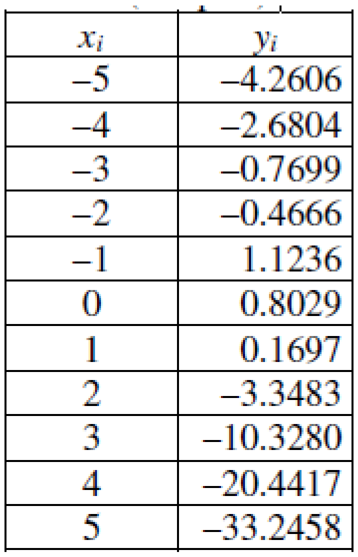
ENUME - Project 3

Author: Michał Łezka
ID: 303873

1. For the following experimental measurements (samples):

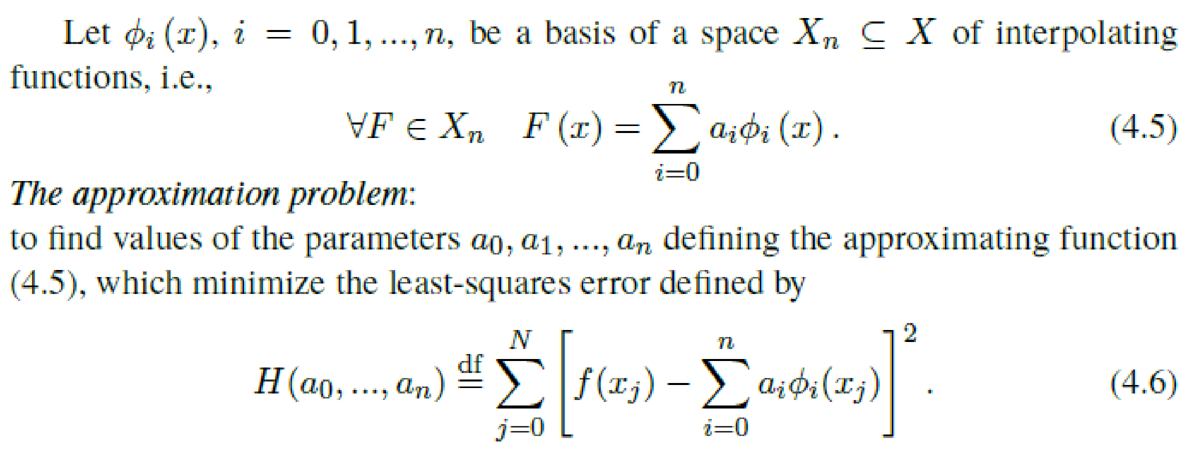


x_i	y_i
–5	–4.2606
–4	–2.6804
–3	–0.7699
–2	–0.4666
–1	1.1236
0	0.8029
1	0.1697
2	–3.3483
3	–10.3280
4	–20.4417
5	–33.2458

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.

Background

We have 11 points and 11 corresponding to them values that we can graph. Then



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

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

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

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

All screenshots in this paragraph are from the book Numerical Methods by Piotr Tatjewski unless stated otherwise

Now we need to minimise it. To this end, we look for the 1st order derivative of this error to be equal to 0:

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

i.e.,

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

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

$$\vdots$$

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

This system of equations is called the set of normal equations, and its matrix is called the Gram's matrix. The normal equations can be written in a simpler form:

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

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

Let us also define a matrix A:

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

and

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

Using this definition, we can write that $\mathbf{A}^T\mathbf{A}\mathbf{a} = \mathbf{A}^T\mathbf{y}$ where A^TA is the Gram's matrix. Since A^TA can be badly conditioned – in that case, it is better to solve the approximation problem using the method based on the thin QR factorization of A. Then the set of normal equations can be rewritten as

$$R^TQ^TQRa = R^TQ^Ty$$

because columns of the matrix Q are orthonormal, we can shorten it to:

$$Ra = Q^Ty$$

Since we are approximating a polynomial, we use the natural polynomials basis (the power basis)

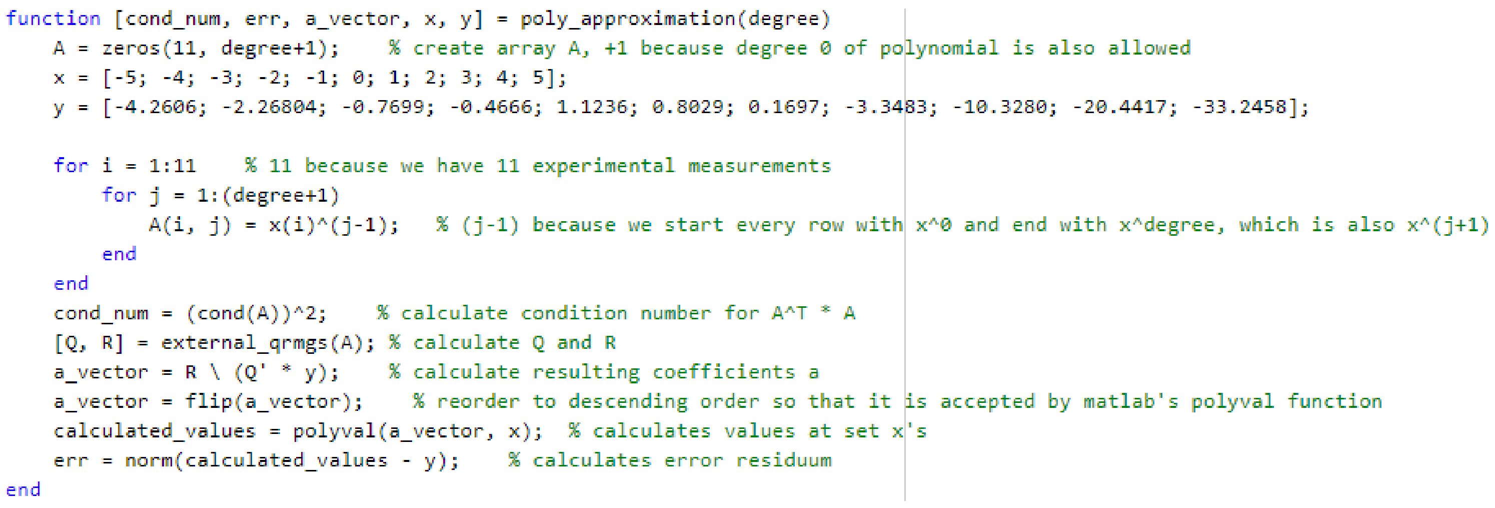
$$\phi_0(x) = 1, \;\; \phi_1(x) = x, \; \phi_2(x) = x^2, \;\; \cdots, \;\; \phi_n(x) = x^n$$

That means the initial A can be rewritten as:

$$A = \begin{bmatrix} 1 & x_0 & \cdots & x_0^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{11} & \cdots & x_{11}^n \end{bmatrix}$$

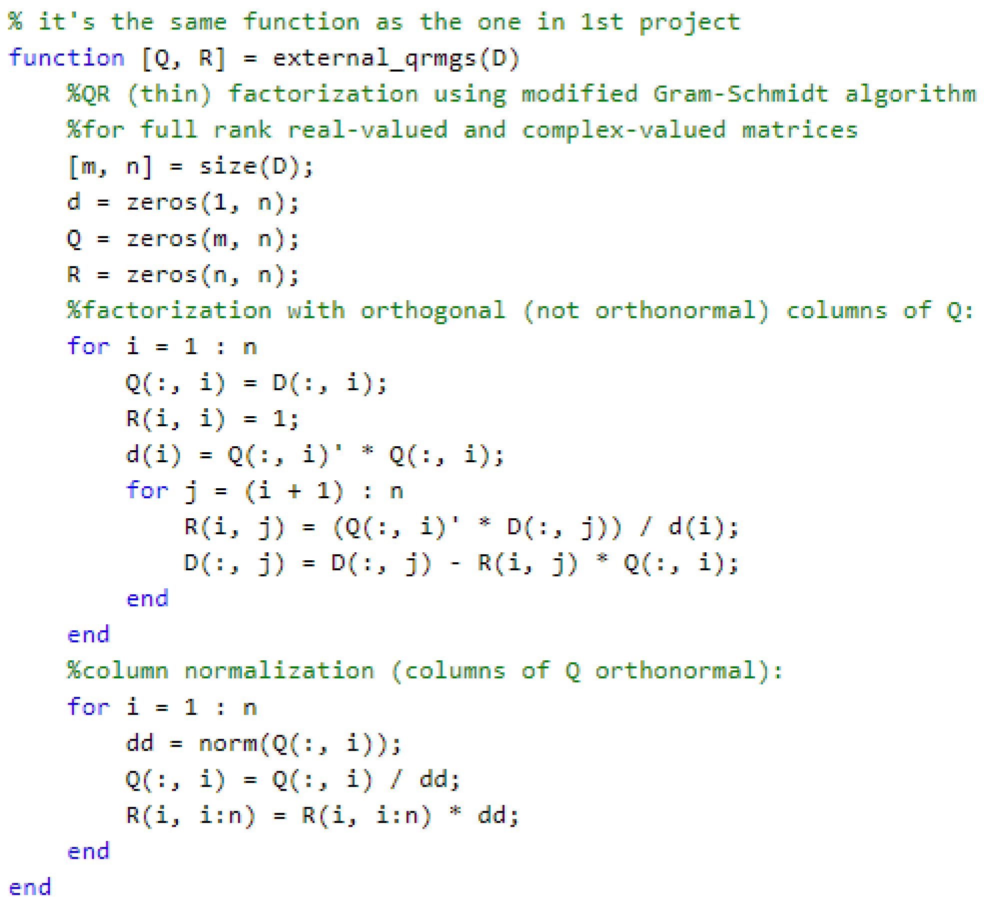
MATLAB implementation

I will start with the most important function in the program that approximates the polynomial and calculates its condition number and error residuum:



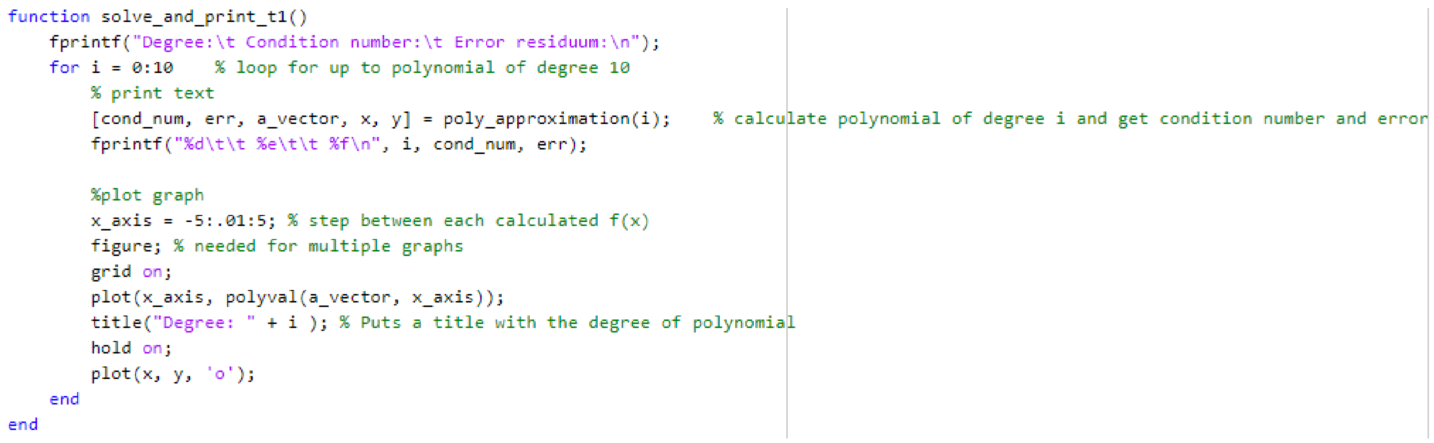
```
function [cond_num, err, a_vector, x, y] = poly_approximation(degree)
    A = zeros(11, degree+1);    % create array A, +1 because degree 0 of polynomial is also allowed
    x = [-5; -4; -3; -2; -1; 0; 1; 2; 3; 4; 5];
    y = [-4.2606; -2.26804; -0.7699; -0.4666; 1.1236; 0.8029; 0.1697; -3.3483; -10.3280; -20.4417; -33.2458];

    for i = 1:11    % 11 because we have 11 experimental measurements
        for j = 1:(degree+1)
            A(i, j) = x(i)^(j-1);   % (j-1) because we start every row with x^0 and end with x^degree, which is also x^(j+1)
        end
    end
    cond_num = (cond(A))^2;    % calculate condition number for A^T * A
    [Q, R] = external_qrmgs(A); % calculate Q and R
    a_vector = R \ (Q' * y);    % calculate resulting coefficients a
    a_vector = flip(a_vector);    % reorder to descending order so that it is accepted by matlab's polyval function
    calculated_values = polyval(a_vector, x);  % calculates values at set x's
    err = norm(calculated_values - y);    % calculates error residuum
end
```

This piece of code uses an external function external_qrmgs that have been already used in the 1st project:



```
% it's the same function as the one in 1st project
function [Q, R] = external_qrmgs(D)
    %QR (thin) factorization using modified Gram-Schmidt algorithm
    %for full rank real-valued and complex-valued matrices
    [m, n] = size(D);
    d = zeros(1, n);
    Q = zeros(m, n);
    R = zeros(n, n);
    %factorization with orthogonal (not orthonormal) columns of Q:
    for i = 1 : n
        Q(:, i) = D(:, i);
        R(i, i) = 1;
        d(i) = Q(:, i)' * Q(:, i);
        for j = (i + 1) : n
            R(i, j) = (Q(:, i)' * D(:, j)) / d(i);
            D(:, j) = D(:, j) - R(i, j) * Q(:, i);
        end
    end
    %column normalization (columns of Q orthonormal):
    for i = 1 : n
        dd = norm(Q(:, i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:n) = R(i, i:n) * dd;
    end
end
```

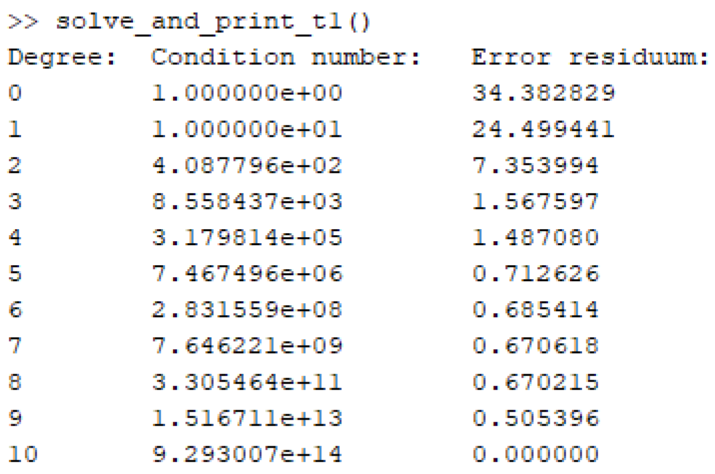
The last function is responsible for the output, to that end it uses the previously presented functions:



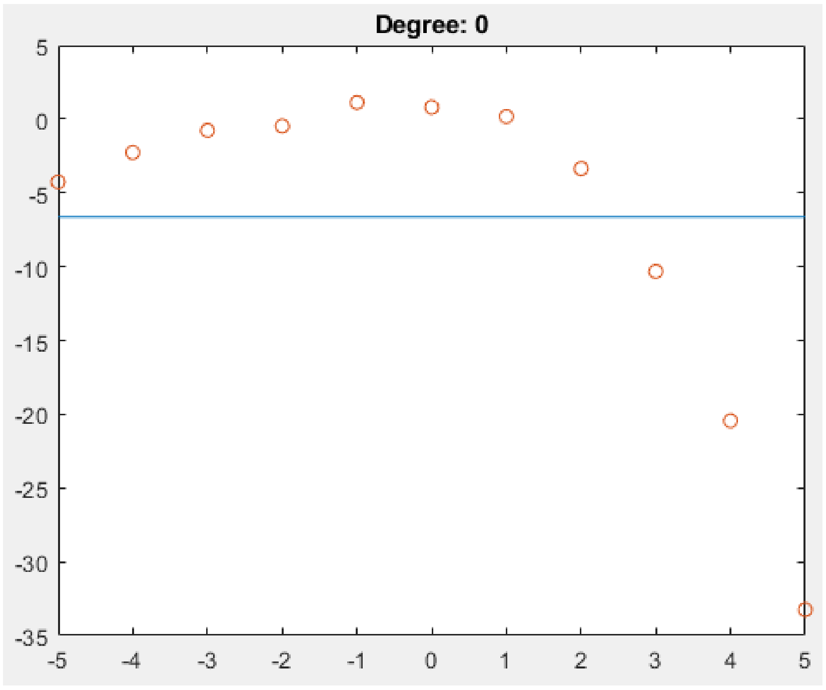
```
function solve_and_print_t1()
    fprintf("Degree:\t Condition number:\t Error residuum:\n");
    for i = 0:10    % loop for up to polynomial of degree 10
        % print text
        [cond_num, err, a_vector, x, y] = poly_approximation(i);    % calculate polynomial of degree i and get condition number and error
        fprintf("%d\t\t %e\t\t %f\n", i, cond_num, err);

        %plot graph
        x_axis = -5:.01:5; % step between each calculated f(x)
        figure; % needed for multiple graphs
        grid on;
        plot(x_axis, polyval(a_vector, x_axis));
        title("Degree: " + i ); % Puts a title with the degree of polynomial
        hold on;
        plot(x, y, 'o');
    end
end
```

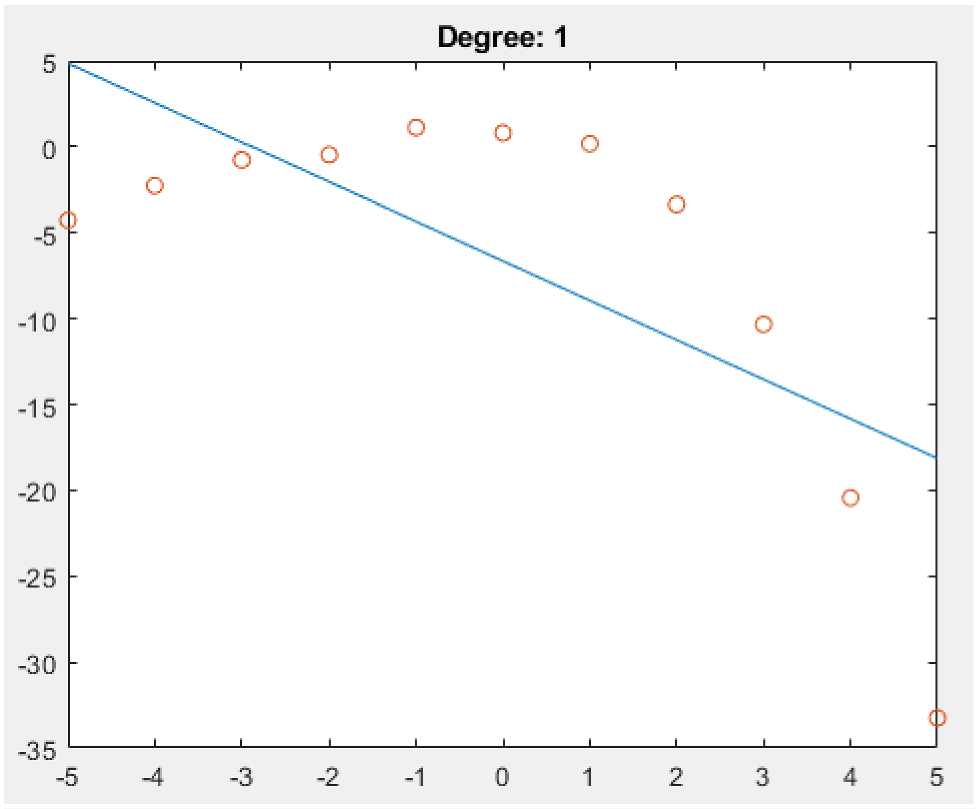
Results

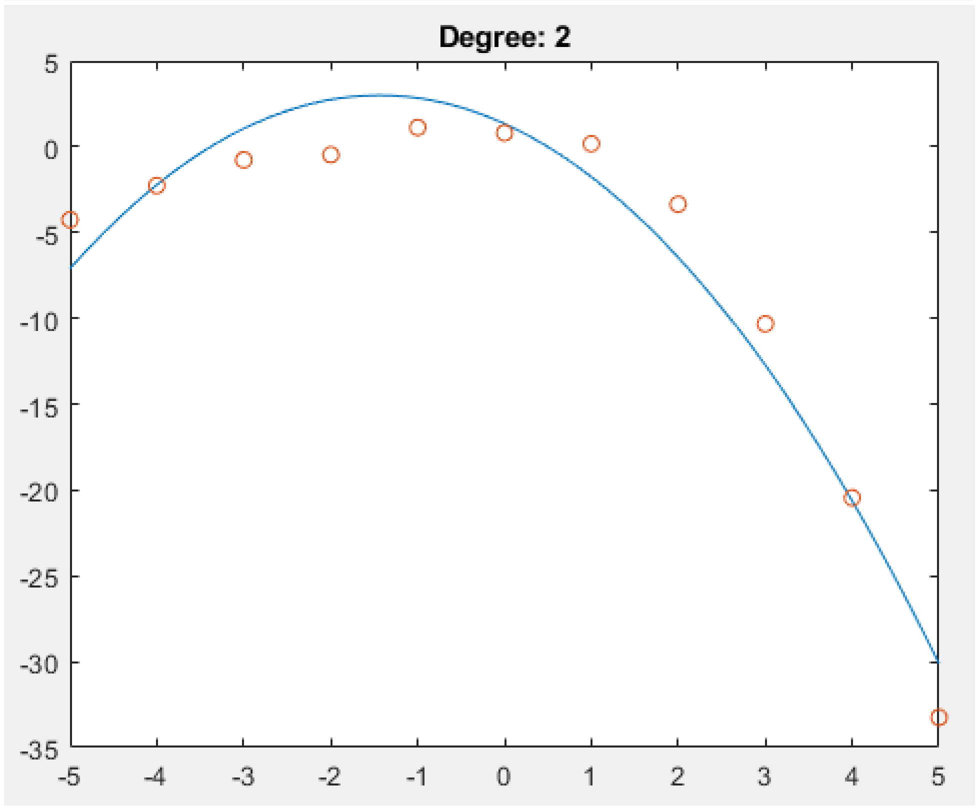
To start the program, use **solve_and_print_t1()**

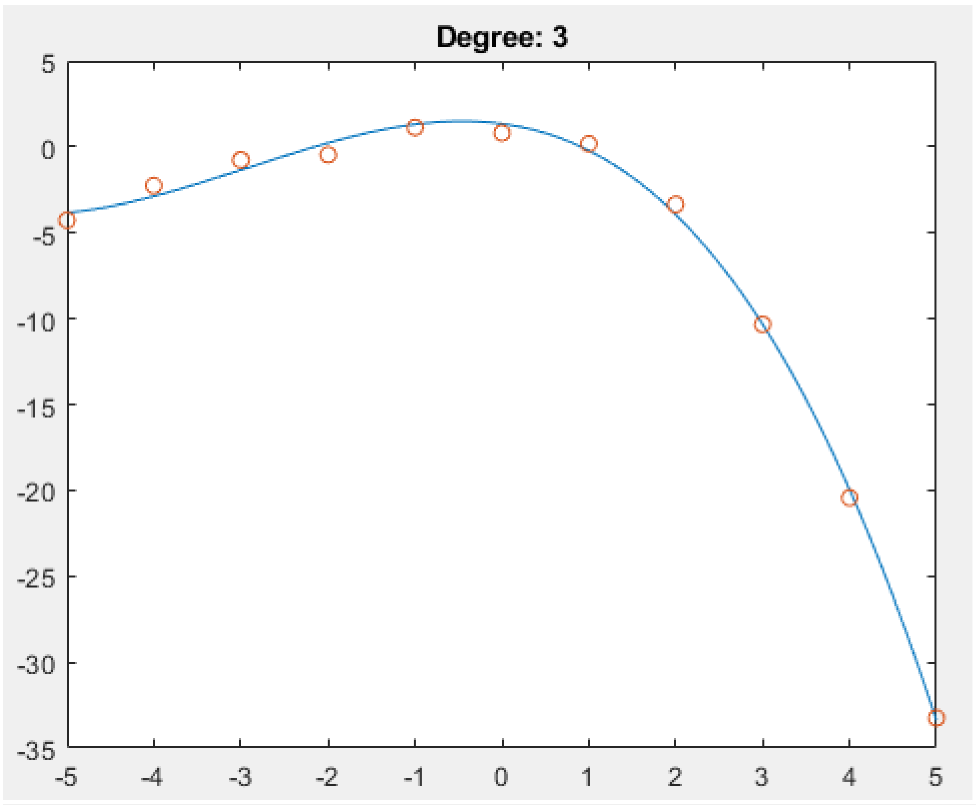


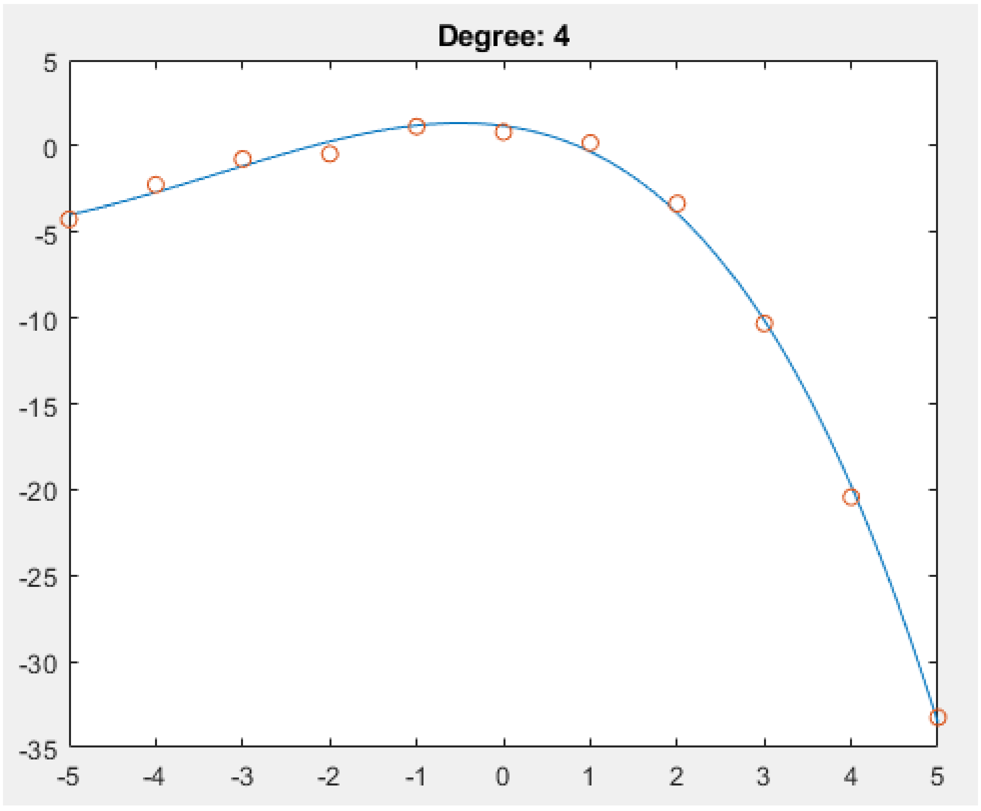
```
>> solve_and_print_t1()
Degree:  Condition number:    Error residuum:
0        1.000000e+00         34.382829
1        1.000000e+01         24.499441
2        4.087796e+02         7.353994
3        8.558437e+03         1.567597
4        3.179814e+05         1.487080
5        7.467496e+06         0.712626
6        2.831559e+08         0.685414
7        7.646221e+09         0.670618
8        3.305464e+11         0.670215
9        1.516711e+13         0.505396
10       9.293007e+14         0.000000
```

