Matlab Practical: Solving Differential Equations

Introduction

This practical is about solving differential equations numerically, an important skill. Initially you will code Euler’s method (to get some more practice with simple Matlab programming), but we will soon move on to using the built-in functions for solving differential equations, as these built-in functions are invariably used in practice.

By the end of the practical you should be able to get Matlab to solve coupled non-linear systems of differential equations of the type that often arises in modelling studies. In particular we will investigate the equations governing enzyme kinetics.

The first part of this practical concerns the differential equation

\[
\frac{dy}{dt} = ty \quad \text{subject to} \quad y(0) = 2
\]

Again the practical consists of a series of exercises for you to do: these are boxed. Before each exercise, any necessary Matlab commands and concepts are briefly introduced; anything that should be typed into Matlab is denoted by courier. As you are actually typing the Matlab commands, think about why they behave as they do; if at any time you do not understand then please ask.

Euler’s method

As discussed in the lecture, Euler’s method is the simplest way of solving differential equations numerically. The method relates the current value of the right hand side of a first order differential equation to the new value of the numerical approximation. In particular, in solving the first order differential equation

\[
\frac{dy}{dt} = f(y,t)
\]

subject to the initial condition \( y(0) = y_0 \), Euler’s method creates a sequence of values \( y_n \), where \( y_n \approx y(n\Delta t) \) for \( n > 0 \). The value \( y_n \) is an approximation to the value of the solution \( y \) at \( t = n\Delta t \), where \( \Delta t \) is the step length. A single step of Euler’s method follows the rule

\[
y_{n+1} = y_n + (\Delta t) f(y_n, n\Delta t)
\]

Exercise One

Use Matlab to solve Equation (1) using Euler’s method with time steps \( \Delta t = 0.01 \), up to maximum time \( T_{\text{max}} = 3 \). Plot your approximate solution.

(Hint: one way of doing this would be to work out in advance how many steps are required, and then to use a for loop to automate the calculation of \( y_{n+1} \) from \( y_n \) and \( n\Delta t \), storing the values of \( y \) and \( t \) in vectors [for later plotting]).
As you may have realised, it is in fact relatively easy to solve Equation (1) using a pen and paper.

**Exercise Two**
Solve Equation (1) analytically
(Hint: this differential equation can be solved using separation of variables)

**Exercise Three**
Create a new function `exactSoln(t)`, which finds the value of the exact solution you found in Exercise Two. Make sure it works with both scalar and vector inputs `t`.

Now we have an explicit form for the solution, a useful thing to do would be to plot it and compare with the numerical solution via Euler’s method. To date you have plotted all graphs on separate axes. However a natural goal is often to plot two curves on the same axes. Consider the following (again type it in line by line so you can see what is happening).

```matlab
x = [0:0.01:6*pi];
y = sin(x);
z = cos(x);
plot(x,y);
plot(x,z,'r');
hold on;
plot(x,y);
xlabel('x')
ylabel('y')
```

By issuing the commands

```
help hold
```

investigate what is going on.

**Exercise Four**
Plot the exact solution and your numerical approximation according to Euler’s method on the same axes. By investigating the results of `help print`, work out how to save the figure to a file in jpeg format.

---

**Using built in Matlab functions for solving differential equations**

Euler’s method is impractical for solving differential equations, since very small steps are required to achieve satisfactory accuracy. Instead Matlab provides a number of built-in functions for solving differential equations; the workhorse is `ode45` (technically this is “a fourth order Runge-Kutta method with fifth order error control”; the basic idea is similar to Euler’s method, but a more sophisticated approximation is
used to find the slope of the tangent, and by default the method chooses the sizes of its individual steps to ensure satisfactory accuracy).

According to the Matlab help, ode45 is what you should “try first”, and it is all that will be required in this practical. The easiest way of using the built-in differential equation solvers relies on the right hand side of the differential equation being specified in an appropriate function.

