Author: Sławomir Batruch
Numerical Methods project 3

Numerical Methods Project 3
Linear square approximation & ODE algorithms

Task 1

The aim of the first task is as follows:

I For the following experimental measurements (samples):

x_i	y_i
–5	–4.9606
–4	–3.3804
–3	–1.4699
–2	–1.1666
–1	0.4236
0	0.1029
1	–0.5303
2	–4.0483
3	–11.0280
4	–21.1417
5	–33.9458

determine a polynomial function $y=f(x)$ that best fits the experimental data by using the least-squares approximation (test polynomials of various degrees). Present the graphs of obtained functions along with the experimental data. To solve the least-squares problem use the system of normal equations with QR factorization of a matrix **A**. 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. Compare the results in terms of solutions' errors.

We will need to use the least-square approximation for polynomials of various degrees using a system of normal equations with QR factorisation of matrix A. Furthermore, the norm of the vector of residuum and Gram's condition number is going to be calculated for every obtained polynomial.

Theoretical background

Every information, text snippets and equations in this report are taken from *Numerical Methods - Piotr Tatjewski* book. If the source is different, it will be stated.

We know 11 finite points xi and 11 values corresponding to every of the points.

First

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

$$\forall F \in X_n \quad F(x) = \sum_{i=0}^{n} a_i \phi_i(x). \tag{4.5}$$

our function can be described the following way. The basis will be explained later.

Then:

The approximation problem:
to find values of the parameters $a_0, a_1, ..., a_n$ defining the approximating function (4.5), which minimize the least-squares error defined by

$$H(a_0, ..., a_n) \stackrel{\text{df}}{=} \sum_{j=0}^{N} \left[f(x_j) - \sum_{i=0}^{n} a_i \phi_i(x_j) \right]^2 . \qquad (4.6)$$

This is how the approximation problem is defined. Since we need to minimize it, we seek for the 1st order derivative of this error to be equal to 0, as shown below:

$$\frac{\partial H}{\partial a_k} = -2 \sum_{j=0}^{N} \left[f(x_j) - \sum_{i=0}^{n} a_i \phi_i(x_j) \right] \cdot \phi_k(x_j) = 0, \qquad k = 0, ..., n,$$

Which results in:

$$a_0 \sum_{j=0}^{N} \phi_0(x_j) \cdot \phi_0(x_j) + a_1 \sum_{j=0}^{N} \phi_1(x_j) \cdot \phi_0(x_j) + \cdots + a_n \sum_{j=0}^{N} \phi_n(x_j) \cdot \phi_0(x_j) = \sum_{j=0}^{N} f(x_j) \cdot \phi_0(x_j),$$

$$a_0 \sum_{j=0}^{N} \phi_0(x_j) \cdot \phi_1(x_j) + a_1 \sum_{j=0}^{N} \phi_1(x_j) \cdot \phi_1(x_j) + \cdots + a_n \sum_{j=0}^{N} \phi_n(x_j) \cdot \phi_1(x_j) = \sum_{j=0}^{N} f(x_j) \cdot \phi_1(x_j),$$

$$\vdots$$

$$a_0 \sum_{j=0}^{N} \phi_0(x_j) \cdot \phi_n(x_j) + a_1 \sum_{j=0}^{N} \phi_1(x_j) \cdot \phi_n(x_j) + \cdots + a_n \sum_{j=0}^{N} \phi_n(x_j) \cdot \phi_n(x_j) = \sum_{j=0}^{N} f(x_j) \cdot \phi_n(x_j).$$

This is a set of normal equations. These create Gram's matrix.

Theoretical background - cont.

The formulae below will be a basis for the task realization.

With the definition of a scalar product, these equations can be written as:

$$\langle \phi_i, \phi_k \rangle \stackrel{\text{df}}{=} \sum_{j=0}^{N} \phi_i(x_j)\, \phi_k(x_j)\,.$$

From this, we obtain an easier representation of the equations:

$$\begin{bmatrix} \langle \phi_0, \phi_0 \rangle & \langle \phi_1, \phi_0 \rangle & \cdots & \langle \phi_n, \phi_0 \rangle \\ \langle \phi_0, \phi_1 \rangle & \langle \phi_1, \phi_1 \rangle & \cdots & \langle \phi_n, \phi_1 \rangle \\ \vdots & \vdots & \vdots & \vdots \\ \langle \phi_0, \phi_n \rangle & \langle \phi_1, \phi_n \rangle & \cdots & \langle \phi_n, \phi_n \rangle \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \langle \phi_0, f \rangle \\ \langle \phi_1, f \rangle \\ \vdots \\ \langle \phi_n, f \rangle \end{bmatrix}.$$

The leftmost matrix is defined then as matrix A:

$$\mathbf{A} = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_n(x_N) \end{bmatrix},$$

We also define the following, in accordance with the ‘easier representation’:

$$\begin{aligned} \mathbf{a} &= [a_0\, a_1 \cdots a_n]^{\mathrm{T}}, \\ \mathbf{y} &= [y_0\, y_1 \cdots y_N]^{\mathrm{T}}, \quad y_j = f(x_j), \quad j = 0, 1, ..., N. \end{aligned}$$

Then, using the defined matrices A, a and y, we can rewrite the set of normal equations as:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\,\mathbf{a} = \mathbf{A}^{\mathrm{T}}\mathbf{y}.$$

But, per the *Numerical Methods* book:

> Because the matrix $\mathbf{A}$ has full rank, then the Gram’s matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is nonsingular. This implies uniqueness of the solution of the set of normal equations. But, even being nonsingular, the matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ can be badly conditioned – its condition number is a square of the condition number of $\mathbf{A}$.

Theoretical background - QR factorisation

The last statement from the previous paragraph shows that transpose(A)A can be badly conditioned, as the condition number is a square of the condition number of A. To avoid this situation, QR factorisation should be used. The following book snippet explains the application of the QR factorisation. In this case, we consider x as our 'a' matrix/vector, and b as our 'y' matrix/vector:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b}, \tag{3.33}$$

which is known as the set of *normal equations*. In the considered case, it has a unique solution. However, for a badly conditioned matrix $\mathbf{A}$, the condition number for the matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ becomes even much worse, because

$$\mathrm{cond}_2(\mathbf{A}^{\mathrm{T}}\mathbf{A}) = (\mathrm{cond}_2\mathbf{A})^2 = (\frac{\sigma_1}{\sigma_{\mathrm{n}}})^2,$$

where σ_1 and σ_n are the largest and the smallest singular values of $\mathbf{A}$. In such cases, it is recommended to find the solution using the QR factorization of the matrix $\mathbf{A}$, and it is convenient to use the thin (economical) QR factorization (3.8), $\mathbf{A}_{m\times n} = \mathbf{Q}_{m\times n}\mathbf{R}_{n\times n}$. The set of normal equations (3.33) can then be written in the form

$$\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{Q}\mathbf{R}\mathbf{x} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{b}. \tag{3.34}$$

We indeed see the risk of condition number taking relatively big values, which would mean that A is badly conditioned.
Since it is recommended to use thin QR factorisation, we will reuse the matlab code for the thin QR factorisation which was also used in a previous project.
We see that our obtained:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\,\mathbf{a} = \mathbf{A}^{\mathrm{T}}\mathbf{y}.$$

With applied QR factorisation is rewritten as:

$$\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{Q}\mathbf{R}\mathbf{x} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{b}.$$

Which can be reduced:

Due to the orthonormality of columns of the matrix $\mathbf{Q}$, we have $\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I}$. Taking also into account that the matrix $\mathbf{R}$ is nonsingular (because $k = n$), we obtain finally the following well defined system of linear equations

$$\mathbf{R}\mathbf{x} = \mathbf{Q}^{\mathrm{T}}\mathbf{b}. \tag{3.35}$$

Replacing 'x' with 'a' and 'b' with 'y' we solve for 'a' and obtain our result. Rewriting all the transformation with proper labeling we get:

$$\boldsymbol{A^T A a = A^T y}$$
$$\boldsymbol{R^T Q^T Q R a = R^T Q^T y}$$
$$\boldsymbol{R a = Q^T y}$$

Source: Own work

Theoretical background cont. 2 - Basis & resulting A matrix

The basis for our task is as follows:

$$\phi_0(x) = 1, \quad \phi_1(x) = x, \; \phi_2(x) = x^2, \quad \cdots, \quad \phi_n(x) = x^n, \qquad (4.13)$$

It is a common *natural polynomial basis*, also known as *the power basis.* The required basis is given in the task description.

So, with prior knowledge that:

$$\mathbf{A} = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_n(x_N) \end{bmatrix},$$

We can use our basis in this equation, which results in the following matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^n \end{bmatrix}$$

Source: Own work

This matrix will be used in the MATLAB implementation.

MATLAB implementation

Below I show snippets of the MATLAB algorithm implementation.
Comments about the code are shown in the code itself, hence no additional information might be written here. The comments are deemed exhaustive.

Class variables:

```
classdef task_1
    properties
        x; % Experimental x values
        y; % Experimental y values
        a; % Result matrix a (a coefficients)
    end
    % function [Q, R] = external_QR_factor(D) will be used, but is not
    % inside the class
```

Class constructor:

```
% Class initialiser
function obj = task_1()
    % Assign neccessary x and y values, per task description
    obj.x = -5:5; % x = [-5, -4, -3, [...] , 4, 5];
    obj.y = [-4.9606; -3.3804; -1.4699; -1.1666; 0.4236; 0.1029; ...
            -0.5303; -4.0483; -11.0280; -21.1417; -33.9458];
    % Transpose x (due to how -5:5 works)
    obj.x = obj.x';
end
```

Plot single line:

```
% This method plots a single line using stored A matrix
% It can be chosen whether to mark our 'experimental samples'
function plot_single(obj, markers)
    if (isempty(obj.a) == true) % Least_Square_Approx_QR should be run prior to running of this function
        error("'a' matrix is empty! Unable to plot nonexistent data!");
    end
    a_poly = poly_a(obj); % Obtain a modified A matrix that will be parsed in poly MATLAB functions
    % Our computed a matrix is not fit for the syntax of polyval
    x_axis = -5:.1:5; % Define the x axis values. -5 to 5 with intervals of 0.1
    figure; % Creates an empty figure.
    %This is done so that there can be many windows with different plots
    plot(x_axis, polyval(a_poly, x_axis)); % Plots our defined x_axis vs values of the computed polynomial
    title("Degree " + (size(obj.a, 1) - 1) ); % Puts a title on the plot to show what degree the plot shows
    grid on;
    if (markers == true) % Passed as argument, whether points should be marked
        hold on;
        plot(obj.x, obj.y, 'LineStyle', 'none', 'Marker', 'o', 'Color', 'red');
        hold off;
    end
end
```

Modification of ‘a’ matrix for polyval usage:

```
% This method returns a new A matrix, one that can be used in poly MATLAB functions
function a_poly = poly_a(obj) % polyval takes the 'a' coefficients in a reverse way
    a_poly = obj.a;
    a_poly = flip(a_poly); % Reverse the order
    a_poly = a_poly'; % Transpose
end
```

Least square approximation with QR, error and Gram's matrix norm calculation

```
% This function runs the Least Square Approximation Algorithm with
% the use of QR factorisation. Result is stored in obj.a
function [obj, cond_gram, error_norm] = Least_Square_Approx_QR(obj, deg)
    % Note: Before the last line of this algorithm, obj.a is not
    % our 'a' matrix, but 'A' matrix instead.
    % I chose to reuse obj.a as 'A' before storing the actual
    % result to optimise memory use
    obj.a = zeros(size(obj.x, 1), deg+1); % Define empty A
    % Of the size being (length of x) x (degree + 1)
    % Degree 0 is valid, hence +1

    % Refer to the computed a matrix at page 5 of the report:
    for i = 1:size(obj.x, 1) % Amount of rows is the amount of data points
        for j = 1:deg+1 % Amount of columns is the degree+1 (because e.g. degree 0 has 1 col)
            obj.a(i, j) = obj.x(i)^(j-1); % Since arrays in MATLAB start at 1
            % But we need x^0 power to be considered as well, we use j-1 as exponent
        end
    end
    % Condition number of our Gram's matrix needs to be calculated now as Gram's matrix is discarded
    cond_gram = cond(obj.a' * obj.a); %

    [Q, R] = external_QR_factor(obj.a); % Perform QR factorisation with the use of thin algorithm
    obj.a = R\(Q'*obj.y); % Ra = Q'y (page 5 in report), solve, store result in a.

    % Error calculation:
    a_poly = poly_a(obj); % Fetch modified a so that polyval can process our 'a' matrix
    computed_values = polyval(a_poly, obj.x); % Evaluate our obtained y values.
    % It evaluates polynomial values with coefficients A_poly at
    % the points from obj.x
    error_norm = norm(computed_values - obj.y); % Residuum, norm(Ax - b)
end
```

Note: residuum is actually norm(Ax - y). Comment has a mistake (last line)

Display plot, error and Gram's condition number (pretty formatted):

```
% Function below calls all neccessary methods that the task
% description mentions: Least Square Approximation, error_norm
% calculation and then plots the computed graph with marked points
function obj = all_in_one(obj, deg, display_a)
    [obj, cond_gram, error_norm] = Least_Square_Approx_QR(obj, deg); % Execute least Square approx algorithm
    % Also save the condition number of the Gram's matrix
    % Display all the information:
    fprintf("Degree ; Condition ; Error_norm (csv format)\n%d ; %.5f ; %.10f\n", deg, cond_gram, error_norm);
    if (display_a == true) % We choose whether to display 'a' matrix or not
        disp("a transposed: [a0, a1, ...]^T (csv format)");
        for i = 1:size(obj.a, 1)
            fprintf("%.5f ; ", obj.a(i)'); % Display all matrix a values in a csv format
        end
        fprintf("\n\n");
    end
    plot_single(obj, true); % Display the plot
end
```

Compute Least Square Approx; display plot, error and Gram's condition number for a range of degrees:

```
% This function allows to execute multiple least square algorithms
% for different degress, as well as print neccessary data after
% every algorithm execution for a degree
function obj = execute_multiple(obj, min_deg, max_deg, display_a)
    for i = min_deg:max_deg % We execute in the range of the provided min and max deg
        obj = all_in_one(obj, i, display_a); % Refer to the function comments
        pause(0.5); % To avoid fast window pop ups
    end
end
```

MATLAB code usage and output

After trial and error, I have chosen the max degree to be 10, as this is the first degree of the polynomial for which the norm is 0.

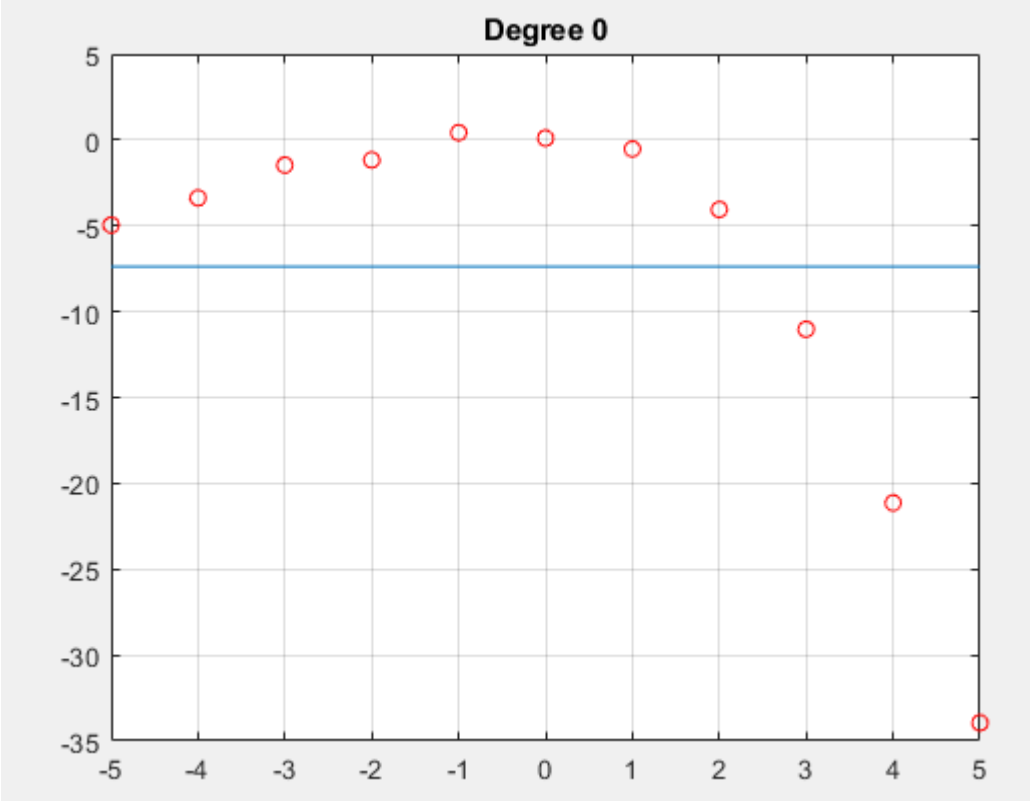
I have run the execute_multiple method, creating a task_1 object beforehand:

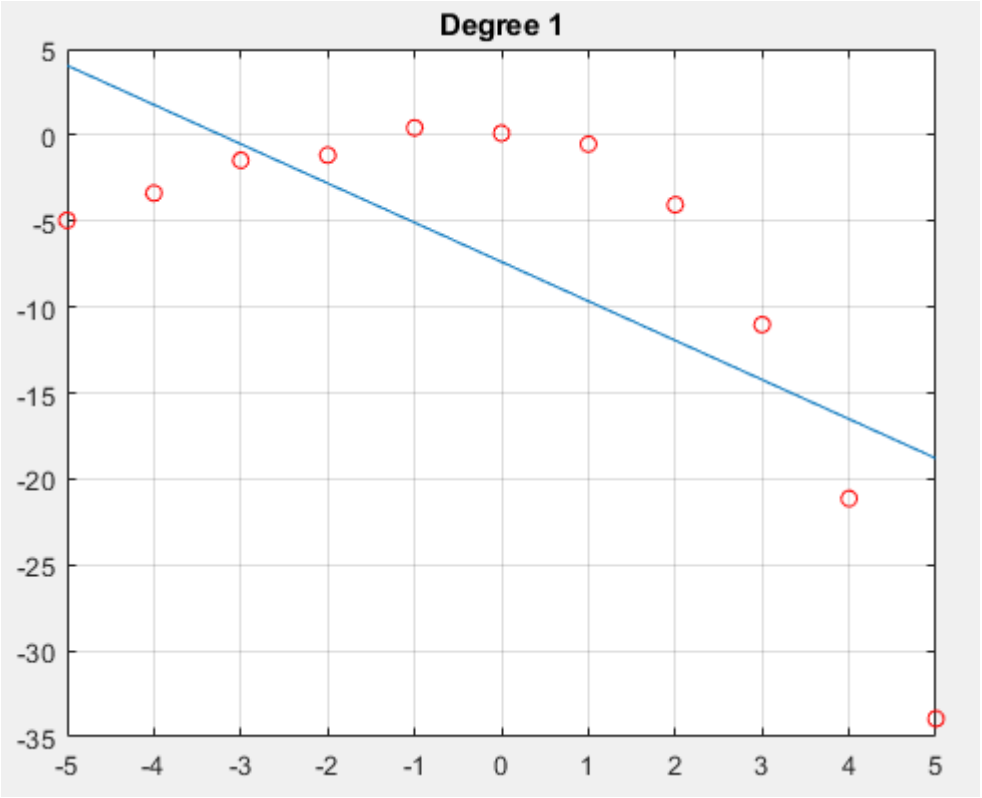
```
task_object = task_1()
task_object = execute_multiple(task_object, 0, 10, true)
```

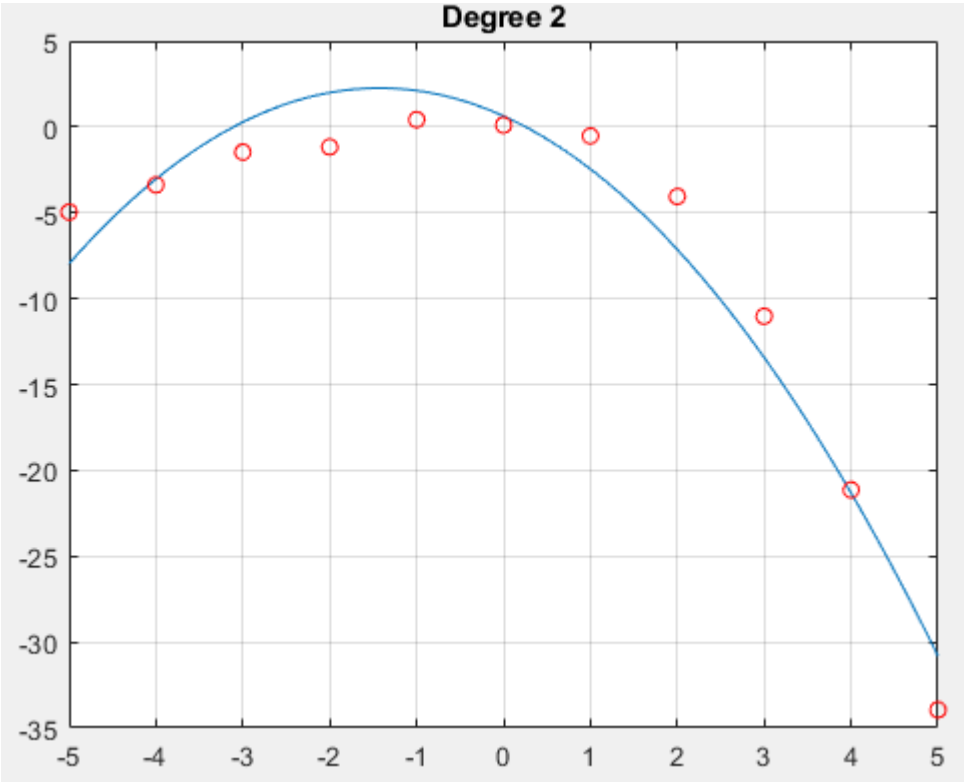
This yielded the following output as seen on the next page:

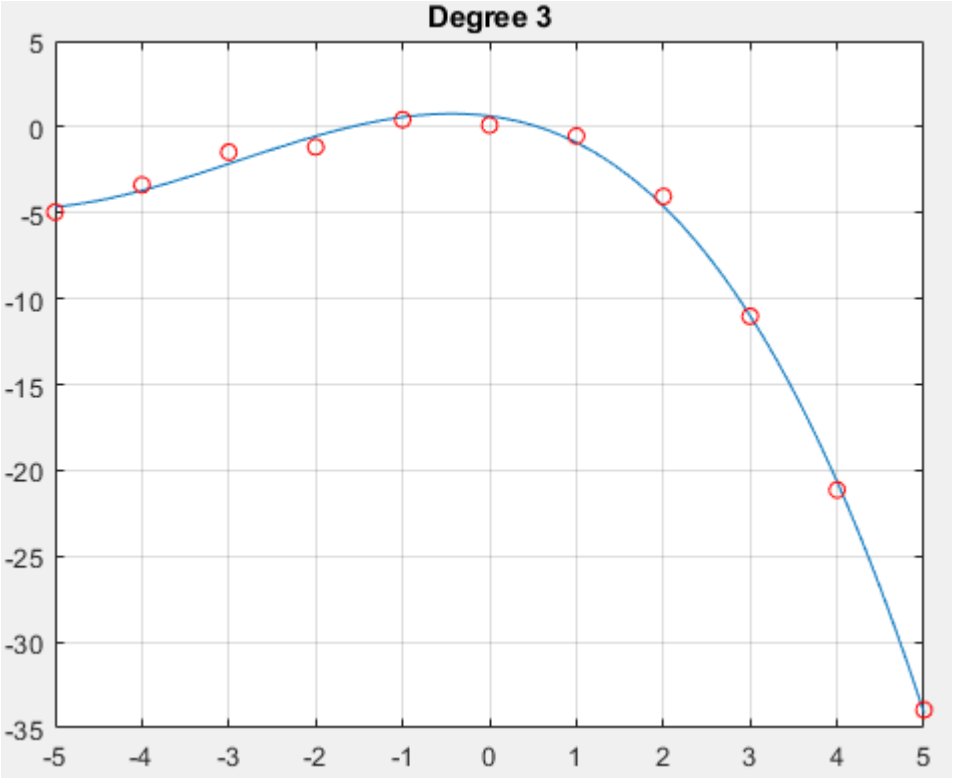
```
>> task_object = execute_multiple(task_object, 0, 10, true)
Degree ; Condition ; Error_norm (csv format)
0 ; 1.00000 ; 34.3326143520
a transposed: [a0, a1, ...]^T (csv format)
-7.37683 ;

Degree ; Condition ; Error_norm (csv format)
1 ; 10.00000 ; 24.5832292580
a transposed: [a0, a1, ...]^T (csv format)
-7.37683 ; -2.28512 ;

Degree ; Condition ; Error_norm (csv format)
2 ; 408.77960 ; 7.3646946352
a transposed: [a0, a1, ...]^T (csv format)
0.63028 ; -2.28512 ; -0.80071 ;

Degree ; Condition ; Error_norm (csv format)
3 ; 8558.43658 ; 1.4389643960
a transposed: [a0, a1, ...]^T (csv format)
0.63028 ; -0.64938 ; -0.80071 ; -0.09190 ;

Degree ; Condition ; Error_norm (csv format)
4 ; 317981.44720 ; 1.3958411179
a transposed: [a0, a1, ...]^T (csv format)
0.50624 ; -0.64938 ; -0.75764 ; -0.09190 ; -0.00172 ;

Degree ; Condition ; Error_norm (csv format)
5 ; 7467495.65952 ; 0.8500953194
a transposed: [a0, a1, ...]^T (csv format)
0.50624 ; -0.22686 ; -0.75764 ; -0.16207 ; -0.00172 ; 0.00222 ;

Degree ; Condition ; Error_norm (csv format)
6 ; 283155896.03388 ; 0.7594776315
a transposed: [a0, a1, ...]^T (csv format)
0.36202 ; -0.22686 ; -0.64455 ; -0.16207 ; -0.01419 ; 0.00222 ; 0.00033 ;

Degree ; Condition ; Error_norm (csv format)
7 ; 7646220977.72670 ; 0.7069019089
a transposed: [a0, a1, ...]^T (csv format)
0.36202 ; -0.06232 ; -0.64455 ; -0.21987 ; -0.01419 ; 0.00708 ; 0.00033 ; -0.00011 ;

Degree ; Condition ; Error_norm (csv format)
8 ; 330546434323.18176 ; 0.6996772047
a transposed: [a0, a1, ...]^T (csv format)
0.40483 ; -0.06232 ; -0.70971 ; -0.21987 ; 0.00061 ; 0.00708 ; -0.00070 ; -0.00011 ; 0.00002 ;

Degree ; Condition ; Error_norm (csv format)
9 ; 151670890112l5.34961 ; 0.5149896456
a transposed: [a0, a1, ...]^T (csv format)
0.40483 ; -0.52453 ; -0.70971 ; 0.09120 ; 0.00061 ; -0.04674 ; -0.00070 ; 0.00318 ; 0.00002 ; -0.00006 ;

Degree ; Condition ; Error_norm (csv format)
10 ; 9292965365191l0.25000 ; 0.0000000000
a transposed: [a0, a1, ...]^T (csv format)
0.10290 ; -0.52453 ; 0.18797 ; 0.09120 ; -0.39862 ; -0.04674 ; 0.05759 ; 0.00318 ; -0.00325 ; -0.00006 ; 0.00006 ;
```

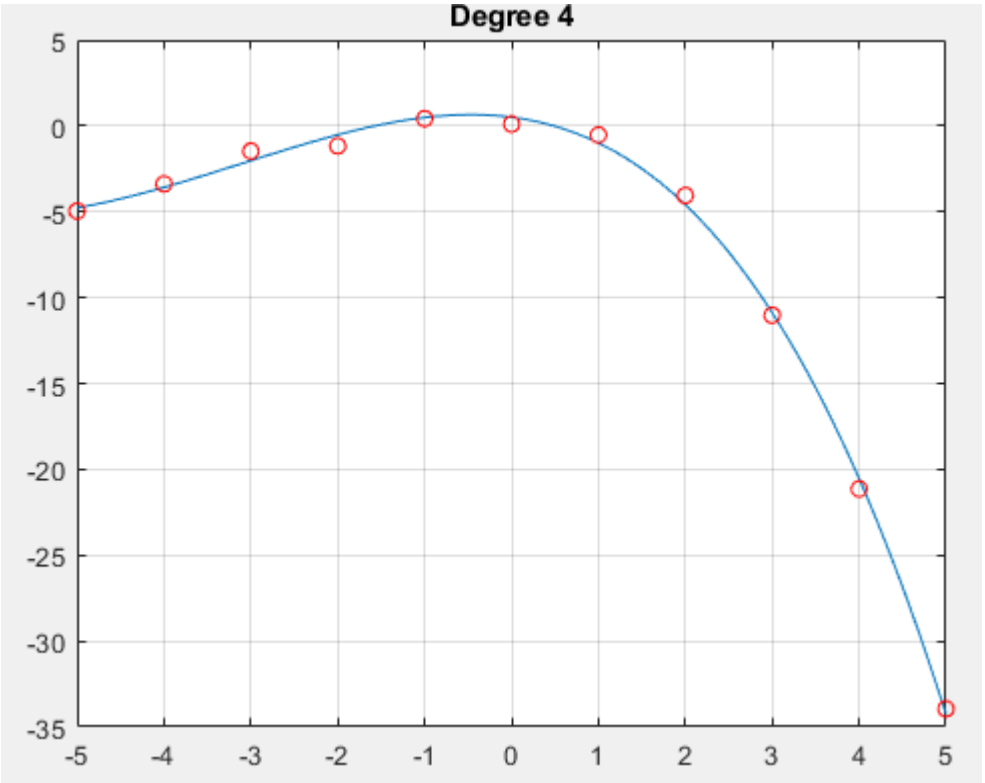
The following plots were obtained:

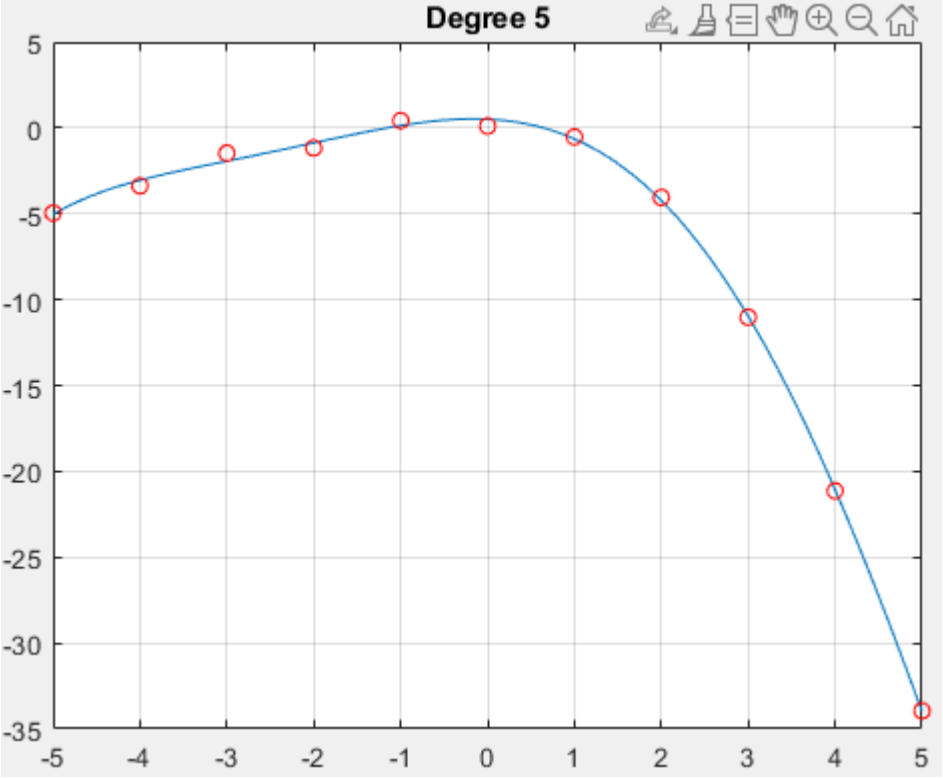


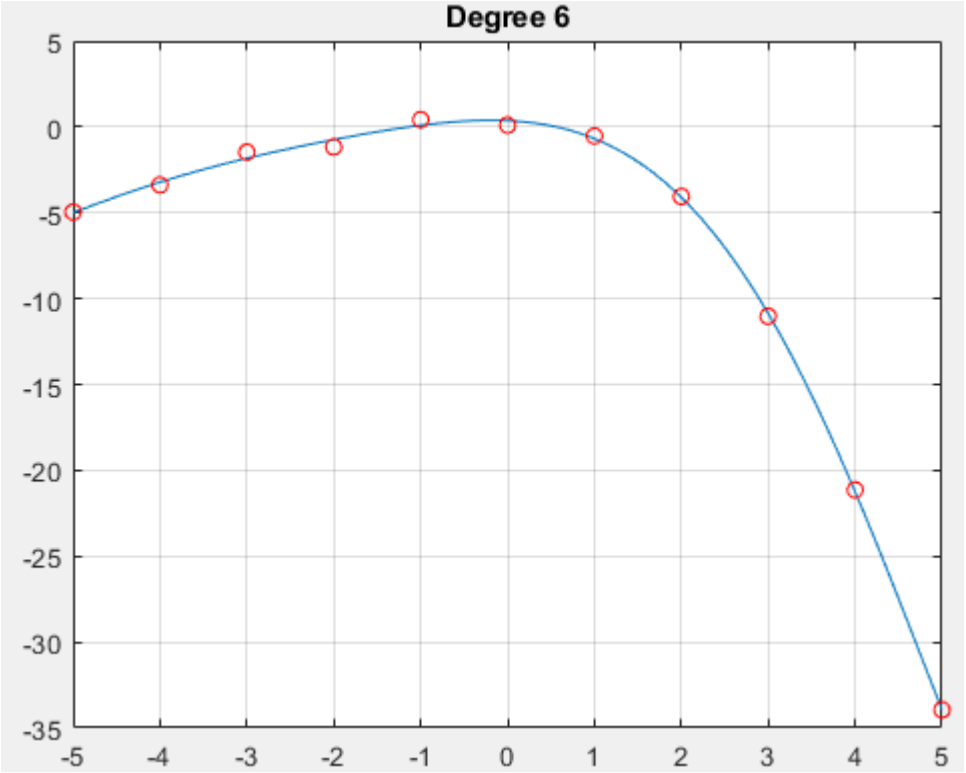


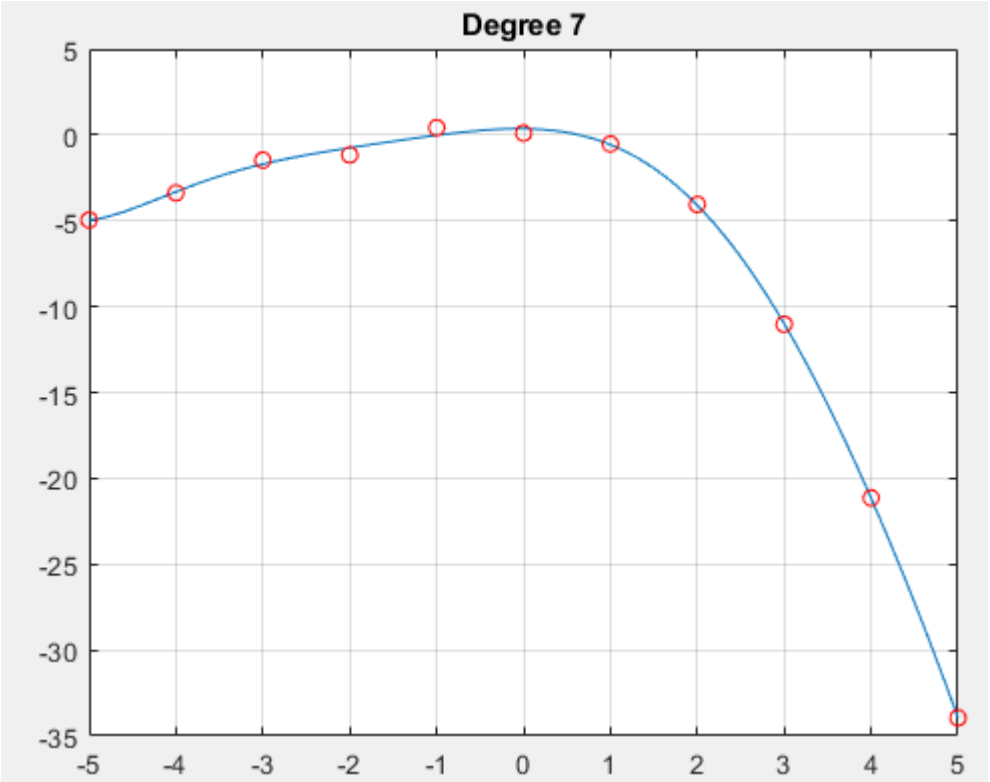

Degree 2
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

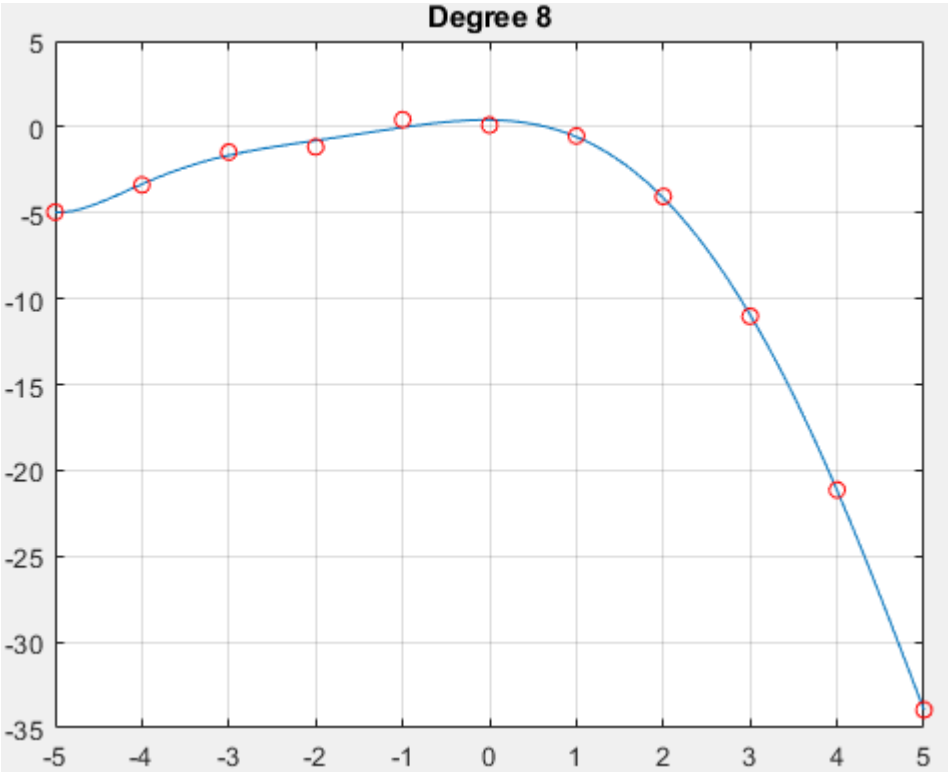

Degree 3
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

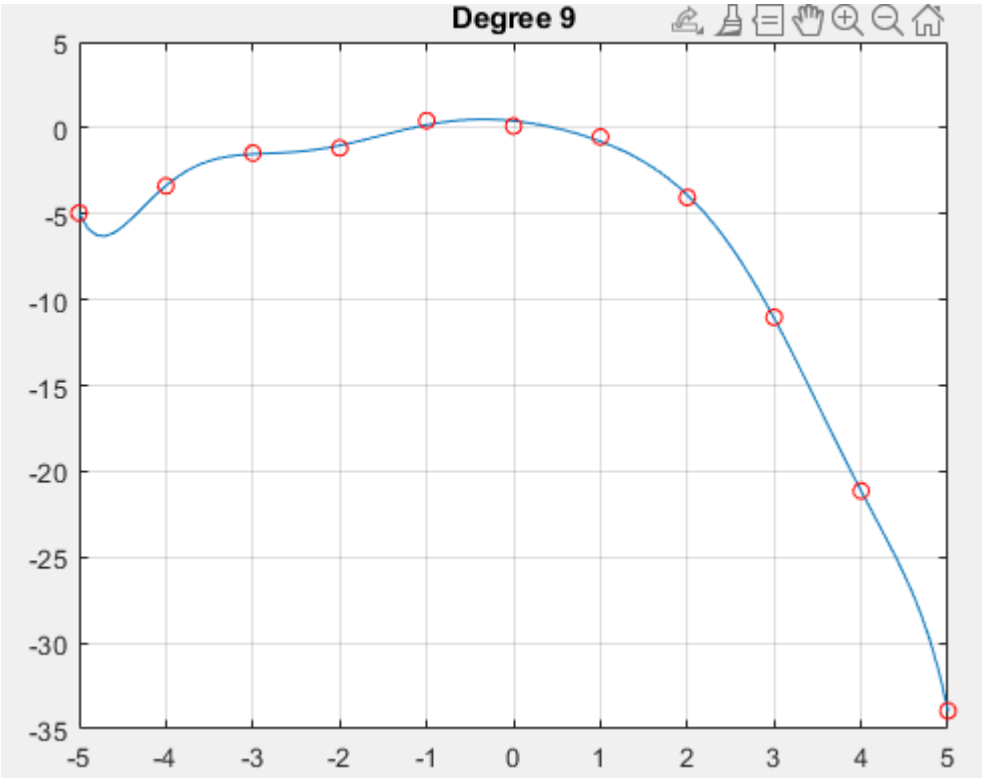

Degree 4
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

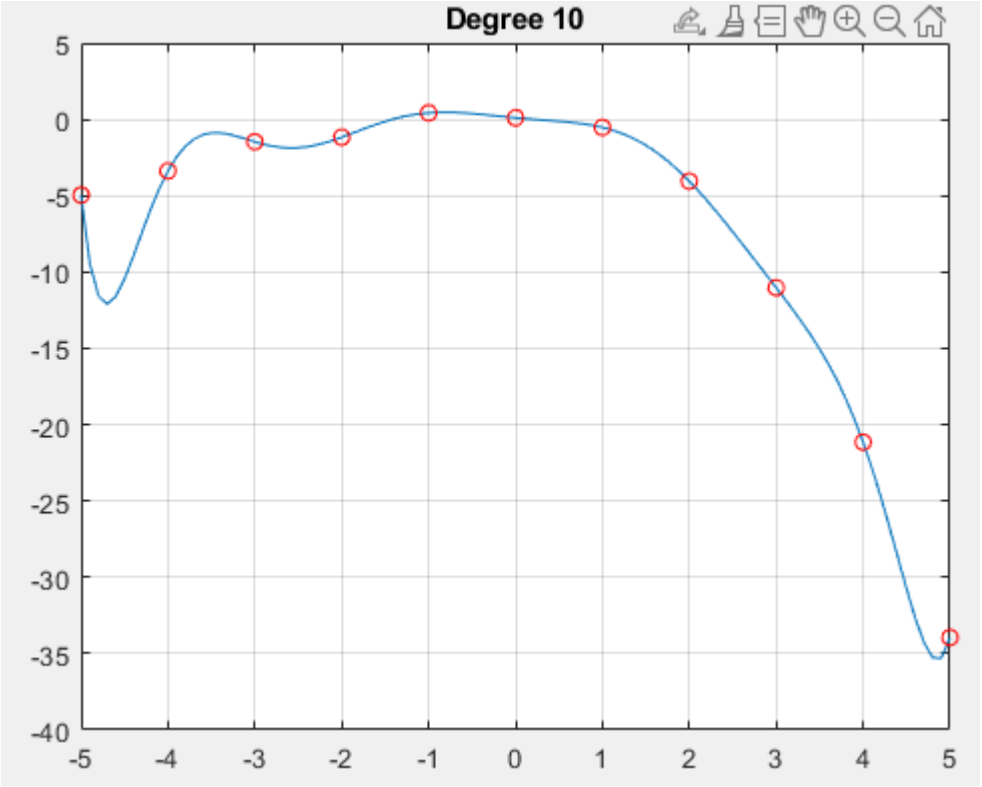

Degree 5
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Degree 6
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Degree 7
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Degree 8
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Degree 9
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


Degree 10
5
0
-5
-10
-15
-20
-25
-30
-35
-40
-5
-4
-3
-2
-1
0
1
2
3
4
5

Output data analysis

The obtained data was formatted then in Excel to better show the results. Note that ‘a’ vector values were rounded off to the 5th decimal place:

$$a_{10}x^{10} + a_9x^9 + a_8x^8 + a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Degree	Gram's condition num	Error	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0
0	1	34.33261	0	0	0	0	0	0	0	0	0	0	-7.37683
1	10	24.58323	0	0	0	0	0	0	0	0	0	-2.28512	-7.37683
2	408.7796	7.364695	0	0	0	0	0	0	0	0	-0.80071	-2.28512	0.63028
3	8558.43658	1.438964	0	0	0	0	0	0	0	-0.0919	-0.80071	-0.64938	0.63028
4	317981.4472	1.395841	0	0	0	0	0	0	-0.00172	-0.0919	-0.75764	-0.64938	0.50624
5	7467495.66	0.850095	0	0	0	0	0	0.00222	-0.00172	-0.16207	-0.75764	-0.22686	0.50624
6	283155896	0.759478	0	0	0	0	0.00033	0.00222	-0.01419	-0.16207	-0.64455	-0.22686	0.36202
7	7646220978	0.706902	0	0	0	-0.00011	0.00033	0.00708	-0.01419	-0.21987	-0.64455	-0.06232	0.36202
8	3.30546E+11	0.699677	0	0	0.00002	-0.00011	-0.0007	0.00708	0.00061	-0.21987	-0.70971	-0.06232	0.40483
9	1.51671E+13	0.51499	0	-0.00006	0.00002	0.00318	-0.0007	-0.04674	0.00061	0.0912	-0.70971	-0.52453	0.40483
10	9.29297E+14	0	0.00006	-0.00006	-0.00325	0.00318	0.05759	-0.04674	-0.39862	0.0912	0.18797	-0.52453	0.1029

Furthermore, the following table was created to show the difference between Error and Gram’s condition num between degrees:

Degree	Gram's condition num delta	Error abs delta
0	1	34.33261435
1	9	9.749385094
2	398.7796	17.21853462
3	8149.65698	5.925730239
4	309423.0106	0.043123278
5	7149514.212	0.545745799
6	275688400.4	0.090617688
7	7363065082	0.052575723
8	3.229E+11	0.007224704
9	1.48365E+13	0.184687559
10	9.14129E+14	0.514989646

Both tables are own work

Conclusions & Closing remarks

Looking at the table above showing all the obtained data we can notice that the best resulting polynomial is of degree 3.

Explanation:

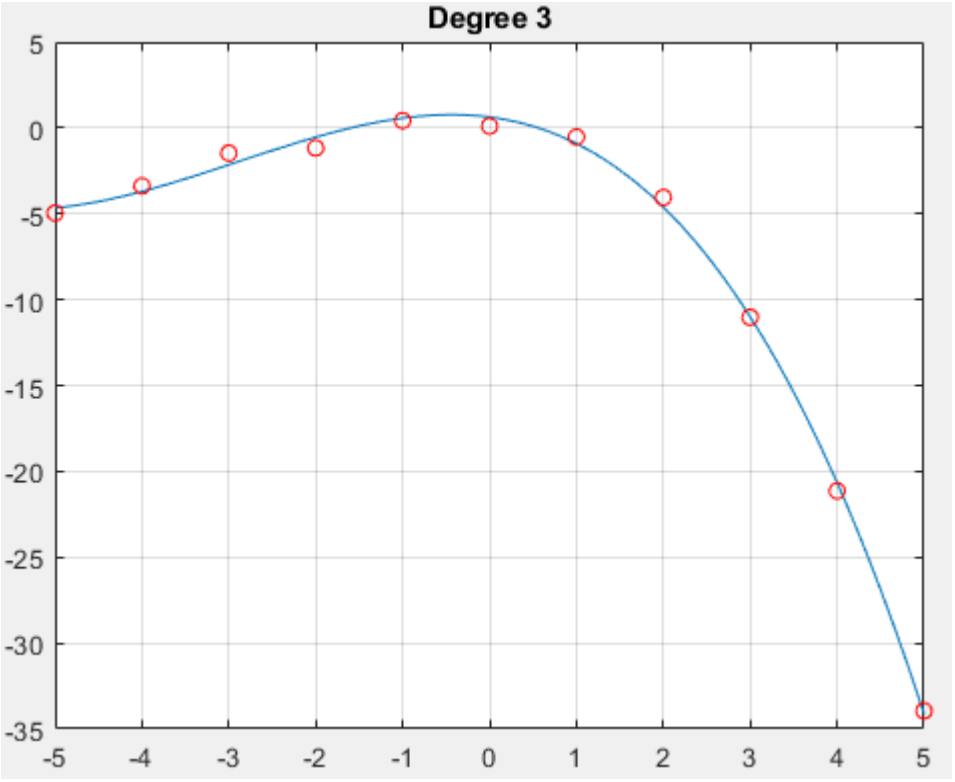
This is due to the fact that error is relatively small (~1.44), and the Gram's condition number is not "too big".

Above was concluded by looking at the previous and next error values. We can see on the 2nd table with delta values, that previous error delta was ~5.93 for deg = 3, whereas next deltas of the error are very small, which indicates small changes in regards to the error (also noticeable on the main table).

Furthermore, it is important to notice the Gram's condition number. The smaller the error, the bigger the number. Also - the higher the degree, the higher the difference in the Gram's condition number (seen on 2nd table).

Gram's condition number reaching a very high value indicates high probability of errors. That is why degree = 10 is not chosen. The error being 0 indicates that the function values at given points match our experimental data values, although looking at the degree = 10 polynomial, we see it is "malformed" in some places like between x = -5 and x = -4, or right before x = 5. It looks different than all other solutions from degree = 3 to degree = 8 range.

To remind, I show degree = 3 plot:



Task 2

The aim of this task is as follows:

II A motion of a point is given by equations:

$$dx_1/dt = x_2 + x_1 (0.5 - x_1^2 - x_2^2),$$
$$dx_2/dt = -x_1 + x_2 (0.5 - x_1^2 - x_2^2).$$

Determine the trajectory of the motion on the interval [0, 15] for the following initial conditions: $x_1(0) = 0$, $x_2(0) = -0.3$. Evaluate the solution using:

a) Runge-Kutta method of 4th order (RK4) and Adams PC (P_5EC_5E) – 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 4th order (RK4) with a variable step size automatically adjusted by the algorithm, making error estimation according to the step-doubling rule.

Compare the results with the ones obtained using an ODE solver, e.g. ode45.

Furthermore:

Ad IIa) A discussion of constant step size selection illustrated by: two solution curves x_2 versus x_1 on one plot: first for the optimal constant step size and second for the larger step-size (for which the solution visibly differs from the first one); plots of problem solution versus time, obtained for the same optimal and larger step-sizes,

Ad IIb) A discussion of the chosen value of h_{min} (minimal step size), absolute and relative tolerances, and the following plots:

1) x_2 versus x_1, 2) problem solution versus time, 3) step size versus time, 4) error estimate versus time.

A flow diagram of the algorithm in IIb) should be also attached.

Task 2a

For this subtask we need to consider Runge-Kutta method of 4th order (RK4) and Adams Predictor-Corrector method of the form P5EC5E

These methods will need to be run *a few* times for different constant step-sizes.
I will need to find an optimal step-size such that the decrease of it doesn't influence the solution in a significant way, but its increase does (similar as for task 1).

Runge-Kutta 4th order algorithm - theoretical background

In general, the family of Runge-Kutta methods is defined by the following formulas:

$$y_{n+1} = y_n + h \cdot \sum_{i=1}^{m} w_i k_i, \tag{7.19a}$$

where

$$k_1 = f(x_n, y_n), \tag{7.19b}$$

$$k_i = f(x_n + c_i h, y_n + h \cdot \sum_{j=1}^{i-1} a_{ij} k_j), \quad i = 2, 3, ..., m, \tag{7.19c}$$

and also

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

We are interested in the 4th order RK method, also known as “classical”, RK4:

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right), && \text{(7.20a)}\\
k_1 &= f\left(x_n, y_n\right), && \text{(7.20b)}\\
k_2 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1), && \text{(7.20c)}\\
k_3 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2), && \text{(7.20d)}\\
k_4 &= f\left(x_n + h, y_n + hk_3\right). && \text{(7.20e)}
\end{aligned}
$$

Where ‘h’ is a step that is chosen by us. We first calculate every k coefficient and then plug it in into the y formula to obtain our next iteration result.
Explanation regarding formulas is as follows:
“ [...] we have four approximate values of the solution derivative over the one step interval: one at the initial point, two at the midpoint and one 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”
Quote from Numerical Methods - Piotr Tatjewski

Our k’s are the mentioned solution derivatives over one step interval. k1 is at the initial, k2 and k3 at the midpoint, and k4 at the endpoint.
Then we can indeed see weight = 1 for k1 and k4, but weight = 2 for k2 and k3 by looking at the coefficients next to k’s in the yn+1 formula.