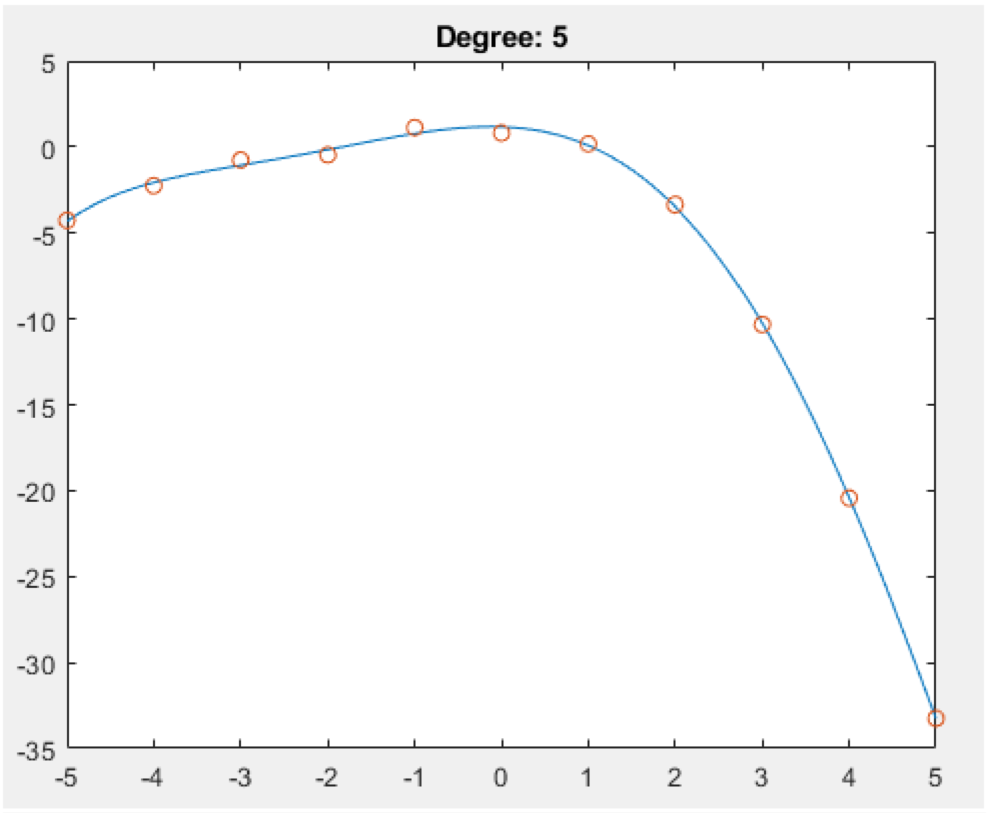


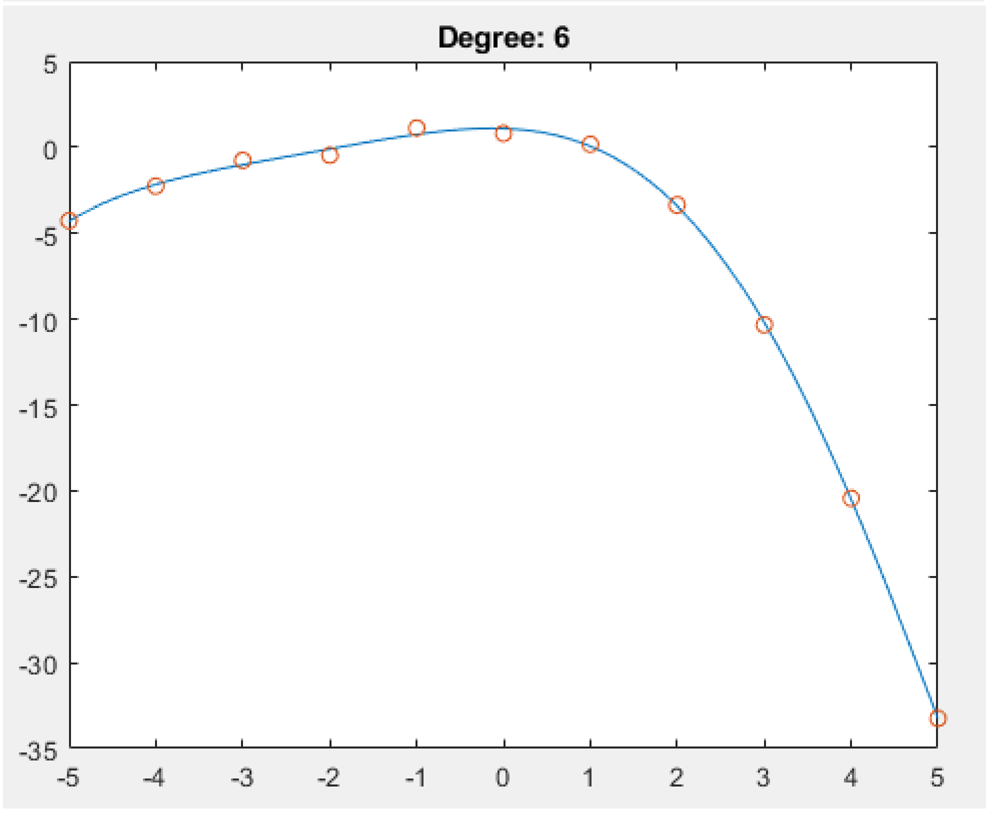

Degree: 1
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

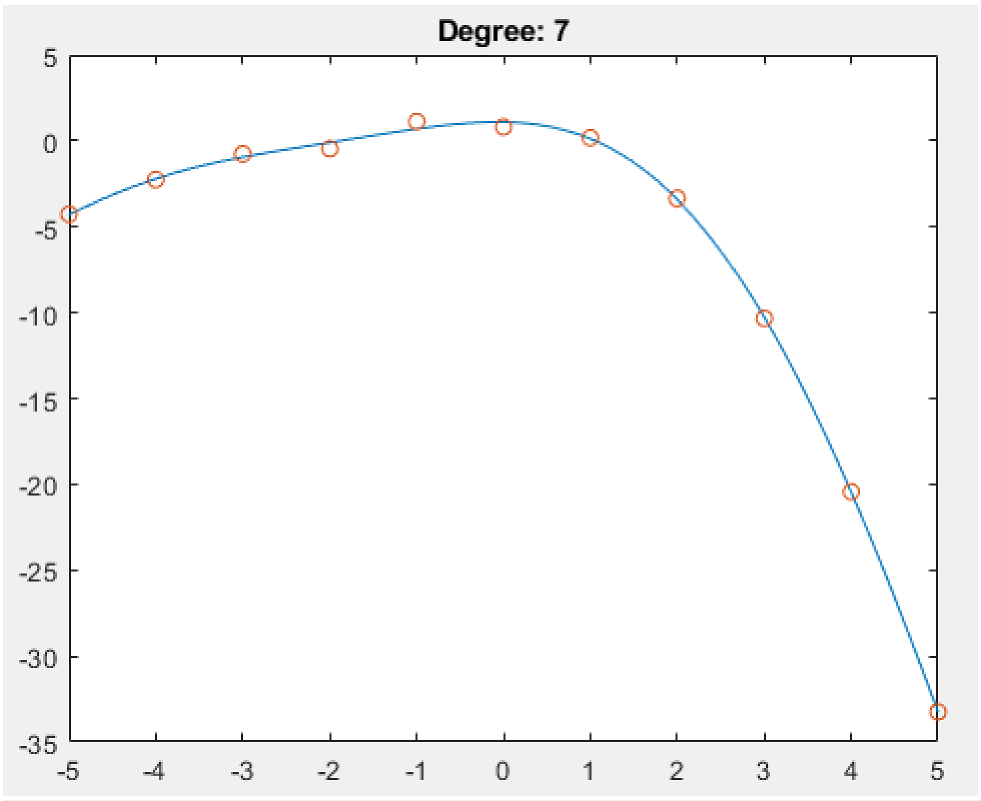
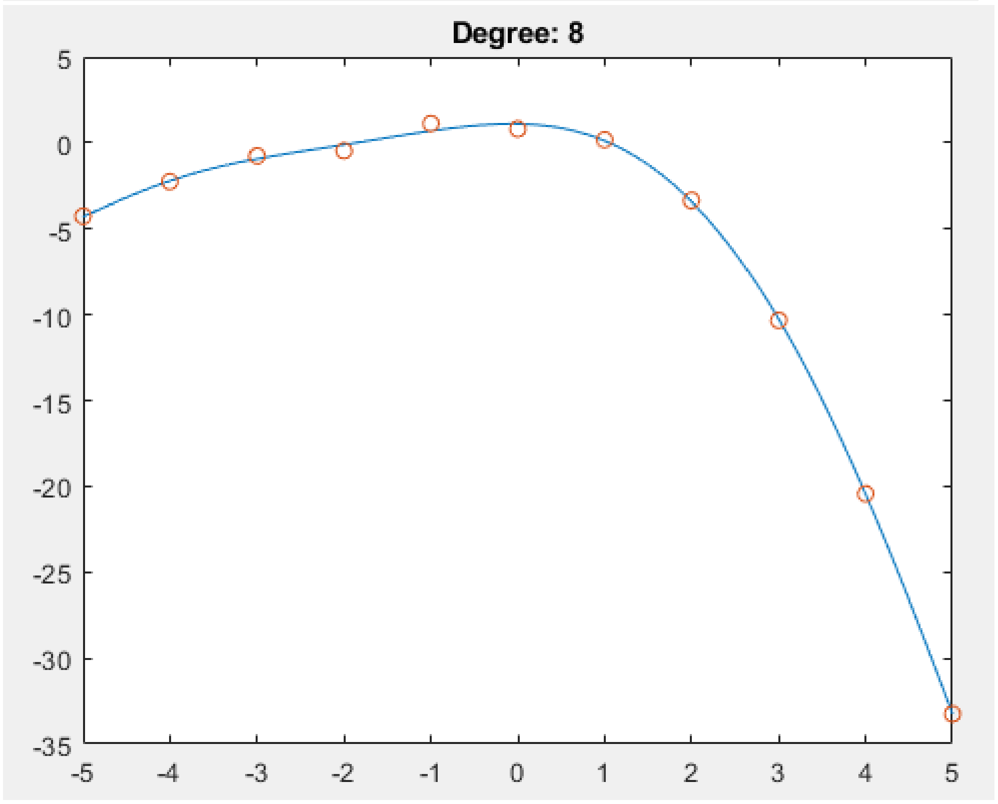

Degree: 2
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

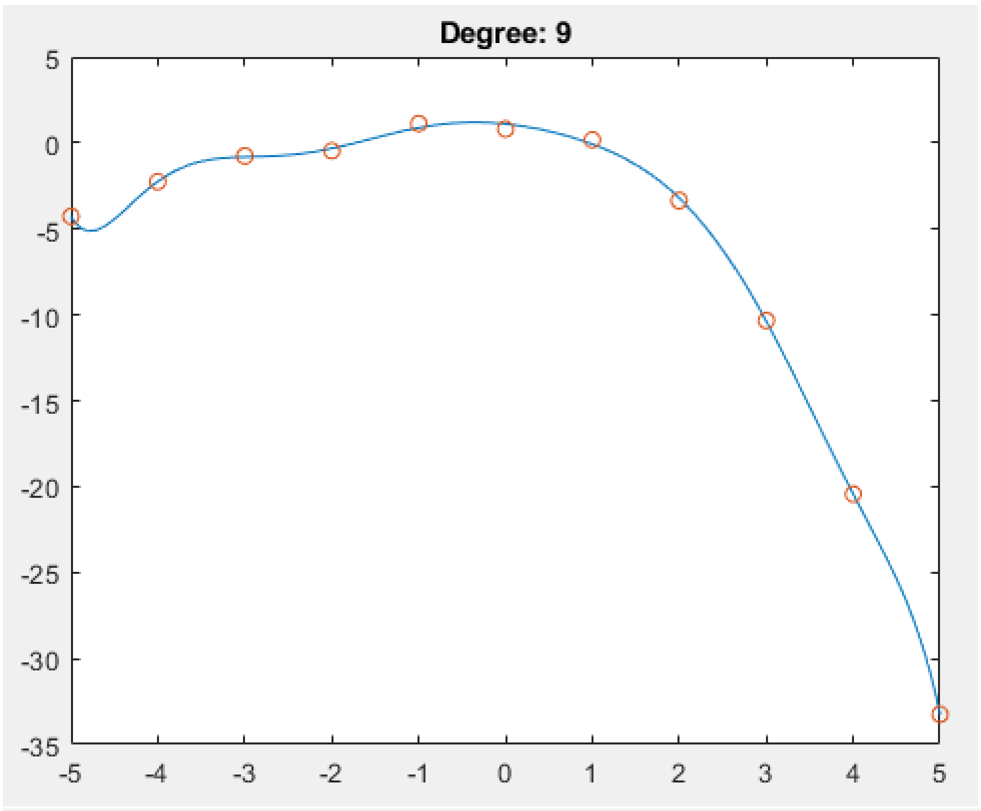

Degree: 3
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5

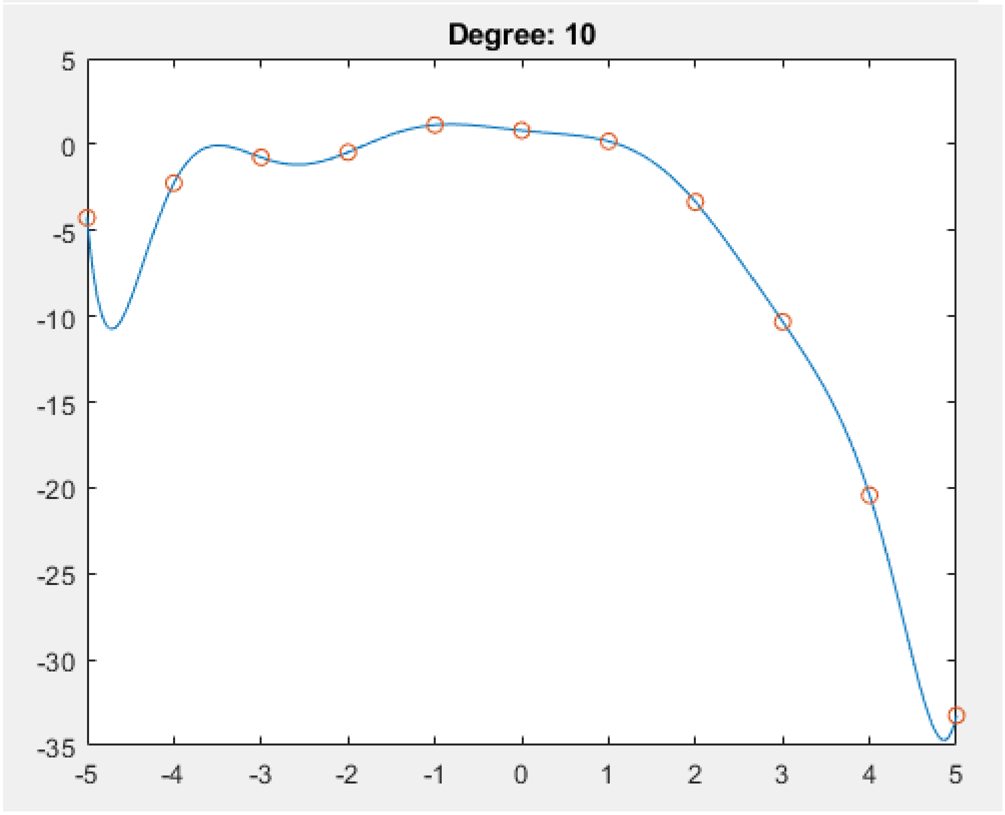

Degree: 4
5
0
-5
-10
-15
-20
-25
-30
-35
-5
-4
-3
-2
-1
0
1
2
3
4
5


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
-5
-4
-3
-2
-1
0
1
2
3
4
5

Conclusions

If we look at the table (1st screenshot), we can see that there is a rapid decrease in error up until the 3rd degree, after that it changes slowly. Condition number for a polynomial of degree 3 is also still reasonable so choosing the polynomial of degree 3 seems the most reasonable.
All polynomials of lower degree have a significant error residuum, 7.35 and more, while for the one of order 3 it is only 1.57, which is acceptable. If we look at the condition number, the higher it is, the lower the error is. However, the higher it is, the higher the chance for errors. Even though polynomial of degree 10 has error equal to 0, its condition number is abysmal: $\sim 9 \cdot 10^{14}$, that is why it was not chosen.

