MATLAB Session 3 Web Support Supplement

Time Response Simulation Functions	
`ode23, ode45`	Runge-Kutta integration functions

M3.3 Runge-Kutta algorithm

Features covered in this session:

- Use of `ode23()` and `ode45()`
- Plotting the results

We've been using `step()`, `lsim()`, etc. to simulate differential equation models, which in essence, is a set of first order differential equations. These functions actually make use of state space and digital control algorithms. MATLAB can also integrate a set of ordinary differential equations using more common numerical techniques such as the Runge-Kutta algorithm.

The "look" of the M-file will be different from C or FORTRAN programs because MATLAB is so matrix-oriented. Warning: interpreted MATLAB runs much slower than a compiled FORTRAN or C code. MATLAB is designed to be an environment for "experimentation," and not number crunching. Nevertheless, computers have gotten so fast and your homework so simple that you'll hardly notice the difference.

For general numerical integration of differential equations, we can use the two functions `ode23()` and `ode45()`. In a nutshell, `ode23` uses a 2nd and 3rd order pair of formulas, while `ode45` employs 4th and 5th order Runge-Kutta-Fehlberg formulas to do automatic step size integration. MATLAB supports other stiff differential equation solvers, but it is not likely that you will need them in homework problems. For more information, enter

```
help matlab/funfun
```

or enter "`help ode23`" or "`help ode45`." What follows is an example taken from the MATLAB `demo`.

In this example, we take from the MATLAB demo two coupled ordinary differential equations which model predator-prey dynamics:

$$\frac{dy_1}{dt} = (1 - \alpha\, y_2)\, y_1 \quad \text{and} \quad \frac{dy_2}{dt} = (-1 + ß\, y_1)\, y_2$$

where y_1 is the prey, y_2 is the predator, and the equilibrium point is (1,1).

For simplicity, we will use the default options for tolerance, which generally work quite well.[1] Both `ode23()` and `ode45()` have identical parameter lists. To use `ode45()`, all we need is to replace `ode23` with `ode45` in the M-file below. Finally, the example illustrates how we may pass model parameters to the function where we have our differential equations. (We could be sloppy and define the parameters as global in both the driver program and the function with the differential equations.)

[1] To learn how to override the default, use "`help ode23`" and "`help odeset`."

Store the following statements in an M-file, say, named "`odesolver.m`" The two differential equations are stored in a separate M-file named `lotka.m`. You can skip typing all the comments. Without them, the M-file is actually very short (5 statements without the plotting). When we are inside MATLAB, enter "`odesolver`" to run it.

```
% First: define the parameters to be passed to the
% derivative function stored in lotka.m.
%
% Let's pick some arbitrary numbers:

alpha=0.4;
beta=0.5;

% Now set the beginning and end times and initial conditions
% for the odes to be integrated.
% MATLAB figures out how many odes we have by using the
% length of the initial condition value vector y0[].

tspan = [0 30];          % Integrate from t=0 to t=30 [time unit]
y0=[5 3];                % Initial condition vector at t=0

% Finally let's do the integration.
% The 'lotka' means all the odes will be stored in the
% file lotka.m
% The [] is a necessary place holder for using default options.
% Variable names after the [] are parameters for the model.

[t,y] = ode23('lotka',tspan,y0, [],alpha,beta);

% Now we can plot the results

plot(t,y)
title('Lotka-Volterra  model')
xlabel('Time'),ylabel('Populations')
figure(2)
plot(y(:,1),y(:,2))
title('Phase plane plot')
xlabel('Prey'),ylabel('Predator')

figure(1)     %reset to fig 1 [2]

% end of "program"
```

Next, enter the following statements in a file named "`lotka.m`" and it will be "called" by the "program"`odesolver.m`. The function `lotka.m` contains the two Lotka-Volterra predator-prey differential equations written above.

The function argument list uses an empty dummy flag, which is all we need under most circumstances. For more details, enter "`help odefile`."

[2] If you do not mind the plots being small, you can use `subplot(221)` and `subplot(222)` to put the two plots in the same window. When you are done, enter `clf` (for clear figure) to set the graphics screen back to normal.

```
% ____________________  lotka.m  ____________________
%
% This file returns the derivatives dydt[] (dy/dt)
% to ode23() or ode45()
%
% The flag here is a dummy and lotka() requires the
% parameters alpha and beta.
%
% ___ Make sure all the statements end with a ; ___
%
function dydt = lotka(t,y, flag,alpha,beta)

yp1 = (1 - alpha*y(2))*y(1);
yp2 = (-1 + beta*y(1))*y(2);

dydt = [yp1; yp2];      % dydt is a column vector

%end of lotka.m
```

This last example also allows us to examine the difference between a simple M-file (`odesolver.m` here) and a function (`lotka.m`). We may consider "`odesolver.m`" as just a "loose" collection of statements, except of course they are organized to do something useful by calling the `ode23()` function and plotting the results. There are no real rules on how this M-file must be structured.

We cannot say the same with the function `lotka()` defined inside `lotka.m`. But there is really only one rule: a function must begin with the function statement defining the input arguments (**`t`**, **`y`**, etc.) on the right of the function name, and the output variable (**`dydt`** here) to its left. Other than that, a function looks pretty much like an M-file.

Unlike a formal compiled language like C, we do not need to define the data types. We do not even need an "end" statement. (Note that the one in our examples is really a comment.) The "return" statement that we are familiar with in C is replaced by the output variable in the function statement.