**Author** : Wiktor Łazarski

**Index number** : 281875

**Field of study** : Computer Science

**Faculty** : Electronics and Information Technology, Warsaw University of Technology

1. Task 1 : Least-square approximation.
   1. Introduction.
   2. Approximation using system of normal equation – Results.
   3. Approximation using QR distribution – Results.
   4. Comparison both method of finding least square approximation.
2. Task 2 : Systems of ODE.
   1. Introduction.
   2. Runge-Kutta method of 4th order, a constant step size – Results
   3. Adams PC (P5EC5E) – Results
   4. Testing and comparing both method from subpoint a.
   5. Runge-Kutta method of 4th order with a variable step size.
3. Appendices – MATLAB codes  
   3.1 Approximation using system of normal equations.

3.2 Approximation using QR distribution.

3.3 Runge-Kutta of 4th order with a constant step.

3.5 Adams PC (P5EC5E).

3.6 Runge-Kutta method of 4th order with a variable step size.

**The first task** is to determine a polynomial function *y = f(x)* that fits the best, using least-square approximation, for the following data samples :

|  |  |
| --- | --- |
| xi | yi |
| -5 | -35.7986 |
| -4 | -19.4300 |
| -3 | -9.7370 |
| -2 | -3.1635 |
| -1 | -0.6503 |
| 0 | 1.5879 |
| 1 | 1.5176 |
| 2 | 2.1830 |
| 3 | 5.1024 |
| 4 | 11.0910 |
| 5 | 22.0003 |

The *approximation problem* can be defined in the following way : to find a function *F\** belonging to set *Xn* closest to *f* in a certain sense, usually in the sense of a certain distance δ*(f – F)* defined by a norm ||∙||. Thus, approximation of the function *f* means finding the coefficients *a0,…,an* of *F* such that the norm ||*f – F*|| is minimized.

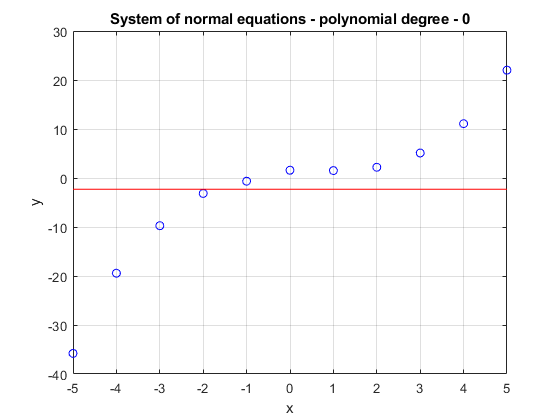
To perform least-square approximation, we must define A as a matrix , where N – number of samples, n – degree of polynomial, for every

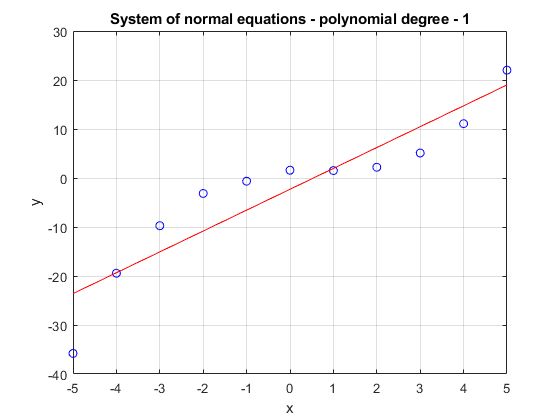
Solving least-square task comes down to finding the vector *a* which contain coefficients of polynomial. In this case we will research two approaches :

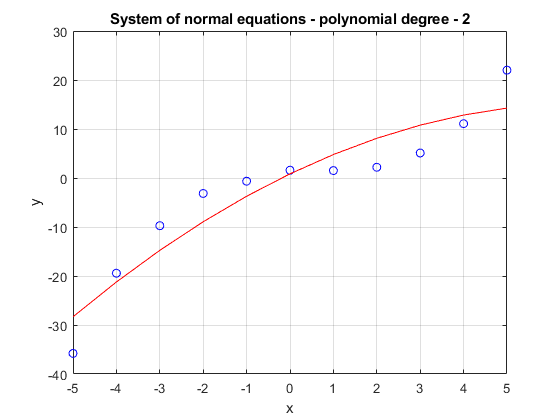
Using and solving system of normal equations :

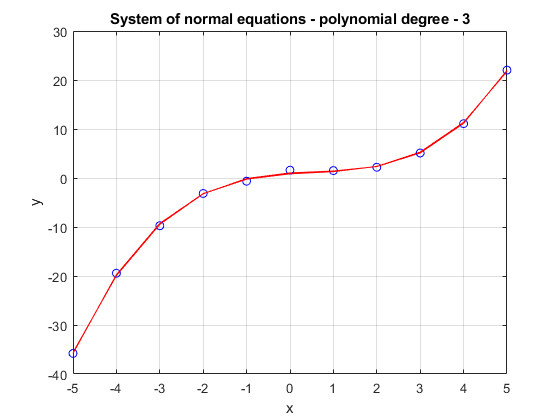
Using and solving system resulting in QR factorization :

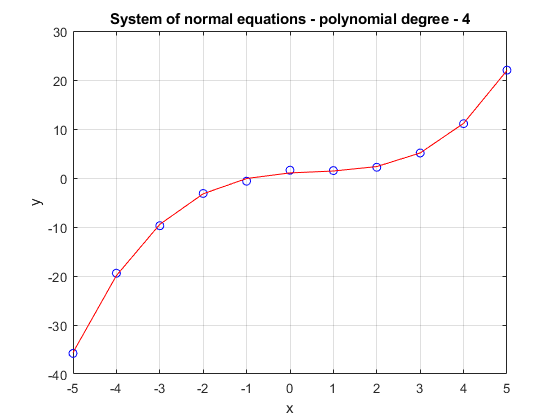
**The following approximations** of given function where obtain :

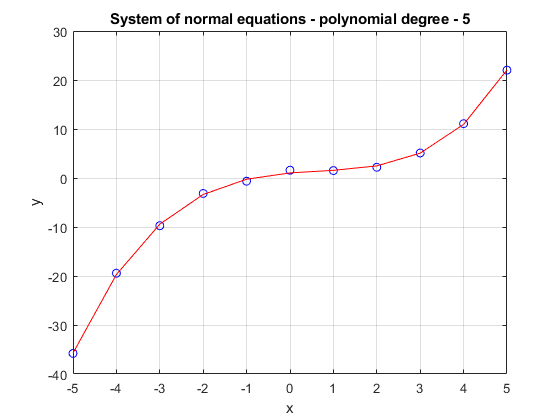




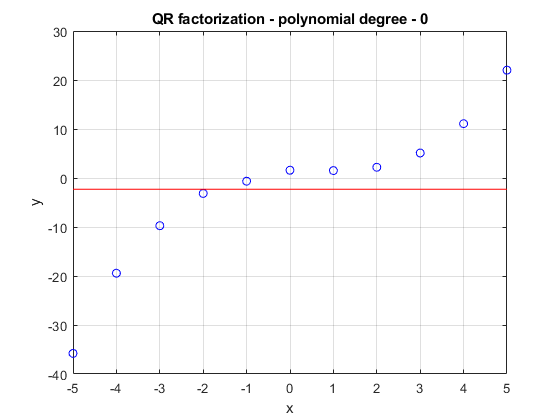


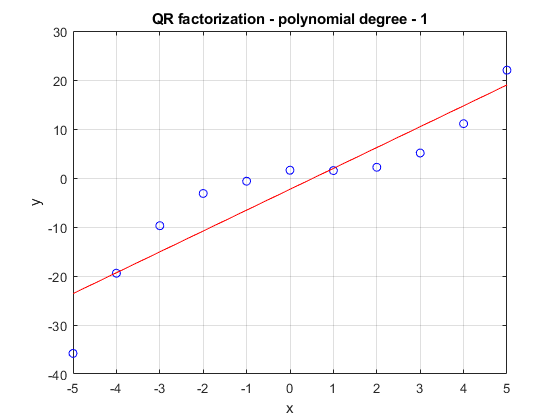


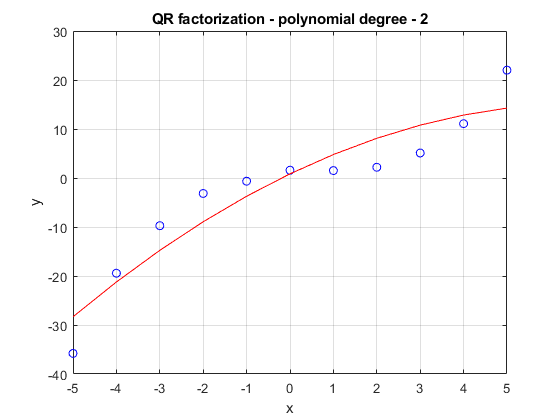


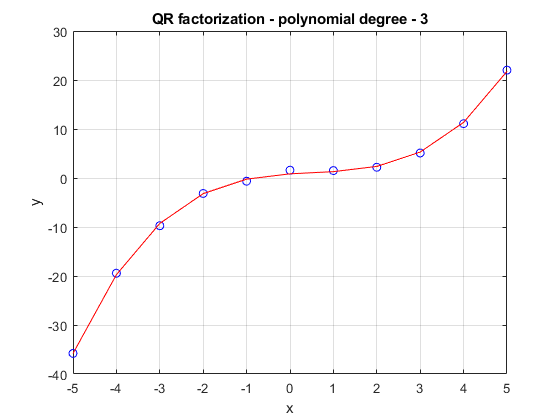


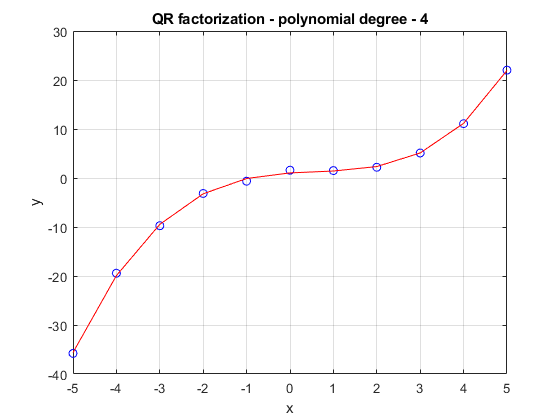
**The following approximations** of given function where obtain :

