Contents

---

1 Introduction of Methods

Most engineering problems can be modelled by ordinary differential equations (ODEs), and there exist several numerical techniques to approximate the solution of these ODEs. Some of these methods are going to be discussed in this report namely, Forward Euler, Heun and Runge-Kutta 4th order and their implementation to a 2nd order ODE.

1.1 Forward Euler Method

The basic idea of the Euler method is to approximate the derivative in the current position with an incremental quotient, and below is the numerical scheme of the method after negelecting the truncation error;

$$Y_{i+1} = Y_i + \Delta x f(x_i,\ Y_i) \tag{1}$$

1.2 Heun Method

The method is sometimes called the Runge-Kutta 2nd order method, and the idea behind it is to use the trapezoidal rule to approximate the integral below;

$$Y_{i+1} = Y_i + \int_{x_i}^{x_{i+1}} f(x, y(x))\ dx \tag{2}$$

resulting in an implicit equation shown below;

$$Y_{i+1} = Y_i + \frac{\Delta x}{2}[f(x_i,\ Y_i) + f(x_{i+1},\ Y_{i+1})] \tag{3}$$

After some manipulation the resulting explicit equation can be written as;

$$\begin{aligned} Y^*_{i+1} &= Y_i + \Delta x f(x_i,\ Y_i) \\ Y_{i+1} &= Y_i + \tfrac{\Delta x}{2}[f(x_i,\ Y_i) + f(x_{i+1},\ Y^*_{i+1})] \end{aligned} \tag{4}$$

1.3 Runge-Kutta 4th Order Method

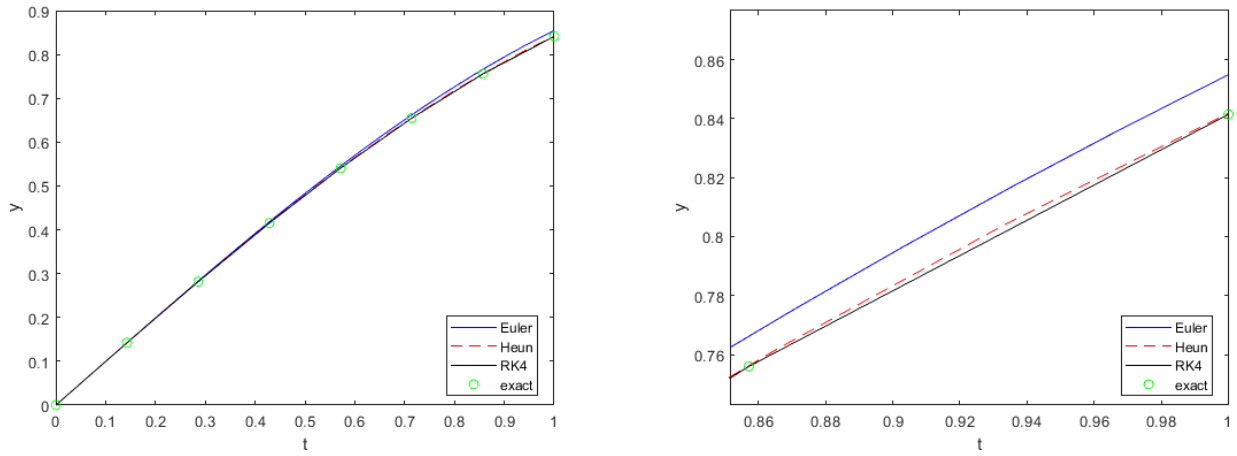
Like the second order method the 4th order method is based on approximating the integral in equation 2 by numerical quadrature only of higher degree in this case, and below is the numerical scheme of the method;

$$\begin{aligned} Y_{i+1} &= Y_i + \tfrac{\Delta x}{6}[k_1 + 2k_2 + 2k_3 + k4] \\ k_1 &= f(x_i,\ Y_i) \\ k_2 &= f(x_i + \tfrac{\Delta x}{2},\ Y_i + \tfrac{\Delta x}{2}\ k_1) \\ k_3 &= f(x_i + \tfrac{\Delta x}{2},\ Y_i + \tfrac{\Delta x}{2}\ k_2) \\ k_4 &= f(x_i + \Delta x,\ Y_i + \Delta x\ k_3) \end{aligned} \tag{5}$$

2 Comparison of Methods

In order to assess which of the explained methods in section 1 is performing better, it is compared the error along step size against the analytical solution. The below figure 2 compares for the same computational effort the Euler's method, the Heun's method and Runge-Kutta's 4th-Order method.

