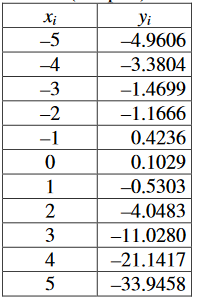
**Author: Damian Bukowski**

**Analysis of finding polynomials and solving ODE**

# Task 1

In this task We are asked to determine a polynomial function f(x), which would best fit the following experimental data:



We are supposed to use the least-squares approximation. The gist of the method is that we find the error of the measurements and plot the graph that will cross each point or cross an area within the error. We test polynomial of varying degrees, up to 10. The error is calculated based on the 2-norm of matrix A, which is filled with subsequent powers of xi (for x1 -> 1, -5, 25, -125; for x2 -> 1, -4, 16 etc.). We also calculate the condition number of Gram’s matrix. Condition number is a value that describes how much the function is sensitive to changes of x. In our case that would mean that the higher the condition number, the more the function is sensitive to errors. We calculate the Gram’s Matrix using the following formula:

Having calculated the Gram’s Matrix, We can calculate Its condition number using built-in function cond() from Matlab.

|  |  |  |
| --- | --- | --- |
| Degree of the polynomial | Condition number of Gram’s Matrix | Solution error |
| 0 | 1 | 0 |
| 1 | 10 | 9.0094 |
| 2 | 408.7769 | 3.0205 |
| 3 | 8.5584e+03 | 0.8099 |
| 4 | 3.1798e+05 | 0.8938 |
| 5 | 7.4675e+06 | 0.7103 |
| 6 | 2.8316e+08 | 0.7169 |
| 7 | 7.6462e+09 | 0.7047 |
| 8 | 3.3055e+11 | 0.6951 |
| 9 | 1.5167e+13 | 0.5148 |
| 10 | 9.2930e+14 | 4.3683e-12 |

As can be seen from the above results, the error is very high at the beginning, but then lowers down to about 0.5, with slight fluctuations here and there. When it reaches degree 10, the error is very small, but the condition number of Gram’s Matrix is huge. Because of that We can’t really assess the accuracy of the result, since even a very small error will cause tons of noise on the graph. Because of that I believe that the most accurate representation of the results is for the polynomial of degree 3. Worth noting is the fact that plots of degree 3 and 4 are almost identical, as can be seen here:

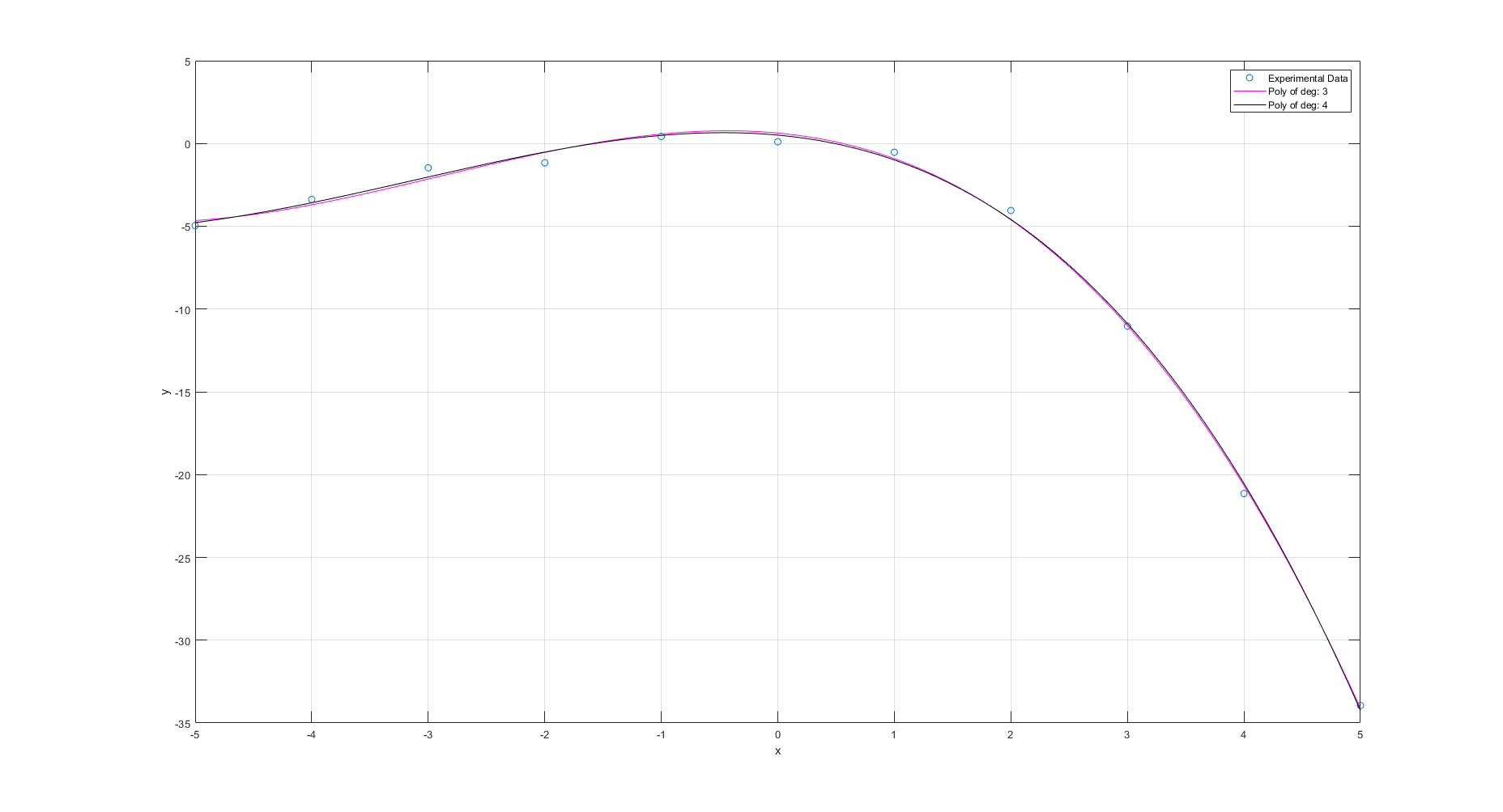


Figure 1 Poly of degree 3 and 4 with experimental data

The reason that I chose degree 3 to be best fitting is that it has smaller condition number and error.

The calculated coefficients of the polynomial f(x) = y are:

Degree of polynomial: 0

-7.3768

Degree of polynomial: 1

-2.2851 -7.3768

Degree of polynomial: 2

-0.8007 -2.2851 0.6303

Degree of polynomial: 3

-0.0919 -0.8007 -0.6494 0.6303

Degree of polynomial: 4

-0.0017 -0.0919 -0.7576 -0.6494 0.5062

Degree of polynomial: 5

0.0022 -0.0017 -0.1621 -0.7576 -0.2269 0.5062

Degree of polynomial: 6

0.0003 0.0022 -0.0142 -0.1621 -0.6445 -0.2269 0.3620

Degree of polynomial: 7

-0.0001 0.0003 0.0071 -0.0142 -0.2199 -0.6445 -0.0623 0.3620

Degree of polynomial: 8

0.0000 -0.0001 -0.0007 0.0071 0.0006 -0.2199 -0.7097 -0.0623 0.4048

Degree of polynomial: 9

-0.0001 0.0000 0.0032 -0.0007 -0.0467 0.0006 0.0912 -0.7097 -0.5245 0.4048

Degree of polynomial: 10

0.0001 -0.0001 -0.0033 0.0032 0.0576 -0.0467 -0.3986 0.0912 0.1880 -0.5245 0.1029

Chart

Description automatically generated

Figure 2 All polys up to degree 10 with zoomed in graph on the right

# Task 2a: Runge-Kutta 4th order Method

In this task we are given two ODEs and are asked to approximate their values as well as values of x1 and x2, based on initial points. We use two methods: single-step Runge-Kutta and multi-step Adam’s P5EC5E. ODEs for my algorithm are:

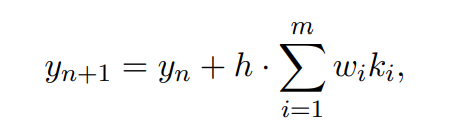
Given a fixed step-size, a single-step method can be described as:



