Chmielewska Urszula

300167

Numerical Methods

Project C No. #14

Advisor: Dr inż. Piotr Marusak

TASK 1.

For the following experimental measurements:

x_i	y_i
-5	–32.4591
-4	–20.2011
-3	–12.1986
-2	–4.6508
-1	–1.1893
0	0.6266
1	0.5743
2	–0.3709
3	–1.2371
4	–3.4952
5	–4.3987

Determine a polynomial function $y = f(x)$ that best fits the experimental data by using the least-square approximation. The obtained functions will be presented as a graphs with the the experimental data. To solve the least-squares problem, the system of normal equations with QR factorization of a matrix A will be used. For each solution calculate the error defined as the Euclidean norm of the vector of residuum and the condition number of the Gram's matrix. Make a comparison of the results in terms of solutions' errors.

Theoretical background

Least-squares problem

Least-squared problems can be treated as problems in which it is necessary to solve a set of m linear equations with n unknowns, where n>m. The obtained solution may not be unique, therefore a minimal norm of the solution is usually required to make it unique.

For the cases in which the matrix composed of the set of equations may be ill-conditioned, it is recommended to use the QR factorization to obtain the solution. However to LLSP, the narrow, not the normalized QR factorization, can be applied. The system of equations obtained from QR factorization needs less computations than the normal system of linear equations, because it creates a triangular matrix.

Discrete least-squares approximation

The main idea of the approximation is to find values of parameters a_0, a_1, … , a_n defining the approximating function:

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

which minimize the least-squares error defined by:

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

The way to find those parameters is to form a system of linear equations called the set of normal equations. Such matrix is known as the **Gram's matrix**.

If we define the scalar products, the normal equations can be written as:

$$\langle \Phi_i, \Phi_k \rangle = \sum_{j=0}^{N} \Phi_i(x_j) \cdot \Phi_k(x_j)$$

Then, the set of normal equations takes a form:

$$\begin{bmatrix} \langle \Phi_0, \Phi_0 \rangle & \langle \Phi_1, \Phi_0 \rangle & \dots & \langle \Phi_n, \Phi_0 \rangle \\ \langle \Phi_0, \Phi_1 \rangle & \langle \Phi_1, \Phi_1 \rangle & \vdots & \langle \Phi_n, \Phi_1 \rangle \\ \vdots & \vdots & & \vdots \\ \langle \Phi_0, \Phi_n \rangle & \langle \Phi_1, \Phi_n \rangle & \dots & \langle \Phi_n, \Phi_n \rangle \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_2 \end{bmatrix} = \begin{bmatrix} \langle \Phi_0, f \rangle \\ \langle \Phi_1, f \rangle \\ \vdots \\ \langle \Phi_n, f \rangle \end{bmatrix}$$

Polynomial approximation

Polynomials are often used as approximating functions. Usually, the order n of the approximating polynomial is much lower than the number of data points given N, so $\boldsymbol{N >> n}$.

Natural polynomial basis, also called the power basis is defined by:

$$\Phi_0(x) = 1, \quad \Phi_1(x) = x, \quad \Phi_2(x) = x^2, \quad \ldots, \quad \Phi_n(x) = x^n$$

and in other form:

$$F(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

The system of normal equations can be presented as follows:

$$\boldsymbol{G} \cdot \boldsymbol{a} = \varrho$$

Where **G** is the **Gram's matrix** composed of elements g_{ik},

$$g_{ik} = \sum_{j=0}^{N} (x_j)^{i+k}$$

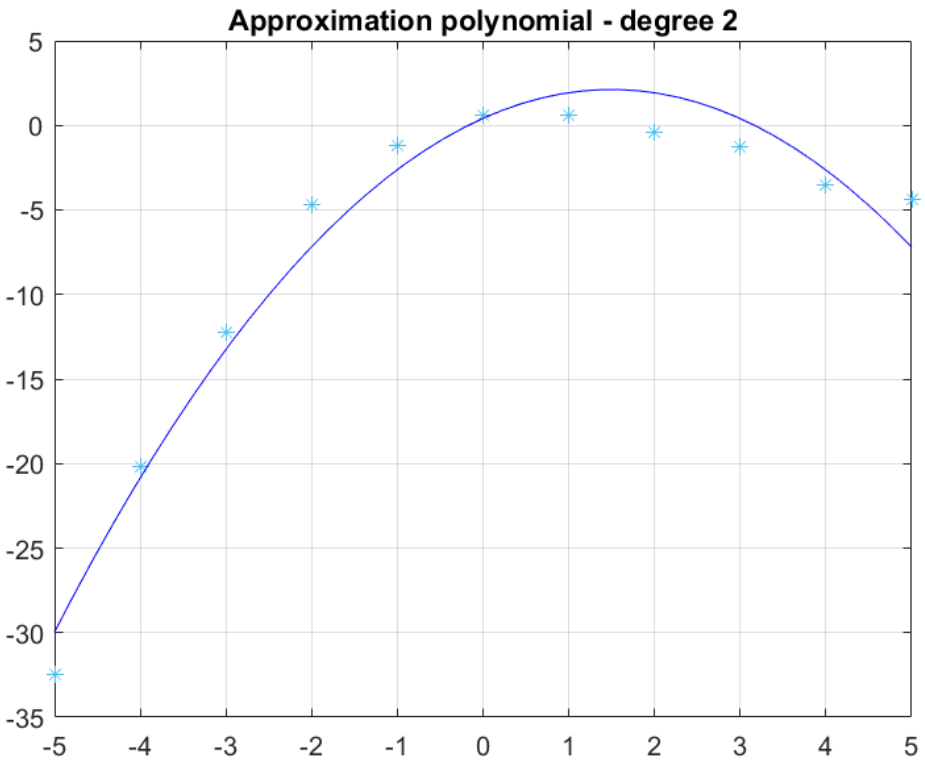
Elements of a, are the unknown parameters a_0, a_1, …, a_n, and ϱ is the vector with elements defined as:

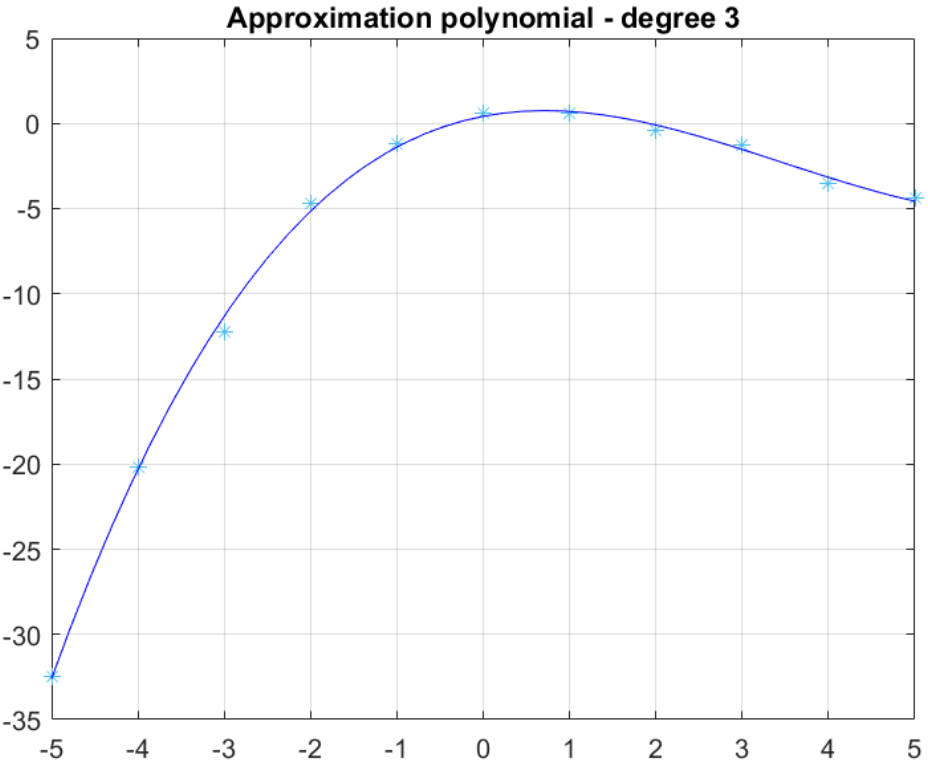
$$\varrho_k = \sum_{j=0}^{N} f(x_j)(x_j)^k$$

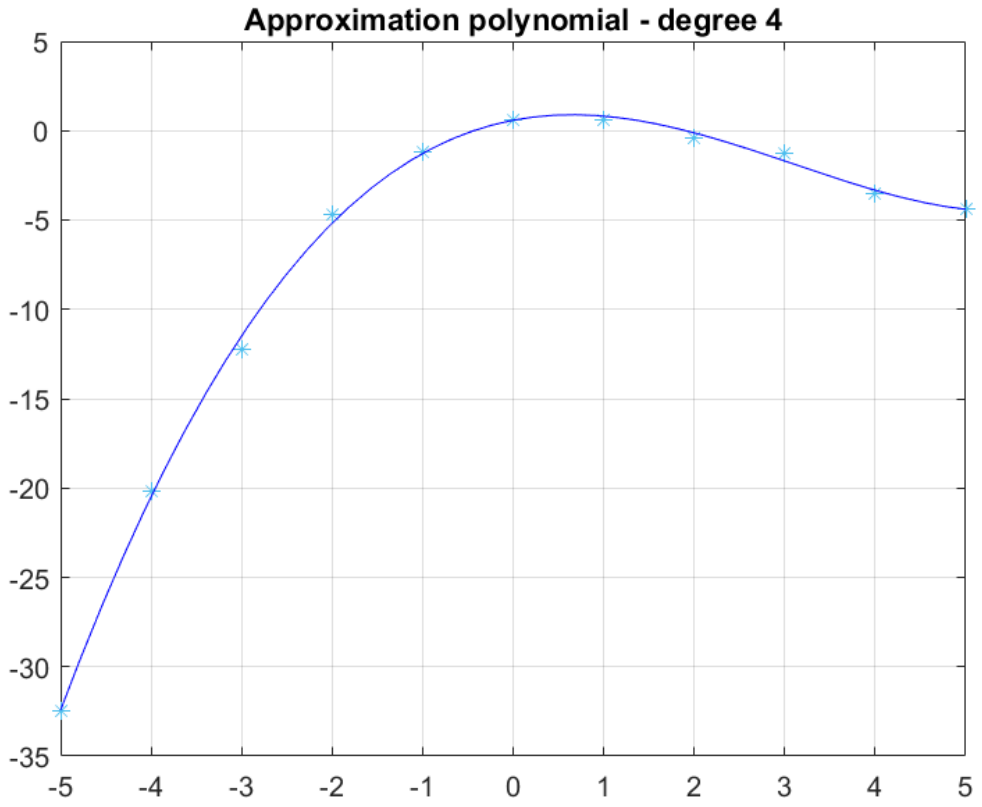
Moreover, for the system of normal equations, Gram's matrix is of Hilbert type, which is known as a one which very quickly becomes ill-conditioned with an increase of the order of the polynomial. The way to avoid ill-conditioning is to apply QR factorization to orthogonalize the polynomials in the basis. When the basis functions are orthogonal, then the Gram's matrix is a diagonal one and we obtain a well-conditioned set of equations.