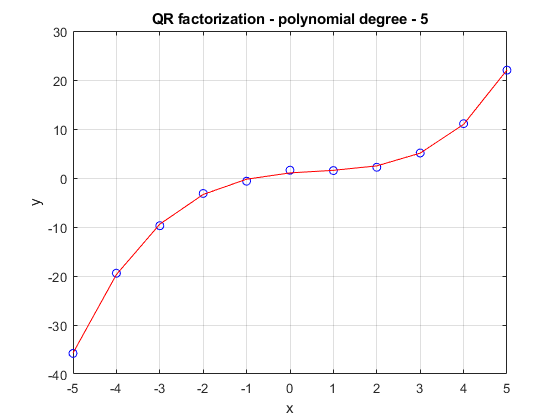












**To compare** both methods we will look at the residuum error obtain during computing approximate polynomial with a given degree. Table below shows the results of those calculations.

|  |  |  |
| --- | --- | --- |
| Degree of polynomial | Residuum error  using system of normal equations | Residuum error  using QR distribution |
| 0 | 3.5527e-15 | 0 |
| 1 | 5.6954e-14 | 2.6645e-15 |
| 2 | 5.6954e-14 | 5.0242e-15 |
| 3 | 3.5527e-15 | 8.7023e-15 |
| 4 | 1.8189e-12 | 9.6710e-15 |
| 5 | 2.9104e-11 | 1.4947e-14 |
| 6 | 4.1319e-11 | 2.4027e-14 |
| 7 | 9.3178e-10 | 6.4035e-14 |
| 8 | 2.0827e-09 | 7.2071e-14 |
| 9 | 5.9815e-08 | 3.6645e-13 |
| 10 | 4.47089e-06 | 3.4017e-13 |

**Conclusions :**  As you can see, the system of equations resulting from the QR distribution maintains a good conditioning longer in contrast to the system of normal equations, which quickly loses accuracy of result. Nevertheless, for the degrees of polynomials of a degree greater than or equal to 10, the approximate function in both cases begins to differ from the previous results. This is due to the fact that at some point we stop approximating the function and only measuring data. In both cases we can observe that approximation errors decrease with increasing polynomial degree.

**The second task** is to determine the trajectory of the motion of a point define by the equations:

on the interval [0, 20]. The following initial conditions are given : x1(0) = 0, x2(0) = 0.3. We will evaluate the solution in two following ways :

1. Runge-Kutta method of 4th order (RK4) and Adams PC (P5EC5E) – each method a few times, with different constant step-sizes unital an „optimal” constant step size is found, i.e., when its decrese does not influence the solutions significantly but its increase does.
2. Runge-Kutta method of 4th order (RK4) with a variable step size automatically adjusted by the algorithm, making error estimation according to the step-doubling rule.

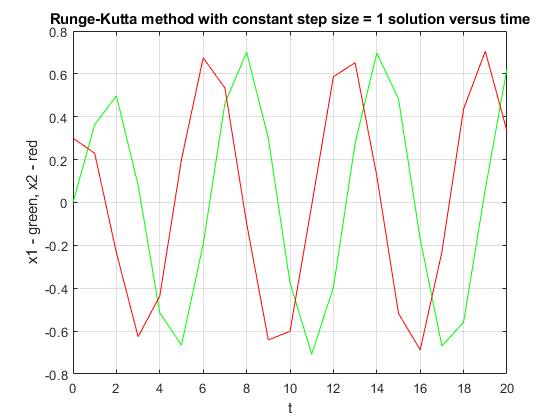
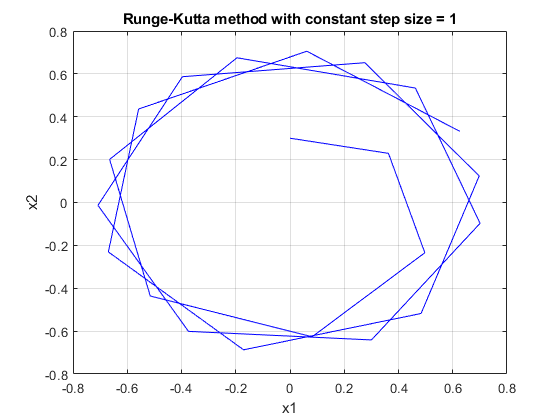
**Runge-Kutta method 4th order**(RK4, “classical”) can be define using following formulas :

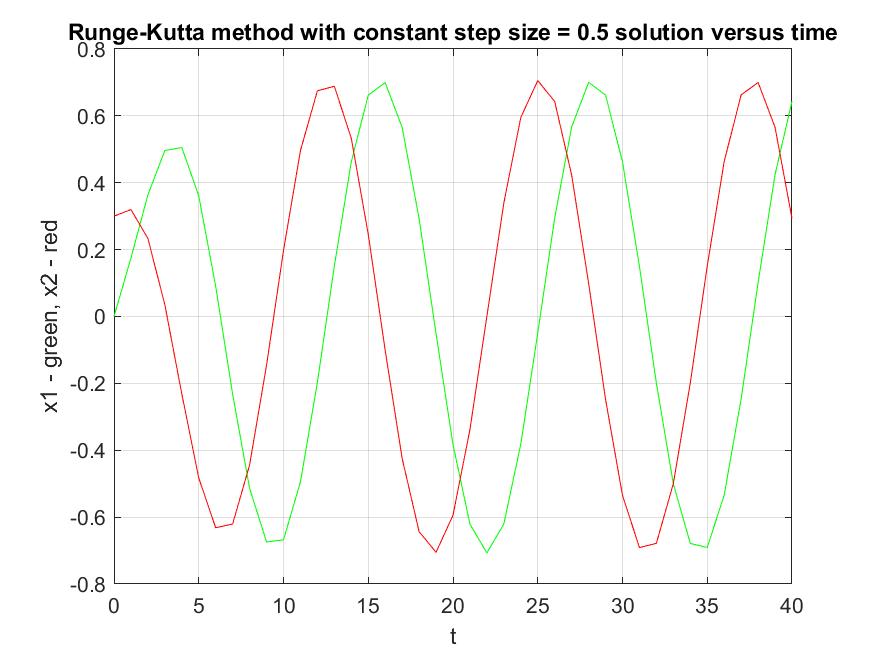
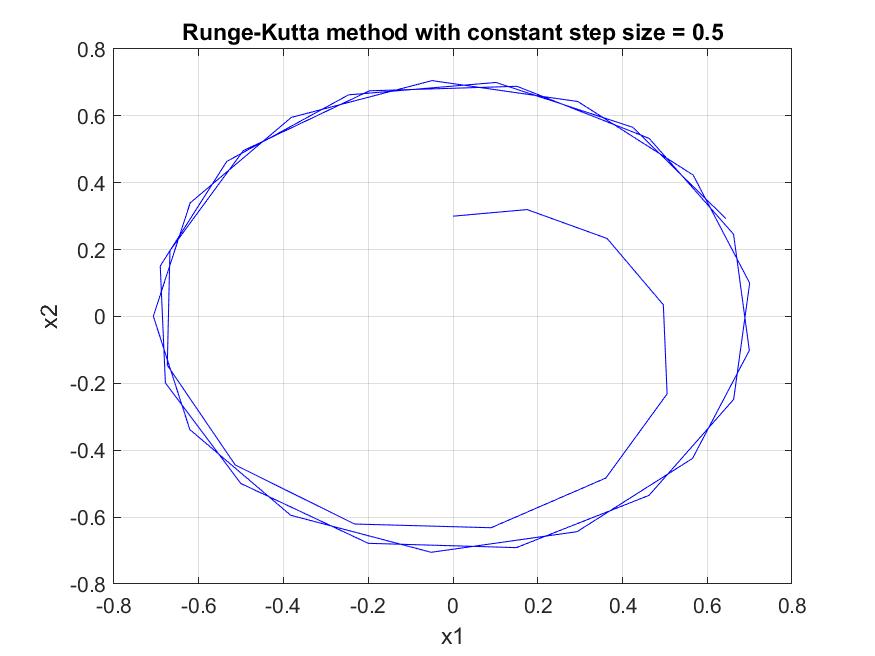
The coefficient k1 represents the derivative at . The value k2 is calculated as in the modifed Euler's method – as a derivative of the solution calculated by the standard Euler's method at the midpoint , the dashed tangent line correspond to this derivative. Next, the value k3 is calculated similarly as it was for k2, but this time at the point -i.e., with a tangent line corresponding to k2.. Finally, we start with this line from the initial point until the endpoint ), i.e., the derivative k4 of a solution at the point over sthe one step intevral: one at the initial point, two at the midpoint and on at the endpoint. The final approximation of the solution derivative for the final full step of the method is calculated as a weighted mean value of these derivatives, with the weight 1 for the initial and end points and the weight 2 for the midpoint.

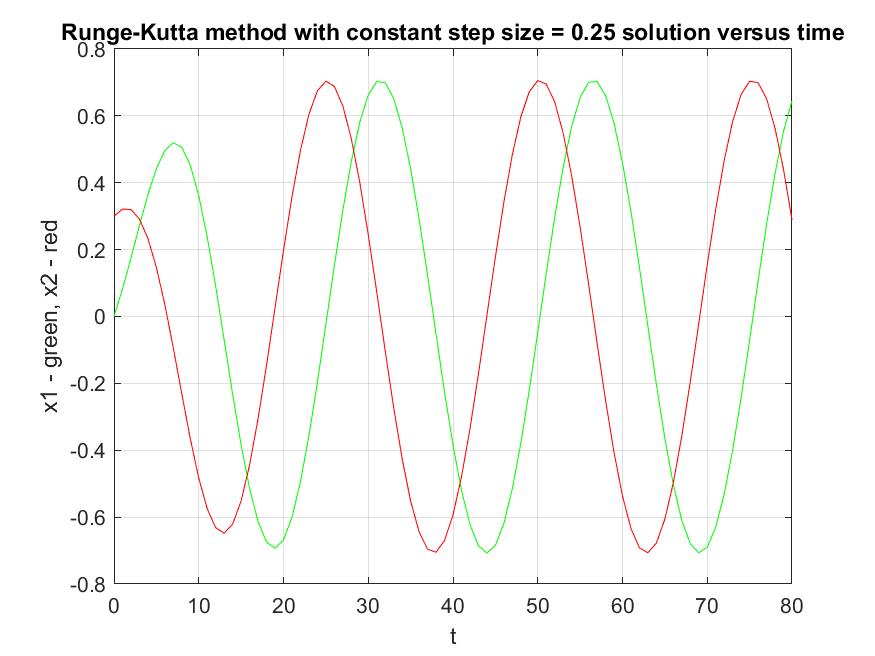
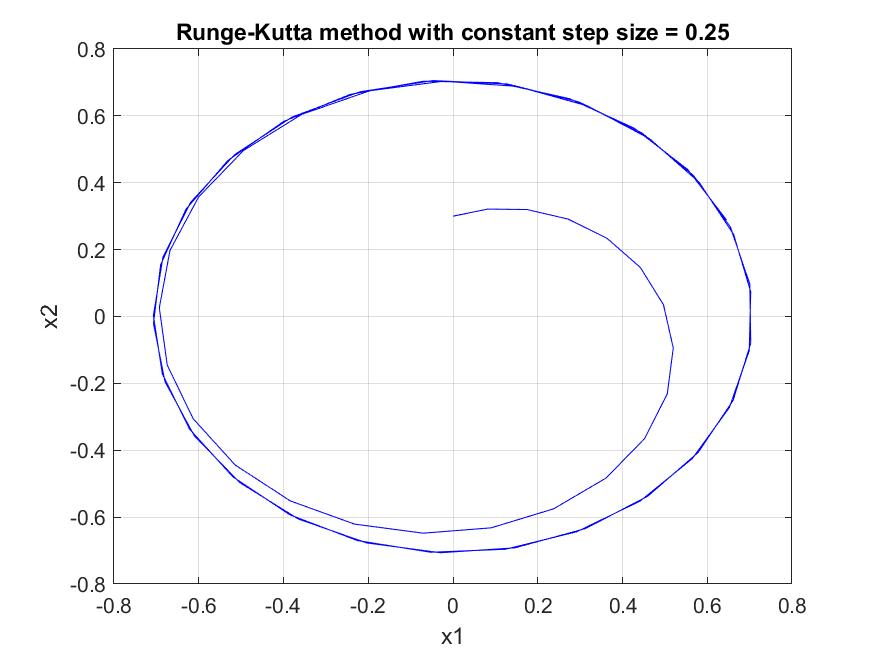
The step was decreased until the plot started to present sufficient accuracy. Error of single step can be calculated using formula :