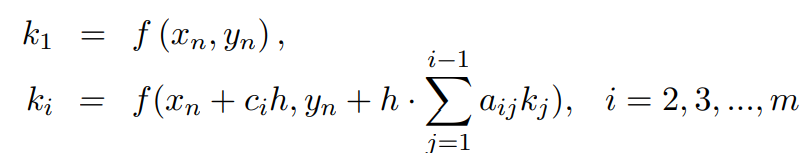
Where



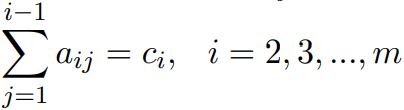
In our case, the Runge-Kutta algorithm is defined as:



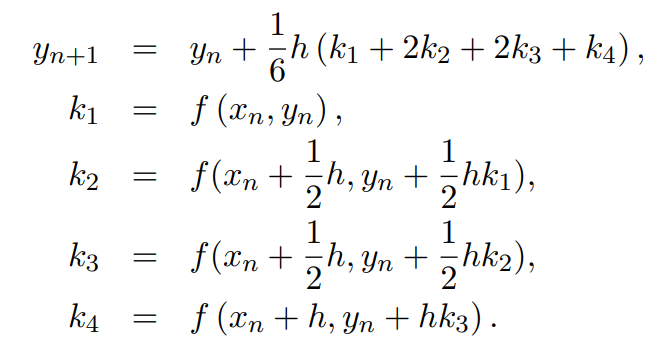
Where



Also



We use the classical version of the RK, called the Runge-Kutta method of order for, or RK4 for simplicity. The steps for each iteration of the RK4 are:



In MATLAB, the first line must be the last to correctly run the *for* loop. We will be comparing our results to the results from built-in MATLAB function *ode45*.

