sketch a graph of the maximum error in each of the three methods (Euler, RK2, RK4) versus the step size.

This can be done by writing a script M-file that is only a minor modification of `batch3`. Create a script M-file with the contents

```

h = 1; eul_vect = []; rk2_vect = []; rk4_vect = []; h_vect=[];
t = 0:.05:3;
y = -t - 1 + 2*exp(t);
for k=1:8
    [teuler,yeuler] = eul(@yplust,[0,3],1,h);
    [trk2,yrk2] = rk2(@yplust,[0,3],1,h);
    [trk4,yrk4] = rk4(@yplust,[0,3],1,h);
    plot(t,y,teuler,yeuler,'.',trk2,yrk2,'+',trk4,yrk4,'x')
    legend('Exact','Euler','RK2','RK4',2)
    shg
    zeuler = -teuler - 1 + 2*exp(teuler);
    eulerror = max(abs(zeuler - yeuler));
    zrk2 = -trk2 - 1 + 2*exp(trk2);
    rk2error = max(abs(zrk2 - yrk2));
    zrk4 = -trk4 - 1 + 2*exp(trk4);
    rk4error = max(abs(zrk4 - yrk4));
    h_vect = [h_vect,h]
    eul_vect = [eul_vect,eulerror]
    rk2_vect = [rk2_vect,rk2error]
    rk4_vect = [rk4_vect,rk4error]
    pause
    h = h/2;
end
loglog(h_vect,eul_vect,h_vect,rk2_vect,h_vect,rk4_vect)
legend('eul','rk2','rk4',4)
grid, xlabel('Step size'), ylabel('Maximum error')
title('Maximum error vs. step size'), axis tight

```

Save the file as `batch4.m`.

Most of the features of `batch4` are already present in `batch3`. The major difference is that we have added the plot commands for the loglog graph at the end, so that it will automatically be executed when the error data has been computed. Executing all eight steps in `batch4` will result in a plot of the error curves for each of the three methods, as shown in Figure 5.6.⁴

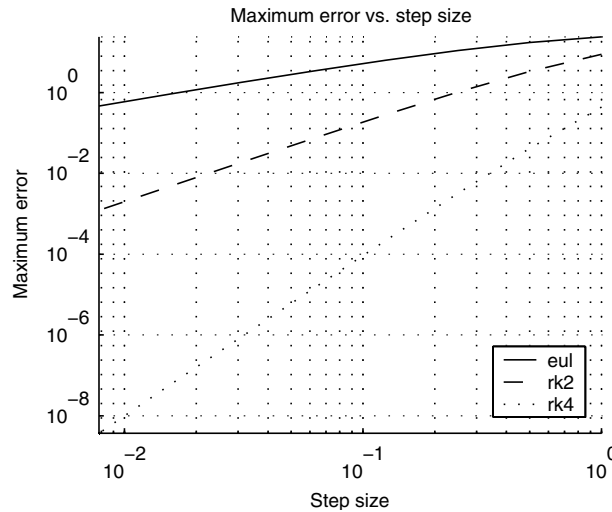


Figure 5.6. Error comparison for Euler, RK2, and RK4.

Note that Euler's method has the largest error for a given step size, with Runge-Kutta 2 being slightly better, while Runge-Kutta 4 is significantly better than the other two. Also notice that the curves are nearly straight lines in the loglog plot in Figure 5.6, indicating a power function relationship between the error and the stepsize, as predicted by the inequalities (5.5), (5.6), and (5.7). The slopes of the curves are also roughly comparable to the order of the corresponding method. (Exercises 13–20 demonstrate the relation between a power function and its loglog plot. Exercise 21 experimentally determines the orders of the methods `eu1`, `rk2`, and `rk4`)

Exercises

In Exercises 1–6 perform the following tasks for the given initial value problem on the specified interval.

- Find the exact solution.
- Use MATLAB to plot on a single figure window the graph of the exact solution, together with the plots of the solutions using each of the three methods (`eu1`, `rk2`, and `rk4`) with the given step size h . Use a distinctive marker (type `help plot` for help on available markers) for each method. Label the graph appropriately and add a legend to the plot.

⁴ As you execute `batch4.m` repeatedly, you will note that the vector `rk4_vect` appears to have zero entries except for the first two. Is the error from `rk4` actually zero at this point? The answer is no. To see what the entries really are we need to print the vector with more accuracy. To do this, enter `format long` and then `rk4_vect` to see what you get.

