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

Lab 2: The Logistic Equation

The growth of a certain population $P(t)$ can be modeled using an autonomous, non-linear, 1st-order differential equation:

$$\frac{dP}{dt} = rP \cdot \left(1 - \frac{P}{K}\right)$$

This is called the **logistic equation**. Above, the constant *r* defines the rate of growth, while *K* is the carrying capacity.

Part A: Functions in MATLAB

Before studying this differential equation, let's review the **logistic function** (or logistic curve), which is the common *S*-shaped curve or **sigmoid curve** defined by:

This function was named by Pierre François Verhulst, in his studies of population growth.

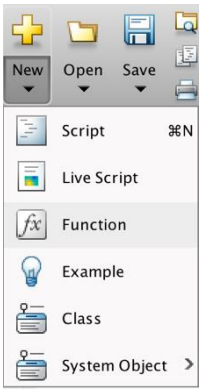
$$f(t) = \frac{L}{1+e^{-k(t-t_0)}}$$

Pierre François Verhulst

Above, L is the limiting maximum value of the sigmoidal curve, t_0 marks the midpoint and k is the steepness.
We'll show **three** different ways to enter this function into MATLAB. These skills can be used any time you need to define a function.

Method 1: In MATLAB, create a new **function** file.
The easiest way to do this is to click the **New** button at the top far left in MATLAB. Select the **Function** option. This gives the following template for a Function script.

```
function [outputArg1,outputArg2] = untitled(inputArg1,inputArg2)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```



Method 1 to define a function.
Create a function file.

a. Name the input arguments t, t0, L, k (in this exact order!)
b. Name the first output argument f. **Delete** the second output argument. We don't need it. (See **red**.)
c. Name the function as **logistic**. (replace the untitled default name)
d. Write in the summary. This should be something like:

```
% The logistic function logistic(t, t0, L, k)
% t is time and t0 marks the midpoint
% L is the limiting maximum value of the sigmoidal curve,
% and k is the steepness.
```

e. Define the function by inserting the following line of code just before the **end** keyword. Then save the file as **logistic.m**

```
f = L ./ (1 + exp(-k*(t-t0)));
```

Notice the dot to assure pointwise operations.

Required to prevent data dumps into your command window.

The **standard** logistic function is defined by: $f(t) = \frac{1}{1+e^{-t}}$

Thus, the standard values are $t_0 = 0$, $L = 1$, and $k = 1$. Verify your logistic function is working by confirming the next line returns the midpoint value of 0.5

```
>> logistic(0, 0, 1, 1)   % time t=0, midpoint t0=0, Limit L=1, steepness k=1.
```

Here's some code to plot two logistic curves with $L = +10$ and $L = -10$, and the resulting graph. This should help you quickly review some standard plotting options. It assumes you have saved the function file logistic.m in your present working directory or **pwd**.

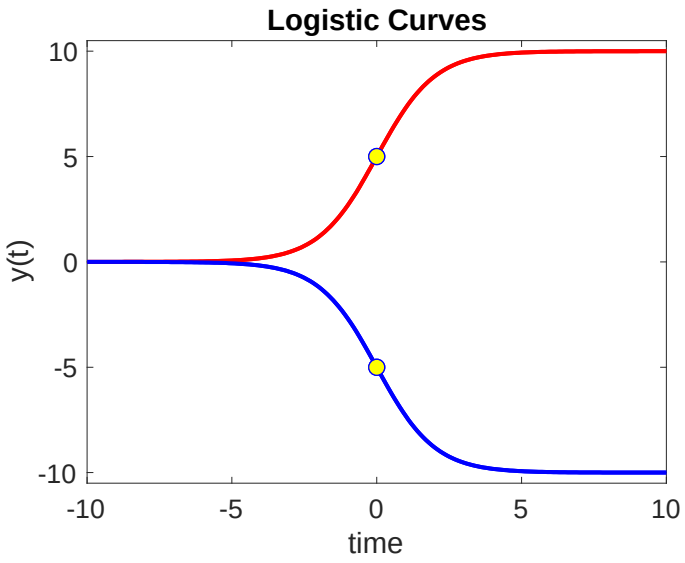
```
%% Let's first plot two standard logistic curves with L = ±10.
clear, clc
figure(1)
time_pts = -10 : 0.1 : 10;
y_pts = logistic(time_pts, 0, 10, 1);
plot(time_pts, y_pts, 'red', 'LineWidth', 3) % The standard curve.
grid on, hold on

