MAT 275 Laboratory 3
Numerical Solutions by Euler and Improved Euler Methods (scalar equations)

In this session we look at basic numerical methods to help us understand the fundamentals of numerical approximations. Our objective is as follows.

1. Implement Euler's method as well as an improved version to numerically solve an IVP.
2. Compare the accuracy and efficiency of the methods with methods readily available in MATLAB.
3. Apply the methods to specific problems and investigate potential pitfalls of the methods.

Instructions: For your lab write-up follow the instructions of LAB 1.

Euler's Method

To derive Euler's method start from $y(t_0) = y_0$ and consider a Taylor expansion at $t_1 = t_0 + h$:

$$\begin{aligned} y(t_1) &= y(t_0) + y'(t_0)(t_1 - t_0) + \dots \\ &= y_0 + hf(t_0, y(t_0)) + \dots \\ &= y_0 + hf(t_0, y_0) + \dots \end{aligned}$$

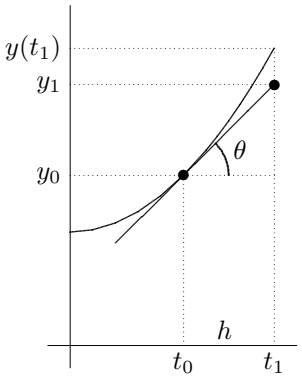
For small enough h we get an approximation y_1 for $y(t_1)$ by suppressing the ..., namely

$$y_1 = y_0 + hf(t_0, y_0) \tag{L3.1}$$

The iteration (L3.1) is repeated to obtain $y_2 \simeq y(t_2), \dots$ such that

$$\boxed{\begin{aligned} y_{n+1} &= y_n + hf(t_n, y_n) \\ t_{n+1} &= t_n + h \end{aligned}}$$

Geometrically, the approximation made is equivalent to replacing the solution curve by the tangent line at (t_0, y_0). From the figure we have



$$f(t_0, y_0) = f(t_0, y(t_0)) = y'(t_0) = \tan\theta = \frac{y_1 - y_0}{h},$$

from which (L3.1) follows.

As an example consider the IVP

$$y' = 2y = f(t, y) \quad \text{with} \quad y(0) = 3.$$

Note that here f does not explicitly depend on t (the ODE is called *autonomous*), but does implicitly through $y = y(t)$. To apply Euler's method we start with the initial condition and select a step size h. Since we are constructing arrays t and y without dimensionalizing them first it is best to clear these names in case they have been used already in the same MATLAB work session.

```
>> clear t y  % no comma between t and y! type help clear for more info
>> y(1)=3; t(1)=0; h=0.1;
```

Since f is simple enough we may use the `inline` syntax:

```
>> f=inline('2*y','t','y')
f =
   Inline function:
   f(t,y) = 2*y
```

Note that the initialization `y(1)=3` should not be interpreted as "the value of y at 1 is 3", but rather "the first value in the array y is 3". In other words the `1` in `y(1)` is an index, not a time value! Unfortunately, MATLAB indices in arrays must be positive (a legacy from FORTRAN...). The successive approximations at increasing values of t are then obtained as follows:

```
>> y(2)=y(1)+h*f(t(1),y(1)), t(2)=t(1)+h,
y =
  3.0000    3.6000
t =
       0    0.1000
>> y(3)=y(2)+h*f(t(2),y(2)), t(3)=t(2)+h,
y =
  3.0000    3.6000    4.3200
t =
       0    0.1000    0.2000
```

The arrays `y` and `t` are now 1×3 row arrays. The dimension increases as new values of y and t are computed.
It is obviously better to implement these steps in a `for` loop.

```
y(1)=3; t(1)=0; h = 0.1;
for n = 1:5
   y(n+1)= y(n)+h*f(t(n),y(n));
   t(n+1) = t(n)+h;
end
```

Note that the output in each command has been suppressed (with a ;). The list of computed y values vs t values can be output at the end only by typing:

```
>> [t(:),y(:)]    % same as [t',y'] here
ans =
       0    3.0000
  0.1000    3.6000
  0.2000    4.3200
  0.3000    5.1840
  0.4000    6.2208
  0.5000    7.4650
```

The next step is to write a *function* file that takes in input the function defining the ODE, the time span $[t_0, t_{\text{final}}]$, the initial condition and the number of steps used. Note that, if we are given the number of steps used, N, then the stepsize h can be easily computed using $h = \dfrac{t_{\text{final}} - t_0}{N}$.
The following function implements these ideas.