**Exercise Five**
Create a function \( \text{myRHS}(t, y) \) which returns the right hand side of Equation (1) (i.e. given a value of \( t \) and \( y \), returns their product). It does not need to be vectorised; i.e. it is sufficient to write a function which expects only scalar input arguments, and which returns a single value.

The basic syntax for the built-in solvers follows

\[
\text{ode45}(<\text{function}>,<\text{domain}>,<\text{initCond}>)
\]

where \(<\text{function}>, <\text{domain}> \) and \(<\text{initCond}> \) are specified by the user. In particular, try the following

\[
\begin{align*}
t\text{Range} &= [0,1]; & \text{% set up range for } t \\
y\text{Zero} &= 1; & \text{% and an initial condition} \\
[\text{myT,myY}]=\text{ode45}(@\text{myRHS},t\text{Range},y\text{Zero}); & \text{% solve} \\
\text{plot(myT,myY,'c');} & \text{% plot}
\end{align*}
\]

**Exercise Six**
By altering the above code, produce a plot of the solution of Equation (1) using ode45 over the range \( t=0 \ldots 3 \) and with \( y(0) = 2 \). By plotting the exact solution you created in Exercise 3 on the same axes, test the quality of the numerical approximation. (Hint: you may wish to request that Matlab plots one of the graphs using a symbol such as ‘o’, as otherwise the graphs may be so close that one covers the other).

Examine the contents of the vector \( \text{myT} \) and check the number of elements it contains using \( \text{length}() \). Note that in approximating the solution, Matlab has selected a particular partition of the range specified, and returned the values of \( y \) and \( t \) at those points. Often is it desired that the user specifies these values (e.g. when they are to be used in some further calculation). Check you understand the following (note the definition of \( t\text{Range} \) is different)

\[
\begin{align*}
t\text{Range} &= [0:0.01:1]; & \text{% set up range for } t \\
y\text{Zero} &= 1; & \text{% and an initial condition} \\
[\text{myT,myY}]=\text{ode45}(@\text{myRHS},t\text{Range},y\text{Zero}); & \text{% solve} \\
\text{plot(myT,myY,'c');} & \text{% plot}
\end{align*}
\]
It is important to point out here that Matlab continues to use the same partition of values internally; the only thing that changes is the values of \( t \) for which it returns the solution.

**Solving coupled differential equations**

The solvers in Matlab work as you would expect for coupled differential equations; the important thing to remember here is that the function corresponding to the right hand side system must now return a (column) vector. Consider the following system (which may be familiar to you as the simplest version of the Lotka-Volterra equations for a predator \( P \) and its prey \( V \))

\[
\frac{dV}{dt} = 0.1V - 0.2VP \\
\frac{dP}{dt} = 0.4PV - 0.2P
\]

The following function implements the right hand side of these equations, taking some care to ensure that the return value is a column vector:

```matlab
function ret=lvRHS(t,y)  
    % set up ret in correct form (i.e. a column vector) 
    ret = zeros(2,1); 
    % extract current values of V and P 
    V = y(1); 
    P = y(2); 
    % return updates 
    ret(1) = 0.1 * V - 0.2 * V * P; 
    ret(2) = 0.4 * P * V - 0.2 * P; 
end 
```

Type this into a file called `lvRHS.m` and save it. Then enter the following code at the command line:

```matlab
% set up range for t 
% and an initial condition 
% solve 
[tRange,yZero]=ode45(@lvRHS,tRange,yZero); 
plot(tRange,yZero(:,1),'c'); 
hold on 
plot(tRange,yZero(:,2)); 
legend('prey','predator'); 
xlabel('t'); 
ylabel('population densities'); 
```

The time evolution of predator and prey is displayed, showing the characteristic cycles.
Chemical kinetics