y_pts = logistic(time_pts, 0, -10, 1);
plot(time_pts, y_pts, 'blue', 'LineWidth', 3) % The standard curve.

plot(0, +5, 'bo', 'MarkerFaceColor', 'yellow', 'MarkerSize', 8) % Add midpoint for each logistic curve.
plot(0, -5, 'bo', 'MarkerFaceColor', 'yellow', 'MarkerSize', 8) % Add midpoint for each logistic curve.

set(gca, 'FontSize', 20)
title('Logistic Curves')
xlabel('time');  ylabel('y(t)')
axis([-10 10 -10 5 10 5])
```

Resulting plot:



Questions 1-2: Adjust the code above to plot a total of 21 sigmoidal curves by varying the limit *L* from +10 to –10 in steps of –1, using a **for** loop. Have the colors gradually transition from **red** on the top to **blue** on the bottom by specifying the color option using RGB triplets. You can see more about color options here: http://www.mathworks.com/help/matlab/ref/colorspec.html

The triplets must gradually change from **blue** = [0 0 1] to **red** = [1 0 0] as L varies from +10 to –10.

The colors in the middle will be shades of purple. **Hint:** Once you update my_color inside your **for**-loop, you can use it to plot, by giving it as the option following the **'Color'** keyword.

```
plot(time_pts, y_pts, 'Color', my_color, 'LineWidth', 3)
```

Tip: Your color triplet [r, 0, b] will be zero in the middle and the values for r and b will be **lines** depending on L but always satisfying $r + b = 1$.

Don't draw the midpoints anymore.

Now add another plot command inside your **for** loop, and draw each curve twice, once with $k = +1$ (as above) and then again with $k = -1$. That's the last argument.

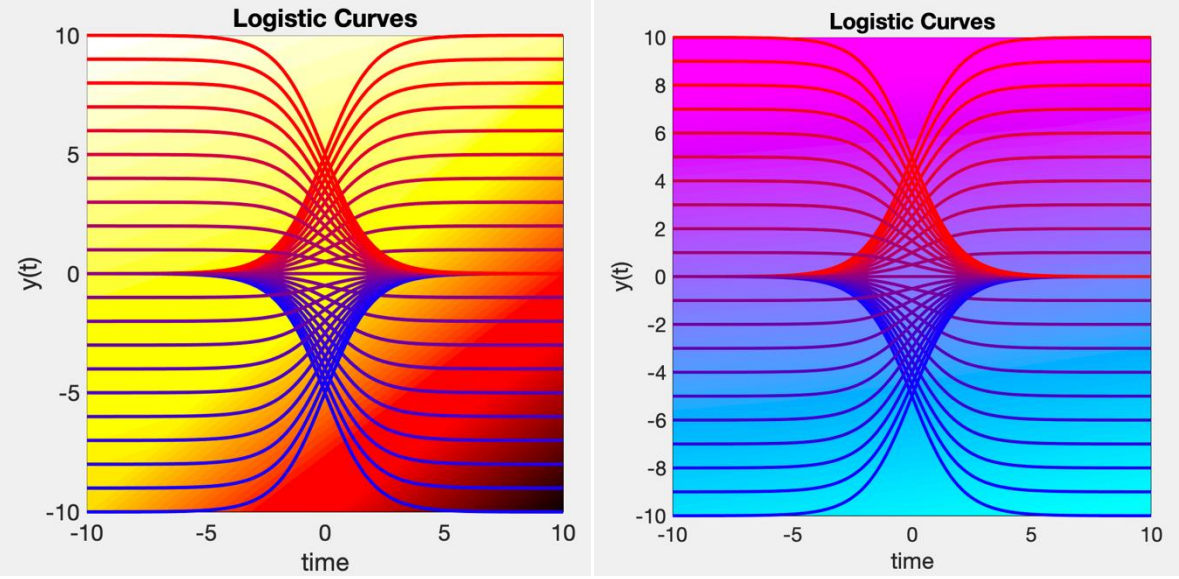
k

