

Chapter 3

Euler's Method and Functions

The simplest method for approximately solving a differential equation is Euler's method. One starts with a particular initial value problem of the form

$$\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0, \quad (3.0.1)$$

such as

$$\frac{dx}{dt} = x(1 - x), \quad x(0) = 0.1$$

or

$$\frac{dx}{dt} = x(1 - x) + 1 + \sin(t), \quad x(0) = 0.1$$

The equation and the initial condition combine to tell us the derivative of the solution $x(t)$ at time t_0 ,

$$x'(t_0) = f(t_0, x_0).$$

Knowing the derivative, we can expect that the solution will look linear, with the given slope, if t is near t_0 . Pick a value t_1 , and let

$$h = t_1 - t_0.$$

The (Euler) approximate solution at t_1 is

$$x(t_1) \simeq x_1 = x(t_0) + hx'(t_0) = x(t_0) + hf(t_0, x_0).$$

This value x_1 can be used with the equation to obtain an approximate value for the derivative at t_1 ,

$$x'(t_1) \simeq x'_1 = f(t_1, x_1).$$

The method continues in this fashion, with sample times

$$t_n = t_0 + nh, \quad t_{n+1} - t_n = h,$$

and approximate solution values defined iteratively,

$$x(t_{n+1}) \simeq x(t_n) + hf(t_n, x_n).$$

Program 2 uses the Matlab commands and arrays previously introduced to compute an approximate solution of the initial value problem

$$\frac{dx}{dt} = x(1 - x), \quad x(0) = 0.1$$

Recall that Matlab arrays have initial index 1, so we let

$$T(i) = t_{i-1}, \quad X(i) = x_{i-1}, \quad i = 1, \dots, N.$$

3.1 Program 2

```
% This program is designed to exercise some of the basic
% capabilities of Matlab for studying differential equations.
% The initial problem is solving the differential equation
%  $x' = x(1-x)$ 
% using Euler's method, then plotting the results.

% Euler's method computes values of an approximate solution
% on an interval [a,b]
% In this example the interval will be [0,10], and
% the solution will be computed at 1000 points.

% Initialize parameters and vectors
N = 1000; % N is the vector size for the problem.
T = zeros(N,1); % T will contain samples from the t-axis.
X = zeros(N,1); % X will contain samples of the solution.
% The vector T will be needed for plotting the results.
% It is not part of the basic Euler's method.

%This method uses a step size
% h which needs to be defined.
% We also need an initial value for the solution.
h = .01;
X(1) = 0.1;

% Next, Euler's method calculates the values of the solution X,
% and the corresponding value of T.
for i=2:N
    T(i) = (i-1)*h;
    X(i) = X(i-1) + h*X(i-1)*(1 - X(i-1));
end

% The plot can be generated using the following commands.
% The 'axis' command helps control the display, forcing the
% x-axis to be displayed from 0 to 10, and the y-axis
% to be displayed from 0 to 1.
plot(T,X); axis([0 10 0 1]);
```

Although Euler's method is simple, it is usually inefficient for high accuracy computations. As an example, you could consider the problem of computing the constant $e = \exp(1)$ to 12 digits, which is typically the accuracy of your calculator. Since $\exp(t) = e^t$ satisfies the initial value problem

$$\frac{dx}{dt} = x, \quad x(0) = 1,$$

the value of e could be calculated using Euler's method.

Program 3 carries out this computation. Since scientific software has highly precise algorithms for basic computations like the value of e^x , we can easily compare Euler's method with the 'known' value. This code allows you to compare results graphically.

Notice that the command

```
plot(T,X,T,Y);
```

plots the two arrays X and Y as functions of T . The second set of plotting commands is more sophisticated. The command

```
axis([0 5 0 200]);
```

controls the displayed domain and range. A title and a label for t -axis are added with the instructions

```
title('Comparing exp(t) with Euler computation');
xlabel('t');
```

One of the functions is plotted with dashes using

```
plot(T,Y,'--');
```

A legend connecting the graphs to their display formats is created with

```
legend('Euler computation', 'exp(t)');
```

Finally, the computer has to be warned that it should not throw away the first graph before displaying the second. This is handled with the commands

```
hold on;
.
.
.
hold off;
```

I've also tossed in a command which is useful for interrupting the program and restarting it, for instance after you have examined some result. The command

```
control = input('Hit enter (or return) to continue');
```

causes the computer to type the string 'Hit enter (or return) to continue' to the screen. It then waits until you hit 'return' before continuing to execute the program.