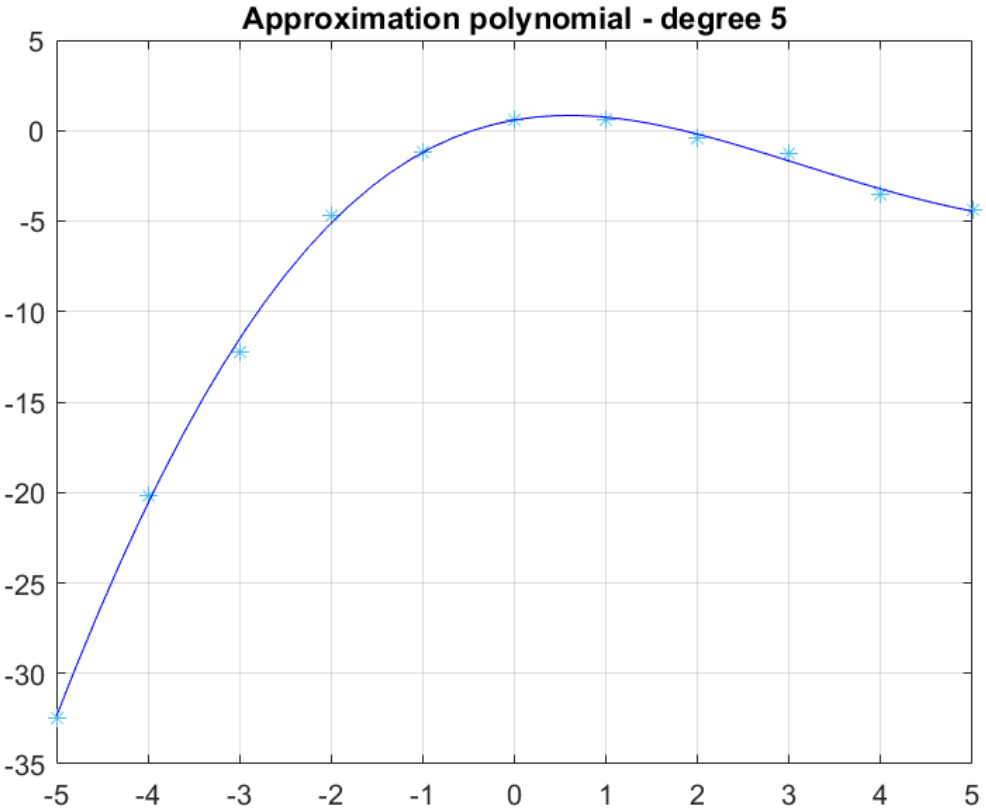
Results

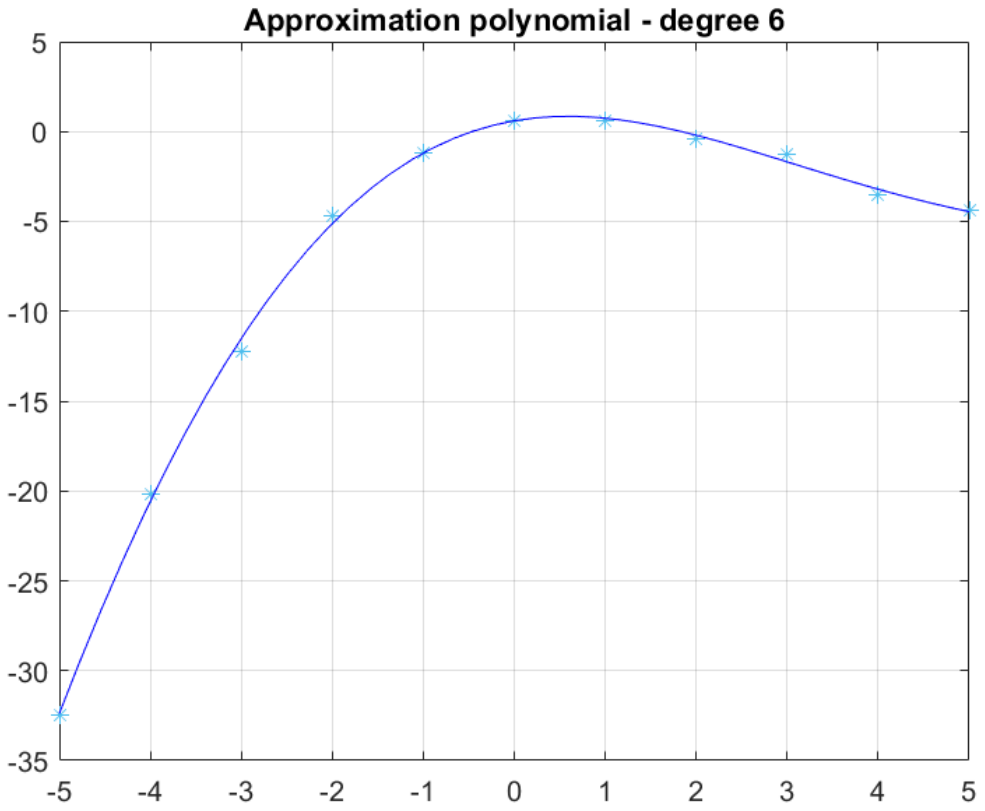
Graphs of approximated polynomials with increasing polynomial degree

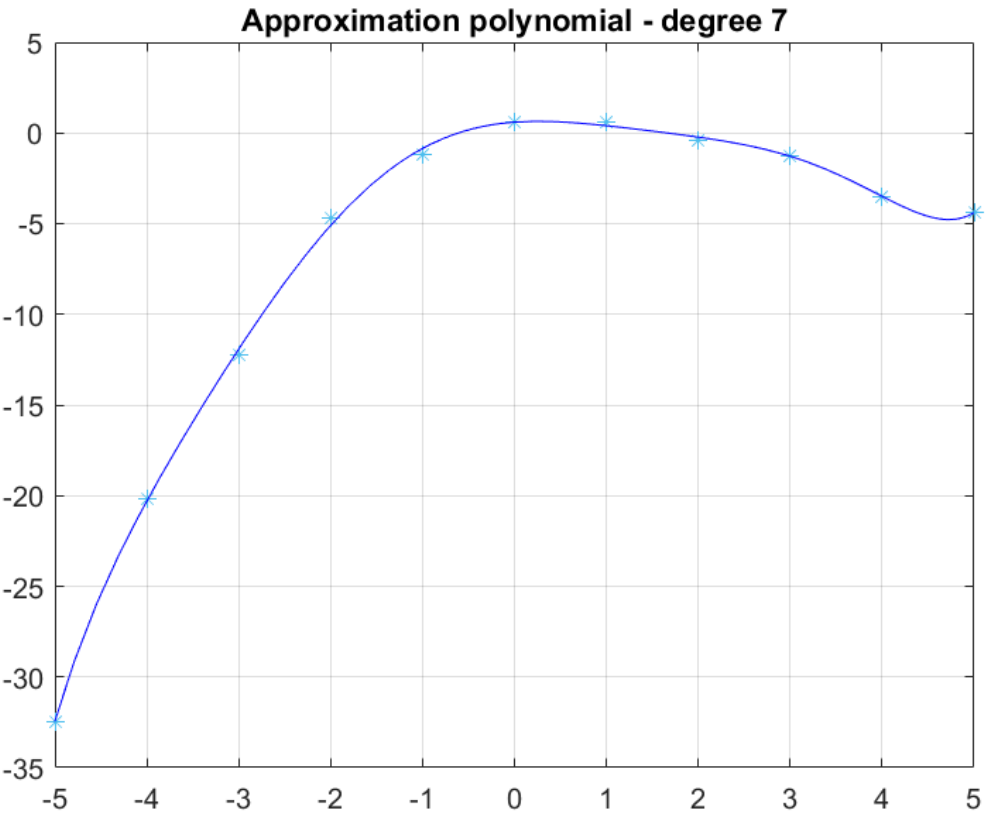




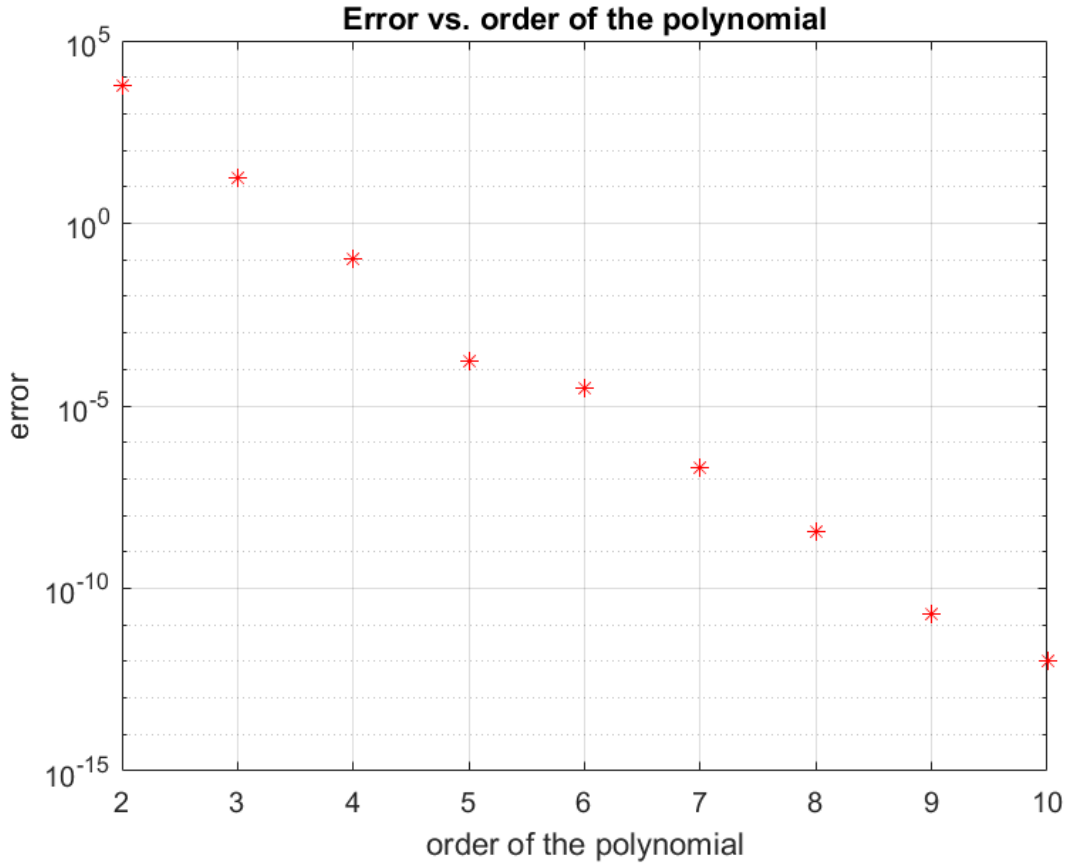

Approximation polynomial - degree 4
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Approximation polynomial - degree 5
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Approximation polynomial - degree 6
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Approximation polynomial - degree 7
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

The relation between solution error and order of the polynomial



Polynomial order	Error	Condition number
2	5.9003e+03	408.7796
3	16.8897	8.5584e+03
4	0.1046	3.1798e+05
5	1.6427e-04	7.4675e+06
6	3.0812e-05	2.8316e+08
7	2.0032e-07	7.6462e+09
8	3.4796e-09	3.3055e+11
9	2.0682e-11	1.5167e+13
10	1.0804e-12	9.2930e+14

Conclusions

While analyzing the graphs representing approximated polynomials with different order, we can observe that approximated polynomial with order 2 did not fit data points perfectly. We can notice that we have obtained a significant improvement, in an approximated function shape, already for order 3. With increasing polynomial order, function covers data points better. For polynomial of order 7, points are covered the most accurately out of all presented cases.