```
y_pts = logistic(time_pts, 0, L, +1);
```

Questions 1-2: Paste your graph with 21 sigmoidal curves and their 21 mirror images (with $k = -1$) into the Answer Template for 2 points credit. Your logistic curves should gradually transition from red to blue through shades of purple.

Sample Plots: In the samples, a colorful background has been provided. Your graph must not include this background.

Samples



Grader
1. One point if the curves are correct.
2. One point if the color gradually shifts from red to blue.

Animate!
You can animate your curves, so they appear one after another by adding a brief pause inside your for-loop.

```
pause(0.2)
```

Grader: Solution must have no background

Challenge 1: No points, but important to master before the MATLAB Final. Don't skip!!
Let's create the same plot, but instead of using the file logistic.m, we'll invoke MATLAB's **anonymous** function syntax. We'll just name our function f this time.

```
f = @(t, t0, L, k)   L ./ (1 + exp(-k*(t-t0)));
```

Method 2 to define a function.
Declare an anonymous function.

We saw this before, but you can review anonymous functions here:

https://www.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html

Recreate the same plot, but use your anonymous function f instead of logistic.m.

Method 3 to define a function.
Create a symbolic expression then use **matlabFunction**.

Challenge 2: No points, but important to master before the MATLAB Final. Don't skip!!
Let's create the same plot, but instead of using the file logistic.m, or the anonymous function f(t, t0, L, k) we'll use **matlabFunction**.

Note: G = matlabFunction(F) generates a MATLAB anonymous function G from the symbolic object F.
Learn more about **matlabFunction** here: https://www.mathworks.com/help/symbolic/matlabfunction.html

Generating a multi-argument function from a symbolic expression using **matlabFunction**.

```
syms t t0 L k
h(t,t0,L,k) =    L / (1 + exp(-k*(t-t0)));
g = matlabFunction(h)

g(0,0,1,1)  % Returns 0.5 just like before.
g(t,0,1,1)  % Returns the standard logistic function.
```

Note: We did **not** need the dot above to indicate pointwise operations, as **matlabFunction** will take care of that automatically for us, as you can see by this output for g.

g = @(t,t0,L,k) L./(exp(-k.*(t-t0))+1.0)

Now create the same plot, but instead of using the file logistic.m, or the anonymous function f, use the above function **g** which was defined via **matlabFunction**. Success? You are now a master at defining functions in MATLAB!

Part B: Finding solutions of the Logistic Differential Equation

After the above review of MATLAB **functions**, let's return to the logistic differential equation that describes the growth of a population with a carrying capacity *K*.

$$\frac{dP}{dt} = rP \cdot \left(1 - \frac{P}{K}\right)$$

Above, the constant *r* defines the maximum (per capita) rate of growth, while *K* is the carrying capacity and time *t* is in days. For this lab, we will assume the growth rate is $r = 1/2$ and the carrying capacity is $K = 100$ in **millions**.
This might describe the population $P(t)$ of bacteria in a Petri dish. Assume that after 10 days, the population has already reached **half** the carry capacity so that $\boldsymbol{P(10) = K/2 = 50}$ (in millions).
We thus have the Initial Value Problem (IVP):

$$\frac{dP}{dt} = \frac{1}{2}P \cdot \left(1 - \frac{P}{100}\right) \qquad P(10) = 50$$

Find the **exact** solution using **dsolve**, and plot over the range from *t* = 0 to 20 days.
Use a **blue** curve with a 'LineWidth' of 3. Here is some starter code.