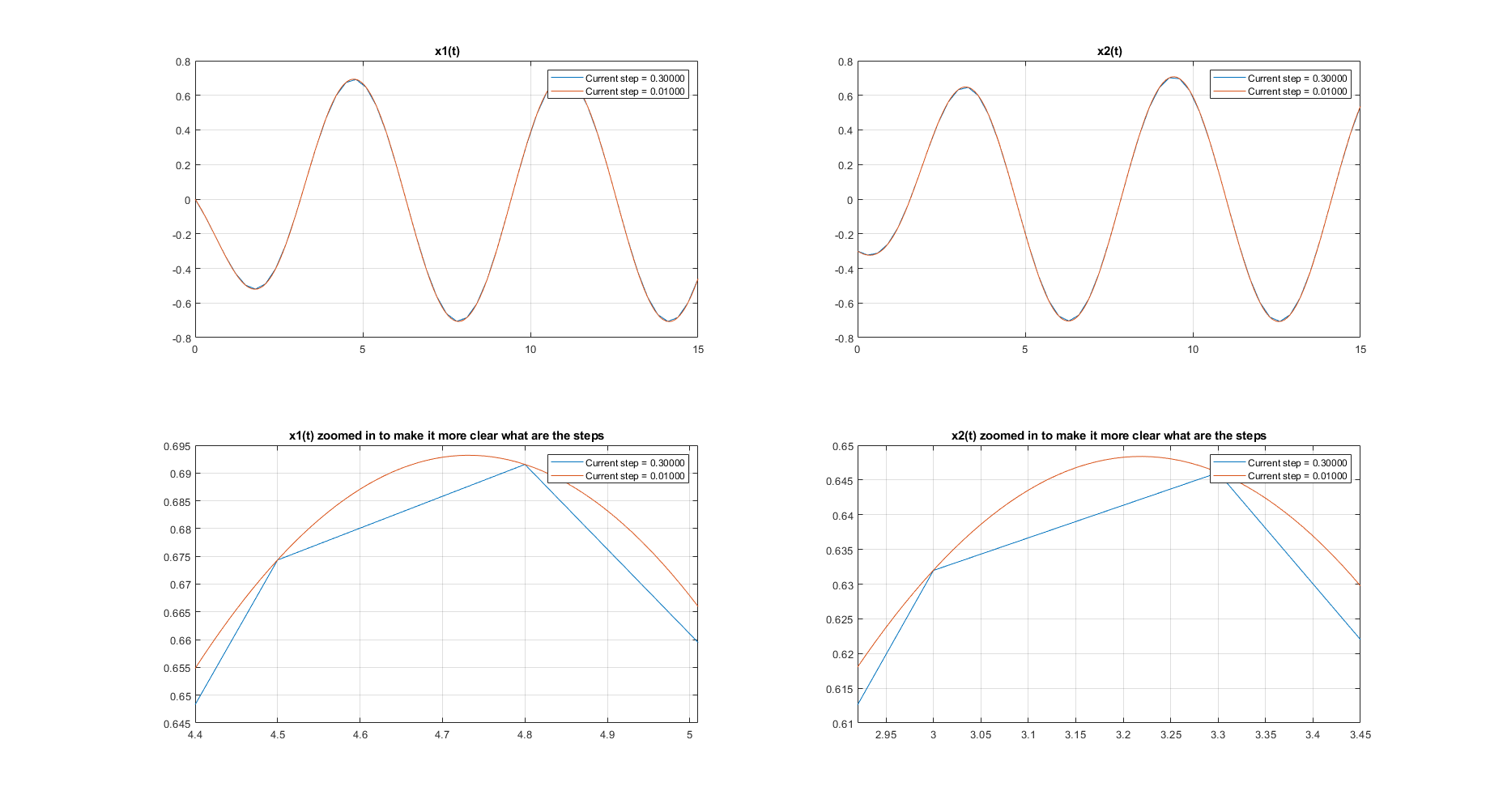


Figure 3 graphs with x1(t) and x2(t) plots, showing two different steps

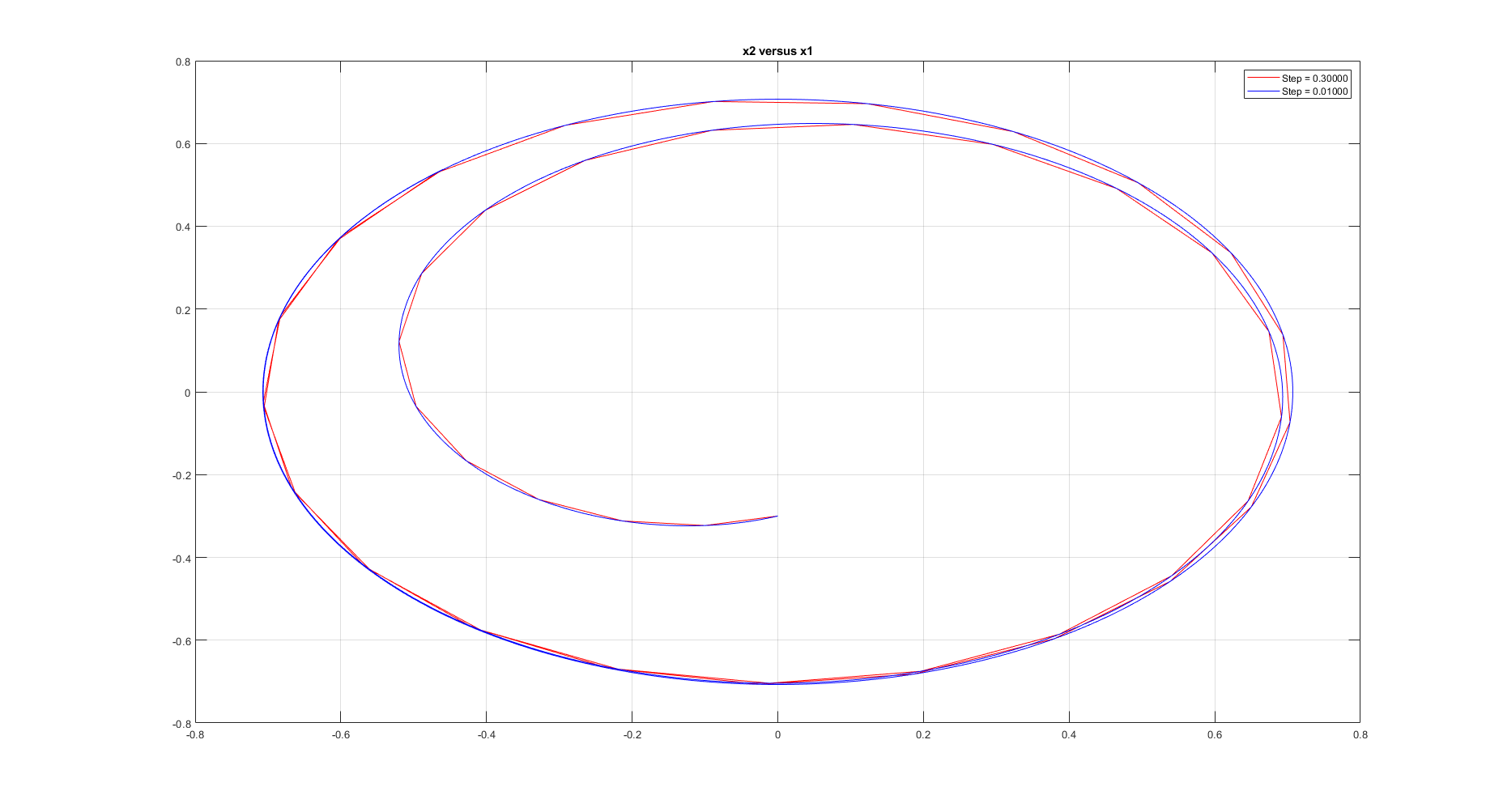


Figure 4 x2 versus x1 showing two different steps

Figures 3 and 4 show two different step-sizes: 0.3 and 0.01. The differences in results can be clearly seen: for 0.3 the graph is much sharper, because it plots less points on the graph. On the other hand, since 0.01 plots more points It will cause the program to run slower. After running and timing the program with a built-in MATLAB profiler, I saw that the program runs 0.3 seconds slower for step-size 0.01 than for step-size 0.3. I think that it’s not a huge difference so an optimal step-size 0.01 is good enough.

