ENUME
PROJECT C No. 17

Done by Filip Korzeniewski

1. Approximation

Approximation task is to find simpler function F(x) in specified interval or in finite number of points of its interval, belonging to chosen class of functions. Approximation of continuous function in established interval using simpler function is used when complex analytical form of the continuous one makes it hard to use e.g. in projects methods or analysis methods. Approximation using continuous function of function known in finite number of points (discrete approximation) is widely used in modeling techniques.

The least-squares approximation

We can distinguish many approximation types. One of them is the least-squares approximation. The least-squares approximation of function f(x) is defined in finite set of N+1 points:

$$||F - f|| = \sqrt{\sum_{j=0}^{N} p(x_j)[F(x_j) - f(x_j)]^2}$$

Assume that for a given finite number of points $x_0, x_1, \dots x_N$ ($x_i \neq x_j$), the values $y_j = f(x_j)$, $j = 0,1,2, \dots, N$ are known.

Let $\Phi_i(x), i = 0,1, \dots, n$ be a basis of a space $X_n \subseteq X$ of interpolating functions, i.e.,

$$\forall F \in X_n \qquad F(x) = \sum_{i=0}^{n} a_i \, \Phi_i(x)$$

The approximation problem: to find values of the parameters $a_0, a_1, \dots, a_n$ defining the approximating function mentioned above which minimize the least-squares error defined by:

$$H(a_0, \dots, a_n) \overset{\text{def}}{=} \sum_{j=0}^{N} \left[f(x_j) - \sum_{i=0}^{n} a_i \Phi_i(x_j) \right]^2$$

The weighting function is not present to simplify the presentation.

The formula for the coefficients $a_0, a_1, \dots, a_n$ can be derived from the necessary condition for a minimum (being here also the sufficient condition, as the function is convex):

$\frac{\partial H}{\partial a_k} = -2 \sum_{j=0}^{N} \left[f(x_j) - \sum_{i=0}^{n} a_i \Phi_i(x_j) \right] * \Phi_k(x_j) = 0,$ where $k = 0, \dots, n$

The system of linear equations with the unknowns $a_0, a_1, \ldots, a_n$ is called the *set of normal equations* and its matrix *the Gram's matrix*.

Let be defined the following matrix A:

$$A = \begin{bmatrix} \Phi_0(\mathrm{x}_0) & \cdots & \Phi_n(\mathrm{x}_0) \\ \vdots & \ddots & \vdots \\ \Phi_0(\mathrm{x}_N) & \cdots & \Phi_n(\mathrm{x}_N) \end{bmatrix}$$

And also be defined:

a = $[a_0, a_1, \ldots, a_n]^T$,

y = $[y_0, y_1, \ldots, y_n]^\mathrm{T}$, $\quad y_j = \mathrm{f}(x_j)$, $\qquad$ j = 0, 1, …, N.

The performance function of the approximation problem can be now written as

H(a) = $(||y-Aa||_2)^2$.

Therefore, the problem of the least-squares approximation is a linear least-squares problem (LLSP).

All columns of the matrix A are linearly independent (which follows from linear independence of the basic functions). Hence, the matrix has full rank (n+1).

The set of normal equations can be written in the following form:

$$A^TAa=A^Ty.$$

Since, the matrix A has full rank, then the Gram's matrix A^TA is nonsingular. This implies uniqueness of the solution of the set of normal equations. But, even being nonsingular, the matrix A^TA can be badly conditioned – its condition number is a square of the condition number of **A**. That's why it is recommended to solve the approximation problem using the method based on the QR factorization of A.

Given task:

Given data:

x_i	y_i
-5	-7.7743
-4	-0.2235
-3	1.9026
-2	0.6572
-1	0.1165
0	-1.8144
1	-1.0968
2	-0.8261
3	1.3327
4	6.1857
5	8.2892

Task outline:

The task was to determine a polynomial function that best fits the given data by using the least-squares approximation. For each solution there supposed to be calculated error defined as Euclidean norm of vector residuum and the condition number of the Gram's matrix.

Results and conclusions:

The task was solved using polynomials of different degrees (from 0 to 9).

Gram's matrix differs for every next degree of polynomial just with the number of columns (for higher degree there is additional column). The number of columns is linearly dependent on polynomial's degree.

1	-5	25	-125	625	-3125	15625	-78125	390625	-1953125
1	-4	16	-64	256	-1024	4096	-16384	65536	-262144
1	-3	9	-27	81	-243	729	-2187	6561	-19683
1	-2	4	-8	16	-32	64	-128	256	-512
1	-1	1	-1	1	-1	1	-1	1	-1
1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	2	4	8	16	32	64	128	256	512
1	3	9	27	81	243	729	2187	6561	19683
1	4	16	64	256	1024	4096	16384	65536	262144
1	5	25	125	625	3125	15625	78125	390625	1953125

Figure 1: Gram's matrix for 9th degree of polynomial

Degree	Condition
0	1,00
1	3,16
2	20,22
3	92,51
4	563,90
5	2732,67
6	16827,24
7	87442,67
8	574931,68
9	3894497,74

Figure 2: Condition number for every degree of Gram's matrix

We can see that conditions grows significantly for higher degrees. It relates to the fact that for higher degrees Gram's matrix has bigger values at last places.

	Degree of polynomial									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Coefficients	0,61353	0,61353	0,18183	0,18183	-1,58185	-1,58185	-1,20260	-1,20260	-1,32594	-1,32594
		0,90968	0,90968	0,84366	0,84366	0,85836	0,85836	0,44225	0,44225	0,74267
			0,07954	0,07954	0,56565	0,56565	0,26826	0,26826	0,45600	0,45600
				0,09850	0,09850	0,10094	0,10094	0,04523	0,04523	0,15696
					0,01944	0,01944	0,01334	0,01334	0,02929	0,02929
						-0,00008	-0,00008	0,01223	0,01223	0,02275
							0,00087	0,00087	0,00211	0,00211
								0,00029	0,00029	0,00185
									-0,00006	-0,00006
										-0,00004

Figure 3: Calculated coefficients of approximating polynomials

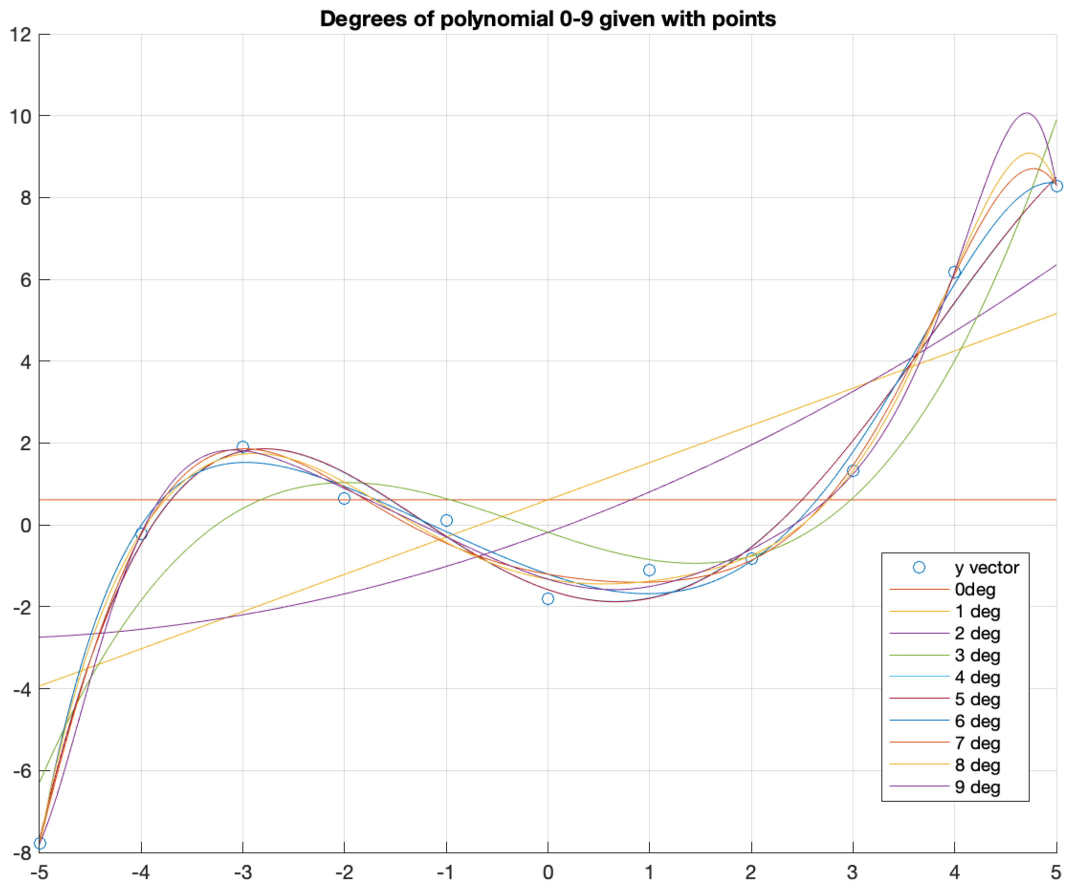


Figure 4: Points of given vector and various degrees polynomial approximation

We can claim just by seeing that for polynomial with degree smaller than 3 data is not very well approximated. It is seen with bare eye that polynomial with 3rd degree should be enough to approximate it properly. For higher degrees the differences are not significant.