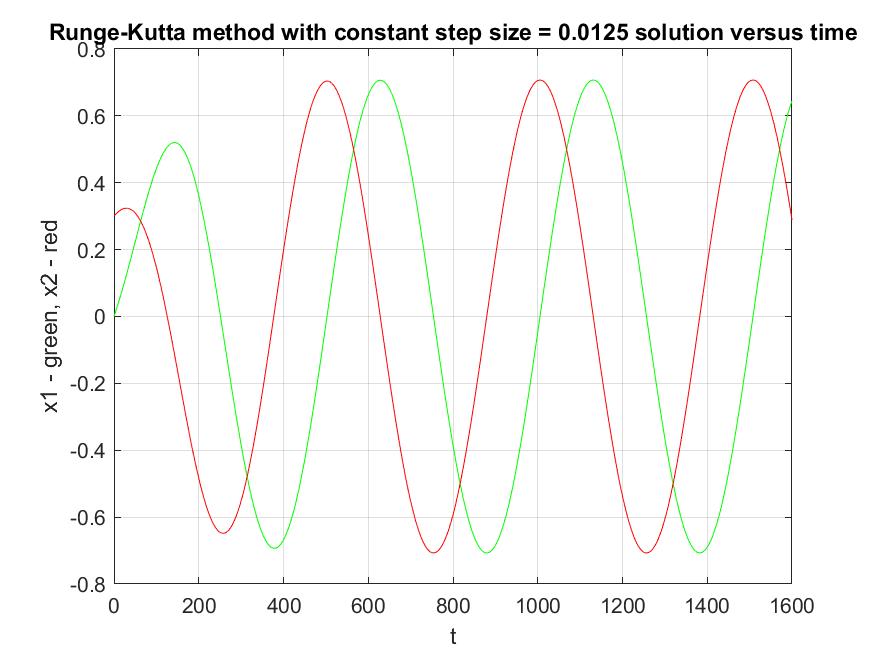
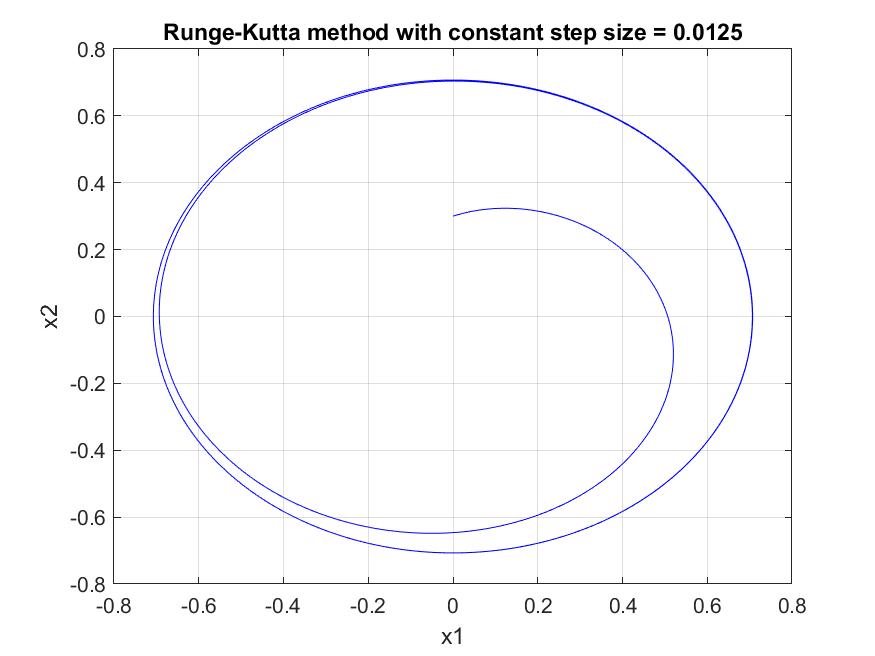
where, – new point obtained from the step of length , – new point obtain from two additional steps of length – order of method.

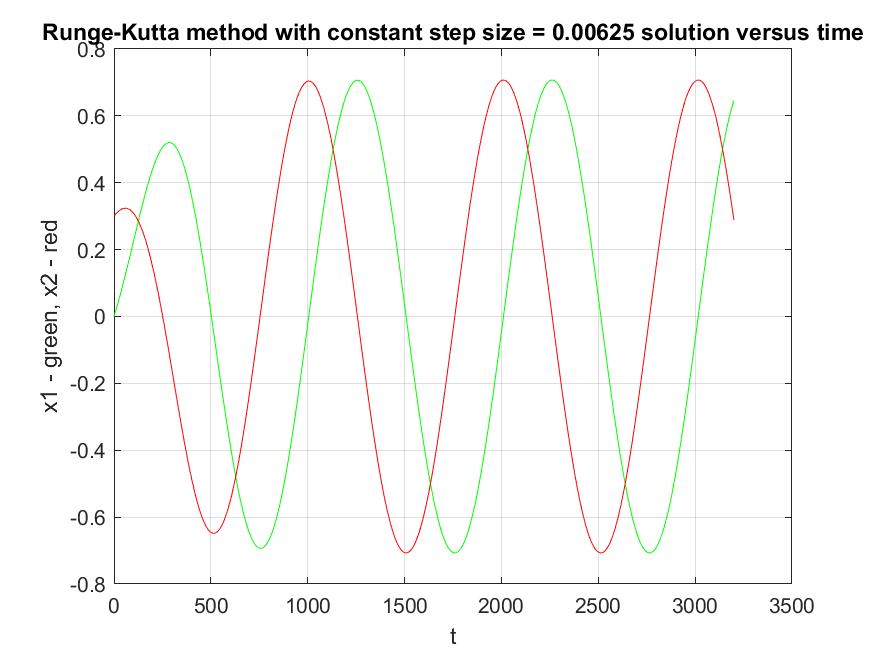
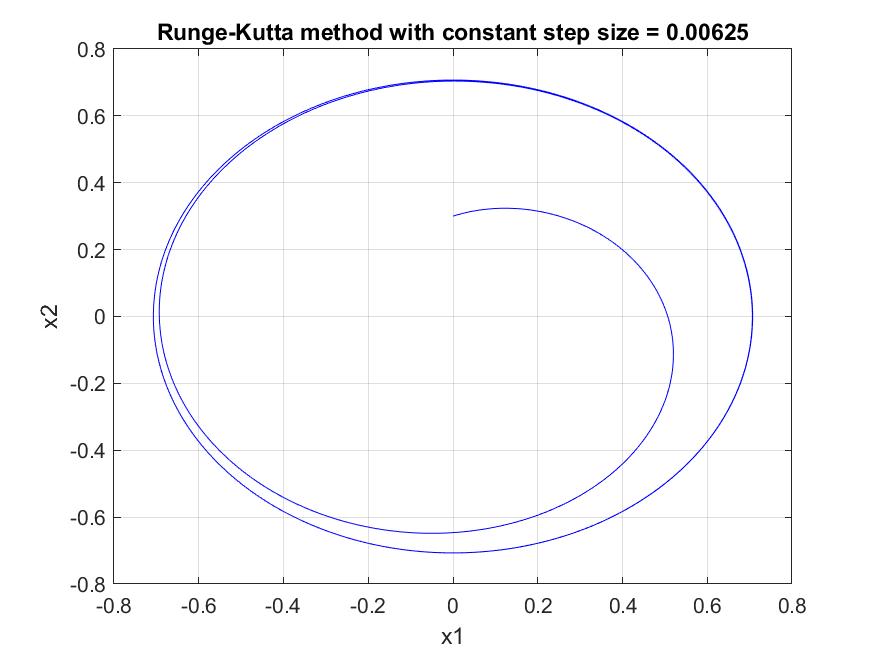
**Results(for different step sizes)** **obtained** using Runge-Kutta algorithm of 4th order presents as follow :

