Numerical Methods

Ex. 10. Approximate Solving of Ordinary Differential Equations.

Kamila Kwiecińska, Karol Latos

Group **2**, team **2**, 22.05.2020

1 Introduction

Differential equations are the mathematical backbone of myriad of useful models not only in physics, chemistry and biology, but also in economics, social sciences and more. However, solving them is not always a straightforward task; moreover, in the age of computers, numerical methods are needed for acquiring solutions to differential-type problems. Some of these methods were implemented in Octave language in order to solve following differential equations:

$$y^{(1)}(x) = -5.8y(x) + 4.7x - 0.9x^2, \quad y(x_0 = 0) = 1.9, \quad 0 \leq x \leq 9. \tag{1}$$

$$\begin{aligned} y^{(2)}(x) + 2y^{(1)}(x) = 30\cos(x), \quad y(x_0 = 0) = 8, \\ y^{(1)}(x_0 = 0) = 11, \quad 0 \leq x \leq 10. \end{aligned} \tag{2}$$

1.1 Requirements for the report

1. Solve Equation (1) using Taylor series method. Restrict the solution to three terms of the Taylor series. Present this solution on a graph. Discuss the results.
2. Use your program for Euler method to solve Equation (1) numerically. Present the obtained solutions in the form of a single graph for integration steps $h \in \{0.01, 0.1, 1.0\}$. Discuss the results.
3. Use your program for fourth order Runge-Kutta method to solve Equation (1) numerically. Present the obtained solutions in the form of a single graph for integration steps $h \in \{0.01, 0.1, 1.0\}$. Discuss the results.
4. Explain the differences between the solutions obtained in 1., 2. and 3.

5. Use your program for fourth order Runge-Kutta method to solve Equation (2) numerically. Present the obtained solutions in the form of a single graph for integration steps $h \in \{0.01, 0.1, 1.0\}$. Discuss the results. Plot a separate graph, that will present the relation between y_i (X axis) and $y_i^{(1)}$ (Y axis), for each integration step h value.

6. Estimate, discuss, and compare errors of each numerical method you implemented.

7. Write down your observations and conclusions from the exercise.

8. Include all source codes.

2 Solving Equation (1) using three methods

2.1 Taylor series method

To obtain three terms of the Taylor series for $y(x)$, two additional derivatives ought to be calculated:

$$y^{(2)}(x) = -5.8y^{(1)}(x) + 4.7 - 1.8x$$
$$y^{(3)}(x) = -5.8y^{(2)}(x) - 1.8$$

From abbreviated Taylor expansion it holds that

$$y(x) \approx y(x_0) + y^{(1)}(x_0)(x - x_0) + \frac{y^{(2)}(x_0)}{2}(x - x_0)^2 + \frac{y^{(3)}(x_0)}{6}(x - x_0)^3, \quad (3)$$

using $x_0 = 0$ from the initial condition and substituting consecutive values gives

$$y(x_0) = 1.9, \qquad y^{(1)}(x_0) = -11.02,$$
$$y^{(2)}(x_0) = 68.616, \qquad y^{(3)}(x_0) = -399.7728.$$

Now equation (3) is used to finally obtain

$$y(x) \approx -66.6288x^3 + 34.308x^2 - 11.02x + 1.9 \quad (4)$$

It is nothing unusual for Taylor series polynomials approximating non-polynomial functions to quickly diverge from the approximated function, especially when the number of terms is low. That's why we see values around negative 45000 at $x = 9$ (Fig. 1). Taking a closer look at the interval near x_0, one can observe closeness to the approximated function (Fig. 2). One cannot help but wonder, what was the purpose of that exercise in relation to the report, except for proving that the student can calculate simple derivatives and plug the number 0 as argument of functions.

