Numerical Methods
Project C No. 29

Michał Tchórzewski

Warsaw, 19 January 2019

Task 1

1.1 Aim

The aims of task are:
1. Finding function formula based on given points (see Table 1), with use of approximation method called the least-squares.
2. Finding function formula based on given points (see Table 1), with use of approximation method called the least-squares with use of QR distribution.
3. Results comparison obtained by both methods.

Measurement	**x_i**	**y_i**
1	-5	23.4523
2	-4	11.9631
3	-3	4.4428
4	-2	1.1010
5	-1	-1.6826
6	0	-1.2630
7	1	-0.0357
8	2	-1.3156
9	3	-3.4584
10	4	-8.4294
11	5	-18.4654

Table 1. Task 1 experimental measurements

1.2 Approximation

The aim of the approximation is to find a simpler function, from a chosen class of approximating functions, which is appropriately close to origin function at given points (measurements).

1.3 Method

The approximation problem can be defined in the following way : to find a function F^* belonging to set X_n 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 $a_0,...,a_n$ of F such that the norm ||f – F|| is minimised.
To perform least-square approximation, we must define A as a matrix Nxn, where N – number of samples, n – degree of polynomial, for every

$$A_{(i,\ j)} = x_j^{i-1}$$

$$i = 1,2,3,...,n;\ j = 1,2,3,...,N$$

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 :

$$A^T Aa = A^T y$$

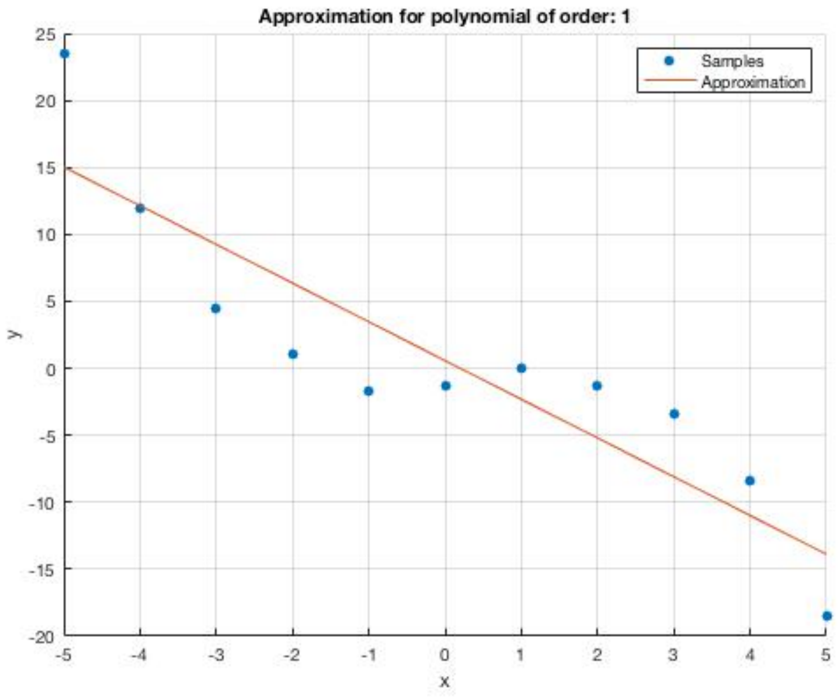
Using and solving system resulting in QR factorisation :

$$A = QR \ Ra = Qy$$

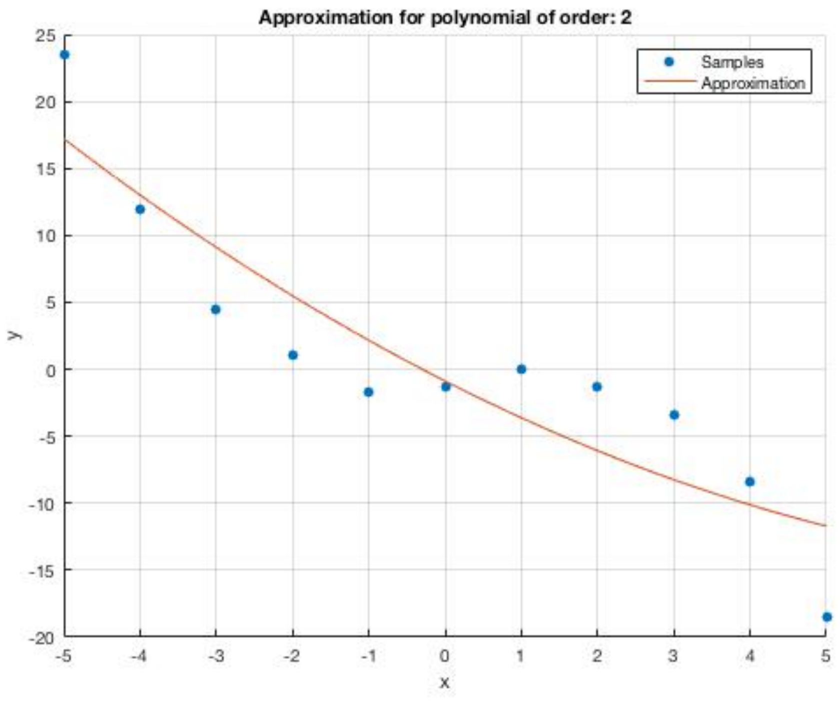
1.4 Graphs

1.4.1 System of Normal Equations Graphs

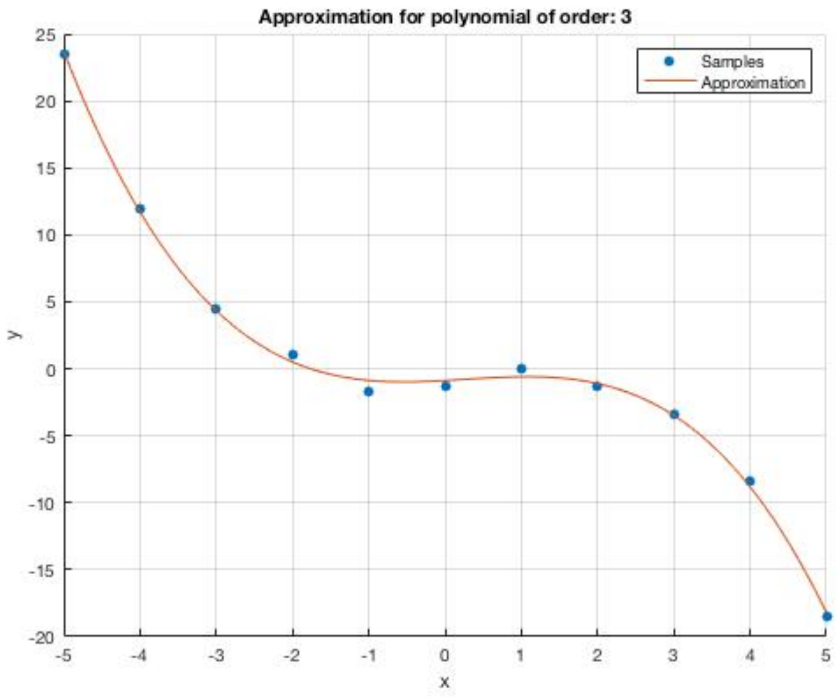
The Least-Squares Approximation and given samples graph. In order of transparency, only the most crucial graphs were enclosed i.e. ones which differentiate the most from each others.