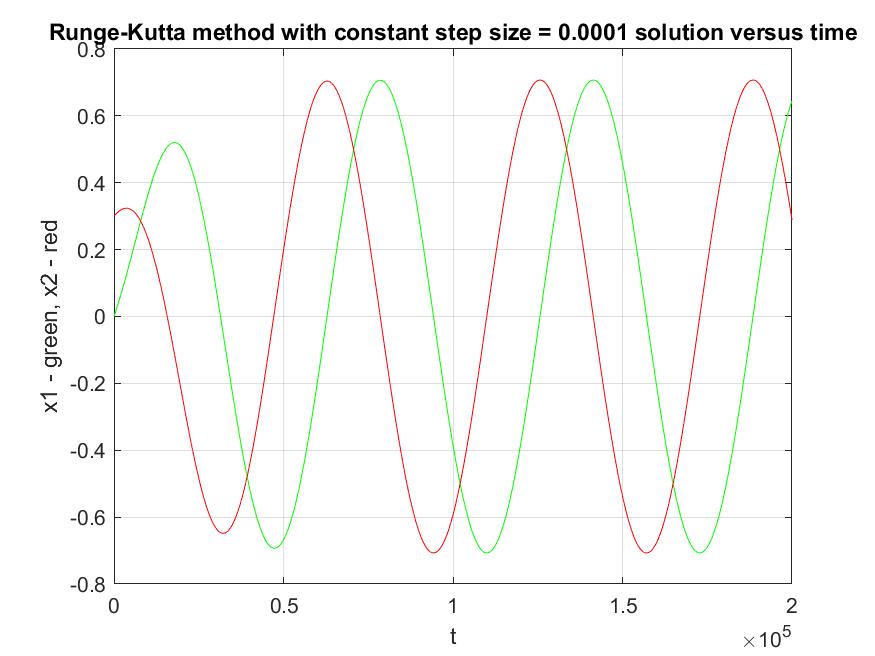
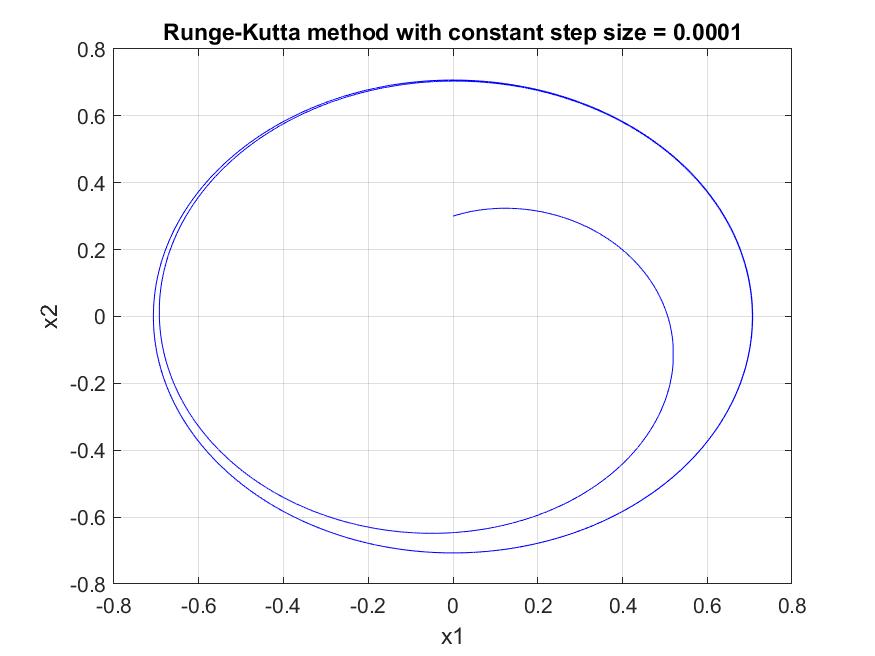












**Conclusion :**  We can see that starting from *h = 0.0125* obtained trajectory does not change a lot(probably if we zoom and compare plots we would see some differences). For that matter, we can conclude that it presents a proper function.

**The second method** that will be researched is so called Adams PC (P5EC5E) method.

An initial differential equation problem :

can be equivalently formulated in the form of an integral equation:

Consider this integral on the interval [xn-1, xn]:

leads to Adams methods.

There can be two forms of Adams method distinguished – explicit and implicit.

**Explicit Adams methods :**

The function *f* (*x , y* (*x* )) is replaced by an interpolation polynomial *W(x)* of order *k-1*, calculated at the points *xn*−1 *,* ... *, xn*−*k* . Using the Lagrange interpolation formula we have:

where *Lj* (*x*) are the Lagrange polynomials:

Assuming that the points are equally spaced *xn*− *j* = *xn* − *jh , j* = 1, 2, ... *, k* , the integration process yields:

where values of the coefficients β *j* are given.

**Implicit Adams methods:**

The function *f* (*x , y* (*x* )) is now replaced by an interpolation polynomial *W\*(x)* of order *k*, calculated at the points *xn ,* ... *, xn*−*k* with the corresponding solution values *y* (*xn*−1)≈ *yn*− *j* . Reasoning in the same way as it was done in the case of the explicit methods, we finally get:

where values of the parameters *,* for *k* = 1, ... *,* 7, are given.

**The predictor-corrector method PkECkE:**

For the Adams methods the PkECkE algorithm has the following form:

P:

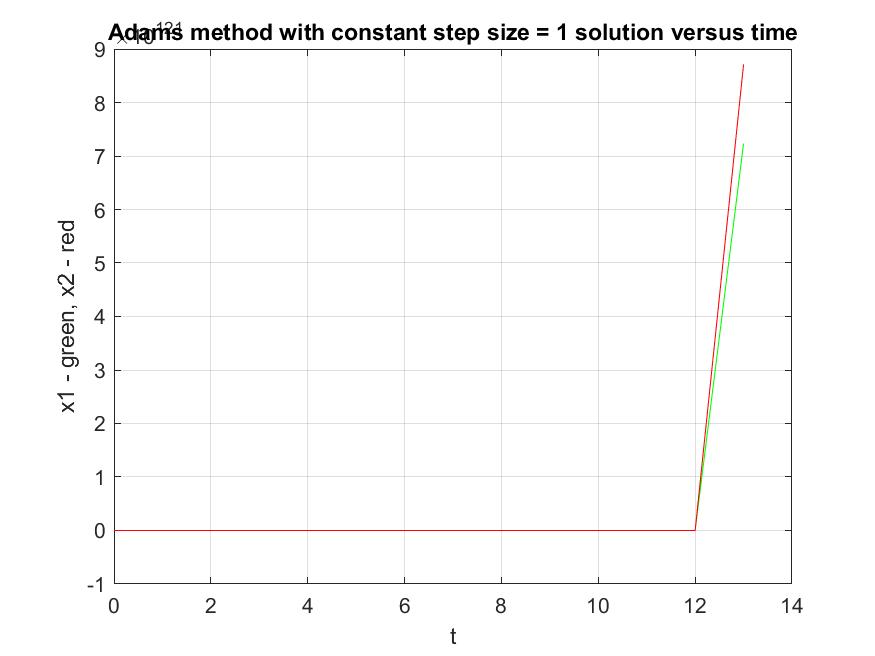
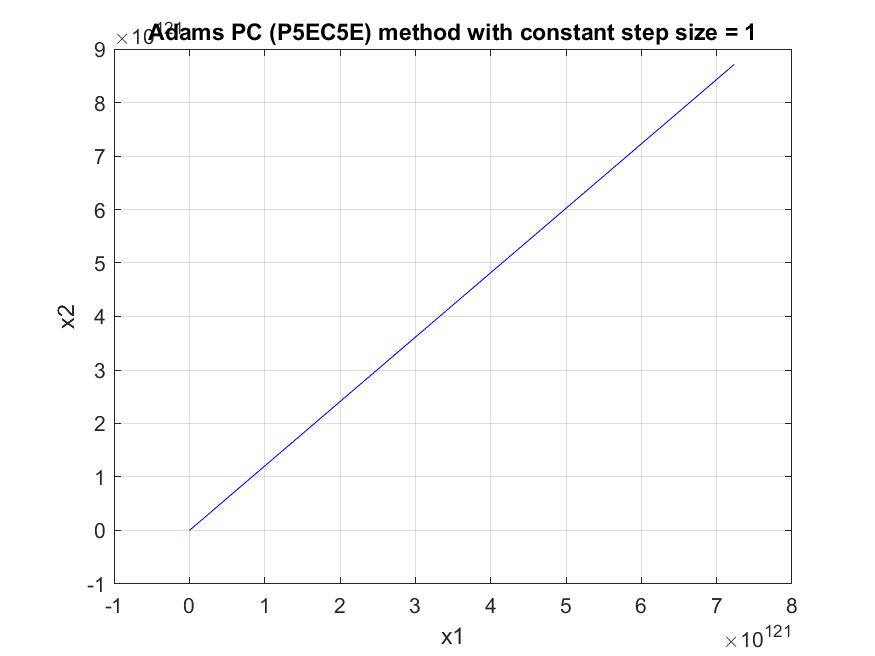
E:

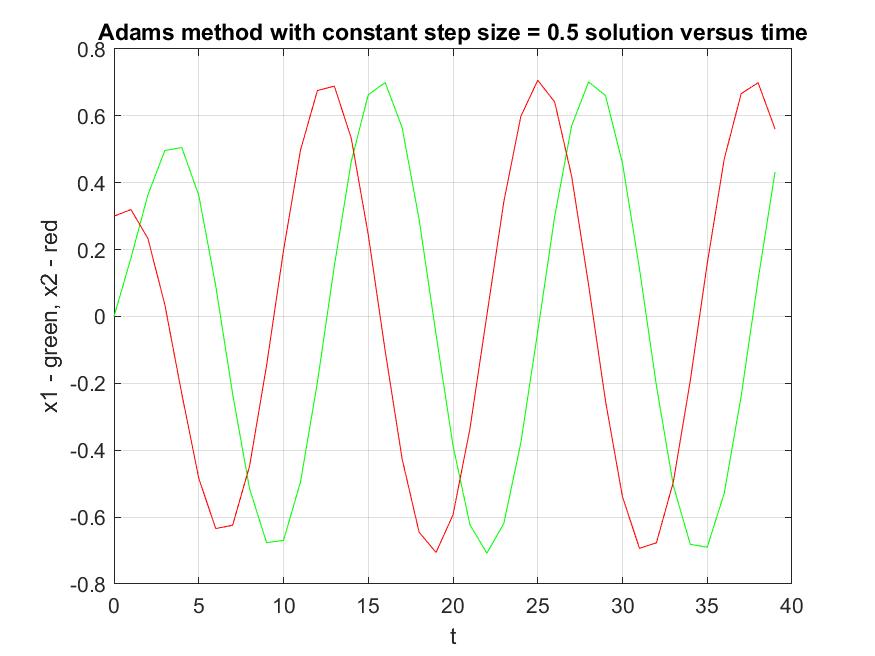
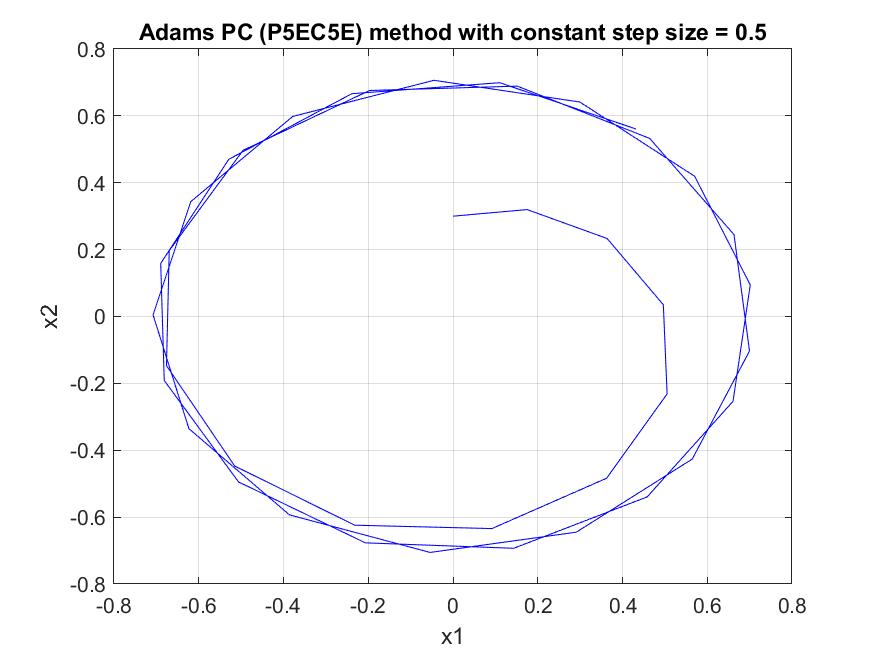
K:

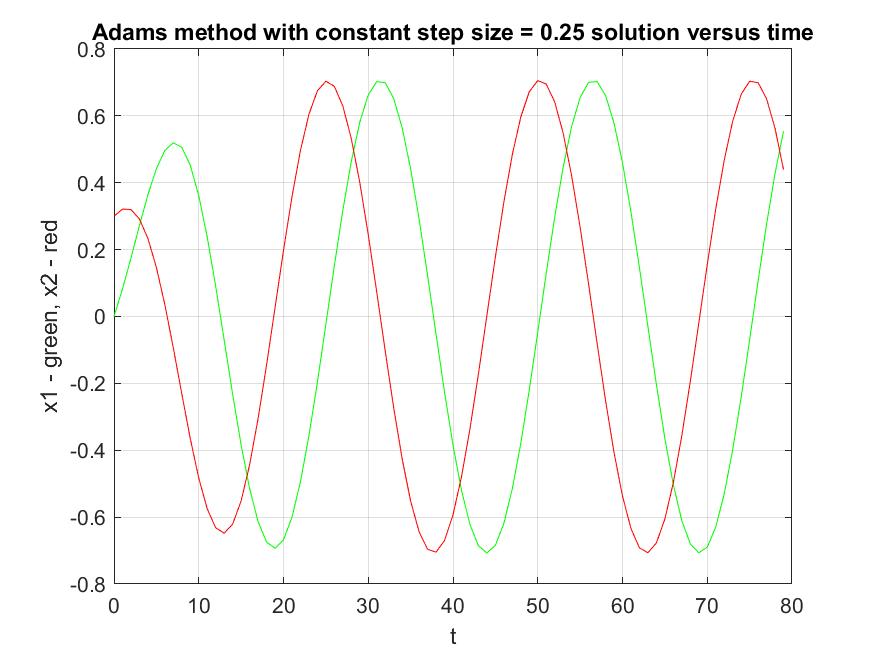
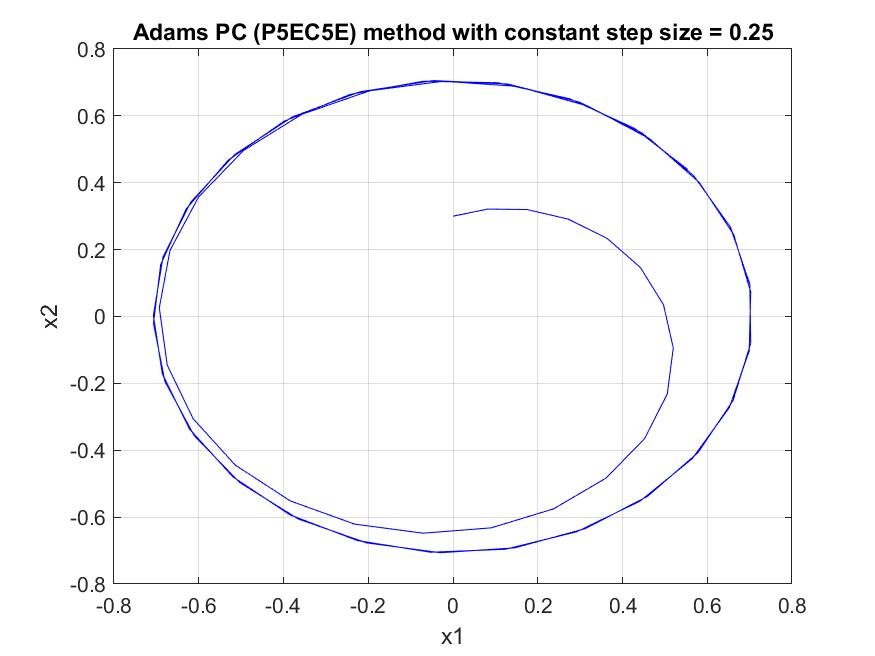
E:

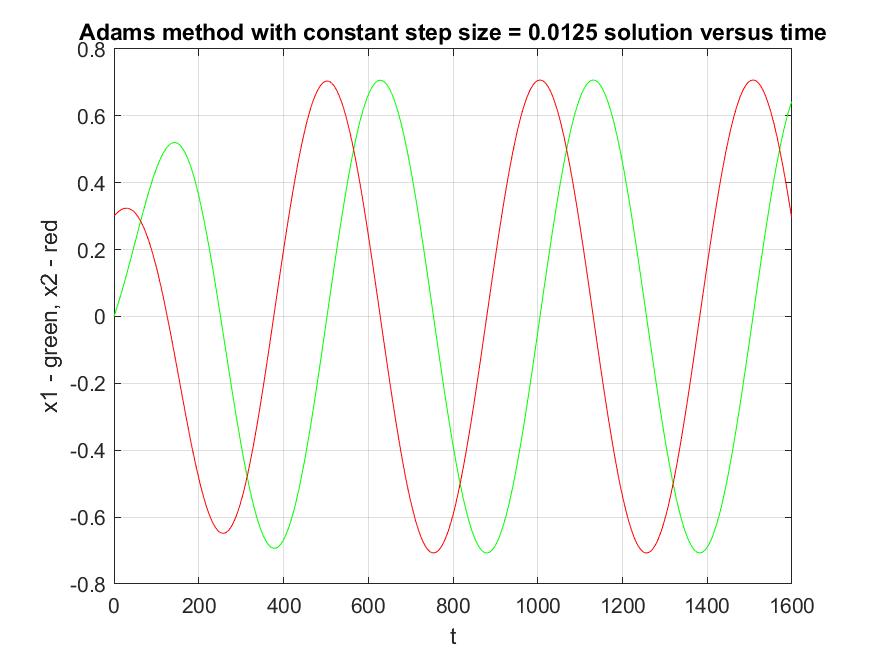
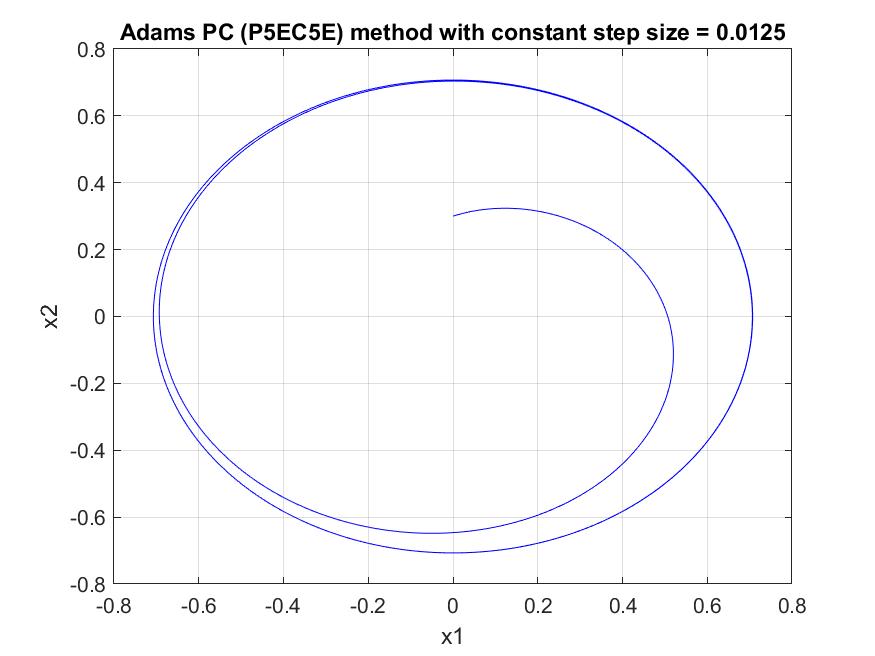
**The error approximation** is calculated by the formula:

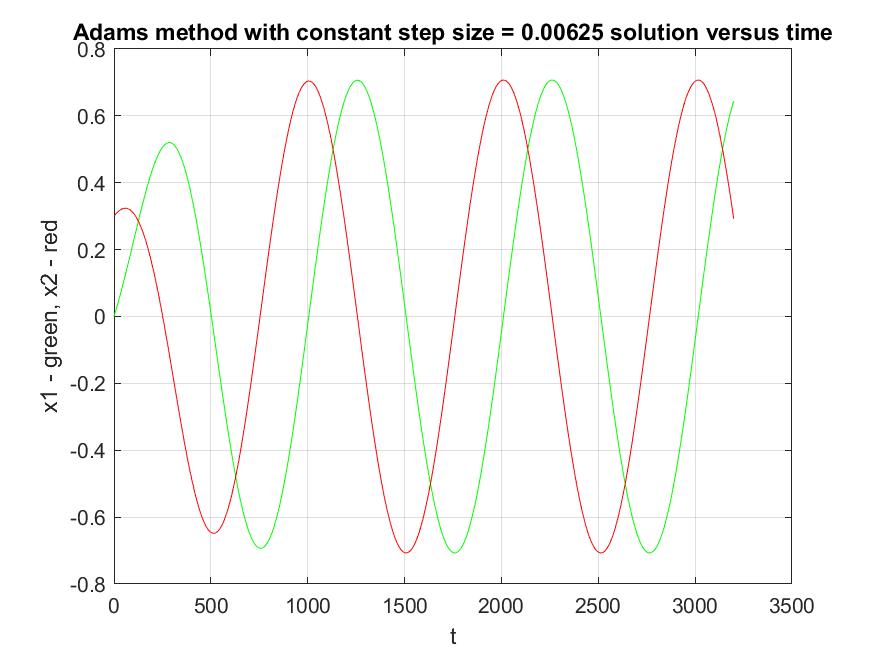
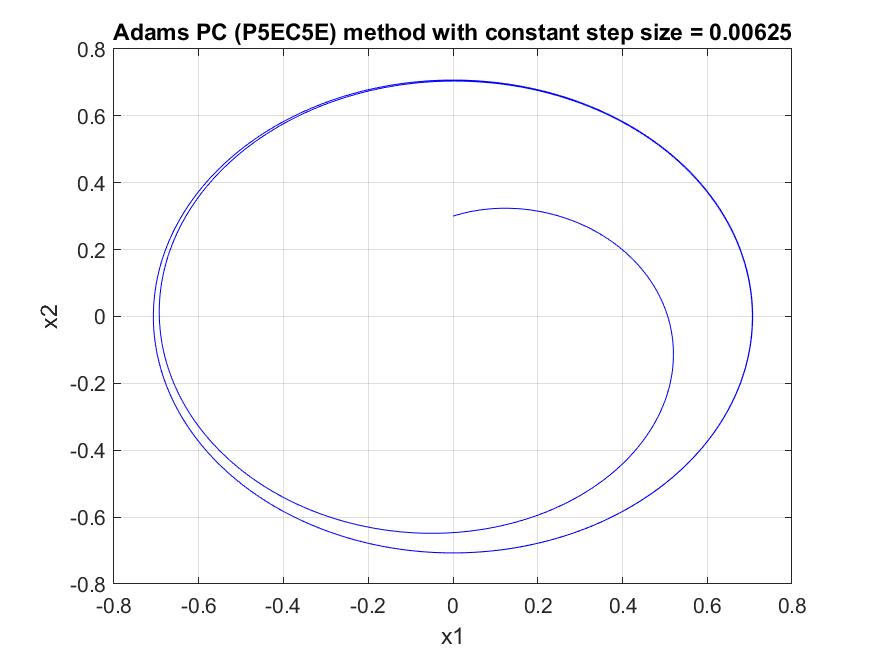
**Results(for different step sizes)** **obtained** using Adams PC (P5EC5E) algorithm presents as follow :

