1. Analytical solution of the first order equation

$$y^{(1)}(x) = -2y(x) + 7e^{-2x}, \qquad y(0) = -10$$

Using:

$Y(s) = \mathcal{L}\,[y(x)]$

Formulas:

$\mathcal{L}\,[f'(x)] = s * Y(s) - f(0)$

$\mathcal{L}\,[e^{ax}] = \dfrac{1}{s-a}$

Calculations:

$$s * Y(s) + 10 = -2 * Y(s) + 7 * (\frac{1}{s+2})$$

$$Y(s) * (s+2) = \frac{7}{s+2} - \frac{10s+20}{s+2}$$

$$Y(s) = \frac{-10s-13}{(s+2)^2}$$

Using partial fractions

$$\frac{-10s-13}{(s+2)^2} = \frac{A}{(s+2)} + \frac{B}{(s+2)^2}$$

$$-10s - 13 = As + 2A + B$$

$$A = -10; \quad B = 7$$

$$Y(s) = \frac{-10}{(s+2)} + \frac{7}{(s+2)^2}$$

Finally, using the inverse Laplace transform to obtain the solution

$$\mathcal{L}^{-1}[\mathcal{L}[y(x)]] = \mathcal{L}^{-1}\left[\frac{-10}{(s+2)}\right] + \mathcal{L}^{-1}\left[\frac{7}{(s+2)^2}\right]$$

$$-10e^{-2x} = \mathcal{L}^{-1}[\mathcal{L}[-10e^{-2x}]] = \mathcal{L}^{-1}[\frac{-10}{(s+2)}]$$

$$7xe^{-2x} = \mathcal{L}^{-1}[\mathcal{L}[7xe^{-2x}]] = \mathcal{L}^{-1}[\frac{7}{(s+2)^2}]$$

$$\boldsymbol{y = \frac{7x - 10}{e^{2x}}}$$

2. Full program codes

1) Implement Euler method

```
#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;

double fun(double x, double y)
{
    return (-2*y+7*pow(2.71,-2*x));
}

double analytSol(double x)
{
    return (7*x-10)/(exp(2*x));
}

int main()
{
    ofstream X, Y, globalError, localError;
    X.open ("dataGraphX.txt");
    Y.open ("dataGraphY.txt");
    globalError.open ("globalError.txt");

    double h,maxv=5;
    cout << "Enter value of h: ";
    cin >> h;

    int arrsize=(5/h);
    double x[arrsize];
    double y[arrsize];
    double func[arrsize];

    x[0] = 0;
    //y[0] = -10;
    func[0] = 27;

    cout << "\ni\t\t" << "x\t\t" << "y" << endl << endl;
```

```

    for(int i=0; i<=arrsize; i++) {
        x[i] = x[0]+i*h;
        if (i == 0) {
            y[i] = -10;
        } else {
            y[i] = y[i-1] + (h*func[i-1]);
        }
        globalError << y[i] - analytSol(x[i]) << ",";
        //X << x[i] << ",";
        //Y << y[i] << ",";

        // for calculating simple error for Y2
        if (i%2 == 0) {
            X << x[i] << ",";
            Y << y[i] << ",";
        } //

        func[i] = fun(x[i],y[i]);
        cout << i << "       " << x[i] << "       " << y[i] << "       " << endl;
    }


    return 0;
}
```

2) Implement 4th order Runge-Kutta method

```
#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;

double fun(double a, double b, double c, double x, double y)
{
    return (a*y)+(b*exp(c*x));
}

double analytSol(double x)
{
    return (7*x-10)/(exp(2*x));
}

int main()
{
    ofstream X, Y, globalError, localError;
    X.open ("dataGraphX.txt");
    Y.open ("dataGraphY.txt");
    globalError.open ("globalError.txt");

    double a, b, c, d, e, h, sumdy;
    cout << "Give parameters: ";
    cin >> a; cin >> b; cin >> c; cin >> d; cin >> e; cin >> h;
    double x0 = 0;
    double y0 = d; // initial condition
    int arrSize = (e/h)+1;

    double x[arrSize][4];
    double y[arrSize][4];
    double k[arrSize][4];
    double dy[arrSize][4];

