10.34. Numerical Methods Applied to Chemical Engineering

Optional MATLAB programming warm-up exercises

NOTE : consult the on-line MATLAB tutorial for help in completing these exercises

1. Make a plot of the function

$$f(x) = \exp\left(-\frac{(x-2\pi)^2}{4}\right)\sin(6x) \qquad \text{(EQ 1)}$$

over the range $[-\pi, 5\pi]$. Add labels to the *x* and *y* axes and a title.

2. Write a MATLAB function, in an m-file, that takes as input a single real number and checks to see if it is a power of two (see *log2* command). The routine should return an integer that is 1 if the input number is a power of 2 and 0 if it is not.

To make things more interesting, you might add two optional output variables. The first returns the closest power of 2 below the input value and the second returns the closest power of 2 above.

As a further fluorish, modify the routine to include an optional second input argument, B. The routine then checks to see whether the first input argument is a power of B (HINT - use the properties of the natural logarithm to calculate the base-B logarithm).

3. van der Pol oscillator

MATLAB is well-suited to the solution of common technical problems because it contains many built-in routines to perform common tasks. For example, one can simulate the behavior of a system governed by a set of ordinary differential equations by marching forward in small time increments, and using the present value of the state variables and their derivatives to predict their values at the next time step. The built-in MATLAB function *ode45* uses a Runge-Kutta method and requires only that the user supply the initial values of the state variables, the time span of the simulation (e.g. start and end time), the values of any fixed parameters, and a routine that, given the current time and values of the state variables and fixed parameters, returns the vector of time derivatives.

The MATLAB program below simulates the behavior of a van der Pol oscillator, governed by the differential equation

$$\frac{d^2u}{dt^2} - \lambda(1-u^2)\frac{du}{dt} + u = 0 \qquad \text{(EQ 2)}$$

This second order ODE is converted to a set of two first order ODE's by defining the following variables

$$y_1 = u \qquad y_2 = \frac{du}{dt} \qquad \text{(EQ 3)}$$

to yield

$$\frac{dy_1}{dt} = f_1(y_1, y_2) = y_2 \qquad \text{(EQ 4)}$$

$$\frac{dy_2}{dt} = f_2(y_1, y_2) = \lambda(1-y_1^2)y_2 - y_1 \qquad \text{(EQ 5)}$$

The MATLAB program below uses *ode45* to simulate the behavior of this system given the value of λ, the values of y_1 and y_2 at the initial time $t = 0$, and the time value at which the simulation ends, t_{end}. The actual MATLAB commands are in bold print. Try running the simulation by saving this text as the m-file simulate_van_der_Pol.m (you need not add all of the comments), and typing at the MATLAB prompt *simulate_van_der_Pol(1,[1; 0], 25);*

Now try writing your own program to perform the same simulation without using the built-in MATLAB function *ode45*. Given the values of y_1 and y_2 at the current time t, pick a small value of the time step Δt and estimate the values of the state variables at the next time step using the explicit Euler rule,

$$y_1(t+\Delta t) = y_1(t) + (\Delta t)f_1(y_1(t), y_2(t)) \qquad \text{(EQ 6)}$$

$$y_2(t+\Delta t) = y_2(t) + (\Delta t)f_2(y_1(t), y_2(t)) \qquad \text{(EQ 7)}$$

Compare the results of this simulation to those obtained from the more accurate Runge-Kutta method. To compare the simulation results more carefully, at each time step of the explicit Euler simulation, calculate the difference between the explicit Euler result and the result using the Runge-Kutta method. Since the Runge-Kutta results are reported at different times than those used in the explicit Euler calculation, you will beed to use the MATLAB 1-D interpolation function *interp1*. Try making a plot of the maximum absolute value of the difference between the two simulations as a function of time step.

```
% simulate_van_der_Pol.m
%
```

```
% This MATLAB program simulates the van der Pol
% oscillator using the MATLAB function ode45().
%
% van der Pol oscillator equation :
%   u'' - \lambda*(1-u^2)*u' + u = 0
%
% input arguments
%   lambda = value of lambda parameter
%   y0 = 2-D column vector with initial guess of state variables
%   t_end = the end time of the simulation
%
% K. Beers. MIT ChE. 8/28/2002

function iflag_main = simulate_van_der_Pol(lambda,y0,t_end);

% This integer is set to 0 to show no successful completion.
iflag_main = 0;

% Call the built-in MATLAB routine ode45() that uses a Runge-Kutta
% algorithm to integrate the equations of motion of the oscillator from
% t = 0 to t = t_end
[t_val,y_val] = ode45(@calc_f_van_der_Pol,[0 t_end],y0,[],lambda);

% Plot the results
figure;  % declare a new figure
plot(t_val,y_val(:,1));
xlabel('t');
ylabel('f(t)');
title('Van der Pol oscillator');

iflag_main = 1;

return;


% ----- calc_f_van_der_Pol()
function f = calc_f_van_der_Pol(t,y,lambda);

f = zeros(size(y));

f(1) = y(2);
f(2) = lambda*(1-y(1)^2)*y(2) - y(1);

return;
```