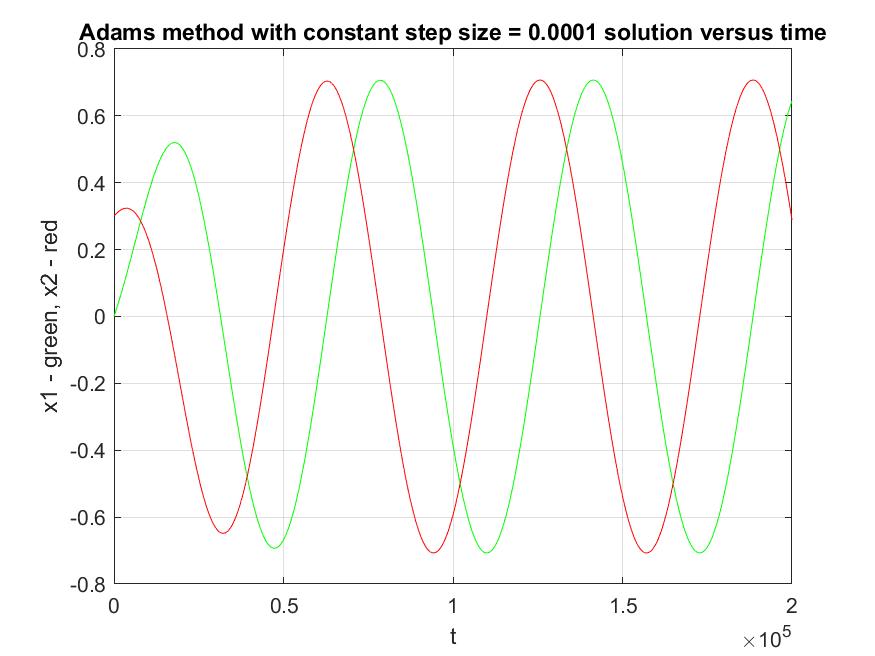
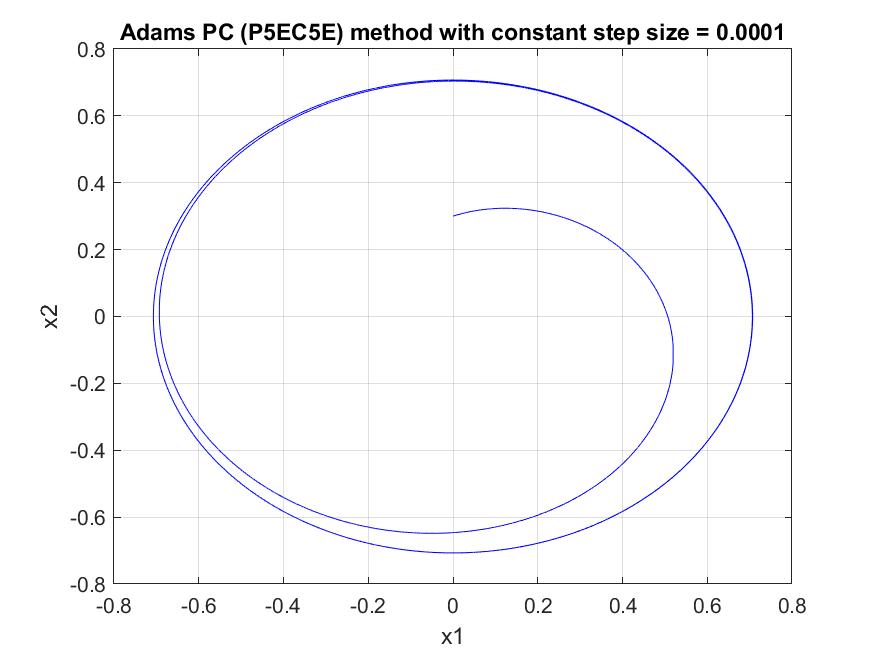










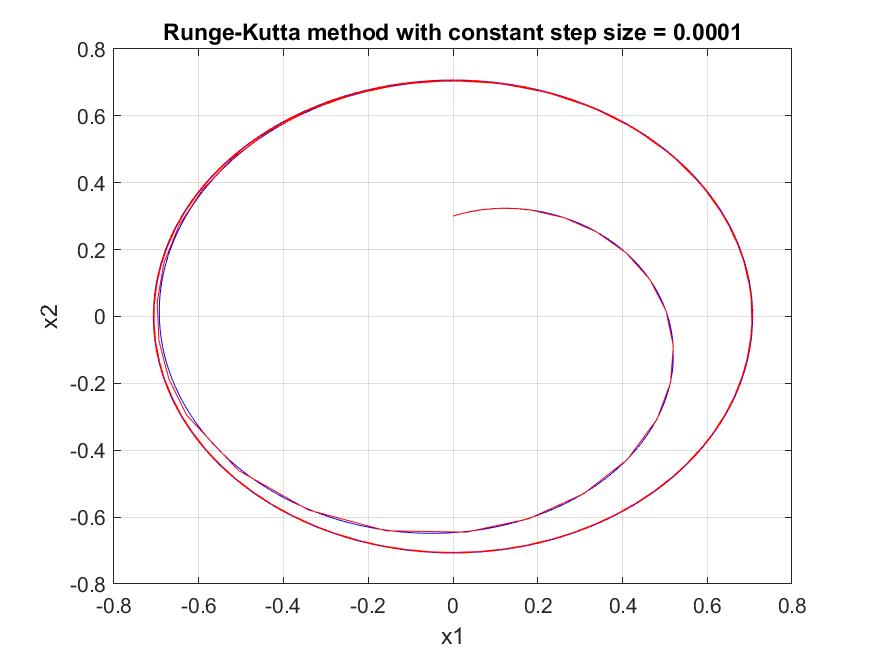


**Conclusion :**  We can see that starting from *h = 0.0125* obtained trajectory does not change a lot(probably if we zoom and compare plots we would see some differences). For that matter, we can conclude that it presents a proper function.

**To test** my implementations of algorithms I used ***ode45*** build-in Matlab function. Next I plot two functions, one obtained from my algorithm and the second one from *ode45* function, and see whether they differ or not. The step size that I choose for my algorithm is equal 0.0001.

*Runga-Kutta method of order 4th, my implementation*  *–* color BLUE

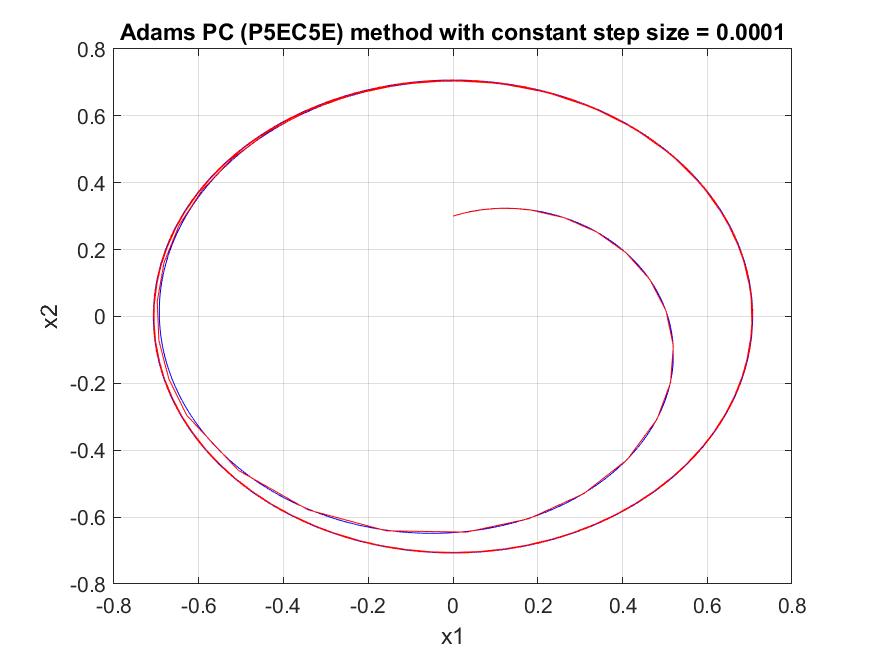
*ode45, build-in Matlab function –* color *RED*



**Conclusion :** We can assume that our algorithm works fine even though there are some small differences which may be erased if we choose smaller step size.

*Adams PC (P5EC5E), my implementation*  *–* color BLUE

*ode45, build-in Matlab function –* color *RED*



**Conclusion :** We can assume that our algorithm works fine even though there are some small differences which may be erased if we choose smaller step size.

As we can see, in each case the methods gave results similar to the results obtained with the command *ode45*. However, in all cases the predictor-corrector method found the result faster. Moreover, each time it generated smaller errors: in the case of the first two at the very beginning they are larger, but remember that the first four points in the Adams PC (P5EC5E)method are calculated using the RK4 method, then the errors stabilize quickly and reach very small values. However, the Adams PC (P5EC5E) method requires more calculation than RK4 due to the double evaluation of the function value. It can be reduced by examining the variability of functions and introducing a variable step (larger for less variable intervals and smaller for intervals of high variability).