1. $x' = x \sin(3t)$, with $x(0) = 1$, on $[0, 4]$; $h = 0.2$.
2. $y' = (1 + y^2) \cos(t)$, with $y(0) = 0$, on $[0, 6]$; $h = 0.5$.
3. $z' = z^2 \cos(2t)$, with $z(0) = 1$, on $[0, 6]$; $h = 0.25$.
4. $x' = x(4 - x)/5$, with $x(0) = 1$, on $[0, 6]$; $h = 0.5$.
5. $y' = 2ty$, with $y(0) = 1$, on $[0, 2]$; $h = 0.2$.
6. $z' = (1 - z)t^2$, with $z(0) = 3$, on $[0, 2]$; $h = 0.25$.

Using the code of Example 6 as a template, adjust the code to run with each of the initial value problems in Exercises 7–12. Note that you will first have to find an analytical solution for the given IVP. Observe how the approximations improve at each iteration and obtain a printout of the final loglog graph.

7. $y' = -y + 3t^2 e^{-t}$, with $y(0) = -1$, on $[0, 6]$.
8. $y' = 1 + 2y/t$, with $y(1) = 1$, on $[1, 4]$.
9. $y' = 3 - 2y/t$, with $y(1) = -1$, on $[1, 4]$.
10. $y' = y/t - y^2$, with $y(1) = 1/2$, on $[1, 4]$. *Hint: Look up Bernoulli's equation in your text.*
11. $y' = y + 2 \cos t$, with $y(0) = -1$, on $[0, 2\pi]$.
12. $y' = 2te^t + y$, with $y(0) = -1$, on $[0, 2]$.

The function defined by the equation $y = a \cdot x^b$ is called a *power function*. You can plot the power function $y = 10x^2$ on the interval $[0.1, 2]$ with the commands `x = linspace(0.1,2)`; `y = 10*x.^2`; `plot(x,y)`. Take the logarithm base ten of both sides of $y = 10x^2$ to get $\log_{10} y = 1 + 2 \log_{10} x$. Thus, if you plot $\log_{10} y$ versus $\log_{10} x$, the graph should be a line with slope 2 and intercept 1. You can create a loglog plot with the commands `figure`, `plot(log10(x),log10(y))`, `grid on`, then compare the slope and intercept of the result with the predicted result. Follow this procedure for each of the power functions in Exercises 13–16. In each case, plot over the interval $[0.1, 2]$.

13. $y = x^3$
14. $y = 100x^4$
15. $y = 10x^{-2}$
16. $y = 1000x^{-5}$

MATLAB represents polynomials by placing the coefficients of the polynomial in a vector, always assuming that the polynomial is arranged in descending powers. For example, the vector `p = [2, -5, 3, -4]` would be used to represent the polynomial $p(x) = 2x^3 - 5x^2 + 3x - 4$. The MATLAB function `POLYFIT(X,Y,N)` will fit a polynomial of degree `N` to the data stored in `X` and `Y`. Take the logarithm base ten of both sides of the power function $y = 10x^3$ to get $\log_{10} y = 1 + 3 \log_{10} x$. As we saw in Exercises 13–16, the graph of $\log_{10} y$ versus $\log_{10} x$ is a line with slope 3 and intercept 1, plotted with the commands `x = linspace(0.1,2)`; `y = 10*x.^3`; `plot(log10(x), log10(y))`, `grid on`. One can read the slope and intercept from the plot, or you can allow MATLAB to calculate them with `p = polyfit(log10(x),log10(y),1)`, which responds with the polynomial `p = [3 1]`. The coefficients of this polynomial represent the slope and intercept, respectively, of $\log_{10} y = 3 \log_{10} x + 1$. Exponentiating base ten, we recapture the power function $y = 10x^3$. In Exercises 17–20, follow this procedure to perform a loglog plot (base ten) of the given power function, then use the `polyfit` command to find the slope and intercept of the resulting line. Exponentiate the resulting equation to obtain the original power function.

