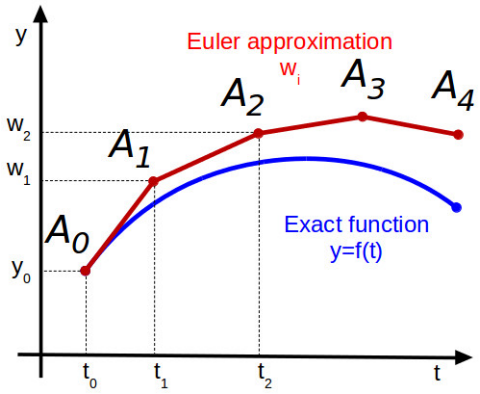
Euler:
A basic ODE solver

MATH1902: Numerical Solution of Differential Equations

http://people.sc.fsu.edu/~jburkardt/classes/math1902_2020/euler/euler.pdf

---



The Euler method steps off in the right direction, and gradually drifts away.

Euler

Writing a simple, general Euler code to solve a differential equation will give you an idea of the kind of mathematical problems we are interested in, and the computational methods we use to solve them.

1 Introduction

This is an introduction for all of us. You would like to know what we are working on. We would like to know how well prepared you are for the computational efforts in this directed study class.

We ask you to read this introductory material, which outlines a method for approximating the solution of an ordinary differential equation (ODE). This method can be expressed in the MATLAB programming language, and so we ask you to try to write *euler.m*, a file of MATLAB commands that carry out the procedure for a general ODE. Then we consider a specific ODE, which describes the shape of a curve by giving a starting value and the direction of motion at all subsequent times, and we write a MATLAB file *humps_deriv.m* which expresses the direction information. Now we are ready to solve the problem, but we need one more file *euler_humps.m*, which specifies the initial value, tells the euler method which problem to solve, and makes a plot of the results.

We encourage you to try to follow the directions below that guide you in writing these programs and using them. Please let us know if you can't figure out to form a particular MATLAB expression, or can't understand an error message.

We hope that you will be able to write the three programs, run them together, and get a nice plot of the results. You will be asked to submit your plot file as evidence of your work.

2 A differential equation

Consider the following differential equation, which can be thought of as defining the slope of a curve $y = f(t)$ at every point.

$$\frac{dy}{dt} = \frac{-2(t-0.3)}{((t-0.3)^2+0.01)^2} + \frac{-2(t-0.9)}{((t-0.9)^2+0.04)^2}$$

If we also know the value of the function at a particular time, then theoretically we may have enough information to recover the formula for the original function. However, often this process is not possible. Nonetheless, we can use this information to construct an approximate picture of the solution, by estimating the value of the function at a sequence of points. This is the meaning of the numerical solution of a differential equation.

For this example, we will assume the initial condition $y(0) = 5.1765$, and we will want to construct a plot of $y(t)$ over the interval $0 \leq t \leq 2$.

3 *humps_deriv.m*: the derivative function

In MATLAB, a derivative function has a typical structure. If we follow this structure, then we can use the same derivative function with many different ODE solvers that MATLAB provides. Even though we aren't ready to do that yet, we will follow the pattern, and this will pay off in the future.

Write a MATLAB function named *humps_deriv.m*, with the following format, filling in the details:

```
1 function dydt = humps_deriv ( t, y )
2   dydt = (the formula above);
3   return
4 end
```

Listing 1: Outline of the derivative function humps_deriv.m.

In MATLAB, the symbols *, /, and ^ are used to indicate multiplication, division, and exponentiation. You will need to be careful to use parentheses to correctly define your fractions. In general, you should terminate each MATLAB command with a semicolon; otherwise MATLAB will print out the value computed by that command, which is usually not what you want. If your text line is very long, you can break it into two by using three periods, as in:

```
1   x = 1 + 2 + 3 + 4 + 5 + 6;
2
3   x = 1 + 2 + 3 ...
4     + 4 + 5 + 6;
```

Listing 2: You can use a triple period to break a long line.

4 *euler.m*: approximate ODE solver

The forward Euler method for solving an ODE is very simple:

- Assume your position is (t,y);
- Assume you have chosen a stepsize dt;
- Compute the current slope dy/dt;
- Update your position to t+dt, y+dy/dt*dt;
- Repeat as often as desired.

Now we will write a corresponding MATLAB function, named *euler.m*. Note that this function is "general", that is, it is not written for a specific problem, but allows the user to specify, through input variables, the details of the particular problem to be solved:

The function will have the rough form:

```
function [ t, y ] = euler ( dydt, tspan, y0, n )
  (initialization)
  for i = 1 : n
    compute next t and y
  end
  return
end
```

Listing 3: Outline of the general ODE solver euler.m.

Here, the input variables are:

- `dydt` is the name of a derivative function. When we specify a particular name, we will precede it with an @ sign.
- `tspan` is a vector of length 2. `t(1)` is the start, and `t(2)` the final time.
- `y0` contains the value of y at the initial time.
- `n` is the number of equal steps to take from `t(1)` to `t(2)`.

and the output is:

- `t` a vector of length `n+1`, containing the initial time, and the `n` later times.
- `y` is a vector of length `n+1`, containing the initial y0, and the `n` later estimates.