Degree	Norm
0	13,20385
1	9,12770
2	8,82538
3	4,23651
4	1,54156
5	1,54108
6	1,16890
7	0,93449
8	0,88821
9	0,83315

Figure 5: Euclidean norm of difference between given data and value of polynomial in each point

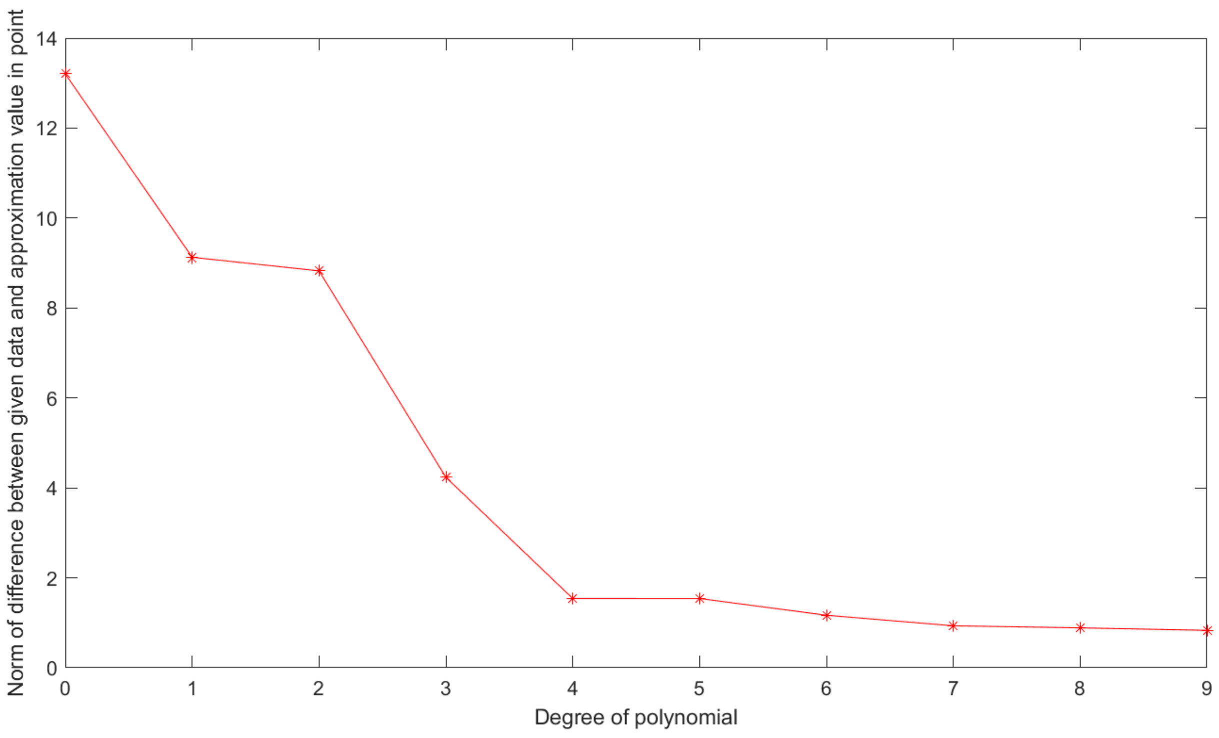


Figure 6: Euclidean norm of difference between given data and value of polynomial in each point dependent on degree of polynomial

As it has been said, we can see on the graph that for polynomials' degrees higher than 3 the differences are not to big. It confirms the conclusions.

Degree	**Norm**
0	4,44E-16
1	4,44E-16
2	1,88E-15
3	1,60E-15
4	5,84E-15
5	3,47E-15
6	5,27E-15
7	9,62E-15
8	1,75E-14
9	3,32E-14

Figure 7: Euclidean norm of residuum vector for each polynomial

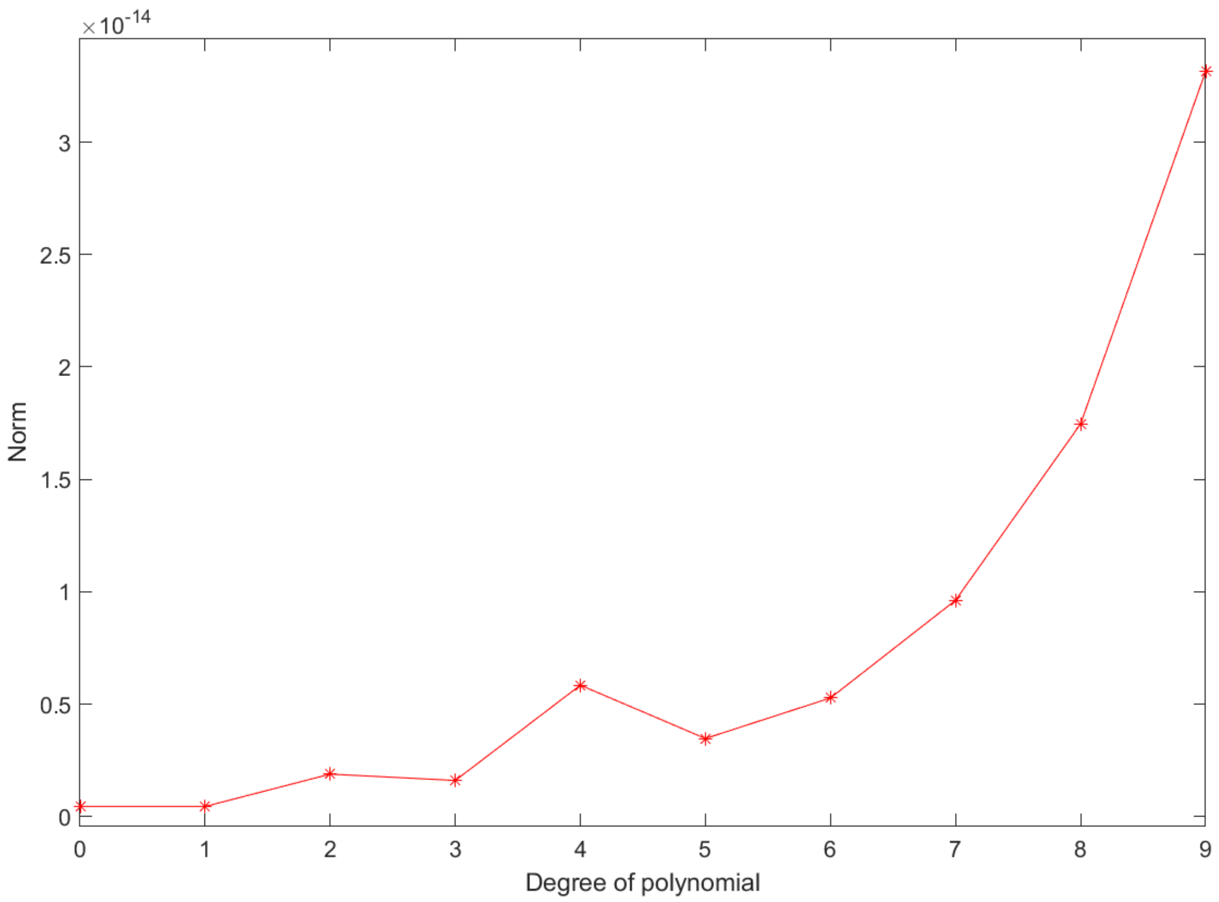


Figure 8: Euclidean norm of residuum vector dependent on polynomials' degrees

We can see on the graph that for QR decomposition norm of residuum vector increases with increasing degree of polynomial. For degree higher than 5 it starts increasing very fast but generally the values of norm are still very small.

2. Ordinary differential equations

Differential equations are commonly used for mathematical modeling of dynamical systems. Systems of differential equations are generally nonlinear and there are not known any methods of calculating their analytical solutions. The only way is to solve their using numerical methods. Numerical methods of solving Cauchy problem are algorithmic base of continuous simulations of dynamical systems. Moreover, they are element of algorithms of more complex systems of differential equations like partial differential equations.

Task:

A motion of a point is given by equations:

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

There is to determine the trajectory of the motion on the interval [0, 20] for conditions:

$x_1(0) = 8, x_2(0) = 7$

a) Runge-Kutta method of 4th order (RK4) and Adams PC (P_5EC_5E) with different constant step-sizes,

b) Runge-Kutta method of 4th order (RK4) with variable step size automatically adjusted by the algorithm

Runge-Kutta method of 4th order (RK4)

Generally, in numerical analysis, the Runge–Kutta methods are a family of implicit and explicit iterative methods, which include the well-known routine called the Euler Method, used in temporal discretization for the approximate solutions of ordinary differential equations.

Runge-Kutta methods can be defined by:

$$y_{n+1} = y_n + h * \sum_{i=1}^{m} w_i k_i$$

where:

$$k_1 = f(x_n, y_n),$$

$$k_i = f\left(x_n + c_i h, y_n + h * \sum_{j=1}^{i-1} a_{ij} k_j \right), \quad i = 2, 3, \dots, m$$

and:

$$\sum_{j=1}^{i-1} a_{ij} = c_i, \qquad i = 2, 3, \dots, m$$

For performing one step it is needed to calculate values of differential equations right sides exactly *m* times. Parameters w_i, a_{ij}, c_i are taken in a way so that in current *m* order of method be as high as possible. If *p(m)* is maximum available to achieve order, then:

$p(m) = m$	for $m = 1, 2, 3, 4$
$p(m) = m - 1$	for $m = 5, 6, 7$
$p(m) \leq m - 2$	for $m \geq 8$.