17. $y = 10x^2$
18. $y = 100x^3$
19. $y = 10x^{-3}$
20. $y = 1000x^{-4}$
21. Execute `batch4.m` of Example 6 to regenerate the data sets and the plot shown in Figure 5.6. Now plot `log10(rk4_vect)` versus `log10(h_vect)` with the command `plot(log10(h_vect),log10(rk4_vect), 'ro')`. The points produced by this plot command should appear to maintain a linear relationship. Use the following code to determine and plot the “line of best fit” through these data points (type `help polyfit`

and help polyval to get a full explanation of these commands).

```
hold on %hold the prior plot
p = polyfit(log10(h_vect),log10(rk4_vect),1)
plot(log10(h_vect), polyval(p,log10(h_vect)), 'g')
```

Write the equation of the line of best fit in the form $\log_{10}(\text{rk4_vect}) = C \log_{10}(\text{h_vect}) + B$, where C and B are the slope and intercept captured from the polyfit command. Solve this last equation for rk4_vect. How closely does this last equation compare with the inequality (5.7)?

- a) Perform a similar analysis on eul_vect and h_vect and compare with the inequality (5.5).
 - b) Perform a similar analysis on rk2_vect and h_vect and compare with the inequality (5.6).
22. Simple substitution will reveal that $x(t) = \pi/2$ is a solution of $x' = e^t \cos x$. Moreover, both $f(t, x) = e^t \cos x$ and $\partial f / \partial x = -e^t \sin x$ are continuous everywhere. Therefore, solutions are unique and no two solutions can ever share a common point (solutions cannot intersect). Use Euler's method with a step size $h = 0.1$ to plot the solution of $x' = e^t \cos x$ with initial condition $x(0) = 1$ on the interval $[0, 2\pi]$ and hold the graph (hold on). Overlay the graph of $x = \pi/2$ with the command `line([0, 2*pi], [pi/2, pi/2], 'color', 'r')` and note that the solutions intersect. Does reducing the step size help? If so, does this reduced step size hold up if you increase the interval to $[0, 10]$? How do rk2 and rk4 perform on this same problem?
23. The accuracy of any numerical method in solving a differential equation $y' = f(t, y)$ depends on how strongly the equation depends on the variable y . More precisely, the constants that appear in the error bounds depend on the derivatives of f with respect to y . To see this experimentally, consider the two initial value problems

$$\begin{aligned} y' &= y, & y(0) &= 1, \\ y' &= e^t, & y(0) &= 1. \end{aligned}$$

You will notice that the two problems have the same solution. For each of the three methods described in this chapter compute approximate solutions to these two initial value problems over the interval $[0, 1]$ using a step size of $h = 0.01$. For each method compare the accuracy of the solution to the two problems.

24. Remember that $y(t) = e^t$ is the solution to the initial value problem $y' = y$, $y(0) = 1$. Then $e = e^1$, and in MATLAB this is `e = exp(1)`. Suppose we try to calculate e approximately by solving the initial value problem, using the methods of this chapter. Use step sizes of the form $1/n$, where n is an integer. For each of Euler's method, the second order Runge-Kutta method, and the fourth order Runge-Kutta method, how large does n have to be to get an approximation e_{app} which satisfies $|e_{\text{app}} - e| \leq 10^{-3}$?
25. In the previous problem, show that the approximation to e using Euler's method with step size $1/n$ is $(1+1/n)^n$. As a challenge problem, compute the formulas for the approximations using the two Runge-Kutta methods.

Appendix: M-files for Numerical Methods

In this appendix we provide a listing of the programs eul.m, rk2.m, and rk4.m used in the chapter. The first listing is the file eul.m.

```
function [tout, yout] = eul(FunFcn, tspan, y0, ssize)
%
% EUL Integrates a system of ordinary differential equations using
% Euler's method. See also ODE45 and ODEDEMO.
% [t,y] = eul('yprime', tspan, y0) integrates the system
% of ordinary differential equations described by the M-file
% yprime.m over the interval tspan = [t0,tfinal] and using initial
% conditions y0.
% [t, y] = eul(F, tspan, y0, ssize) uses step size ssize
```

```

% INPUT:
% F      - String containing name of user-supplied problem description.
%         Call: yprime = fun(t,y) where F = 'fun'.
%         t      - Time (scalar).
%         y      - Solution vector.
%         yprime - Returned derivative vector; yprime(i) = dy(i)/dt.
% tspan = [t0, tfinal], where t0 is the initial value of t, and tfinal is
%         the final value of t.
% y0     - Initial value vector.
% ssize - The step size to be used. (Default: ssize = (tfinal - t0)/100).
%
% OUTPUT:
% t      - Returned integration time points (column-vector).
% y      - Returned solution, one solution row-vector per tout-value.

% Initialization
t0 = tspan(1); tfinal = tspan(2);
pm = sign(tfinal - t0); % Which way are we computing?
if (nargin < 4), ssize = abs(tfinal - t0)/100; end
if ssize < 0, ssize = -ssize; end
h = pm*ssize;
t = t0; y = y0(:);
tout = t; yout = y.';

% We need to compute the number of steps.
dt = abs(tfinal - t0);
N = floor(dt/ssize) + 1;
if (N-1)*ssize < dt, N = N + 1; end

% Initialize the output.
tout = zeros(N,1);
tout(1) = t;
yout = zeros(N,size(y,1));
yout(1,:) = y.';
k = 1;

% The main loop
while k < N
    if pm*(t + h - tfinal) > 0
        h = tfinal - t;
        tout(k+1) = tfinal;
    else
        tout(k+1) = t0 + k*h;
    end
    k = k+1;
    % Compute the slope
    s1 = feval(FunFcn, t, y); s1 = s1(:); % s1 = f(t(k),y(k))
    y = y + h*s1; % y(k+1) = y(k) + h*f(t(k),y(k))
    t = tout(k); yout(k,:) = y.';
end;