3.2 Program 3

```
% One problem with numerical computations like Euler's method
% is that they produce results which may not be accurate.
% One way to assess accuracy is to compare their computations
% with known results. For instance, we could solve the simple
% differential equation  $y' = y$ , with solution  $y(t) = \exp(t)$ .
% Here's a piece of code that computes this function with
% Euler's method, and graphically compares the result with
% the exact solution.
```

```
% Initialize parameters and vectors
N = 100; % N is the vector size for the problem
T = zeros(N,1); % T holds the t-axis samples
X = zeros(N,1); % X holds the computed solution
Y = zeros(N,1); % Y holds the 'exact' solution.
```

```
% Let's 'solve' the differential equation
%  $x' = x$ ,  $x(0) = 1$ 
% on the interval  $[0,5]$  using Euler's method.
```

```
h = 5/N; % Define step size h
X(1) = 1; % Set initial value for computed solution
Y(1) = 1; % Set initial value for exact solution
T(1) = 0; % Initialize T
```

```
for i=2:N
    T(i) = (i-1)*h;
    Y(i) = exp(T(i));
    X(i) = X(i-1) + h*X(i-1);
end
```

```
plot(T,X,T,Y);
```

```
% Obviously, the functions are not the same over the
% range  $[0,5]$ .
% To make the plot more useful, here is an alternative set of
% commands that adds captions as it plots one graph at a time.
```

```
% The 'hold on' command is needed to overlay plots.

% Wait for a key to be pressed on the keyboard
control = input('Hit enter (or return) to continue');

plot(T,X);
axis([0 5 0 200]);
title('Comparing exp(t) with Euler computation');
xlabel('t');

hold on;
plot(T,Y,'--');
legend('Euler computation', 'exp(t)');
hold off;
```

Program 4 is a revision of the Euler's method code which includes two important features:

- (i) input of parameters from the keyboard, and
- (ii) use of function subprograms.

Both features are helpful if you want to develop a piece of software that can be used repeatedly for solving similar problems.

In this case we want to solve initial value problems using Euler's method. Euler's method computes values of an approximate solution to

$$x' = f(t, x), \quad x(t_0) = x_0,$$

on an interval $[a, b]$. The algorithm also requires us to specify a number N of algorithmic increments, corresponding to $N + 1$ sample points t_0, \dots, t_N .

Input of the parameters N , a and b from the keyboard is controlled by the following instructions.

```
N = input('Enter the number of sample points, then hit return \n')
a = input('Enter the value of "a" to define the interval [a,b] \n')
b = input('Enter the value of "b" to define the interval [a,b] \n')
```

Each command prints the character string in single quotes to the screen, waits until you enter a value and hit return, then assigns the typed value to the variable on the left of the equal sign. The

`\n`

at the end of the strings tells the computer to advance one line on the display device. This makes it easier to see what values are being entered.

3.3 Program 4 - Main program

```

% This program revises the Euler's method code to include a function.
% In addition, part of the problem description will be input from
% the keyboard.

% Euler's method computes values of an approximate solution to
%  $x' = f(t,x)$ ,  $x(t_0) = x_0$ 
% on an interval  $[a,b]$ 

% In this example the interval and the number of sample points
% will be input from the keyboard.
flag = 0;
while(flag == 0)
    N = input('Enter the number of sample points, then hit return \n')
    a = input('Enter the value of "a" to define the interval [a,b] \n')
    b = input('Enter the value of "b" to define the interval [a,b] \n')
    flag = input('Enter 1 to proceed, or 0 to redefine parameters \n');
end

% Initialize vectors
T = zeros(N,1); % T will contain sample values from the t-axis.
X = zeros(N,1); % X will contain samples of the computed solution.
% The vector T will be needed for plotting the results. It is not
% part of the basic Euler's method.

%This method uses a computed step size h.
h = (b-a)/N

% We also need an initial value for the solution.
X(1) = input('Enter the initial value of the solution x at t = a \n');
T(1) = a;

% Next, Euler's method calculates the values of the solution X, and
% the corresponding value of T.
for i=2:N
    T(i) = a + (i-1)*h;
    X(i) = X(i-1) + h*Myfunction1(X(i-1));
end

```

```
end
```

```
% The plot can be generated using the following commands.  
% The 'axis' command helps control the display, forcing the  
% x-axis to be displayed from 0 to 10, and the y-axis  
% to be displayed from 0 to 1.  
plot(T,X);  
axis([a b 0 1]);
```

Notice that the program contains extra code, which is a bit more complex than is absolutely necessary. Experience teaches that this approach is useful. When you are entering control parameters for a familiar program, mistakes are fairly common. Instead of executing the program with incorrect values, which in some cases could mean a delay of hours or more, or interrupting the computer, a practical idea is to deliberately stop the program execution until you confirm that you want to proceed. That is the purpose of the given segment of instructions.

```
flag = 0;
while(flag == 0)
    N = input('Enter the number of sample points, then hit return \n')
    a = input('Enter the value of "a" to define the interval [a,b] \n')
    b = input('Enter the value of "b" to define the interval [a,b] \n')
    flag = input('Enter 1 to proceed, or 0 to redefine parameters \n');
end
```

The 'while loop' repeatedly executes the 'input' commands until the value of 'flag' changes from 0 to another value. If you enter a 1 at the final prompt, the loop will stop repeating, and the program will continue.