```

```
    cout << "\ni\t" << "x\t" << "y" << endl << endl;

    for (int i = 0; i <= arrSize; i++) {

        x[i][0] = x0;
        y[i][0] = y0;

        globalError << y[i][0] - analytSol(x[i][0]) << ",";
        X << x[i][0] << ",";
        Y << y[i][0] << ",";

        /* for calculating simple error for Y2
        if (i%2 == 0) {
            X << x[i][0] << ",";
            Y << y[i][0] << ",";
        } */

        if (i == 0) {
            k[i][0] = h*fun(a, b, c, x[i][0], y[i][0]);
        } else {
            k[i][0] = k[i-1][3];
        }
        dy[i][0] = k[i][0];

        cout << i << "\t" << x[i][0] << "\t" << y[i][0] << endl;

        x[i][1] = x0 + (0.5)*h;
        y[i][1] = y0 + (0.5)*k[i][0];
        k[i][1] = h*fun(a, b, c, x[i][1], y[i][1]);
        dy[i][1] = 2*k[i][1];

        cout << i << "\t" << x[i][1] << "\t" << y[i][1] << endl;

        x[i][2] = x0 + (0.5)*h;
        y[i][2] = y0 + (0.5)*k[i][1];
        k[i][2] = h*fun(a, b, c, x[i][2], y[i][2]);
        dy[i][2] = 2*k[i][2];

        cout << i << "\t" << x[i][2] << "\t" << y[i][2] << endl;

        x[i][3] = x0 + h;
        y[i][3] = y0 + k[i][2];
        k[i][3] = h*fun(a, b, c, x[i][3], y[i][3]);
        dy[i][3] = k[i][3];

        cout << i << "\t" << x[i][3] << "\t" << y[i][3] << endl;

        x0 = x[i][3];
        sumdy = ((dy[i][0] + dy[i][1] + dy[i][2] + dy[i][3]))/6;
        y0 = y0 + sumdy; // sum + y0

        cout << endl;
    }

    X.close();
    Y.close();
    globalError.close();
    return 0;
}
```

3. Programs’ outputs

1) Program outputs for Euler method

Working on parameters:

$$h = 0.2$$

```
Enter value of h: 0.2
```

The program outputs

```
i               x               y

0               0               -10
1               0.2             -4.6
2               0.4             -1.82041
3               0.6             -0.461645
4               0.8             0.146231
5               1               0.371777
6               1.2             0.413696
7               1.4             0.376156
8               1.6             0.311558
9               1.8             0.244562
10              2               0.185413
11              2.2             0.137205
12              2.4             0.0997433
13              2.6             0.0715376
14              2.8             0.0507693
15              3               0.0357278
16              3.2             0.0249711
17              3.4             0.0173547
18              3.6             0.0120048
19              3.8             0.00827131
20              4               0.00567986
21              4.2             0.00388917
22              4.4             0.00265649
23              4.6             0.00181066
24              4.8             0.00123188
25              5               0.000836767
```

2) Program outputs for 4th order Runge-Kutta method

Working on parameters:

$$a = -2, \qquad b = 7, \qquad c = -2, \qquad d = -10, \qquad e = 5, \qquad h = 0.2$$

```
Give parameters: -2 7 -2 -10 5 0.2
```