Furthermore, it is necessary to analyze also error for each polynomial order. Error was defined as the Euclidean norm of the vector of residuum. There is a high difference between error for polynomial order 2 and order 10. For 2, error was equal to 5.9003e+03 and for order 10 we obtained a significant improvement, because error was equal to 1.0804e-12 (very close to 0).

The above results and conclusions are caused by a fact that in case of polynomial of order 2, we have only 3 parameters for approximating 11 data points. On the contrary, in case of polynomial of order 10, we have 10 parameters for approximating the same number of data points. Due to this fact, higher order of the polynomial gives us a much better results in solving a set of equations.

As it was stated in the theoretical background for the polynomial approximation section, the Gram's matrix for the system of normal equations obtained for the approximating polynomials based on power basis, quickly becomes ill-conditioned. It can be prove by observing the table with condition numbers for different polynomial orders. The condition numbers grow rapidly.

TASK 2.

A motion of a point is given by equations:

$$\frac{dx_1}{dt} = x_2 + x_1(0.5 - {x_1}^2 - {x_2}^2)$$

$$\frac{dx_2}{dt} = -x_1 + x_2(0.5 - {x_1}^2 - {x_2}^2)$$

Determine the trajectory of the motion on the interval [0, 20] for the following initial conditions:

$x_1(0) = -0.001$

$x_2(0) = -0.02$

Then evaluate the solution using:

a) Runge-Kutta method of 4^{th} order (RK4) and Adams PC (P_5EC_4E – 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 solution significantly but its increase does,
b) Runge-Kutta method of 4^{th} order (RK4) with a variable step size automatically adjust by the algorithm, making error estimation according to the step-doubling rule.

Compare the results with ones obtained using an ODE solver (ode45).

Theoretical background 2a

Runge – Kutta methods

Family of Runge-Kutta methods are described as:

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

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

$$k_i = f(x_n + c_i h, y_n + h \cdot \sum_{j=1}^{i-1} a_{ij} k_j$$

$$\sum_{j=1}^{i-1} a_{ij} = c_i$$

Each step of the method requires precise calculations, of the values of the right-hand sides of the equation, *m times.* This is the reason why the method can be described as an *m–stage* one. Parameters a_{ij}, w_i, c_i are not unique.

The most important parameters are *m = 4* and *p = 4*, because of their goof numerical properties.

The Runge – Kutta method of order 4 (Rk4, "classical")

$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

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

$$k_2 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1)$$

$$k_3 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2)$$

$$k_4 = f(x_n + h, y_n + hk_3)$$

The 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. This is an explanation of the first formula in this section.

Step-size selection

A decision about the right step-size is a fundamental problem for solving differential equations, because:

1. If the step h_n becomes smaller, then the approximation error of the method becomes smaller, **but**
2. If the step h_n becomes smaller, then also the number of steps needed to find a solution increases and therefore, the number of arithmetic calculations to find a solution increases and it also increases the numerical roundoff errors

To sum up, it is necessary to take sufficiently small step to perform calculations with a desired accuracy, but it should not be smaller than necessary.

Error estimation (the step-doubling approach)

To estimate the approximation error, for every step of the size h, two additional steps of the size $h/2$ are additionally performed in parallel. The notation is as follows:

$y_n^{(1)}$ – a new point obtained using the step-size *h*

$y_n^{(2)}$ – a new point obtained using two consecutive steps of size $\frac{h}{2}$

From the equation evaluations we obtain the following expression:

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

which can be treated as an estimate of the error of two consecutive steps of the size $h/2$.

Multistep methods

A single iteration of a linear multistep method with a constant step-size h can be defined by:

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

$$y_0 = y(x_0) = y_a, \qquad x_n = x_0 + nh, \qquad x \in [a = x_0, b]$$

A multistep method is explicit if $\beta_0 = 0$. In such case, the value y_n depends explicitly on values of the solution and its derivative at previously calculated points only.

The implicitly of the method is obtained when $\beta_0 \neq 0$. In such case, the value y_n depends not only on previously calculated points, but also on current point. Therefore, in order to calculate y_n, it is necessary to first perform a starting procedure to evaluate values and current point, for example by using RK4 method.

Predictor-corrector methods

The following properties are required for the best multistep:

1. A high order and a small error constant
2. A large set of the absolute stability
3. A small number of arithmetic operations performed during one iteration

The explicit methods fulfill the last property, while implicit methods fulfill the first two properties. Therefore the best approach is a *predictor-corrector structure* (PC) which combines the explicit and implicit method into one algorithm.

Adams PC (P_5EC_5E)

Algorithm used to solve this task has a following form:

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

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

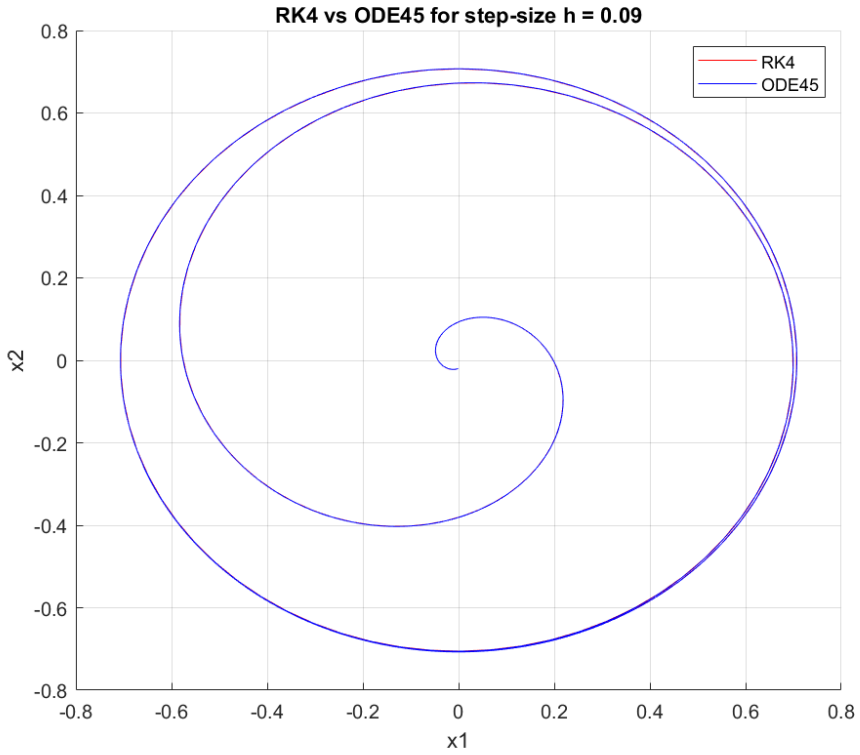
C: $y_n = y_{n-1} + h \sum_{j=1}^{k} \beta_j^* f_{n-j} + h\beta_0^* f_n^{[0]}$