Matlab has based much of its programming structure on the programming language C. Along with C, Matlab has an AWFUL feature illustrated by the 'while' statement

```
while(flag == 0)
```

Notice that there is a double equal sign ==. To execute the 'while' command, the computer evaluates the truth of the statement 'flag equals 0' if you type 'flag == 0'. However, if you type 'flag = 0' with a single equal sign, you get an error message. The situation is much worse in C, where the computer assigns the value 0 to the variable 'flag', then tries to evaluate the truth of that statement. This can lead to major problems.

The second feature in program 4 is the use of a function subprogram, which in this case is called 'Myfunction1'. In this example the function is simply used to provide a concrete example of the general function $f(t, x)$ that we see in the problem or algorithm description.

Function subprograms are an essential part of good software development. Suppose we wanted to expand this program to include a graph of $f(t, x) = x(1-x)$, as well as a comparison of several different algorithms for solving the differential equation, each of which requires the $x(1-x)$, perhaps at different sample points. We could easily end up writing the expression $x(1-x)$ several times.

As long as $f = x(1-x)$, the repetitions don't seem like much of a problem. Suppose now that we have to replace $x(1-x)$ by a complex expression requiring many lines of code. For instance, f may have a piecewise definition with 10 different pieces. Now the problem of repetitions is much more severe. Not only is there more typing, but each time we type a new copy there is a chance of making an error. Suppose the function needs to be changed. Then we have to track down each place that it occurs, and make sure the changes are identical. This situation quickly becomes a maintenance nightmare.

The solution is to use a function subprogram. This is a separate piece of code, kept (in Matlab) in a separate file. Every time the main program needs to evaluate $x(1-x)$ we send a message to the function, which sends back the desired value. If we have to make changes, or keep several copies of different functions, the typing of changes is minimized, and the changes that do occur do not effect the integrity of the rest of the program.

It is hard to overstate the importance of using functions to organize programs. The programs we have seen so far are tiny, with a transparent organization. Out in the 'real world', programs frequently have hundreds of thousands or millions of instructions. It is generally impossible to understand what the software is doing, get it to function correctly, or maintain it as it goes through revisions, unless it is carefully decomposed into easily understood (and well documented) functions.

There are various rules for writing functions. In Matlab, functions should be in a file whose name matches the function name. As you can see from the example following the main program 4, the first line announces that this is a function, and identifies the variable which will be output, y , and names the input x . (These names are arbitrary.) After handling these preliminaries, we simply write instructions to find y as a function of x . In this example, the original function $x(1-x)$ has been modified for values of x when the original function is negative.

3.4 Program 4 - function definition

```
function [y] = Myfunction1(x);  
% This file defines a function for use with  
% Euler's method for solving initial value problems  
y = x*(1-x);  
  
if y < 0  
    y = 0;  
end  
  
if y > 1  
    y = 1;  
end
```

3.5 Exercises

1. Modify program 3 so you can find the error in computing e using Euler's method on the interval $[0, 1]$ using N steps. Approximately how many steps will be required to calculate e by this method with an error smaller than 10^{-12} ? Why isn't this a practical method?

2. One way to assess the accuracy of an algorithm like Euler's method is to increase the number of steps N and see how much the solution changes. The usual assumption is that the error in computation with the smaller number of steps is approximately the difference between the two solutions. That is, the case with larger N is treated as if it were the exact solution.

Rewrite program 2 so that it solves the same problem with $N = 10$, $N = 100$, and $N = 1000$ steps. Evaluate the differences both numerically and graphically. When N is changed, the value of h will also change. Modify the code so that h is automatically computed based on N . The graphical evaluation should show all three plots on the same graph. Estimate the largest error made for each value of N .

3. Our Euler's method code is a bit sloppy in its handling of sample points. Recall that the array value $T(n)$ corresponds to the sample point t_{n-1} . Typically, we would really like sample points t_0, \dots, t_N and sample values x_0, \dots, x_N . The code is actually computing x_0, \dots, x_{N-1} . Repair the code so that this defect is corrected.