7 Using the Matlab ODE solvers

7.1 Overview

Having looked at relatively simple schemes for solving ODE's, we will now look at the advanced solvers which are available in Matlab.

Additional reading for this lecture:
When writing Matlab code I would encourage you to use the helpfile included in Matlab. All the material we will cover in this lecture can be found by searching for 'ode'

Chapter 5 of "*Numerical Methods for Chemical Engineers with Matlab Applications"* by Constantinides and Mostoufi

7.2 The Matlab ode solvers

The Matlab ODE solvers are written to solve problems of the form

$$\frac{dx_1}{dt} = f_1(x_1, x_2, x_3.., t)$$
$$\frac{dx_2}{dt} = f_2(x_1, x_2, x_3.., t) \quad \text{i.e.} \quad \frac{d\underline{x}}{dt} = F(t, \underline{x}).$$
$$\frac{dx_3}{dt} = f_3(x_1, x_2, x_3..., t)$$
$$....$$

The Matlab ODE solvers are accessed by calling a function of the form

```
[X,T] = ode**(@F,TimeSpan,Xo,Options,P1,P2,P3)
```

Optional

`@F`	a handle to a function which returns a vector of rates of change
`Timespan`	a row vector of times at which the solution is needed OR a vector of the form [start, end]
`Xo`	A vector of initial values
`Options` (if omitted or set to [], the default settings are used)	A data structure which allows the user to set various options associated with the ode solver
`P1,P2,P3...`	These are additional arguments which will be passed to `@F`

F must have the following form

```
function [dx_dt] = F(t,x,P1,P2,P3...)

dx_dt = ......

return
```

WARNING - THIS IS A SLIGHTLY DIFFERENT SYNTAX TO THAT USED FOR THE EULER SOLVERS WE LOOKED AT PREVIOUSLY

There are several different ode solvers supplied with Matlab.

Solver	**Implicit/Explicit**	**Accuracy**
ode45	Explicit	4th order, medium accuracy
ode23	Explicit	2nd/3rd order, low accuracy
ode113	Explicit	Very accurate (13th Order)
ode15s	Implicit	Anything from 1st-5th order
ode23s	Implicit	Low accuracy (but may be more stable than ODE15s)
ode23tb	Implicit	Low accuracy (but may be more stable than ODE15s)

We will generally use ODE45 and ODE15s (the 's' signifies that it uses an implicit method).

- The Matlab ode solvers will generally be better than anything you would program yourself
- They are able to estimate the error in the solution at each time step, and decide whether to not the time step is too large (error to high) or too small (inefficient).
- It is more important that you understand concepts such as implicit and explicit schemes and the order of accuracy of the scheme, rather than the actual detail of what each routine does internally.

ODE45 (an explicit Runge-Kutta method) is efficient, but can become unstable with stiff systems. This will manifest itself by the solver taking shorter and shorter time steps to compensate. The solution will either take a long time, or the time step will be reduced to the point where machine precision causes the routine to fail.

ODE15s should only be used for stiff problems. Because it is an implicit scheme, it will have to solve (possibly large) sets of equations at each time step.