```
clear, clc
syms P(t)
r=1/2; K = 100;
DE = diff(P,t) == r*P * (1 - P/K)
sol = simplify( dsolve(DE, P(10)==50) )
p = matlabFunction(sol)  % Use little p for the solution.
```

Question 3: Simply enter p(t) in the command window to obtain the exact solution for the DE satisfying P(10) = 50 million.

Question 3: The exact solution satisfying P(10) = 50 is $p(t) =$ _____

or

Tip: Arrange your answer in the form of a standard logistic function: $f(t) = \frac{L}{1+e^{-k(t-t_0)}}$
Recall *L* corresponds to the limit, which for the DE is the carrying capacity *K*. In the DE, $k = r$ is the maximum (per capita) growth rate *r*. Finally, $t_0 = 10$ is the location of the midpoint.

Question 4: Plot the exact solution (found with **dsolve**) over the interval from $t = 0$ to $t = 20$. Use a **blue** curve with a thickness of 3, turn the **grid on**, set the **title** to 'Logistic Equation with r=1/2, K=100 million', set the **xlabel** to 'Time in days' and the **ylabel** to 'P(t) in millions'. Turn **hold on**, so all the later graphs will appear in the same figure. Place a **yellow** dot to indicate the given point P(10) = 50.

Legend: We'll build a fairly complicated legend by the end of the lab. Be sure to create a **handle** for each component we will need in the legend. For example, when I plotted the **exact** solution, I gave that plot the handle **exact** as shown below.

```
exact = plot(t_pts , p_pts, 'blue', 'LineWidth', 3)
```

handle

Similarly, when I plotted the midpoint as a yellow dot, I gave it the handle **midpoint**.

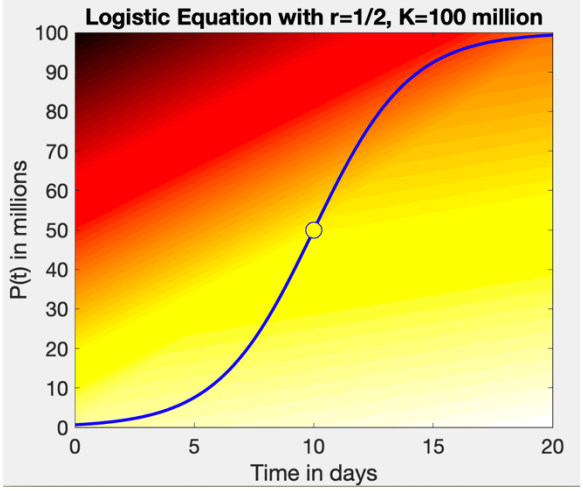
You can then add a legend that includes just the exact solution and the midpoint using their handles:

```
legend([exact, midpoint], 'Exact solution', 'Midpoint',  'Location', 'southeast')
```

[handles] **Labels for each handle** **Optional arguments**

Q4: Replace the sample below, with your own graph. Your answer must not include the colorful background pattern.

Sample



Next, we'll compare the exact solution, to several numerical approximations. First to an **Euler approximation** with a fairly large step size, and then to a better Euler approximation with smaller steps. Finally, we'll get a very good approximation using **ode45** and combine them all on the same graph.

Question 5: Slope Function for the DE

To help us explore this differential equation numerically, we will need to define its slope as a function.

$$\frac{dP}{dt} = f(t,P) = rP \cdot \left(1 - \frac{P}{K}\right), \qquad P(10) = 50 \text{ and } r = 1/2, K = 100$$

We saw there are at least three different methods to do that, but we will just use an **anonymous function** for $f(t,P)$. Enter this now in a new section of your MATLAB script for Lab 2.