The program outputs

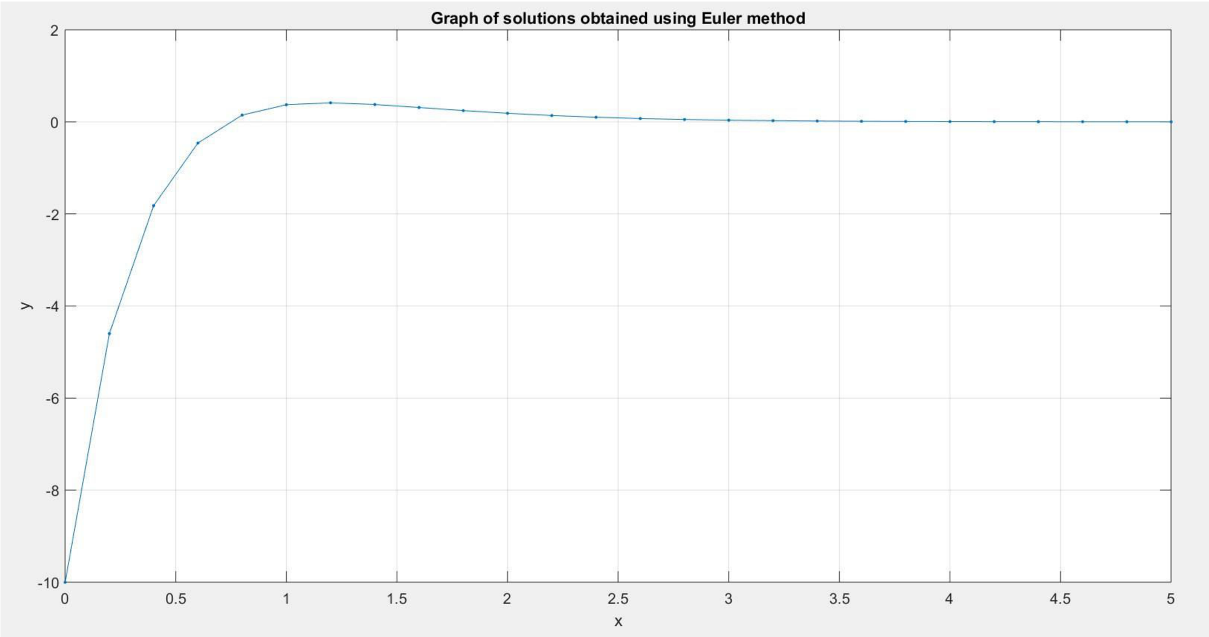
```
i       x       y

0       0       -10
0       0.1     -7.3
0       0.1     -7.96689
0       0.2     -5.66702

1       0.2     -5.76606
1       0.3     -4.16343
1       0.3     -4.5492
1       0.4     -3.17804

2       0.4     -3.24123
2       0.5     -2.29109
2       0.5     -2.52549
2       0.6     -1.716

3       0.6     -1.75427
3       0.7     -1.20023
3       0.7     -1.34161
3       0.8     -0.872392

4       0.8     -0.895254
4       0.9     -0.579448
4       0.9     -0.663655
4       1       -0.398373

5       1       -0.411823
5       1.1     -0.237413
5       1.1     -0.286778
5       1.2     -0.141987

6       1.2     -0.149744
6       1.3     -0.0578444
6       1.3     -0.086184
6       1.4     -0.0112878

7       1.4     -0.0156438
7       1.5     0.0291808
7       1.5     0.013371
7       1.6     0.0487097
```

```
8       1.6     0.046356
8       1.7     0.0651476
8       1.7     0.0566878
8       1.8     0.0704035

9       1.8     0.0692055
9       1.9     0.0742514
9       1.9     0.0700148
9       2       0.0725187

10      2       0.0719704
10      2.1     0.0702877
10      2.1     0.0684098
10      2.2     0.0656003

11      2.2     0.0654037
11      2.3     0.0608778
11      2.3     0.0602645
11      2.4     0.0553705

12      2.4     0.0553534
12      2.5     0.0500401
12      2.5     0.050062
12      2.6     0.0447618

13      2.6     0.0448272
13      2.7     0.0397365
13      2.7     0.0400415
13      2.8     0.0351338

14      2.8     0.0352293
14      2.9     0.030791
14      2.9     0.0311904
14      3       0.0269917

15      3       0.0270903
15      3.1     0.0234271
15      3.1     0.0238255
15      3.2     0.0204013

16      3.2     0.0204906
16      3.3     0.0175735
16      3.3     0.0179282
```

```
17      3.4     0.0152993
17      3.5     0.0130342
17      3.5     0.0133308
17      3.6     0.0112436

18      3.6     0.0113047
18      3.7     0.00957855
18      3.7     0.00981684
18      3.8     0.00823369

19      3.8     0.00828161
19      3.9     0.00698518
19      3.9     0.00717138
19      4       0.00598668

20      4       0.00602351
20      4.1     0.00506099
20      4.1     0.00520356
20      4.2     0.00432659

21      4.2     0.00435443
21      4.3     0.00364652
21      4.3     0.003754
21      4.4     0.00311058

22      4.4     0.00313136
22      4.5     0.00261475
22      4.5     0.00269479
22      4.6     0.00222621

23      4.6     0.00224156
23      4.7     0.00186705
23      4.7     0.00192606
23      4.8     0.00158695

24      4.8     0.00159819
24      4.9     0.00132821
24      4.9     0.00137136
24      5       0.00112728

25      5       0.00113545
25      5.1     0.000941775
25      5.1     0.000973115
25      5.2     0.000798243
```