7.3 Examples of using the Matlab ODE solvers

7.3.1 Example 1, using ODE 45 to solve the coupled ODEs from Handout 6

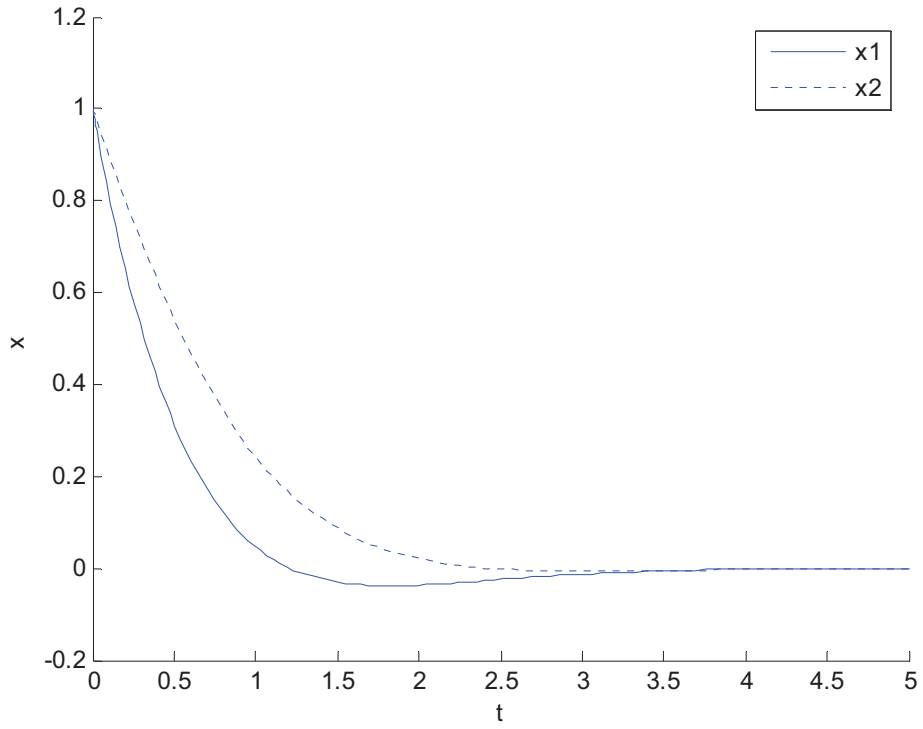
In Handout 6, we solved the following problem using our own ODE solvers:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ 1 & -2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \text{(6-24)} \quad \text{with initial conditions} \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Below is a complete Matlab program which solves this problem using ODE45

```
function main
% a simple example to solve ODE's
% Uses ODE45 to solve
%    dx_dt(1) = -1*x(1)-1*x(2)
%    dx_dt(2) = 1*x(1) -2*x(2)
%set an error
options=odeset('RelTol',1e-6);

%initial conditions
Xo = [1;1];

%timespan
tspan = [0,5];

%call the solver
[t,X] = ode45(@TestFunction,tspan,Xo,options);

%plot the results
figure
hold on
plot(t,X(:,1));plot(t,X(:,2),':')
legend('x1','x2');ylabel('x');xlabel('t')
return
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [dx_dt]= TestFunction(t,x)
%a function which returns a rate of change vector
M = [-1,-1;...
      1,-2]

dx_dt = M*x;
return
```

The result of this code is the same as that produced by the Euler solver.



Solution to Eq. (6-24) produced by ODE45

Some points to note

- *We did not have to specify a step size.* ODE45 uses the (explicit) fourth order Runge-Kutta-Fehlberg[1] method, which also gives an estimate of the truncation error at each step. THE SOLVER IS ABLE TO CHOOSE A STEP SIZE WHICH MEETS THE ERROR TOLERANCE WE SPECIFY.
- *All the Matlab solvers will vary the step size* to produce a solution which is accurate to a given error. You can override this by setting a maximum step size (which can force the solver to take too small a step). It is generally best to let the solver choose the step size, and use the refine option. Refine smoothes the solution by interpolating between points.
- Options are set by creating an options structure with the `odeset` command

[1] See Table 5-2 in "*Numerical Methods for Chemical Engineers with Matlab Applications"* by Constantinides and Mostoufi

7.3.2 Example 2, a stiff problem

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ 1 & -5000 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{7-1}$$

