ENME202 Matlab

FLOW CONTROL: FOR and WHILE loops

FOR LOOPS

The "for" loop repeats a set of commands a specified number of times.

Here's a simple loop to repeat an action a specified number of times. In this case, the loop repeats 5 times, and simply displays "Hello world" on the screen:

```
for k = [1,2,3,4,5]
        disp('Hello world');
end
```

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

The variable k is called the "loop variable", and the loop executes as follow:

1. The loop variable is initialized to the first value in the array (k=1), and code in the loop is executed.
2. When the loop code is completed, the next value in the array is assigned to the loop variable k (k=2) and the loop code is executed again.
3. This process repeats for each value in the array. When all values are exhausted, the loop ends and the code continues on the next line below the loop block.

You can see this process explicitly by having the loop show the value of k:

```
for k = 1:3
  disp(k)
end
```

```
     1

     2

     3
```

Any valid array can be used in a for loop. For example, we can define the loop variable range as an array with non-unity steps:

```
for k = 2:-.5:0
  disp(k)
end
```

```
     2

    1.5000

     1
```

```
    0.5000

     0
```

The array values don't need to be sequential, and can be real or complex:

```
for k = [4 0 1 99 pi 2+3i]
  disp(k)
end
```

```
     4

     0

     1

    99

    3.1416

   2.0000 + 3.0000i
```

The array can also be assigned to a variable, and then used in the for loop as follows:

```
a = [4 0 1 99 pi 2+3i];

for k = a
  disp(k)
end
```

```
     4

     0

     1

    99

    3.1416

   2.0000 + 3.0000i
```

The loop variable can be used for calculations within the loop. For example, the loop below displays the powers of 2 from 1 to 10:

```
for k = 1:10
  disp(2^k)
end
```

```
     2

     4

     8
```

```
    16

    32

    64

   128

   256
```

Here is another loop that repeatedly adds 1 to variable foo (which has its initial value set to 0) and re-assigns the result back to foo:

```
foo = 0;
for k = 1:10
  foo = foo + 1;
end
disp(foo)
```

```
    10
```

Sum first N inverse powers of 2: 2^0 + 2^(-1) + 2^(-2) + ... :

```
s = 0; N=5;
for k = 0:N
  s = s + 2^(-k);
  disp(s)
end
```

```
     1

    1.5000

    1.7500

    1.8750

    1.9375

    1.9688
```

We can also use the loop index variable as an **array index** within the loop. For example, let's implement the equivalent of Matlab's polyval() function manually, by evaluating a 2nd order polynomial defined by p (assuming standard Matlab order) at each of the points in array x, returning the results to a new array f.

Also, add up all the f(k) values as part of the same loop:

```
% Manual version of polyval()

p = [1 -2 3];   % polynomial coefficients
x = -3:.02:1;   % independent variable values
s = 0;          % summation variable for use in loop
f = zeros(length(x),1);  % pre-allocate memory for f to avoid
```

```
                        % expensive task of increasing array
                        % size during each iteration

for k = 1:length(x)      % loop through each value of x
    f(k) = p(1)*x(k)^2 + p(2)*x(k) + p(3);   % evaluate p(x_k)
    s = s + f(k);                            % add result to cumulative sum
end

length(x)   % ans = 201
```

```
ans = 201
```

```
length(f)   % ans = 201
```

```
ans = 201
```

```
s           % s = 1476.7
```

```
s = 1.4767e+03
```

```
sum(f)      % ans = 1476.7
```

```
ans = 1.4767e+03
```

NESTED LOOPS

We can "nest" loops, i.e. put a loop, within a loop, within a loop, etc...