5 Filling in the details

The initialization part of the code will look something like this

```
  t = ?;
  y = ?;
  t(1) = ?;
  y(1) = ?;
  dt = ?;
```

Listing 4: The initialization commands for euler.m

Specifically, we need to do the following things:

- create vectors t and y, to store the results. We can use the MATLAB command

  ```
  vector = zeros ( rows, 1 );  % rows is the number of rows in the vector.
  ```

  We are going to take n steps, but we also want to include the initial values, so both vectors should be created with `n+1` rows;
- Set the first entries of t (contained in tspan(1)) and y (the value y0).
- Set the time step dt. What is the formula for dt so that n steps of size dt will take us from tspan(1) to tspan(2)?

Inside the `for` loop, we need to evaluate the next values of the variables t(i+1) and y(i+1).

```
  t(i+1) = t(i) + dt;
  y(i+1) = y(i) + dt * dydt ( t(i), y(i) );  % We evaluate the dydt function here.
```

Now the euler code should be ready to go.

6 Use the solver and the derivative functions to make an estimate

Let's try a quick test run of your codes to see if things are correct. Don't end this command with a semicolon. We want to see the value of the output: You can issue the following command:

```
[ t, y ] = euler ( @ humps_deriv, [0.0,2.0], 5.1765, 10 )
```

Listing 5: Try 10 steps from 0 to 2.

```
t =

   0.00000
   0.20000
   0.40000
   0.60000
   0.80000
   1.00000
   1.20000
   1.40000
   1.60000
   1.80000
   2.00000

y =

   5.1765e+00
   1.7675e+01
   1.1867e+02
   2.1050e+01
   1.6150e+01
   2.9192e+01
   1.2072e+01
   4.4357e+00
   1.7620e+00
   5.8524e-01
  -3.0498e-02
```

Listing 6: Output from a 10 step run.

Now, to see a picture of your results, type:

```
plot ( t, y );
```

Listing 7: Plotting your data.

7 *euler_humps.m*: Collecting commands into a script

Computing is an experimental process. In a MATLAB interactive session, you may need to type 6 or 10 commands in a row before you get a result. If you want to see the result again, you have to remember what those commands were, and type them all over again correctly. If you want to repeat the experiment, but with one parameter changed, you again may have to type in your commands again. This happens especially when the result of one command depends on the output of previous commands.

So while interactive experimentation is very helpful in feeling your way towards a solution, I strongly recommend that, once you know what you want to do, you create a *script file* that lists your commands in order. That way, you can easily execute the entire experiment with a single command. If you want to change a small part of the experiment, you use an editor to modify just that one item, and run it again. If there are always one or two variables that should be specified interactifely, these can simply be input to your script.

Here's how it would work for the example we just ran. First, we choose a name for the script file. Here, I would choose the name *euler_humps.m*. As it turns out, the number of steps to be taken really ought to be chosen at the last minute, so we will make that an input variable:

```
function euler_humps ( n )
  return
end
```

Listing 8: Script version 1.

Now we stick in the commands that we used before, except that, instead of using the value 10, we now use the input value `n`:

```
function euler_humps ( n )

  [ t, y ] = euler ( @humps_deriv, [0.0,2.0], 5.1765, n );
  plot ( t, y );

  return
end
```

Listing 9: Script version 2.

At this point, you can run your script with 20 steps. This should immediately produce a plot, and it should be a little smoother than the previous one:

```
euler_humps ( 20 );
```

Listing 10: Run the script.

Now our script includes some quantities that show up only as numbers. I prefer to use named variables, that suggest what these quantities mean. So, even though it's more work, I would rewrite the script as:

```
function euler_humps ( n )

  dydt = @ humps_deriv;
  tstart = 0.0;
  tstop = 2.0;
  tspan = [ tstart, tstop ];
  y0 = 5.1765;

  [ t, y ] = euler ( dydt, tspan, y0, n );

  plot ( t, y );

  return
end
```

Listing 11: Script version 3.

and you can see that my call to `euler()` uses the same names as were used in the text of the `euler()` function.

Finally, it's possible to improve the look of the plot. MATLAB plotting commands can be a bit mysterious, so here I will simply list them, and note that the `print()` command actually allows us to save a copy of the plot as a **png** file.

```
function euler_humps ( n )

  dydt = @ humps_deriv;
  tstart = 0.0;
  tstop = 2.0;
```

```
6     tspan = [ tstart, tstop ];
7     y0 = 5.1765;
8
9     [ t, y ] = euler ( dydt, tspan, y0, n );
10
11    plot ( t, y, 'linewidth', 3 );
12    grid ( 'on' );
13    xlabel ( '<-- T -->' );
14    ylabel ( '<-- Y(T) -->' );
15    title ( 'Euler approximation to humps ODE' );
16    print ( '-dpng', 'euler_humps.png' );
17
18    return
19  end
```

Listing 12: Script version 4.

Try this improved version of the script, using `n=40` as input. You should see the plot on the screen, but you should also find a PNG plot file *euler_humps.png* in your current directory. Remember the `print()` command, since we will often ask you to verify a homework problem by sending us a PNG plot of the results!

8 Homework #1

If you followed the commands in the previous section, then you have created a PNG plot file *euler_humps.png* for the case `n=40`. As a verification, send this file as an attachment to `trenchea@pitt.edu`.