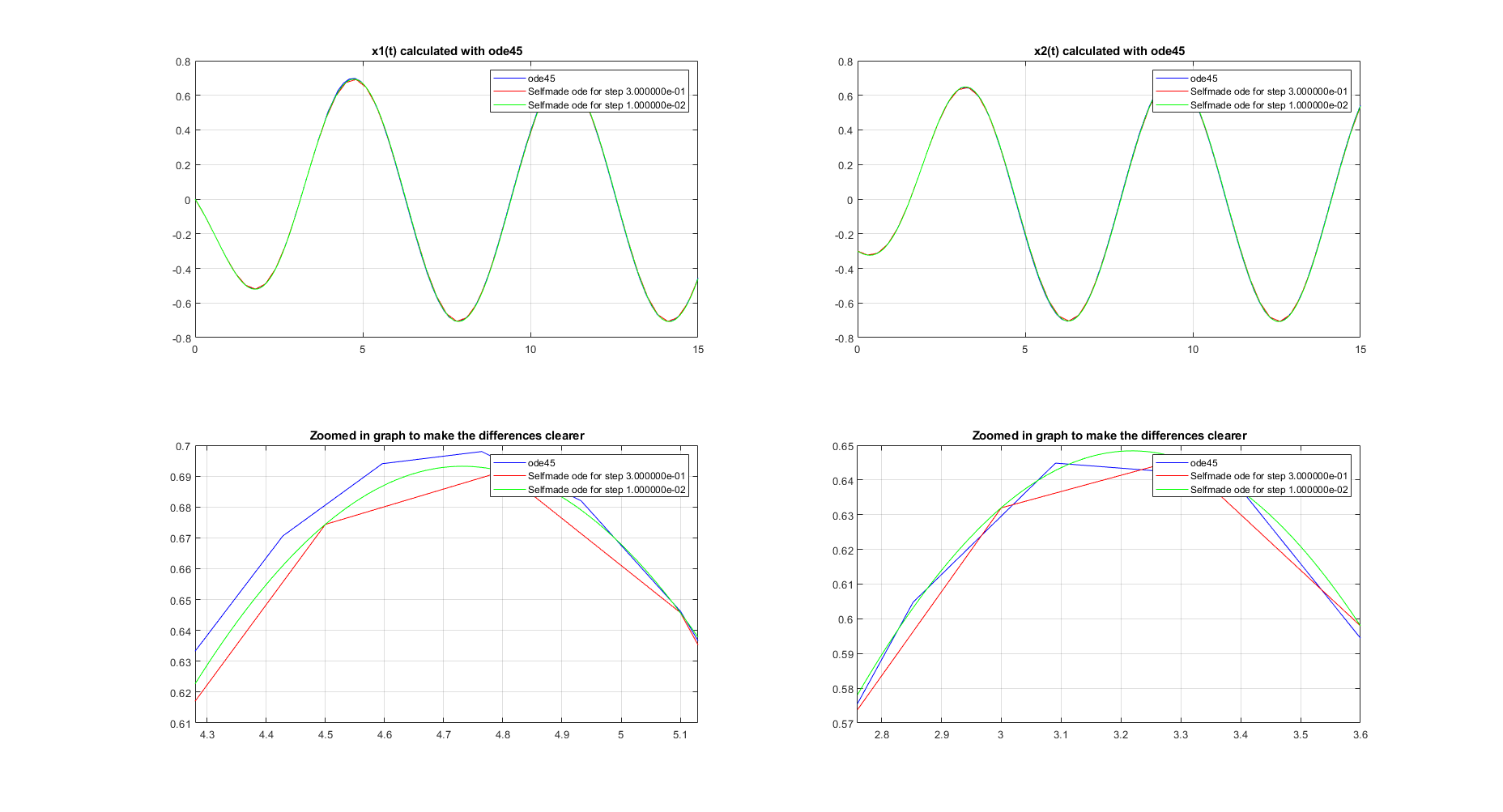


Figure 5 Two steps compared to ode45

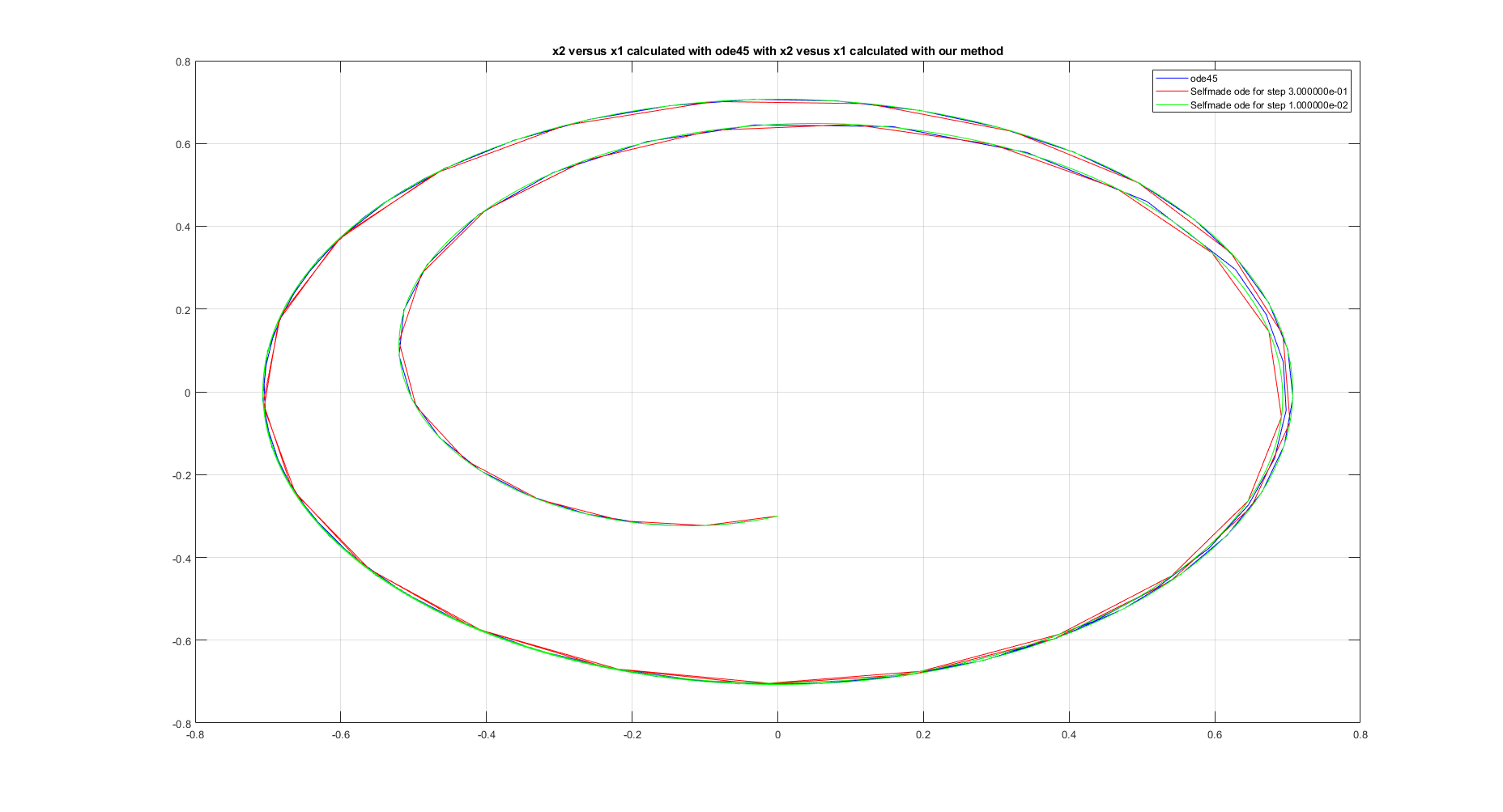


Figure 6 x2 versus x1 with two steps compared to ode45

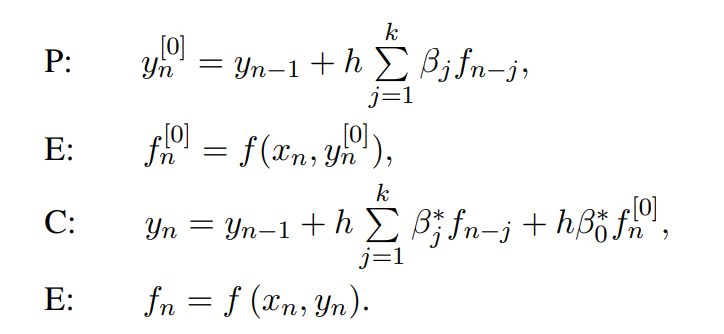
Figures 5 and 6 show that acquired results are very similar to the ones from built-in ode45 function, which means that our program is running correctly.

# Task 2a: Adam’s Predictor-Corrector Method

The second algorithm that we use is called Adam’s P5EC5E or Adam’s Predictor-Corrector method. It’s a multistep method, which uses two other algorithms within itself. As the predictor we use an explicit algorithm called Adams-Bashforth method and as a corrector we use implicit algorithm called Adams-Moulton method. The reason why we do this is because a PC algorithm must fulfil the following properties:

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

Explicit methods fulfil the last property, whereas the implicit methods fit the first two but not the last one. Hence if we combine them in a predictor-corrector method we can arrive at an optimal multi-step algorithm. Firstly, we use the Adam-Bashforth algorithm to find a somewhat accurate initial point yn, which we then use in an Adam-Moulton algorithm to find a more accurate result. Adam’s PC method looks as follows:



K in our case is 5. β and β\* are taken from the following tables:

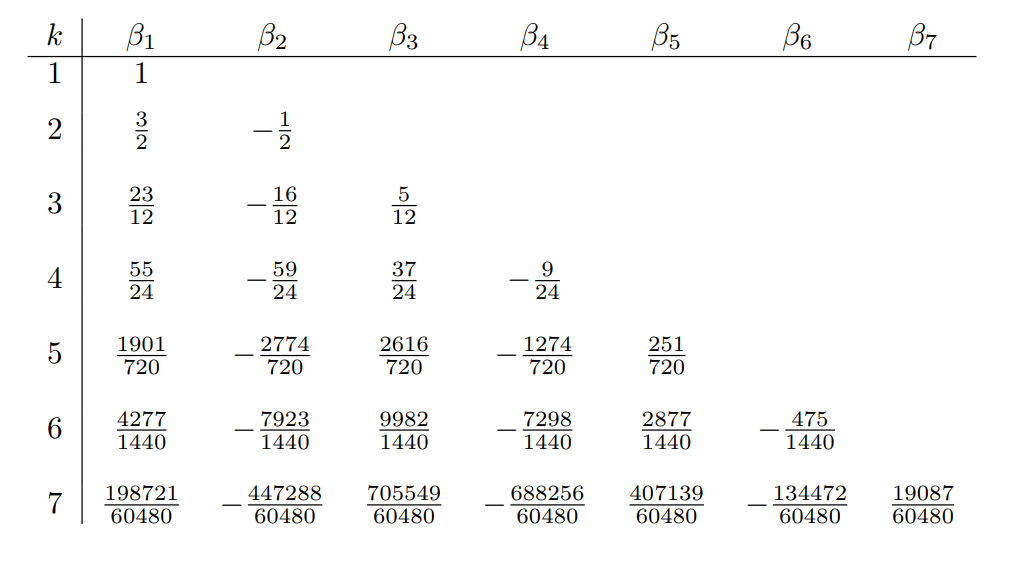
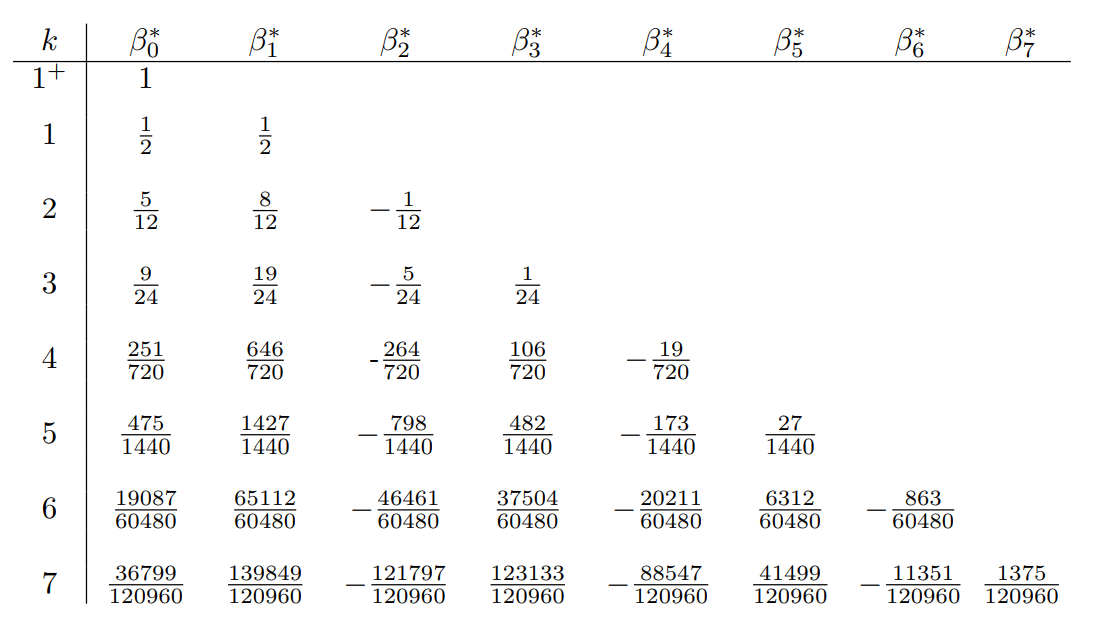
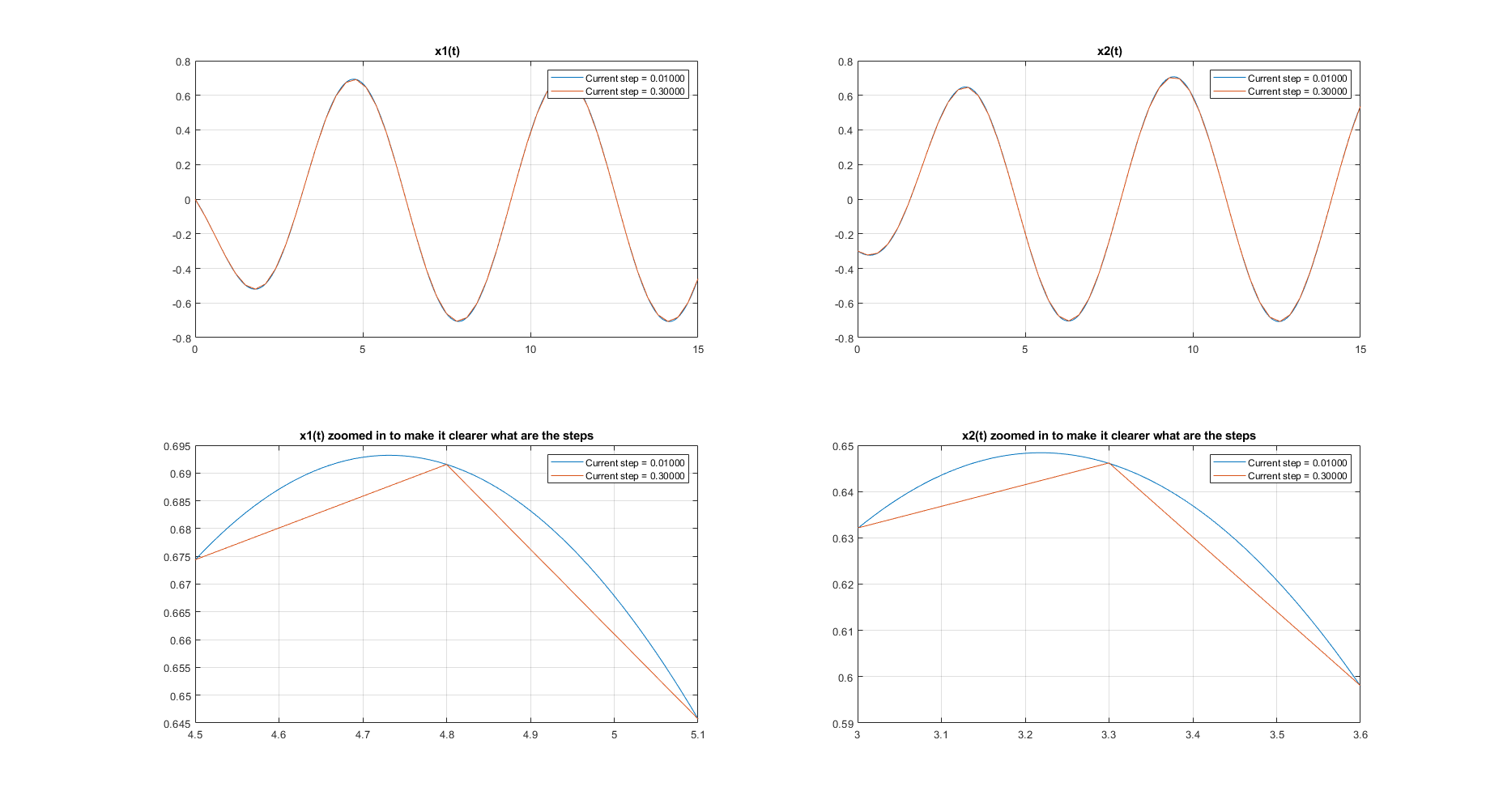
**

