Drexel University
Office of the Dean of the College of Engineering
ENGR 232 – Dynamic Engineering Systems

Week 2 – Pre Lab

Numerical Methods

For first-order system models, if the equation is separable we can find a solution through separation of variables, providing that the integrals in the separated form exist in closed form. For linear first-order system models, we also have the added benefit of being able to solve using the integrating factor method.

Not all equations will have a closed-form solution, and at times, the integrals required to solve systems may be difficult to compute. Furthermore, it is not always easy (or possible) to express the solution in the **explicit** form: $y = f(t)$. Sometimes, an **implicit** form $F(t, y) = c$ for the solution is the best one can do.

Example 1: Consider the 1st-order, separable, non-linear, non-autonomous differential equation:

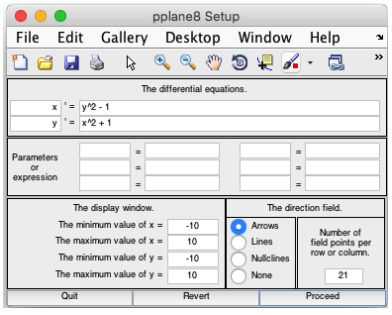
$$\frac{dy}{dx} = \frac{x^2 + 1}{y^2 - 1}$$

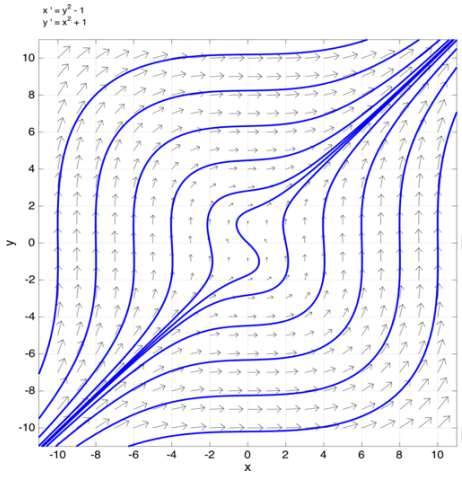
This has an implicit solution via separation of variables: $y^3 - 3y - (x^3 + 3x + 3c) = 0$

Let's show that. Just separate and integrate!

$(y^2 - 1)\,dy = (x^2 + 1)\,dx$	Separate the variables.
$\int (y^2 - 1)\,dy = \int (x^2 + 1)\,dx$	Prepare to integrate both sides.
$\frac{y^3}{3} - y = \frac{x^3}{3} + x + c$	Integrate, and add the constant of integration.
$y^3 - 3y - (x^3 + 3x + 3c) = 0$	Rearrange.

This is not the most straightforward equation to plot, since you need to solve the equation for $y(t)$ unless you have access to an implicit plotting routine. In these situations, a numerical solution such as shown below using **pplane8** will suffice to analyze the system behavior.





Challenge: See if you can create these solution curves using pplane8

In lecture last week we showed how to obtain graphical solutions using direction fields. This week we will study more precise numerical methods – Euler's method and the Runge-Kutta 4th-order method using MATLAB for the implementation.

Example 2: We will see an implementation of these methods using the following 1st-order, non-linear, autonomous example:

$$\frac{dy}{dt} = f(t,y) = 4y - y^2, \quad y(0) = 1$$

a. Euler's Method – (See lecture notes from week 1)

Euler's method, as we derived in the class notes last week, requires us to first write a function that gives the derivative. Start by defining an anonymous function in MATLAB as follows:

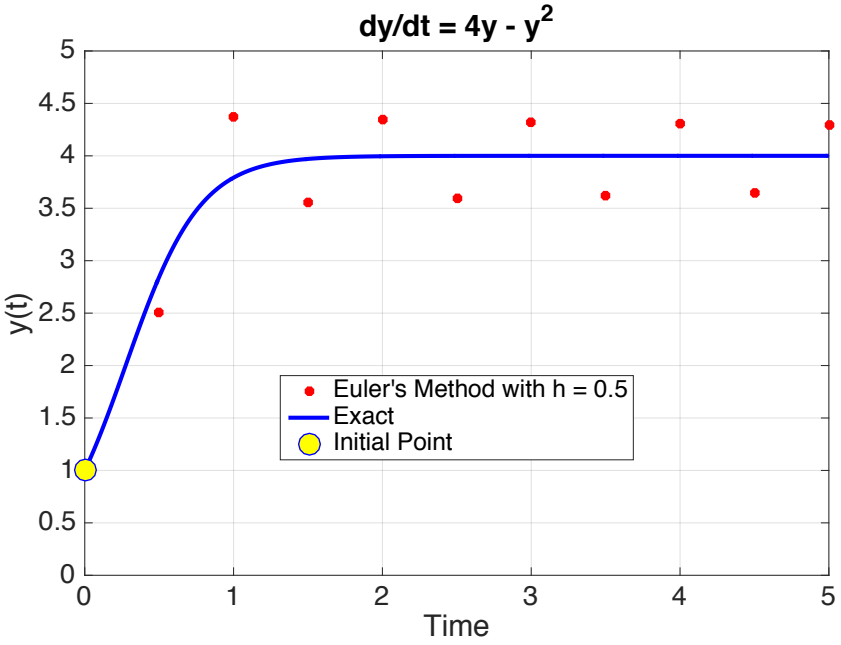
```
f = @(t,y)  4*y - y^2  % Anonymous function for the slope dy/dt.
```

Inputs **Output**

Learn more about anonymous functions here: **http://www.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html**

Note: The function has to be written with inputs (t, y) in the correct order. This is needed later for **ode45** so we will adopt this convention for defining all first order functions.