2. 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: x1(0) = –0.4, x2(0) = 0.5. Evaluate the solution using:
a) Runge-Kutta method of 4th order (RK4) and Adams PC (P5EC5E) – each method a few times, with different constant step-sizes until an „optimal" constant step size is found, i.e., when its decrease does not influence the 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.

Background

Runge-Kutta method

The family of Runge-Kutta methods can be defined in the following way:

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

where

$$k_1 = f\,(x_n, y_n)\,, \qquad (7.19b)$$
$$k_i = f(x_n + c_i h, y_n + h \cdot \sum_{j=1}^{i-1} a_{ij} k_j), \quad i = 2, 3, \ldots, m, \qquad (7.19c)$$

and also

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

All screenshots are taken from book Numerical Methods by Piotr Tatjewski unless stated otherwise

However, what we are most interested in, is Runge-Kutta method of order 4, also called classical. For that, we use the following set of equations:

$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{7.20a}$$

$$k_1 = f\left(x_n, y_n\right), \tag{7.20b}$$

$$k_2 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1), \tag{7.20c}$$

$$k_3 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2), \tag{7.20d}$$

$$k_4 = f\left(x_n + h, y_n + hk_3\right). \tag{7.20e}$$

where *h* is the step size that we can set manually. It should be chosen in a way that it's increase won't improve the solution significantly, but its lowering will worsen it a lot.

Adams Predictor-corrector (PC) method

The predictor corrector method is a mix of Adam's explicit and implicit method. It connects explicit method small number of arithmetic operations performed during one iteration with implicit method high order, small error constant and a large set of the absolute stability, making it much less prone to errors.

For us, the most interesting across Adams method will be P_kEC_kE. It can be summarised in those few equations:

$$\begin{aligned}
&\text{P:} && y_n^{[0]} = y_{n-1} + h\sum_{j=1}^{k}\beta_j f_{n-j},\\
&\text{E:} && f_n^{[0]} = f(x_n, y_n^{[0]}),\\
&\text{C:} && y_n = y_{n-1} + h\sum_{j=1}^{k}\beta_j^* f_{n-j} + h\beta_0^* f_n^{[0]},\\
&\text{E:} && f_n = f\left(x_n, y_n\right).
\end{aligned}$$

So the predictor part after modification to fit with the notation from the task, would look like this:

$$Predictor\colon\ x_n^{[0]} = x_{n-1} + h\sum_{j=1}^{k}\beta_j f_{n-j}$$

$$Corrector\colon x_n = x_{n-1} + h\sum_{j=1}^{k}\beta_j^* f_{n-j} + h\beta_0^* f_n^{[0]}$$

for *k*-step method we have again, just like in RK4 method step size h that should be manually adjusted, but also β and β^*. Those values we can simply retrieve from tables on the next page, by choosing appropriate k.

For Predictor:

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	$\frac{198721}{60480}$	$-\frac{447288}{60480}$	$\frac{705549}{60480}$	$-\frac{688256}{60480}$	$\frac{407139}{60480}$	$-\frac{134472}{60480}$	$\frac{19087}{60480}$

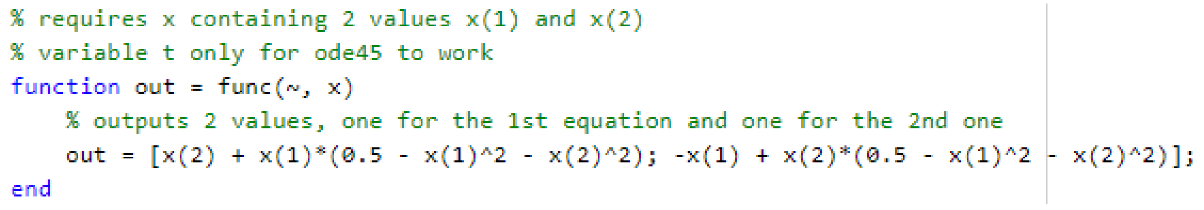
For Corrector:

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	$\frac{36799}{120960}$	$\frac{139849}{120960}$	$-\frac{121797}{120960}$	$\frac{123133}{120960}$	$-\frac{88547}{120960}$	$\frac{41499}{120960}$	$-\frac{11351}{120960}$	$\frac{1375}{120960}$

Because Adams PC is a multi-step method, it uses previous results to create new points on the graph. However that means that we have to start with k-1 data points. For that purpose we can for example use beforementioned Runga-Kutta method.

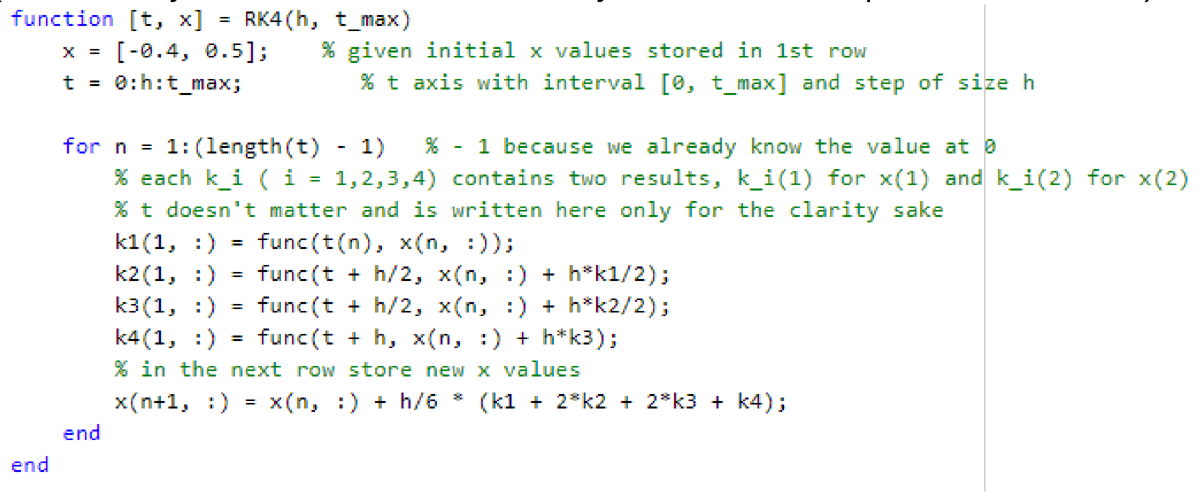
MATLAB implementation

Starting from one of the most trivial functions, this one is responsible for calculating values for x, as output we get a 2x1 vector containing value for x1 and x2, compatible with ode45.



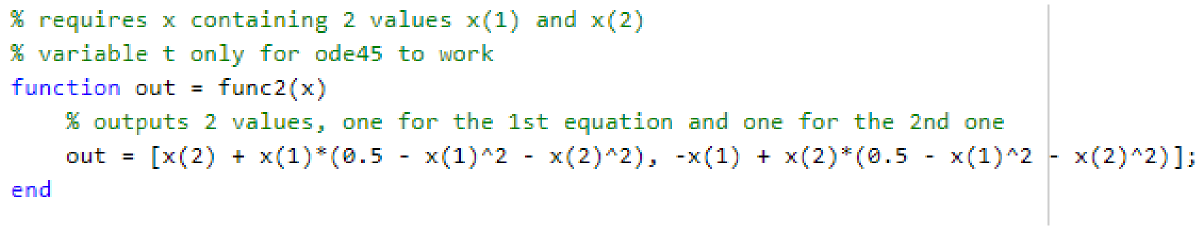
```
% requires x containing 2 values x(1) and x(2)
% variable t only for ode45 to work
function out = func(~, x)
    % outputs 2 values, one for the 1st equation and one for the 2nd one
    out = [x(2) + x(1)*(0.5 - x(1)^2 - x(2)^2); -x(1) + x(2)*(0.5 - x(1)^2 - x(2)^2)];
end
```

Runge-Kutta method didn't take up much space, as input there is required step size and upper limit of the axis (needed for Adams method), as output we get time axis and x axis filled with data. x_1 and x_2 values are stored in separate columns (contradictory to function func.m where they were stored in separate rows instead).



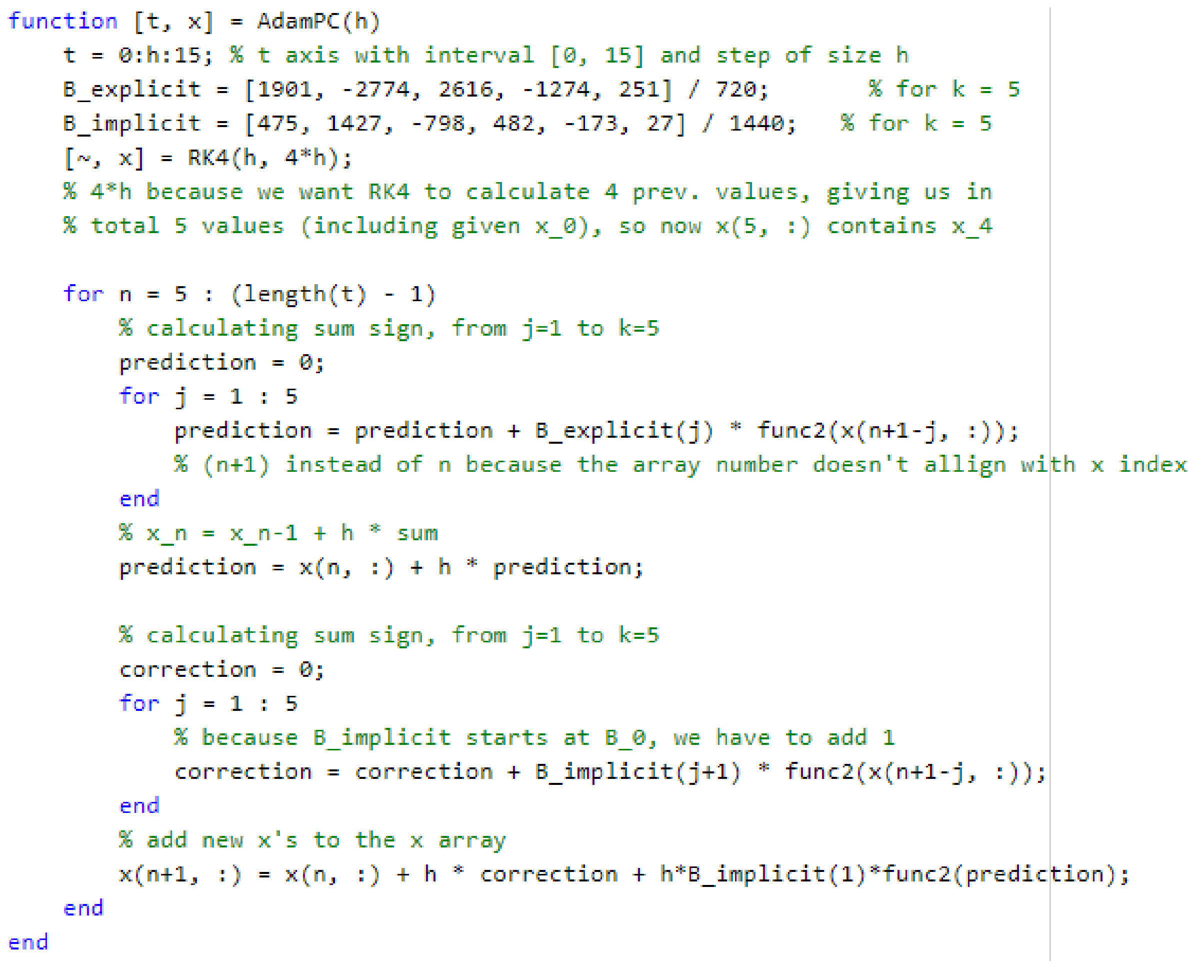
```
function [t, x] = RK4(h, t_max)
    x = [-0.4, 0.5];    % given initial x values stored in 1st row
    t = 0:h:t_max;         % t axis with interval [0, t_max] and step of size h

    for n = 1:(length(t) - 1)   % - 1 because we already know the value at 0
        % each k_i ( i = 1,2,3,4) contains two results, k_i(1) for x(1) and k_i(2) for x(2)
        % t doesn't matter and is written here only for the clarity sake
        k1(1, :) = func(t(n), x(n, :));
        k2(1, :) = func(t + h/2, x(n, :) + h*k1/2);
        k3(1, :) = func(t + h/2, x(n, :) + h*k2/2);
        k4(1, :) = func(t + h, x(n, :) + h*k3);
        % in the next row store new x values
        x(n+1, :) = x(n, :) + h/6 * (k1 + 2*k2 + 2*k3 + k4);
    end
end
```

For Adams PC method it was required to have func.m function output x's in separate columns and for that was created this very slightly altered version:



```
% requires x containing 2 values x(1) and x(2)
% variable t only for ode45 to work
function out = func2(x)
    % outputs 2 values, one for the 1st equation and one for the 2nd one
    out = [x(2) + x(1)*(0.5 - x(1)^2 - x(2)^2), -x(1) + x(2)*(0.5 - x(1)^2 - x(2)^2)];
end
```

Adams method is I believe sufficiently commented and just follows equations from the theoretical part of the report, it requires step size as input and outputs time and x:



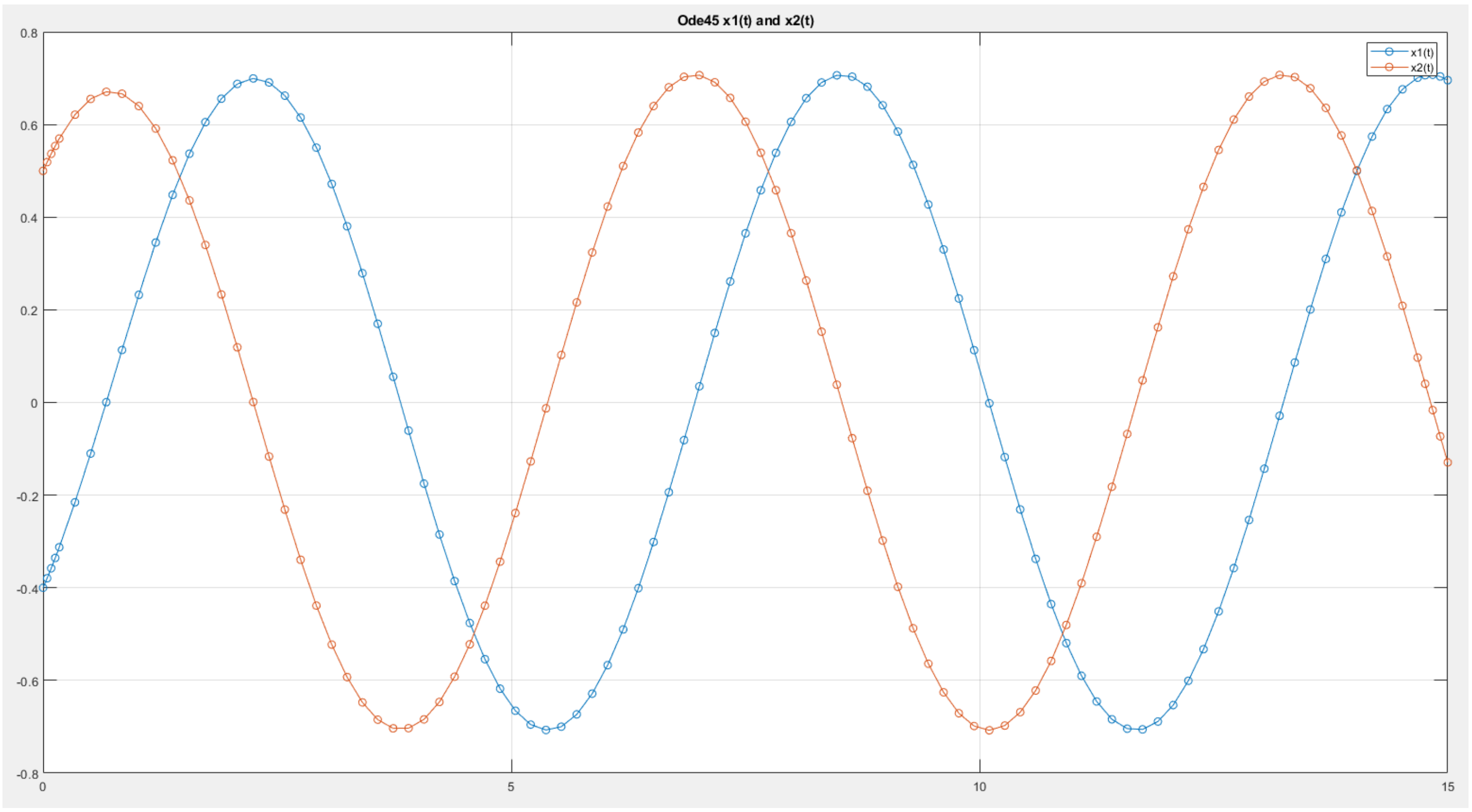
```
function [t, x] = AdamPC(h)
    t = 0:h:15; % t axis with interval [0, 15] and step of size h
    B_explicit = [1901, -2774, 2616, -1274, 251] / 720;        % for k = 5
    B_implicit = [475, 1427, -798, 482, -173, 27] / 1440;    % for k = 5
    [~, x] = RK4(h, 4*h);
    % 4*h because we want RK4 to calculate 4 prev. values, giving us in
    % total 5 values (including given x_0), so now x(5, :) contains x_4

    for n = 5 : (length(t) - 1)
        % calculating sum sign, from j=1 to k=5
        prediction = 0;
        for j = 1 : 5
            prediction = prediction + B_explicit(j) * func2(x(n+1-j, :));
            % (n+1) instead of n because the array number doesn't allign with x index
        end
        % x_n = x_n-1 + h * sum
        prediction = x(n, :) + h * prediction;

        % calculating sum sign, from j=1 to k=5
        correction = 0;
        for j = 1 : 5
            % because B_implicit starts at B_0, we have to add 1
            correction = correction + B_implicit(j+1) * func2(x(n+1-j, :));
        end
        % add new x's to the x array
        x(n+1, :) = x(n, :) + h * correction + h*B_implicit(1)*func2(prediction);
    end
end
```

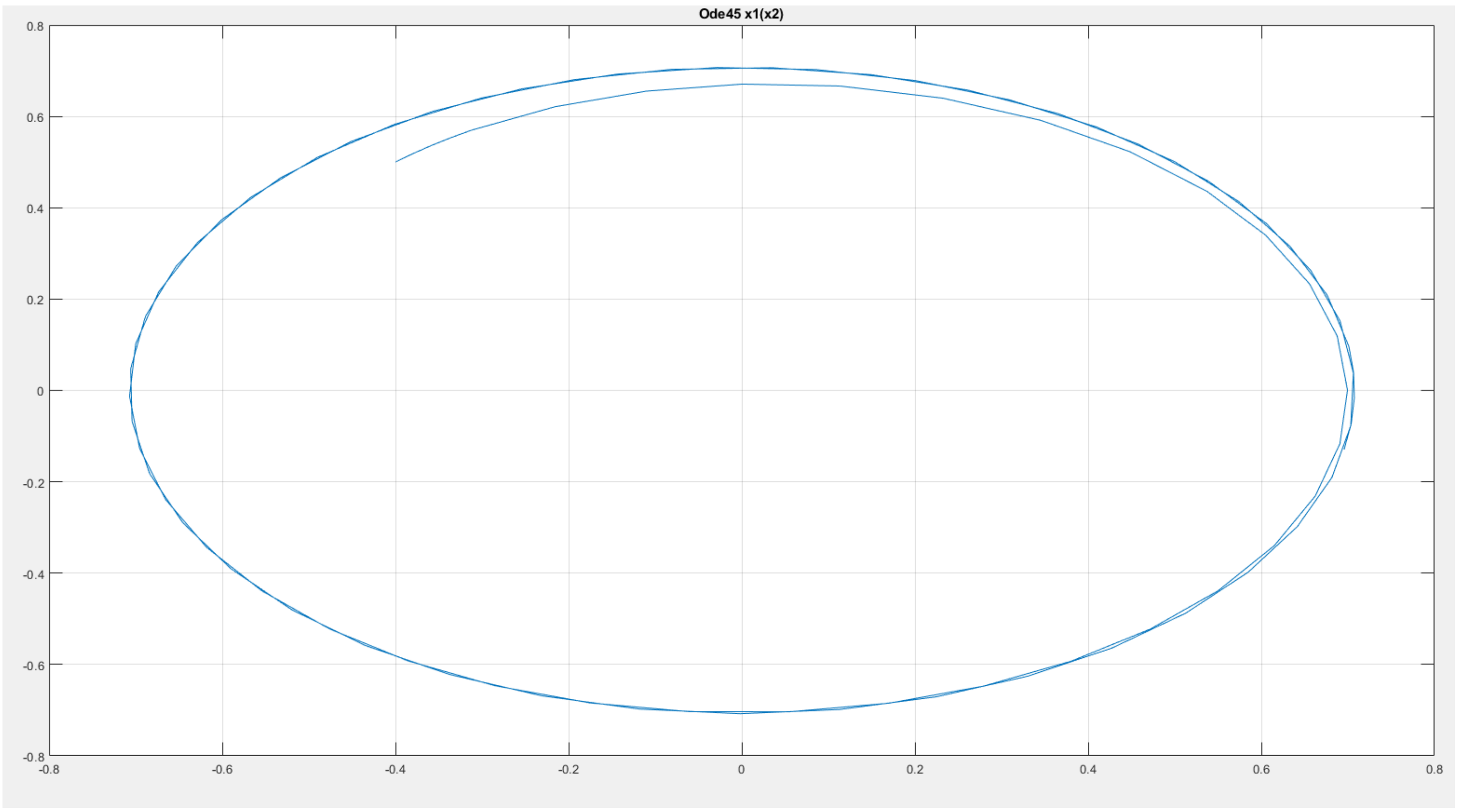
The last function is responsible for drawing is rather long so here I will only include the part of it that includes RK4 method, it mostly repeats itself for Adams PC method. It is also rather simple, just printing graphs with certain differences:

```
function draw_graph(type)
    if type == "ode_45"
        % x1(t) and x2(t) on the same graph
        figure;
        t_bounds = [0, 15];
        init_x = [-0.4, 0.5];
        ode45(@func, t_bounds, init_x);
        grid on;
        title("Ode45 x1(t) and x2(t)");
        legend("x1(t)", "x2(t)");
        % x1(x2)
        figure;
        [~, x] = ode45(@func, t_bounds, init_x);
        plot(x(:, 1), x(:, 2));
        grid on;
        title("Ode45 x1(x2)");

    elseif type == "RK4"
        % draws x1(t) with different steps
        figure;
        h = 1;
        while (h >= 0.0625)
        [t, x] = RK4(h, 15);
        plot(t, x(:, 1));
        grid on;
        hold on;
        h = h/2;
        end

        legend("Step: " + 1, "Step: " + 0.5, "Step: " + 0.25, "Step: " + 0.125, "Step: " + 0.0625);
        title("RK4: x1(t)");
        hold off;
        % draws x2(t) with different steps
        figure;
        h = 1;
        while (h >= 0.0625)
        [t, x] = RK4(h, 15);
        plot(t, x(:, 2));
        grid on;
        hold on;
        h = h/2;
        end
        legend("Step: 1", "Step: 0.5", "Step: 0.25", "Step: 0.125", "Step: 0.0625");
        title("RK4: x2(t)");
        hold off;
        % draw x1(t) and x2(t) versus time for 2 different steps
        figure;
        [t, x] = RK4(0.25, 15);
        plot(t, x(:, 1));
        hold on;
        plot(t, x(:, 2));
        [t, x] = RK4(0.5, 15);
        plot(t, x(:, 1));
        plot(t, x(:, 2));
        grid on;
        title("RK4: x1(t) and x2(t)");
        legend("x1, step: 0.25", "x2, step: 0.25", "x1, step: 0.5", "x2, step: 0.5");
        hold off;
        % draw x1(x2) for 2 different steps
        figure;
        [~, x] = RK4(0.25, 15);
        plot(x(:, 1), x(:, 2));
        hold on;
        [~, x] = RK4(0.5, 15);
        plot(x(:, 1), x(:, 2));
        grid on;
        title("RK4: x1(x2)");
        legend("x1, step: 0.25", "x2, step: 0.5");
        hold off;
```

Test graph to compare to made with ode45:

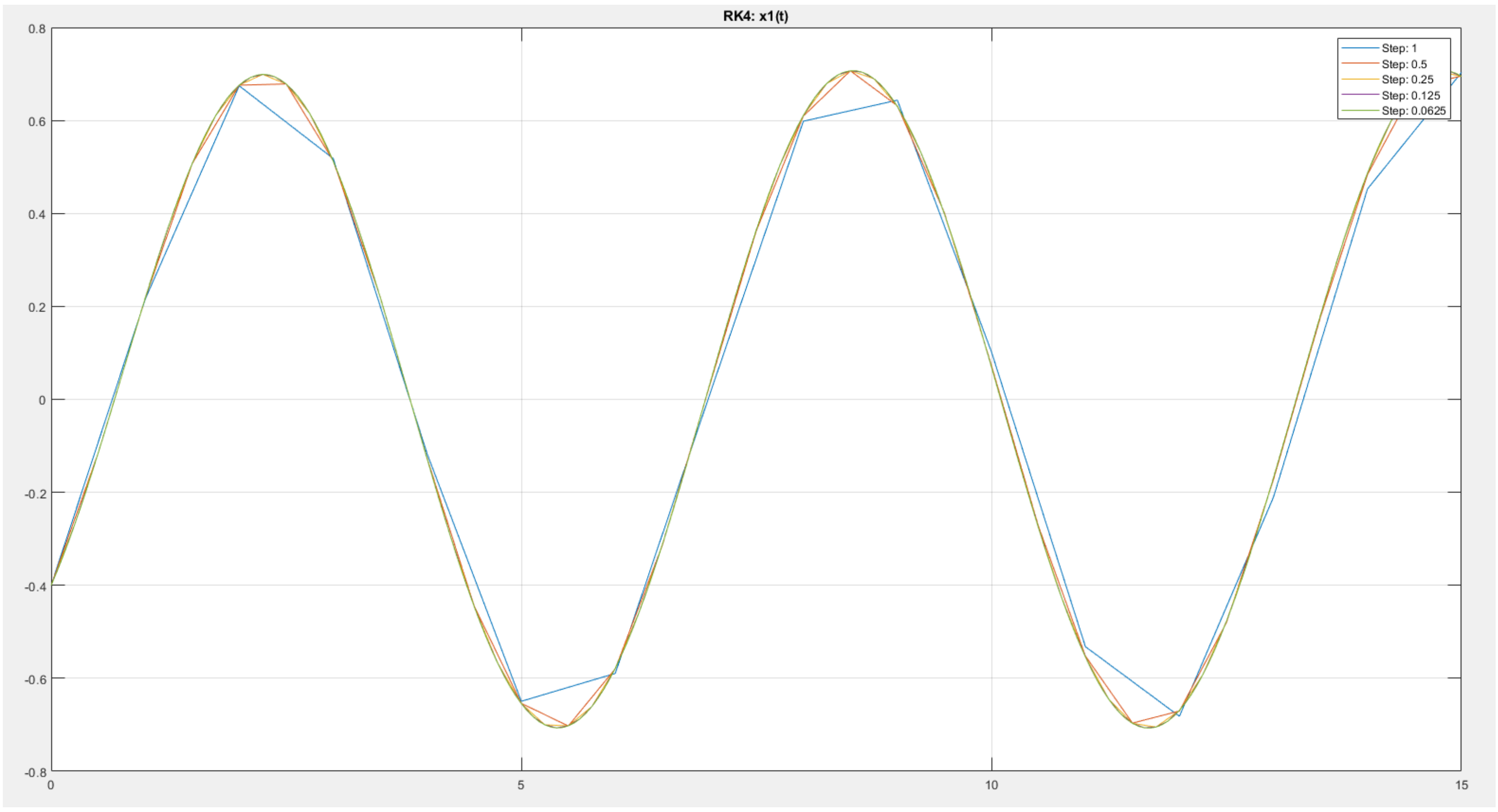
```
>> draw_graph("ode_45")
```



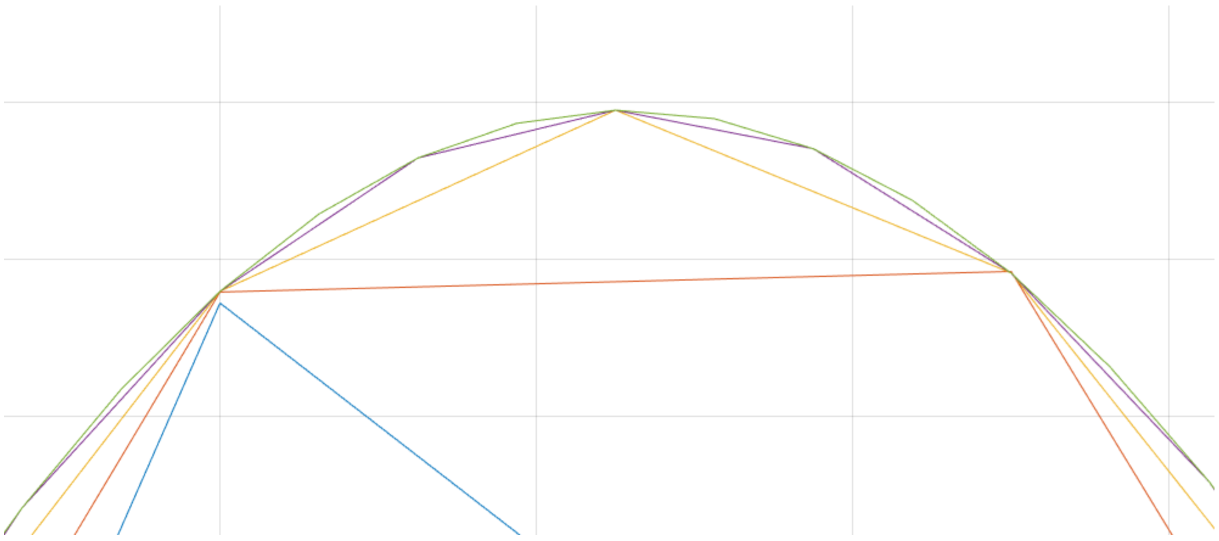


Runge-Kutte graphs

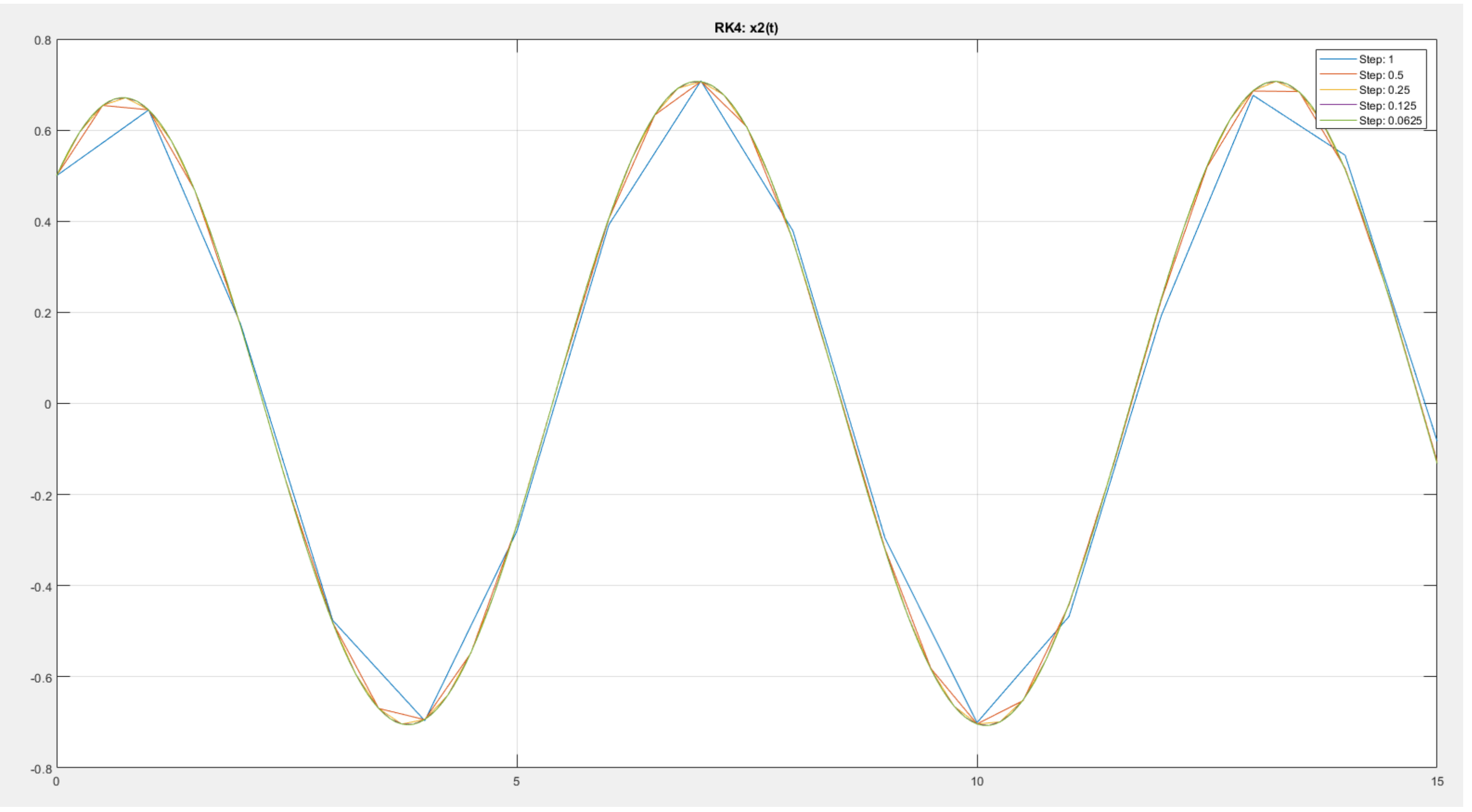
```
>> draw_graph("RK4")
```



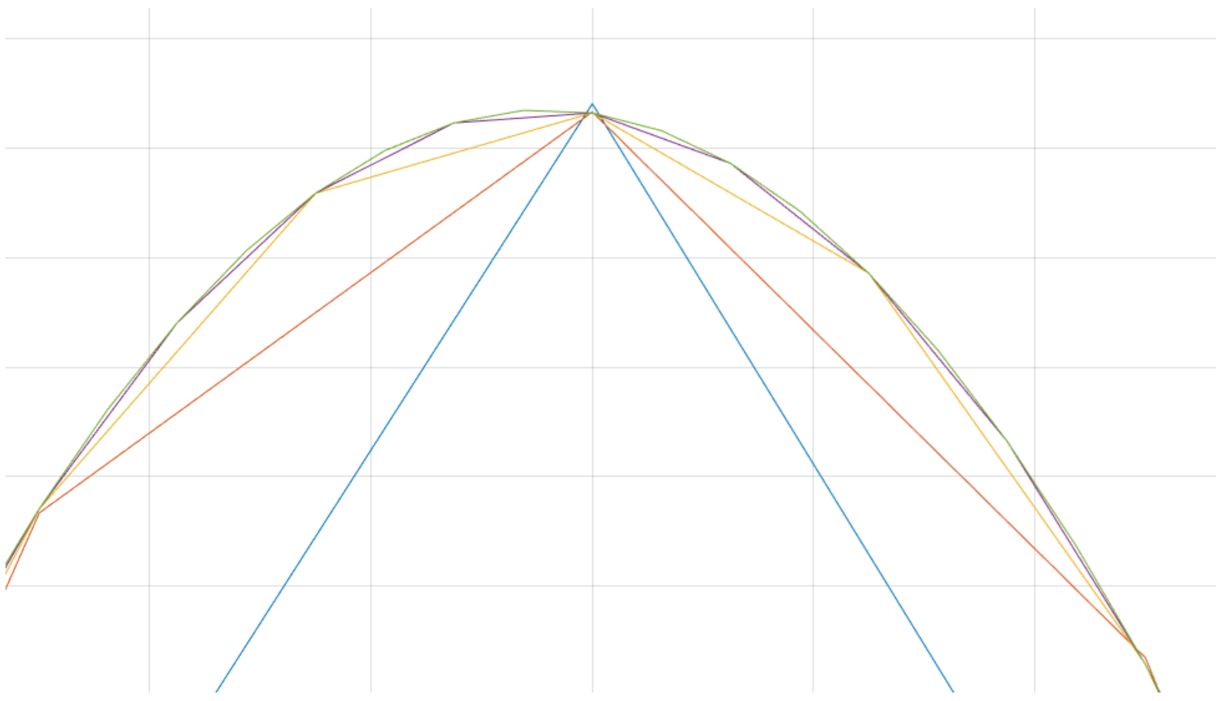
Zoomed in:



Based on my observation, I have decided that the most optimal line is the yellow one, representing step 0.25. Step size of 0.5 and 1 gave much bigger errors, and step size of 0.125 didn’t improve the result significantly.

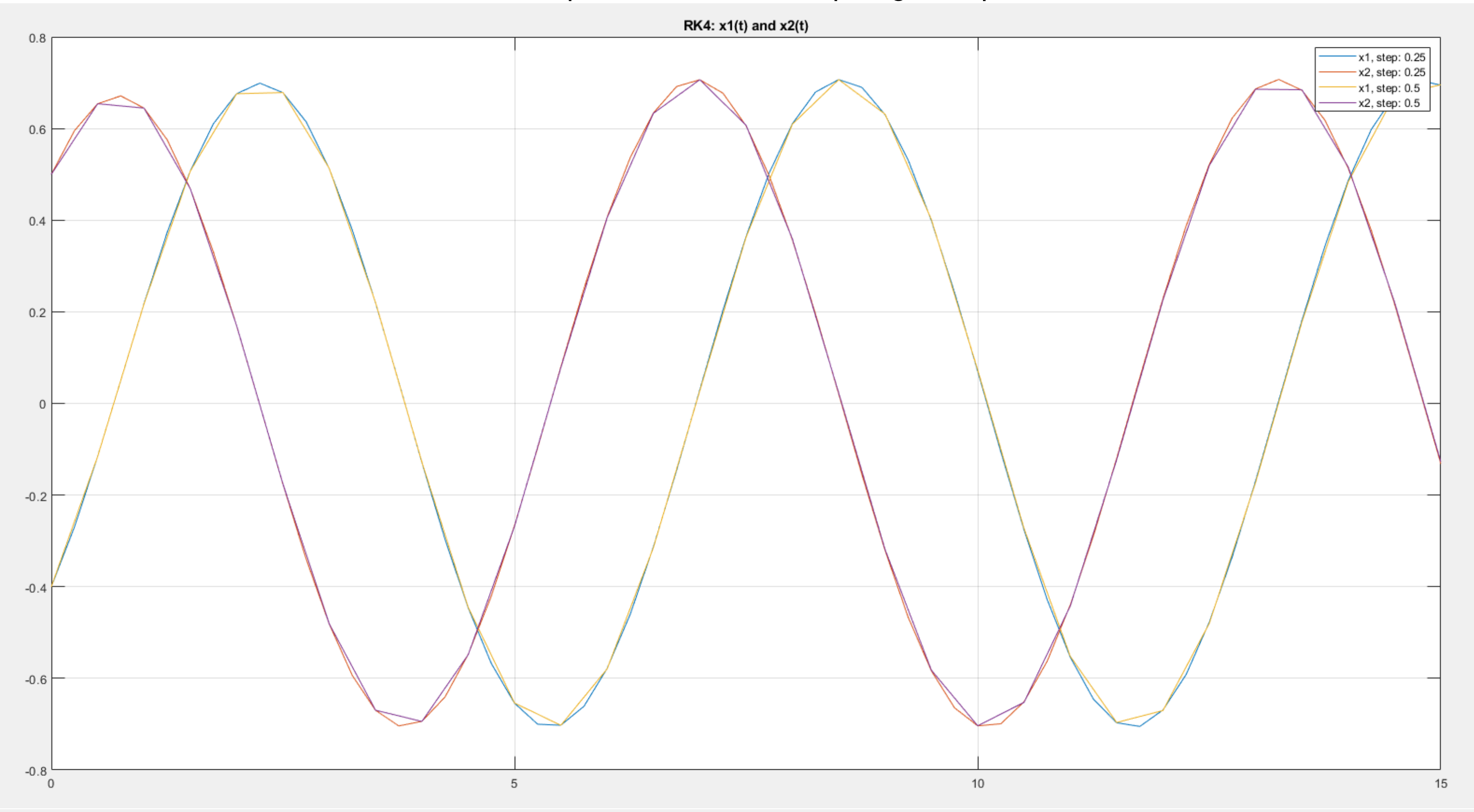


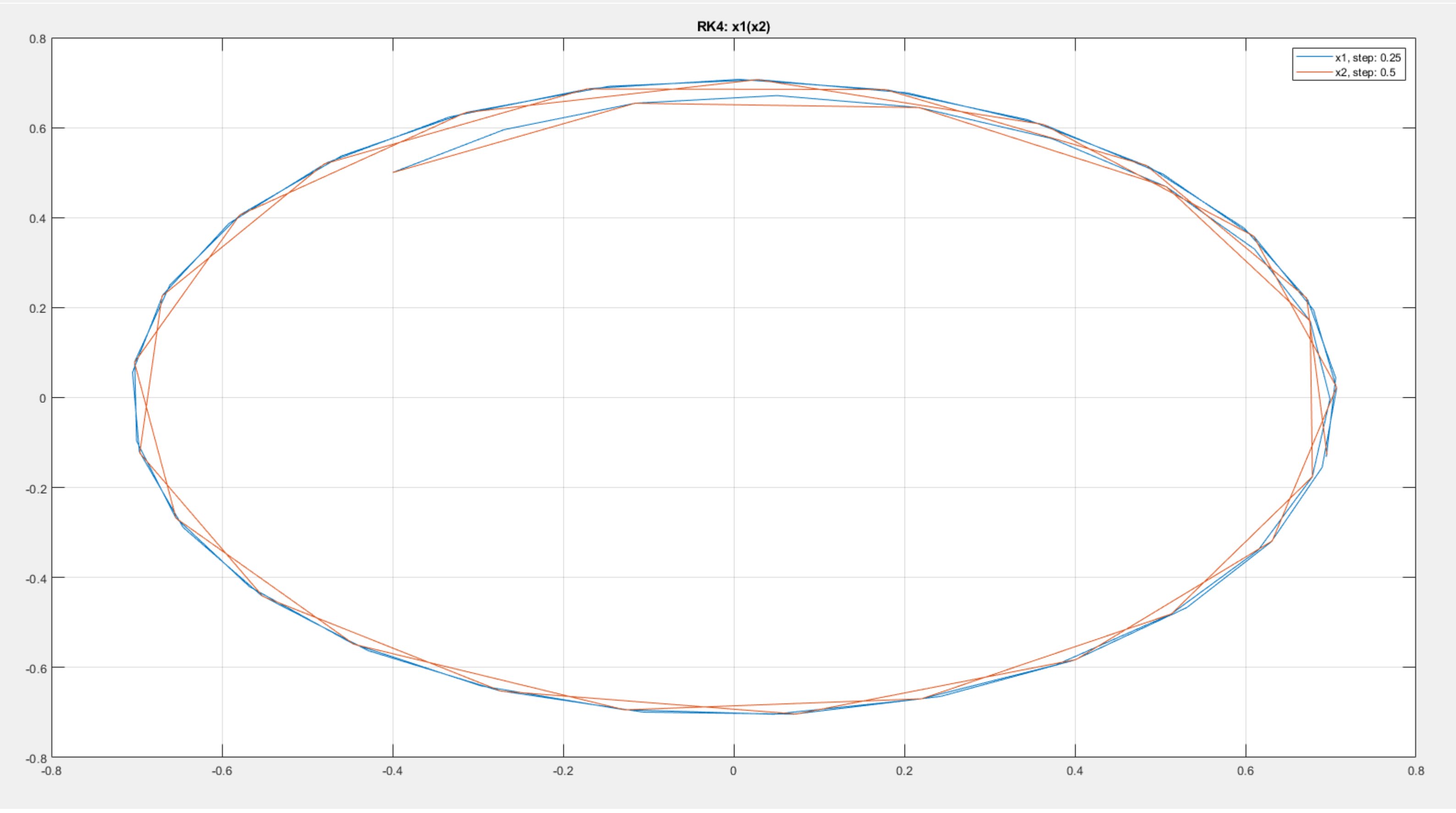
Zoomed in:



Once again there is a significant change between 0.5 and 0.25, but not that big between 0.125 and 0.25. so x_2 graph also seems to uphold the 0.25 step size.