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

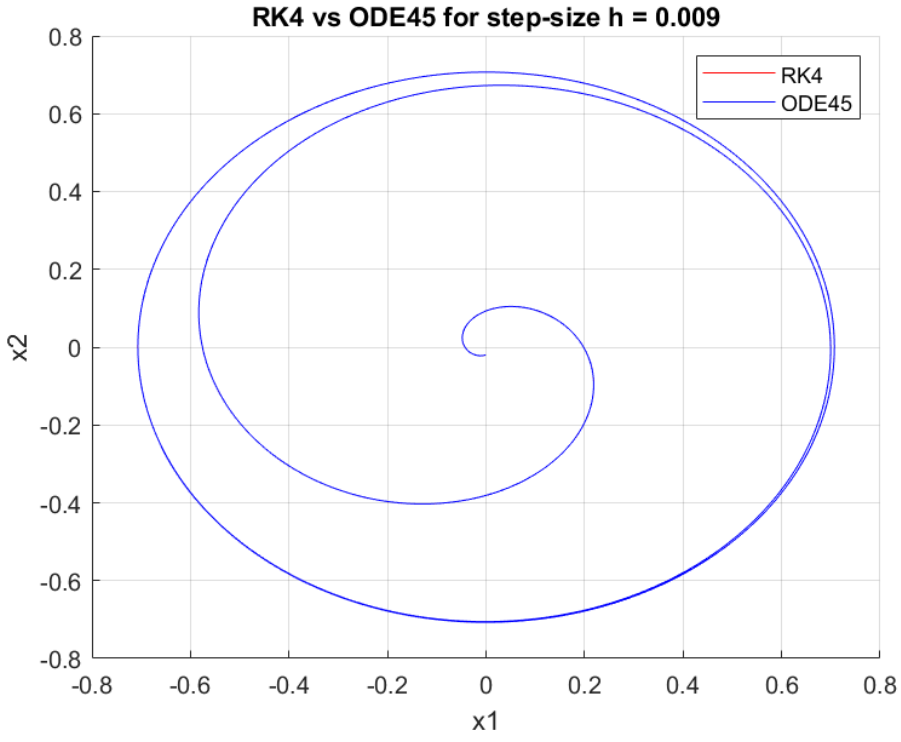
Results

Runge-Kutta method of 4th order

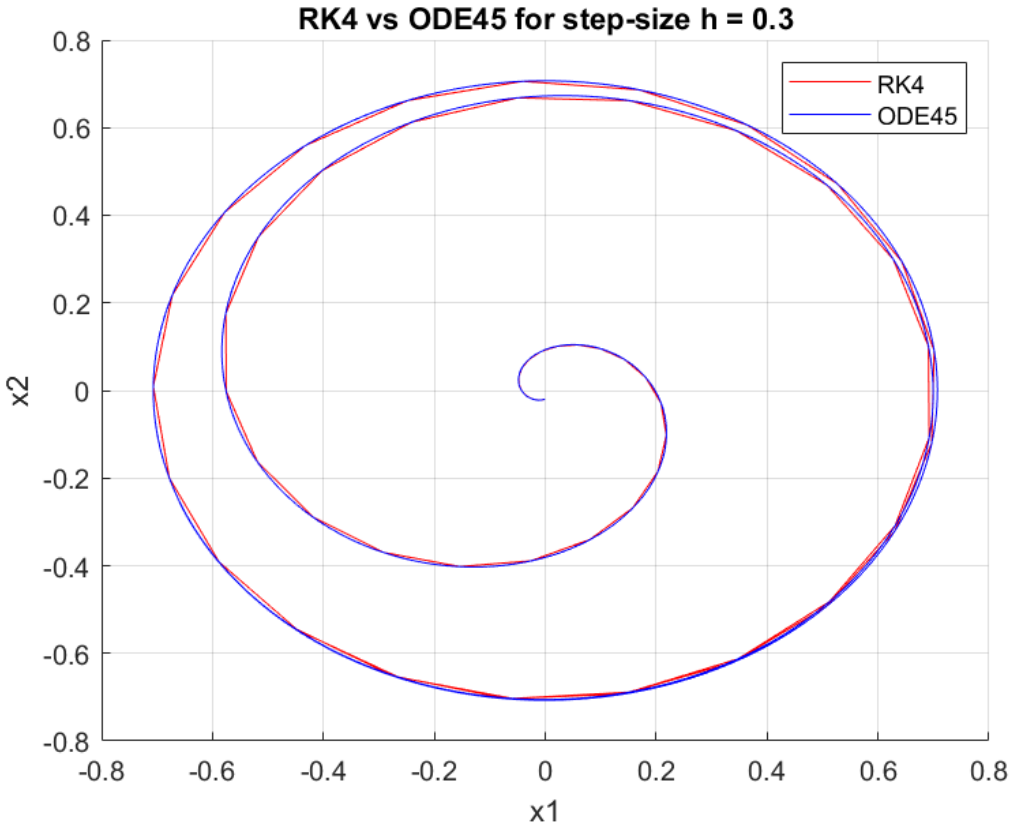
Optimal step-size



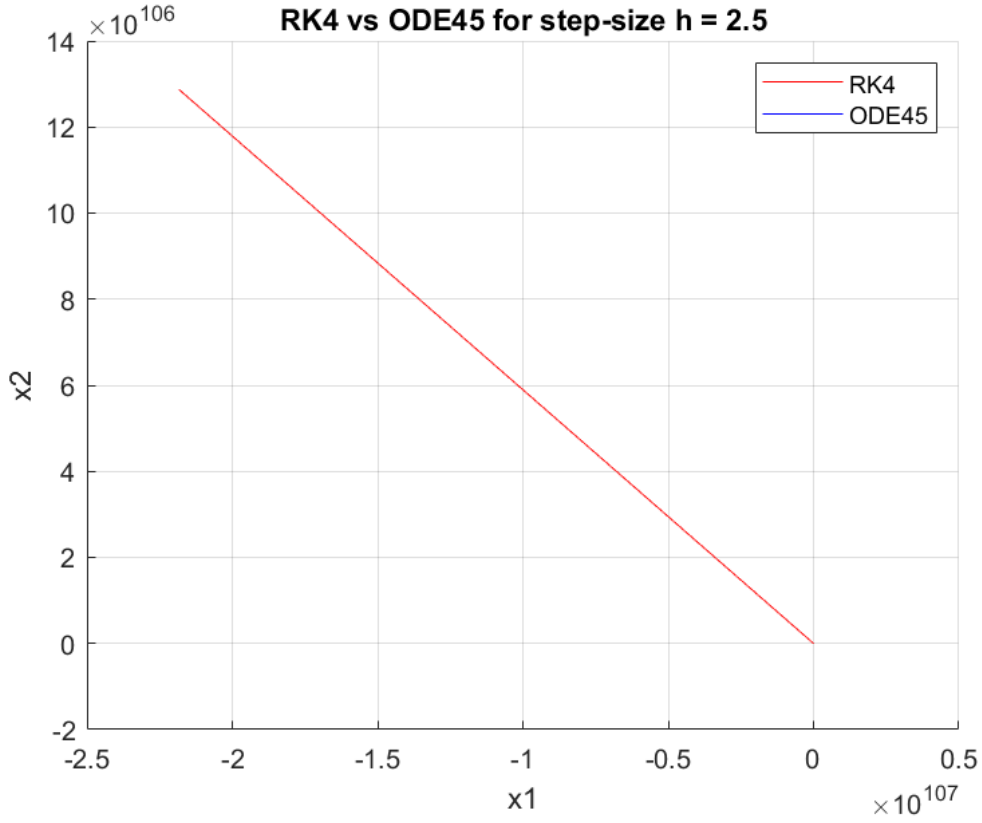
Step-size decreased – no improvement



Step-size increased – visible influence

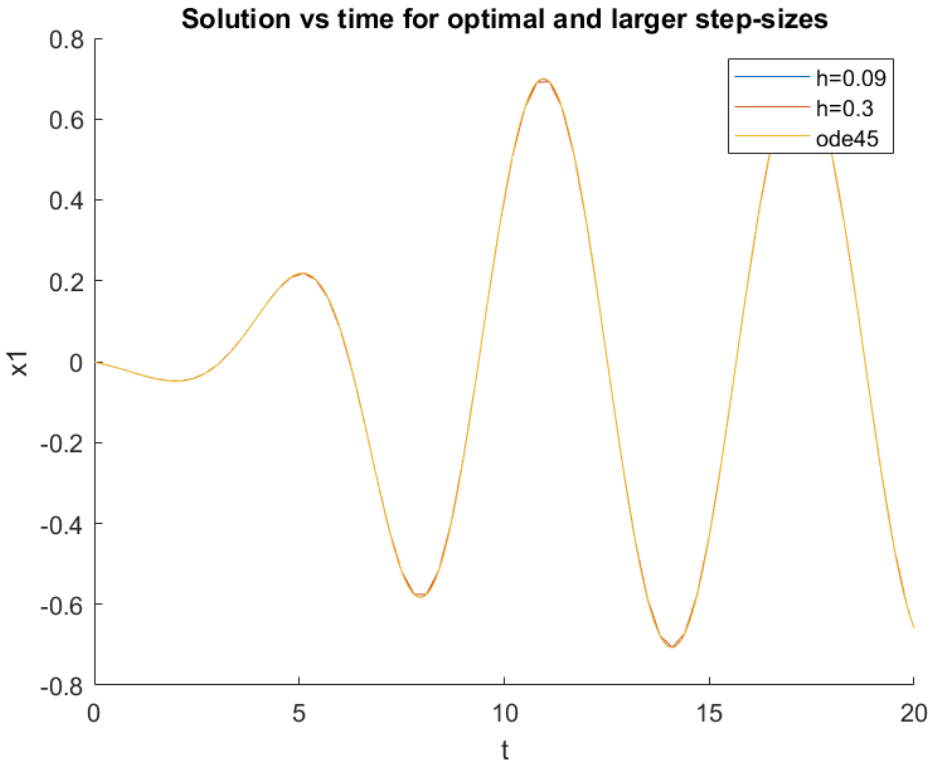


Step-size too big

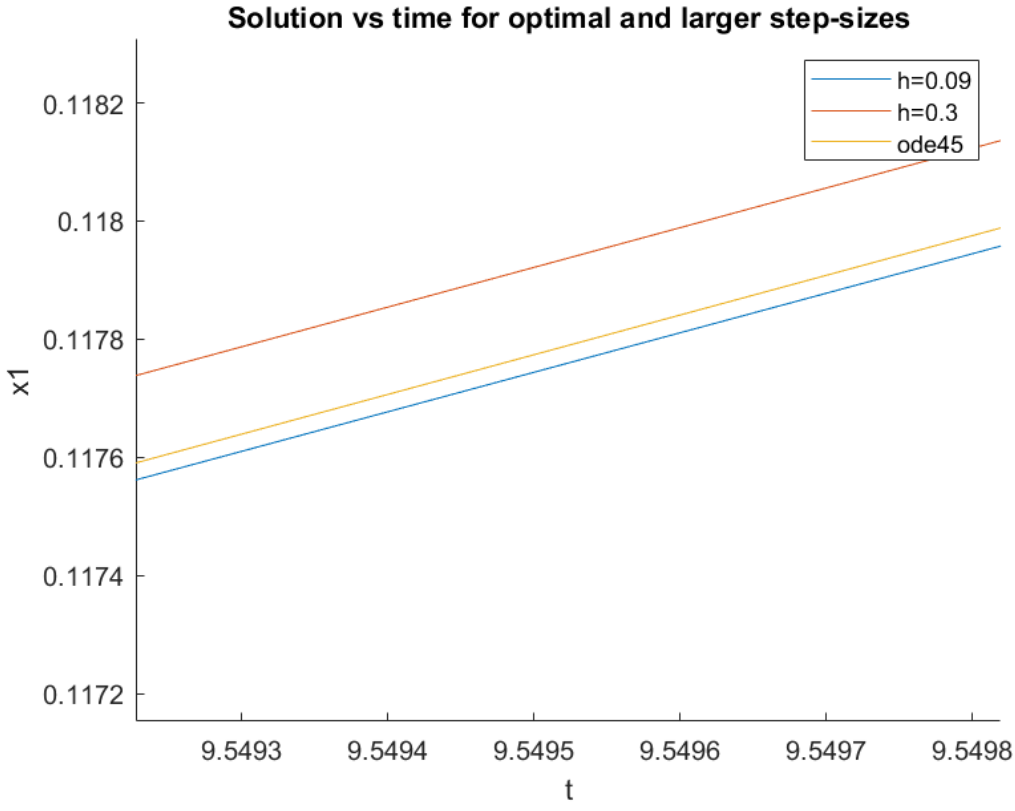


Solution vs time

$x_1(t)$

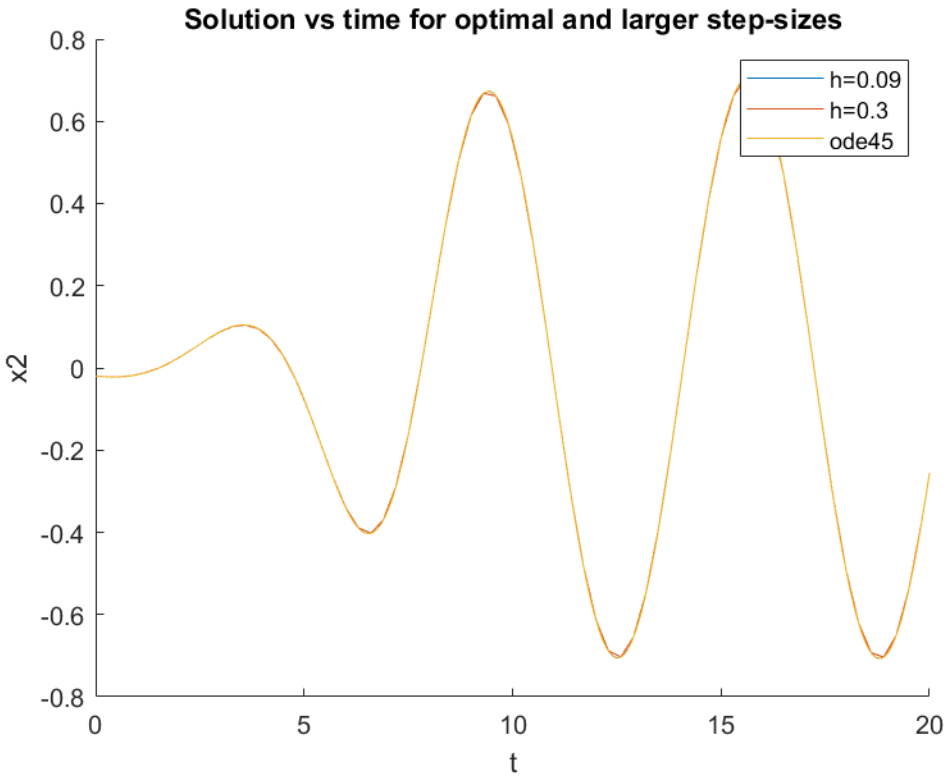


Zoomed in solution vs time

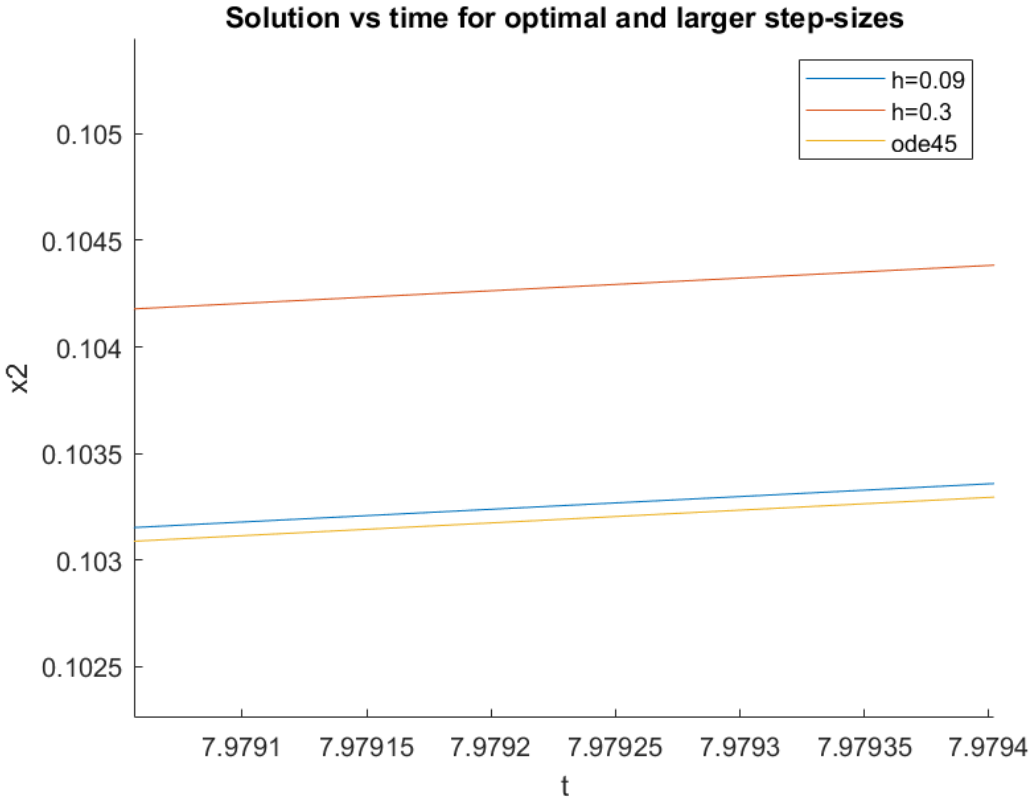


Solution vs time

$x_2(t)$

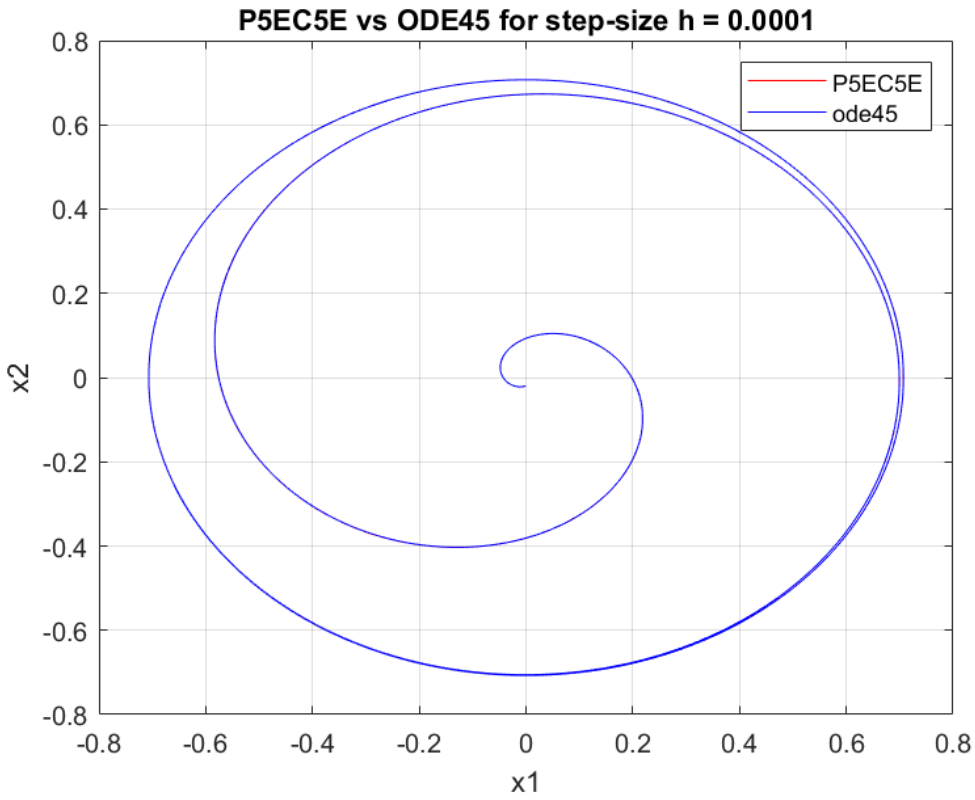


Zoomed in solution vs time

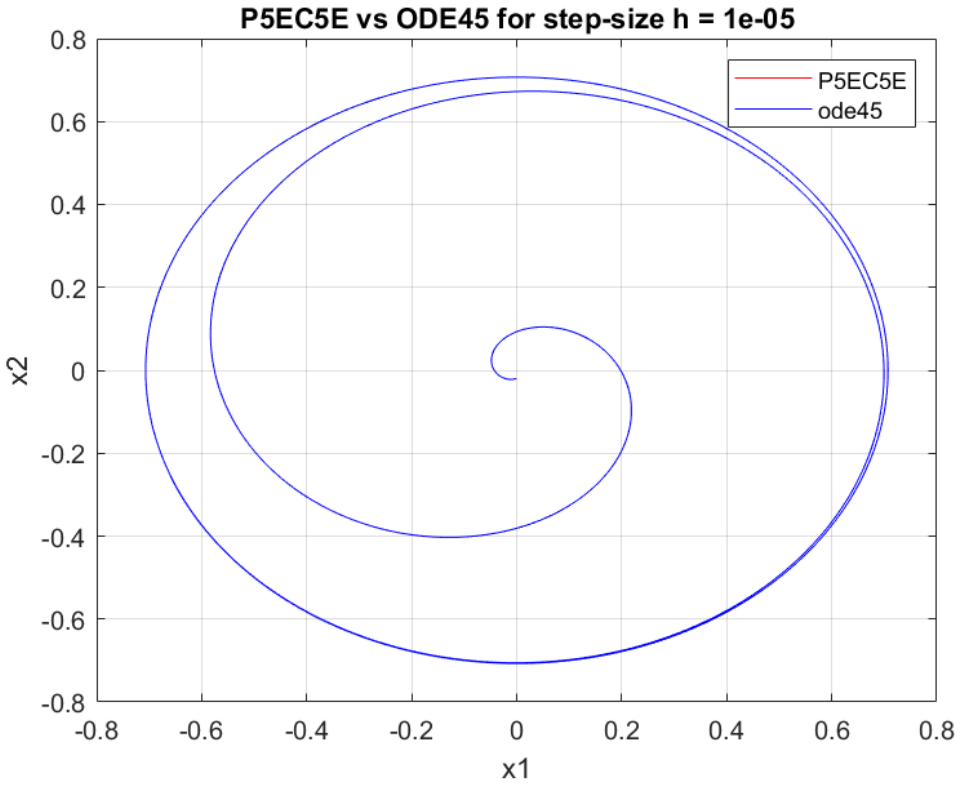


Adams PC P_5EC_5E

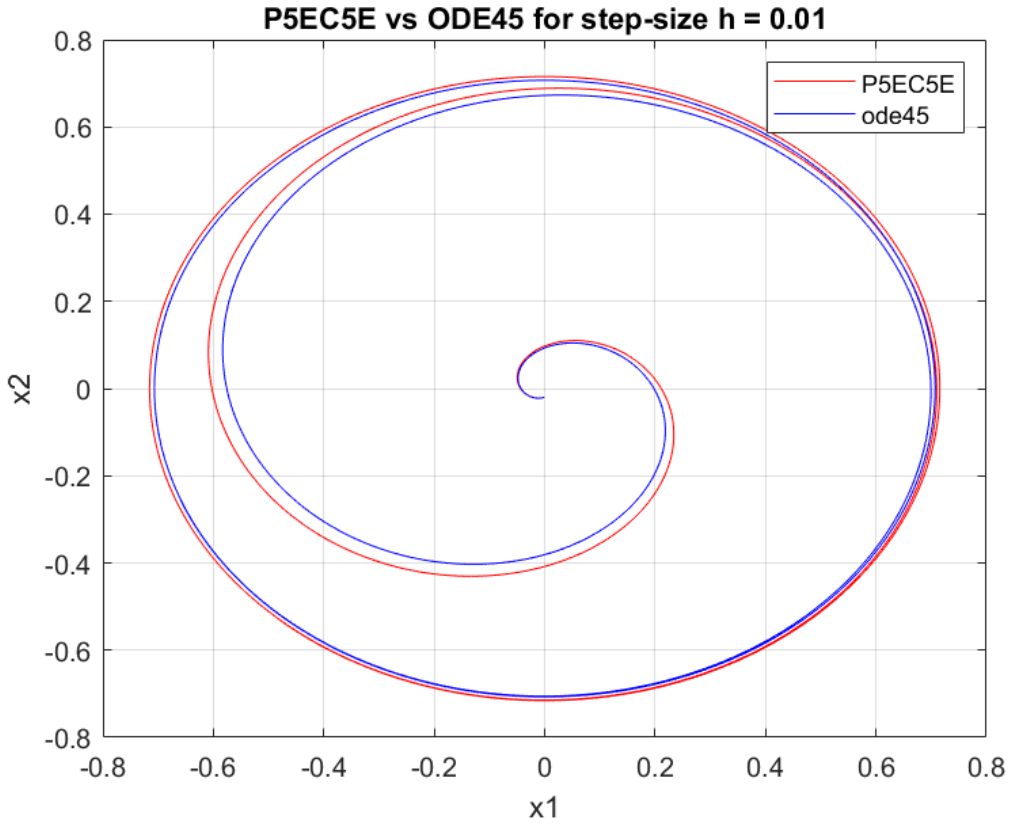
Optimal step-size



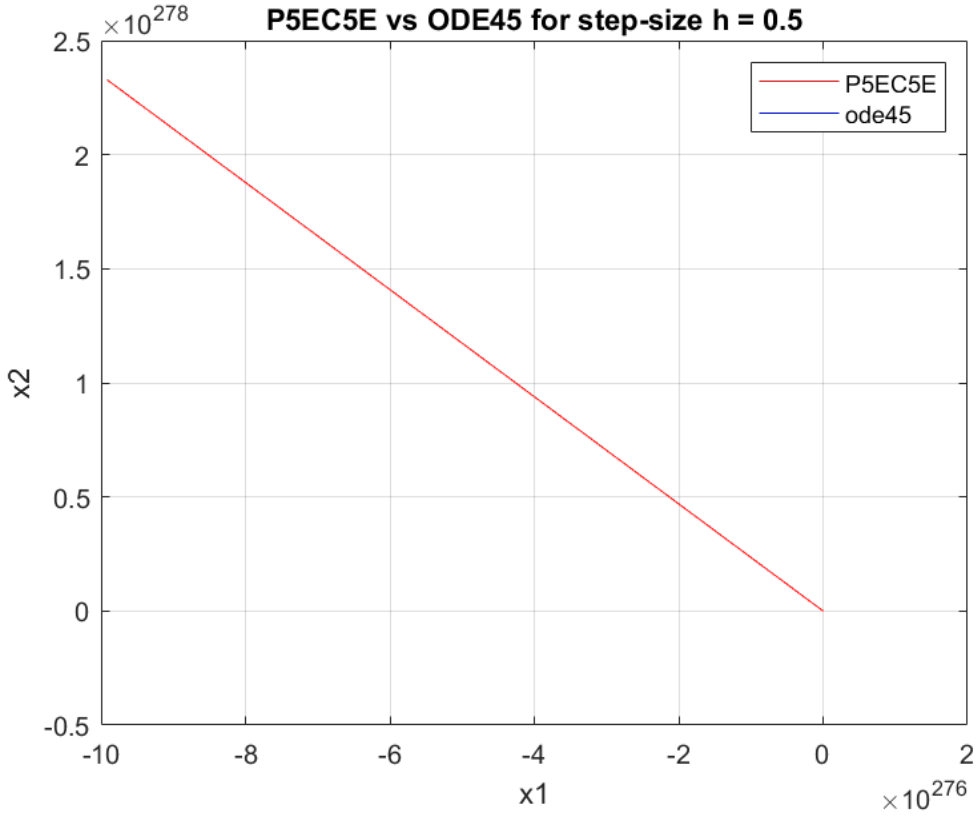
Step-size decreased – no improvement



Step-size increased – visible influence

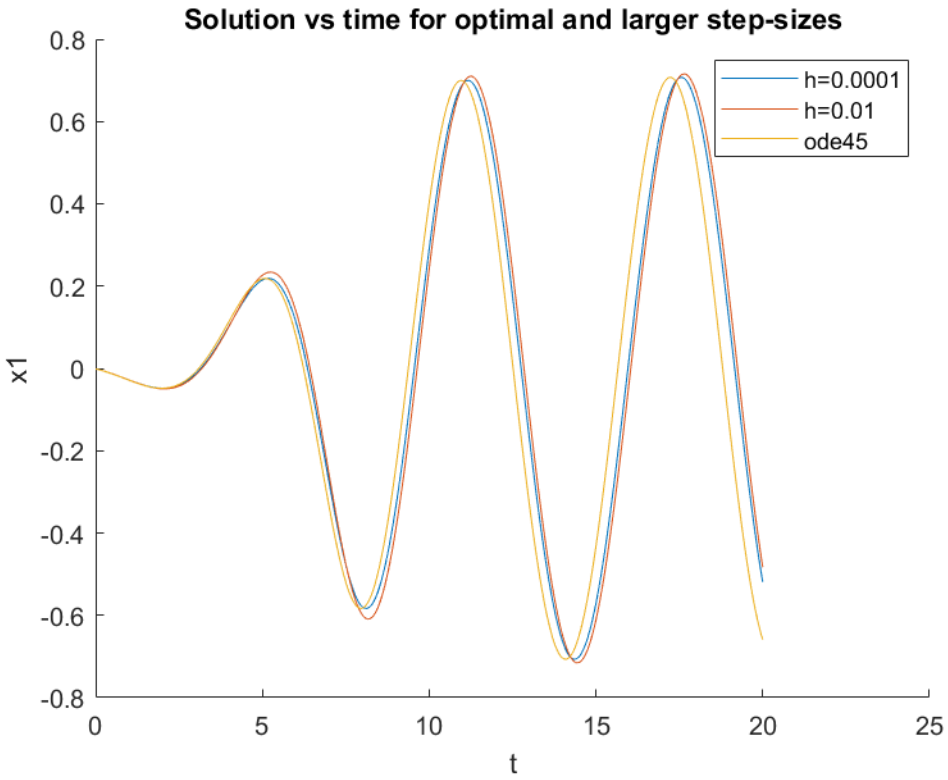


Step-size too big

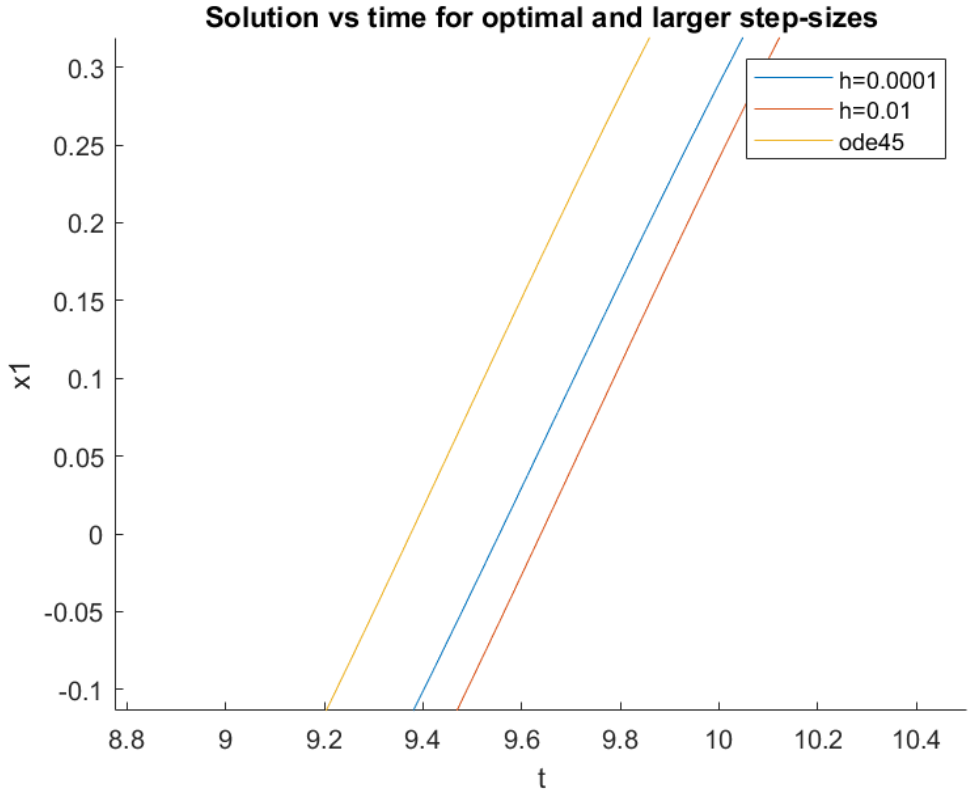


Solution vs time

$x_1(t)$

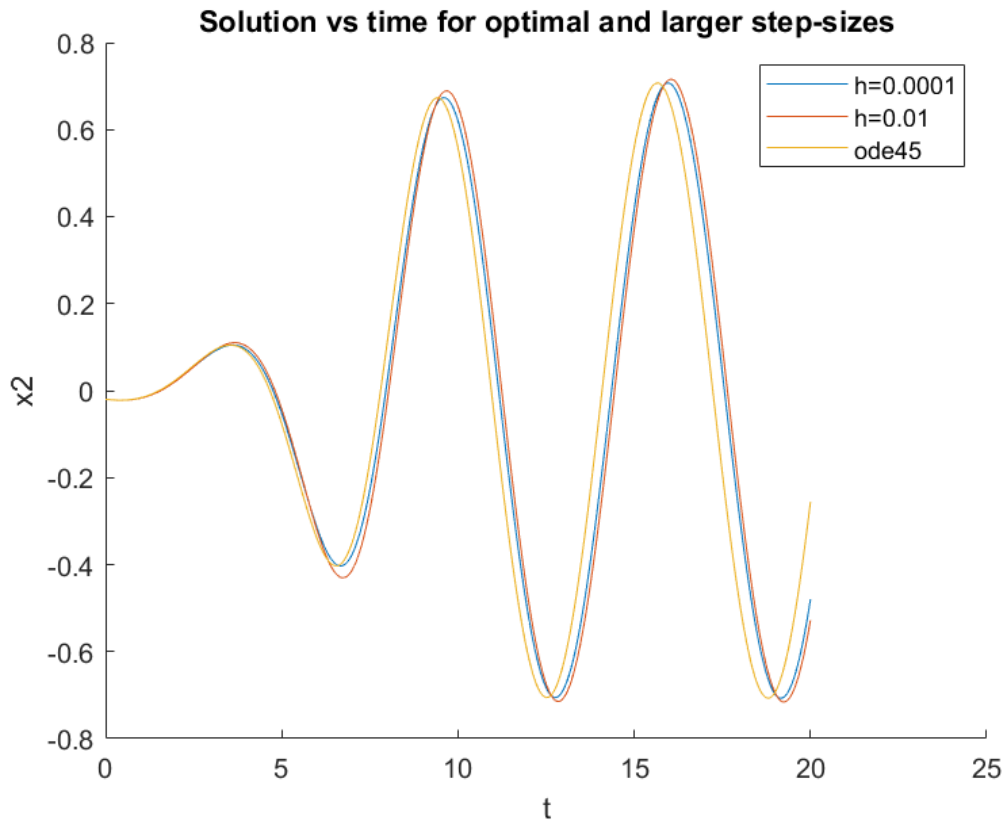


Zoomed in solution vs time

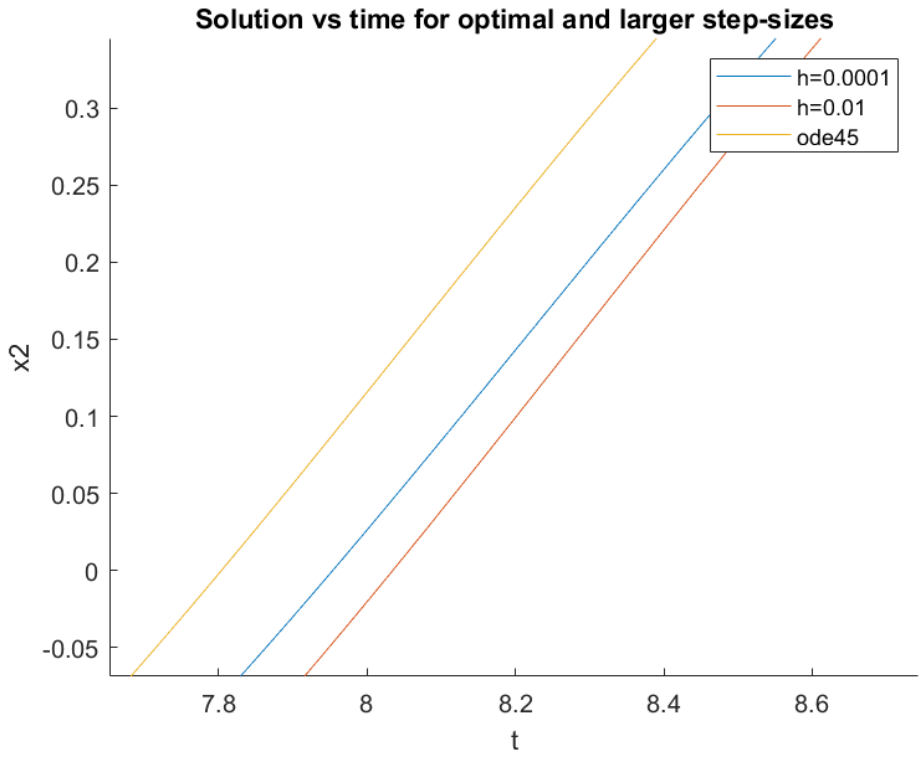


Solution vs time

$x_2(t)$



Zoomed in solution vs time



Conclusions

The goal of this task was to choose proper step-sizes for both methods, Runge-Kutta of order 4^{th} and Adams PC P_5EC_5E. While performing tests for different step-sizes values, I noticed that the decrease of step caused that the approximation error also decreased, however it increased the number of needed arithmetic operations. More arithmetic operations mean more roundoff errors and less efficient program. Both methods are very sensitive to the changes of the step-sizes. It was very easy to observe the influence of single digit change even in the case with step-sizes smaller than 0.001.