The most significant are methods with m = 4 and order 4th – compromise between accuracy and amount of calculations per one iteration and related to this, errors.
The 4th order method can be shown in the way:

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

Adam's method P_kEC_kE:

Adam's methods are classified as the multistep methods. General form of the formula defining step (iteration) of k-step linear method with constant step *h* is:
$y_n = \sum_{j=1}^{k} \alpha_j \, y_{n-j} + h\sum_{j=1}^{k} \beta_j \, f(x_{n-j}, y_{n-j})$,
Where $y_0 = y(x_0) = y_a$, $x_n = x_0 + nh$, $x_0 = a$, $x \in [a,b]$
The multistep method can be explicit or implicit. The method is explicit if $\beta_0 = 0$ or implicit if $\beta_0 \neq 0$.
K-step method is convergent if and only if when is stable and at least of order 1($c_0 = c_1 = 0$ – approximation, consistency condition).
The most useful multistep method would be method with:

a) a high order and a low error constant
b) possibly large set of the absolute stability
c) possibly small number of arithmetic operations per iteration

The explicit methods are worse as for first two conditions. Implicit methods are much better in this criterium, but they do not fulfill the third one because in each iteration there is need to solve nonlinear equation versus y_n.
Practical aspects of using multistep methods are algorithms predictor-corrector types, being combination of explicit and implicit methods. For k-step method realization of predictor-connector P_kEC_kE is:

$$P-prediction: \qquad y_n^{[0]}=\sum_{j=1}^{k}\alpha_j^* y_{n-1}+h\sum_{j=1}^{k}\beta_i f_{n-j}$$

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

$$C-correction: \qquad y_n=\sum_{j=1}^{k}\alpha_j^* y_{n-1}+h\sum_{j=1}^{k}\beta_j^* f_{n-j}+h\beta_0^* f_n^{[0]}$$

$$E-evaluation: \qquad f_n=f(x_n,y_n)$$

For Adam's method predictor-connector P_kEC_kE algorithm is:

$$P-prediction: \qquad y_n^{[0]}=y_{n-1}+h\sum_{j=1}^{k}\beta_i f_{n-j}$$

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

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

$$E-evaluation: \qquad f_n=f(x_n,y_n)$$

Left endpoint values of the absolute stability intervals for the Adam's methods:

k	explicit	The Adam's method implicit	P_kEC_kE
1	-2	-∞	-2
2	-1	-6	-2.4
3	-0.55	-3	-2
4	-0.3	-1.83	-1.4
5	-0.18	-1.18	-1.05
6	-0.12	0.78	-0.76

Runge-Kutta method of 4th order with a variable step size automatically adjusted

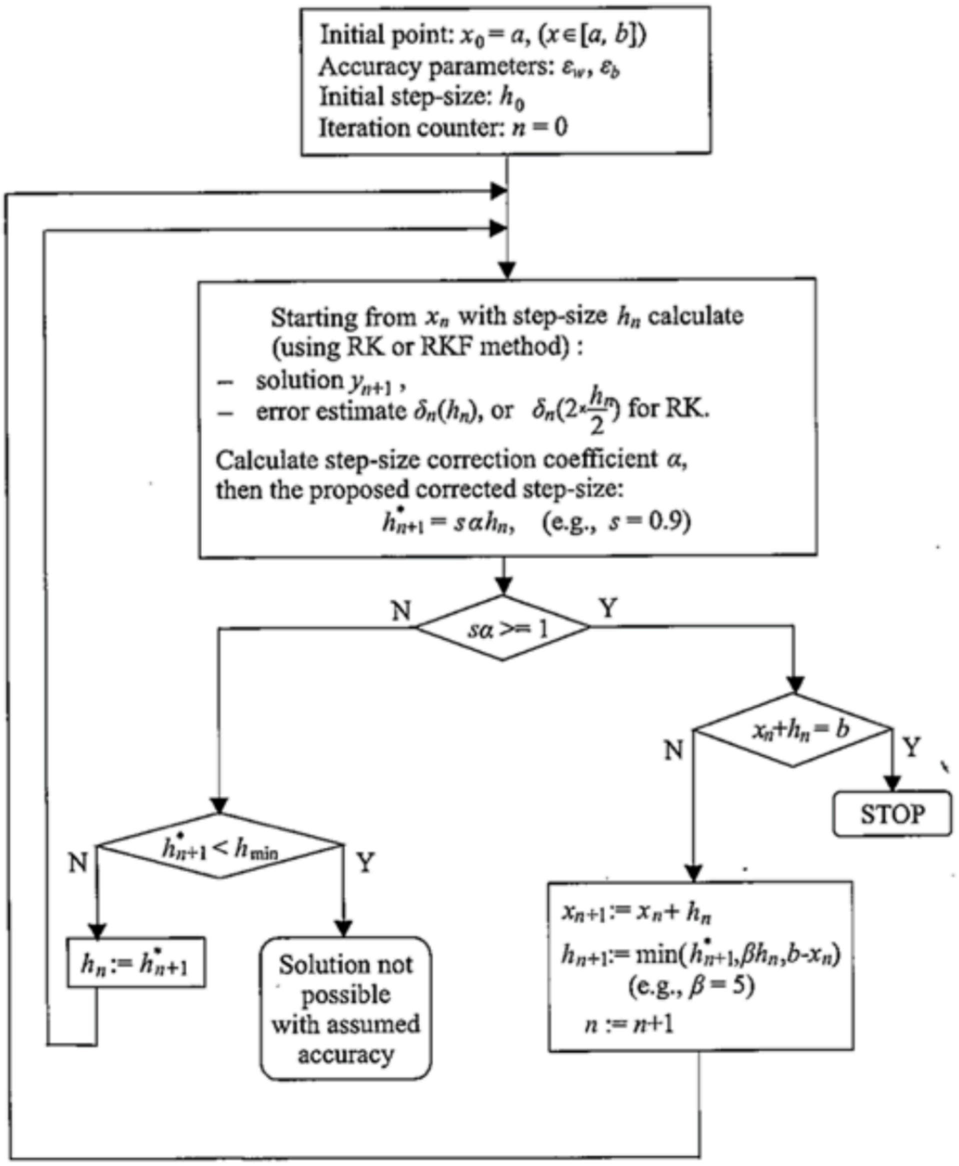

Initial point: $x_0 = a$, $(x \in [a, b])$
Accuracy parameters: ε_w, ε_b
Initial step-size: h_0
Iteration counter: $n = 0$
Starting from x_n with step-size h_n calculate (using RK or RKF method):
– solution y_{n+1},
– error estimate $\delta_n(h_n)$, or $\delta_n(2 \times \frac{h_n}{2})$ for RK.
Calculate step-size correction coefficient α, then the proposed corrected step-size:
$\overset{*}{h}_{n+1} = s\alpha h_n$, (e.g., $s = 0.9$)
N
Y
$s\alpha >= 1$
$x_n + h_n = b$
N
Y
STOP
$\overset{*}{h}_{n+1} < h_{\min}$
N
Y
$h_n := \overset{*}{h}_{n+1}$
Solution not possible with assumed accuracy
$x_{n+1} := x_n + h_n$
$h_{n+1} := \min(\overset{*}{h}_{n+1}, \beta h_n, b - x_n)$
(e.g., $\beta = 5$)
$n := n+1$

Figure 9: flowchart of algorithm

The only update is that for 2 equations α is computed for the most critical equation (the worst case).

<u>**Results 2a:**</u>

The task was solved using RK4 method. P_5EK_5E for Adam's method and in-built function ODE45. The RK4 is one-step method and Adam's P_5EK_5E is multistep method.