Adam’s predictor-corrector algorithm - theoretical background

Adam’s PC method is a connection between Adam’s implicit and Adam’s explicit method.
“The idea behind the predictor-corrector methods is to use a suitable combination of an explicit and an implicit technique to obtain a method with better convergence characteristics”
Source: https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node7.html

Below I show the general formula for the P_kEC_kE method.
Based on the following formulas from the book. Explicit method (predictor):

$$y_n = y_{n-1} + h\sum_{j=1}^{k}\beta_j f\left(x_{n-j}, y_{n-j}\right), \qquad (7.43)$$

Implicit method (corrector):

$$
\begin{aligned}
y_n &= y_{n-1} + h\sum_{j=0}^{k}\beta_j^* \cdot f\left(x_{n-j}, y_{n-j}\right)\\
&= y_{n-1} + h\cdot\beta_0^*\cdot f\left(x_n, y_n\right) + h\sum_{j=1}^{k}\beta_j^*\cdot f\left(x_{n-j}, y_{n-j}\right), \qquad (7.44)
\end{aligned}
$$

In our case k = 5.
To better show the formula I decided to rewrite it. I replaced n = n + 1. This indicates more clearly that we actually compute the next y value (y_n+1). Furthermore, the use of predicted value is now indicated in the corrector formula. Also - in our case the function is f(t,x) = x(t) rather than f(x,y) = y(x).
In that case, the general formulae are as follows:

$$Prediction: x_{n+1}^{p} = x_n + h \cdot \sum_{j=1}^{k} \beta_j \cdot f\left(t_{n-j+1}, x_{n-j+1}\right)$$

$$Correction: x_{n+1} = x_n + h \cdot \sum_{j=0}^{k} \beta_j^{*} \cdot f\left(t_{n-j+1}, x_{n-j+1}\right) =$$

$$x_{n+1} = x_n + h \cdot \beta_0^{*} \cdot f\left(t_{n+1}, x_{n+1}^{p}\right) + h \cdot \sum_{j=1}^{k} \beta_j^{*} \cdot f\left(t_{n-j+1}, x_{n-j+1}\right)$$

Adam's PC method constraint

Because Adam's PC method is a multistep method, we can notice that for our k = 5, we will only be able to compute the 6th value. We can see this by looking at the prediction formula that the sum from j = 1 up to k = 5 will require us to compute f(t_n, x_n) up to f(t_(n-4), x_(n-4)).
This indicates we need to start by computing the 6th t value, and we need to obtain previous 5 t values in some other way, in our case - RK4 method:
"[...] first k values of the solution, y0, ..., yk−1, are necessary – which cannot be evaluated using a k-step method. Therefore, a special starting procedure is needed to evaluate these values. Application of an appropriate (having the same accuracy) RK algorithm can be a solution."
Quote from Numerical Methods - Piotr Tatjewski

Adam's PC method β and β* tables - appendix

Values of β and β* used in the Adam's PC method are shown below in the tables:

Table 7.6. Parameters of the explicit Adams methods (Adams-Bashforth methods)

k	β_1	β_2	β_3	β_4	β_5	β_6	β_7
1	1						
2	$\frac{3}{2}$	$-\frac{1}{2}$					
3	$\frac{23}{12}$	$-\frac{16}{12}$	$\frac{5}{12}$				
4	$\frac{55}{24}$	$-\frac{59}{24}$	$\frac{37}{24}$	$-\frac{9}{24}$			
5	$\frac{1901}{720}$	$-\frac{2774}{720}$	$\frac{2616}{720}$	$-\frac{1274}{720}$	$\frac{251}{720}$		
6	$\frac{4277}{1440}$	$-\frac{7923}{1440}$	$\frac{9982}{1440}$	$-\frac{7298}{1440}$	$\frac{2877}{1440}$	$-\frac{475}{1440}$	
7	$\frac{198721}{60480}$	$-\frac{447288}{60480}$	$\frac{705549}{60480}$	$-\frac{688256}{60480}$	$\frac{407139}{60480}$	$-\frac{134472}{60480}$	$\frac{19087}{60480}$

Table 7.7. Parameters of the implicit Adams methods (Adams-Moulton methods)

k	β_0^*	β_1^*	β_2^*	β_3^*	β_4^*	β_5^*	β_6^*	β_7^*
1^+	1							
1	$\frac{1}{2}$	$\frac{1}{2}$						
2	$\frac{5}{12}$	$\frac{8}{12}$	$-\frac{1}{12}$					
3	$\frac{9}{24}$	$\frac{19}{24}$	$-\frac{5}{24}$	$\frac{1}{24}$				
4	$\frac{251}{720}$	$\frac{646}{720}$	$-\frac{264}{720}$	$\frac{106}{720}$	$-\frac{19}{720}$			
5	$\frac{475}{1440}$	$\frac{1427}{1440}$	$-\frac{798}{1440}$	$\frac{482}{1440}$	$-\frac{173}{1440}$	$\frac{27}{1440}$		
6	$\frac{19087}{60480}$	$\frac{65112}{60480}$	$-\frac{46461}{60480}$	$\frac{37504}{60480}$	$-\frac{20211}{60480}$	$\frac{6312}{60480}$	$-\frac{863}{60480}$	
7	$\frac{36799}{120960}$	$\frac{139849}{120960}$	$-\frac{121797}{120960}$	$\frac{123133}{120960}$	$-\frac{88547}{120960}$	$\frac{41499}{120960}$	$-\frac{11351}{120960}$	$\frac{1375}{120960}$

Numerical Methods - Piotr Tatjewski

Algorithm implementation in MATLAB - RK4 & Adam's PC k = 5

Below I show how I implemented the RK4 algorithm and Adam's PC algorithm in MATLAB. To understand how to solve a system of ODE equations, ode45 function documentation helped me:
https://www.mathworks.com/help/matlab/ref/ode45.html#bu00_4l-2
Especially note the "Input arguments - `odefun`" paragraph. A clear example is demonstrated regarding a system of ODEs.

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
end
```

Despite this being an example for ode45, a similar solution worked for the algorithms I have designed. I decided to declare the odefun as an anonymous function from the beginning rather than as a normal function, which is later passed as an anonymous function.

It should be also noted that the odefun does not actually have 't' as a variable inside, but the notation @(t,x) is done so that ode45 accepts this function handle as the argument when we use ode45 later on in the report.
In code, it theoretically does not matter what t is put if we execute obj.odefun(t,x) to evaluate our ODEs values, but for the sake of clarity and completeness, I decided to put t values as if they were necessary.

Below I show code snippets for the implementation:

Body of the class:

```
classdef task_2
    properties
        odefun; % Our set of ode equations per task def as anonymous function
        t_interval; % Interval per task def
        x_init; % Initial values per task def
    end
```

Method initialisation:

```
function obj = task_2()
    % Values are taken from the task .pdf
    % Anonymous function type:
    obj.odefun = @(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)]; % Explained in report
    obj.t_interval = [0, 15];
    obj.x_init = [0, -0.3]; % x1(0) = 0 ; x2(0) = -0.3 from task desc
end
```

RK4 algorithm:

```
function [t, x] = RK4(obj, h, special_mode)
    t = obj.t_interval(1):h:obj.t_interval(2); % Define our t axis values
    if (exist('special_mode', 'var') == 0) % if special_mode was not passed e.g. in manual fun call
        % Set it to false
        special_mode = false;
    end
    if (special_mode == true) % 5 first values for Adams only
        t = t(1):h:t(5); % Reduce out t array to account for above
    end
    x(:, 1) = obj.x_init; % Assign initial x values
    for n = 1 : (length(t) - 1)
        % All below are formulas from the report for k1, k2, k3, k4 and
        % next iteration y_n+1
        k1 = obj.odefun(t(n), x(:, n));
        k2 = obj.odefun(t(n) + h/2, x(:, n) + h/2 * k1);
        k3 = obj.odefun(t(n) + h/2, x(:, n) + h/2 * k2);
        k4 = obj.odefun(t(n+1), x(:, n) + h * k3); % note: t(n+1) = t + h

        x(:, n+1) = x(:, n) + h * (k1 + 2*k2 + 2*k3 + k4)/6;
    end
end
```

Adam's PC algorithm. Predictor:

```
function [t, x] = Adams_PECE_5(obj, h)
    t = obj.t_interval(1):h:obj.t_interval(2); % t axis values
    B_exp = [1901, -2774, 2616, -1274, 251] / 720; % Beta_explicit consts.
    B_imp = [475, 1427, -798, 482, -173, 27] / 1440; % Beta_implicit consts.
    % Note that beta_implicit starts at b*_0, but b_explicit at b_1
    % "Special starting procedure"
    % Explained in the theoretical part of the report
    % It will fetch first five initial values we will use for first step of
    % Adams PECE method
    [~, x] = RK4(obj, h, true); % last arg indicates a special mode for adams pc. Refer to rk4 comments
    for i = 5 : (length(t) - 1)
        sum_temp = 0;
        % Predictor formula. Provided in the report
        for j = 0 : 4
            % Indexes used can be explained by looking at the "no for" loop
            % formula provided few lines below
            sum_temp = sum_temp + B_exp(j + 1) * obj.odefun(t(i - j), x(:, i - j));
        end
        prediction = x(:, i) + h * sum_temp; % Plug in the sum and finalise
        % Without the use of for loop, the full formula would be:
        % prediction = x(:, i) + ...
                        %h * (B_exp(1) * obj.odefun(t(i), x(:, i))...
                        %+ B_exp(2) * obj.odefun(t(i-1), x(:, i-1))...
                        %+ B_exp(3) * obj.odefun(t(i-2), x(:, i-2))...
                        %+ B_exp(4) * obj.odefun(t(i-3), x(:, i-3))...
                        %+ B_exp(5) * obj.odefun(t(i-4), x(:, i-4)));
```

Adam's PC Corrector:

```
        % Corrector formula. Provided in the report. First calculation is
        % done outside for loop as it uses prediction variable from before.
        sum_temp = B_imp(1) * obj.odefun(t(i+1), prediction);
        for j = 0 : 4
            % Indexes used can be explained by looking at the "no for" loop
            % formula provided few lines below
            sum_temp = sum_temp + B_imp(j + 2) * obj.odefun(t(i - j), x(:, i - j));
        end
        x(:, i+1) = x(:, i) + h * sum_temp; % Plug in the sum and finalise

        % Without the use of for loop, the full formula would be:
        %t(:, i+1) = t(:, i) + ...
                    %h * B_imp(1) * obj.odefun(t(i+1), prediction) + ...
                    %h * (B_imp(2) * obj.odefun(t(i), x(:, i))...
                        %+ B_imp(3) * obj.odefun(t(i-1), x(:, i-1))...
                        %+ B_imp(4) * obj.odefun(t(i-2), x(:, i-2))...
                        %+ B_imp(5) * obj.odefun(t(i-3), x(:, i-3))...
                        %+ B_imp(6) * obj.odefun(t(i-4), x(:, i-4)));
    end
end
```

Plotting a single graph:

```
function plot_single_alg(obj, h, alg_type, extra_plot)
    % Check the alg_type argument that indicates which alg to use
    if (alg_type == 1)
        [t, x] = RK4(obj, h, false); % alg_type 1 means RK4
        alg_name = "RK4";
    elseif (alg_type == 2)
        [t, x] = Adams_PECE_5(obj, h); % alg_type 2 means PC method
        alg_name = "Adams PC";
    else
        error("Invalid alg_type! Choose 1 for RK4, 2 for Adams PECE.");
    end

    plot(t,x); % Plot x(t)
    legend("x1(t)", "x2(t)"); % Label the lines
    title(alg_name + " x1(t) x2(t) for step = " + h); % Displays algorithm type, graph type and step chosen
    % Same as above, but we plot x1(x2)
    if (extra_plot == true) % This is done so that when we compare functions for different steps
        % In some cases, we might not want to show x1(x2) plot
        figure;
        plot(x(1, :), x(2, :));
        legend("x1(x2)");
        title(alg_name + " x1(x2) for step = " + h); % Displays algorithm type, graph type and step chosen
    end
end
```

Plotting graphs showing results for different steps:

```
function compare_plots(obj, h, iter_count)
    RK4_plot = [figure; figure]; % RK4_plot(1) will be for x1, (2) for 2
    Adams_plot = [figure; figure]; % Same logic here
    % Set plot titles
    figure(RK4_plot(1)); title("RK4 x1(t)");
    figure(RK4_plot(2)); title("RK4 x2(t)");
    figure(Adams_plot(1)); title("Adams x1(t)");
    figure(Adams_plot(2)); title("Adams x2(t)");
    legend_text = strings(1,iter_count); % Used to label every line
    % Show which line corresponds to which step
    for i = 1 : iter_count
        % RK4 plots part:
        [t, x] = RK4(obj, h, false); % Fetch results from RK4 alg
        figure(RK4_plot(1)); % Choose current figure as the 1st RK4
        hold on;
        plot(t, x(1, :)); % Add x1(t) for current h to the plot
        hold off;
        figure(RK4_plot(2));
        hold on;
        plot(t, x(2, :)); % Then add x2(t) for current h to the 2nd plot
        hold off;
        % Adams PC plots part:
        [t, x] = RK4(obj, h, false); % Fetch results from RK4 alg
        figure(Adams_plot(1)); % Choose current figure as the 1st RK4
        hold on;
        plot(t, x(1, :)); % Add x1(t) for current h to the plot
        hold off;
        figure(Adams_plot(2));
        hold on;
        plot(t, x(2, :)); % Then add x2(t) for current h to the 2nd plot
        hold off;

        legend_text(i) = "Step = " + h; % Array of strings that will be used to display the legend
        h = h/2; % Decrease the precision by half
    end
    % Add the legends to the plots to show which line corresponds
    % to which step
    figure(RK4_plot(1)); legend(legend_text);
    figure(RK4_plot(2)); legend(legend_text);
    figure(Adams_plot(1)); legend(legend_text);
    figure(Adams_plot(2)); legend(legend_text);
end
```

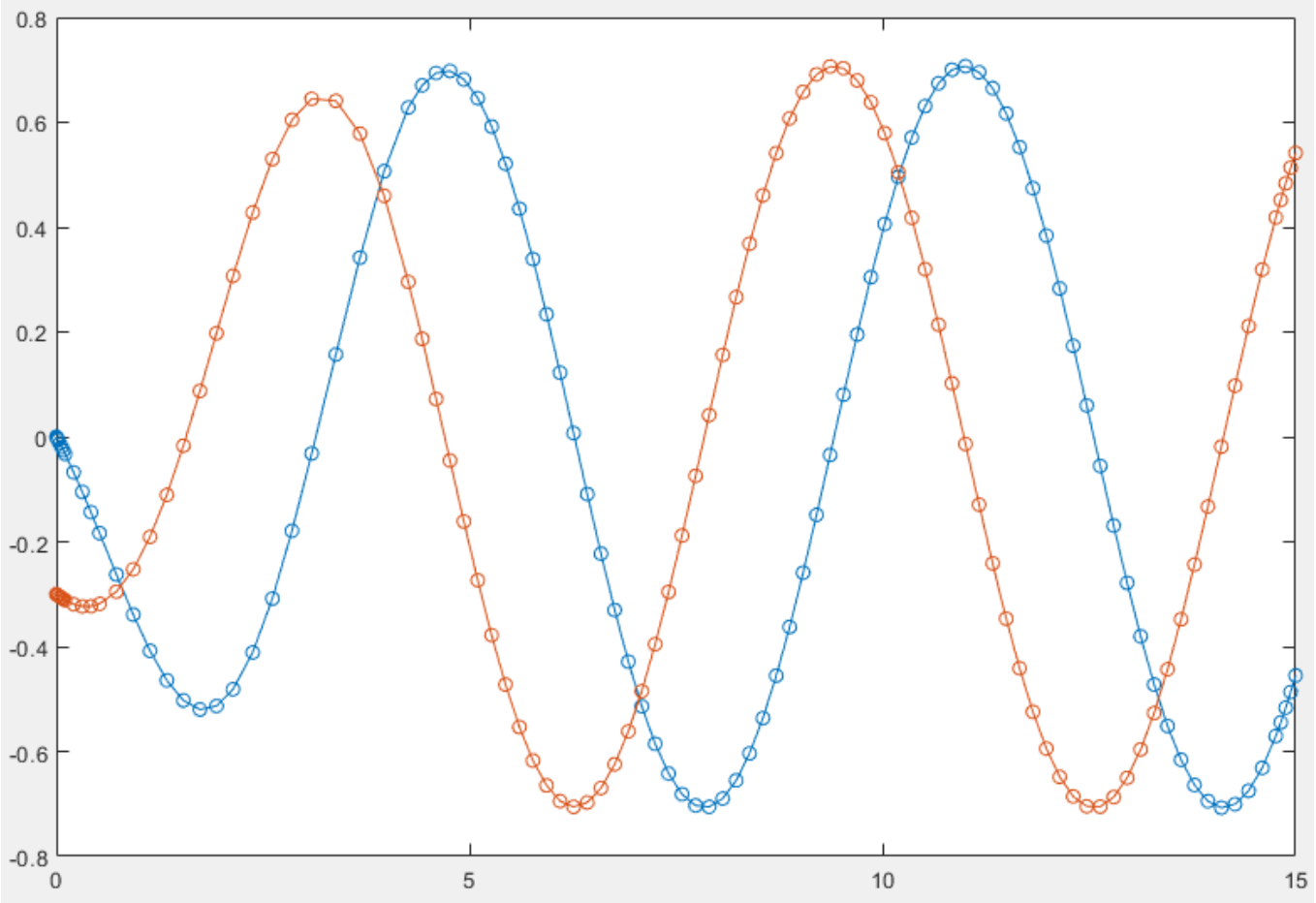
Plotting two x1(x2) lines for two different step for comparison:

```
% After manually finding the best step, this function is called
function x1x2_plots(obj, initial_h, final_h)
    %x1(x2) plot part:
    x1x2_plot = [figure; figure]; % 1st figure is for RK4, 2nd for Adams
    % RK4
    figure(x1x2_plot(1)); % Choose the 1st figure (for RK4)
    hold on;
    % Write the title to show initial and final h, as well as
    % algorithm type:
    title("x1(x2) RK4 initial-h " + initial_h + " final-h " + final_h);
    [~, x] = RK4(obj, initial_h, false);
    plot(x(1, :), x(2, :)); % Plot RK4 for initial_h
    [~, x] = RK4(obj, final_h, false);
    plot(x(1, :), x(2, :));% Plot RK4 for final_h
    legend("initial-h", "final-h"); % Add the legend
    hold off;
    % Adams PC
    figure(x1x2_plot(2)); % Choose the 2nd figure (for Adams)
    hold on;
    % Write the title to show initial and final h, as well as
    % algorithm type:
    title("x1(x2) Adams PC initial-h " + initial_h + " final-h " + final_h);
    [~, x] = Adams_PECE_5(obj, initial_h);
    plot(x(1, :), x(2, :)); % Plot Adams for initial_h
    [~, x] = Adams_PECE_5(obj, h);
    plot(x(1, :), x(2, :)); % Plot Adams for final_h
    legend("initial-h", "final-h"); % Add the legend
    hold off;
end
```

ode45 MATLAB method solution discussion