`euler.m`

```
function [t,y] = euler(f,tspan,y0,N)

% Solves the IVP y' = f(t,y), y(t0) = y0 in the time interval tspan = [t0,tf]
% using Euler's method with N time steps.
% Input:
%  f     = name of inline function or function M-file that evaluates the ODE
%          (if not an inline function,  use: euler(@f,tspan,y0,N))
```

```
%          For a system, the f must be given as column vector.
%  tspan = [t0, tf] where t0 = initial time value and tf = final time value
%  y0    = initial value of the dependent variable. If solving a system,
%          initial conditions must be given as a vector.
%  N     = number of steps used.
% Output:
% t = vector of time values where the solution was computed
% y = vector of computed solution values.

m = length(y0);
t0 = tspan(1);
tf = tspan(2);
h = (tf-t0)/N;              % evaluate the time step size
t = linspace(t0,tf,N+1);    % create the vector of t values
y = zeros(m,N+1);   % allocate memory for the output y
y(:,1) = y0';               % set initial condition
for n=1:N
    y(:,n+1) = y(:,n) + h*f(t(n),y(:,n));   % implement Euler's method
end
t = t'; y = y';    % change t and y from row to column vectors
end
```

Remark: You should notice that the code above is slightly different from the first one we wrote (in particular, note the use of ":" when creating the output `y`). Although the two codes are equivalent in the scalar case, only the second one will work also for systems of Differential Equations.

We can implement the function with, say, $N = 50$ by typing the following commands:

```
>> [t,y] = euler(f,[0,.5],3,50); % use @f if defined in separate function
>> [t,y]
ans =
       0    3.0000
  0.0100    3.0600
  0.0200    3.1212
  0.0300    3.1836
  0.0400    3.2473
  0.0500    3.3122
    :         :
  0.4500    7.3136
  0.4600    7.4598
  0.4700    7.6090
  0.4800    7.7612
  0.4900    7.9164
  0.5000    8.0748
```

An even longer output is obtained for larger values of N. To compare the two approximations with $N = 5$ and $N = 50$ we plot both approximations on the same figure, together with the exact solution $y(t) = 3e^{2t}$. Note that we can no longer call the output simply `[t,y]` because every time we call the function we will lose the output from the previous computations. Thus we will call `[t5,y5]` the output from Euler with $N = 5$ and `[t50,y50]` the output with $N = 50$. The exact solution of the IVP is $y = 3e^{2t}$. We will store the exact solution in the vector `y`.

```
>> [t5,y5] = euler(f,[0,.5],3,5);  % use @f if defined in separate function
>> [t50,y50] = euler(f,[0,.5],3,50);
>> t = linspace(0,.5,100); y = 3*exp(2*t);    % evaluate the exact solution
```

```
>> plot(t5,y5,'ro-',t50,y50,'bx-',t,y,'k-'); axis tight;
>> legend('Euler N = 5','Euler N = 50','Exact',2);
```

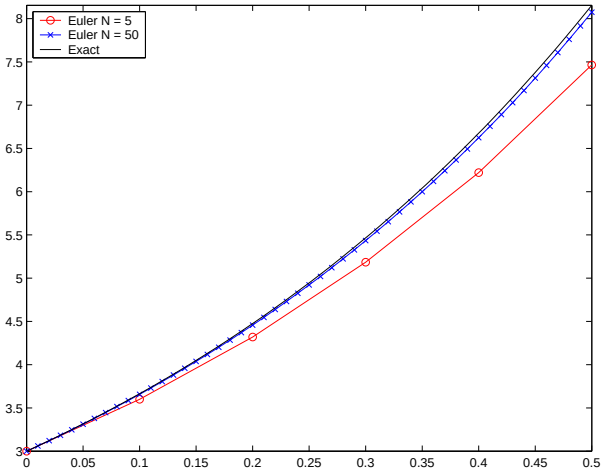


Figure L3a: Euler's method applied to $y' = 2y$, $y(0) = 3$ with step $h = 0.1$ and $h = 0.01$, compared to the exact solution.