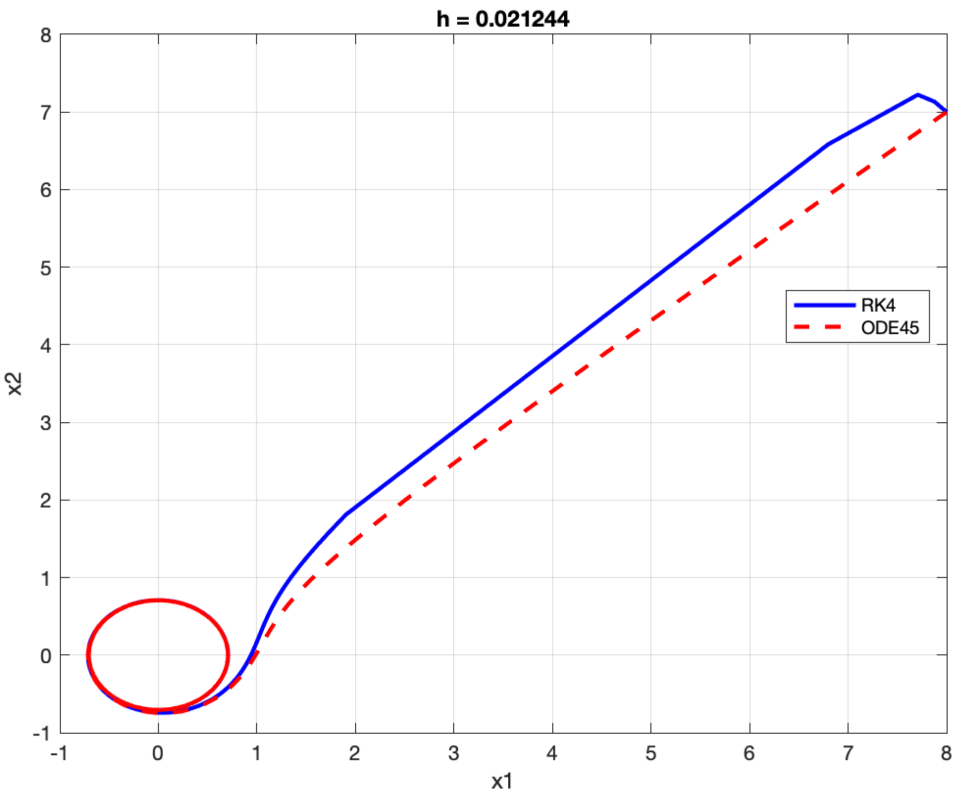


Figure 10: trajectory of the point computed using step h=0.021244

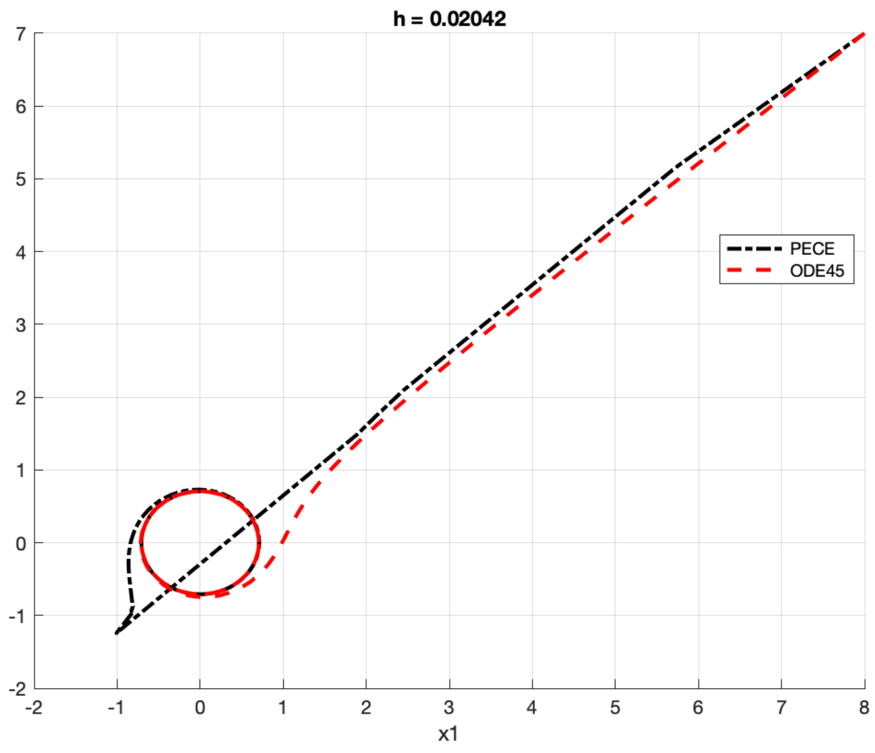


Figure 11: trajectory of the point computed using step h=0.02042

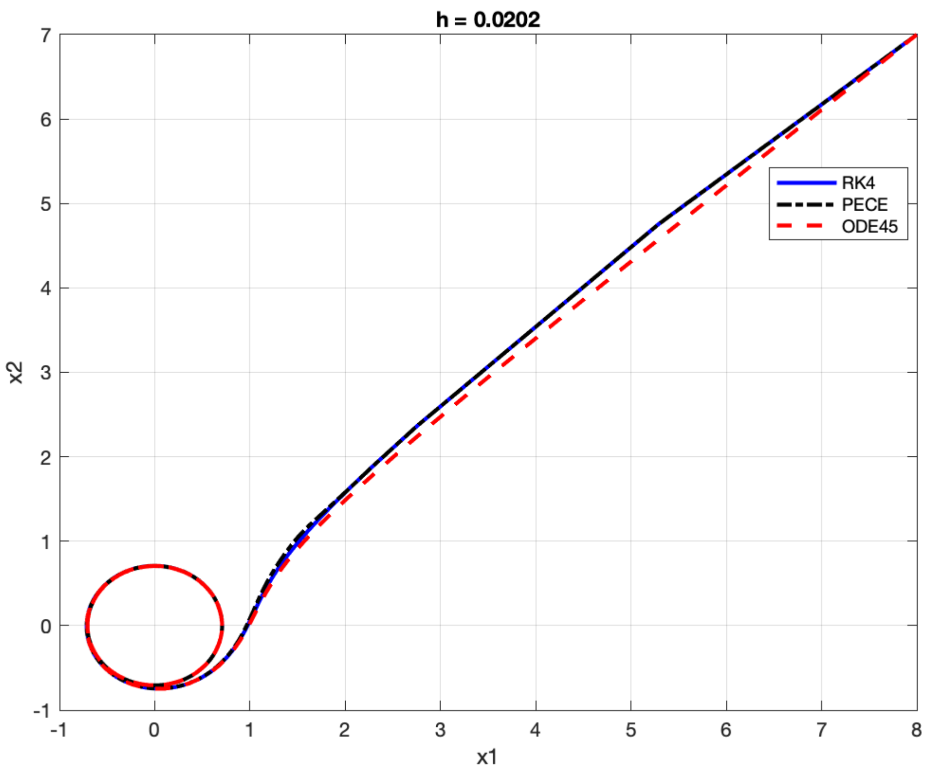


Figure 12: trajectory of the point computed using h=0.0202

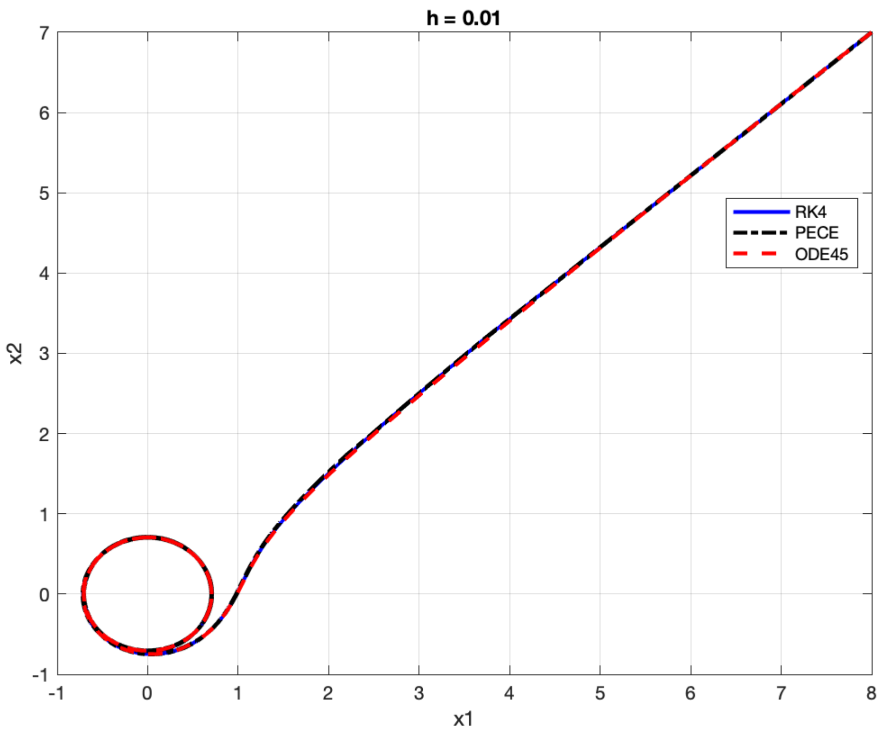


Figure 13: trajectory of the point computed using step h=0.01

We can see on the above plots that step h=0.021244 for Adam's PECE method and step h=0.02042 for RK4 method too big to compute it with proper accuracy. Step h=0.0202 is the threshold value for Adam's and h=0.0212 for RK4. Increasing these steps, trajectory starts to be significantly unstable. On the third plot with step h=0.01 (lower than threshold) we can see that trajectories are almost the same.