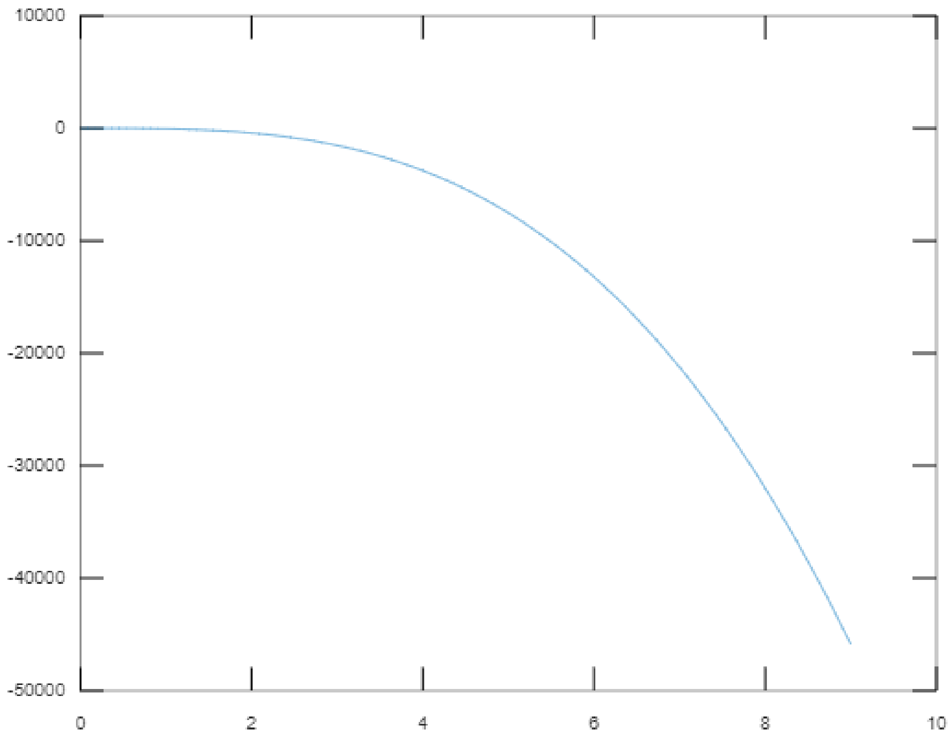

Figura 1: Taylor series polynomial approximating y

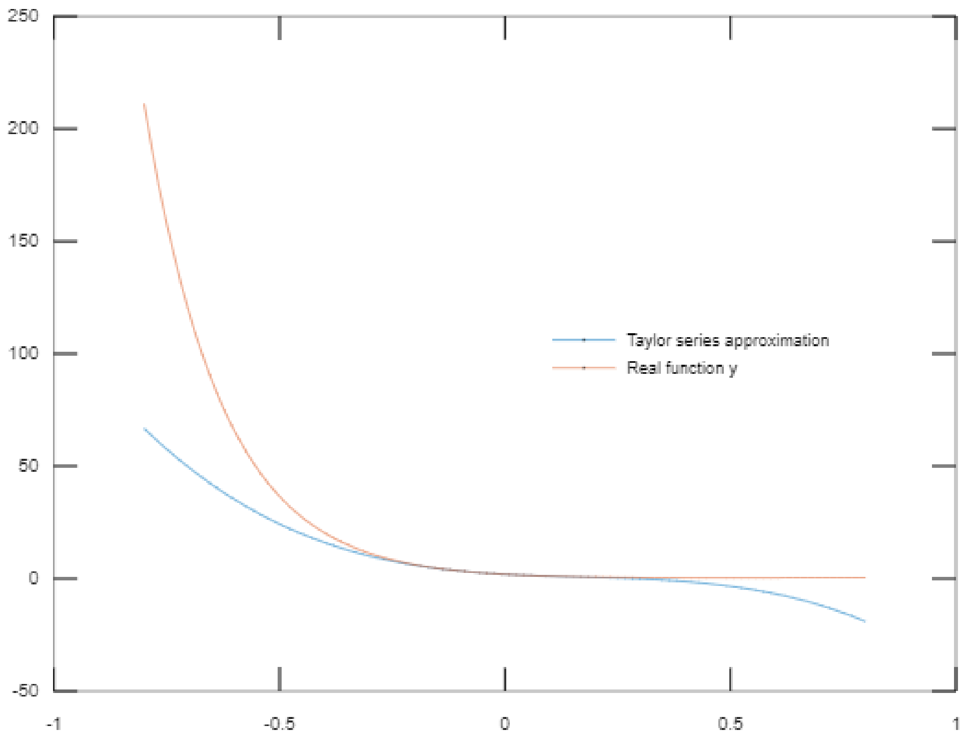

Figura 2: Taylor series polynomial approximating y - interval around x_0

2.2 Euler-Cauchy method

Following graphs show the approximation of y function being a solution to Equation (1). Additional graph without step $h = 1$ has been included, as step values above 0.7 yield unexpected results with very high values, rising in an exponential fashion.