Consider an example where we want to display each element in a matrix, row by row:

```
A = [ 1 2 3
      6 5 4
      0 0 9 ];

for i=1:3
  for j=1:3
    disp(A(i,j))
  end
end
```

```
     1

     2

     3

     6

     5

     4

     0

     0
```

As another example, consider the polynomial earlier problem, where p was required to be a 3-element array (2nd order polynomial). Let's go one step further by modifying the code to work with an arbitrary length array:

```
p = [1 2 3 -4 0];   % 4th order polynomial
x = -3:.02:1;       % independent variable values
f = zeros(length(x),1);  % pre-allocate memory for f
for i = 1:length(x) % loop through each value of x
    f(i) = 0;       % initialize result (can you see why this is needed?)
    r = length(p);  % polynomial order
    for j = 1:r     % evaluate each polynomial term, and sum the results
        f(i) = f(i) + p(j)*x(i)^(r-j);
    end
end
```

Here are some additional loop examples, set up as functions (commented out since we will use mysinN() in the following code, so the function must be defined at the end of the file):

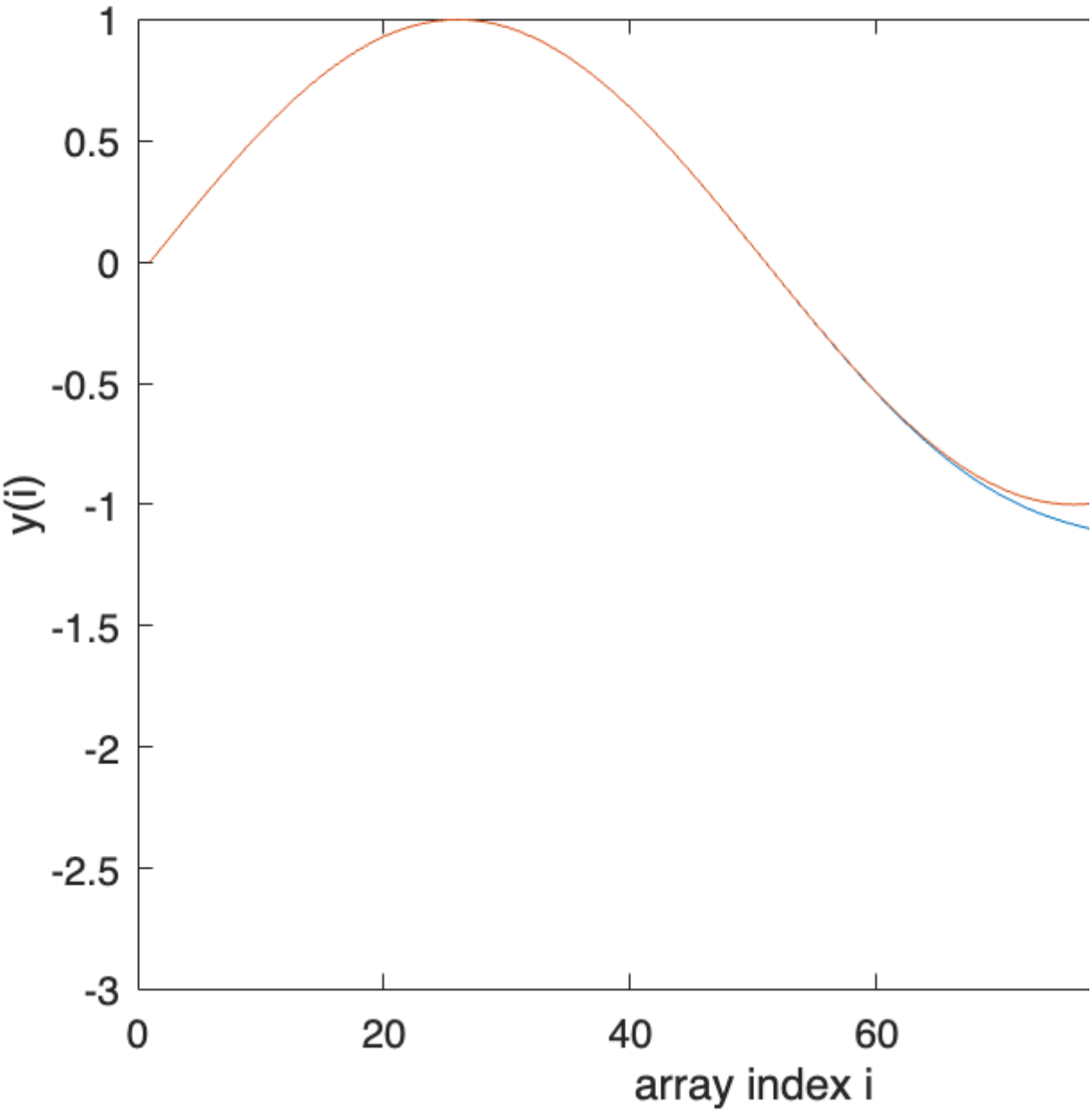
```
%{

function s = mysin(x)
  % Compute 3 term approximation to sin(x), without loop
  s = x - x^3/6 + x^5/120;
end


function s = mysin2(x)
  % Compute a 3 term approx to sin(x) using a loop
  s = 0;
  for k=0:2
    s = s + ((-1)^k)*x^(2*k+1)/factorial(2*k+1);
  end
end


function s = mysinN(x,N)
  % Compute an N-term approx to sin(x) using a loop,
  % with N passed as an argument to the function
  s = 0;
  for k=0:N-1
    s = s + ((-1)^k) * x^(2*k+1) / factorial(2*k+1);
  end
end

%}
```