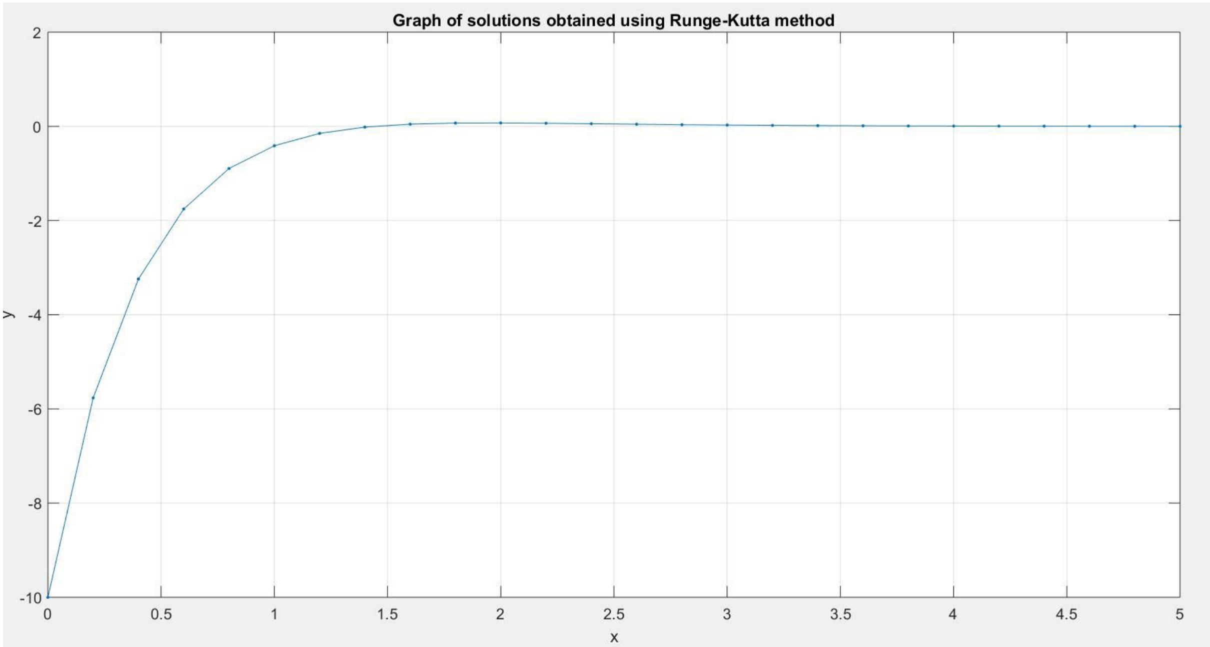
4. Graph solutions obtained using the programs