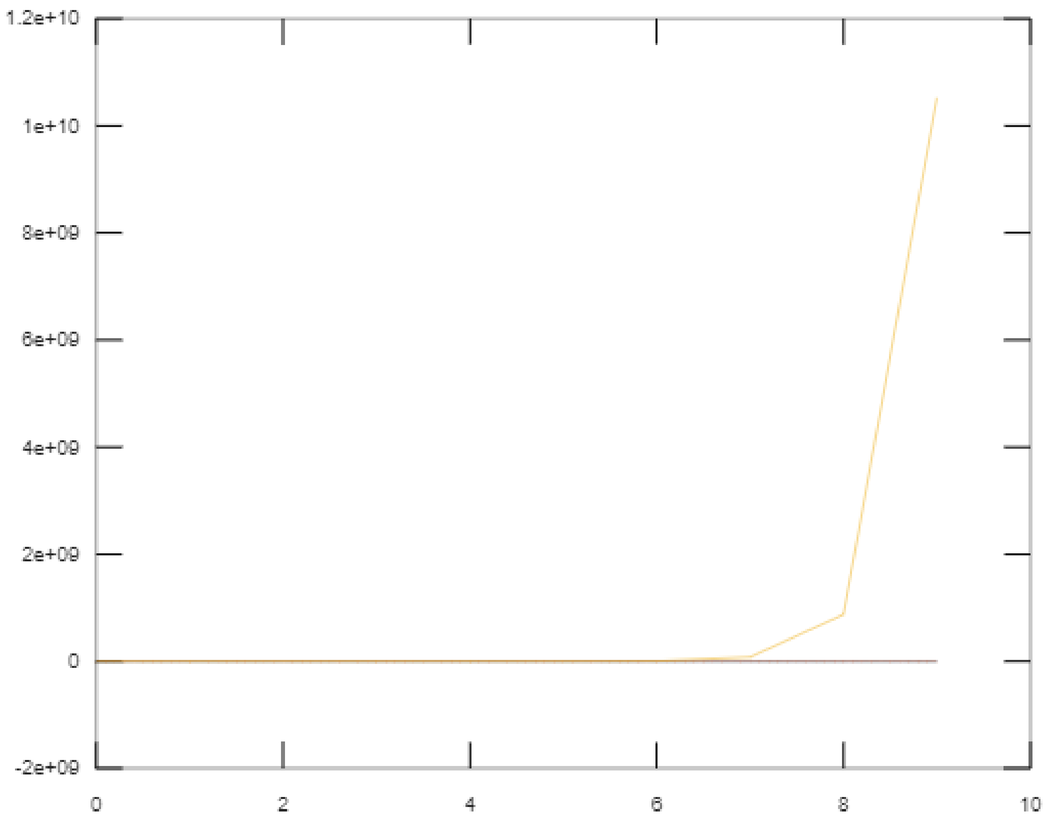

Figura 3: Euler-Cauchy method with h $= 0.01, 0.1$ and 1

It can be seen that, despite some little difference around the local minimum, there is not much discrepancy between step 0.1 and 0.01, as both approximate the y function closely (Fig. 4).

2.3 Runge-Kutta method

Following graphs show the approximation of y function being a solution to Equation (1). Additional graph without step $h = 1$ has been included, again, as step values above 0.7 yield unexpected results.