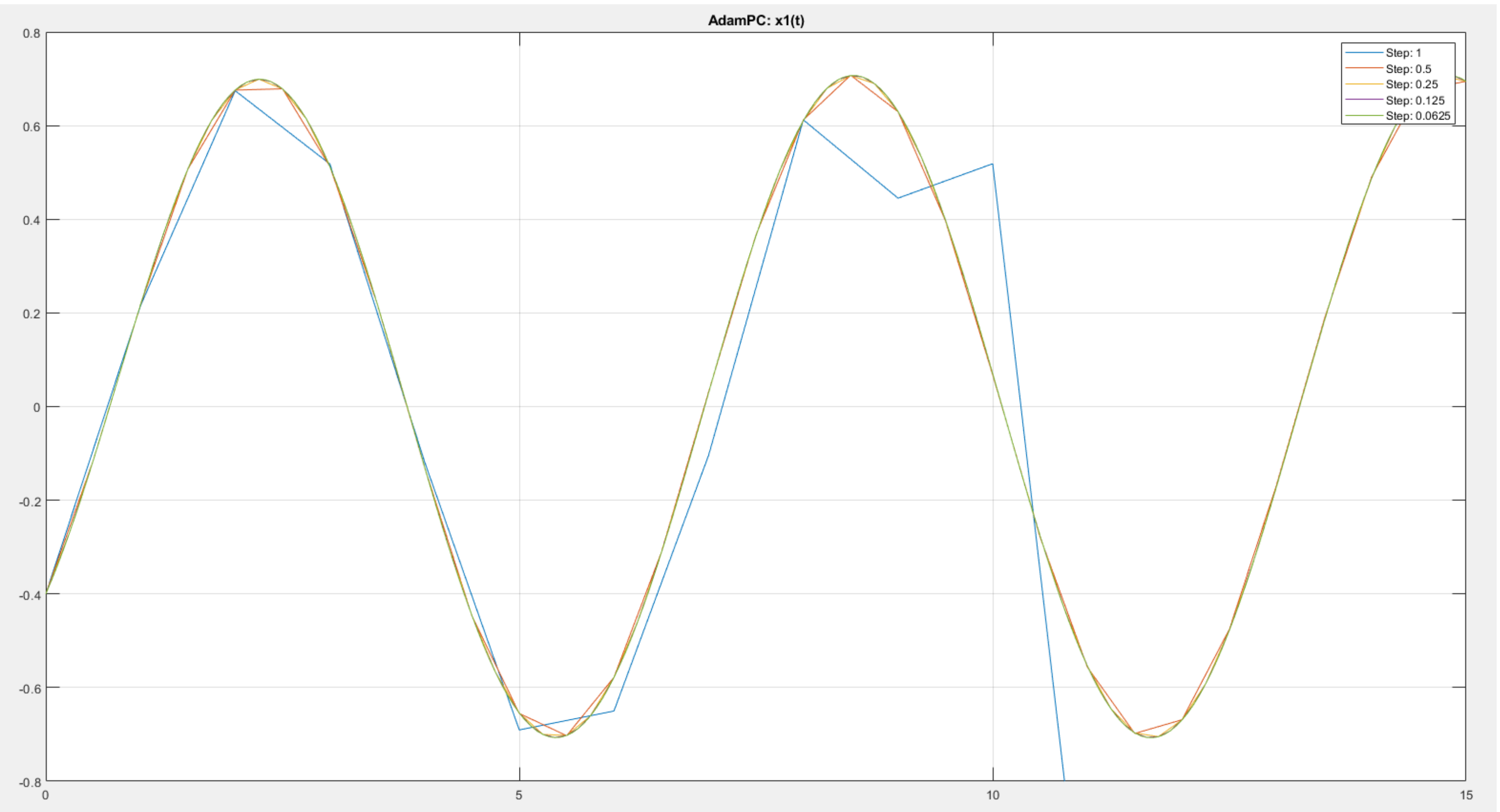
Here we can see both x_1 and x_2 on the same graph, this time with only their optimal constant step size and a one step larger step size.

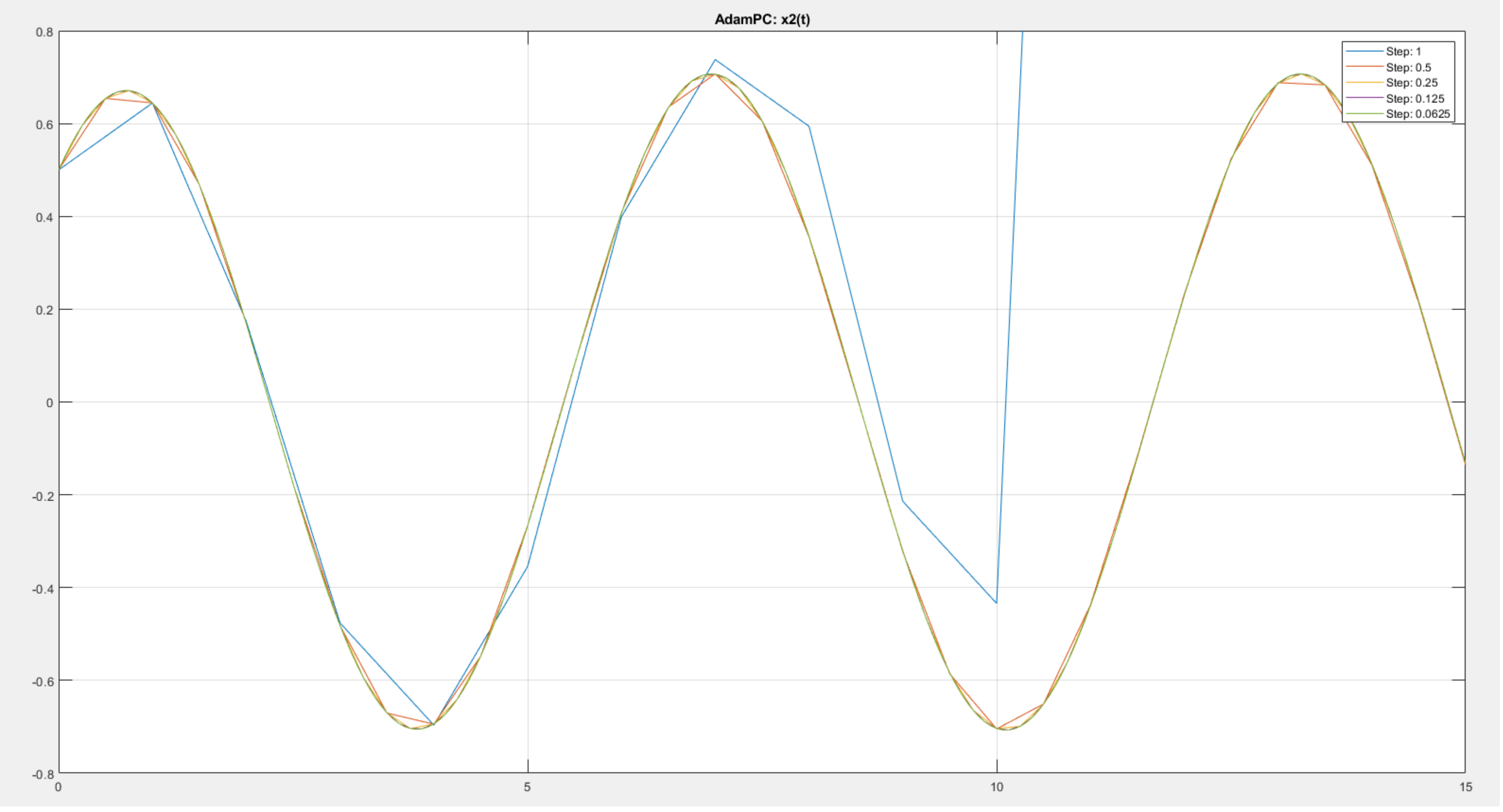




Adams P_5EC_5E graphs

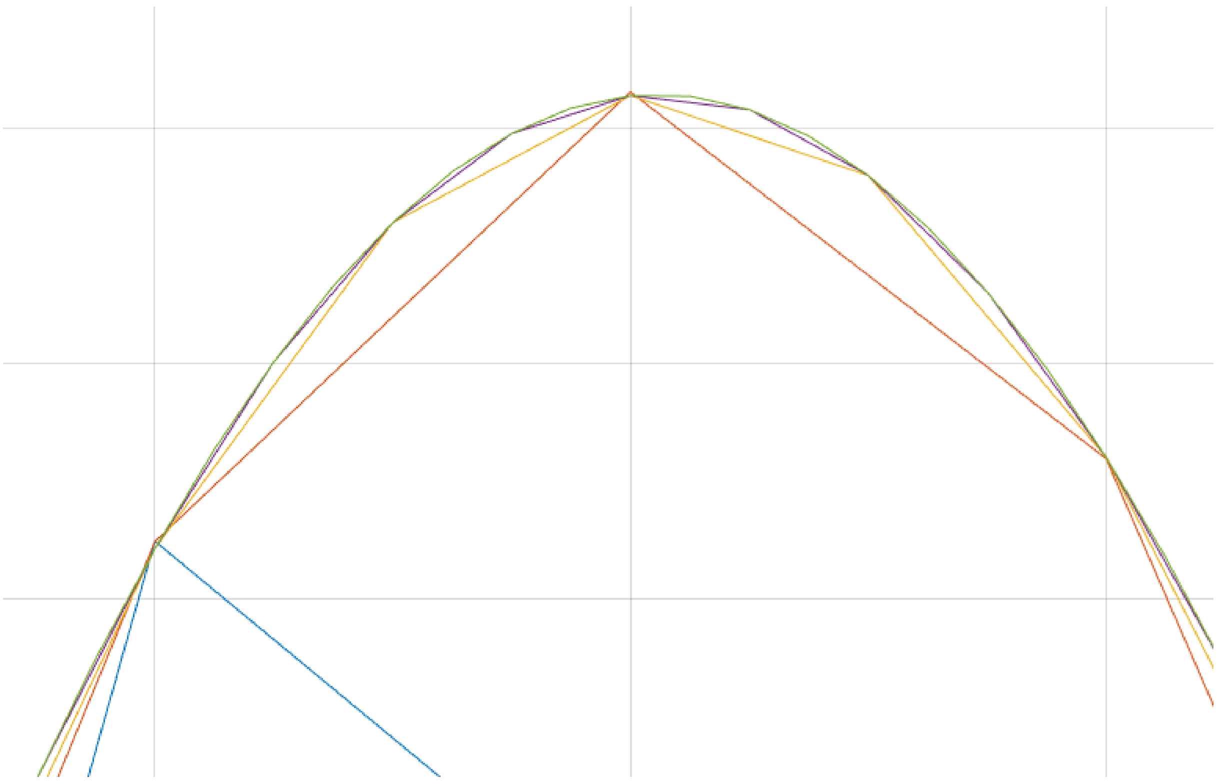
```
>> draw_graph("AdamPC")
```



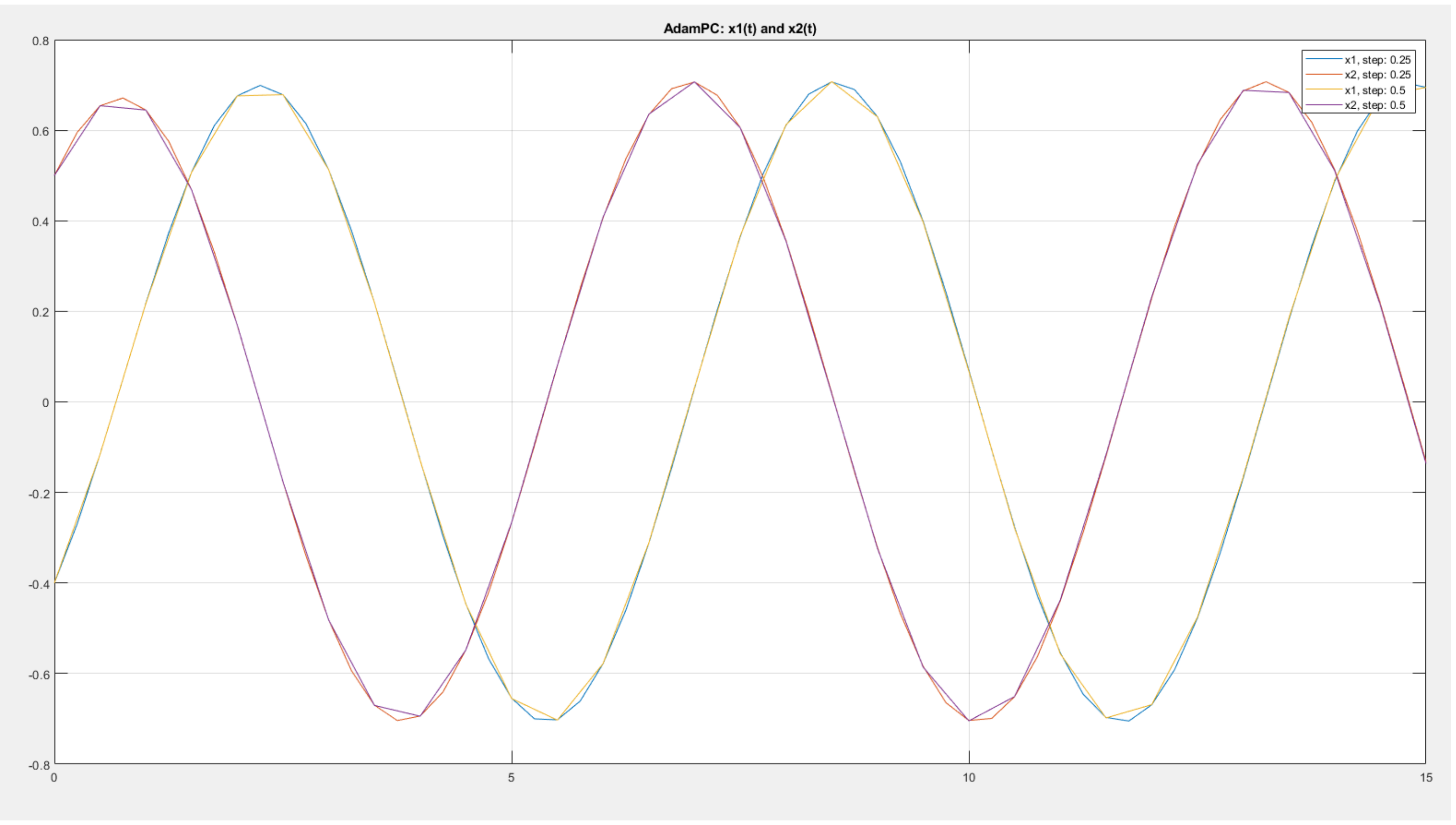


Surprisingly, for Adams PC method, when step size was equal to 1, the method failed and started heading towards (+/-) infinity. However if we ignore this fact, results of Adams method and Runge-Kutta method yielded similar resultats.