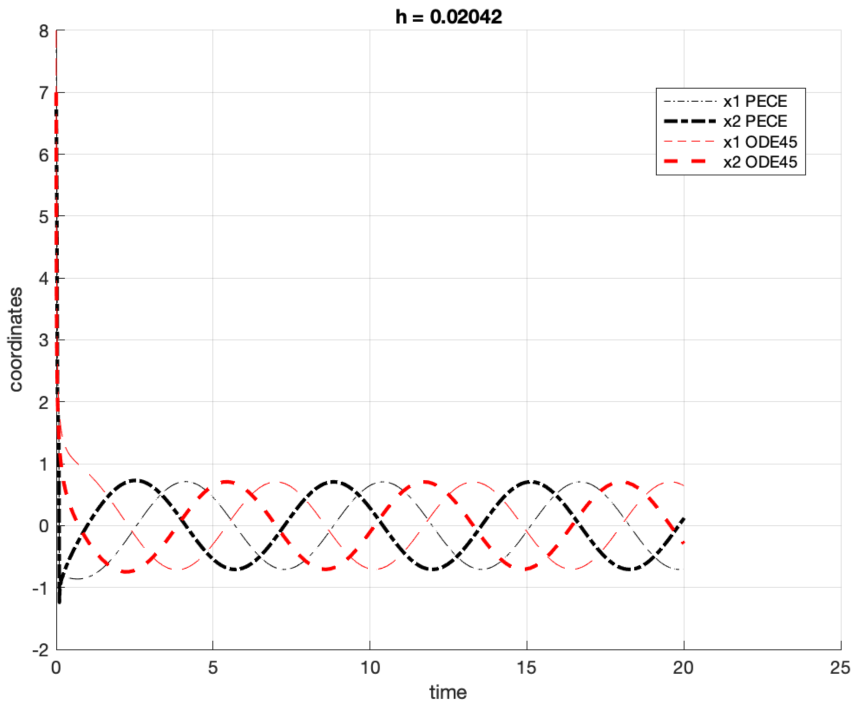


Figure 14: problem solution versus time for h=0.02042

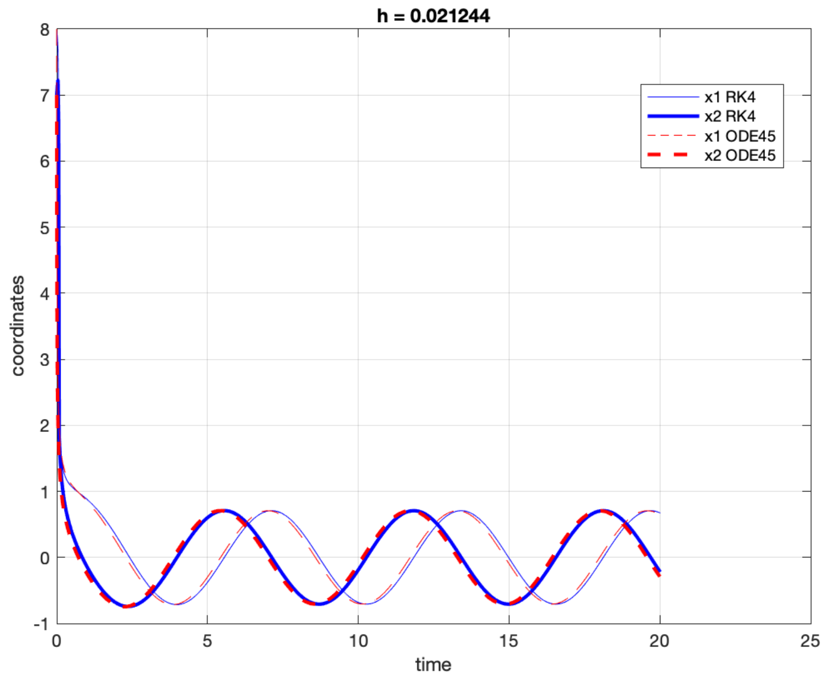


Figure 15: problem solution versus time for h=0.021244

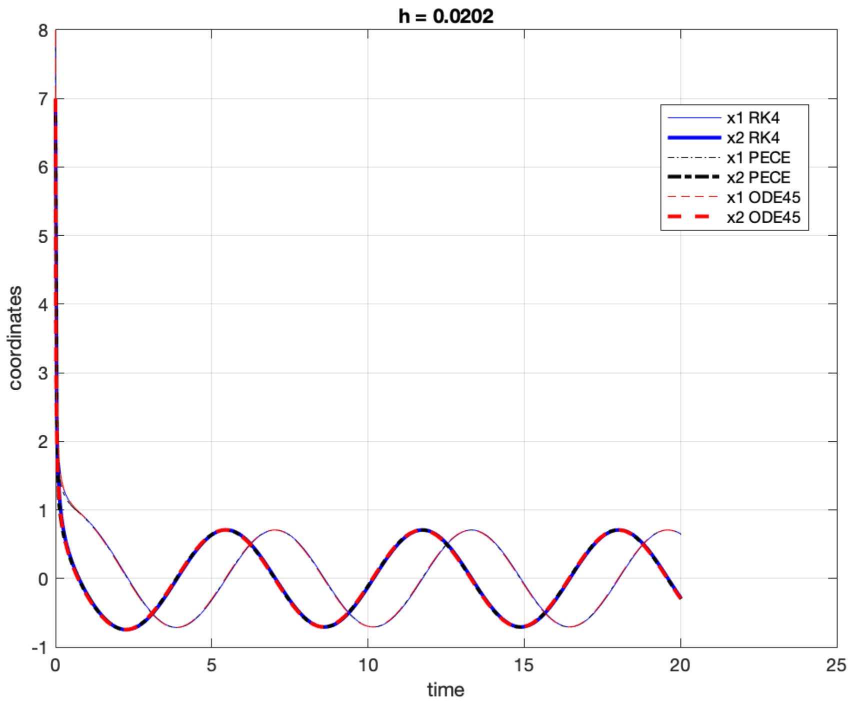


Figure 16: problem solution versus time for h=0.0202

For h=0.02042 the coordinates of Adam's PECE method are definitely different than for in-built function ODE45. The coordinates seem to be shifted. The coordinates of RK4 starts to differ with step h = 0.021244. The coordinates of Adam's versus ODE45 function are different in 1st part of time.

Results 2b:

The task was computed using algorithm showed in theoretical part using error estimation according to the step-doubling rule.

Epsilon value, the accuracy was taken as 10e-7. The main assumption was to take it as bigger of machine epsilon which should be obvious.

The hmin was taken in the way not to be smallest than 10e-9 because as it had been showed in previous examples it would have been not efficient. Moreover, hmin must be as small as the epsilon allows it. When hmin was too big and the assumed accuracy to precise the algorithm would not be able to compute it.

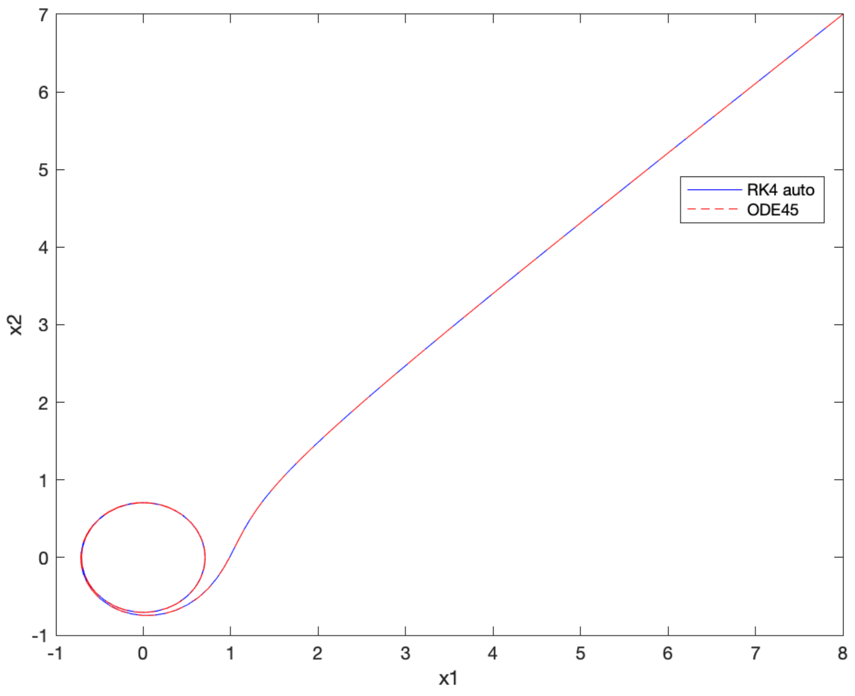


Figure 17: trajectory of the point computed with RK4 auto and ODE45

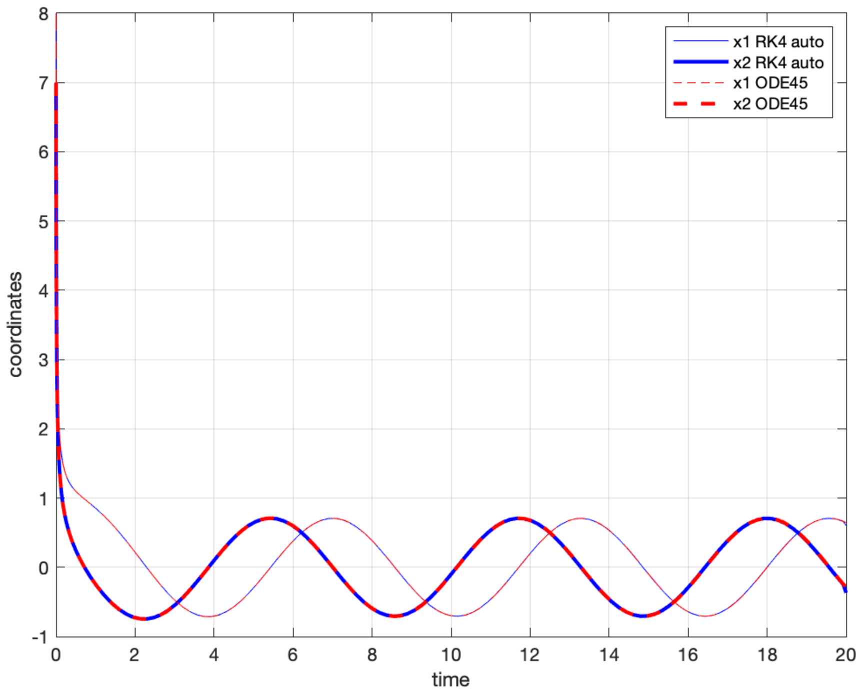


Figure 18: coordinates of x1 and x2 versus time

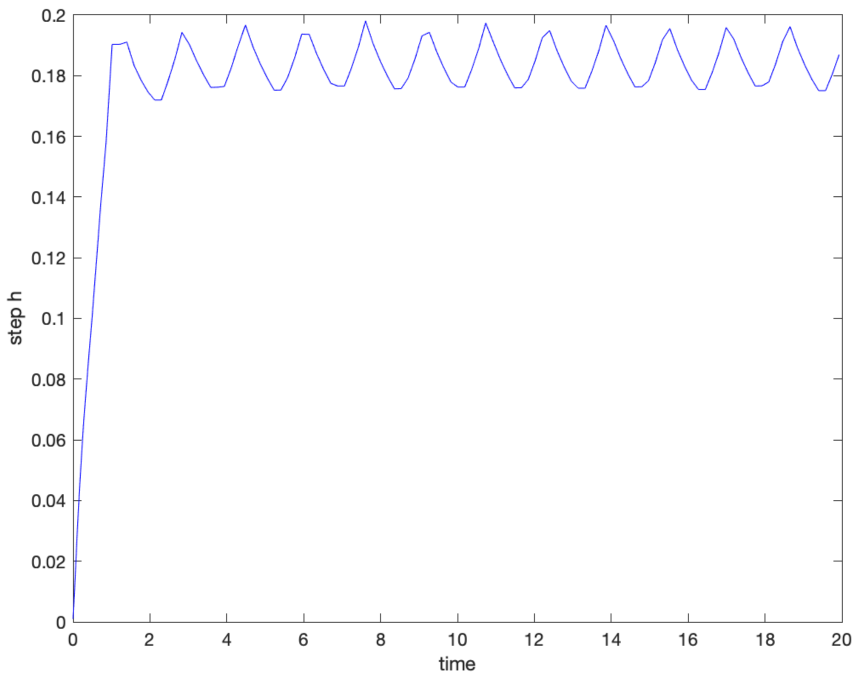


Figure 19: change of step h in time domain

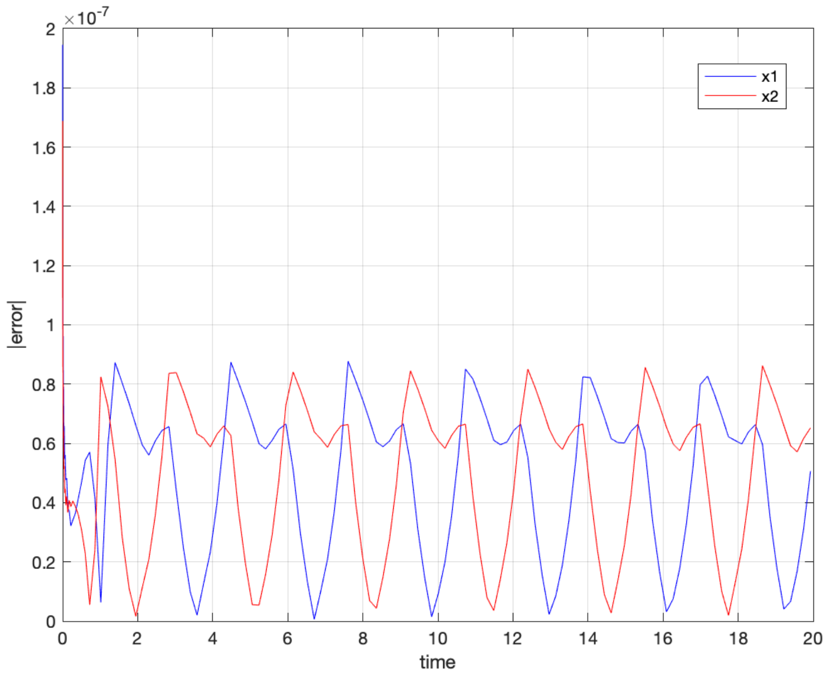


Figure 20: absolute value of error estimation changes versus time

We can see that the result is the same as the in-built function (either trajectory or time domain). With the given epsilon = 10e-7 we can see that algorithm's step is oscillating around h=0.17÷0.195 for most of the time interval. The ODE45 needed to perform 169 iterations while RK4 automatic method 130 (with given accuracy). It seems that this automatic function is very well designed and very efficient.