IMPORTANT REMARK
When using 5 steps of size $h = 0.1$ the approximation of the exact value $y(0.5) = 3e \simeq 8.1548$ is stored in `y5(6)`. On the other hand when using 50 intervals of size $h = 0.01$, it is stored in `y50(51)`. To avoid confusion we can reference these values by `y5(end)` and `y50(end)`, respectively. The exact value of the function at $t = 0.5$ can be retrieved using `y(end)`.

We can find the error (exact - approximation) at $t = 0.5$ by entering

```
>> e5 = y(end) - y5(end)        % error with N = 5
e5 =
    0.6899
>> e50 = y(end) - y50(end)      % error with N = 50
e50 =
    0.0801
```

and we can calculate the ratio of the errors to see how the approximation improves when the number of steps increases by a factor 10

```
>> ratio = e5/e50
ratio =
    8.6148
```

EXERCISES

1. (a) If you haven't already done so, enter the following commands:

   ```
   >> f=inline('2*y','t','y');
   >> t=linspace(0,.5,100); y=3*exp(2*t);  % defines the exact solution of the ODE
   >> [t50,y50]=euler(f,[0,.5],3, 50); % solves the ODE using Euler with 50 steps
   ```

   Determine the Euler's approximation for $N = 500$ and $N = 5000$ and fill in the following table with the values of the approximations, errors and ratios of consecutive errors at $t = 0.5$. Some of the values have already been entered based on the computations we did above. Include the table in your report, as well as the MATLAB commands used to find the entries.

N	**approximation**	**error**	**ratio**
5	7.4650	0.6899	
50		0.0801	8.6148
500			
5000			

(b) Examine the last column. How does the ratio of consecutive errors relate to the number of steps used? Your answer to this question should confirm the fact that Euler's method is of " order h ", that is, every time the stepsize is decreased by a factor k, the error is also reduced (approximately) by the same factor.

(c) Recall the geometrical interpretation of Euler's method based on the tangent line. Using this geometrical interpretation, can you explain why the Euler approximations underestimate the solution in this particular example?

2. Consider the IVP

$$y' = -2y,\ y(0) = 3$$

for $0 \leq t \leq 10$.
The exact solution of the ODE is $y = 3e^{-2t}$.
The goal of this exercise is to visualize how Euler's method is related to the slope field of the differential equation. In order to do this we will plot the direction field together with the approximations and the exact solution.

(a) To plot the slope field we will use the MATLAB commands `meshgrid` and `quiver`. Enter the following commands:

```
t = 0:.45:10; y = -30:6:42 ;   % define grid of values in t and y direction
[T,Y]=meshgrid(t,y);           % creates 2d matrices of points in the ty-plane
dT = ones(size(T));            % dt=1 for all points
dY = -2*Y;                     % dy = -2*y; this is the ODE
quiver(T,Y,dT,dY)              % draw arrows (t,y)->(t+dt, t+dy)
axis tight                     % adjust look
hold on
```

After running these commands you should get the graph of the slope field.

(b) Use `linspace` to generate a vector of 200 t-values between 0 and 10. Evaluate the solution y at these t-values and plot it in black together with the direction field (use `'linewidth',2`).

(c) Enter the function defining the ODE as `inline`.
Use `euler.m` with $N = 8$ to determine the approximation to the solution. Plot the approximated points in red (use `'ro-','linewidth',2` as line-style in your plot), together with the exact solution and the direction field.

You should get Figure L3b. Based on the slope field and the geometrical meaning of Euler's method explain why the approximations are so inaccurate for this particular value of the stepsize.