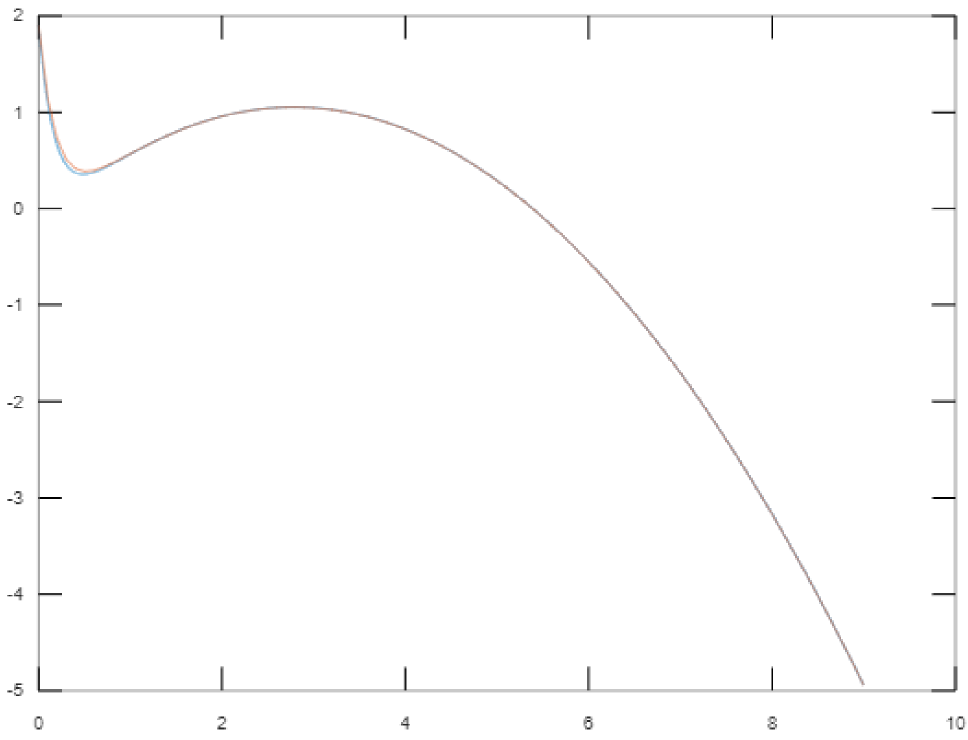

Figura 4: Euler-Cauchy method with h = 0.01 and 0.1

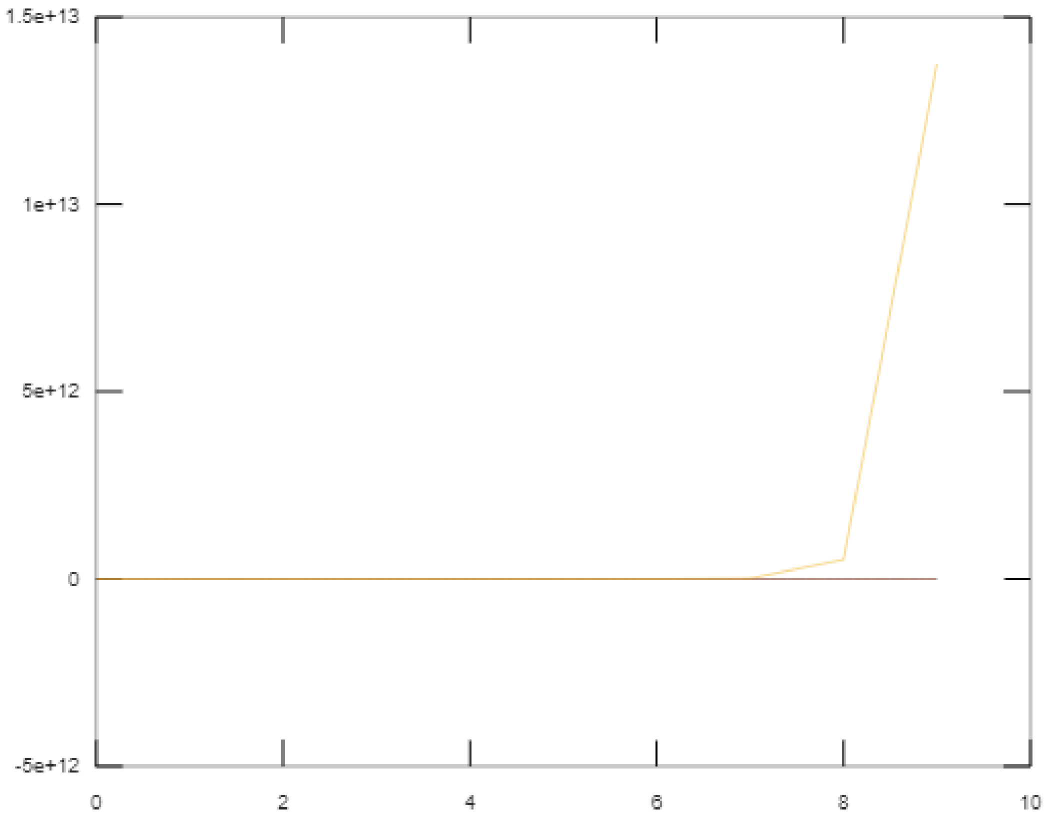

Figura 5: Runge-Kutta method with $h = 0.01, 0.1$ and 1

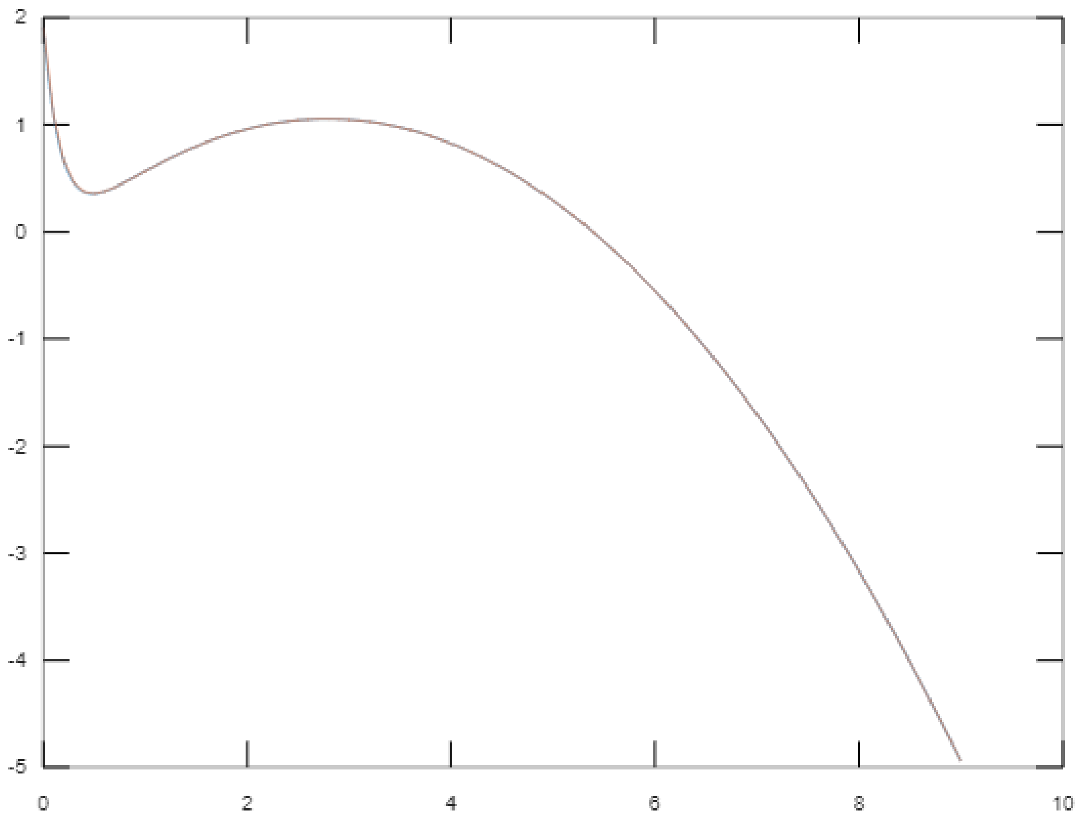

Figura 6: Runge-Kutta method with $h = 0.01$ and 0.1

It can be seen, again, that there is not much difference between step 0.1 and 0.01, as both approximate the y function closely.

2.4 Results commentary

Both Euler-Cauchy and Runge-Kutta methods were easy to implement and are not very resource- nor time-consuming. At first glance the noticeable differences between these two methods are the number of lines of code used in the main iteration loop (3 and 6, respectively) and the slight separation of approximation curves in Euler-Cauchy, which may signify lower accuracy. Thus, the reader is advised to try both methods on their own, as so far most of the conclusions concerning the differences between the methods have appeared as feelings and therefore cannot be easily transcribed.

3 Solving Equation (2) using Runge-Kutta method

The second order differential equation (2) can be rewritten as presented:

$$y^{(2)}(x) = -2y^{(1)}(x) + 30\cos(x) \tag{5}$$

Using a simple substitution, a system of first order differential equations is obtained.

$$\begin{cases} y^{(1)}(x) = z(x) \\ z^{(1)}(x) = -2y^{(1)}(x) + 30\cos(x) \end{cases} \tag{6}$$

Solving this system using fourth order Runge-Kutta method yields accurate results, with better and better approximation of the y function, as h gets smaller and smaller (Fig 7).