1) Using Euler method



h = 0.2

2) Using Runge-Kutta method



h = 0.2

5. Analysis of both methods' error

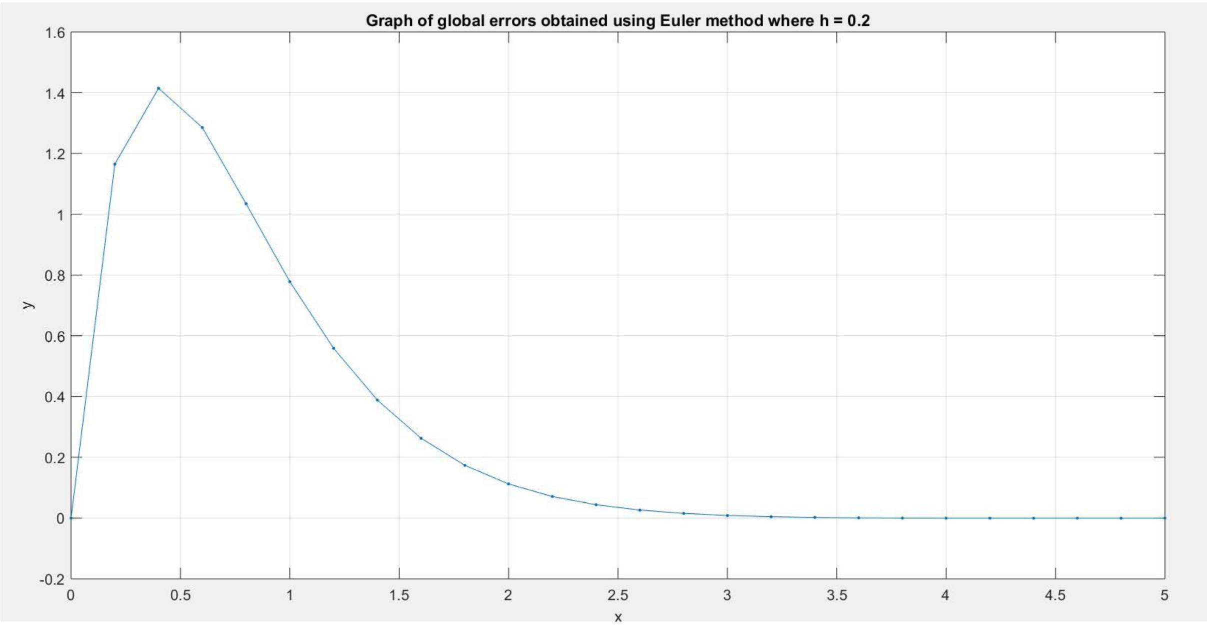
1) Global error

$$\boldsymbol{g_k = y_k - y(t_k)}$$

where,

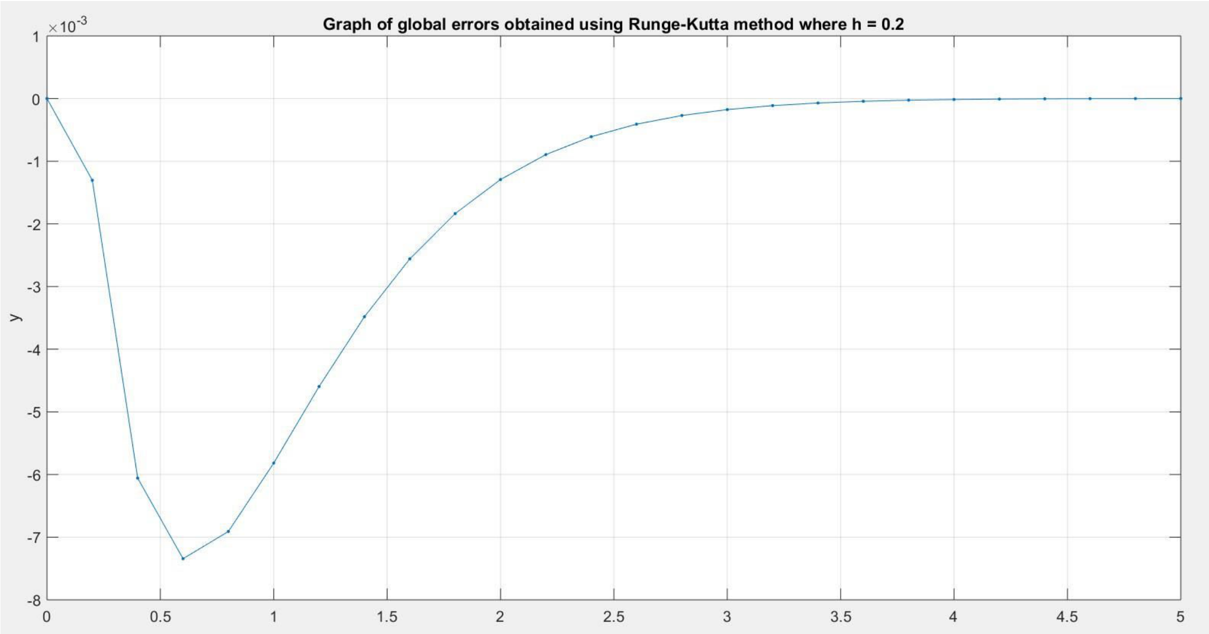
y_k is a computed solution and $y(t_k)$ is a true solution.

1. Using Euler method



h = 0.2

2. Using Runge-Kutta method



h = 0.2

2) Error obtained using simple calculations

To estimate the error, there have to be chosen two intervals of length $h_1 = h$ and $h_2 = \frac{h}{2}$.