Optimal step-size results from my experiments are:

RK4: 0.09

Adams PC: 0.0001

RK4 method failed for step-size equal to 2.5, while Adams PC method failed for 0.5. There were no solutions for these and higher values.

Furthermore, while observing the plots of the problem solution versus time for RK4 method, the difference between plots of the optimal step-size and larger one is visible while we zoom in the graph. We can also notice, that the solution obtained with optimal step-size is much closer to ODE solver, than the solution with a larger step-size.

On the other hand, from my experiments I obtained more significant differences between plots of the problem solutions versus time for Adams PC method while combining optimal step-size, larger one and ODE solver solution. Shapes of these plots differ significantly even without zooming in, however plot of my optimal step-size solution is still closer to the ODE solver plot.

From the above results and observations, we can conclude that both, Runge-Kutta and Adams PC methods, are sensitive to the step-size changes, however the Adams PC method proved to be more sensitive in this aspect than Runge-Kutta's. It is also important to state, that both methods gave us reliable results, however due to the fact that while comparing problem solution versus time, the obtained results were worse for Adams PC method, we can conclude that Adams PC method is less reliable or requires more precise step-size selection.

Theoretical background 2b

The goal of this task was to perform an automatic step-size adjustment instead of experimentally testing proper step-sizes by hand as it was conducted in the previous task 2a. The theoretical background for Runge-Kutta of 4th order method for this task is the same as it was described in theoretical background for task 2a.

Error estimation – the step-doubling approach

To perform error estimation using the step-doubling approach it is necessary to for every step h, perform two additional steps of size $\frac{h}{2}$ in parallel. Let's denote:

$y_n^{(1)}$ - a new point obtained using the step-size h

$y_n^{(2)}$ - a new point obtained using two consecutive steps of the size $\frac{h}{2}$

$r^{(1)}$ - the approximation error after the single step h

$r^{(2)}$ - the summed approximation errors after two smaller steps (of length $\frac{h}{2}$ each)

We can simplify the assumption by assuming the same approximation error for each of the smaller steps of the size $\frac{h}{2}$:

After a single step: $$y(x_n + h) = y_n^{(1)} + \frac{r_n^{(p+1)}(0)}{(p+1)!} \cdot h^{p+1} + O(h^{p+2})$$

After a double step: $$y(x_n + h) \cong y_n^{(2)} + 2 \cdot \frac{r_n^{(p+1)}(0)}{(p+1)!}\left(\frac{h}{2}\right)^{p+1} + O(h^{p+2})$$

Where red parts are main parts of the error.

Error estimate for a single step-size h: $$\delta_n(h) = \frac{2^p}{2^p - 1}(y_n^{(2)} - y_n^{(1)})$$

Error estimate for two consecutive steps of the size $\frac{h}{2}$: $$\delta_n\left(2 \cdot \frac{h}{2}\right) = \frac{y_n^{(2)} - y_n^{(1)}}{2^p - 1}$$