3. Code:

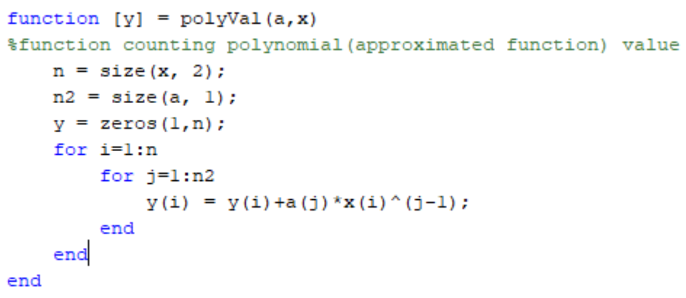
1st task:

```
function [Q,R] = qrmgs(A)
%QR decomposition code taken from prof. Tatjewski's "Numerical methods"
    [m, n] = size(A);
    Q = zeros(m,n);
    R = zeros(n,n);
    d = zeros(1,n);

    %QR decomposition
    for i =1:n
        Q(:,i) = A(:,i);
        R(i,i) = 1;
        d(i) = Q(:,i)' * Q(:,i);
        for j = i+1:n
            R(i,j) = (Q(:,i)'*A(:,j))/d(i);
            A(:,j) = A(:,j) - R(i,j)*Q(:,i);
        end
    end
    %set normalization
    for i=1:n
        dd = norm(Q(:,i));
        Q(:,i) = Q(:,i)/dd;
        R(i,i:n) = R(i,i:n) *dd;
    end
end
```

Figure 21: QR decomposition from prof. Tatjewski's book

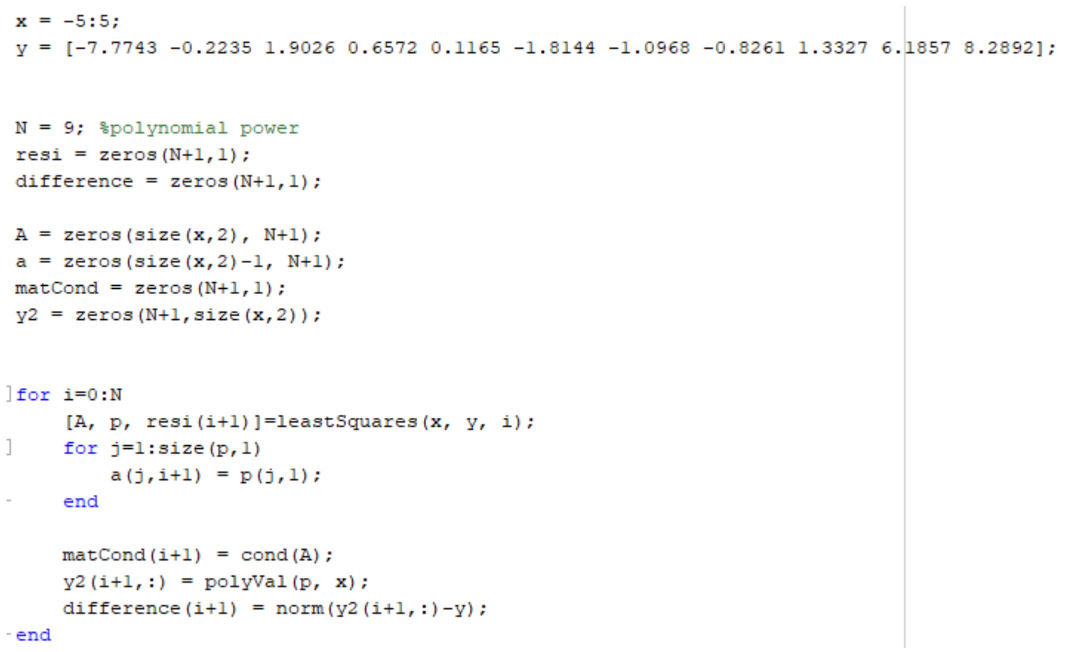
```
function [A, a,resi] = leastSquares(x,y,N)
%discrete least-squares

    y = y';
    N = N+1;
    length = size(x, 2);
    A = zeros(length, N);

    for i = 1:length
        for j = 1:N
            A(i, j)=x(i)^(j-1);
        end
    end

    [Q,R] = qrmgs(A);
    a = R\Q'*y; %coefficients according to linear least squares
    resi = norm(R*a - Q'*y); %residuum vector
end
```

Figure 22: function calculating coefficients residuum vector and Gram's matrix for given data and degree of polynomial



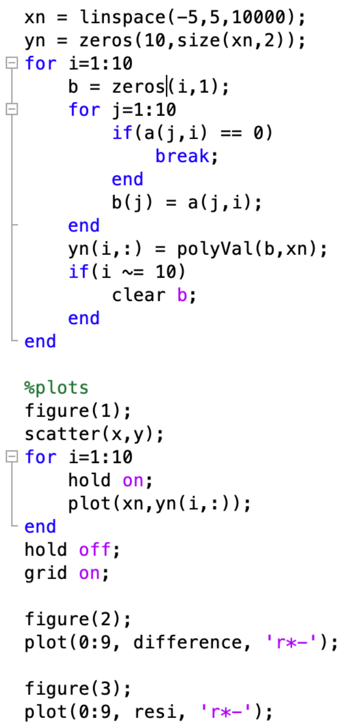
```
function [y] = polyVal(a,x)
%function counting polynomial(approximated function) value
    n = size(x, 2);
    n2 = size(a, 1);
    y = zeros(1,n);
    for i=1:n
        for j=1:n2
            y(i) = y(i)+a(j)*x(i)^(j-1);
        end
    end
end
```

Figure 23: function counting values in given points using given coefficients



```
x = -5:5;
y = [-7.7743 -0.2235 1.9026 0.6572 0.1165 -1.8144 -1.0968 -0.8261 1.3327 6.1857 8.2892];


N = 9; %polynomial power
resi = zeros(N+1,1);
difference = zeros(N+1,1);

A = zeros(size(x,2), N+1);
a = zeros(size(x,2)-1, N+1);
matCond = zeros(N+1,1);
y2 = zeros(N+1,size(x,2));


for i=0:N
    [A, p, resi(i+1)]=leastSquares(x, y, i);
    for j=1:size(p,1)
        a(j,i+1) = p(j,1);
    end

    matCond(i+1) = cond(A);
    y2(i+1,:) = polyVal(p, x);
    difference(i+1) = norm(y2(i+1,:)-y);
end
```

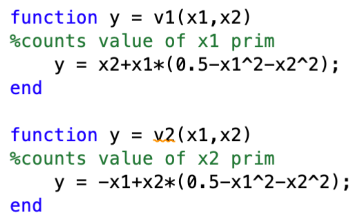
Figure 24: core of program using functions to perform calculations on given data



```
xn = linspace(-5,5,10000);
yn = zeros(10,size(xn,2));
for i=1:10
    b = zeros(i,1);
    for j=1:10
        if(a(j,i) == 0)
            break;
        end
        b(j) = a(j,i);
    end
    yn(i,:) = polyVal(b,xn);
    if(i ~= 10)
        clear b;
    end
end

%plots
figure(1);
scatter(x,y);
for i=1:10
    hold on;
    plot(xn,yn(i,:));
end
hold off;
grid on;

figure(2);
plot(0:9, difference, 'r*-');

figure(3);
plot(0:9, resi, 'r*-');
```

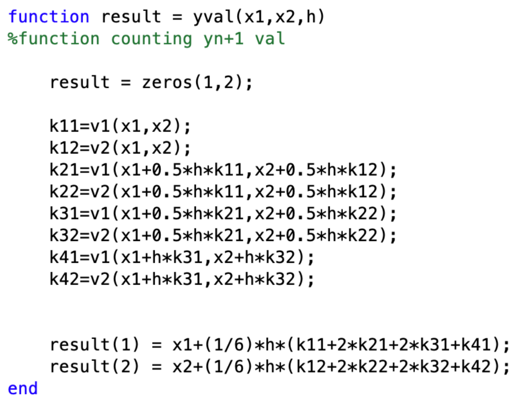
Figure 25: plots needed to achieve results

2a:



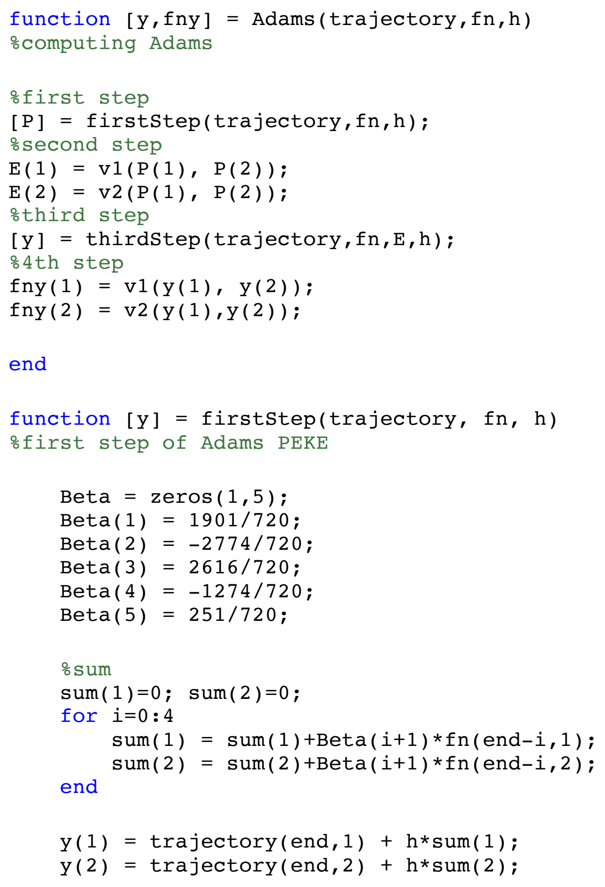
```
function y = v1(x1,x2)
%counts value of x1 prim
    y = x2+x1*(0.5-x1^2-x2^2);
end

function y = v2(x1,x2)
%counts value of x2 prim
    y = -x1+x2*(0.5-x1^2-x2^2);
end
```