```

Since we are not teaching programming, we will not explain everything in this file, but a few things should be explained. The % is MATLAB's symbol for comments. MATLAB ignores everything that appears on a line after a %. The large section of comments that appears at the beginning of the file can be read using the MATLAB help command. Enter help eu1 and see what happens. Notice that these comments at the beginning take up more than half of the file. That's an indication of how easy it is to program these algorithms. In addition, a couple of lines of the code are needed only to allow the program to handle systems of equations.

Next, we present the M-file for the second order Runge-Kutta method, without the help file, which is pretty much identical to the help file in eu1.m. You can type help rk2 to view the help file in MATLAB's command window.

```
function [tout, yout] = rk2(FunFcn, tspan, y0, ssize)

% Initialization
t0 = tspan(1);
tfinal = tspan(2);
pm = sign(tfinal - t0); % Which way are we computing?
if nargin < 4, ssize = (tfinal - t0)/100; end
if ssize < 0, ssize = -ssize; end
h = pm*ssize;
t = t0;
y = y0(:);

% We need to compute the number of steps.
dt = abs(tfinal - t0);
N = floor(dt/ssize) + 1;
if (N-1)*ssize < dt
    N = N + 1;
end

% Initialize the output.
tout = zeros(N,1);
tout(1) = t;
yout = zeros(N,size(y,1));
yout(1,:) = y.';
k = 1;

% The main loop
while k < N
    if pm*(t + h - tfinal) > 0
        h = tfinal - t;
        tout(k+1) = tfinal;
    else
        tout(k+1) = t0 +k*h;
    end
    k = k + 1;
    s1 = feval(FunFcn, t, y); s1 = s1(:);
    s2 = feval(FunFcn, t + h, y + h*s1); s2 = s2(:);
    y = y + h*(s1 + s2)/2; t = tout(k);
    yout(k,:) = y.';
end;
```

Finally, we have the M-file for the fourth order Runge-Kutta method, without the help file. Type `help rk4` to view the help file in MATLAB's command window.

```
function [tout, yout] = rk4(FunFcn, tspan, y0, ssize)

% Initialization

t0 = tspan(1);
tfinal = tspan(2);
pm = sign(tfinal - t0); % Which way are we computing?
if nargin < 4, ssize = (tfinal - t0)/100; end
if ssize < 0, ssize = -ssize; end
h = pm*ssize;
t = t0;
y = y0(:);

% We need to compute the number of steps.

dt = abs(tfinal - t0);
N = floor(dt/ssize) + 1;
if (N-1)*ssize < dt
    N = N + 1;
end

% Initialize the output.

tout = zeros(N,1);
tout(1) = t;
yout = zeros(N,size(y,1));
yout(1,:) = y.';
k = 1;

% The main loop
while (k < N)
    if pm*(t + h - tfinal) > 0
        h = tfinal - t;
        tout(k+1) = tfinal;
    else
        tout(k+1) = t0 +k*h;
    end
    k = k + 1;
    % Compute the slopes
    s1 = feval(FunFcn, t, y); s1 = s1(:);
    s2 = feval(FunFcn, t + h/2, y + h*s1/2); s2 = s2(:);
    s3 = feval(FunFcn, t + h/2, y + h*s2/2); s3 = s3(:);
    s4 = feval(FunFcn, t + h, y + h*s3); s4 = s4(:);
    y = y + h*(s1 + 2*s2 + 2*s3 +s4)/6;
    t = tout(k);
    yout(k,:) = y.';
end;
```