Example: Write a loop that calculates a 6-term sin() approximation over the range of x = 0 --> 2*pi, and plot the result together with the true sin curve for comparison.

Hint: To generate the plot, we will need to store the values in an array, and thus will need to keep track of the index into the array as new values are being assigned during each loop iteration. Think about this before we look at the solution below...

[...]

```
clear
figure(1); clf

n = 100;          % # of values to calculate over 0->2pi range
step = 2*pi/n;    % step size

sin_approx = zeros(n,1);
sin_true = zeros(n,1);

for i = 1:n           % loop over the number of values
    x = (i-1)*step;   % determine x for each index
    sin_approx(i) = mysinN(x,6);  % custom function to approximate sin(x)
    sin_true(i) = sin(x);         % find Matlab's "true" sin(x) value
end
plot(sin_approx)
hold on
plot(sin_true)
set(gca,'fontsize', 14)        % change the plot font size
xlabel('array index i'); ylabel('y(i)')
```



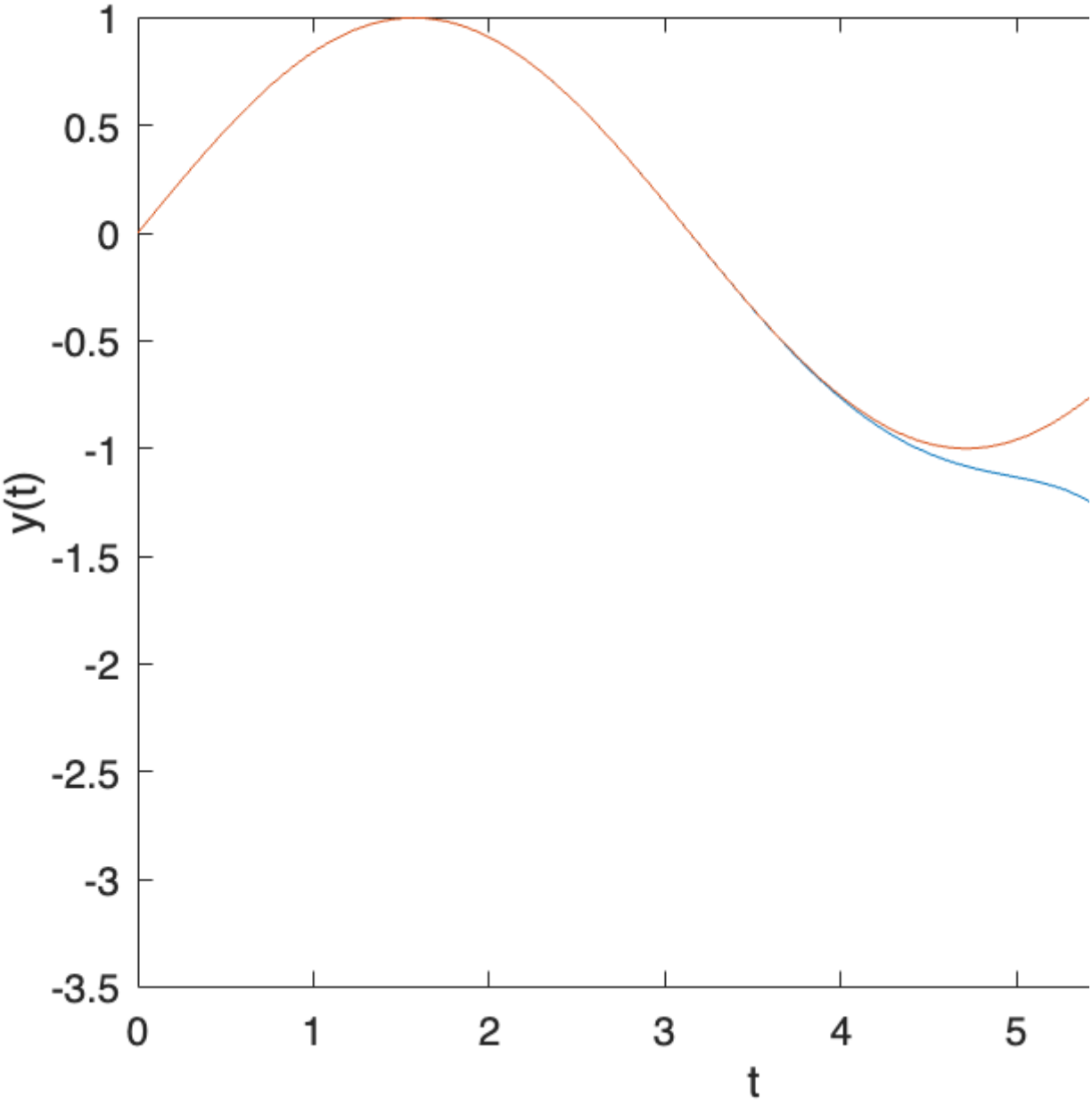
Note: can you think about a way to avoid using a loop for this process by modifying our mysinN() function? Specfically, what if we could pass an array of x values (instead of single x value) to mysinN() (and still using a loop in the function, but not in our function call)? Always consider whether you can take advantage of Matlab's array capabilties before writing unnecessary code!