Figure 26: functions allowing to compute f(xn,yn)

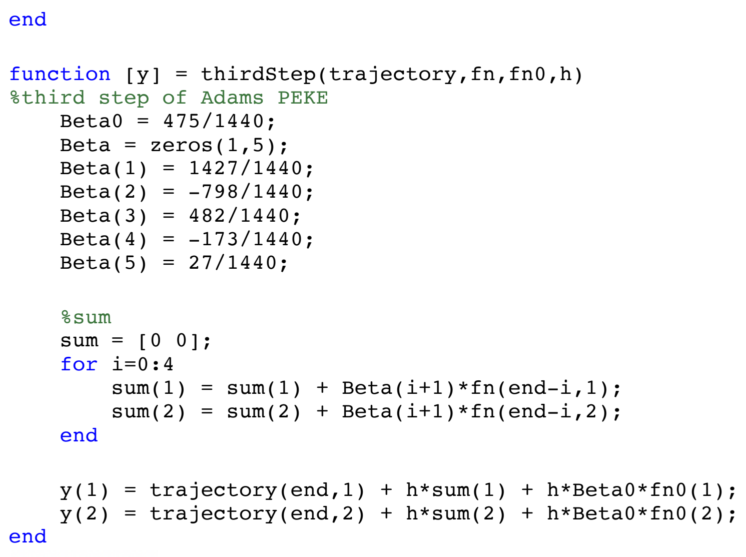


```
function result = yval(x1,x2,h)
%function counting yn+1 val

    result = zeros(1,2);

    k11=v1(x1,x2);
    k12=v2(x1,x2);
    k21=v1(x1+0.5*h*k11,x2+0.5*h*k12);
    k22=v2(x1+0.5*h*k11,x2+0.5*h*k12);
    k31=v1(x1+0.5*h*k21,x2+0.5*h*k22);
    k32=v2(x1+0.5*h*k21,x2+0.5*h*k22);
    k41=v1(x1+h*k31,x2+h*k32);
    k42=v2(x1+h*k31,x2+h*k32);


    result(1) = x1+(1/6)*h*(k11+2*k21+2*k31+k41);
    result(2) = x2+(1/6)*h*(k12+2*k22+2*k32+k42);
end
```

Figure 27: function used for computing coefficients and next value using RK4 method

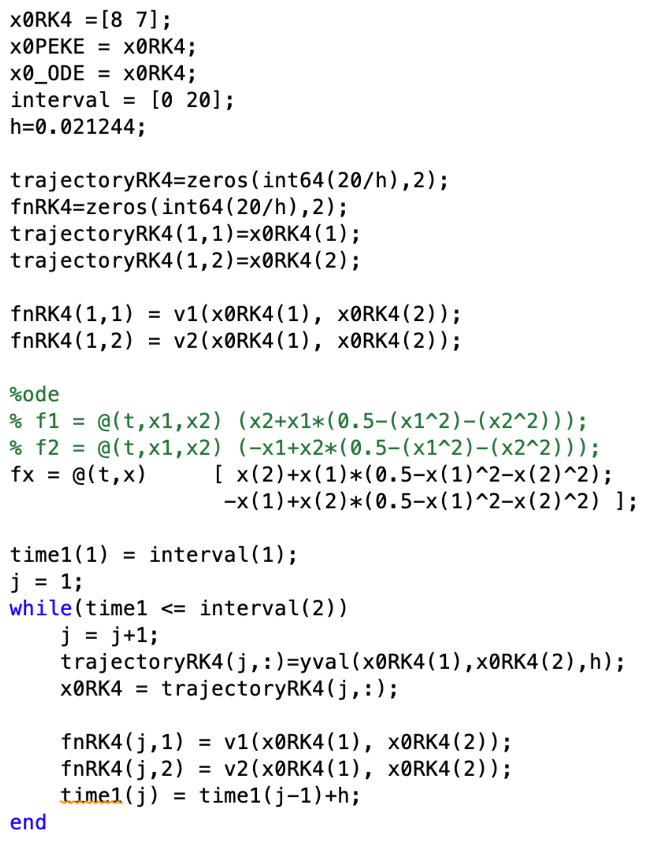


```
function [y,fny] = Adams(trajectory,fn,h)
%computing Adams

%first step
[P] = firstStep(trajectory,fn,h);
%second step
E(1) = v1(P(1), P(2));
E(2) = v2(P(1), P(2));
%third step
[y] = thirdStep(trajectory,fn,E,h);
%4th step
fny(1) = v1(y(1), y(2));
fny(2) = v2(y(1),y(2));

end

function [y] = firstStep(trajectory, fn, h)
%first step of Adams PEKE

    Beta = zeros(1,5);
    Beta(1) = 1901/720;
    Beta(2) = -2774/720;
    Beta(3) = 2616/720;
    Beta(4) = -1274/720;
    Beta(5) = 251/720;

    %sum
    sum(1)=0; sum(2)=0;
    for i=0:4
        sum(1) = sum(1)+Beta(i+1)*fn(end-i,1);
        sum(2) = sum(2)+Beta(i+1)*fn(end-i,2);
    end

    y(1) = trajectory(end,1) + h*sum(1);
    y(2) = trajectory(end,2) + h*sum(2);
```



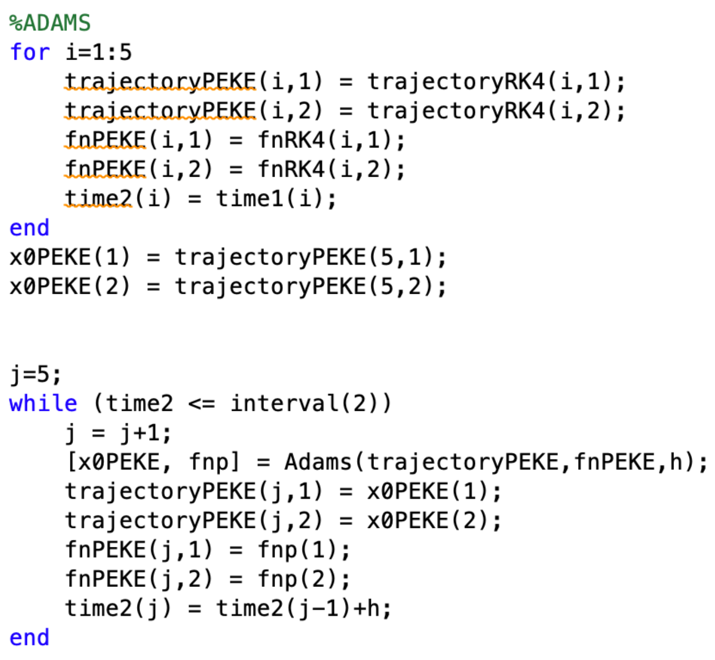
```
end

function [y] = thirdStep(trajectory,fn,fn0,h)
%third step of Adams PEKE
    Beta0 = 475/1440;
    Beta = zeros(1,5);
    Beta(1) = 1427/1440;
    Beta(2) = -798/1440;
    Beta(3) = 482/1440;
    Beta(4) = -173/1440;
    Beta(5) = 27/1440;

    %sum
    sum = [0 0];
    for i=0:4
        sum(1) = sum(1) + Beta(i+1)*fn(end-i,1);
        sum(2) = sum(2) + Beta(i+1)*fn(end-i,2);
    end

    y(1) = trajectory(end,1) + h*sum(1) + h*Beta0*fn0(1);
    y(2) = trajectory(end,2) + h*sum(2) + h*Beta0*fn0(2);
end
```

Figure 28: function used for computing equation with Adam's PECE algorithm



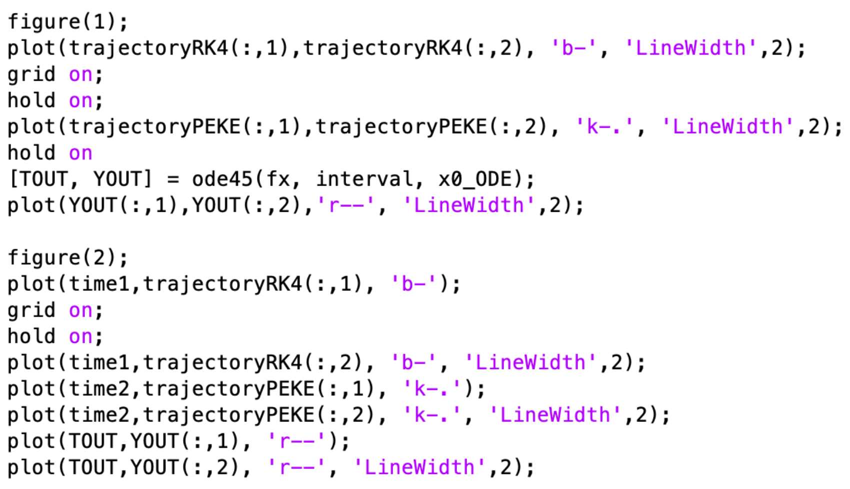
```
x0RK4 =[8 7];
x0PEKE = x0RK4;
x0_ODE = x0RK4;
interval = [0 20];
h=0.021244;

trajectoryRK4=zeros(int64(20/h),2);
fnRK4=zeros(int64(20/h),2);
trajectoryRK4(1,1)=x0RK4(1);
trajectoryRK4(1,2)=x0RK4(2);

fnRK4(1,1) = v1(x0RK4(1), x0RK4(2));
fnRK4(1,2) = v2(x0RK4(1), x0RK4(2));

%ode
% f1 = @(t,x1,x2) (x2+x1*(0.5-(x1^2)-(x2^2)));
% f2 = @(t,x1,x2) (-x1+x2*(0.5-(x1^2)-(x2^2)));
fx = @(t,x)      [ x(2)+x(1)*(0.5-x(1)^2-x(2)^2);
                  -x(1)+x(2)*(0.5-x(1)^2-x(2)^2) ];

time1(1) = interval(1);
j = 1;
while(time1 <= interval(2))
    j = j+1;
    trajectoryRK4(j,:)=yval(x0RK4(1),x0RK4(2),h);
    x0RK4 = trajectoryRK4(j,:);

    fnRK4(j,1) = v1(x0RK4(1), x0RK4(2));
    fnRK4(j,2) = v2(x0RK4(1), x0RK4(2));
    time1(j) = time1(j-1)+h;
end
```