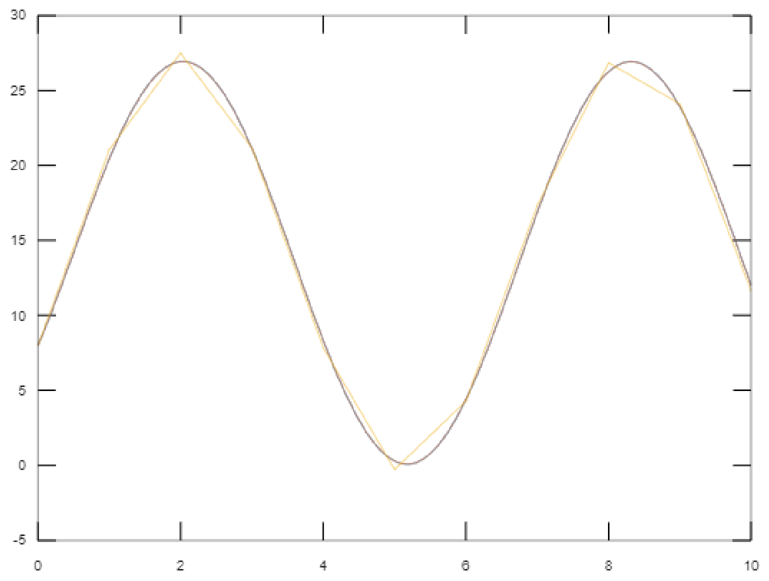

Figura 7: Runge-Kutta method for Eq. (2) with $h = 0.01, 0.1$ and 1

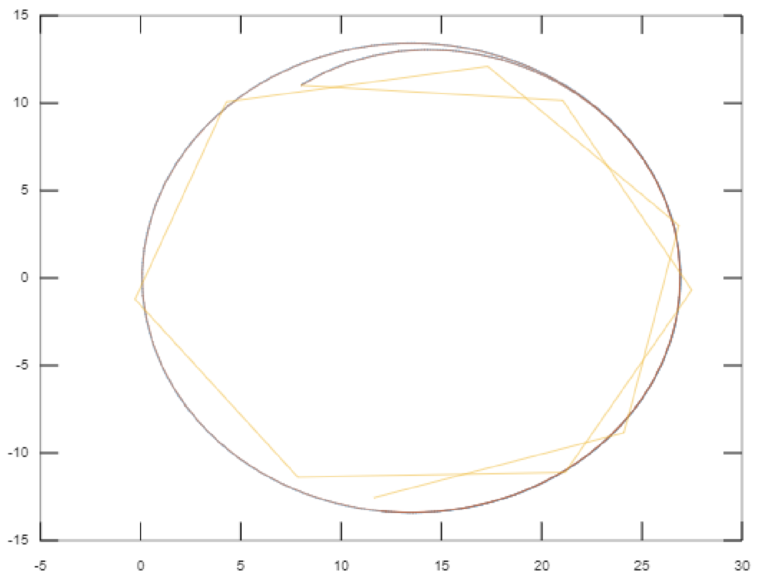

Figura 8: Relationship between y (X axis) and $y^{(1)}$ (Y axis)

4 Errors analysis

The error after n iterations of Euler method can be bound from above by $Mh\frac{e^{Lhn}-1}{2L}$. To perform more useful analysis, original y function is calculated:

$$y(x) = -0.155169x^2 + 0.86383x - 0.14893 + \frac{2.04893}{e^{5.8x}} \tag{7}$$

Now, for each approximating function A, new function is created according to the pattern.

$$m(A, y) = |A(x) - y(x)| \tag{8}$$

Using any numerical integration method, the differences between the approximating and approximated functions are summed up.

$$M(A, y) = \int_{x_0}^{x_n} m(A, y)dx \tag{9}$$

Therefore, $M(A, y)$ returns the total space in between A and y on $[x_0, x_n]$ interval. It's worth noting, that

$$\lim_{A\to y} M(A, y) = 0 \quad \text{and} \quad \lim_{M(A,y)\to 0} A = y.$$

Comparing Euler-Cauchy and Runge-Kutta methods with this approach, with $0.01 \leq h \leq 0.3$, yields results visible in Fig 9. Since Euler-Cauchy method is unstable (the error is propagated with each iteration), there is no point in considering $h > 0.3$, as the values of errors rise, rendering the Runge-Kutta method errors insignificant.