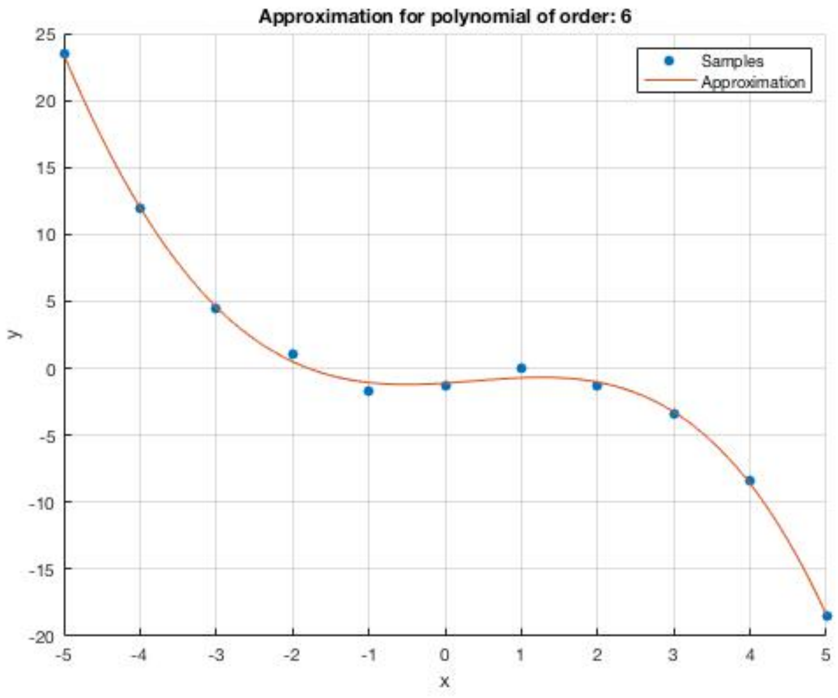
$$f(x) = -2.8913x + 0.5736$$



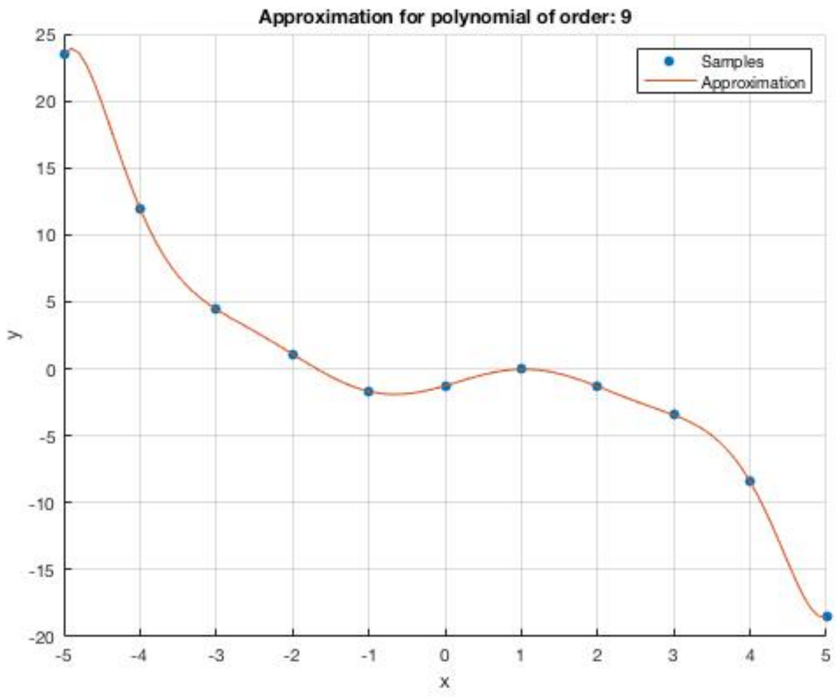
$$f(x) = 0.1450x^2 - 2.8913x - 0.8764$$



$$f(x) = -0.1804x^3 + 0.1450x^2 + 0.3202x - 0.8764$$



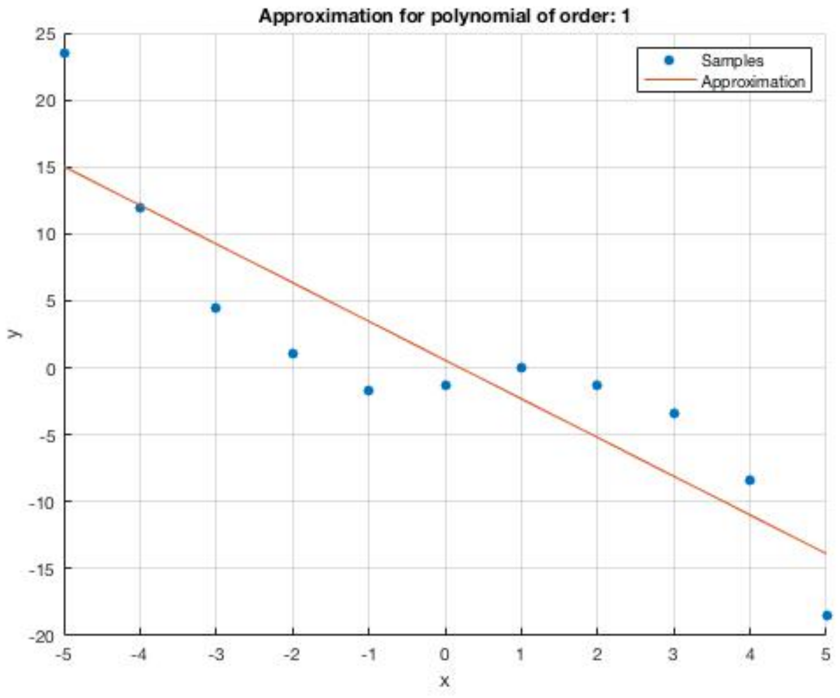
$$f(x) = -0.0001x^6 + 0.0002x^5 - 0.0024x^4 - 0.1877x^3 + 0.2202x^2 + 0.3637x - 1.1059$$



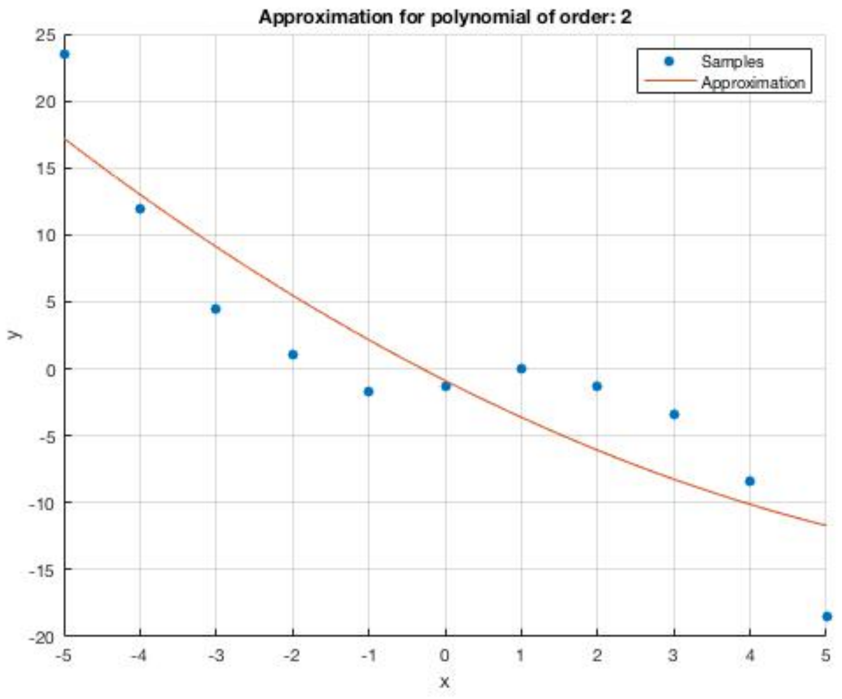
$$f(x) = 0.0001x^9 - 0.0001x^8 - 0.0068x^7 + 0.0039x^6 + 0.1182x^5 - 0.0581x^4 - 0.9346x^3 + 0.4654x^2 + 1.6465x - 1.2669$$

1.4.2 QR distribution Graph

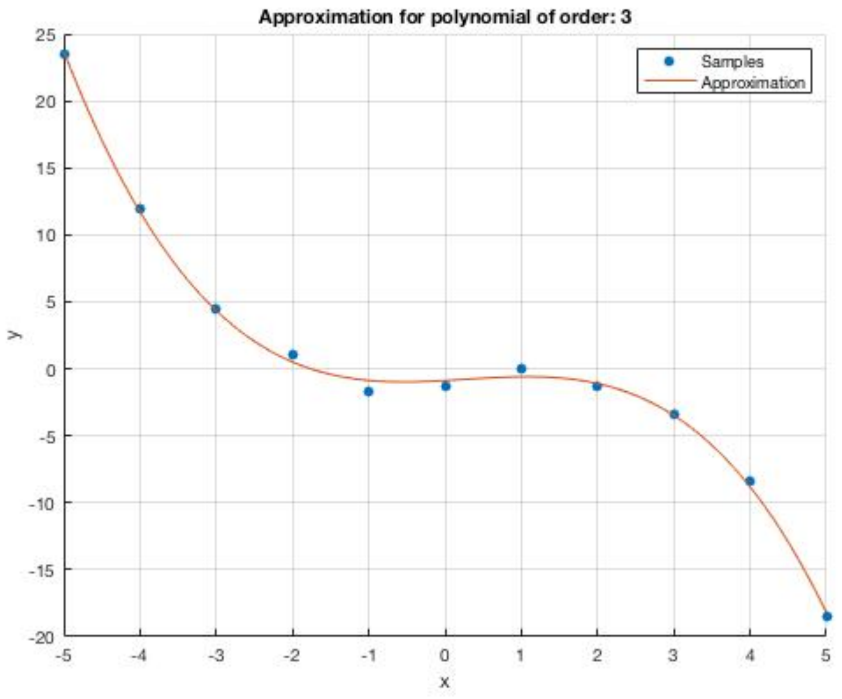
The Least-Squares Approximation with use of QR distribution and given samples graph. In order of transparency, only the most crucial graphs were enclosed i.e. ones which differentiate the most from each others.