with initial conditions

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

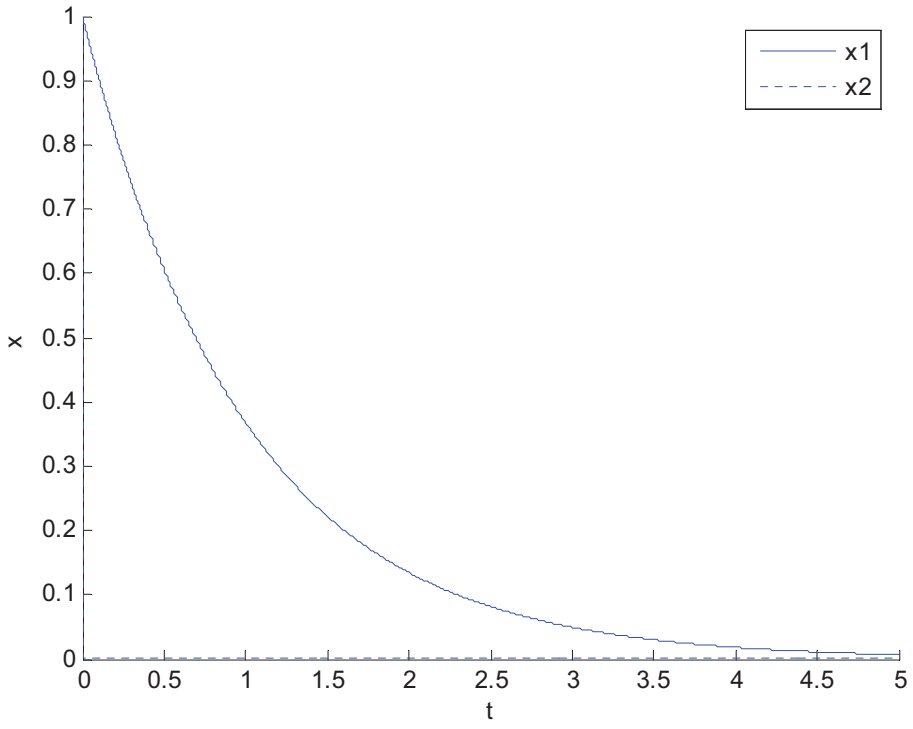
The eigenvalues of M are

λ_1 =-5000

λ_2 =-1.002

So the problem is stiff!

The numerical solution, computed using ODE45 is given below



Solution to Eq. (7-1) produced by ODE45

- A stiff problem will consist of several processes at least one of which will have very small time constant

The code to solve this problem using ODE45 is given below

```
function main
% a simple example to solve ODE's
% Uses ODE45 to solve

%    dx_dt(1) = -1*x(1)-1*x(2)
%    dx_dt(2) = 1*x(1) -5000*x(2)


%set an error
options=odeset('RelTol',1e-6,'Stats','on');

%initial conditions
Xo = [1;1];

%timespan
tspan = [0,5];

%call the solver
tic
[t,X] = ode45(@TestFunction,tspan,Xo,options);
toc

%plot the results
figure;hold on
plot(t,X(:,1));plot(t,X(:,2),':')
legend('x1','x2'); ylabel('x');xlabel('t')

return

function [dx_dt]= TestFunction(t,x)
%a function which returns a rate of change vector

M = [-1,-1;...
      1,-5000];

dx_dt = M*x;
return
```

- The option 'stats' has been turned on. The solver will output to screen information about how it has performed.
- When you run this code it takes a relatively long time to run. The commands 'tic' and 'toc' surrounding the ode45 call measures and displays the time it takes to execute the ode45 call.

```
The stats reported for the solution are

7557 successful steps
504 failed attempts
48367 function evaluations
Elapsed time is 2.672000 seconds.
```

- The solver has been forced to take a very small time step to produce a stable solution.

If we instead use ODE15s, so change the line where the ODE solver is called to

```
[t,X] = ode15s(@TestFunction,tspan,Xo,options);
```

- The code now runs faster

```
The stats reported for the solution are
139 successful steps
3 failed attempts
288 function evaluations
1 partial derivatives
27 LU decompositions
284 solutions of linear systems
Elapsed time is 0.172000 seconds.
```

- The solver can take much bigger time steps (139 compared with 7557 for ode45), which reduces the overall computational time.
- ODE15s is implicit so it is solving equations at each time step.

7.4 Improvements to the code

In the previous example I used ODE15s, which is an implicit routine.

$$\frac{dx_1}{dt} = f(x_1, x_2, x_3.., t)$$
$$\frac{dx_2}{dt} = f(x_1, x_2, x_3.., t) \quad \text{i.e.} \quad \frac{d\underline{x}}{dt} = F(\underline{x}, t).$$
$$\frac{dx_3}{dt} = f(x_1, x_2, x_3..., t)$$
$$....$$

At each time step, ODE15s will solve a set of linear or non-linear equations, for which it will require the Jacobian of $F(t,\underline{x})$. Since the routine was given no information on the Jacobian, it is forced to calculate the full Jacobian, numerically. So, we have two possible options

1. *Supply a routine which returns the Jacobian, or if the Jacobian is a constant, then we can supply the matrix*. To do this, we set the 'Jacobian' option in the odeset structure to the matrix or function name. Generally, this is the most computationally efficient option. Matlab also allows the routine to return a sparse Jacobian (see below).
In this example the Jacobian is a constant, and the following lines are used to supply it to the solver

```
J= [-1,-1;...
      1,-5000];
options=odeset('RelTol',1e-6,'Stats','on','Jac',J);
```