Figure 7 Two step-sizes for x1(t) and x2(t)

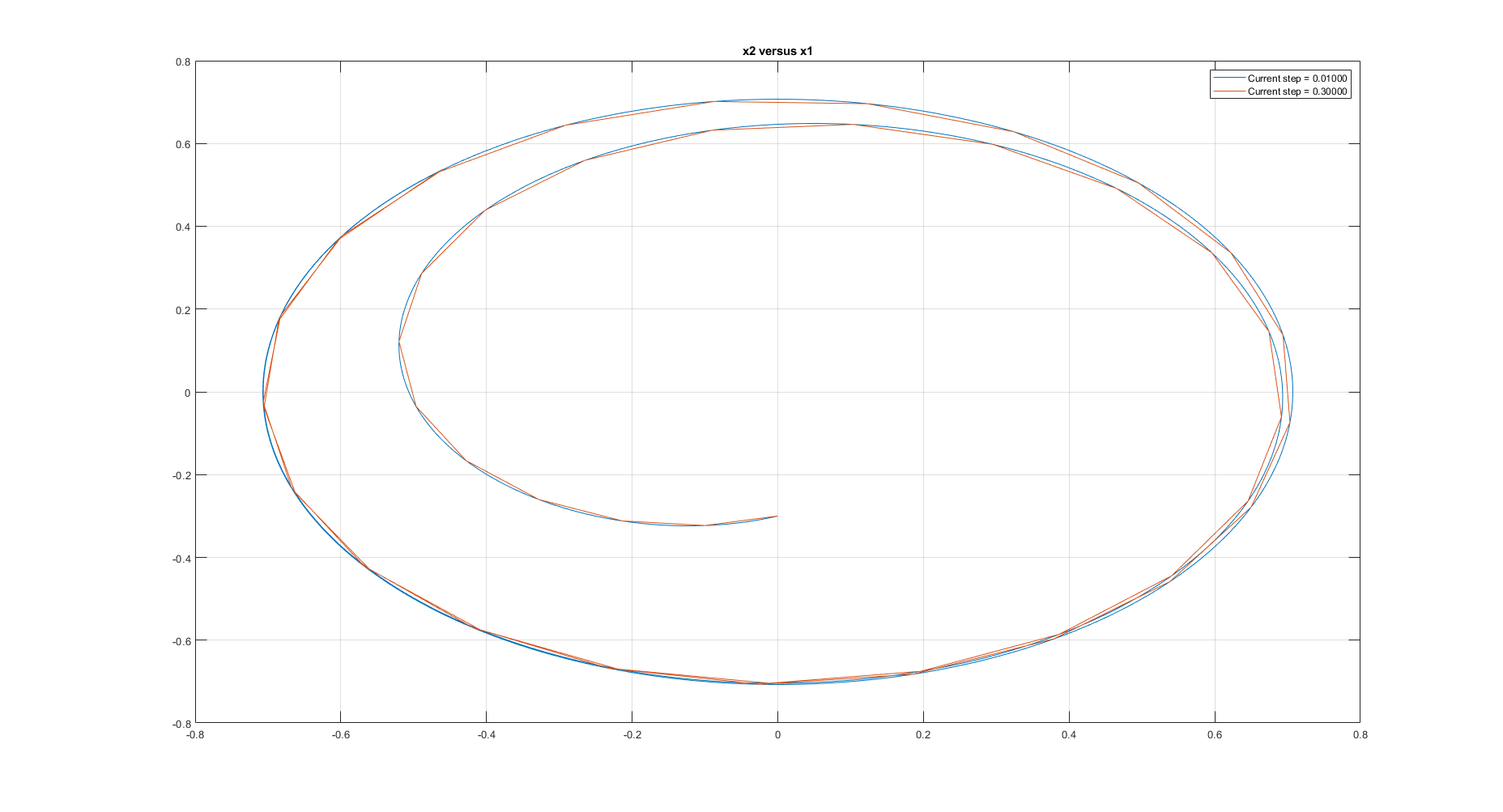


Figure 8 Two step-sizes for x2 versus x1

The process of choosing the optimal step for Adam’s method is exactly the same as in RK4 method: We plot the x1(t) and x2(t) with 2 different step-sizes, then plot x2 versus x1 with 2 different step-sizes and compare to ode45.