$$Y_1 = y_1 + E_1, \qquad Y_2 = y_1 + E_2,$$

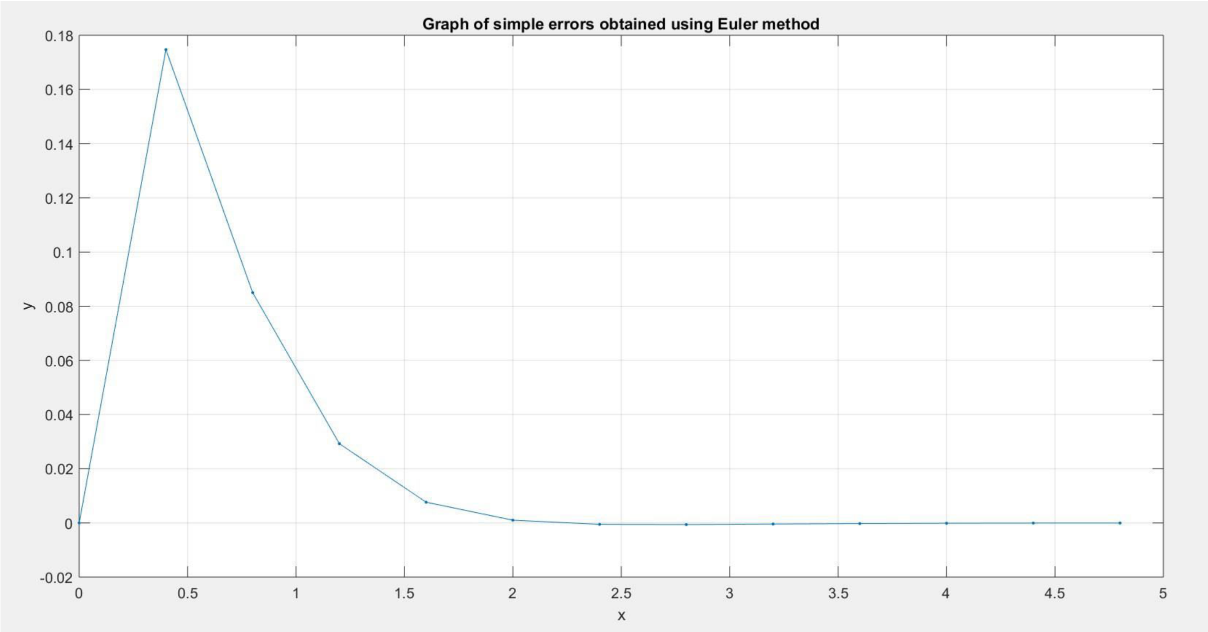
where

$$E_1 = Ch_1^5, \qquad E_2 = 2Ch_2^5 = \frac{E_1}{16},$$

and from that

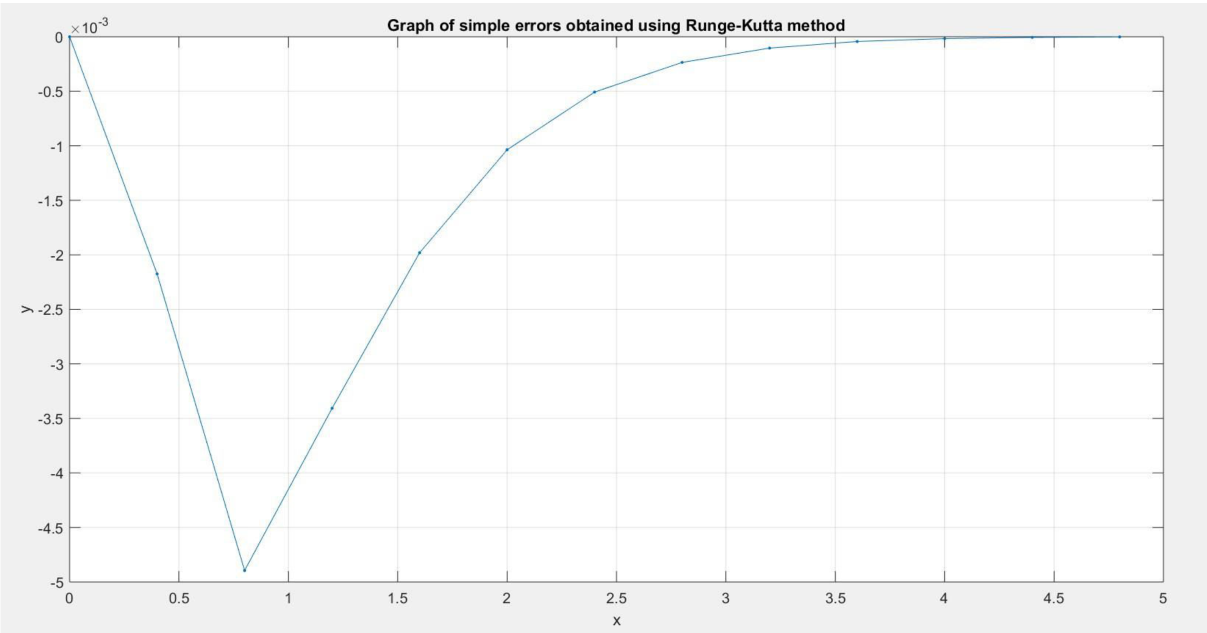
$$\boldsymbol{E_2 = \frac{(Y_1 - Y_2)}{15}}$$

1. Using Euler method



h1 = 0.4, h2 = 0.2

2. Using Runge-Kutta method



h1 = 0.4, h2 = 0.2

3) Error Term (using Taylor series)

Using Runge-Kutta method

The coefficient C_4 of h^5 is

$$C_4 = \frac{1}{6}(k_1^{(4)} + 2k_2^{(4)} + 2k_3^{(4)} + k_4^{(4)})$$

$$e = C_4 - \frac{1}{120} y(0)^{(5)}$$

$$C_4 = -0.00159342$$

$$y(0)^{(5)} = 880$$

$$e = -7.33492675$$

Estimating the error, for $h = 0.2$

$$|\boldsymbol{E}| = \left|\boldsymbol{eh^5}\right| \approx \mathbf{0.00234718}$$