which is 2^p times smaller than $\delta_n(h)$.

Correction of the step-size

A general formula for the main part of the approximation error, after a step of a length h, is:

$$\delta_n(h) = \gamma \cdot h^{p+1}, \text{ where } \gamma = \frac{r_n^{(p+1)}(0)}{(p+1)!}$$

If we change step-size to αh and assuming tolerance ϵ: $|\delta_n(\alpha h)| = \epsilon$

therefore: $\delta_n(\alpha h) = \alpha^{p+1} \cdot \delta_n(h)$

Also, the coefficient α for the step-size correction can be evaluated as: $\alpha = \left(\frac{\epsilon}{|\delta_n(h)|}\right)^{\frac{1}{p+1}}$

which is correct for the step-size h, as well as for the step-size $\frac{h}{2}$.

Moreover it is necessary to take into the consideration the lack of accuracy in the error estimation. It can be achieved by the use of safety factor:

$$h_{n+1} = s \cdot \alpha \cdot h_n \qquad \text{where } s < 1$$

The tolerance parameters should be defined as follows:

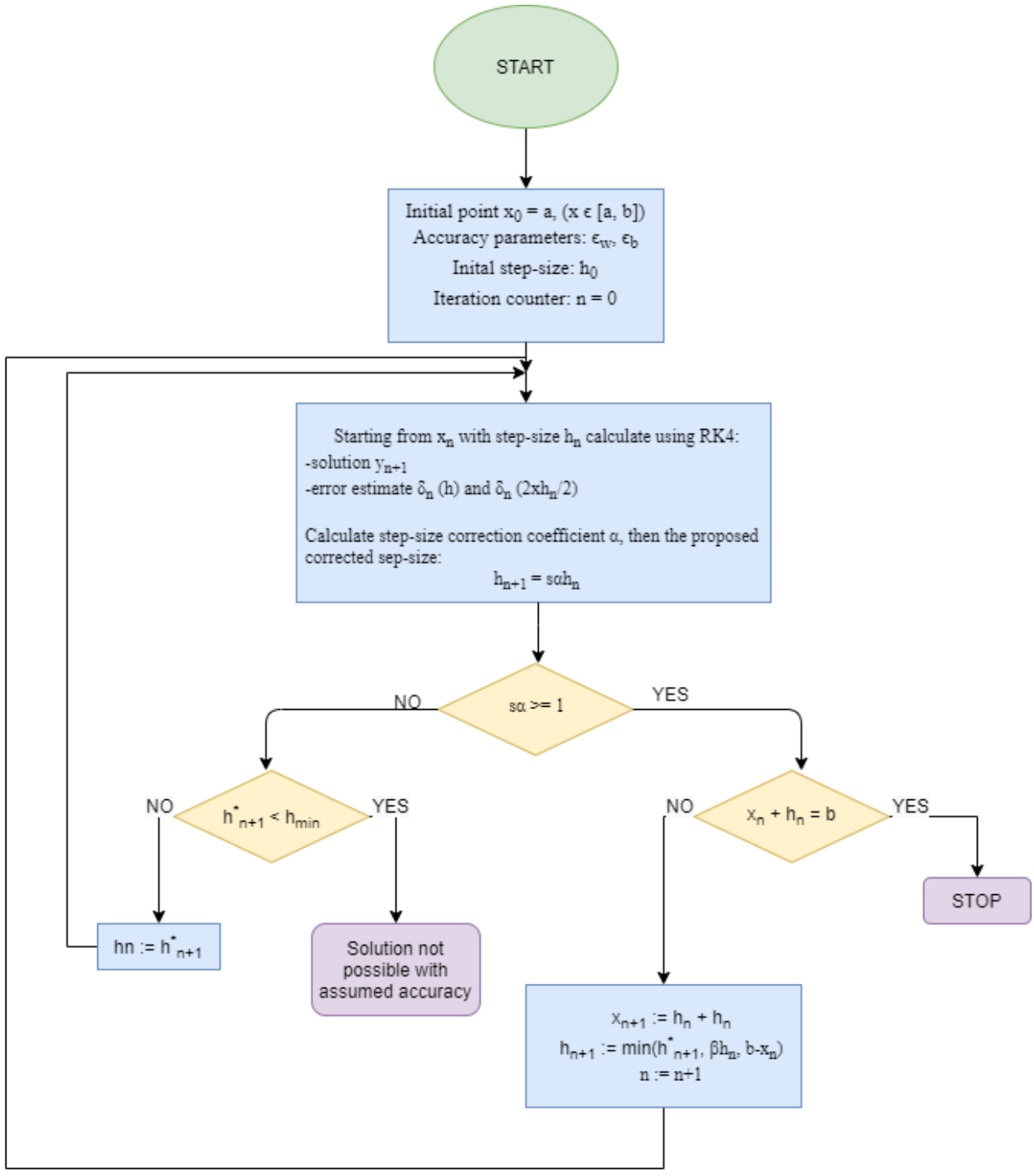
$$\epsilon = |y_n| \cdot \epsilon_r + \epsilon_a$$

where:

ϵ_r - a relative tolerance

ϵ_a - an absolute tolerance

Flow diagram of the algorithm



Results

I have not managed to implement this task in Matlab code. I performed analysis about algorithm with automatically adjusted step-size and I understand the goal of this task. From the previously performed experiments I can draw a hypothesis that this algorithm is much more useful in performing numerical calculations. It should be more efficient and require less arithmetic operations and cause less roundoff errors.