Get the code on the next page running and verify you can produce this figure. It already includes the above function.



There's just lots of great code on the next page. You should be able to reproduce almost all of it before the MATLAB Final in the last week of classes. For now, it is OK to just get the general idea.

As shown in the class notes, Euler's method solves the differential equation at a series of time points. We have to know the start time, the end time for the computation, the initial value of y and the step size. A smaller step size gives a more accurate solution, though it takes longer to compute the solutions.

Code for the exact solution using **dsolve** and a numerical solution using **Euler's Method**.

```
%% Pre-Lab 2 Fall 2016
% ENGR 232 Week 2 Euler's Method Example and exact solution using dsolve

clear, clc, close all
% First declare the function f(t,y) which gives the slope dy/dt.
f = @(t,y)  4*y - y^2  % Anonymous function.

% Initialize the Variables
dt = 0.5;  % Step size. Also called h in the notes.
tStart = 0;    yStart = 1; % Initial point.
tEnd = 5;          % Stopping time.

% Define time points and solution vector
t_points = tStart: dt: tEnd;
y_points = zeros(size(t_points));  % Use zeros as place holders for now.

% Initialize the solution at the initial condition.
y_points(1) = yStart;

% Implement Euler's method using a for loop.
for i=2:length(t_points)
    yprime = f(t_points(i-1),y_points(i-1));
    y_points(i) = y_points(i-1) + dt*yprime;
end

% Plot Solutions
figure(1)
plot(t_points, y_points, 'red*', 'LineWidth', 3)
grid on
xlabel('Time')
ylabel('y(t)')
set(gca, 'FontSize', 20)
title('dy/dt = 4y - y^2')
axis([0 5 0 5])
hold on

% Now let's find the exact solution.
syms y(t)
DE = diff(y,t) == 4*y - y^2;
sol = dsolve(DE, y(0) == 1)
Y = matlabFunction(sol)
t_points = tStart: 0.01: tEnd;
plot(t_points, Y(t_points), 'blue', 'LineWidth', 3)
plot(tStart, yStart, 'blueo', 'MarkerSize', 16, 'MarkerFaceColor', 'yellow')

legend('Euler''s Method with h = 0.5', 'Exact', 'Initial Point', 'Location', 'Best')
```

b. Next we will see how to use MATLAB's **ode45** solver for the same DE. This is generally much more accurate than Euler's method. As you have seen in the lecture notes, the **Runge-Kutta** method is more complicated and requires many more steps in the algorithm. While this is not impossible to code, MATLAB has a function that makes its implementation very easy. It's named **ode45**.

Step 1. Define the function for the differential equation as before. Be aware that the function inputs have to be listed in the order (t, y). Variable names don't matter in fact, and it should always be (independent_variable, dependent_variable)

We can just reuse our anonymous function from before. There is no need to retype it. It's just here for completeness.

```
f = @(t,y)  4*y - y^2  % Anonymous function for the slope dy/dt.
```

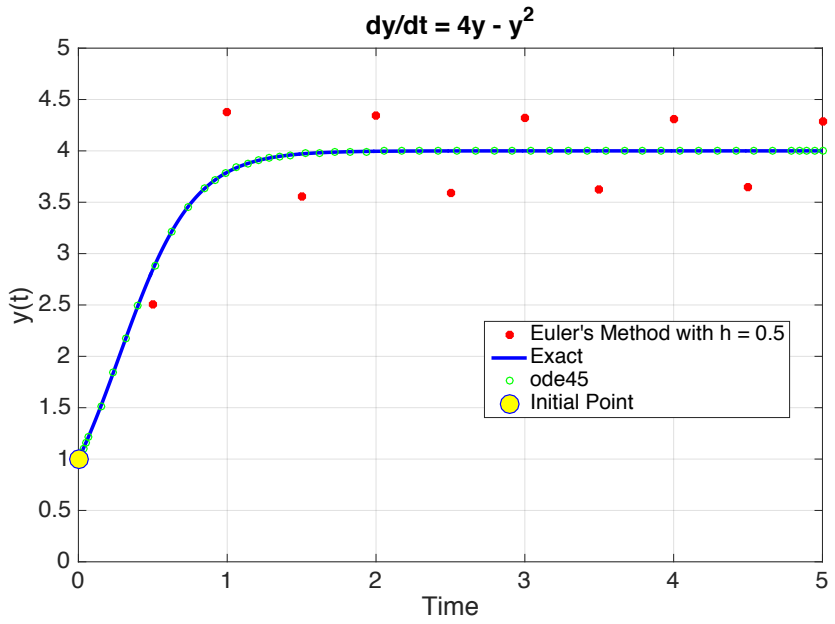
Step 2. Define the parameters to compute the solution: start time, end time, initial condition (`yStart`).

Step 3. Solve the differential equation using MATLAB's **ode45** solver:

```
[t_out, y_out] = ode45(f,[tStart tEnd], yStart);
```

Note all those arguments on the RHS must be specified first!
Step 4: Then plot the solution curve! Code not given!
For more on **ode45** type >> help ode45

Add additional code, and move the code for the legend below your new code to create the new figure shown here. Notice the **ode45** points shown in green are a very good fit! These points are right on top of the exact solution found using **dsolve**. Study how it works. You will need to be able to produce this code during the MATLAB Final.



Rerun your code, but change the step-size for Euler's method to the smaller value h = 0.1 (called dt in the code) instead of 0.5, which was really too large. Create this new image. Be sure to adjust the legend to state the new value for h.