We can modify Runge-Kutta method and make algorithm to choose the best fitting next step size.

**Choosing the length of the step size**

The basic issue in the practical implementation of methods for solving differential equations is the question of choosing the step length . When determining the step length, there are two opposing cases :

* If the step decreases, the method error decreases, for the convergent method the error decreases to zero with h going toward zero
* If the step decreases, the number of iterations needed to determine the solution on the given distance [a, b] increases, hence the number of calculations and related numerical errors also increases.

From the above points it appears that there should be an optimal step, for which both method and numerical errors will not be too great.

**Estimating the error value according to the step-doubling rule**

In order to estimate the error, in addition to the step of length h, we additionally carry out two additional steps of each in parallel and exactly the same method.

Starting from the formula :

– for single step

– for double step

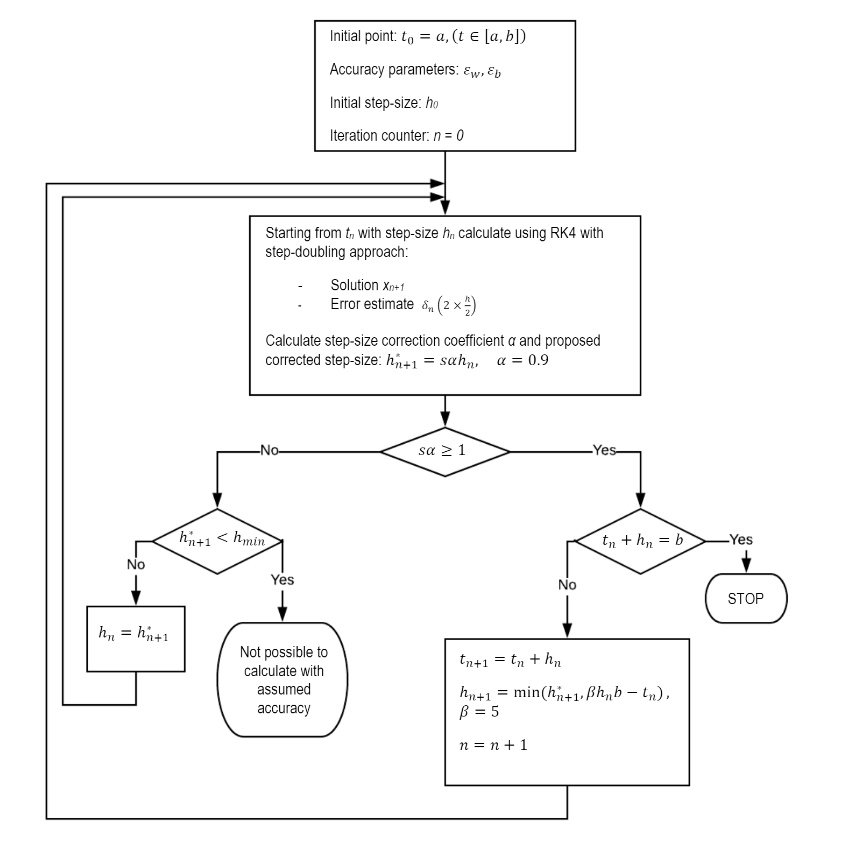
After some transformations we receiving the following error’s formulas :

– estimating error after single step

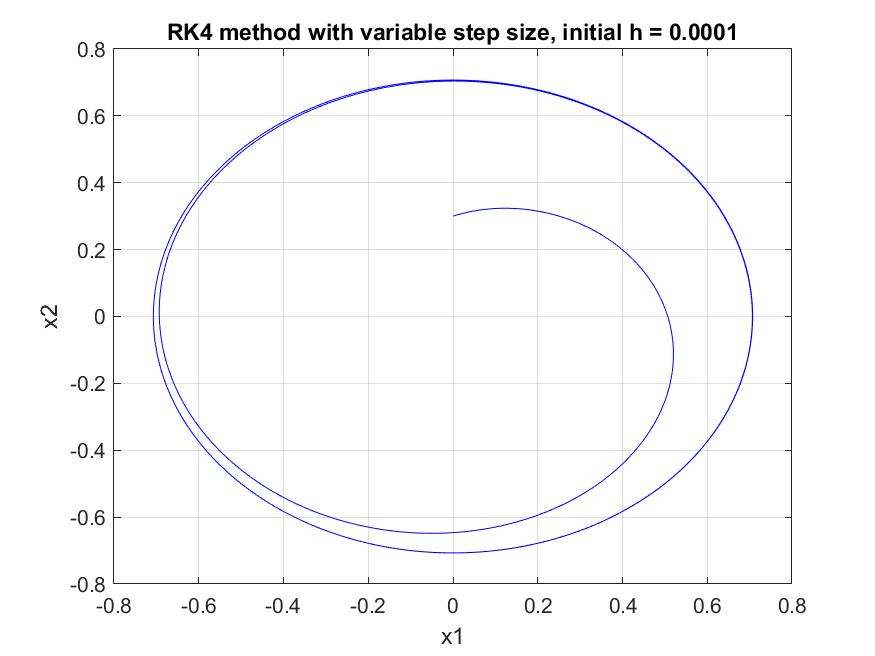
– estimating error after double step

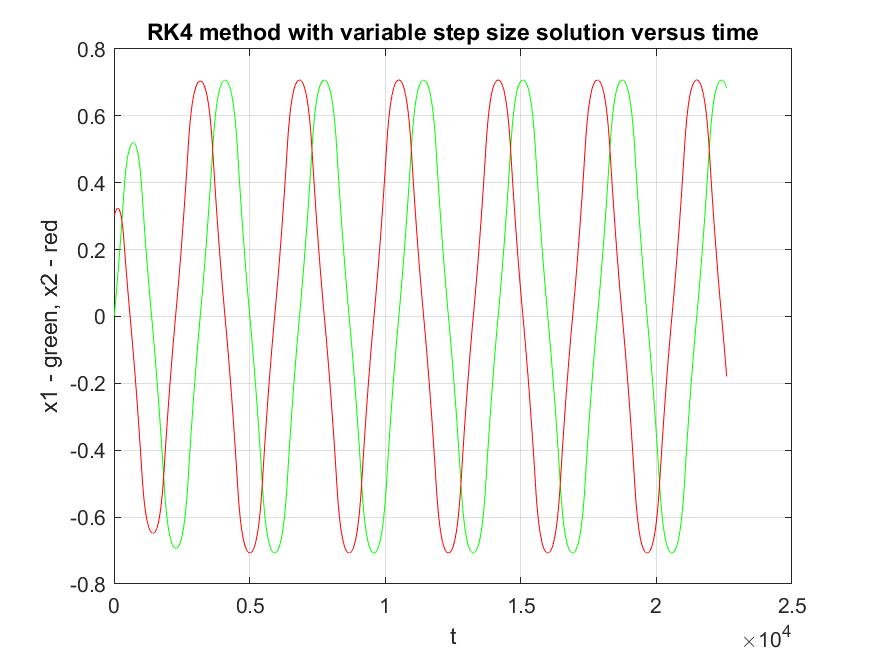
where is a new point received after step of length h, is a new point received after double step of length h.

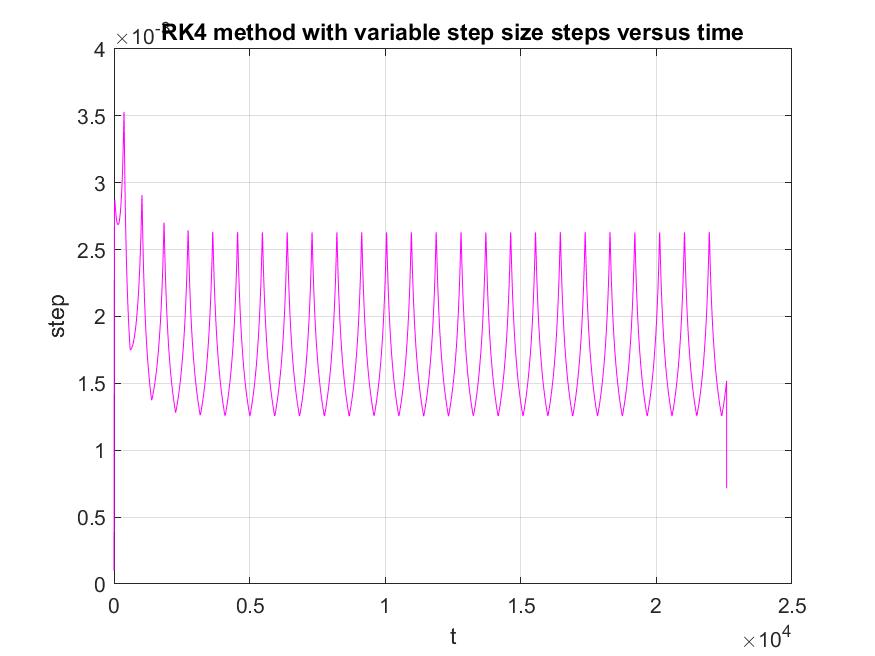
**Flowchart of an algorithm :**

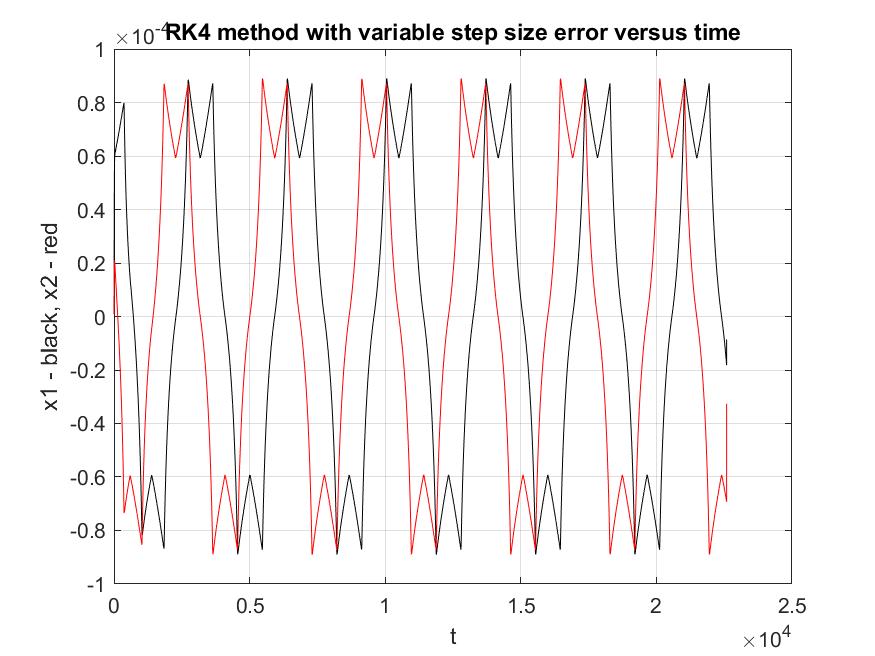


**Result obtained** using Runge-Kutta method of 4th order with variable step size algorithm presents as follow :









**The parameters** hmin, absolute and relative tolerance where chosen using trial and error method. It occurs that it is the best for our problem to assign them as follow :

hmin = 0, because our step size should not be smaller than 0, so it is the safest option