```
r = 1/2; K = 100;
f = @(t,P)  r*P * (1 - P/K)
```

Note we declared t as an input even though f is **autonomous** and so does not depend on t. That will be important later when we invoke **ode45**.

Let's verify your function is working. Find the value of the **slope** at the given point (10,50). That is, find f(10,50).

Q5. The slope is: f(10,50) = _ _ _ _ _ _

The last five points are earned by completing the remaining steps and submitting your completed graph. (5 points)

Recall you already have a figure with the exact solution, and you should have turned hold on, so all these additional graphs are automatically added to the same figure.

Euler's Method with Two Different Step Sizes

Q6. Add a plot of the approximate solution using Euler's method with a step size of $dt = 2$ (This is a very coarse step size. The purpose is so we can actually **see** the error.) Plot each point as a red asterisk * and connect the points with a dotted line.
Hint: You will use the function named f above, in a **for** loop to find the Euler points. Review the Pre-Lab for code ideas. Here's just a bit of code to get you started.

```
% Initialize the variables. Here we are using y for the population.
dt = 2;                          % Step size. Also called h in the notes.
tStart = 10;  yStart = 50;       % Population starts at P(10) = 50.
tEnd = 20;                       % 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.
N = length(t_points);

% forwards in time
for n = 2:N
      % add code here inside the for loop
end
euler1 = plot(t_points, y_points, 'red*:', 'LineWidth', 2); % handle = euler1

% backwards in time
% add more code here
```

Solutions in the Forward Direction – See starter code on previous page

Now implement Euler's method using a **for**-loop. Plot the points in the same figure as your exact solution.
You already have the horizontal coordinates for each point inside the row vector named **`t_points`**.
To find the vertical coordinate `y_points(n)` for the n[th] point, first find the slope at the previous point, then use Euler's formula: rise = slope * run, where of course, run = dt.

Tip: Your **for**-loop should start at **n=2**, and end at the number of points `N = length(t_points)`

i. First calculate the slope at the previous point (with index $n-1$). The slope there is $f(t_{n-1}, y_{n-1})$

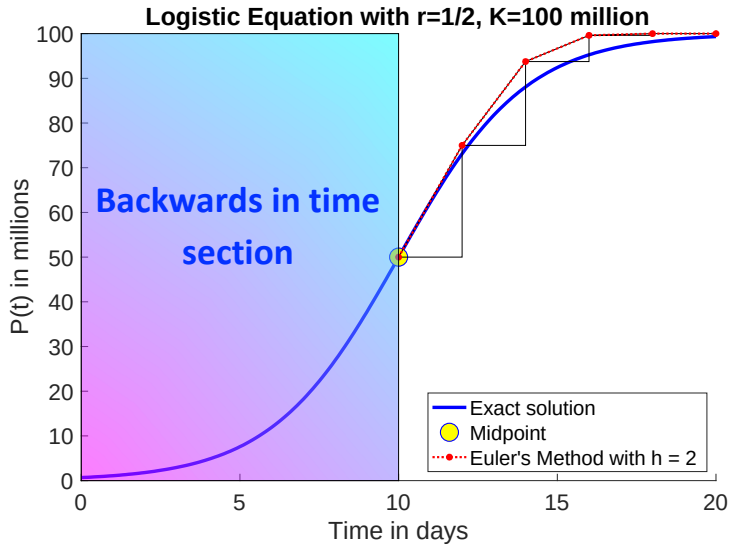
ii. Then calculate the next *y*-value using Euler's formula: $y_n = y_{n-1} + slope \cdot dt$

Recall Euler's formula just says $rise = slope \times run$, where each new point is found using a **triangle** with slope given by the tangent line. Inside your **for**-loop, show these triangles in transparent yellow using the following code.