(d) Open a new figure by typing `figure`. Plot again the direction field but in a different window: `t = 0:.4:10; y = -1:0.4:3;` Repeat part (b) and repeat part (c) but this time with $N = 16$.
You should get Figure L3c. Comment on the result from a geometrical point of view.

Note: Brief explanations of the commands `quiver` and `meshgrid` are included in Appendix A. In Appendix B we describe the Graphical User Interface `dfield8` for plotting slope fields.

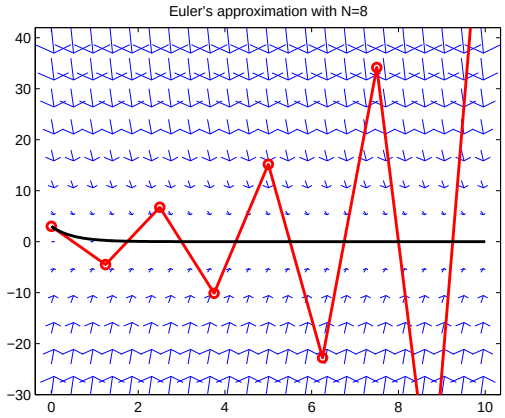


Figure L3b: Euler's method applied to $y' = -2y$, $y(0) = 3$ $N = 8$, compared to the exact solution.

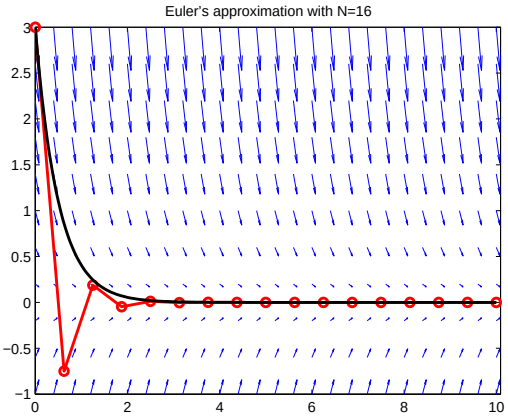


Figure L3c: Euler's method applied to $y' = -2y$, $y(0) = 3$ $N = 16$, compared to the exact solution.

Improved Euler's Method

The improved Euler's method is obtained by using an average of f values, i.e., a slope average:

$$y(t_1) \simeq y(t_0) + \frac{h}{2}\left(f(t_0, y(t_0)) + f(t_1, y(t_1))\right) \tag{L3.2}$$

Then substitute $y(t_0) = y_0$ and define $y_1 \simeq y(t_1)$:

$$y_1 = y_0 + \frac{h}{2}\left(f(t_0, y_0) + f(t_1, y_1)\right). \tag{L3.3}$$

The equation (L3.3) defines the *trapezoidal* method. Unfortunately, this formula defines y_1 only implicitly, i.e., y_1 appears on both sides of the equality so that an equation must be solved to obtain y_1. To avoid this problem, and since we already have made an approximation to get (L3.3), we replace y_1 on the right-hand side by the approximation one would obtain by simply applying Euler's method from (t_0, y_0). The resulting quantity

$$y_1 = y_0 + \frac{h}{2}\Big(f(t_0, y_0) + f(t_1, \underbrace{y_0 + hf(t_0, y_0)}_{\text{Euler } y_1 \text{ from (L3.1)}})\Big) \tag{L3.4}$$

with $t_1 = t_0 + h$ is the *improved Euler* approximation. This approximation can be thought of as a correction to the Euler approximation. The iteration (L3.4) is then repeated to obtain $y_2 \simeq y(t_2), \ldots$, i.e.,

$$\begin{array}{rcl} f_1 & = & f(t_n, y_n) \\ f_2 & = & f(t_n + h, y_n + hf_1) \\ y_{n+1} & = & y_n + \frac{h}{2}(f_1 + f_2) \\ t_{n+1} & = & t_n + h \end{array}$$

Minor modifications are made to the function `euler.m` to implement the improved Euler method.