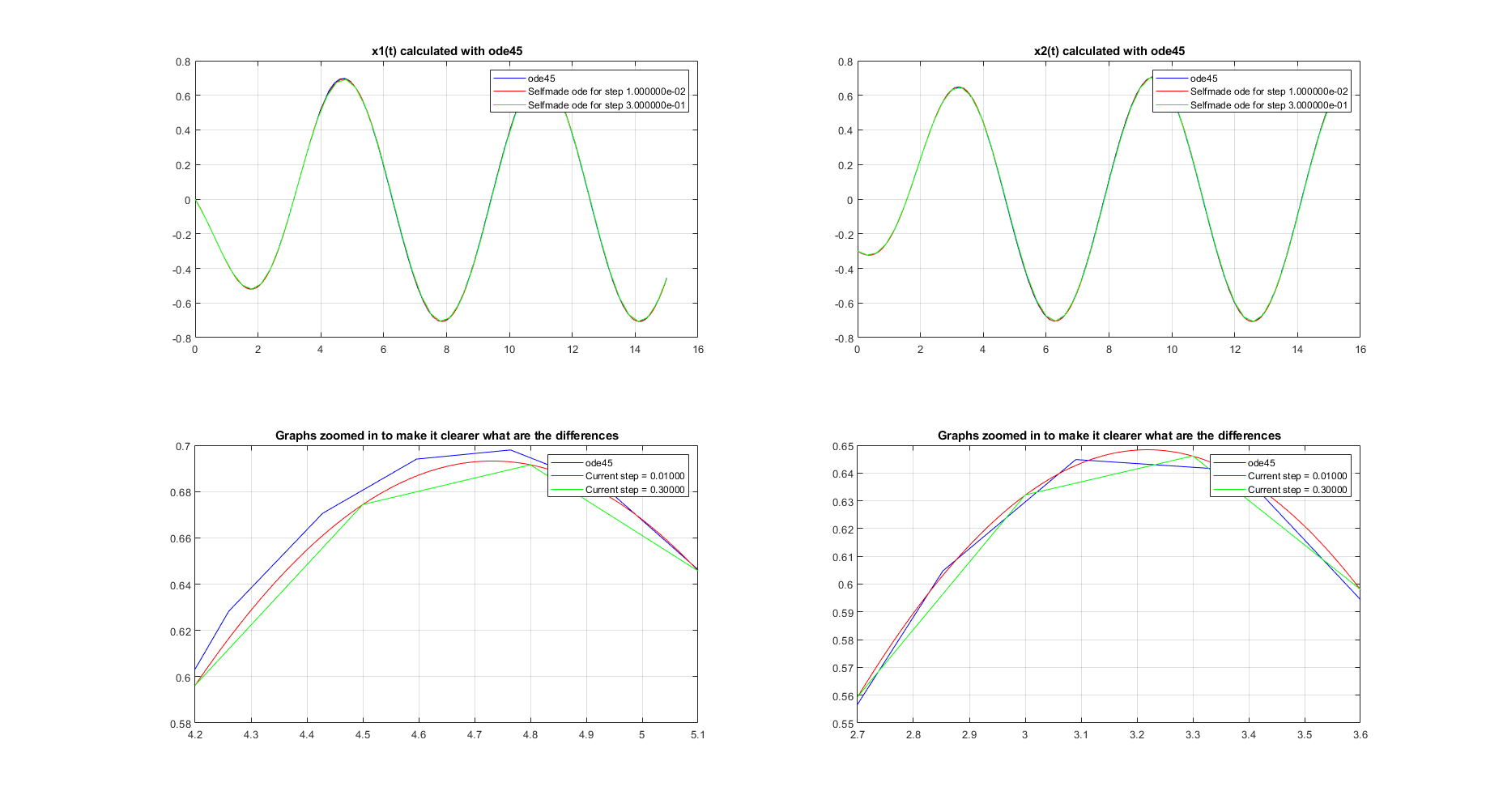


Figure 9 x1(t) and x2(t) with 2 step-sizes compared to ode45

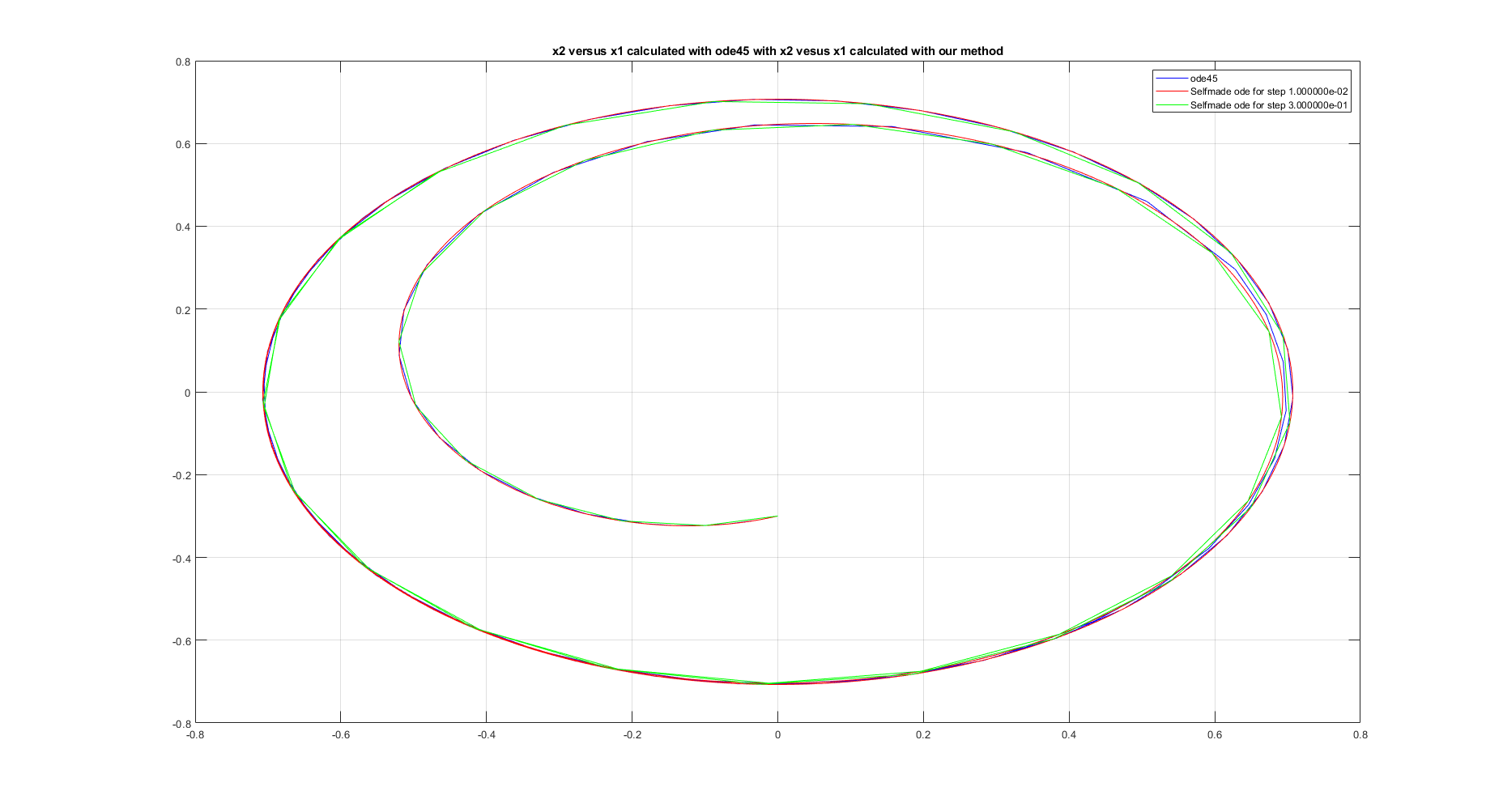
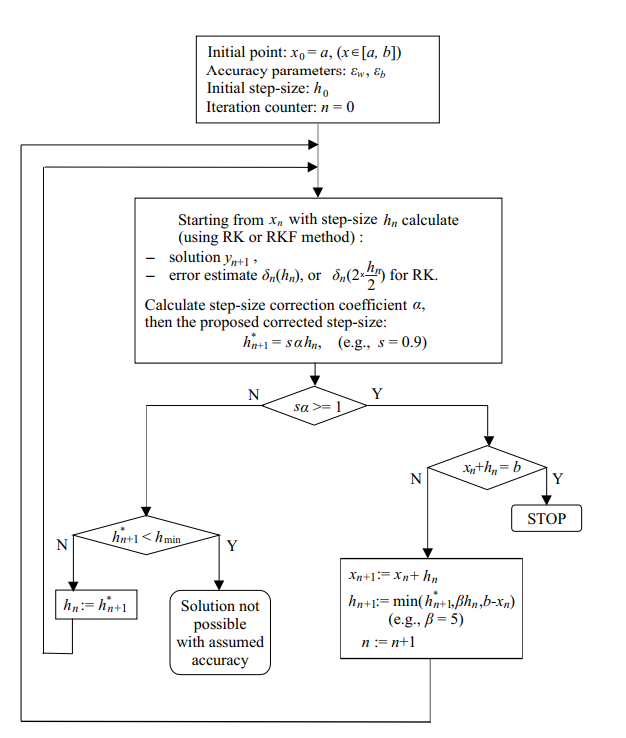


Figure 10 x2 versus x1 with two step-sizes compared to ode45

By looking at Figure 9 and 10 We can conclude that the step-size 0.01 is good, because the results for that step-size are very similar to ode45. As with RK4 I timed how long it takes to run the program with both 0.3 and 0.01 step-sizes. The difference is about 0.45 seconds and again I concluded that (given such a small time difference and a huge accuracy difference) step-size 0.01 is good enough. The program runs correctly.

# Task 2b

In subpoint b We use RK4 method again, except this time We use a variable step-size, automatically adjusted by the algorithm also making error estimation according to the step-doubling rule.



Above flowchart shows the process of running the program.