References

"Numerical Methods" – Piotr Tatjewski

Appendix – source code

```
close all
clear
clc

%plotSolution
%solvetask1

function [x, y] = getData

x = -5:1:5;
y = [-32.4591, -20.2011, -12.1986, -4.6508, -1.1893, 0.6266, 0.5743, -
0.3709, -1.2371, -3.4952, -4.3987];

end

function [G, v] = gramMatrix(x, y, n)

G = zeros(n+1, n+1);
v = zeros(n+1, 1);
N = length(x);
for i = 1:n+1
    for k = 1:n+1
        for j = 1:N
            G(i, k) = G(i, k) + (x(1, j))^(i+k-2);
        end
    end
end

for k = 1:n+1
    for j = 1:N
        v(k,1) = v(k,1) + (y(j)*(x(j)^(k-1)));
    end
end
end

function [Q, R] = QRfactorization(A);

[m, n] = size(A);
Q = zeros(m, n);
R = zeros(n, n);
d = zeros(1, n);
```

```
% orthogonal columns of Q
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

% orthonormal columns of Q
for i=1:n
    dd = norm(Q(:, i));
    Q(:, i) = Q(:, i) / dd;
    R(i, i:n) = R(i, i:n) * dd;
end

end

function [a, error, condNum] = solveNormalEq(G, v)

[Q, R] = QRfactorization(G);
d = Q' * v;

n = length(v);
a = zeros(n, 1);

%back substitution
for i = n:-1:1
    a(i) = d(i)/R(i,i);
    d(1:i-1) = d(1:i-1) - R(1:i-1,i)*a(i);
end

error = norm(G*a - v);
condNum = cond(G);

end

function [error, condprint, orderprint] = solvetask1
[x, y] = getData;

for i=2:1:10

    [G, v] = gramMatrix(x, y, i);
    [~, errorprint, cond] = solveNormalEq(G, v);
    error(11-i) = errorprint;
    condprint(i-1) = cond;
end

for i = 2:1:10
   orderprint(i-1) = i;
   errorprint = error(i-1);
   semilogy(i, errorprint, 'r*')
   hold on
   disp(i);
   disp(condprint(i-1));
   disp(errorprint);
```

```
end

grid on
title("Error vs. order of the polynomial")
xlabel("order of the polynomial")
ylabel("error")
hold off

end

function plotSolution
n=7;
[x, y] = getData;

[G, v] = gramMatrix(x, y, n);
[a, ~, ~] = solveNormalEq(G, v);

figure(n)
plot(x, y, '*','color', [0.3010, 0.7450, 0.9330])
hold on

X = ['Approximation polynomial - degree ', num2str(n)];
title(X)
syms x
if n == 7
    f = @(x)a(8)*x^7+a(7)*x^6+a(6)*x^5+a(5)*x^4 + a(4)*x^3 + a(3)*x^2 +
a(2)*x + a(1);
elseif n == 6
    f = @(x)a(7)*x^6+a(6)*x^5+a(5)*x^4 + a(4)*x^3 + a(3)*x^2 + a(2)*x +
a(1);
elseif n == 5
    f = @(x)a(6)*x^5+a(5)*x^4 + a(4)*x^3 + a(3)*x^2 + a(2)*x + a(1);
elseif n == 4
    f = @(x)a(5)*x^4 + a(4)*x^3 + a(3)*x^2 + a(2)*x + a(1);
elseif n == 3
    f = @(x)a(4)*x^3 + a(3)*x^2 + a(2)*x + a(1);
elseif n == 2
    f = @(x)a(3)*x^2 + a(2)*x + a(1);
end

fplot(f,'color',[0, 0, 1]);
grid on
xticks(-5:1:5);
hold off

end

%[x1, x2, t] = RK4(0.09);
%printRK4vsTime;
%[x1, x2, t] = P5EC5E(0.001);
%printP5EC5EvsTime

function [start, finish, initialx1, initialx2, f1, f2] = getData

start = 0;
finish = 20;
initialx1 = -0.001;
initialx2 = -0.02;

f1 = @(t, x1, x2)(x2 +x1*(0.5 - (x1)^2 - (x2)^2));
f2 = @(t, x1, x2)(-x1 + x2*(0.5 - (x1)^2 - (x2)^2));
```

```
end

function [x1, x2, t] = RK4(h)

[start, finish, initialx1, initialx2, f1, f2] = getData;

%to use when running P5EC5E
finish = (5*h)-start;

t = start:h:finish;
x1(1) = initialx1;
x2(1) = initialx2;

for i = 1:(length(t)-1)
    k11 = f1(t(i), x1(i), x2(i));
    k12 = f2(t(i), x1(i), x2(i));
    k21 = f1(t(i)+0.5*h, x1(i)+0.5*h*k11, x2(i)+0.5*h*k12);
    k22 = f2(t(i)+0.5*h, x1(i)+0.5*h*k11, x2(i)+0.5*h*k12);
    k31 = f1(t(i)+0.5*h, x1(i)+0.5*h*k21, x2(i)+0.5*h*k22);
    k32 = f2(t(i)+0.5*h, x1(i)+0.5*h*k21, x2(i)+0.5*h*k22);
    k41 = f1(t(i)+h, x1(i)+h*k31, x2(i)+h*k32);
    k42 = f2(t(i)+h, x1(i)+h*k31, x2(i)+h*k32);

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

% hold on
% plot(x1, x2,'color',[1, 0, 0])
%
% odeoptions = odeset('RelTol', 10e-10, 'AbsTol', 10e-10);
% [t, x] = ode45(@(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)], [start, finish], [initialx1,initialx2], odeoptions);
% plot(x(:,1),x(:,2),'color',[0, 0, 1])
% xlabel("x1")
% ylabel("x2")
% legend("RK4", "ODE45")
% X = ['RK4 vs ODE45 for step-size h = ', num2str(h)];
% title(X)
% grid on
% hold off

end

function printRK4vsTime

[start, finish, initialx1, initialx2, ~, ~] = getData

[x1s, x2s, ts] = RK4(0.09);
[x1b, x2b, tb] = RK4(0.3);

hold on
plot(ts, x1s)
plot(tb, x1b)
% plot(ts, x2s)
% plot(tb, x2b)

odeoptions = odeset('RelTol', 10e-10, 'AbsTol', 10e-10);
[t, x] = ode45(@(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)], [start, finish], [initialx1,initialx2], odeoptions);
```

```
plot(t,x(:,1))
%plot(t,x(:,2))
legend("h=0.09", "h=0.3", "ode45")
title("Solution vs time for optimal and larger step-sizes")
xlabel("t")
ylabel("x1")
%ylabel("x2")
hold off

end



function [x1, x2, t] = P5EC5E(h)

[start, finish, initialx1, initialx2, f1, f2] = getData;
k = 5;

betaEx = [1901/720, -2774/720, 2616/720, -1274/720, 251/720];
betaIm = [475/1440, 1427/1440, -798/1440, 482/1440, -173/1440, 27/1440];
[x1,x2,t] = RK4(h);

for n = 6:ceil(finish-start)/h
%for n = 6:1

    t(1+n) = t(n)+h;

    sumEx1 = 0;
    sumEx2 = 0;

    for j = 1:k
       sumEx1 = sumEx1 + betaEx(j) * f1(t(n-j), x1(n-j), x2(n-j));
       sumEx2 = sumEx2 + betaEx(j) * f2(t(n-j), x1(n-j), x2(n-j));
    end

    sumIm1 = 0;
    sumIm2 = 0;

    for j = 2:k
       sumIm1 = sumIm1 + betaIm(j) * f1(t(n-j), x1(n-j), x2(n-j));
       sumIm2 = sumIm2 + betaIm(j) * f2(t(n-j), x1(n-j), x2(n-j));
    end

    %P prediction
    x1(n+1) = x1(n) + h*sumEx1;
    x2(n+1) = x2(n) + h*sumEx2;

    %E evaluation
    fn1 = f1(t(n+1), x1(n+1), x2(n+1));
    fn2 = f2(t(n+1), x1(n+1), x2(n+1));

    %C correction
    x1(n+1) = x1(n) + h*sumIm1 + h*betaIm(1)*fn1;
    x2(n+1) = x2(n) + h*sumIm2 + h*betaIm(1)*fn2;

    %E evaluation
    fn1 = f1(t(n+1), x1(n+1), x2(n+1));
    fn2 = f2(t(n+1), x1(n+1), x2(n+1));
end
```

```
% plot(x1, x2,'color',[1, 0, 0])
% hold on;
%
% odeoptions = odeset('RelTol', 10e-10, 'AbsTol', 10e-10);
% [t, x] = ode45(@(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)], [start, finish], [initialx1,initialx2], odeoptions);
% plot(x(:,1), x(:,2),'color',[0, 0, 1]);
%
% X = ['P5EC5E vs ODE45 for step-size h = ', num2str(h)];
% title(X)
% xlabel("x1");
% ylabel("x2");
% legend("P5EC5E", "ode45");
% grid on;
% hold off;

end

function printP5EC5EvsTime

[start, finish, initialx1, initialx2, ~, ~] = getData

[x1s, x2s, ts] = P5EC5E(0.0001);
[x1b, x2b, tb] = P5EC5E(0.01);

hold on
% plot(ts, x1s)
% plot(tb, x1b)
plot(ts, x2s)
plot(tb, x2b)

odeoptions = odeset('RelTol', 10e-10, 'AbsTol', 10e-10);
[t, x] = ode45(@(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)], [start, finish], [initialx1,initialx2], odeoptions);
% plot(t,x(:,1))
plot(t,x(:,2))

legend("h=0.0001", "h=0.01", "ode45")
title("Solution vs time for optimal and larger step-sizes")
xlabel("t")
% ylabel("x1")
ylabel("x2")
hold off

end
```