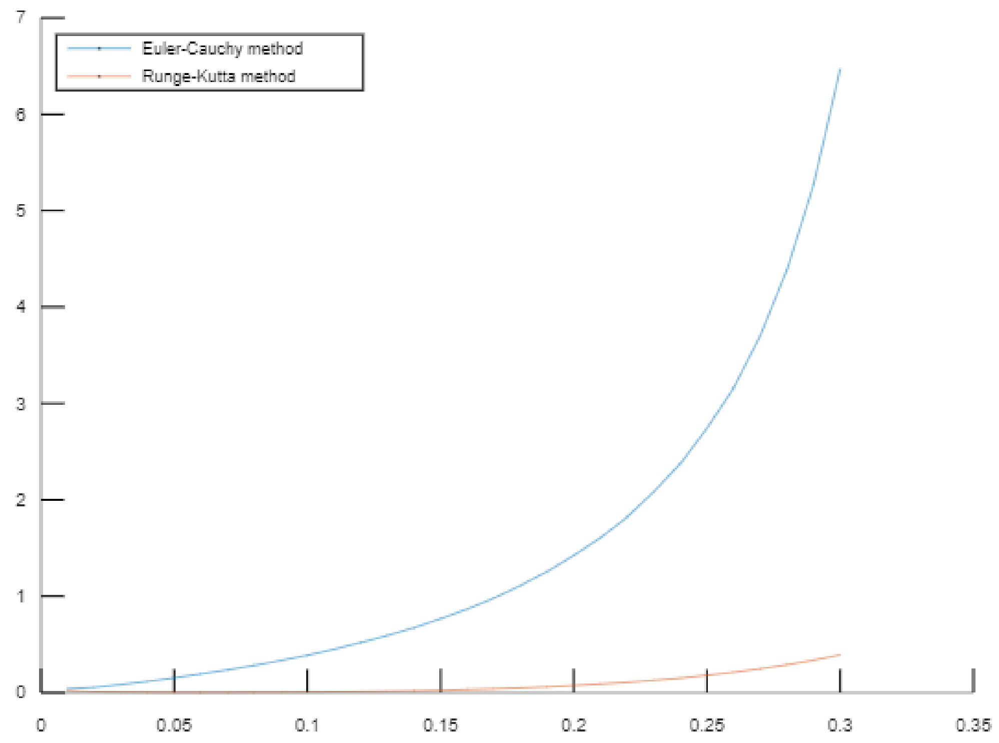

Figura 9: $M(A, y)$ (error) values for E-C and R-K methods

Similarly, y function being a solution to Equation (2) is firstly calculated:

$$y(x) = \frac{1}{2}\left(e^{-2x} + 24\sin(x) - 12\cos(x) + 27\right) \tag{10}$$

Applying function M for approximations of y in relation to the integration step $h \in [0.01, 1]$, gives the following error graph.

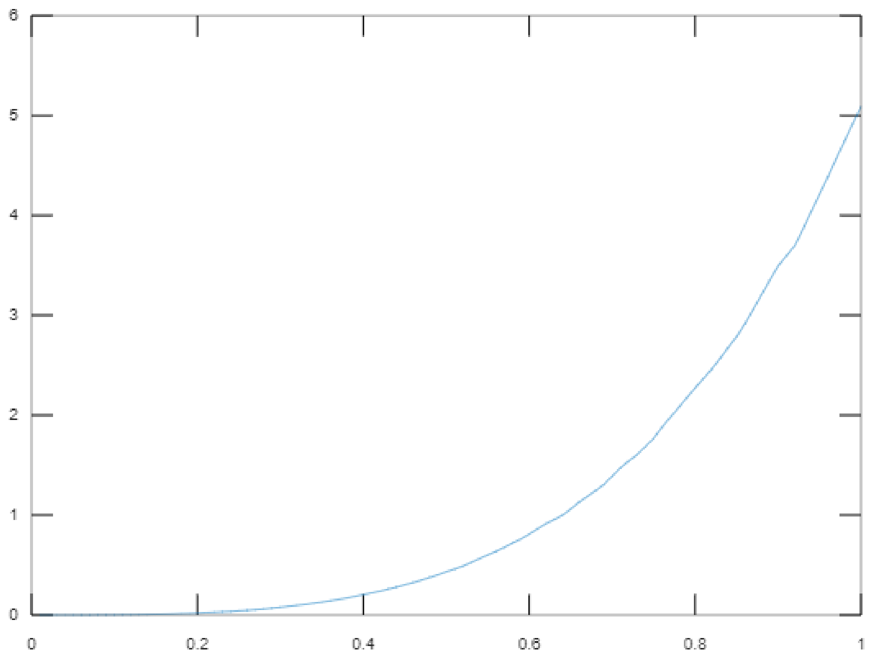

Figura 10: $M(A, y)$ (error) value R-K method in Eq. (2)

The Runge-Kutta method rises far more slowly than Euler-Cauchy, even when used for solving second order differential equations through substitution. We observe the error at $h = 0.2$ is almost as small as at $h = 0$.

5 Conclusions

Implementation of selected methods for solving ordinary differential equations was easy and pleasurable. No obstacles were faced and the approaches resulted in successful approximation of the y functions. The Euler-Cauchy method was the simplest, requiring the least amount of memory, time and lines of code — however, as an unstable method, the error was being propagated through every iteration. This characteristic could be observed in the comparison between the errors of two methods: Euler-Cauchy and Runge-Kutta, in section 4. The latter alogrithm was found to be superior. As comes to solving second order differential equations, the Runge-Kutta method was more than suitable, and combined with converting the equation into a system of first order differential equations, yielded approximations of great accuracy, as could be concluded from error analysis. The exercise, as it is stated in subsection 1.1 and performed in this report, is recommended to anyone in need for a good fun combined with a bit of education.