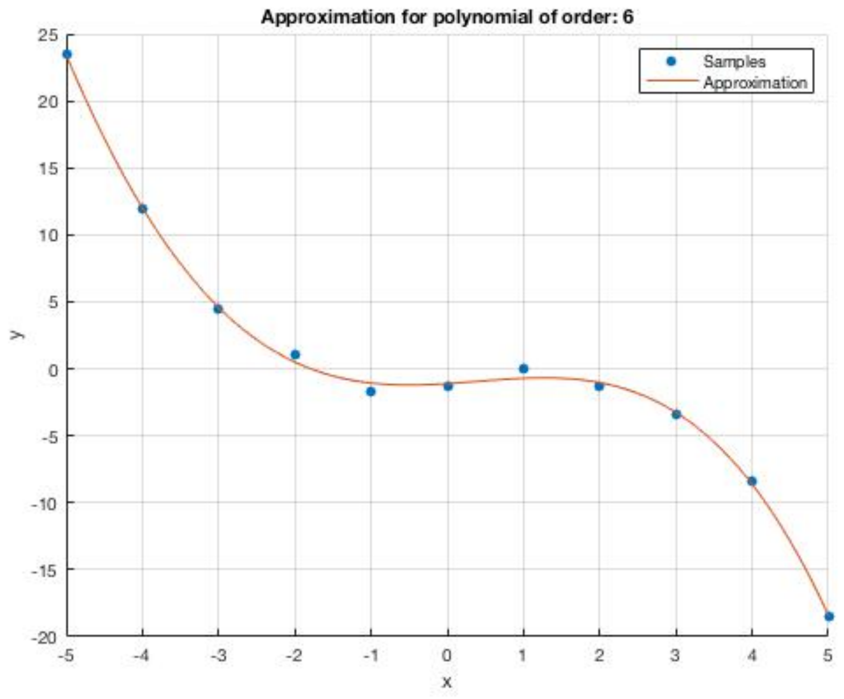
$$f(x) = -2.8913x + 0.5736$$



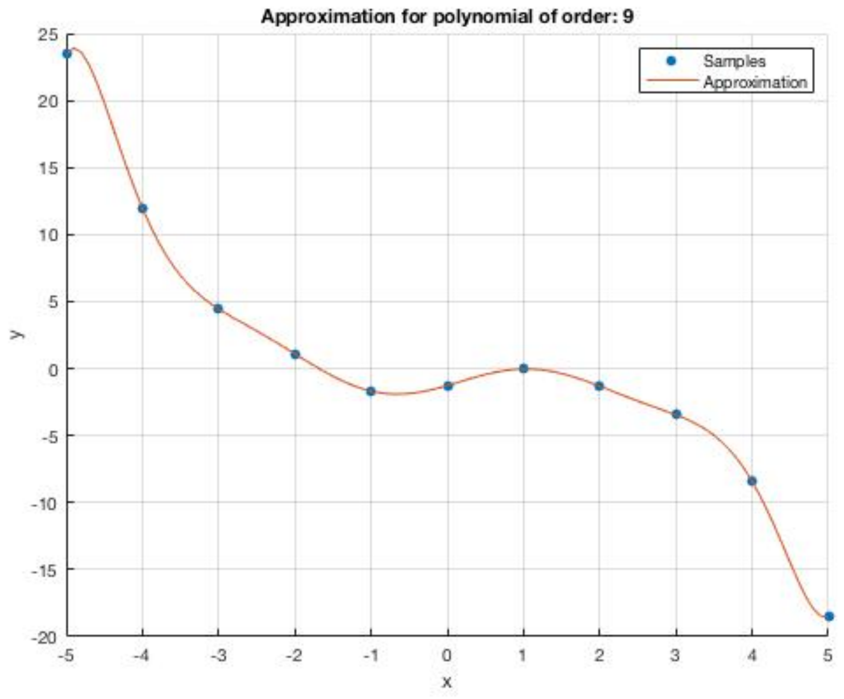
$$f(x) = 0.1450x^2 - 2.8913x - 0.8764$$



$$f(x) = -0.1804x^3 + 0.1450x^2 + 0.3202x - 0.8764$$



$$f(x) = -0.0001x^6 + 0.0002x^5 - 0.0024x^4 - 0.1877x^3 + 0.2202x^2 + 0.3637x - 1.1059$$



$$f(x) = 0.0001x^9 - 0.0001x^8 - 0.0068x^7 + 0.0039x^6 + 0.1182x^5 - 0.0581x^4 - 0.9346x^3 + 0.4654x^2 + 1.6465x - 1.2669$$

1.5 Table of residuum error for both methods

Table shows residuum error using system of normal equations and QR distribution, for various polynomial order.

Polynomial Order	System of Normal Equations	QR Distribution
0	0	2.22044604925031e-16
1	0	3.55964580964350e-15
2	6.35776948178562e-14	3.55964580964350e-15
3	9.09940416490879e-13	2.22044604925031e-16
4	1.28623078181897e-12	0
5	2.91180383313479e-11	0
6	2.91606328102988e-11	0
7	4.66581420149271e-10	2.42365144572834e-16
8	1.47746791894050e-09	2.91075020431387e-15
9	7.45713752940529e-08	6.65674315973653e-15
10	9.10187529783401e-08	1.54856214187050e-14

Range of polynomial order (0-10) is set by system of normal equation for which algorithm can calculate accurate result. Thus, for polynomial order greater than 9 matrix A become close to singular or badly scaled. For QR distribution result is decent even for much higher order.

1.6 Conclusions

Both the normal equation and the QR distribution method well represent the polynomial defined on the given measurements. However, along with the increase in the polynomial order, numerical errors increase (matrix A loses a good condition). The QR distribution is much more accurate method of approximation. Nevertheless, for small degree of polynomial both algorithms output decent results. It should also be noted that by increasing the polynomial row, at some point, it stops approximating the original function, and begins with the one that most closely matches the collected data. Taking into account the values of approximation errors, it can be noticed that as the polynomial row increases, the approximation error decreases (new function crosses marked measurements), which indicates that the approximating function strive to value of collected samples.

Task 2a

2.1 Aim

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

$$x_1' = x_2 + x_1(0.5 - x_1^2 - x_2^2)$$
$$x_2' = -x_1 + x_2(0.5 - x_1^2 - x_2^2)$$

on the interval [0, 20]. The following initial conditions are given : $x_1(0) = 8$, $x_2(0) = 7$. 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 until an „optimal" constant step size is found, i.e., when its decrease 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.

2.2 Runge-Kutta method of 4th order (RK4)

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

$$y_{n+1} = y_n + \frac{1}{6} h\,(k_1 + 2k_2 + 2k_3 + k_4)$$
$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1)$$
$$k_3 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2)$$
$$k_4 = f(x_n + h, y_n + hk_3)$$

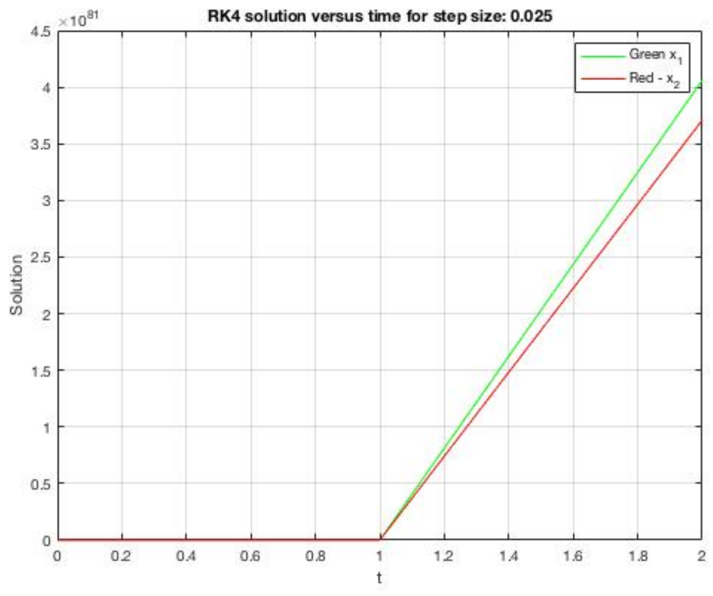
The coefficient k_1 represents the derivative at (x_n,y_n). The value k_2 is calculated as in the modified Euler's method – as a derivative of the solution calculated by the standard Euler's method at the midpoint $(x_n + 1/2\ h, y_n + 1/2\ hk_1)$, the dashed tangent line correspond to this derivative. Next, the value k_3 is calculated similarly as it was for k_2, but this time at the point $(x_n + 1/2\ h, y_n + 1/2\ hk_2)$ - i.e., with a tangent line corresponding to k_2. Finally, we start with this line from the initial point until the endpoint $(x_n + h)$, i.e., the derivative k_4 of a solution at the point $(x_n + h, y_n + hk_3)$-i.e over the one step interval: 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 :