6. Taylor series

$$y^{(1)}(x) = -2y(x) + 7e^{-2x}, \qquad y(0) = \mathbf{-10}$$

$$y^{(1)}(0) = -2y(0) + 7e^{-2*0} = \mathbf{27}$$

$$y^{(2)}(x) = -2y^{(1)}(x) - 14e^{-2x}$$

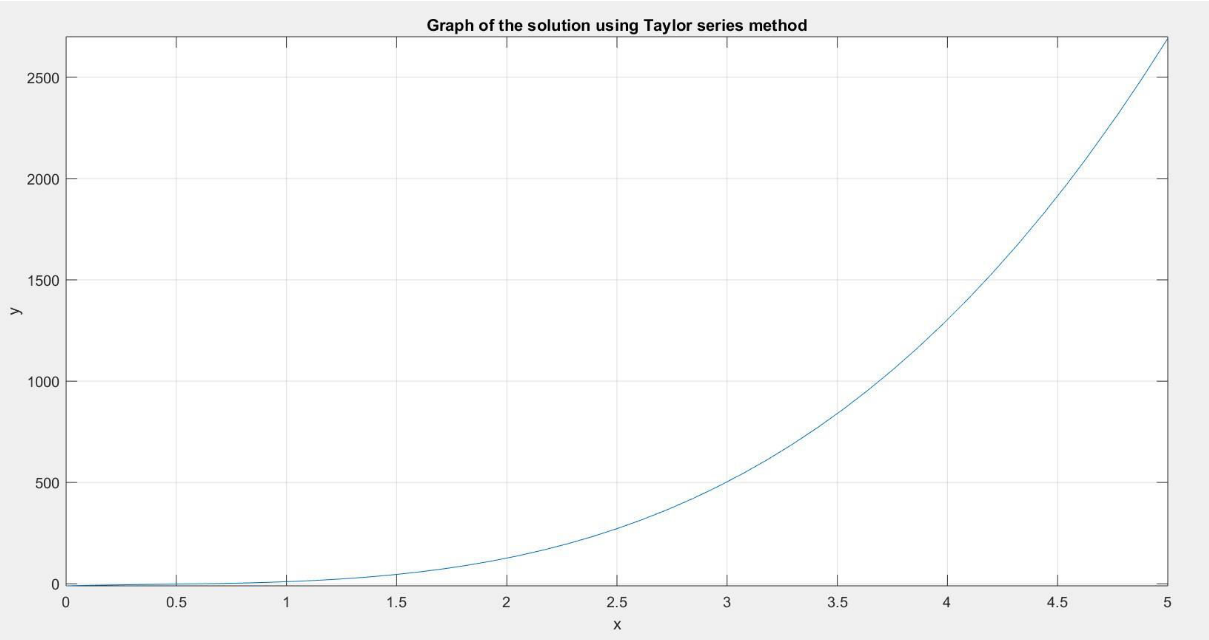
$$y^{(2)}(0) = -2y^{(1)}(0) - 14e^{-2*0} = \mathbf{-68}$$

$$y^{(3)}(x) = -2y^{(2)}(x) + 28e^{-2x}$$

$$y^{(3)}(0) = -2y^{(2)}(0) + 28e^{-2*0} = \mathbf{164}$$

Approximate solution of differential equation is of the form

$$y(x) \approx y(0) + y^{(1)}(0)x + \frac{1}{2}y^{(2)}(0)x^2 + \frac{1}{6}y^{(3)}(0)x^3 = \mathbf{-10 + 27x - 34x^2 + \frac{82}{3}x^3}$$



7. Conclusions

The accurecies of the presented methods can be observed from the task dedicated to analysis of both methods' error. From the obtained values we can conclude, that:

- The Runge-Kutta method is more accurate than the Euler method, giving very good approximation, for every chosens step,

- The Runge-Kutta method has a small error (of the magnitude of 10^{-3}), it seems that a step size doesn't have any noticeable influence on the error,

We can conclude that Euler method has accuracy roughly equal to its step size.