- The code now runs even faster, as the Jacobian no longer needs to be evaluated (even though the gain is very small as the Jacobian is constant in this example, which is detected by the solver – and it would thus not update it after the first evaluation anyway)

```
The stats reported for the solution are
139 successful steps
3 failed attempts
286 function evaluations
0 partial derivatives
27 LU decompositions
284 solutions of linear systems
Elapsed time is 0.109000 seconds.
```

2. *Supply a Jacobian pattern.* By supplying a Jacobian pattern, the routine is able to avoid costly calls to the rate of change function. A Jacobian pattern is simply a sparse matrix of zero's and ones. The ones only appear where the Jacobian is non-zero.

e.g. if $\begin{aligned} \frac{dx_1}{dt} &= x_1 - x_2 \\ \frac{dx_2}{dt} &= -x_2 \\ \frac{dx_3}{dt} &= 10x_1 - x_3^{\,2} \end{aligned}$ then $Jpattern = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$

7.4.1 Example 3, using ODE 45 to solve a non-linear ODE

Consider the above set of ODE's

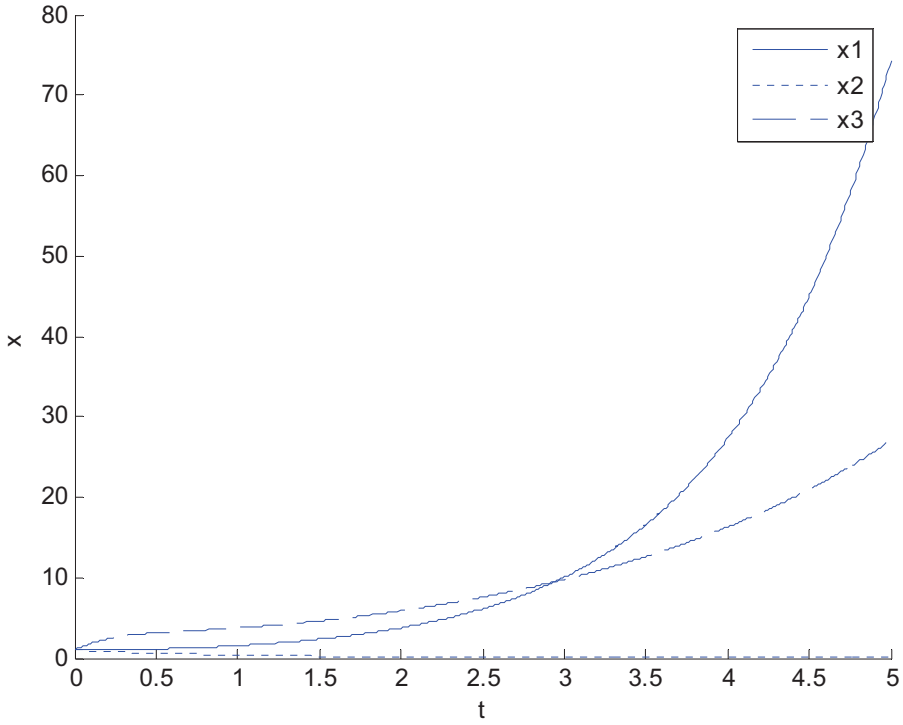
$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 - x_2 \\ -x_2 \\ 10x_1 - x_3{}^2 \end{bmatrix} \tag{7-2}$$

with initial conditions

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Solving non-linear ODEs is straight forward. We just have to write a function which returns the rate of change of the vector $\underline{x}$

```
function [dx_dt]= TestFunction(t,x)
%a function which returns a rate of change vector


dx_dt(1) = x(1) - x(2);
dx_dt(2) = -x(2);
dx_dt(3) = 10*x(1) - x(3)^2;

%transpose dx_dt so it is a column vector
dx_dt = dx_dt';


return
```



Solution to Eq. (7-2) produced by ODE45

- For non-linear equations, it is the eigenvalues of the Jacobian which determine stability. These will vary as the solution proceeds.

7.5 Summary

The Matlab solvers are a very useful tool, but to use them effectively you have to understand a bit about how they work.