```
tri = fill([t_points(n-1), t_points(n), t_points(n)], [y_points(n-1), y_points(n-1), y_points(n)], 'y');
tri.FaceAlpha = 0.25;
```

Update your **legend** to include the Euler approximation with $h = 2$. In the starter code, that graph is given the handle **euler1**. At this stage your combined graph should look like this, except for the colorful background on the LHS, which is there to obscure the backwards-in-time half.

Sample



There are two things to notice. First, the Euler approximation with this large step-size of dt=2, is very coarse! Second, we only obtained the Euler approximation in the forward direction from time 10 to time 20. Let's fix that now by also working backwards in time.

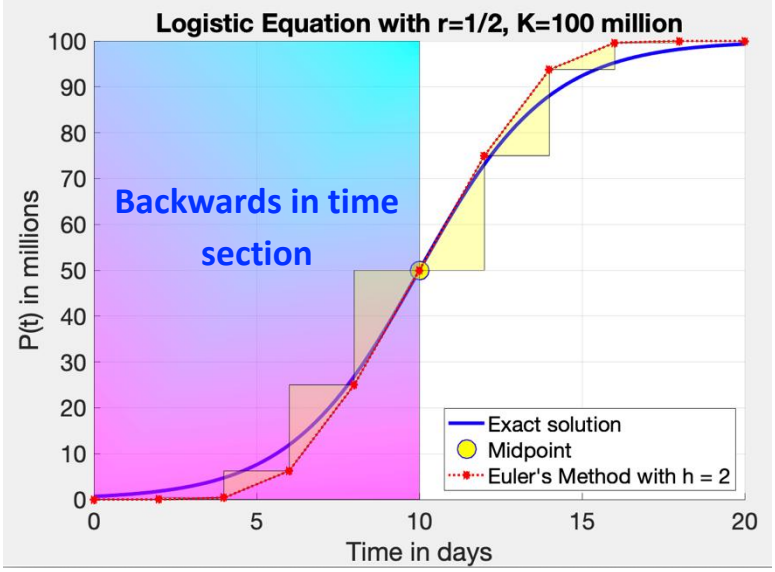
Solutions in the Backwards Direction

Add triangles showing the Euler approximation moving backwards in time, from time 10 to time 0.
Plot red asterisks to show each Euler point.

You should be able to clone most of your forward code after resetting: `dt = -dt` but now:

```
t_points = tStart: dt: 0;
```

Sample

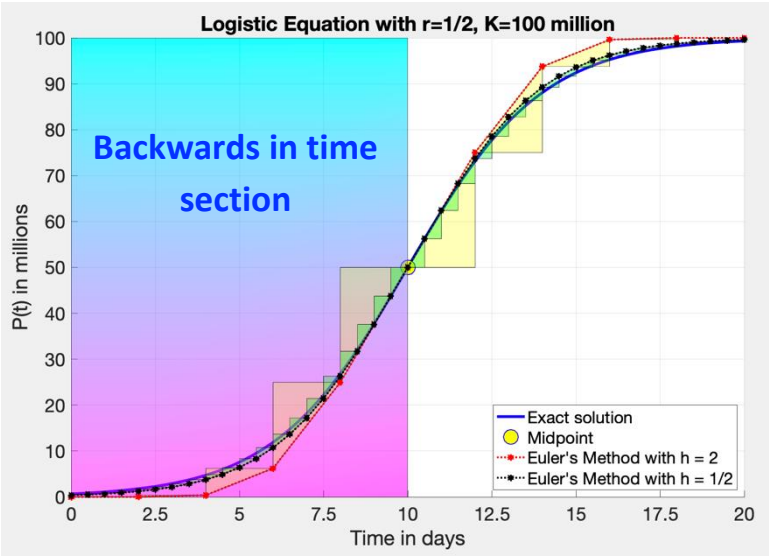


Euler Approximation with a Finer Step Size

Q7. Clone your Euler code, (both forwards **and** backwards in time) and repeat the calculations but now use a finer step-size of $dt = 1/2$. The fit should be quite a bit better.

Add another plot of the approximate solutions using this finer version for Euler's method. Plot each point as a **black** asterisk and connect the points with a dotted line.

Draw the small triangles for this part in green. Emulate the sample code from the previous part.
Except for colorful background on the left, your graph should now look much like this.



Tip: To include a single quote in a legend entry, as in Euler's, just type it twice: `"Euler''s"`

Tip: If some unwanted graphical components are appearing in your legend, just specify this option in the plot or fill command where they are created:

```
'HandleVisibility', 'off'
```

Q8. Now find the approximation using **ode45**. It's usually very good!

Here's the main command you need, provided you have defined all the variables given as its arguments.