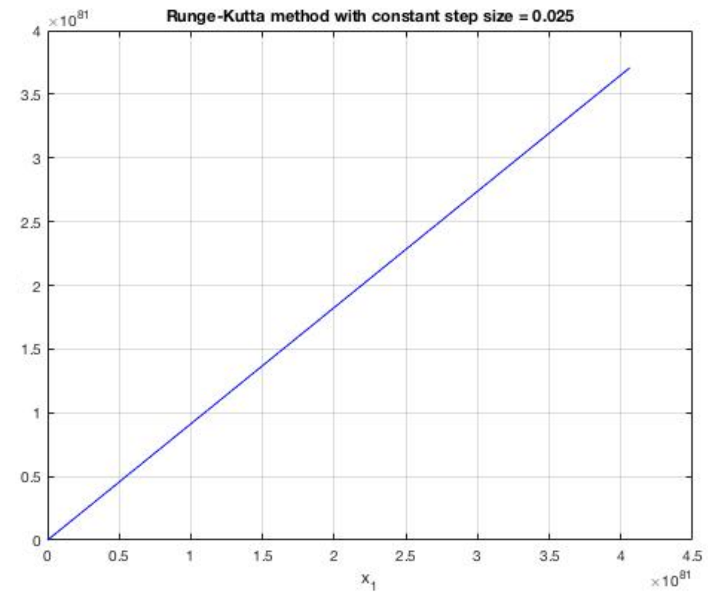
$$\delta_n(h) = \frac{2^p}{2^p - 1}({y_n}^{(2)} - {y_n}^{(1)})$$

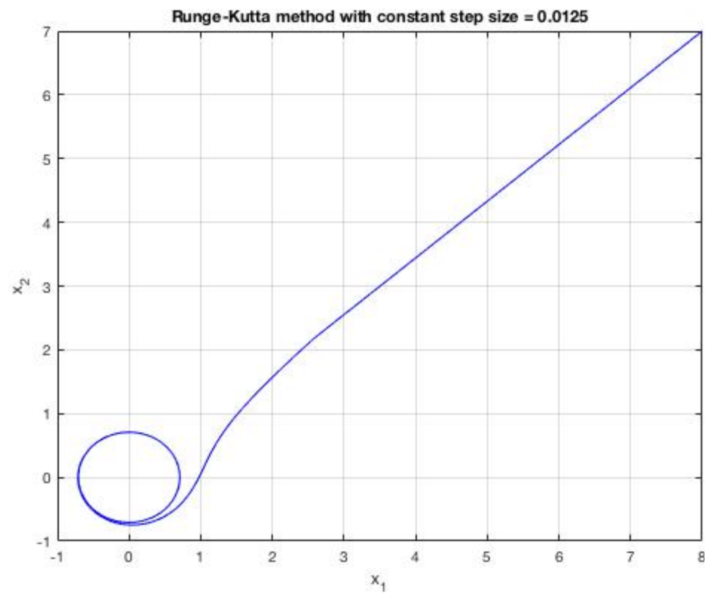
where, $y_n^{(1)}$ – new point obtained from the step of length h, $y_n^{(2)}$ – new point obtain from two additional steps of length 0.5h, p – order of method.

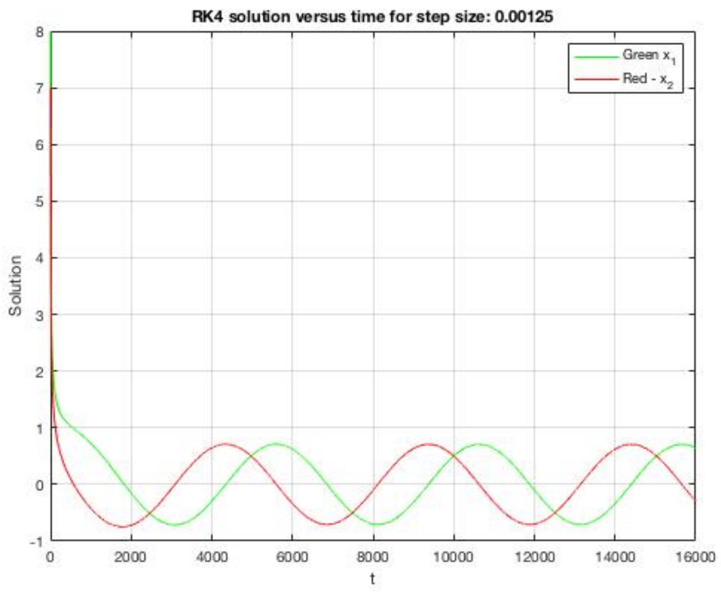
2.2.1 Solution RK4, a constant step size

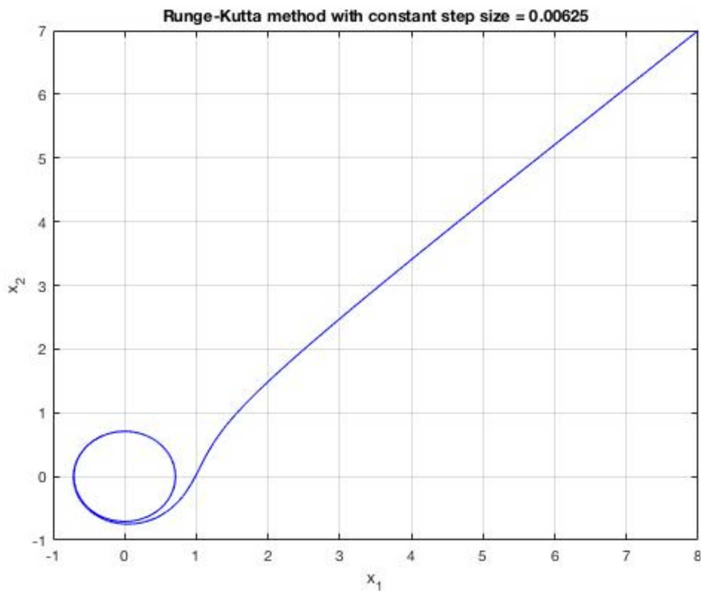
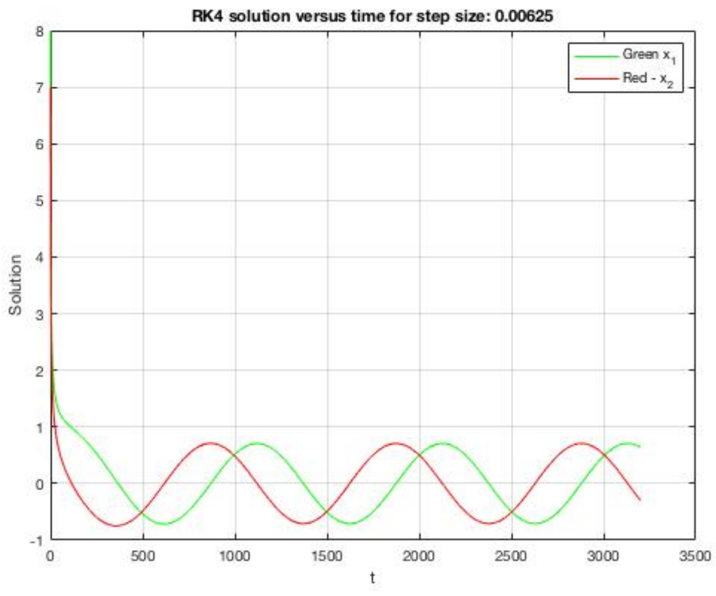
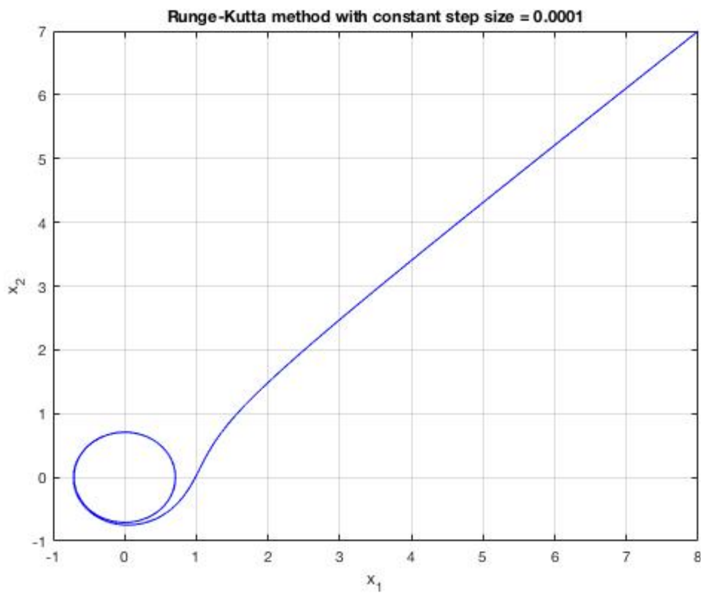
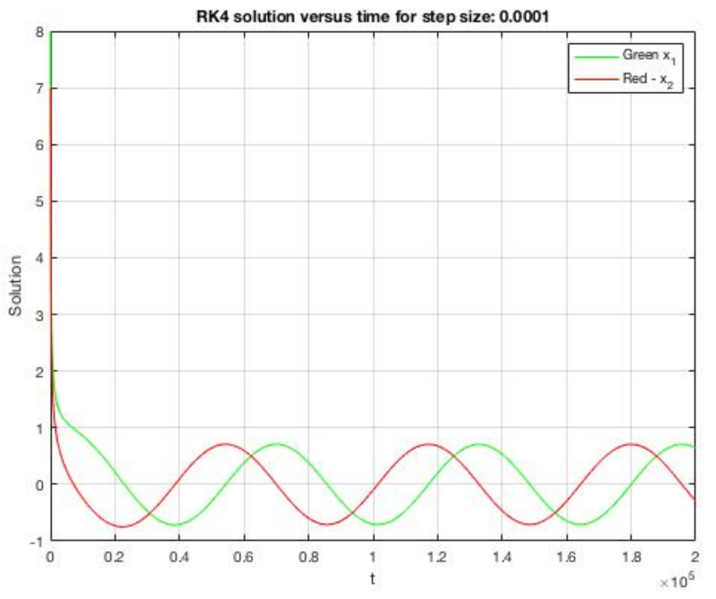
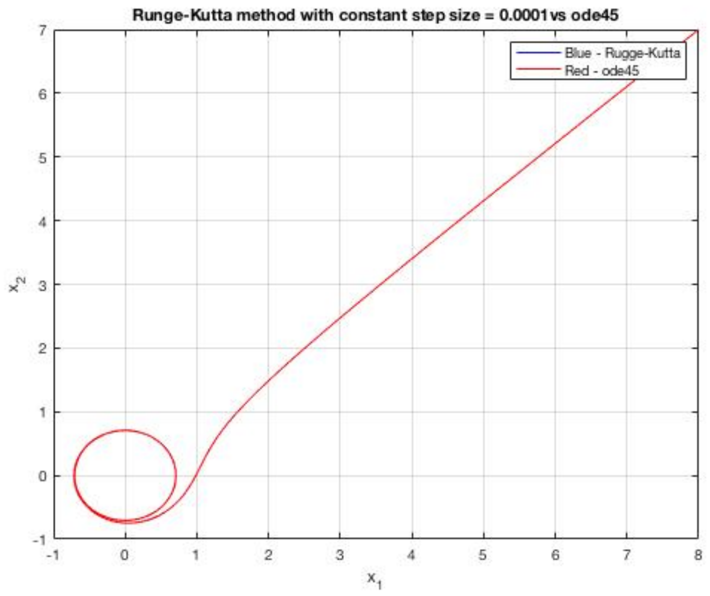
Results for different step size obtained using Runge-Kutta algorithm of 4th order are presented below.:










Runge-Kutta method with constant step size = 0.00625
x_1
x_2

RK4 solution versus time for step size: 0.00625
Green x_1
Red - x_2
Solution
t

Runge-Kutta method with constant step size = 0.0001
x_1
x_2

RK4 solution versus time for step size: 0.0001
Green x_1
Red - x_2
Solution
t
$\times 10^5$

Runge-Kutta method with constant step size = 0.0001vs ode45
Blue - Rugge-Kutta
Red - ode45
x_1
x_2

2.2.2 Conclusion

We can see that starting from h = 0.00625 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.

Comparing obtained result versus our result, 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.

2.3 Adams PC (P_5EC_5E)

The function f (x , y (x)) is now replaced by an interpolation polynomial W*(x) of order k, calculated at the points x_n , ... , x_{n-k} with the corresponding solution values $y(x_{n-1}) \approx y_{n-j}$. Reasoning in the same way as it was done in the case of the explicit methods, we finally get:

$$y_n = y_{n-1} + h\sum_{j=0}^{k} \beta_j^* f(x_{n-j}, y_{n-j}) = y_{n-1} + \beta_0^* f(x_n, y_n) + h\sum_{j=1}^{k} \beta_j^* f(x_{n-j}, y_{n-j})$$

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

The predictor-corrector method P_kEC_kE:

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

P: $\quad y_n^{[0]} = y_{n-1} + h\,\sum_{j=1}^{k} \beta_j f_{n-j}$

E: $\quad f_n^{[0]} = f(x_n, y_n^{[0]})$

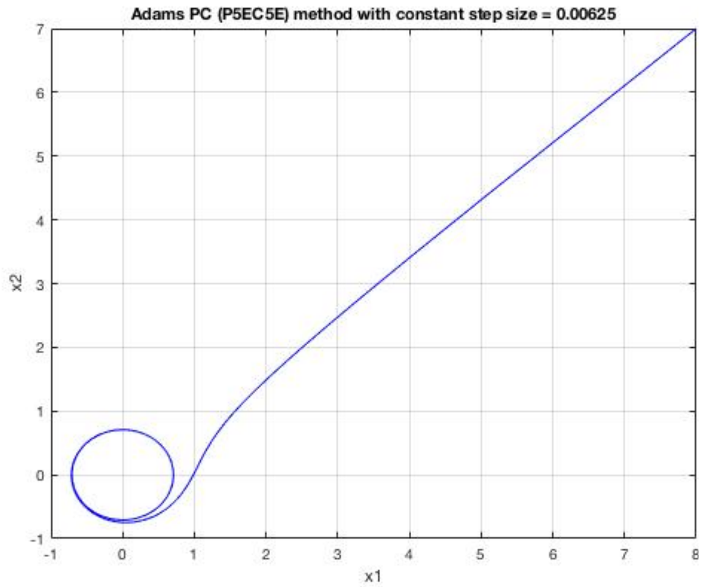
K: $\quad y_n = y_{n-1} + h\,\sum_{j=1}^{k-1} \beta_j^* f_{n-j} + h\,\beta_0^*\, f_n^{[0]}$

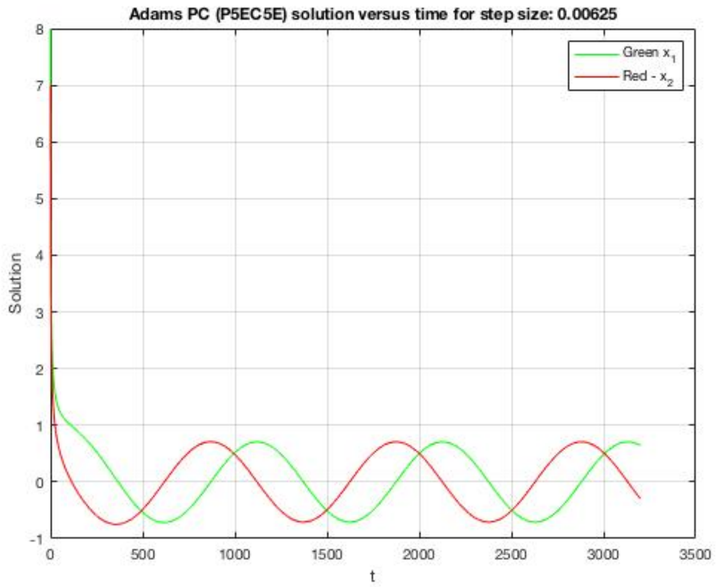
E: $\quad f_n = f(x_n, y_n)$

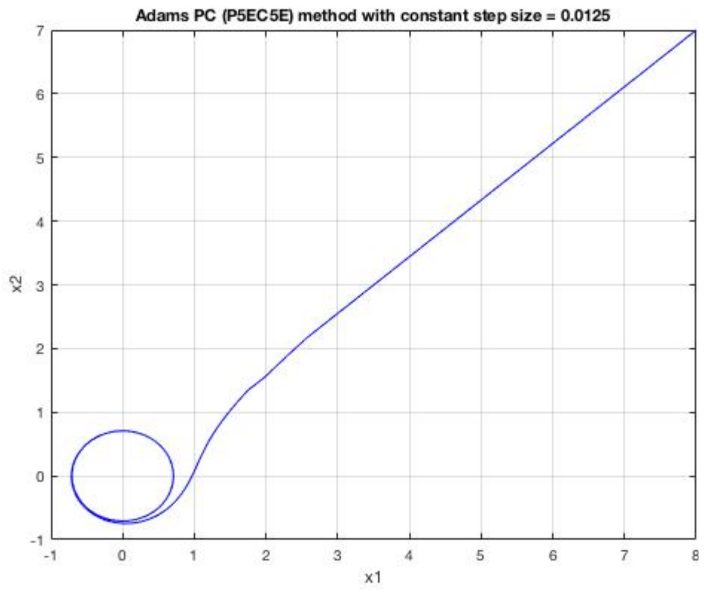
The error approximation is calculated by the formula:

$$\delta_n(h_{n-1}) = -0.0452(y_n^{[0]} - y_n)$$

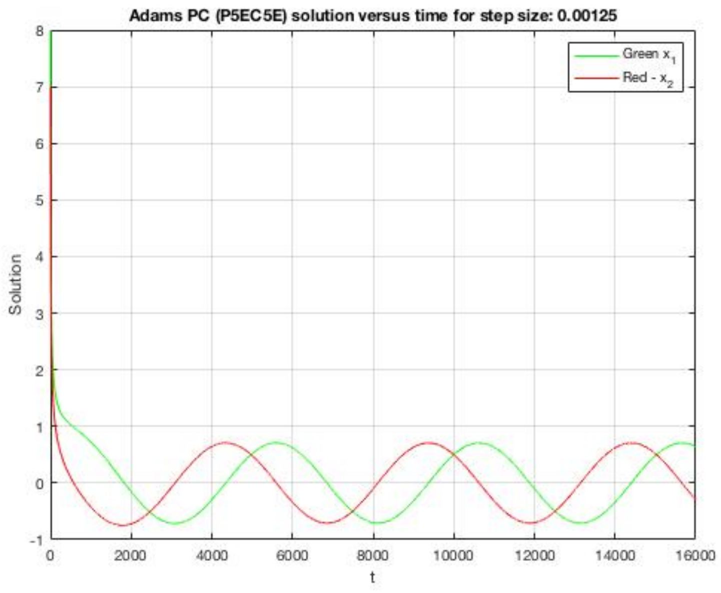
2.3.1 Solution Adams PC (P_5EC_5E)