As you probably know, enzymes are proteins that catalyse a biochemical reaction by reducing the activation energy required for the reaction to proceed. They are often specific to a particular substrate, and catalyse its conversion to a product, themselves remaining unchanged by the reaction. In the simplest and archetypal situation, Michaelis-Menten kinetics, they accomplish this in two steps, first forming a complex with the substrate, which then breaks down to form the product and recover the enzyme

\[ S + E \xrightarrow{k_1} C \xrightarrow{k_2} P + E \]

where we are assuming the back-reaction (i.e. from \( P + E \rightarrow C \)) is so slow as to be negligible. The constants \( k_1, k_1 \) and \( k_2 \) are the rate constants governing the speeds of the individual reaction steps (note that the first reaction is assumed to be reversible). The equations governing the kinetics follow

\[
\begin{align*}
\frac{dS}{dt} &= k_{-1}C - k_1SE \\
\frac{dE}{dt} &= (k_{-1} + k_2)C - k_1SE \\
\frac{dC}{dt} &= k_1SE - (k_{-1} + k_2)C \\
\frac{dP}{dt} &= k_2C
\end{align*}
\]  

(2)

Exercise Seven

Confirm that the total amount of enzyme is conserved (i.e. \( E + C = E_0 \), where \( E_0 \) is the amount of enzyme present initially). Also confirm that, since substrate can only occur in its original form, bound to the enzyme or converted to the product, that \( S + C + P = S_0 \), where \( S_0 \) is the amount of substrate present initially.

The manipulations performed above mean that the four dimensional system can in fact be rewritten as the pair of coupled equations

\[
\begin{align*}
\frac{dS}{dt} &= k_{-1}C - k_1S(E_0 - C) \\
\frac{dC}{dt} &= k_1S(E_0 - C) - (k_{-1} + k_2)C
\end{align*}
\]  

(3)

The values of \( E \) and \( P \) can be determined via

\[
\begin{align*}
E &= E_0 - C \\
P &= S_0 - S - C
\end{align*}
\]  

(4)

Exercise Eight

Write Matlab code to solve the system in Equation (3), and plot the amount of substrate and complex against time for \( k_1 = 5 \text{ M}^{-1}\text{s}^{-1}, k_{-1} = 0.1 \text{ s}^{-1}, k_2 = 1 \text{ s}^{-1}, \) with \( S_0 = 5 \text{ M}, E_0 = 0.1 \text{ M}, \) until \( T_m = 20 \) (seconds).
Exercises for more advanced programmers

Solving higher order equations (optional)

The numerical routines we have discussed so far are restricted to first order systems (i.e. where the highest derivative in any equation is the first derivative, e.g. dY/dt). However, on occasion higher order equations occur; simple second order equations were covered in the lecture. In order to solve these, some trickery is required.

Consider the second order differential Equation

\[
\frac{d^2 Y}{dt^2} + 5 \frac{dY}{dt} + 6Y = e^t, \quad \text{subject to } Y(0) = \frac{25}{12}, \quad \frac{dY}{dt}(0) = -\frac{59}{60} \tag{5}
\]

Exercise Nine

Find the exact solution of Equation (5)

Exercise Ten

By introducing the derived variable \(Z = \frac{dY}{dt}\), show how the second order Equation (5) can be transformed to a pair of coupled first order equations for \(dY/dt\) and \(dZ/dt\).

Exercise Eleven

Solve the equations using Matlab and compare the solution for \(Y\) with that according to the exact solution.

Options to ode45 (optional)

A number of options can be passed to ode45. Two that can sometimes be important are the relative and absolute tolerances, RelTol and AbsTol, respectively. Internally, at each step of the algorithm the routine generates an approximation to the error during that step. If \(y_k\) is the approximation at step \(k\), and \(e_k\) is the approximate error at this step, then MATLAB chooses its partition to ensure

\[e_k \leq \max(\text{RelTol} \times y_k, \text{AbsTol})\]

where the default values are \(\text{RelTol} = 0.001\) and \(\text{AbsTol} = 0.000001\). The following code changes AbsTol, and shows how options can be passed to ode45 (in case you need to control these errors in the future, or indeed want to ever pass other options to these commands).

```matlab
options=odeset('RelTol',1e-10);
tRange = [0:1]; % set up range for t
yZero = 1; % and an initial condition
[myT,myY]=ode45(@myRHS,tRange,yZero,options); % solve
```

Nik Cunniffe