By "same computational effort" it is understood as the same number of function evaluations in the whole interval. Since each of methods implements different number of function evaluations per step, it is necessary to establish different step sizes per each method to have a comparable computational effort.
For instance, Runge-Kutta's method evaluates 4 times the function per each step discretization, while Euler's method only evaluates the function one time per each discretazed step. Likewise, Heun's method evaluates the function 2 times per each step.



All evaluated domain $x \in [0, 1]$ Detailed view

Figure 1: Plot of approximated $y^*(x)$ and analytical solution $y(x)$

On the plotting of figure 2, x has been discretazed differently for each methods to ensure same computational effort, as follows:

- RK 4th-Order method $x \in [0, 1] \rightarrow$ **8 discretizations**.
- Heun method $x \in [0, 1] \rightarrow$ **16 discretizations**.
- Euler method $x \in [0, 1] \rightarrow$ **32 discretizations**.

3 Convergence of Methods

The convergence of each method has been evaluated. For a given point ($x = 1$), the logarithmic error has been measured for different Δx discretizations. From large Δx to small ones.
Note that since all the Δx are less than 1, the $log(\Delta x)$ will be always a negative value. It is necessary to represent the absolute $|log(\Delta x)|$ as values for the horizontal axis.

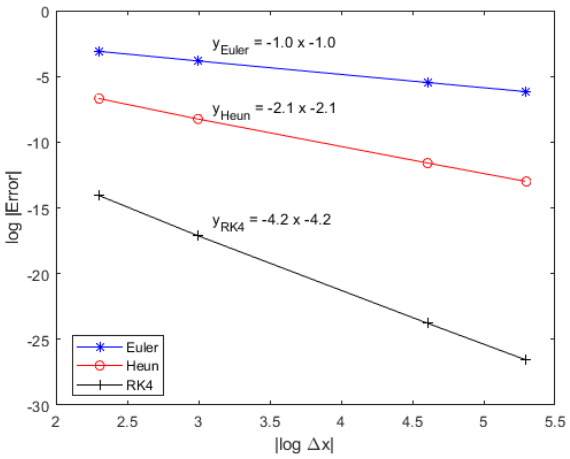


Figure 2: Convergence of the error

All the methods tend to convergence as it is decreased the Δx. The bigger the slope the more faster the method converges. Thus, the Runge-Kutta method converges faster than the other two methods with an slope factor of -4 (4th order convergence). Euler method convergences with -1 factor, and Heun's methods with a factor of -2.

4 ODE45 function

The ODE45 function in Matlab implements the Runge-Kutta 4th and 5th order method with a variable time step for efficient computation. This method produces results with global and local errors of order 4th and 5th respectively, so the method is quite accurate. To improve the accuracy of the solution obtained by this function, the default relative error tolerance in the **option** part has to be changed to the desired error tolerance which corresponds with the desired accuracy. Below is the is a graph of the obtained results with ODE45 compared with the exact solution;

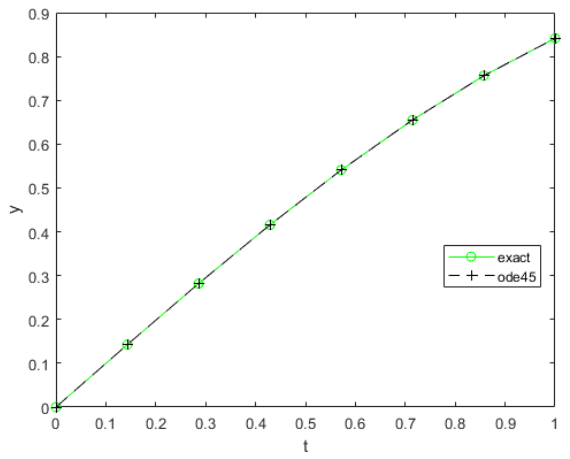


Figure 3: Comparison of ODE45 with the exact solution

To improve the **accuracy** of the solution, *MATLAB* has another parameter called "options" in the *ode45* function, where we can modify the **relative error tolerance** which corresponds with the desired accuracy. Here is the part that can be changed:

```
options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)
```

5 Conclusions

After assessing all approximation ODE's methods, several conclusions arises. Regarding the **error performance** of the methods for the same computational effort:

- *Euler method* for the same computational effort in comparison to others methods, always performs worse. Furthermore, the error gap is several orders of magnitude higher than the gap error between Heun's and Runge-Kutta 4th-order methods.
- *Runge-Kutta 4th-order method* is the one that performs better. Nevertheless, for the studied function, the error is very close to the Heun's method.
- The *Heun's method* performs almost as well as Runge-kutta 4th-Order methods on the studied ODE.

Regarding the **convergence**, the Runge-Kutta method is the best, it converges 4 times faster than the Forward Euler method and 2 times faster than the Heun method. At the same time, the Heun method converges 2 times faster than the Forward Euler method.