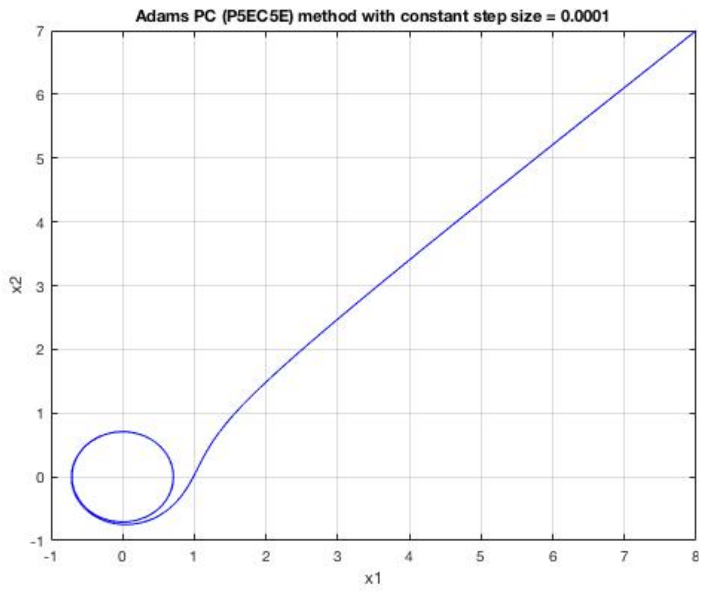




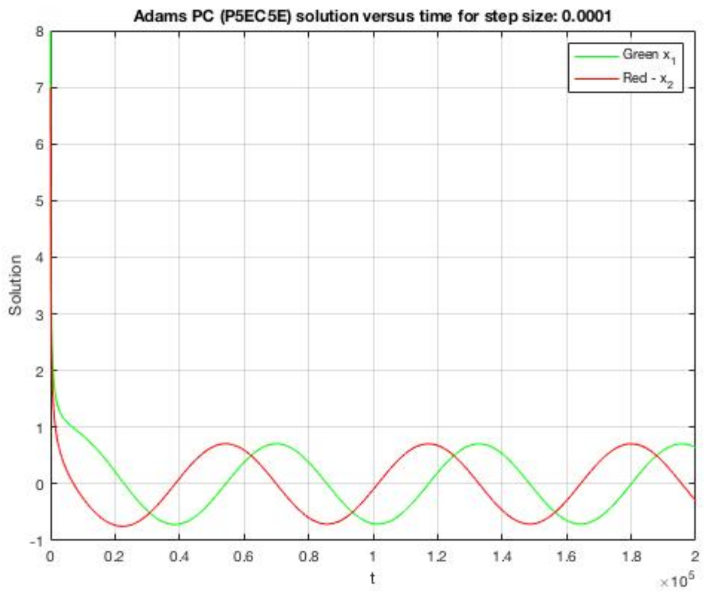
Adams PC (P5EC5E) method with constant step size = 0.0125
x1
x2



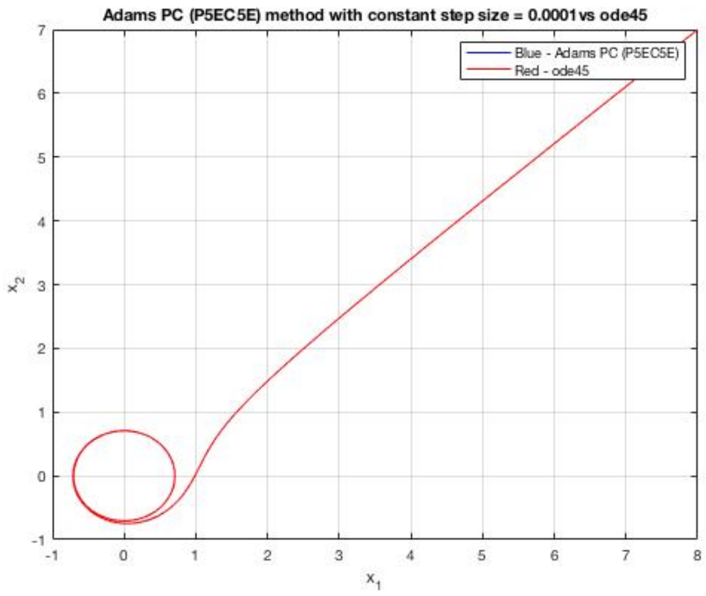
Adams PC (P5EC5E) solution versus time for step size: 0.00125
Green x_1
Red - x_2
Solution
t



Adams PC (P5EC5E) method with constant step size = 0.0001
x1
x2



Adams PC (P5EC5E) solution versus time for step size: 0.0001
Green x_1
Red - x_2
Solution
t
$\times 10^5$



Adams PC (P5EC5E) method with constant step size = 0.0001 vs ode45
Blue - Adams PC (P5EC5E)
Red - ode45
x_1
x_2

2.3.2 Conclusion

We can see that starting from h = 0.00625 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.

Comparing obtained result versus our result, 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.

2.4 General Remarks Task 2 a

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 (P_5EC_5E) method are calculated using the RK4 method, then the errors stabilize quickly and reach very small values. However, the Adams PC (P_5EC_5E) method requires more calculation than RK4 due to the double evaluation of the function value. It can be reduced by examining the variability of

Task 2b

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

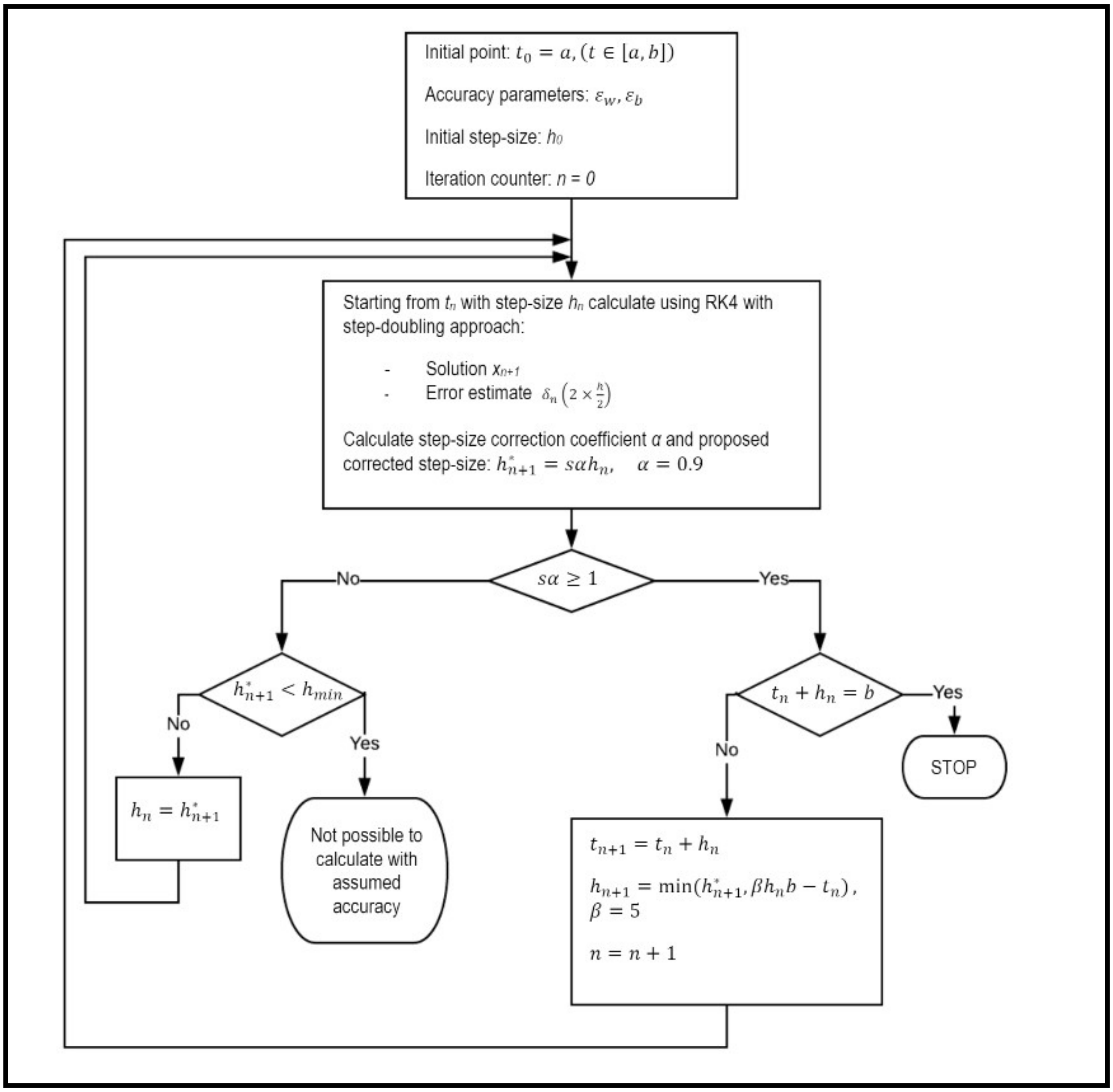
We can modify Runge-Kutta method and make algorithm to choose the best fitting next step size.