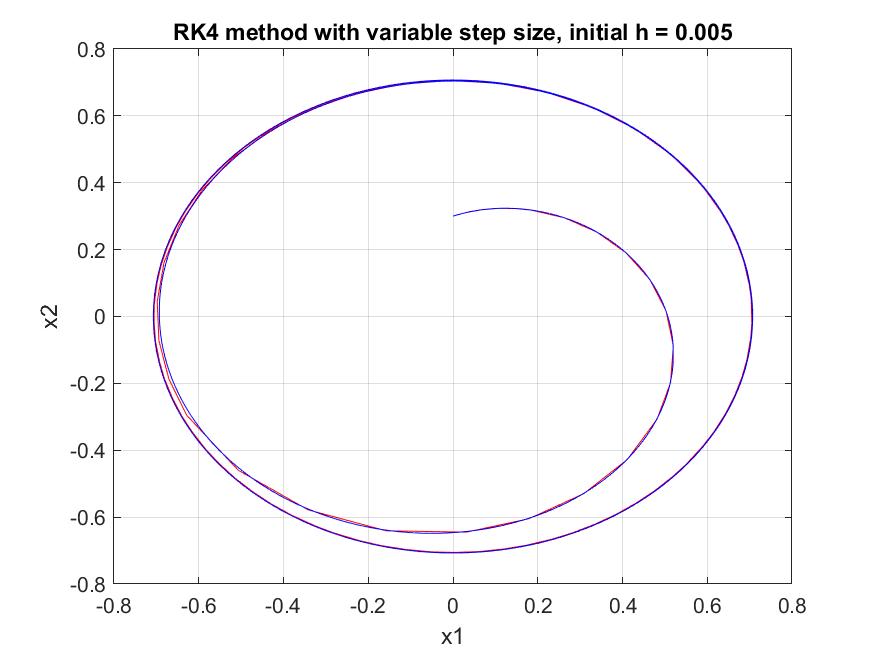
eps\_absolute = 10-4

eps\_relative = 10-4, those configuration allows to get very precise solution as shown above within the sufficient time, because if we decrease both parameters to 10-5 the solution does not differ very much but the period of computation is significantly longer.

**To test** my implementations of algorithms I used ***ode45*** build-in Matlab function. Next I plot two functions, one obtained from my algorithm and the second one from *ode45* function, and see whether they differ or not. The initial step size that I choose for my algorithm is equal 0.005.

*Runga-Kutta method of order 4th variable step size*  *–* color BLUE

*ode45, build-in Matlab function –* color *RED*



**Final conclusions :** This method significantly reduces the number of iterations performed in relation to the method unmodified (the step grows - the number of iterations decreases and, consequently, the number related calculations and numeric errors). However, more iterations are done in each iteration, often repeatedly the step on the corresponding accuracy is changed. The advantage of this method is the fact that automatic step selection depending on the direction of the trajectory, the program itself adapts to the conditions of the function, e.g. in the moments of bending the function increases its accuracy to better approximate it. This method also relieves the user from obligation to determine the optimal step by trial and error, which saves a huge amount of time.

function [a, residuum] = approxUsingSysNormEq(x, y, n)

% approxUsingSysNormEq is a function which computes approximation

% of a function based on

% x - xs of function

% y - values of f(x)

% n - degree of an approximate polynomial

N = size(x, 1);

A = zeros(N, n + 1);

for i = 1:N

for j = 1:n + 1

A(i, j) = x(i, 1) ^ (j - 1);

end

end

a = (A' \* A) \ (A' \* y);

residuum = norm((A' \* y) - (A' \* A) \* a);

end

function [a, residuum] = approxUsingQRdistr(x, y, n)

% approxUsingSysNormEq is a function which computes approximation

% of a function based on

% x - xs of function

% y - values of f(x)

% n - degree of an approximate polynomial

N = size(x, 1);

A = zeros(N, n + 1);

for i = 1:N

for j = 1:n + 1

A(i, j) = x(i, 1) ^ (j - 1);

end

end

[Q, R] = qrmgs(A);

a = R \ Q' \* y;

residuum = norm(R \* a - Q' \* y);

end

function [x1, x2, errX1, errX2] = RK4(fun1, fun2, h, initX1, initX2)

x1(1) = initX1;

x2(1) = initX2;

for i = 1:1:(20 / h)

[k1, k2] = computeKs(fun1, fun2, x1(i), x2(i), h);

x1(i + 1) = x1(i) + h \* (k1(1) + 2 \* k1(2) + 2 \* k1(3) + k1(4)) / 6;

x2(i + 1) = x2(i) + h \* (k2(1) + 2 \* k2(2) + 2 \* k2(3) + k2(4)) / 6;

errX1(i) = (16 / 15) \* abs(x1(i + 1) - x1(i));

errX2(i) = (16 / 15) \* abs(x2(i + 1) - x2(i));

end

function [k1, k2] = computeKs(fun1, fun2, x, y, h)

%first ks

k1(1) = fun1(x, y);

k2(1) = fun2(x, y);

%second ks

k1(2) = fun1(x + 0.5 \* h \* k1(1), y + 0.5 \* h \* k2(1));

k2(2) = fun2(x + 0.5 \* h \* k1(1), y + 0.5 \* h \* k2(1));

%third ks

k1(3) = fun1(x + 0.5 \* h \* k1(2), y + 0.5 \* h \* k2(2));

k2(3) = fun2(x + 0.5 \* h \* k1(2), y + 0.5 \* h \* k2(2));

%fourth ks

k1(4) = fun1(x + h \* k1(3), y + h \* k2(3));

k2(4) = fun2(x + h \* k1(3), y + h \* k2(3));

end

function [x1, x2, errX1, errX2] = adams5(fun1, fun2, h, initX1, initX2)

err\_fac = (863/60480) / ((-95/288) + (863 / 60480));

x1(1) = initX1;

x2(1) = initX2;

betaE = [1901, -2774, 2616, -1274, 251];

betaE = betaE / 720;

betaI = [475, 1427, -798, 482, -173, 27];

betaI = betaI / 1440;

for i = 1:4

[k1, k2] = computeKs(fun1, fun2, x1(i), x2(i), h);

x1(i + 1) = x1(i) + h \* (k1(1) + 2 \* k1(2) + 2 \* k1(3) + k1(4)) / 6;

x2(i + 1) = x2(i) + h \* (k2(1) + 2 \* k2(2) + 2 \* k2(3) + k2(4)) / 6;

end

for i = 6:ceil(20 / h)

sumX1 = 0;

sumX2 = 0;

for j = 1:5

sumX1 = sumX1 + betaE(j) \* fun1(x1(i - j), x2(i - j));

sumX2 = sumX2 + betaE(j) \* fun2(x1(i - j), x2(i - j));

end

tempX1 = x1(i - 1) + h \* sumX1;

tempX2 = x2(i - 1) + h \* sumX2;

sumX1 = 0;

sumX2 = 0;

for j = 1:5

sumX1 = sumX1 + betaI(j + 1) \* fun1(x1(i - j), x2(i - j));

sumX2 = sumX2 + betaI(j + 1) \* fun2(x1(i - j), x2(i - j));

end

x1(i) = x1(i - 1) + h \* sumX1 + h \* betaI(1) \* fun1(tempX1, tempX2);

x2(i) = x2(i - 1) + h \* sumX2 + h \* betaI(1) \* fun2(tempX1, tempX2);

errX1 = err\_fac \* (tempX1 - x1(i));

errX2 = err\_fac \* (tempX2 - x2(i));

end

end

function [x1values, x2values, errors, steps] = RK4variable(fun1, fun2, x1, x2, h, epsr, epsa)

% x1 - first value of x1

% x2 - first value of x2

% h - step

% epsr - relative epsilon

% epsa - absolute epsilon

i = 1;

a = 0;

x1values(1) = x1;

x2values(1) = x2;

steps(i) = h;

while (a < 20)

%next points

[k1, k2] = computeKs(fun1, fun2, x1, x2, h);

tempx1 = x1;

tempx2 = x2;

x1 = x1 + h \* (k1(1) + 2 \* k1(2) + 2 \* k1(3) + k1(4)) / 6;

x2 = x2 + h \* (k2(1) + 2 \* k2(2) + 2 \* k2(3) + k2(4)) / 6;

x1values(i + 1) = x1;

x2values(i + 1) = x2;

%first half-step

h = 0.5 \* h;

[k1, k2] = computeKs(fun1, fun2, x1, x2, h);

tmp1 = x1 + h \* (k1(1) + 2 \* k1(2) + 2 \* k1(3) + k1(4)) / 6;

tmp2 = x2 + h \* (k2(1) + 2 \* k2(2) + 2 \* k2(3) + k2(4)) / 6;

%second half-step

[k1, k2] = computeKs(fun1, fun2, tmp1, tmp2, h);

tmp1 = tmp1 + h \* (k1(1) + 2 \* k1(2) + 2 \* k1(3) + k1(4)) / 6;

tmp2 = tmp2 + h \* (k2(1) + 2 \* k2(2) + 2 \* k2(3) + k2(4)) / 6;

h = 2 \* h;

errors(i, 1) = (tmp1 - x1) / 15;

errors(i, 2) = (tmp2 - x2) / 15;

eps1 = abs(tmp1) \* epsr + epsa;

eps2 = abs(tmp2) \* epsr + epsa;

alpha1 = (eps1 / abs(errors(i, 1)))^(1/5);

alpha2 = (eps2 / abs(errors(i, 2)))^(1/5);

alpha = min(alpha1, alpha2);

hnew = 0.9 \* alpha \* h;

if (0.9 \* alpha >= 1)

if (a + h >= 20)

break;

else

a = a + h;

h = min([hnew, 5 \* h, 20 - a]);

i = i + 1;

steps(i) = h;

continue;

end

else

if (hnew < 10^-10)

error('Cant solve with this epsilon');

else

h = hnew;

end

end

i = i + 1;

steps(i) = h;

end