EXERCISES

3. Modify the M-file `euler.m` to implement the algorithm for Improved Euler. Call the new file `impeuler.m` (include the file in your report).
   Test your code for the ODE $y' = 2y$, $y(0) = 3$.
   First enter the function $f(t, y) = 2y$ as `inline` function and then enter the following:

```
>> [t5,y5] = impeuler(f,[0,.5],3,5); % use @f if defined in separate function
>> [t5,y5]
ans =
       0    3.0000
  0.1000    3.6600
  0.2000    4.4652
  0.3000    5.4475
  0.4000    6.6460
  0.5000    8.1081
```

   Compare your output with the one above.
   Note that the improved Euler approximation with 5 steps is already more accurate than the Euler approximation with 50 steps! (hence the "improved")

4. Consider the IVP $y' = 2y$, $y(0) = 3 \quad 0 \leq t \leq 0.5$

   (a) Determine the Improved Euler's approximation for $N = 50$, $N = 500$ and $N = 5000$. Fill in the following table with the values of the approximations, errors and ratios of consecutive errors at $t = 0.5$. One of the values has already been entered based on the computations we did above. Recall that the exact solution to the ODE is $y = 3e^{2t}$.
   Include the table in your report, as well as the MATLAB commands used to find the entries.

N	**approximation**	**error**	**ratio**
5	8.1081		
50			
500			
5000			

   (b) Examine the last column. How does the ratio of the errors relate to the number of steps used? Your answer to this question should confirm the fact that Improved Euler's method is of " order h^2 ", that is, every time the stepsize is decreased by a factor k, the error is reduced (approximately) by a factor of k^2.

5. Repeat Problem 2 for Improved Euler. Compare the results with the ones obtained with Euler's method.

APPENDIX A: The commands `meshgrid` and `quiver`

The function `meshgrid`: This command is especially important because it is used also for plotting 3D surfaces. Suppose we want to plot a 3D function $z = x^2 - y^2$ over the domain $0 \le x \le 4$ and $0 \le y \le 4$. To do so, we first take several points on the domain, say 25 points. We can create two matrices X and Y, each of size 5×5, and write the xy- coordinates of each point in these matrices. We can then evaluate z at these xy values and plot the resulting surface. Creating the two matrices X and Y is much easier with the `meshgrid` command:

```
x = 0:4;          % create a vector x = [0, 1, 2, 3, 4]
y = -4:2:4;       % create a vector y = [-4, -2, 0, 2, 4]
[X, Y] = meshgrid(x,y); % create a grid of 25 points and store
                        % - their coordinates in X and Y
```

Try entering `plot(X,Y,'o','linewidth',2)` to see a picture of the 25 points.
Then, just for fun try entering

```
Z = X.^2 - Y.^2;
surf(X,Y,Z);
```

to get a picture of the surface representing $z = x^2 - y^2$. (The graph will be quite rough because of the limited number of points, but you should nonetheless recognize a "saddle").
Thus, in our code for the direction field, the `meshgrid` command is used to generate the points in the xy plane at which we want to draw the arrows representing the slope or tangent line to the solution at the point. Type `help meshgrid` for more information.

The function quiver: The command `quiver(X,Y,U,V)` draws vectors (arrows) specified by U and V at the points specified by X and Y. In our examples the vectors drawn are `(dT, dY)` where `dT =1` and `dY = -2*y`, thus these vectors are in the direction of the slope at the given point. The arrows are automatically scaled so as not to overlap. The matrices X and Y are built using the meshgrid command as explained above. Type `help quiver` for more explanation.