A MATLAB Code

Here it is presented the *MATLAB* codes:

Euler method:

Listing 1: Matlab script

```
%-------------------------------------------------------------------------
% IMPLEMENTATION OF THE EULER METHOD
%-------------------------------------------------------------------------
function y = EulerM(y_0, z_0)

global tEu

y = zeros(length(tEu),1);
z = zeros(length(tEu),1);

y(1) = y_0;
z(1) = z_0;

format long

for n = 1:length(tEu)-1
    dt = tEu(n+1) - tEu(n);
    z(n+1) = z(n) - dt*y(n);
    y(n+1) = y(n) + dt*z(n);
end
end
```

Heun method:

Listing 2: Matlab script

```
%-------------------------------------------------------------------------
% IMPLEMENTATION OF THE HEUN METHOD
%-------------------------------------------------------------------------
function y = Heun(y_0, z_0)

f = @(y, z) z;
g = @(y, z) -y;

global tHeu

y = zeros(length(tHeu),1);
z = zeros(length(tHeu),1);

y(1) = y_0;
z(1) = z_0;

for n = 1:length(tHeu) - 1
    dt = tHeu(n+1) - tHeu(n);
    z_star = z(n) + dt*g(y(n), z(n));
    y_star = y(n) + dt*f(y(n), z(n));
    z(n+1) = z(n) + dt/2*(g(y(n), z(n)) + g(y_star, z_star));
    y(n+1) = y(n) + dt/2*(f(y(n),z(n)) + f(y_star, z_star));
```

```
end
end
```

Runge-Kutta 4th-Order method:

Listing 3: Matlab script

```
%-------------------------------------------------------------------------
% IMPLEMENTATION OF THE 4TH ORDER RUNGE KUTTA METHOD
%-------------------------------------------------------------------------
function y = RK4(y_0, z_0)

global tRK4

f = @(y, z) z;
g = @(y, z) -y;

y = zeros(length(tRK4),1);
z = zeros(length(tRK4),1);

y(1) = y_0;
z(1) = z_0;

for n = 1:length(tRK4)-1
    dt = tRK4(n+1) - tRK4(n);
    k1 = f(y(n), z(n));
    l1 = g(y(n), z(n));
    k2 = f(y(n) + dt/2*k1, z(n) + dt/2*l1);
    l2 = g(y(n) + dt/2*k1, z(n) + dt/2*l1);
    k3 = f(y(n) + dt/2*k2, z(n) + dt/2*l2);
    l3 = g(y(n) + dt/2*k2, z(n) + dt/2*l2);
    k4 = f(y(n) + dt*k3, z(n) + dt*l3);
    l4 = g(y(n) + dt*k3, z(n) + dt*l3);
    y(n+1) = y(n) + dt/6*(k1 + 2*k2 + 2*k3 + k4);
    z(n+1) = z(n) + dt/6*(l1 + 2*l2 + 2*l3 + l4);
end
end

%-------------------------------------------------------------------------
% FUNCTION DEFINITION USED IN ODE45
%-------------------------------------------------------------------------
function yprime = ODE_2(~, y)

yprime = [y(2); -y(1)];
end
```

Main Matlab execution code:

Listing 4: Matlab script

```
%-------------------------------------------------------------------------
% DEFINING THE INITIAL CONDITIONS
%-------------------------------------------------------------------------
clear
clc
y_0 = 0;
z_0 = 1;
global tRK4 tEu tHeu
tRK4 = linspace(0,1,8);
tHeu = linspace(0,1,16);
tEu = linspace(0,1,32);

%-------------------------------------------------------------------------
% CALLING THE FUNCTIONS TO SOLVE THE PROBLEM USING GIVEN INITIAL CONDITIONS
%-------------------------------------------------------------------------
Y_Euler = EulerM(y_0, z_0);
Y_Heun  = Heun(y_0, z_0);
Y_RK4   = RK4(y_0, z_0);
[x, y]  = ode45(@ODE_2, tRK4, [0;1]);
r = [Y_Euler(end) Y_Heun(end) Y_RK4(end)];
%-------------------------------------------------------------------------
% PLOTTING THE RESULTS
%-------------------------------------------------------------------------
plot(tEu,Y_Euler,'b-')
hold on
plot(tHeu,Y_Heun,'r--')
plot(tRK4, Y_RK4,'k-')
plot(tRK4, sin(tRK4),'go')
%plot(tRK4, y(:,1),'k+')
% legend('Euler','Heun','RK4','exact','ode45', 'location','best')
legend('Euler','Heun','RK4','exact', 'location','best')
xlabel('x')
ylabel('y')
hold off
```