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 h. When determining the step length, there are two opposing cases :

1. If the step h_n decreases, the method error decreases, for the convergent method the error decreases to zero with h going toward zero
2. If the step h_n 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.

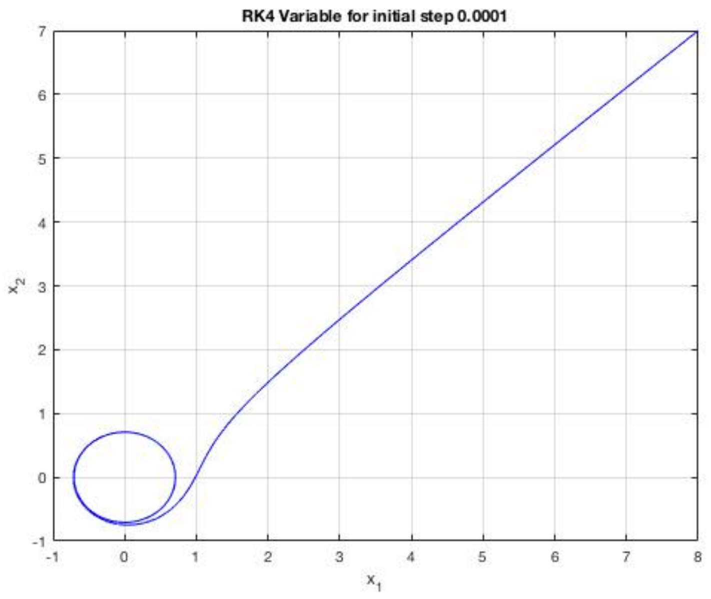

Initial point: $t_0 = a, (t \in [a, b])$
Accuracy parameters: $\varepsilon_w, \varepsilon_b$
Initial step-size: h_0
Iteration counter: $n = 0$
Starting from t_n with step-size h_n calculate using RK4 with step-doubling approach:
- Solution x_{n+1}
- Error estimate $\delta_n\left(2 \times \frac{h}{2}\right)$
Calculate step-size correction coefficient α and proposed corrected step-size: $h^*_{n+1} = s\alpha h_n, \quad \alpha = 0.9$
$s\alpha \geq 1$
No
Yes
$h^*_{n+1} < h_{min}$
No
Yes
$h_n = h^*_{n+1}$
Not possible to calculate with assumed accuracy
$t_n + h_n = b$
Yes
No
STOP
$t_{n+1} = t_n + h_n$
$h_{n+1} = \min(h^*_{n+1}, \beta h_n b - t_n)$, $\beta = 5$
$n = n + 1$

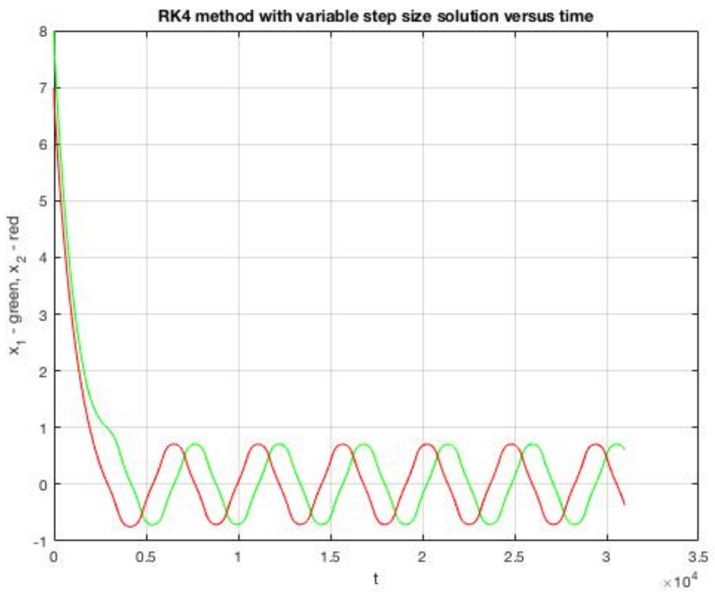
The parameters hmin, absolute and relative tolerance where chosen using trial and error method. It occurs that it is the best for our problem to assign them as follow :

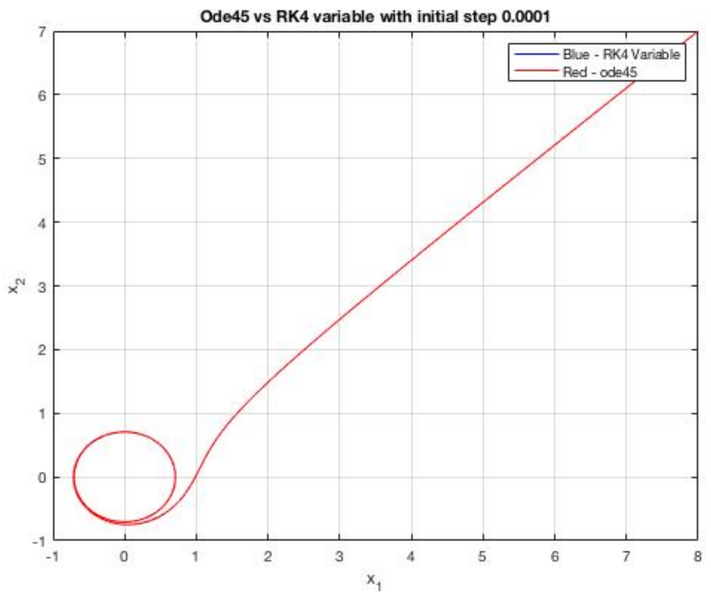
hmin = 10^{-10}, because if we choose smaller the time performance will not be sufficient.

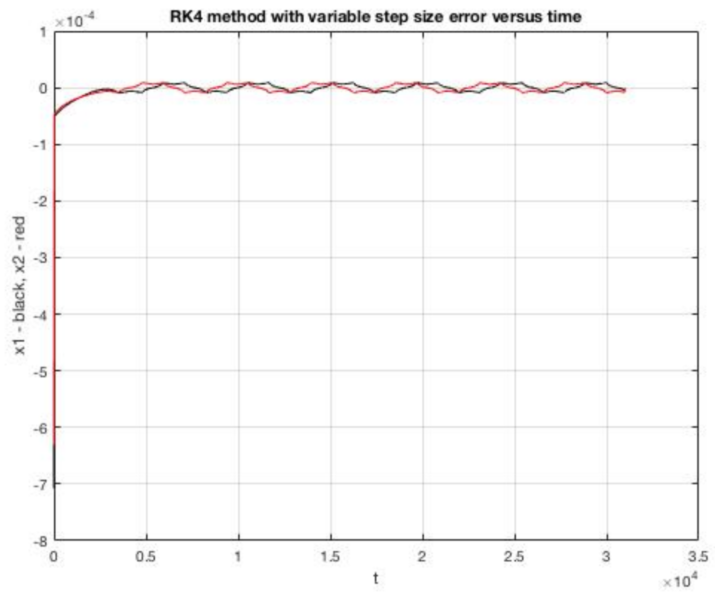
eps_absolute = 10^{-5} eps_relative = 10^{-5}, those configuration allows to get very precise solution as shown above within the sufficient time, because if we decrease both parameters to 10^{-6} the solution does not differ very much but the period of computation is significantly longer.

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









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.