APPENDIX B: Plotting direction fields with `dfield8`

The Graphical User Interface `dfield8` from J. Polking (available at `http://math.rice.edu/~dfield`) can also be used to plot the direction field and selected solution curves. After downloading the file, enter `dfield8` in the command window.
A new window will pop up, one that looks like Figure L3d

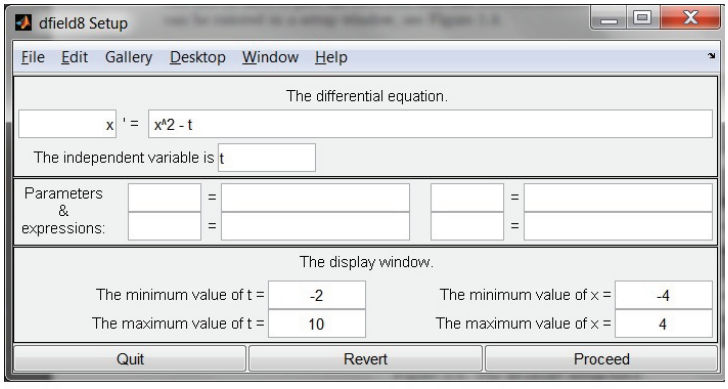
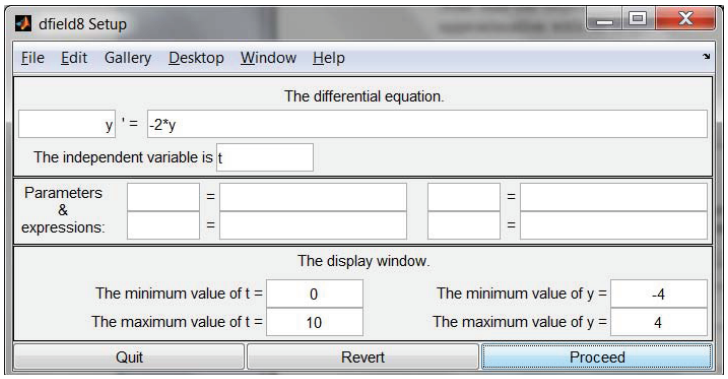


Figure L3d: The `dfield8` setup GUI

ODEs with up to two parameters can be entered in the Setup window. For instance, we can enter the ODE $y' = -2y$, identify the independent variable as t and choose the display window's dimension ($0 \le t \le 10$ and $-4 \le y \le 4$).



Click on *Proceed* to see the slope field. A sample display window is shown in Figure L3e.

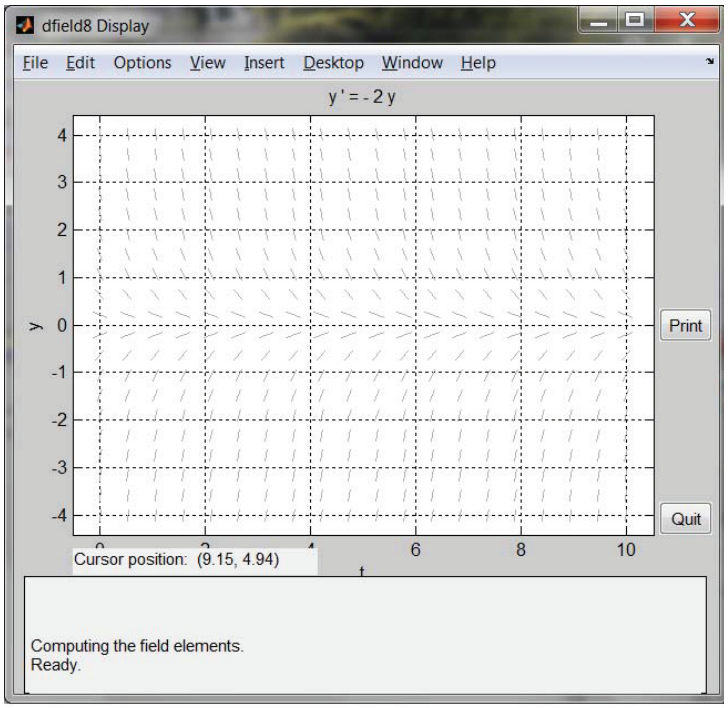


Figure L3e: The `dfield8` direction field plot

Solutions curves can be drawn by clicking on the display window. Specific initial conditions can be entered by clicking on **Options ->Keyboard Input**. A new window will arise prompting the user to input initial values for the independent and dependent variables as an initial condition for the IVP.

More information about various options are available at `http://math.rice.edu/~dfield`.