```
>> ode45(my_ode.odefun, my_ode.t_interval, my_ode.x_init)
>> legend("x1(t)", "x2(t)")
```

yields the following x1(t) and x2(t) graph:

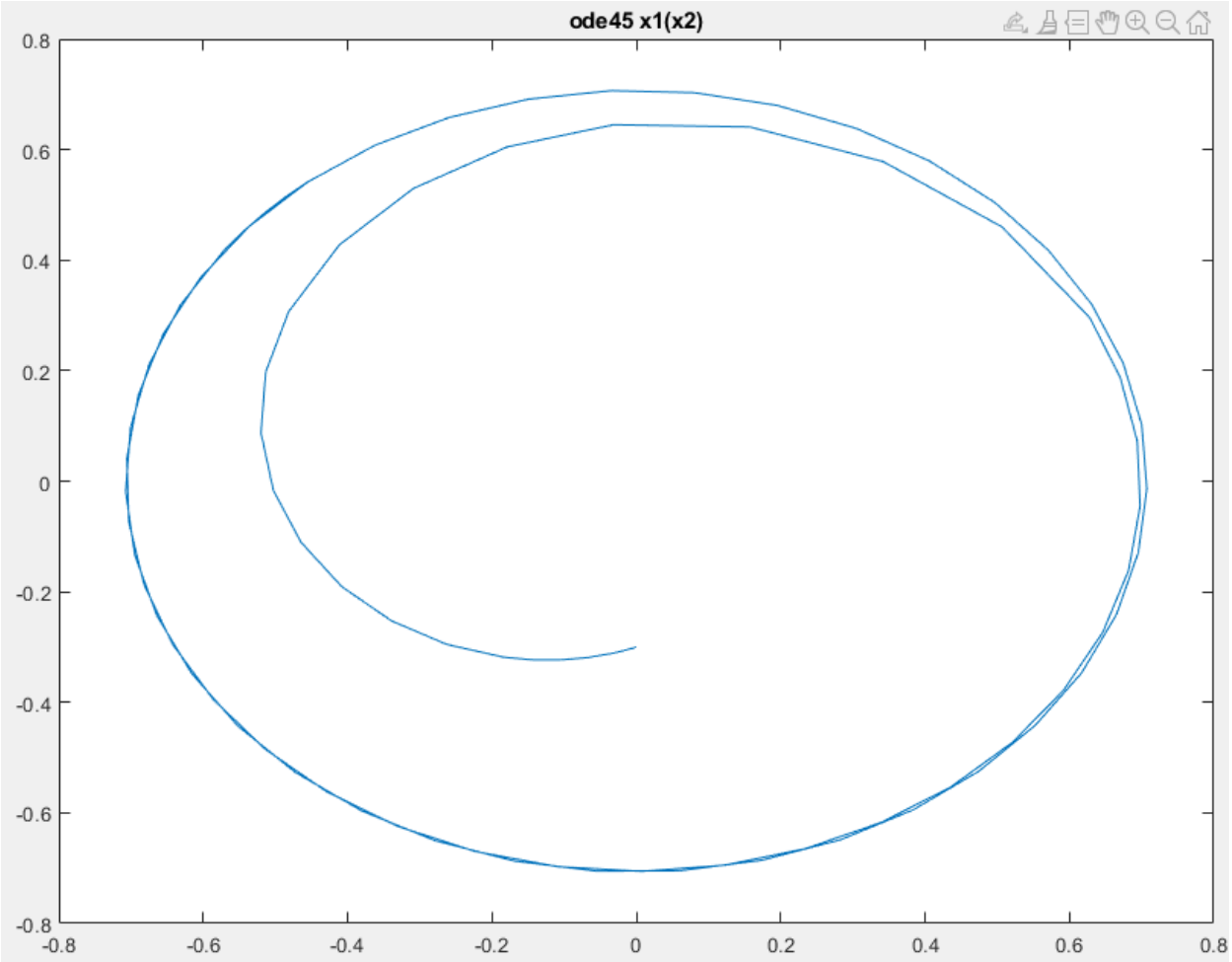


And then running ode45 again, but this time we store result arrays:

```
>> [t,x] = ode45(my_ode.odefun, my_ode.t_interval, my_ode.x_init)
```

Then transposing x, simply by x = x' …
We plot the following x1(x2) graph in the same way:

```
>> plot(x(1, :), x(2, :))
>> title("ode45 x1(x2)")
```



Comparison for different steps

Initialize an object of the class

```
>> my_ode = task_2()

my_ode =

  task_2 with properties:

        odefun: @(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)]
    t_interval: [0 10]
        x_init: [0 -0.3000]
```

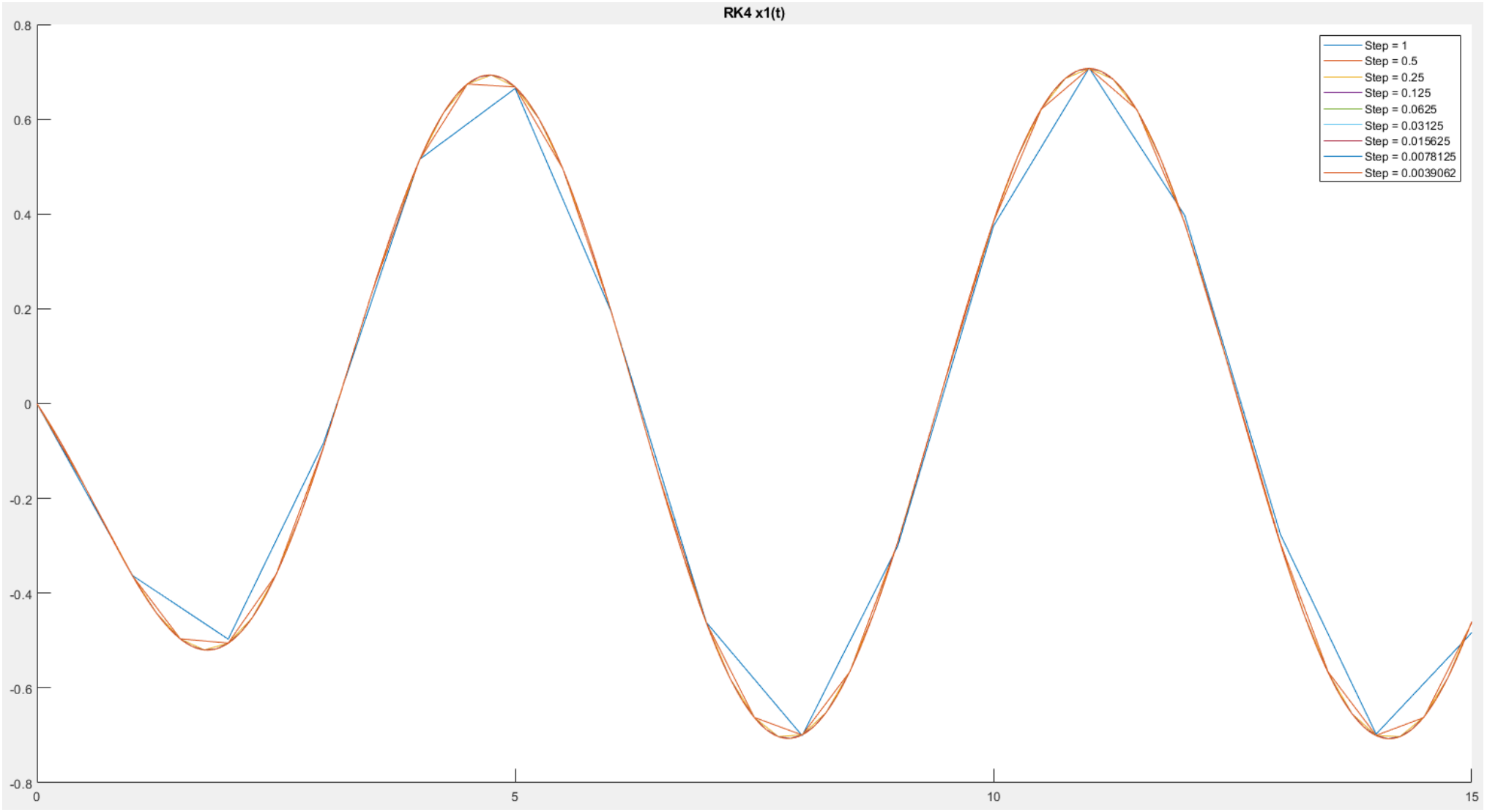
Plot the graphs from different steps and compare them:

Using the following command:

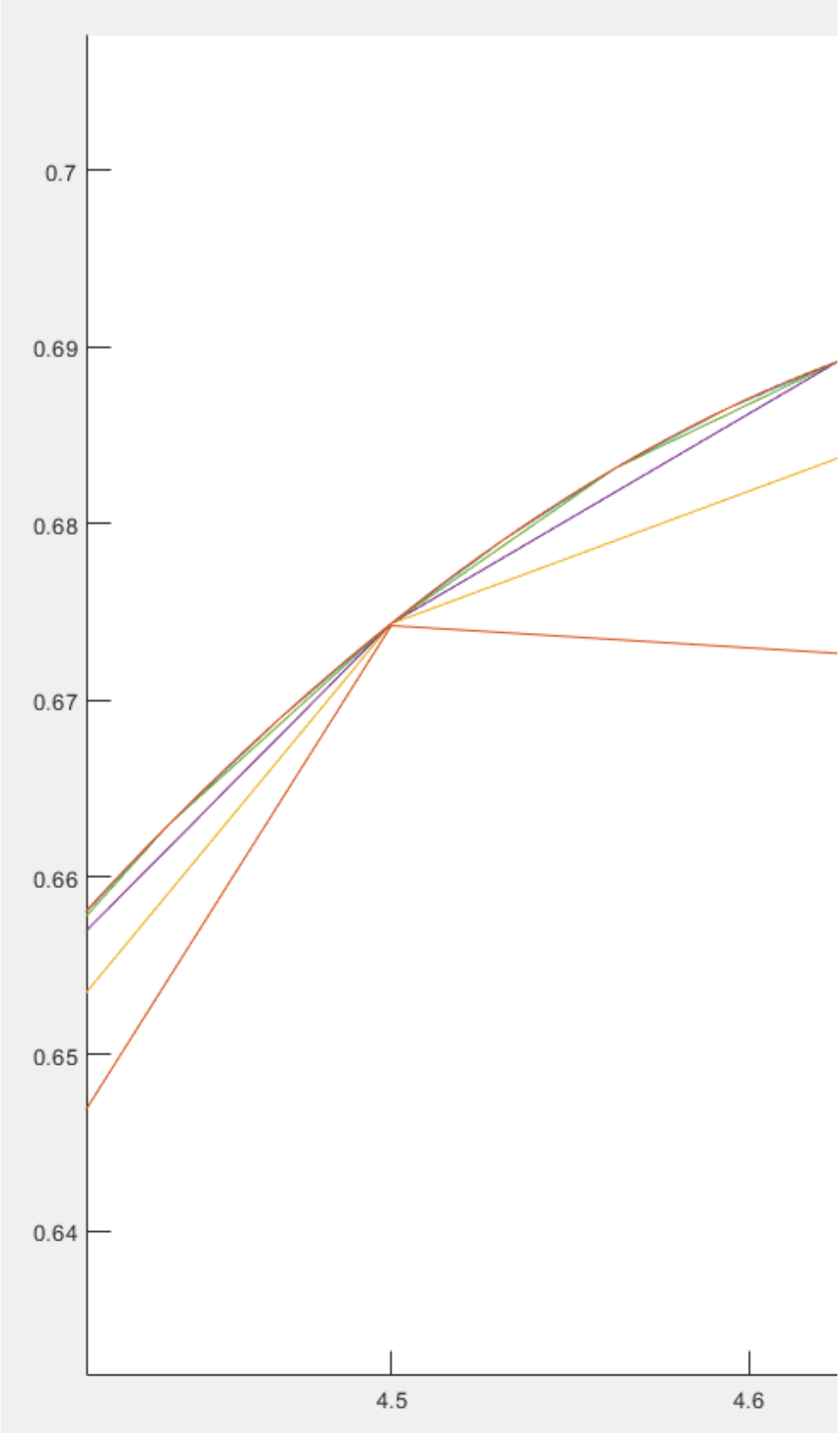
```
>> compare_plots(my_ode, 0.4, 10)
>> compare_plots(my_ode, 1, 9)
```

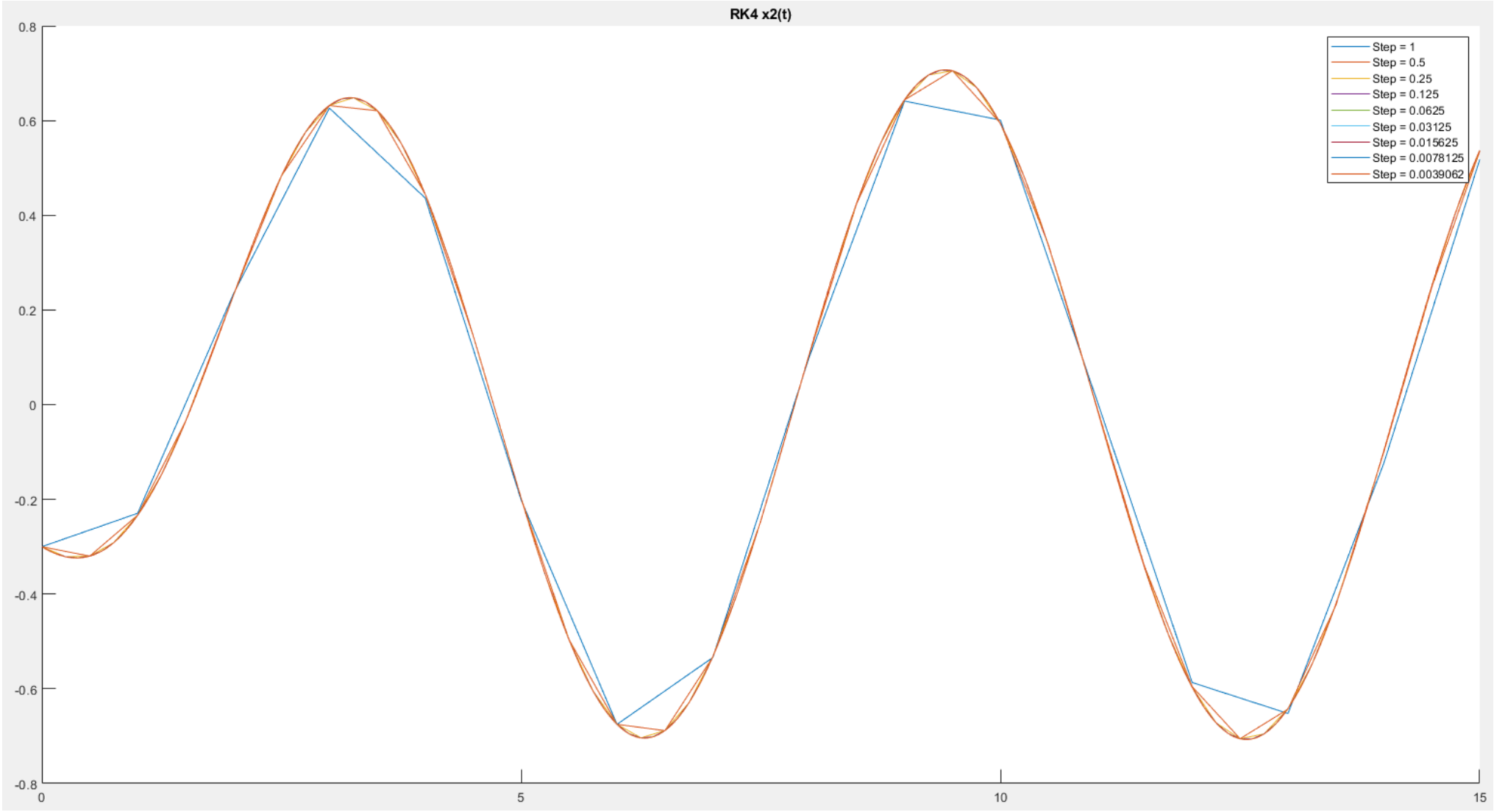
When the 1st command was executed, I noticed the initial step to be too small and the number of iterations too high, which resulted in very slow execution time. On the 2nd execution, the time was OK, and the results were appropriate.

The following plots were obtained:

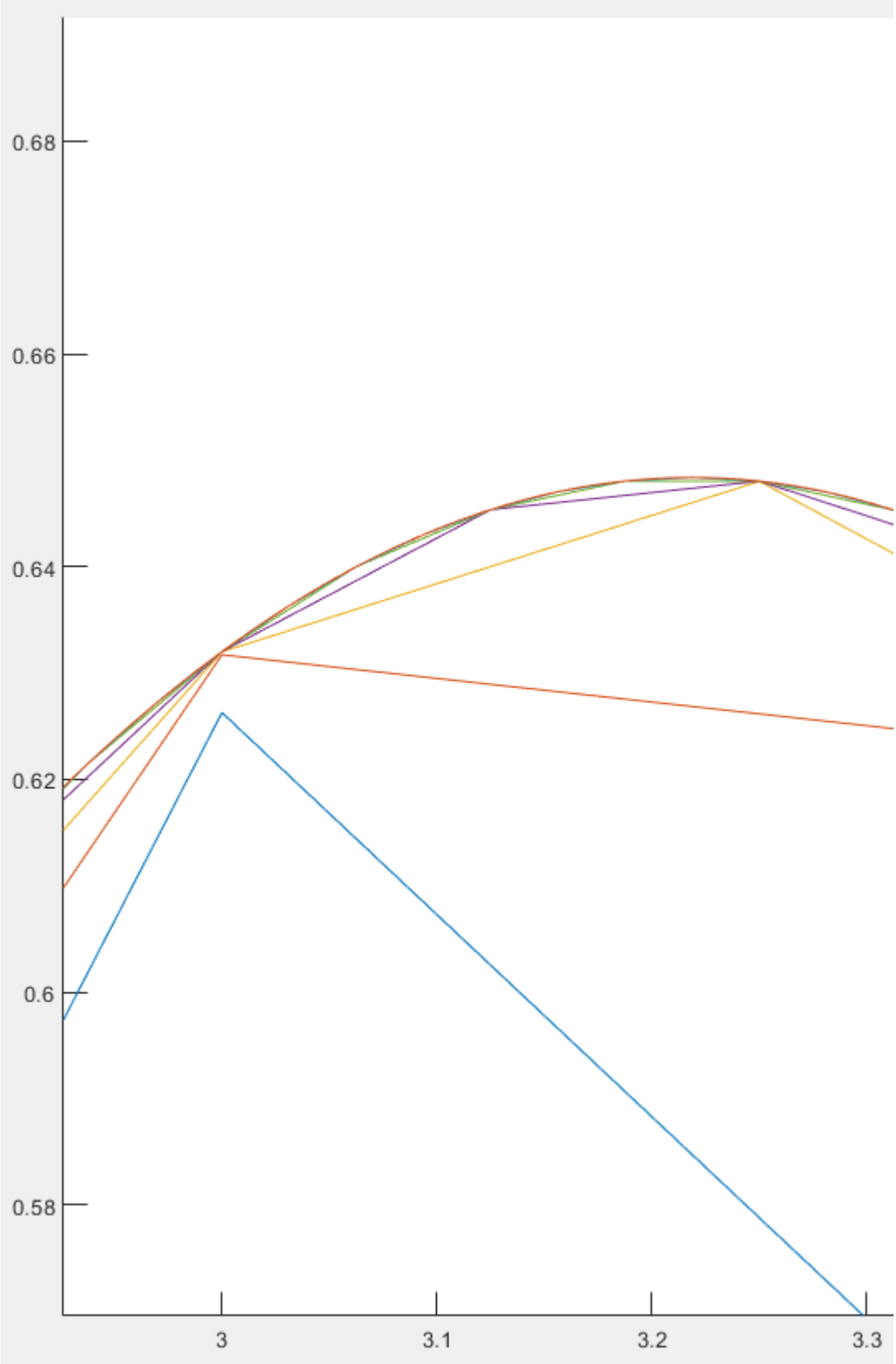

RK4 x1(t)
Step = 1
Step = 0.5
Step = 0.25
Step = 0.125
Step = 0.0625
Step = 0.03125
Step = 0.015625
Step = 0.0078125
Step = 0.0039062
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15

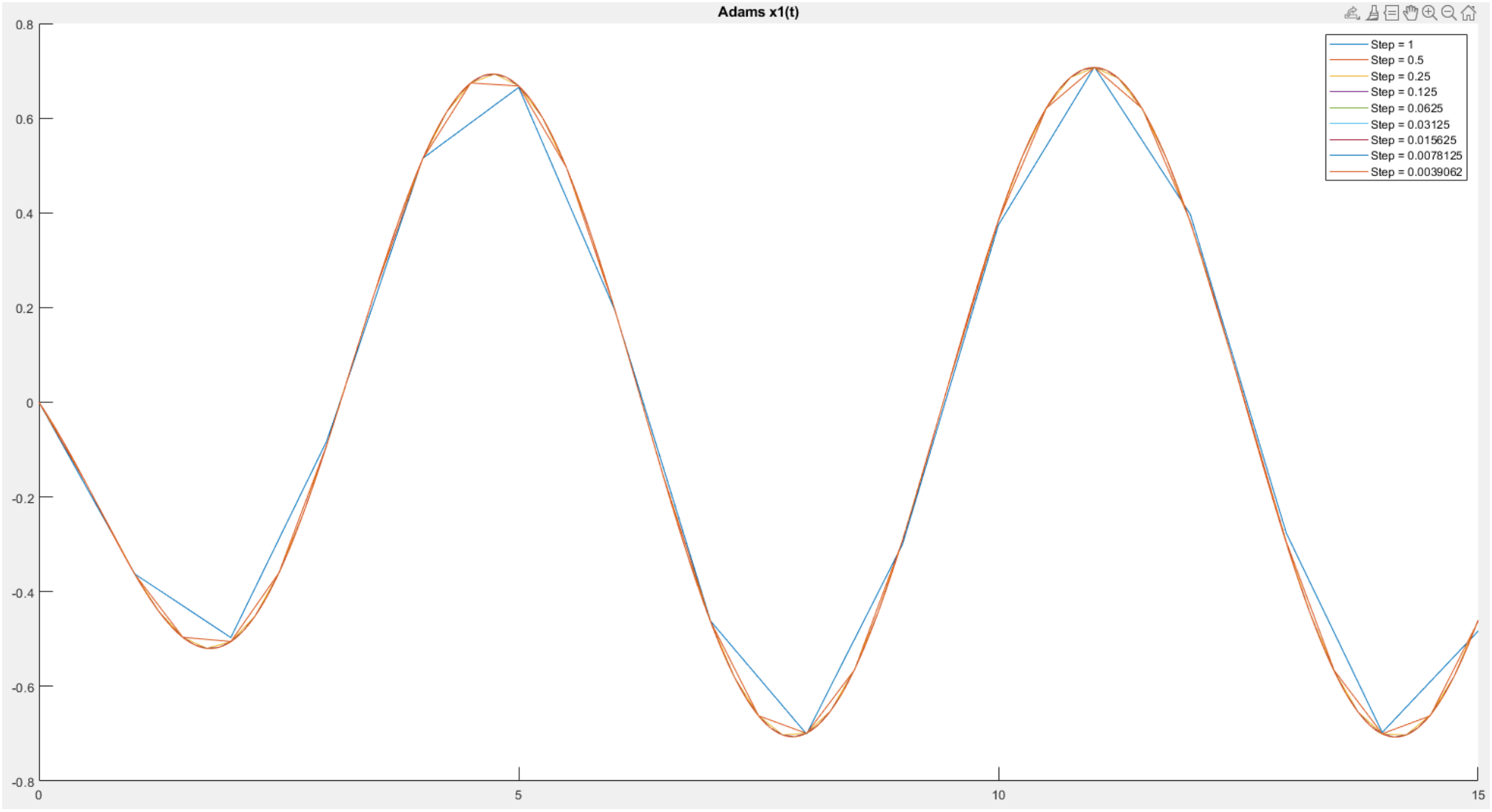
Zoom of RK4 x1(t). Refer to legend on the full screenshot:



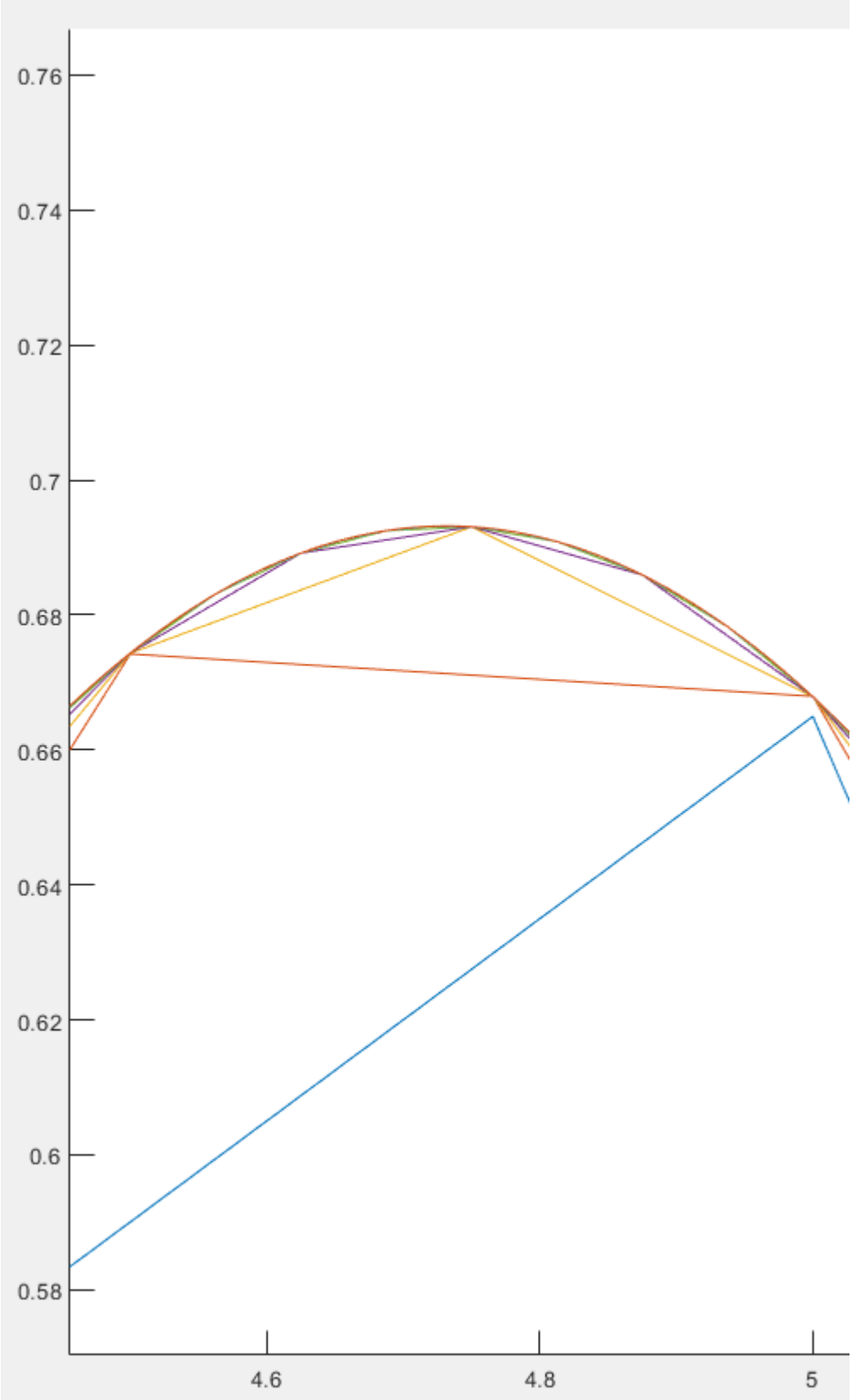

RK4 x2(t)
Step = 1
Step = 0.5
Step = 0.25
Step = 0.125
Step = 0.0625
Step = 0.03125
Step = 0.015625
Step = 0.0078125
Step = 0.0039062
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15

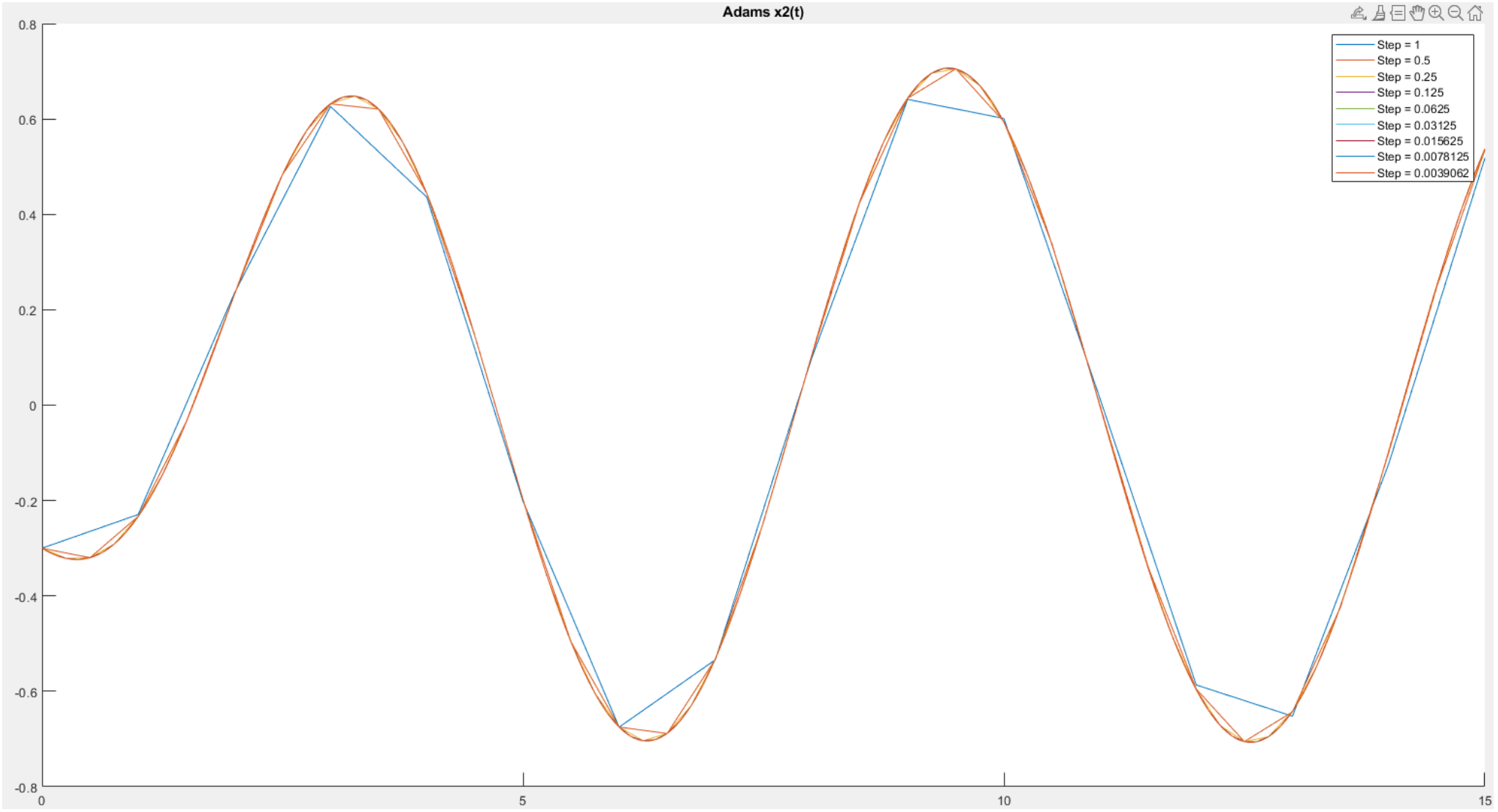
Zoom of RK4 x2(t). Refer to legend on the full screenshot:



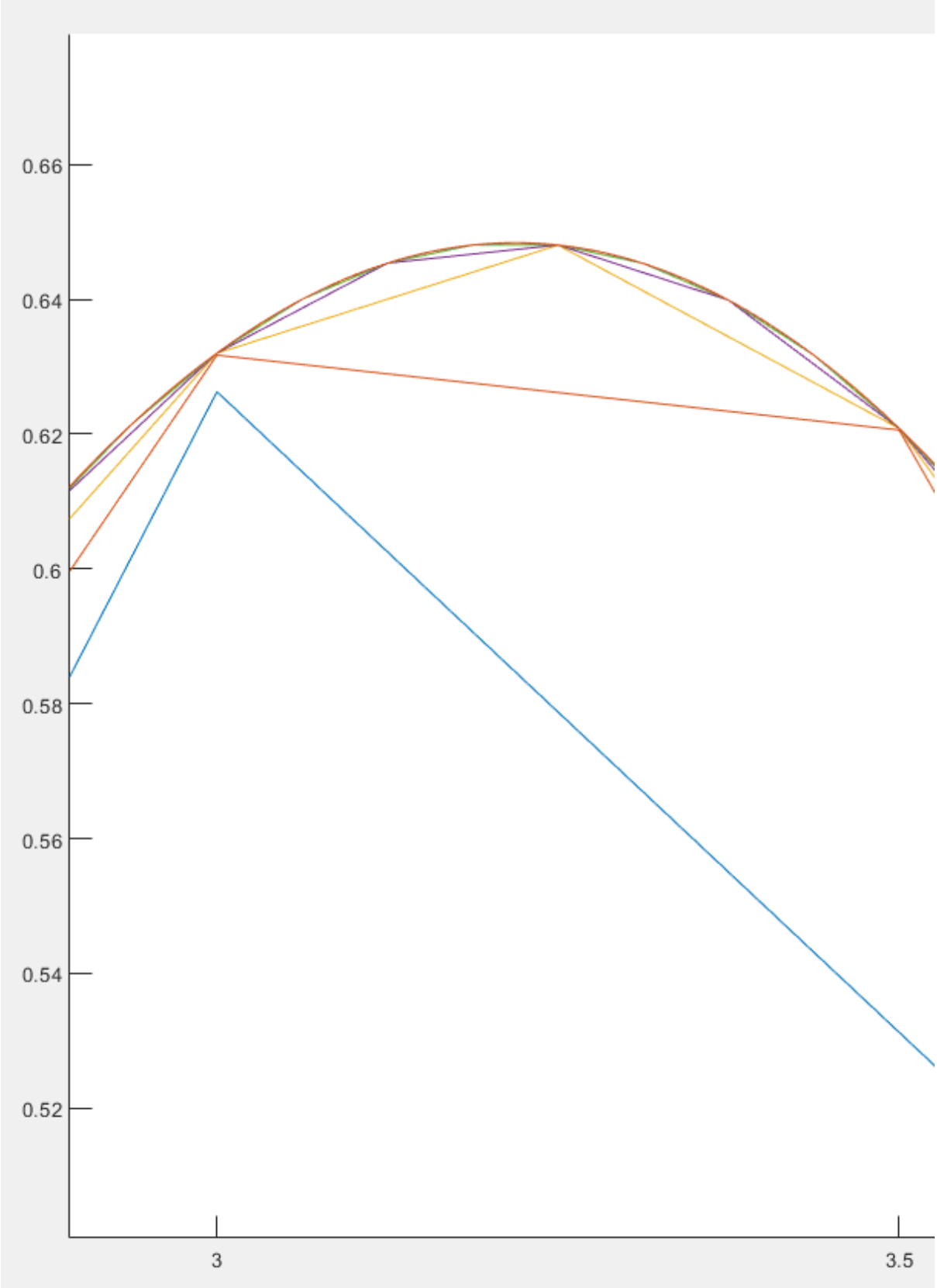

Adams x1(t)
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15
Step = 1
Step = 0.5
Step = 0.25
Step = 0.125
Step = 0.0625
Step = 0.03125
Step = 0.015625
Step = 0.0078125
Step = 0.0039062

Zoom of Adam’s PC x1(t). Refer to legend on the full screenshot:




Adams x2(t)
Step = 1
Step = 0.5
Step = 0.25
Step = 0.125
Step = 0.0625
Step = 0.03125
Step = 0.015625
Step = 0.0078125
Step = 0.0039062
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15

Zoom of Adam’s PC x2(t). Refer to legend on the full screenshot:



Optimal step size - conclusion

Looking at the graphs we can determine the optimal step size. The optimal step size should be chosen according to the rule:

“Increase of the step size does not change the solution significantly, but the decrease has a noticeable influence on the solution”
According to the task .pdf

Taking a look at the zooms of the plots we can notice a difference between the orange lines and yellow lines is definitely noticeable and significant, but between yellow and other higher precision lines, the difference is not that noticeable.
From this we can draw a conclusion that the step corresponding to the yellow line is our optimal step. In that case h_optimal = 0.25
Note: orange lines are for h = 2 * h_optimal = 0.5

Plot comparison - MATLAB implementation

Below I show the function in matlab that was used for two different step comparison:

```
% Function that compares initial_h plots vs optimal_h plots
% Initial_h is h that was one iteration lower than found optimal_h
function initial_vs_optimal(obj, initial_h, optimal_h)
    RK4_plots = figure;
    Adams_plots = figure;
    x1x2_plots = [figure; figure];
    % Below will be used in title() and legend():
    h_info_string = ["initial-h = " + initial_h, "optimal-h = " + optimal_h];
    % Fetch results of the algorithms:
    % Since t has different sizes for initial and optimal_h, we can
    % not use an array. Same goes for x
    [t1, x1(:, :, 1)] = RK4(obj, initial_h, false);
    [t2, x2(:, :, 1)] = RK4(obj, optimal_h, false);
    [~, x1(:, :, 2)] = Adams_PECE_5(obj, initial_h);
    [~, x2(:, :, 2)] = Adams_PECE_5(obj, optimal_h);
    % We dont need to store t(3) and t(4) because the values will
    % be the same as t axis values from RK4 algorithm
    % Last index of x1 and x2 indicates whether data is for RK4 or
    % Adams. 1 means RK4, 2 means Adams
    % x1 indicates result size for initial_h, x2 indicates result
    % size for optimal_h
```

```
figure(RK4_plots); % RK4 plot operations
hold on;
plot(t1, x1(:, :, 1)); % Plot RK4 for initial_h
plot(t2, x2(:, :, 1)); % Plot RK4 for optimal_h
% Set title and legend:
title("x1(t) x2(t) RK4 comparison. " + h_info_string(1) + " " + h_info_string(2));
legend("x1(t) " + h_info_string(1), "x2(t) " + h_info_string(1), "x1(t) " + h_info_string(2), "x2(t) " + h_info_string(2));
hold off;
```

```
figure(Adams_plots); % Adams PECE plot operations, analogical to RK4 plot operations
hold on;
% Remember that t values are the same as for RK4, so we reuse
plot(t1, x1(:, :, 2)); % Plot Adams PC for initial_h
plot(t2, x2(:, :, 2)); % Plot Adams PC for optimal_h
% Set title and legend:
title("x1(t) x2(t) Adams PC comparison. " + h_info_string(1) + " " + h_info_string(2));
legend("x1(t) " + h_info_string(1), "x2(t) " + h_info_string(1), "x1(t) " + h_info_string(2), "x2(t) " + h_info_string(2));
hold off;
```

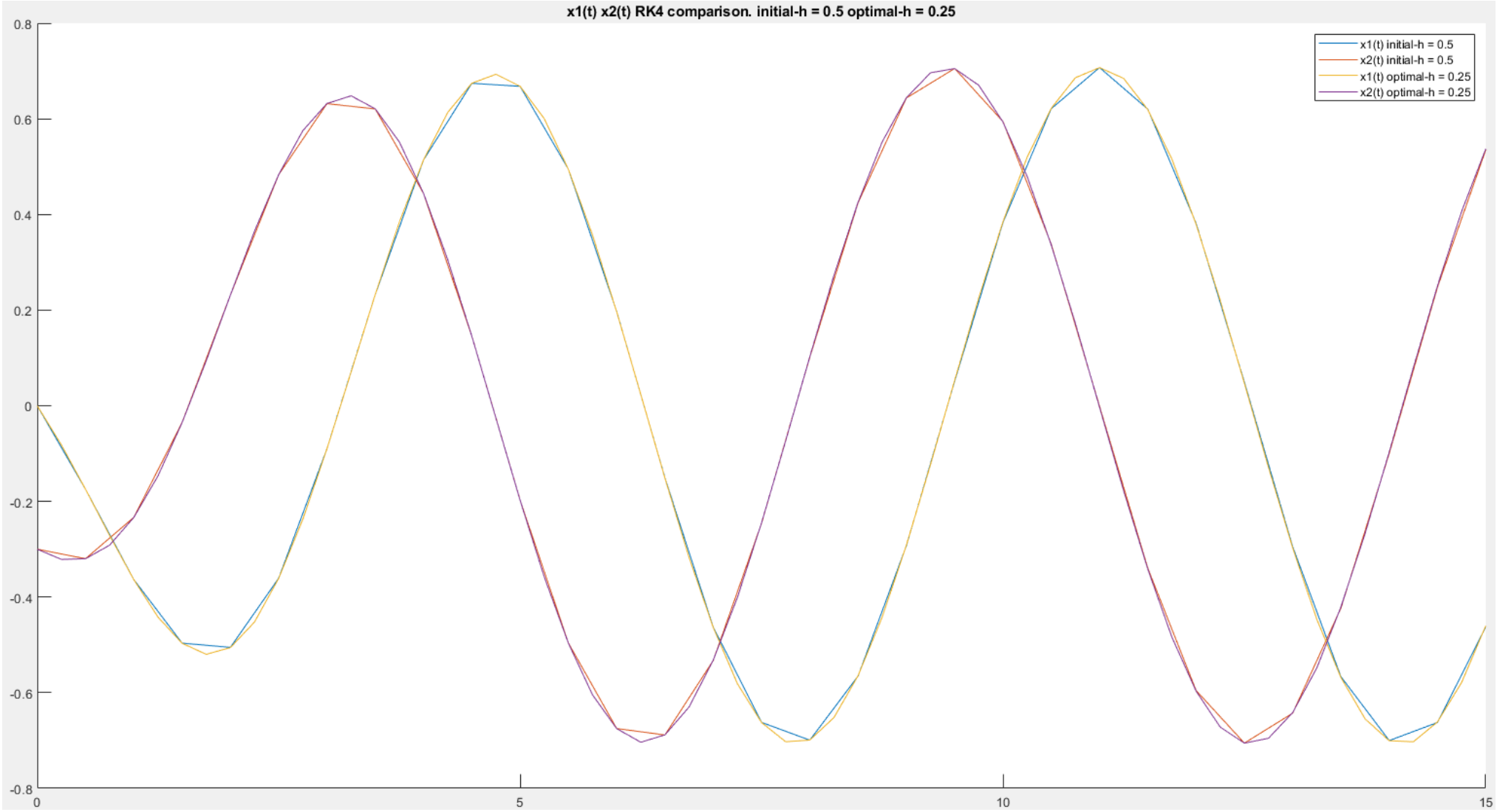
```
figure(x1x2_plots(1)); % x1x2 plots comparison - RK4 part
hold on;
% RK4 needs to have last index = 1. Rest is the same as typical
% x1(x2) plot. x1 with last index = 1 is RK4, initial_h
% x2 with last index = 1 is RK4 for optimal_h
plot(x1(1, :, 1), x1(2, :, 1));
plot(x2(1, :, 1), x2(2, :, 1));
% Set title and legend:
title("x1(x2) RK4 comparison. " + h_info_string(1) + " " + h_info_string(2));
legend(h_info_string(1), h_info_string(2));
hold off;

figure(x1x2_plots(2)); % x1x2 plots comparison - Adams PC part
hold on;
% Adams PC needs to have last index = 2. Rest is the same as typical
% x1(x2) plot. x1 with last index = 2 is Adams PC, initial_h
% x2 with last index = 2 is Adams PC for optimal_h
plot(x1(1, :, 2), x1(2, :, 2));
plot(x2(1, :, 2), x2(2, :, 2));
% Set title and legend:
title("x1(x2) Adams PC comparison. " + h_info_string(1) + " " + h_info_string(2));
legend(h_info_string(1), h_info_string(2));
hold off;
```

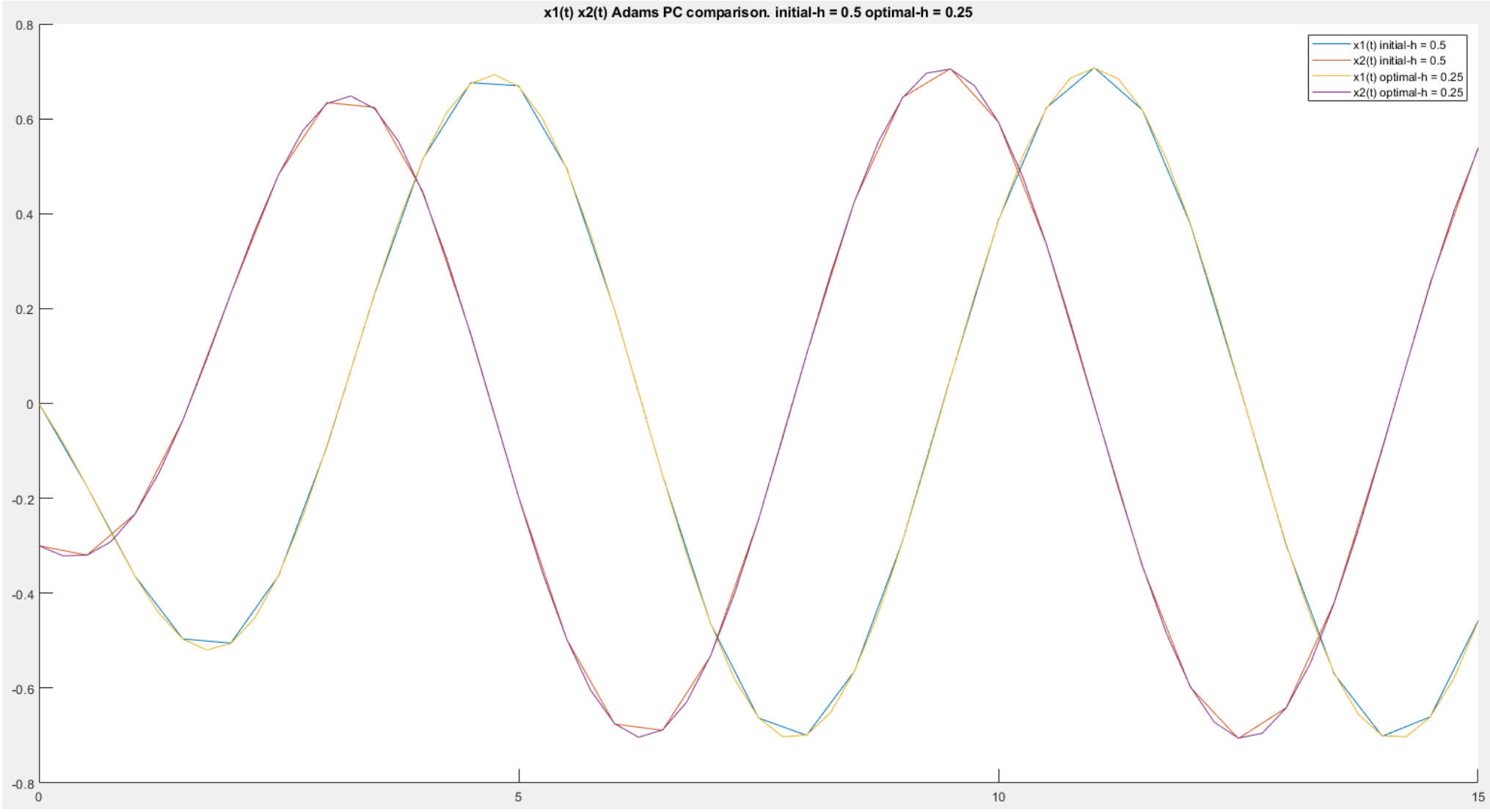
Plot comparison - higher step vs optimal step

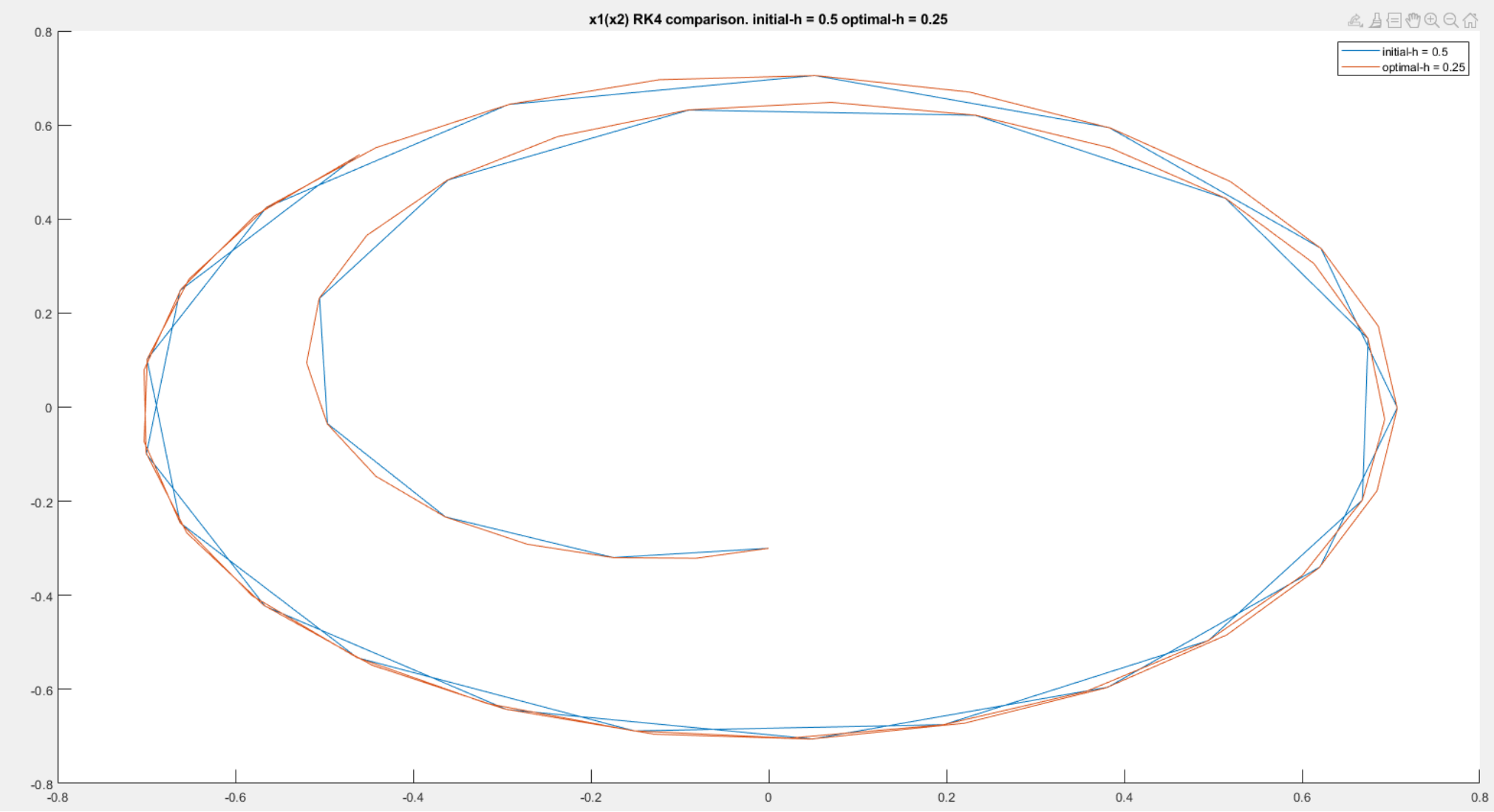
Now I will show plots for RK4, Adam's PC and x1(x2) comparison for each of the mentioned algorithms for two steps - chosen high step and for our obtained optimal. Following commands were run:

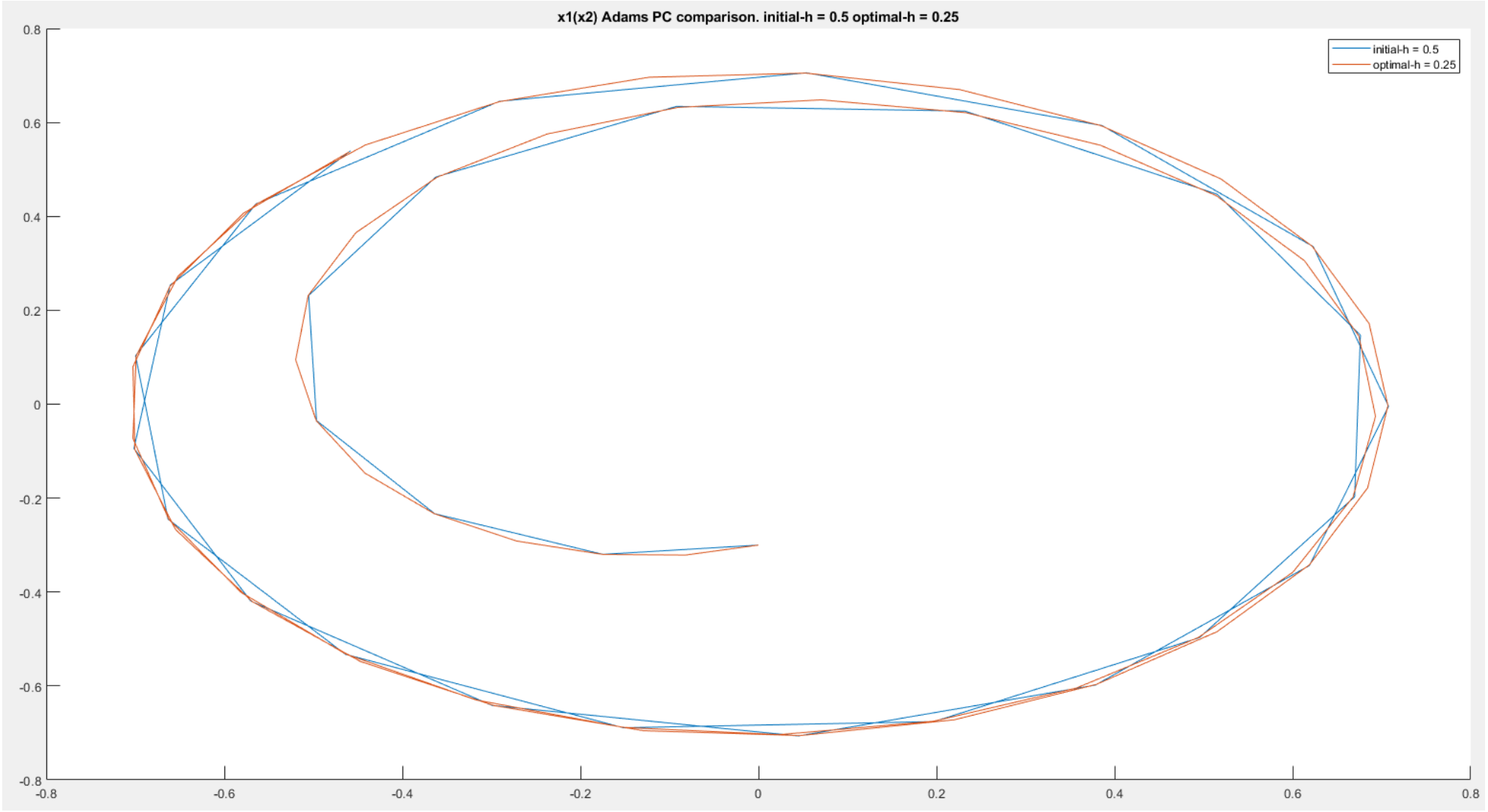
```
>> my_ode = task_2()

my_ode =

  task_2 with properties:

        odefun: @(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)]
    t_interval: [0 10]
        x_init: [0 -0.3000]