3. Appendix

3.1 Least-Squares Approximation code

```
function [a, residuum] = ls_approximation(n, x, y)
    % n - polynomial order
    % x - x samples coordinate vector
    % y - f(x) values vector

    % F(x) = a_0*x^0 + a_1*x^1 + a_2*x^2 + ... + a_n*x^n

    [~, m] = size(x); % m - number of samples

    % Gram's Matrix(set of normal equations)
    A = zeros(m, n+1);

    for i = 1:m
        for j = 1:(n+1)
           A(i,j) = x(1, i)^(j-1);
        end
    end

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

end
```

3.2 Least-Squares Approximation with QR Distribution code

```
function [a, residuum] = ls_approximationQR(n, x, y)
    % n - polynomial order
    % x - x samples coordinate vector
    % y - f(x) values vector

    % F(x) = a_0*x^0 + a_1*x^1 + a_2*x^2 + ... + a_n*x^n

    [~, m] = size(x); % m - number of samples

    % Gram's Matrix(set of normal equations)
    A = zeros(m, n+1);

    for i = 1:m
        for j = 1:(n+1)
           A(i,j) = x(1, i)^(j-1);
        end
    end

    % R*a = QT*y
    [Q, R] = mgs(A);
    a = R \ (Q' * y');
    residuum = norm((R * a) - (Q' * y'));
    a = flipud(a);


end
```

3.2.1 Modified Gram-Schmidt algorithm

```
function [Q,R] =  mgs(X)
    % Modified Gram-Schmidt.  [Q,R] = mgs(X);
    % G. W. Stewart, "Matrix Algorithms, Volume 1", SIAM, 1998.
    [n,p] = size(X);
    Q = zeros(n,p);
    R = zeros(p,p);
    for k = 1:p
        Q(:,k) = X(:,k);
        for i = 1:k-1
            R(i,k) = Q(:,i)'*Q(:,k);
            Q(:,k) = Q(:,k) - R(i,k)*Q(:,i);
        end
        R(k,k) = norm(Q(:,k))';
        Q(:,k) = Q(:,k)/R(k,k);
    end
end
```

3.3 Runge-Kutt method

```
dx_1 = @(x_1, x_2) x_2 + x_1 * (0.5 - x_1^2 - x_2^2);
dx_2 = @(x_1, x_2) -x_1 + x_2 * (0.5 - x_1^2 - x_2^2);

%initial conditions
init_x_1 = 0.8;
init_x_2 = 0.7;

h = 0.0001; % step

x_1 = zeros(1, 20/h+1);
x_2 = zeros(1, 20/h+1);
err_x_1 = zeros(1, 20/h+1);
err_x_2 = zeros(1, 20/h+1);
x_1(1) = init_x_1;
x_2(1) = init_x_2;


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

    [k_1, k_2] = computeKs(dx_1, dx_2, x_1(i), x_2(i), h);

    x_1(i+1) = x_1(i) + h * (k_1(1) + 2 * k_1(2) + 2 * k_1(3) + k_1(4)) / 6;
    x_2(i+1) = x_2(i) + h * (k_2(1) + 2 * k_2(2) + 2 * k_2(3) + k_2(4)) / 6;

    err_x_1(i) = (16/15)*abs(x_1(i+1) - x_1(i));
    err_x_2(i) = (16/15)*abs(x_2(i+1) - x_2(i));

end
```

3.3.1 Compute Ks

```
function [k_1, k_2] = computeKs(fun_1, fun_2, x, y, h)

    %1st Ks
    k_1(1) = fun_1(x, y);
    k_2(1) = fun_2(x, y);

    %2nd Ks
    k_1(2) = fun_1(x + 0.5 * h * k_1(1), y + 0.5 * h * k_2(1));
    k_2(2) = fun_2(x + 0.5 * h * k_1(1), y + 0.5 * h * k_2(1));

    %3rd Ks
    k_1(3) = fun_1(x + 0.5 * h * k_1(2), y + 0.5 * h * k_2(2));
    k_2(3) = fun_2(x + 0.5 * h * k_1(2), y + 0.5 * h * k_2(2));

    %4th Ks
    k_1(4) = fun_1(x + h * k_1(3), y + h * k_2(3));
    k_2(4) = fun_2(x + h * k_1(3), y + h * k_2(3));

end
```

3.4 Adams PC (P_5EC_5E)

```
dx_1 = @(x_1, x_2) x_2 + x_1 * (0.5 - x_1^2 - x_2^2);
dx_2 = @(x_1, x_2) -x_1 + x_2 * (0.5 - x_1^2 - x_2^2);

%initial conditions
init_x_1 = 0.8;
init_x_2 = 0.7;

h = 0.0001; % step

err_fac = (863/60480) / ((-95/288) + (863 / 60480));

x_1(1) = init_x_1;
x_2(1) = init_x_2;

betaE = [1901, -2774, 2616, -1274, 251];
betaE = betaE / 720;

betaI = [475, 1427, -798, 482, -173, 27];
betaI = betaI / 1440;

for i = 1:4
    [k_1, k_2] = computeKs(dx_1, dx_2, x_1(i), x_2(i), h);
    x_1(i + 1) = x_1(i) + h * (k_1(1) + 2 * k_1(2) + 2 * k_1(3) + k_1(4)) / 6;
    x_2(i + 1) = x_2(i) + h * (k_2(1) + 2 * k_2(2) + 2 * k_2(3) + k_2(4)) / 6;
end
```

```
for i = 6:ceil(20 / h)
    sum_x_1 = 0;
    sum_x_2 = 0;
    for j = 1:5
        sum_x_1 = sum_x_1 + betaE(j) * dx_1(x_1(i - j), x_2(i - j));
        sum_x_2 = sum_x_2 + betaE(j) * dx_2(x_1(i - j), x_2(i - j));
    end

    temp_x_1 = x_1(i - 1) + h * sum_x_1;
    temp_x_2 = x_2(i - 1) + h * sum_x_2;

    sum_x_1 = 0;
    sum_x_2 = 0;
    for j = 1:5
        sum_x_1 = sum_x_1 + betaI(j + 1) * dx_1(x_1(i - j), x_2(i - j));
        sum_x_2 = sum_x_2 + betaI(j + 1) * dx_2(x_1(i - j), x_2(i - j));
    end

    x_1(i) = x_1(i - 1) + h * sum_x_1 + h * betaI(1) * dx_1(temp_x_1,
temp_x_2);
    x_2(i) = x_2(i - 1) + h * sum_x_2 + h * betaI(1) * dx_2(temp_x_1,
temp_x_2);

    err_x_1 = err_fac * (temp_x_1 - x_1(i));
    err_x_2 = err_fac * (temp_x_2 - x_2(i));
end
```

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

```
dx_1 = @(x_1, x_2) x_2 + x_1 * (0.5 - x_1^2 - x_2^2);
dx_2 = @(x_1, x_2) -x_1 + x_2 * (0.5 - x_1^2 - x_2^2);

epsr = 10^-4;
epsa = 10^-4;

%initial conditions
x_1 = 0.8;
x_2 = 0.7;

i = 1;
a = 0;

x_1_val(1) = x_1;
x_2_val(1) = x_2;

h = 0.25;
step(i) = h;
```

```
while (a < 20)
    %next points
    [k_1, k_2] = computeKs(dx_1, dx_2, x_1, x_2, h);

    temp_x_1 = x_1;
    temp_x_2 = x_2;

    x_1 = x_1 + h * (k_1(1) + 2 * k_1(2) + 2 * k_1(3) + k_1(4)) / 6;
    x_2 = x_2 + h * (k_2(1) + 2 * k_2(2) + 2 * k_2(3) + k_2(4)) / 6;

    x_1_val(i + 1) = x_1;
    x_2_val(i + 1) = x_2;

    %first half-step
    h = 0.5 * h;

    [k_1, k_2] = computeKs(dx_1, dx_2, x_1, x_2, h);

    temp_1 = x_1 + h * (k_1(1) + 2 * k_1(2) + 2 * k_1(3) + k_1(4)) / 6;
    temp_2 = x_2 + h * (k_2(1) + 2 * k_2(2) + 2 * k_2(3) + k_2(4)) / 6;

    %second half-step
    [k_1, k_2] = computeKs(dx_1, dx_2, temp_1, temp_2, h);

    temp_1 = temp_1 + h * (k_1(1) + 2 * k_1(2) + 2 * k_1(3) + k_1(4)) / 6;
    temp_2 = temp_2 + h * (k_2(1) + 2 * k_2(2) + 2 * k_2(3) + k_2(4)) / 6;

    h = 2 * h;

    errors(i, 1) = (temp_1 - x_1) / 120;
    errors(i, 2) = (temp_2 - x_2) / 120;

    eps_1 = abs(temp_1) * epsr + epsa;
    eps_2 = abs(temp_2) * epsr + epsa;

    alpha_1 = (eps_1 / abs(errors(i, 1)))^(1/5);
    alpha_2 = (eps_2 / abs(errors(i, 2)))^(1/5);

    alpha = min(alpha_1, alpha_2);

    h_new = 0.9 * alpha * h;

    if (0.9 * alpha >= 1)
        if (a + h >= 20)
            break;
        else
            a = a + h;
            h = min([h_new, 5 * h, 20 - a]);
            i = i + 1;
            steps(i) = h;
            continue;
        end
    else
        if (h_new < 0)
            error('Cant solve with this epsilon');
        else
            h = h_new;
        end
    end

    i = i + 1;
    steps(i) = h;
end
```

Modified Gram-Schmidt algorithm, used in ***ls_approximationQR*** function:
https://blogs.mathworks.com/cleve/2016/07/25/compare-gram-schmidt-and-householder-orthogonalization-algorithms/

Piotr Tatjewski, Numerical Methods, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2014