In the above plots, the x-axis is just the index value of the sin arrays. How can we modify the code to include the actual x-axis values?

Think about it....

```
clear
figure(1); clf

n = 100;
```

```
step = 2*pi/n;

sin_approx = zeros(n,1);
sin_true = zeros(n,1);

x = 0 : step : 2*pi;          % (1) set up array of x-axis values, or
for i = 1:n+1
    % x(i) = (i-1)*step;             % (2) define x in loop as alternate to (1)
    sin_approx(i) = mysinN(x(i),6);  % note use of x(i) here...
    sin_true(i) = sin(x(i));         % ...and here
end
plot(x,sin_approx)                   % plot with x array
hold on
plot(x,sin_true)
set(gca,'fontsize', 14)        % change the plot font size
xlabel('t')
ylabel('y(t)')
```



WHILE LOOPS

Unlike the for loop, which runs through the loop a fixed number of times before stopping, the **while** loop executes continuously as long as the loop argument is true.

```
a = 0;
while a < 6    % stay in loop as long as "a < 10" is true
  a = a + 1;
  disp(a)
end
```

```
     1

     2

     3
```

```
     4

     5

     6
```

A while loop can also be used to create a program that never ends (an infinite loop):

```
% while 1             % 1 = true (0 = false)
%   do something
%   ctrl-c to stop the program
% end
```

Note that while loops are required when the condition for stopping the loop changes within the loop itself. Thus while loops can do things that for loops cannot, since for loops require that the start and end points of the loop are pre-defined when the loop is first entered.

On the other hand, any for loop can be re-written as a while loop, for example:

```
a = [-2.5 5.0 54.1 22];
for k = 1:length(a)       % the index k increments automatically
  disp(a(k))
end
```

```
   -2.5000

     5

   54.1000

    22
```

Equivalent while loop:

```
a = [-2.5 5.0 54.1 22];
k = 1;
while k <= length(a)
  disp(a(k))
  k = k+1;    % increment the index k manually
end
```

```
   -2.5000

     5

   54.1000

    22
```

Note that in the above while() loops we use comparisons like "<" and "<=". These are logical comparisons that yields a result that is either true (1) or false (0). We will talk more about logical comparisons in the next lecture topic.

Again: a **while** loop should be used when the number of times the loop needs to be executed is dependent on what is happening inside the loop, and a **for** loop should be used when the number of times to execute the loop is known before the loop is started.