Figure 29: Script used for RK4 method



```
%ADAMS
for i=1:5
    trajectoryPEKE(i,1) = trajectoryRK4(i,1);
    trajectoryPEKE(i,2) = trajectoryRK4(i,2);
    fnPEKE(i,1) = fnRK4(i,1);
    fnPEKE(i,2) = fnRK4(i,2);
    time2(i) = time1(i);
end
x0PEKE(1) = trajectoryPEKE(5,1);
x0PEKE(2) = trajectoryPEKE(5,2);


j=5;
while (time2 <= interval(2))
    j = j+1;
    [x0PEKE, fnp] = Adams(trajectoryPEKE,fnPEKE,h);
    trajectoryPEKE(j,1) = x0PEKE(1);
    trajectoryPEKE(j,2) = x0PEKE(2);
    fnPEKE(j,1) = fnp(1);
    fnPEKE(j,2) = fnp(2);
    time2(j) = time2(j-1)+h;
end
```

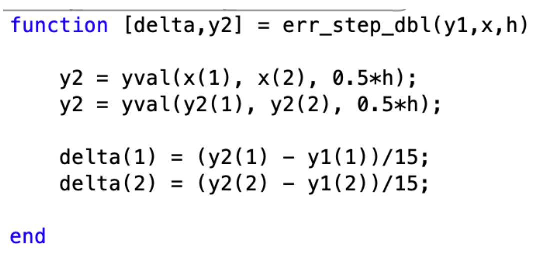
Figure 30: Script used for Adam's PECE method



```
figure(1);
plot(trajectoryRK4(:,1),trajectoryRK4(:,2), 'b-', 'LineWidth',2);
grid on;
hold on;
plot(trajectoryPEKE(:,1),trajectoryPEKE(:,2), 'k-.', 'LineWidth',2);
hold on
[TOUT, YOUT] = ode45(fx, interval, x0_ODE);
plot(YOUT(:,1),YOUT(:,2),'r--', 'LineWidth',2);

figure(2);
plot(time1,trajectoryRK4(:,1), 'b-');
grid on;
hold on;
plot(time1,trajectoryRK4(:,2), 'b-', 'LineWidth',2);
plot(time2,trajectoryPEKE(:,1), 'k-.');
plot(time2,trajectoryPEKE(:,2), 'k-.', 'LineWidth',2);
plot(TOUT,YOUT(:,1), 'r--');
plot(TOUT,YOUT(:,2), 'r--', 'LineWidth',2);
```

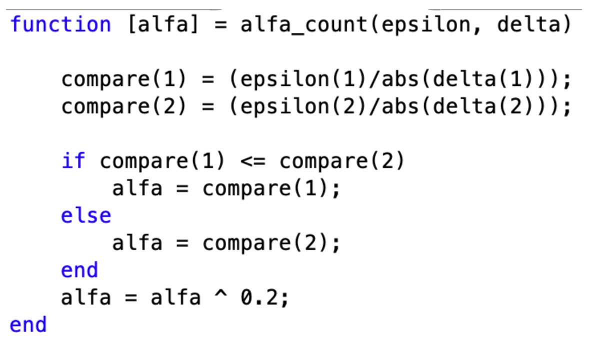
Figure 31: part of script used for making plots

2b:



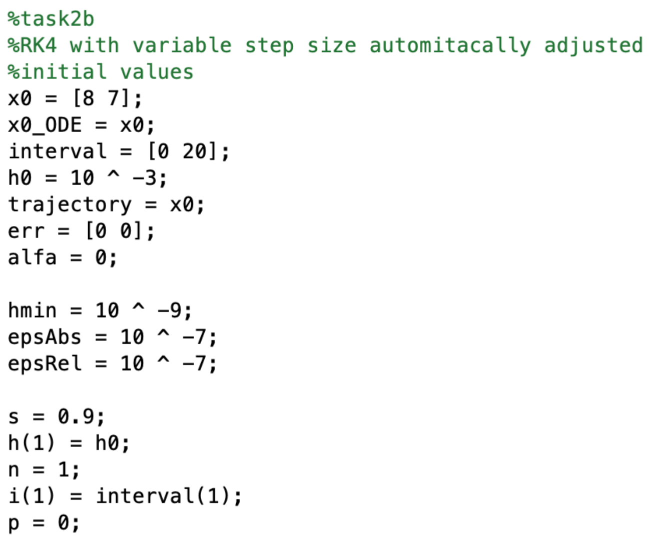
```
function [delta,y2] = err_step_dbl(y1,x,h)

    y2 = yval(x(1), x(2), 0.5*h);
    y2 = yval(y2(1), y2(2), 0.5*h);

    delta(1) = (y2(1) - y1(1))/15;
    delta(2) = (y2(2) - y1(2))/15;

end
```

Figure 32: function counting error estimation with double-step rule



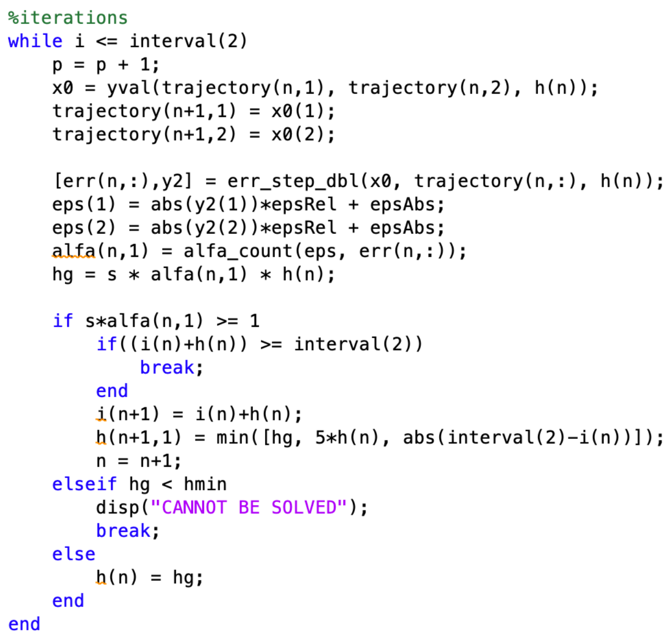
```
function [alfa] = alfa_count(epsilon, delta)

    compare(1) = (epsilon(1)/abs(delta(1)));
    compare(2) = (epsilon(2)/abs(delta(2)));

    if compare(1) <= compare(2)
        alfa = compare(1);
    else
        alfa = compare(2);
    end
    alfa = alfa ^ 0.2;
end
```

Figure 33: function computing alpha coefficient



```
%task2b
%RK4 with variable step size automitacally adjusted
%initial values
x0 = [8 7];
x0_ODE = x0;
interval = [0 20];
h0 = 10 ^ -3;
trajectory = x0;
err = [0 0];
alfa = 0;

hmin = 10 ^ -9;
epsAbs = 10 ^ -7;
epsRel = 10 ^ -7;

s = 0.9;
h(1) = h0;
n = 1;
i(1) = interval(1);
p = 0;
```

Figure 34: initial values of main script of algorithm



```
%iterations
while i <= interval(2)
    p = p + 1;
    x0 = yval(trajectory(n,1), trajectory(n,2), h(n));
    trajectory(n+1,1) = x0(1);
    trajectory(n+1,2) = x0(2);

    [err(n,:),y2] = err_step_dbl(x0, trajectory(n,:), h(n));
    eps(1) = abs(y2(1))*epsRel + epsAbs;
    eps(2) = abs(y2(2))*epsRel + epsAbs;
    alfa(n,1) = alfa_count(eps, err(n,:));
    hg = s * alfa(n,1) * h(n);

    if s*alfa(n,1) >= 1
        if((i(n)+h(n)) >= interval(2))
            break;
        end
        i(n+1) = i(n)+h(n);
        h(n+1,1) = min([hg, 5*h(n), abs(interval(2)-i(n))]);
        n = n+1;
    elseif hg < hmin
        disp("CANNOT BE SOLVED");
        break;
    else
        h(n) = hg;
    end
end
```

Figure 35: loop for computing each iteration of algorithm

```
figure(1);
plot(trajectory(:,1),trajectory(:,2),'b-');
hold on;
fx = @(t,x)      [ x(2)+x(1)*(0.5-x(1)^2-x(2)^2);
                  -x(1)+x(2)*(0.5-x(1)^2-x(2)^2) ];
[TOUT, YOUT] = ode45(fx, interval, x0_ODE);
plot(YOUT(:,1),YOUT(:,2),'r--');

figure(3)
plot(i,h(:,1), 'b-');

figure(4)
plot(i, abs(err(:,1)), 'b-');
grid on;
hold on;
plot(i, abs(err(:,2)), 'r-');

i(end+1) = 20;
figure(2);
plot(i,trajectory(:,1), 'b-');
grid on;
hold on;
plot(i,trajectory(:,2), 'b-', 'LineWidth',2);
% hold on
% [TOUT, YOUT] = ode45(fx, interval, x0_ODE);
plot(TOUT(:,1),YOUT(:,1),'r--');
plot(TOUT(:,1),YOUT(:,2),'r--', 'LineWidth',2);
```

Figure 36: part of script used for making plots