>> initial_vs_optimal(my_ode, 0.5, 0.25)
>> initial vs optimal(my ode, 0.25, 0.125)
```

Note: For x1(x2) graphs, at some point lines overlap. It is better to refer to the middle of the graphs for comparison

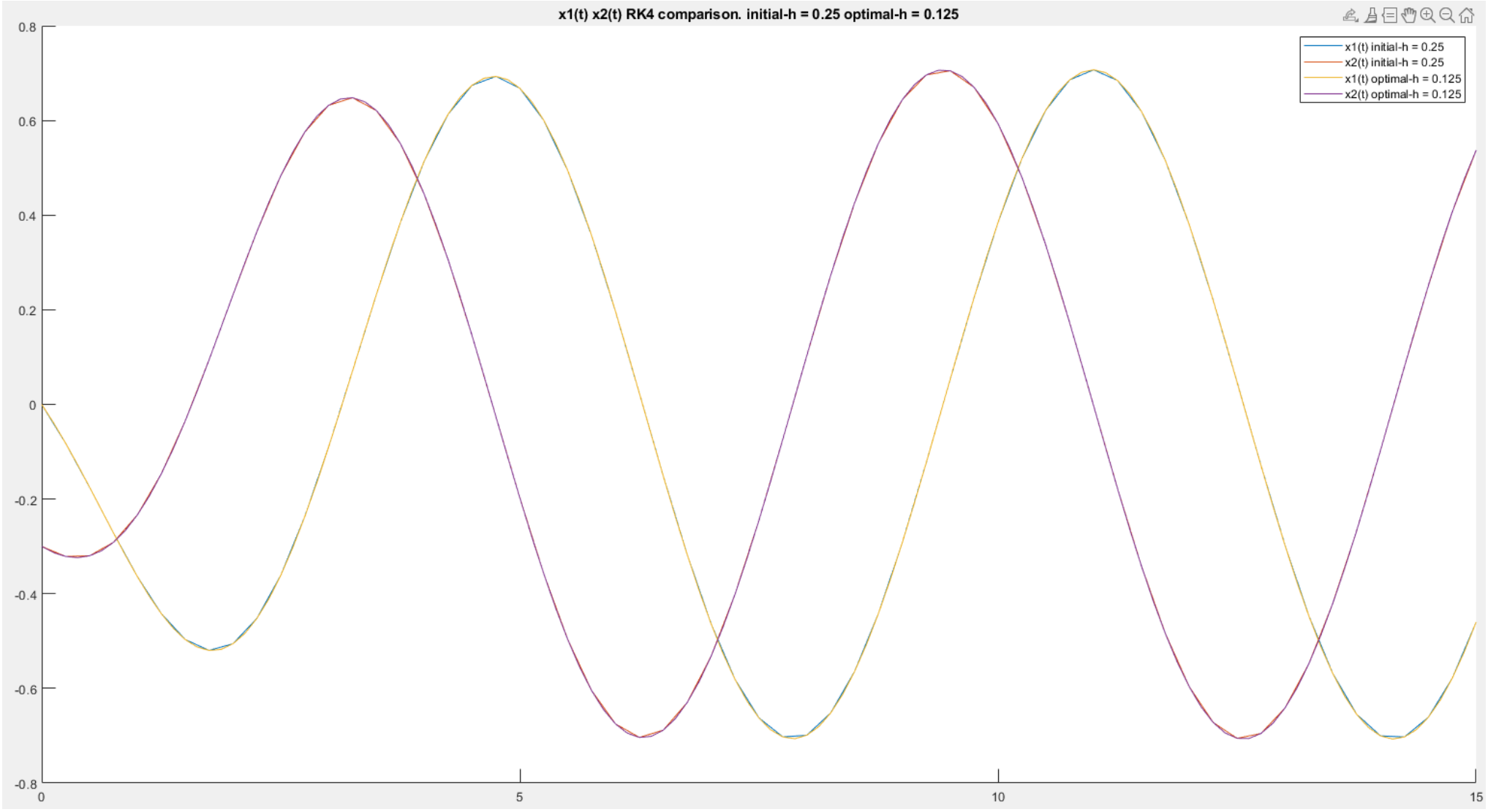

x1(t) x2(t) RK4 comparison. initial-h = 0.5 optimal-h = 0.25
x1(t) initial-h = 0.5
x2(t) initial-h = 0.5
x1(t) optimal-h = 0.25
x2(t) optimal-h = 0.25
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15

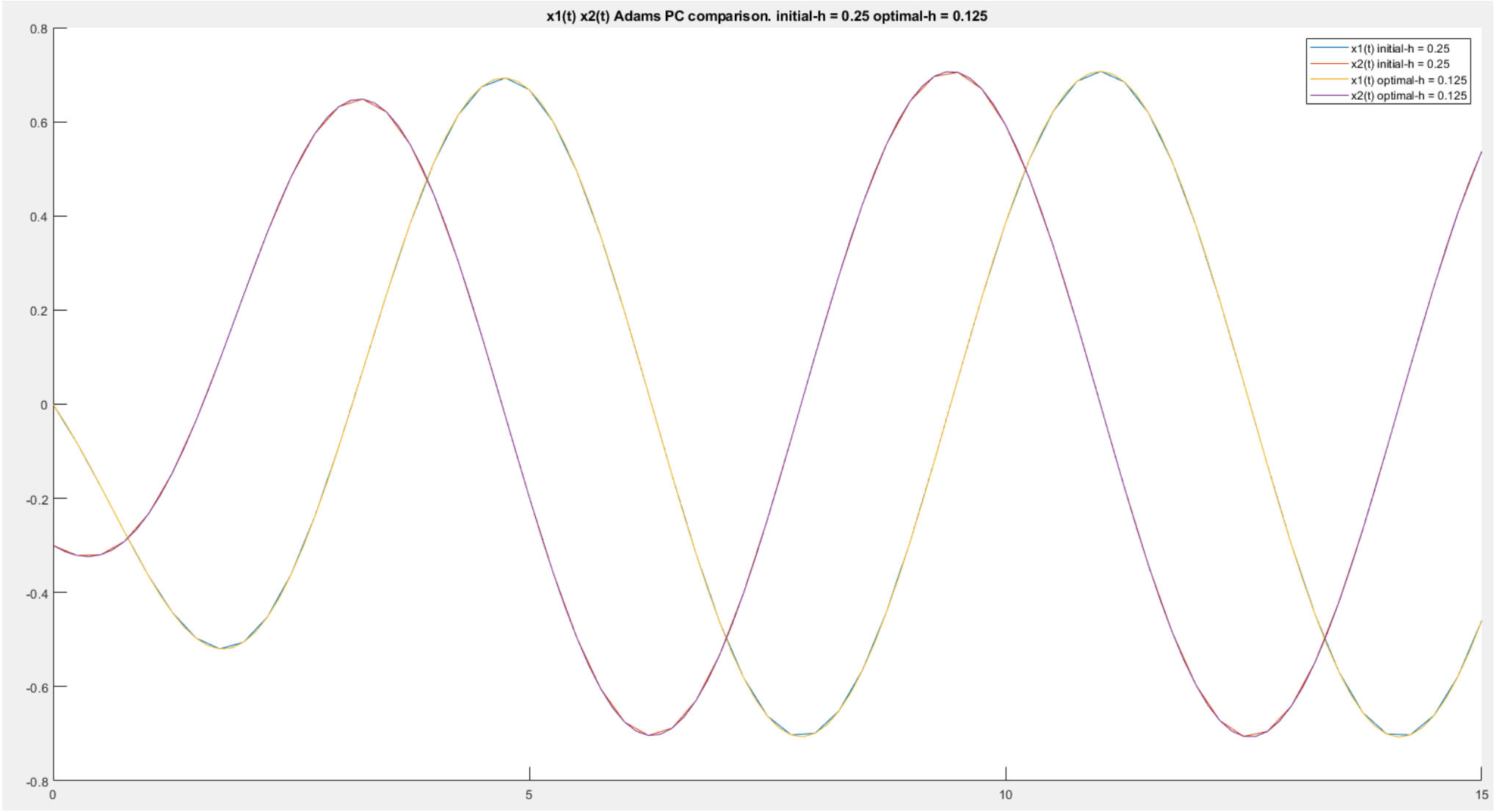

x1(t) x2(t) Adams PC comparison. initial-h = 0.5 optimal-h = 0.25
x1(t) initial-h = 0.5
x2(t) initial-h = 0.5
x1(t) optimal-h = 0.25
x2(t) optimal-h = 0.25
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15

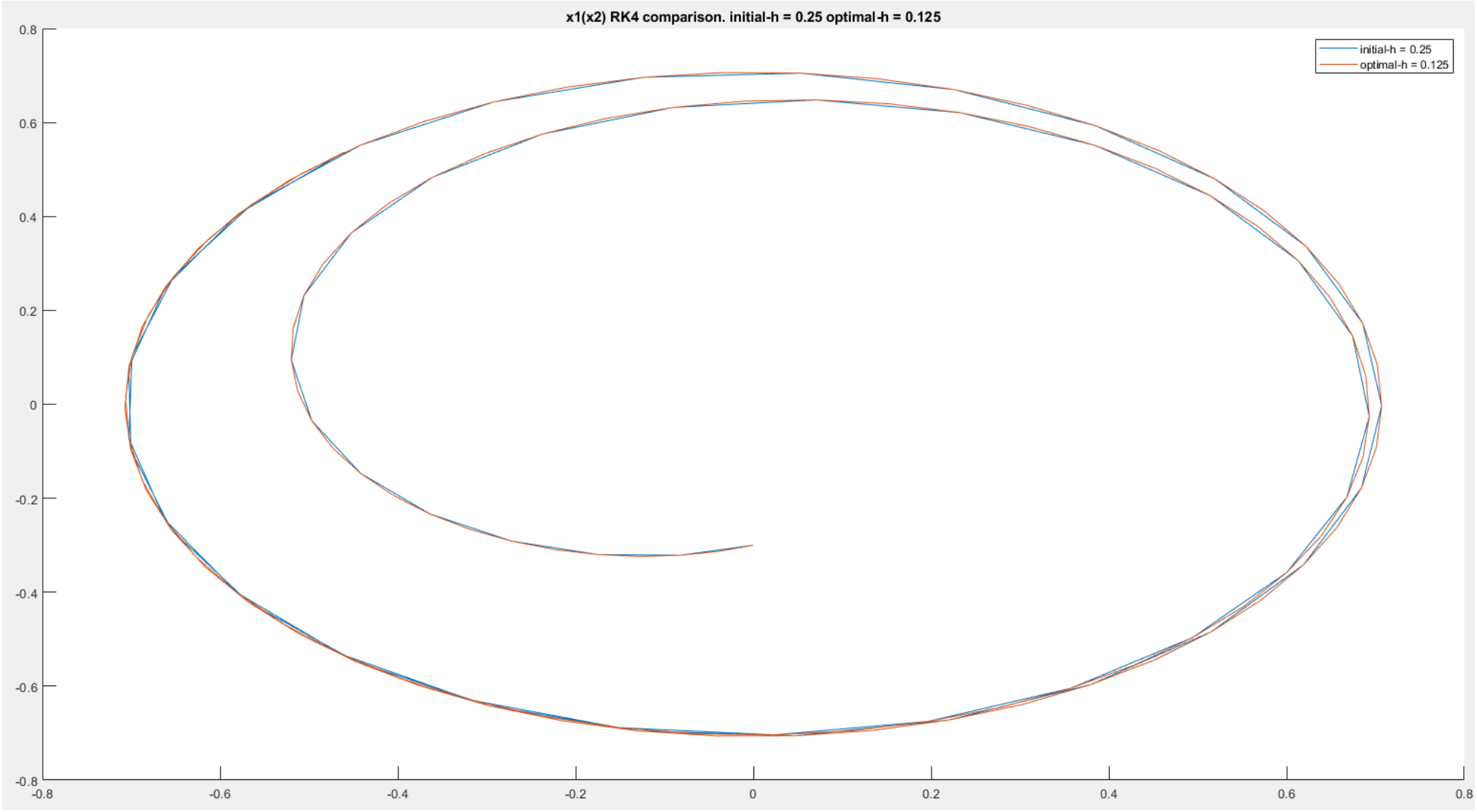

x1(x2) RK4 comparison. initial-h = 0.5 optimal-h = 0.25
initial-h = 0.5
optimal-h = 0.25
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8

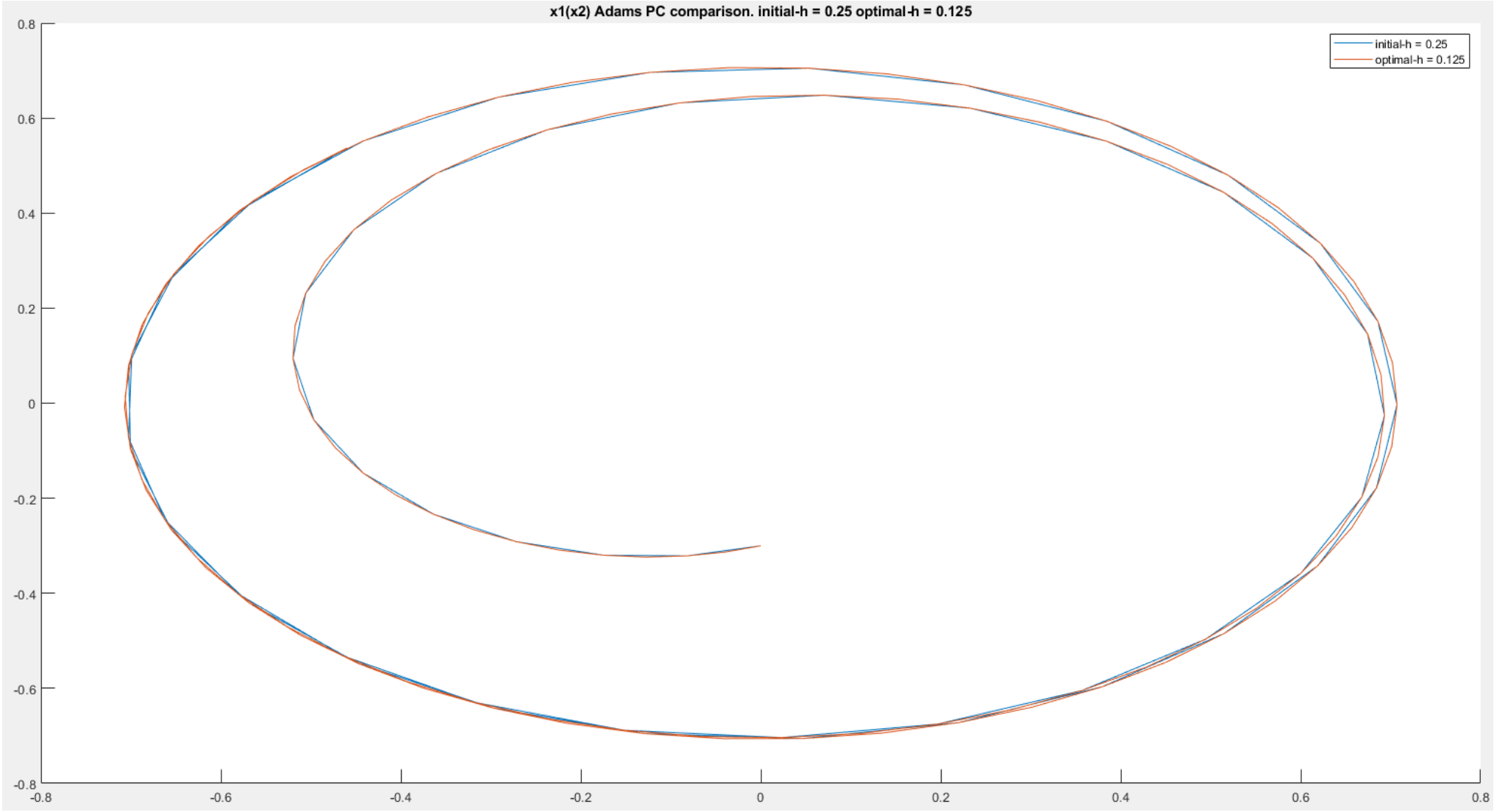

x1(x2) Adams PC comparison. initial-h = 0.5 optimal-h = 0.25
initial-h = 0.5
optimal-h = 0.25
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8

It could theoretically be argued that optimal_h = 0.125, hence I attach plots for h = 0.25 vs h = 0.125 as well. It depends on how we interpret a “big change” between different step sizes:




x1(t) x2(t) Adams PC comparison. initial-h = 0.25 optimal-h = 0.125
x1(t) initial-h = 0.25
x2(t) initial-h = 0.25
x1(t) optimal-h = 0.125
x2(t) optimal-h = 0.125
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15


x1(x2) RK4 comparison. initial-h = 0.25 optimal-h = 0.125
initial-h = 0.25
optimal-h = 0.125
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8


x1(x2) Adams PC comparison. initial-h = 0.25 optimal-h = 0.125
initial-h = 0.25
optimal-h = 0.125
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8

Task 2a - conclusions & closing remarks

As it can be noticed from the plots, the optimal step size is equal to 0.25, given that the difference between 0.25 and 0.125 is deemed as “small”.

The plots definitely show a noticeable difference between h = 0.5 and h = 0.25, whereas the difference between h = 0.25 and h = 0.125 can be deemed noticeable, depending whether we seek an extra accurate solution or not.