For example, here is another version of mysin (but not written as a function) that uses a while loop to keep evaluating terms in the power series until the newest term is below a defined tolerance limit (compared to our previous version, where the number of iterations was defined before starting the loop):

```
x = 3.1415 / 4;    % evalaute sin(x)
kterm = x;
s = 0;
k = 0;
tol = 1e-12;  % tolerance

% Keep adding terms in the series until the size
% of the current term is <= mytol (or equivalently:
% WHILE the size of kterm is GREATER than tol):

while (abs(kterm) > tol)
  kterm = ((-1)^k)*x^(2*k+1)/factorial(2*k+1);
  s = s + kterm;
  k = k + 1;    % Must now increment k ourselves at each cycle!
end

disp(s)
```

```
    0.7071
```

BREAK STATEMENT

A "for" or "while" loop can be exited at any time using "break". This will become very useful later when we can combine break with conditionals ("if" statements) so that a loop can be stopped when some particular condition is met (separate from the loop conditions, that is).

```
for i=1:10
  disp(i)
  break
end
```

```
     1
```

This loop will only execute a single time, since the break statement is reached during the first time through the loop.

A few things to note...

All of Matlab's array arithmetic tricks can be done using loops. However, Matlab's array arithmetic is implemented (using loops!) VERY efficiently using "vectoralized" code, that is MUCH faster than our loops when working with large arrays.

Loops have much more general uses, though, as some of the current (and future) examples show, and programming languages like C++ which don't have Matlab's array arithmetic features will REQUIRE loops when working with arrays.

EXAMPLE – combining FOR and WHILE loops

Let's write an ODE solver using the Euler method (finite difference approximation)

Approximate any derivitive terms in the ODE using discrete time steps dt, so that:

$$t(i) = t(i-1) + dt \qquad \text{[Eq. 1]}$$

and:

$$y' = (y(i) - y(i-1)) / dt \qquad \text{[Eq. 2]}$$

Solve the ODE for y(i) at "sufficiently small" time steps to get the solution. What is small enough? One approach is to guess a starting value for dt, and solve for y using this value. Then, cut dt in half and try again -- if the new solution for y changes more than some critical value, then keep dividing dt by 2 until the solution is below this critical value.

So if we are given the following ODE,

$$y'(t) = 1 - y(t), \quad y(0)=y0$$

the we can solve Eq. 2 for y(i), and substitute y' as defined by the given ODE to yield:

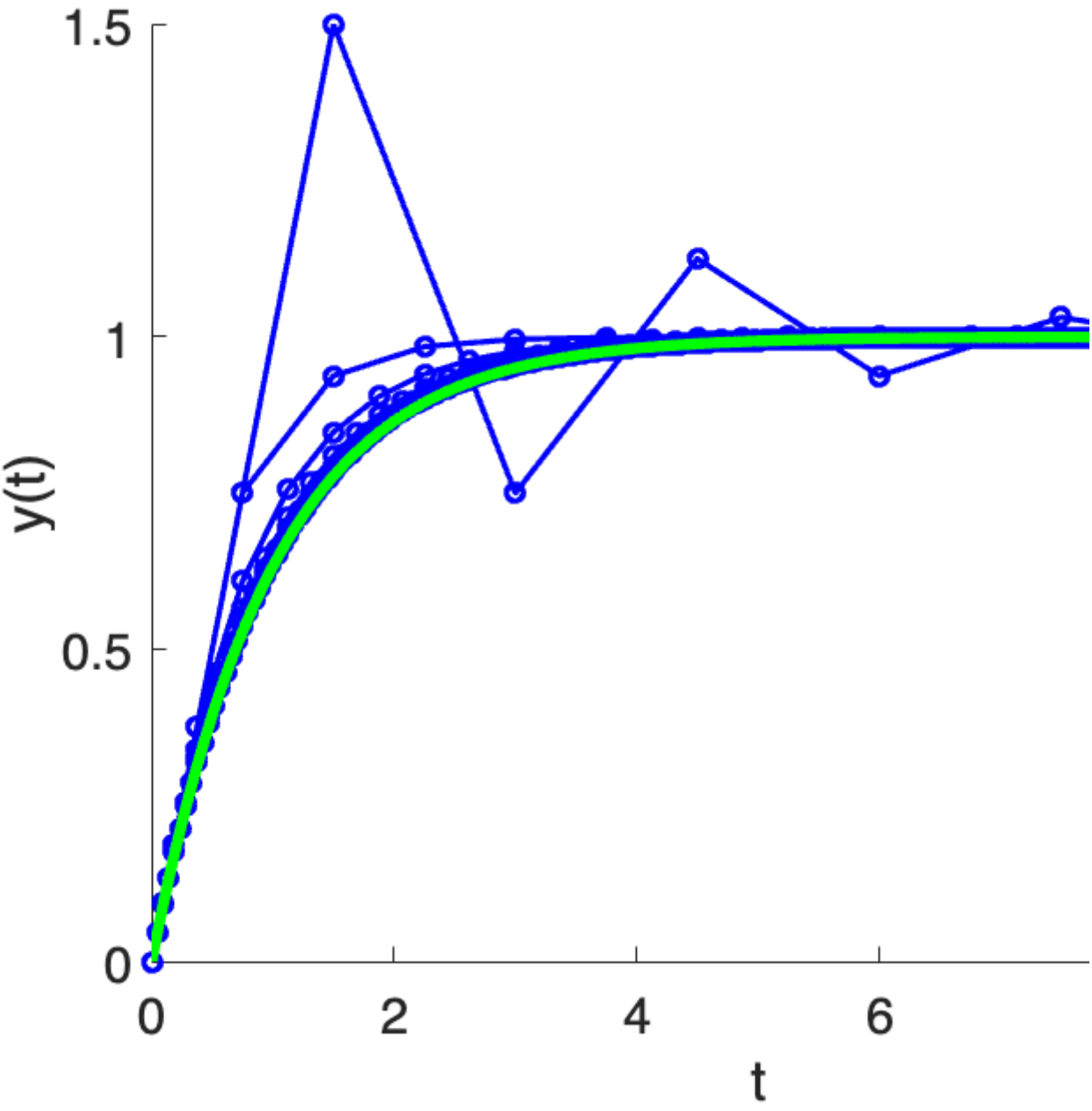
$$y(i) = y(i-1) + dt*y'$$

$$= y(i-1) + dt*(1- y(i-1)) \qquad \text{[Eq. 3]}$$

We can then implement Eqn. 3 (together with Eq. 1 to define the time points) in a loop to solve the ODE:

```
clear

y0 = 0.0 ;   % initial value of y
dt = 1.5 ;   % initial time step
y = [y0] ;   % initial condition (1st element of y(t) array)
t_max = 10;  % max time to run the solver until

% Keep track of the area under the curve (this is just one option,
% others are possible):
area = 100;   % start big to force at least one iteration

% Set the allowable area delta between iterations:
delta_limit = 0.001;

% Initialize the current delta to something larger than the allowable
% limit (to force entry into the while loop below):
delta = delta_limit + 1;

figure(1); clf; hold on   %  Get ready for plotting

while delta > delta_limit

    t = 0:dt:t_max ; % define the time series

    %  Step forward in time, one dt unit at a time
    for i = 1:length(t)
      y(i+1) = y(i) + dt*( 1 - y(i) ) ;  % our ODE expression
    end
```

```
    area_old = area;
    area = sum(y)/length(y);
    delta = abs(area - area_old);
    dt = dt / 2;
    plot(t, y(1:end-1), 'b-o', 'linewidth', 2)
    set(gca,'fontsize', 18)      % change the plot font size
    xlabel('t'); ylabel('y(t)')
    hold on
end

% Finally, add the analytic solution for comparison
% (exponential solution for 1st order linear ODE):
y_exact = 1 + (y0-1)*exp(-t) ;   % analytical solution
plot(t,y_exact(1:i),'g','linewidth',4)
```



Here is our mysinN function definition so that the above code can access this function:

```
function s = mysinN(x,N)
  % Compute an N-term approx to sin(x) using a loop,
  % with N passed as an argument to the function
  s = 0;
  for k=0:N-1
    s = s + ((-1)^k) * x^(2*k+1) / factorial(2*k+1);
  end
end
```