```
% forward in time
[t_out, y_out] = ode45(f, tStart: 1: tEnd, yStart);
```

Show the ode45 solution points as **green** diamonds with **red** outlines. **Hint:** d is for diamonds.
Set the face color to green using the 'MarkerFaceColor' keyword. Set the 'MarkerSize' to 10.
The outline is specified as usual.

Now add the ode45 points going **backwards** in time from 10 to 0. Don't skip!!

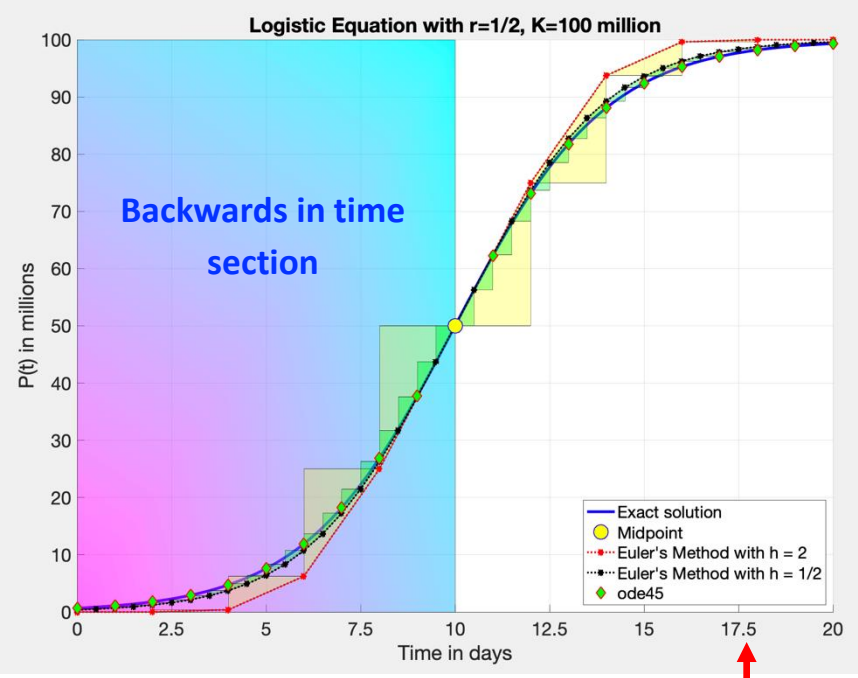
Q9. Add a **legend** to your figure, similar to that shown in the sample plot below.

Q10. Locate your legend to avoid the data points as much as possible. Adding these two arguments at the end of your **legend** command will position the legend to avoid as many data points as possible.

```
'Location', 'Best'   ← Place as the last two arguments inside the legend command.
          or
'Location', 'Southeast'
```

Replace the graph below, with your own graph.
The LHS of your graph must not include the colorful pattern. **(5 points)**

Sample →



Grader will award one point for each feature in this list. 5 points.

1. Both Euler approximate solutions are shown correctly.
2. There are small yellow and green triangles.
3. The ode45 solution is shown with **green** diamonds with **red** outlines
4. Graph includes the points backwards in time on the LHS.
5. The legend does not obscure the data points.

The legend has avoided obstructing the data points thanks to setting the 'Location' to 'Best' or 'Southeast'.

Grader: Image must **not** include the colorful pattern on the LHS.

Ready to Submit?

Be sure all ten questions are answered. When your lab is complete, be sure to submit three files:

1. Your **completed Answer Template** as a PDF file
2. A copy of your **MATLAB Live Script**
3. A **PDF** copy of your **MATLAB Live Script** (Save-Export to PDF…)

The due date is the day after your lab section by **11:59pm** to receive full credit. You have one more day, to submit the lab (but with a small penalty), and then the window closes for good and your grade will be zero.