6 Code

```
# Karol Latos & Kamila Kwiecinska
value = 0.0;

# Coefficients for equations
global a_1 = -5.8;
global b_1 = 4.7;
global c_1 = -0.9;
global d_1 = 1.9;
global e_1 = 9;
global a_2 = 2;
global b_2 = 30;
global c_2 = 8;
global d_2 = 11;
global e_2 = 10;

# Function f, right-hand side of the ODE
function value = f(x, y)
    global a_1 b_1 c_1;
    value = a_1 * y + b_1 * x + c_1 * x^2;
end

# Initial condition as a point satifying y(x_0) = y_0
x_0 = 0;
y_0 = d_1;

# Integration step
h = 0.1;

# Euler-Cauchy method
M = [x_0, y_0];
for i = 1:e_1/h
    M(i + 1, 1) = M(i, 1) + h;
    y_rough = M(i, 2) + h * f(M(i, 1), M(i, 2));
    M(i + 1, 2) = M(i, 2) + 0.5 * h * (f(M(i, 1), M(i, 2)) + f(M(i +
    1, 1), y_rough));
end

printf("Euler-Cauchy method\nStep h = %d\n", h);
printf("Obtained points:\n");
disp(M)
figure(1);
plot(M(:, 1), M(:, 2))

# Runge-Kutta method
M = [x_0, y_0];
for i = 1:e_1/h
    k_1 = h * f(M(i, 1), M(i, 2));
    k_2 = h * f(M(i, 1) + 0.5 * h, M(i, 2) + 0.5 * k_1);
    k_3 = h * f(M(i, 1) + 0.5 * h, M(i, 2) + 0.5 * k_2);
    k_4 = h * f(M(i, 1) + h, M(i, 2) + k_3);
    M(i + 1, 1) = M(i, 1) + h;
    M(i + 1, 2) = M(i,2) + (k_1 + 2*k_2 + 2*k_3 + k_4)/6;
end

```

```
printf("\nRunge-Kutta method\nStep h = %d\n", h);
printf("Obtained points:\n");
disp(M)
figure(2);
plot(M(:, 1), M(:, 2))

# y^(2) = -a_2 * y^(1) + b_2 * cos(x) /\ y^(1) = z(x)
# z^(1) = -a_2 * z(x) + b_2 * cos(x)
function value = f1(x, y, z)
    global a_2 b_2;
    value = -a_2 * z + b_2 * cos(x);
end

function value = f2(x, y, z)
    value = z;
end

# Initial condition points
x_0 = 0;
y_0 = c_2;
y_1 = d_2;

x = x_0:h:e_2;
y = zeros(size(x)(2), 1);
z = zeros(size(x)(2), 1);
y(1) = y_0;
z(1) = y_1;

k_1 = zeros(4, 1);
k_2 = zeros(4, 1);
for i=2:1:size(x)(2)
    k_1(1) = h*f2(x(i-1), y(i-1), z(i-1));
    k_2(1) = h*f1(x(i-1), y(i-1), z(i-1));
    k_1(2) = h*f2(x(i-1) + h/2, y(i-1) + k_1(1)/2, z(i-1) + k_2(1)/
   2);
    k_2(2) = h*f1(x(i-1) + h/2, y(i-1) + k_1(1)/2, z(i-1) + k_2(1)/
   2);
    k_1(3) = h*f2(x(i-1) + h/2, y(i-1) + k_1(2)/2, z(i-1) + k_2(2)/
   2);
    k_2(3) = h*f1(x(i-1) + h/2, y(i-1) + k_1(2)/2, z(i-1) + k_2(2)/
   2);
    k_1(4) = h*f2(x(i-1) + h, y(i-1) + k_1(3), z(i-1) + k_2(3));
    k_2(4) = h*f1(x(i-1) + h, y(i-1) + k_1(3), z(i-1) + k_2(3));
    y(i) = y(i-1) + (k_1(1) + 2*k_1(2) + 2*k_1(3) + k_1(4))/6;
    z(i) = z(i-1) + (k_2(1) + 2*k_2(2) + 2*k_2(3) + k_2(4))/6;
end
M = [x', y];

printf("\nRunge-Kutta method for 2nd order\nStep h = %d\n", h);
printf("Obtained points:\n");
disp(M)
figure(3);
plot(x, y);
```