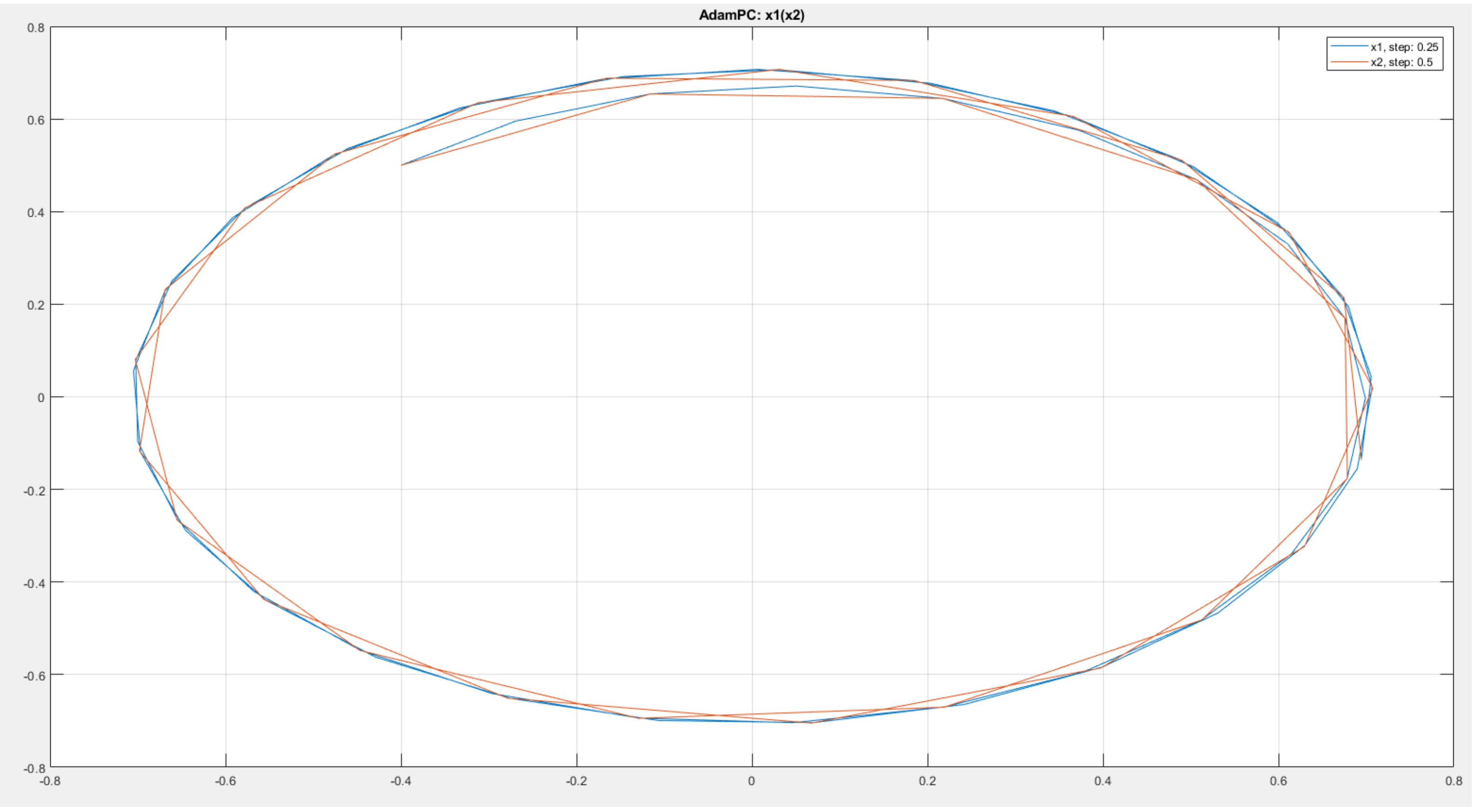
Here we can also see, just like in Runge-Kutta method, yellow lane aka step size 0.25 is the best choice.



Graph containing both x_1 and x_2, this time made with Adams method.


AdamPC: x1(t) and x2(t)
x1, step: 0.25
x2, step: 0.25
x1, step: 0.5
x2, step: 0.5
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
5
10
15

$x_1(x_2)$


AdamPC: x1(x2)
x1, step: 0.25
x2, step: 0.5
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

Conclusions

When we compare graphs created by the GK4 and Adams method, we can see they closely resemble output from ode45 meaning that the program works correctly. Step size 0.25 was chosen as it seemed the most reasonable compromise.

Code:

Task 1

poly_approximation.m

```
function [cond_num, err, a_vector, x, y] = poly_approximation(degree)
    A = zeros(11, degree+1);    % create array A, +1 because degree 0 of
polynomial is also allowed
    x = [-5; -4; -3; -2; -1; 0; 1; 2; 3; 4; 5];
    y = [-4.2606; -2.26804; -0.7699; -0.4666; 1.1236; 0.8029; 0.1697; -3.3483; -
10.3280; -20.4417; -33.2458];

    for i = 1:11    % 11 because we have 11 experimental measurements
        for j = 1:(degree+1)
            A(i, j) = x(i)^(j-1);   % (j-1) because we start every row with x^0
and end with x^degree, which is also x^(j+1)
        end
    end
    cond_num = (cond(A))^2;    % calculate condition number for A^T * A
    [Q, R] = external_qrmgs(A); % calculate Q and R
    a_vector = R \ (Q' * y);    % calculate resulting coefficients a
    a_vector = flip(a_vector);    % reorder to descending order so that it is
accepted by matlab's polyval function
    calculated_values = polyval(a_vector, x);  % calculates values at set x's
    err = norm(calculated_values - y);    % calculates error residuum
end
```

external_qrmgs.m

```
% it's the same function as the one in 1st project
function [Q, R] = external_qrmgs(D)
    %QR (thin) factorization using modified Gram-Schmidt algorithm
    %for full rank real-valued and complex-valued matrices
    [m, n] = size(D);
    d = zeros(1, n);
    Q = zeros(m, n);
    R = zeros(n, n);
    %factorization with orthogonal (not orthonormal) columns of Q:
    for i = 1 : n
        Q(:, i) = D(:, i);
        R(i, i) = 1;
        d(i) = Q(:, i)' * Q(:, i);
        for j = (i + 1) : n
            R(i, j) = (Q(:, i)' * D(:, j)) / d(i);
            D(:, j) = D(:, j) - R(i, j) * Q(:, i);
        end
    end
    %column normalization (columns of Q orthonormal):
    for i = 1 : n
        dd = norm(Q(:, i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:n) = R(i, i:n) * dd;
    end
end

solve_and_print_t1.m
function solve_and_print_t1()
    fprintf("Degree:\t Condition number:\t Error residuum:\n");
    for i = 0:10    % loop for up to polynomial of degree 10
        % print text
        [cond_num, err, a_vector, x, y] = poly_approximation(i);  % calculate
polynomial of degree i and get condition number and error
        fprintf("%d\t\t %e\t\t %f\n", i, cond_num, err);

        %plot graph
        x_axis = -5:.01:5; % step between each calculated f(x)
        figure; % needed for multiple graphs
        grid on;
        plot(x_axis, polyval(a_vector, x_axis));
        title("Degree: " + i ); % Puts a title with the degree of polynomial
        hold on;
        plot(x, y, 'o');
    end
end
```

Task 2a

func.m

```
% requires x containing 2 values x(1) and x(2)
% variable t only for ode45 to work
function out = func(~, x)
    % outputs 2 values, one for the 1st equation and one for the 2nd one
    out = [x(2) + x(1)*(0.5 - x(1)^2 - x(2)^2); -x(1) + x(2)*(0.5 - x(1)^2 -
x(2)^2)];
end
```

func2.m

```
% requires x containing 2 values x(1) and x(2)
% variable t only for ode45 to work
function out = func2(x)
    % outputs 2 values, one for the 1st equation and one for the 2nd one
    out = [x(2) + x(1)*(0.5 - x(1)^2 - x(2)^2), -x(1) + x(2)*(0.5 - x(1)^2 -
x(2)^2)];
end
```

RK4.m

```
function [t, x] = RK4(h, t_max)
    x = [-0.4, 0.5];    % given initial x values stored in 1st row
    t = 0:h:t_max;         % t axis with interval [0, t_max] and step of size h

    for n = 1:(length(t) - 1)   % - 1 because we already know the value at 0
        % each k_i ( i = 1,2,3,4) contains two results, k_i(1) for x(1) and k_i(2)
for x(2)
        % t doesn't matter and is written here only for the clarity sake
        k1(1, :) = func(t(n), x(n, :));
        k2(1, :) = func(t + h/2, x(n, :) + h*k1/2);
        k3(1, :) = func(t + h/2, x(n, :) + h*k2/2);
        k4(1, :) = func(t + h, x(n, :) + h*k3);
        % in the next row store new x values
        x(n+1, :) = x(n, :) + h/6 * (k1 + 2*k2 + 2*k3 + k4);
    end
end
```

AdamPC.m

```
function [t, x] = AdamPC(h)
    t = 0:h:15; % t axis with interval [0, 15] and step of size h
    B_explicit = [1901, -2774, 2616, -1274, 251] / 720;       % for k = 5
    B_implicit = [475, 1427, -798, 482, -173, 27] / 1440;   % for k = 5
    [~, x] = RK4(h, 4*h);
    % 4*h because we want RK4 to calculate 4 prev. values, giving us in
    % total 5 values (including given x_0), so now x(5, :) contains x_4

    for n = 5 : (length(t) - 1)
        % calculating sum sign, from j=1 to k=5
        prediction = 0;
        for j = 1 : 5
            prediction = prediction + B_explicit(j) * func2(x(n+1-j, :));
            % (n+1) instead of n because the array number doesn't allign with x
index
        end
        % x_n = x_n-1 + h * sum
        prediction = x(n, :) + h * prediction;

        % calculating sum sign, from j=1 to k=5
        correction = 0;
        for j = 1 : 5
            % because B_implicit starts at B_0, we have to add 1
            correction = correction + B_implicit(j+1) * func2(x(n+1-j, :));
        end
        % add new x's to the x array
        x(n+1, :) = x(n, :) + h * correction + h*B_implicit(1)*func2(prediction);
    end
end
```

draw_graph.m

```
function draw_graph(type)
    if type == "ode_45"
        % x1(t) and x2(t) on the same graph
        figure;
        t_bounds = [0, 15];
        init_x = [-0.4, 0.5];
        ode45(@func, t_bounds, init_x);
        grid on;
        title("Ode45 x1(t) and x2(t)");
        legend("x1(t)", "x2(t)");
        % x1(x2)
        figure;
        [~, x] = ode45(@func, t_bounds, init_x);
        plot(x(:, 1), x(:, 2));
        grid on;
        title("Ode45 x1(x2)");

    elseif type == "RK4"
        % draws x1(t) with different steps
        figure;
        h = 1;
        while (h >= 0.0625)
        [t, x] = RK4(h, 15);
        plot(t, x(:, 1));
        grid on;
        hold on;
        h = h/2;
        end
        legend("Step: " + 1, "Step: " + 0.5, "Step: " + 0.25, "Step: " + 0.125,
"Step: " + 0.0625);
        title("RK4: x1(t)");
        hold off;
        % draws x2(t) with different steps
        figure;
        h = 1;
        while (h >= 0.0625)
        [t, x] = RK4(h, 15);
        plot(t, x(:, 2));
        grid on;
        hold on;
        h = h/2;
        end
        legend("Step: 1", "Step: 0.5", "Step: 0.25", "Step: 0.125", "Step:
0.0625");
        title("RK4: x2(t)");
        hold off;
        % draw x1(t) and x2(t) versus time for 2 different steps
        figure;
        [t, x] = RK4(0.25, 15);
        plot(t, x(:, 1));
        hold on;
        plot(t, x(:, 2));
        [t, x] = RK4(0.5, 15);
        plot(t, x(:, 1));
        plot(t, x(:, 2));
        grid on;
        title("RK4: x1(t) and x2(t)");
        legend("x1, step: 0.25", "x2, step: 0.25", "x1, step: 0.5", "x2, step:
0.5");
        hold off;
        % draw x1(x2) for 2 different steps
        figure;
        [~, x] = RK4(0.25, 15);
        plot(x(:, 1), x(:, 2));
        hold on;
        [~, x] = RK4(0.5, 15);
        plot(x(:, 1), x(:, 2));
        grid on;
        title("RK4: x1(x2)");
        legend("x1, step: 0.25", "x2, step: 0.5");
        hold off;

    elseif type == "AdamPC"
        % draws x1(t) with different steps
        figure;
        h = 1;
        while (h >= 0.0625)
        [t, x] = AdamPC(h);
        plot(t, x(:, 1));
        grid on;
        hold on;
        ylim([-0.8 0.8]);
        h = h/2;
        end
        legend("Step: " + 1, "Step: " + 0.5, "Step: " + 0.25, "Step: " + 0.125,
"Step: " + 0.0625);
        title("AdamPC: x1(t)");
        hold off;
        % draws x2(t) with different steps
        figure;
        h = 1;
        while (h >= 0.0625)
        [t, x] = AdamPC(h);
        plot(t, x(:, 2));
        grid on;
        hold on;
        ylim([-0.8 0.8]);
        h = h/2;
        end
        legend("Step: 1", "Step: 0.5", "Step: 0.25", "Step: 0.125", "Step:
0.0625");
        title("AdamPC: x2(t)");
        hold off;
        % draw x1(t) and x2(t) versus time for 2 different steps
        figure;
        [t, x] = AdamPC(0.25);
        plot(t, x(:, 1));
        hold on;
        plot(t, x(:, 2));
        [t, x] = AdamPC(0.5);
        plot(t, x(:, 1));
        plot(t, x(:, 2));
        grid on;
        title("AdamPC: x1(t) and x2(t)");
        legend("x1, step: 0.25", "x2, step: 0.25", "x1, step: 0.5", "x2, step:
0.5");
        hold off;
        % draw x1(x2) for 2 different steps
        figure;
        [~, x] = AdamPC(0.25);
        plot(x(:, 1), x(:, 2));
        hold on;
        [~, x] = AdamPC(0.5);
        plot(x(:, 1), x(:, 2));
        grid on;
        title("AdamPC: x1(x2)");
        legend("x1, step: 0.25", "x2, step: 0.5");
        hold off;
    else
        error("Wrong type of graph chosen!");
    end
end
```