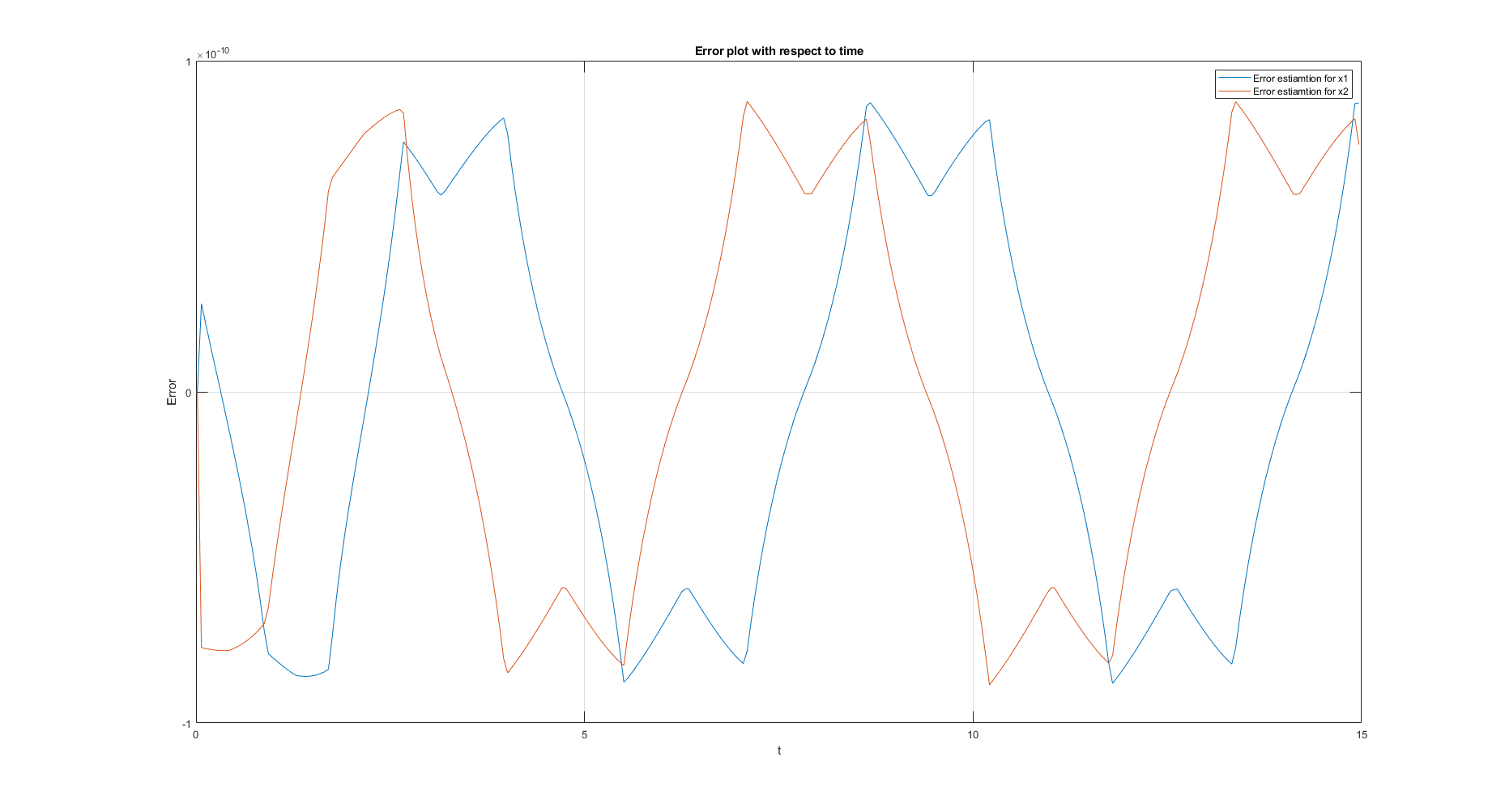


Figure 11 Error estiamtion plot

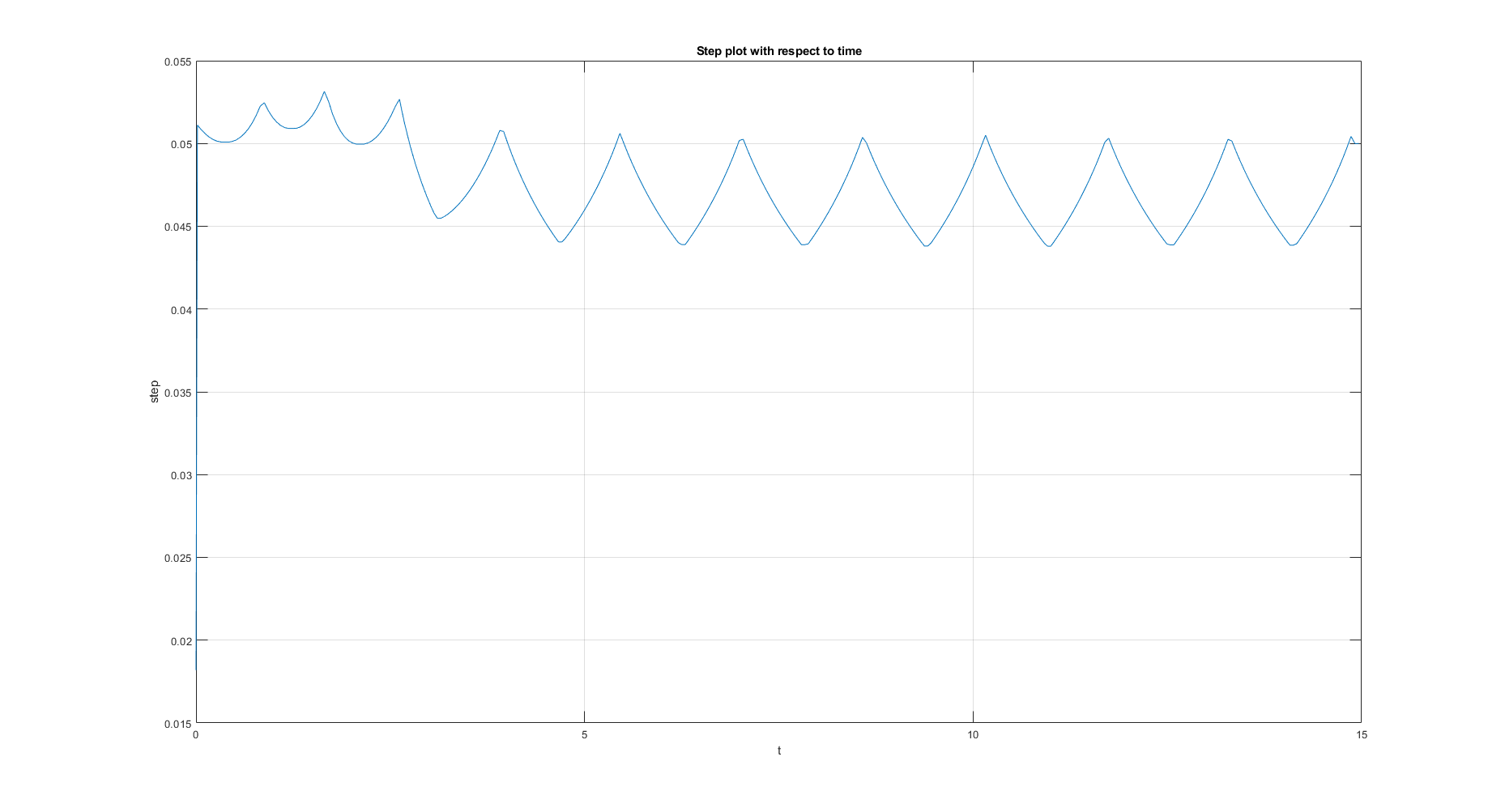


Figure 12 Plot showing how the program estimated the step-size

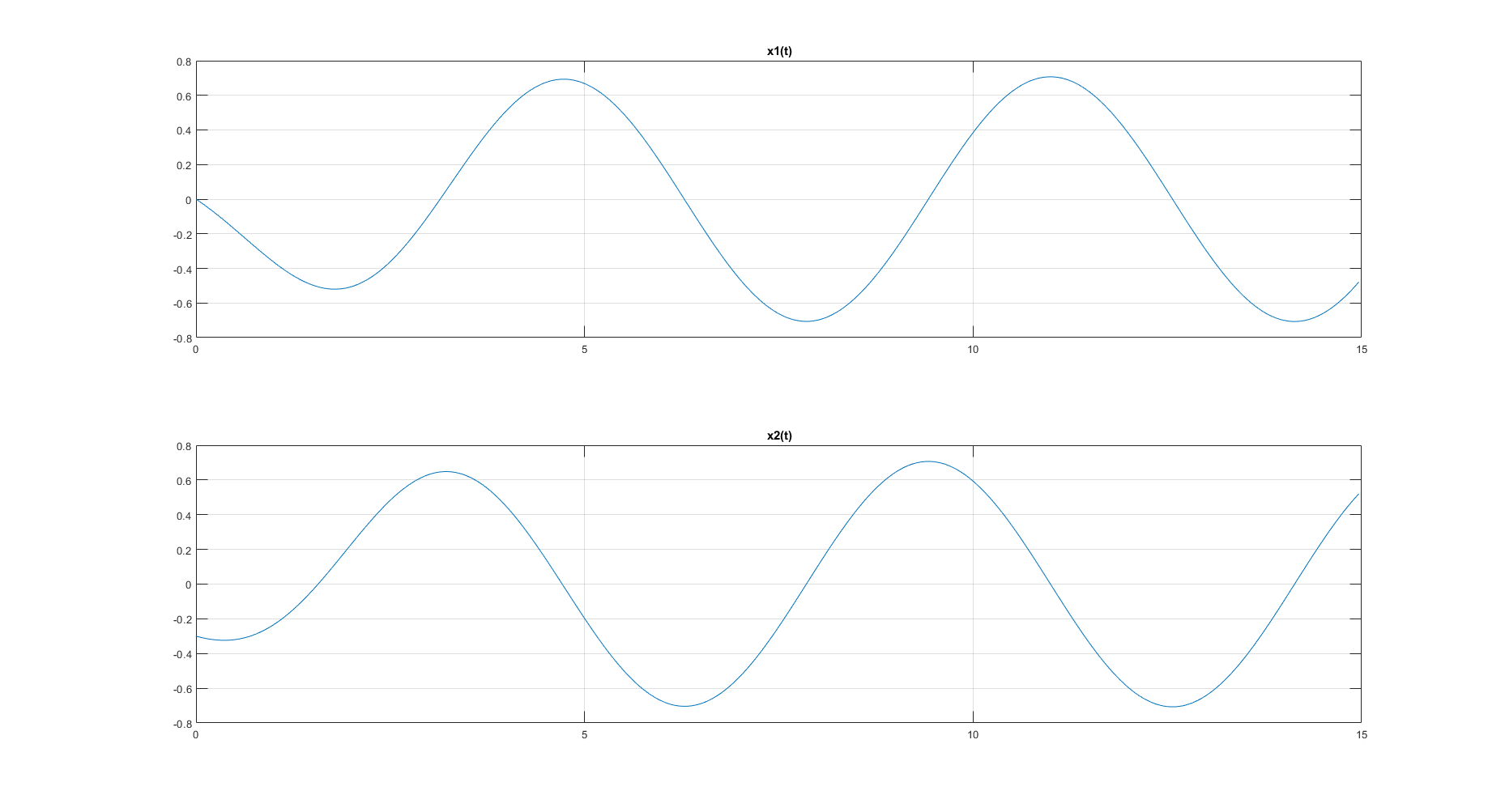


Figure 13 Plot of x1(t) and x2(t)

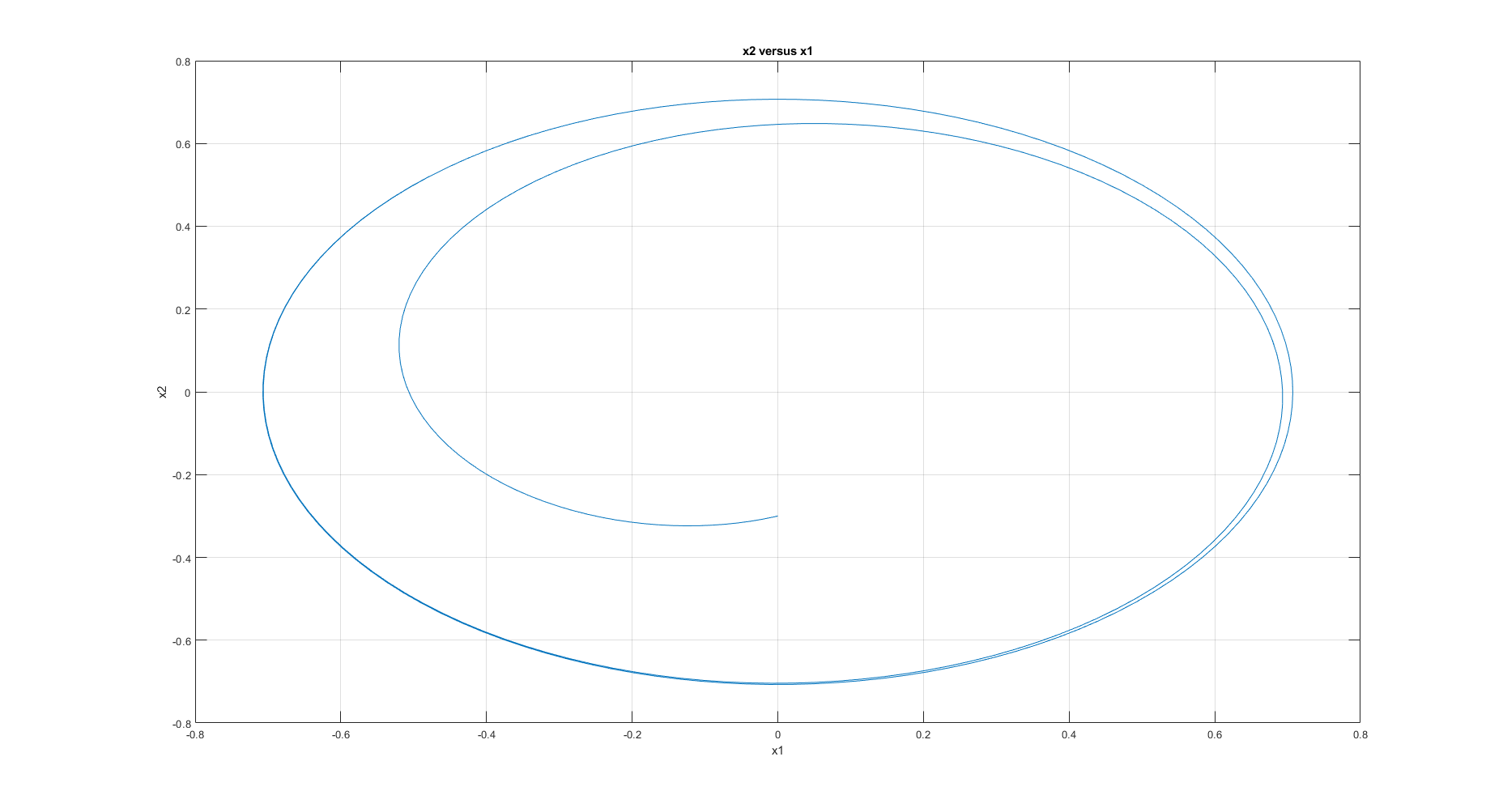


Figure 14 plot of x2 versus x1

Results are pretty much the same as in subpoint a, but the process is quite different. First, we do not have to manually change the step size, which is very convenient. Second, since the step size adjusts automatically, we can be sure that the results will be very accurate. Finally, the estimated error is very low, which is another indicator that the results are accurate.

# Conclusions

The last method that we used (RK4 with variable step) is the most accurate since it doesn’t involve the factor of human error when choosing the step size. It’s also the least tiresome to use. Considering the fact that all methods give almost identical results, it would be advisable to use RK4 with automatically chosen step-size.

# Code

# Task 1

function task1

clc;

xi = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5];

yi = [-4.9606; -3.3804; -1.4699; -1.1666; 0.4236; 0.1029; -0.5303; -4.04830; -11.0280; -21.1417; -33.9458];

n = 0; %starting degree of the polynomial

val = 10; %maximal degree of the polynomial

%Here I draw experimental data on the second graph

figure(2);

plot(xi,yi,'o', 'DisplayName', 'Experimental Data');

hold on

%the loop goes up to val,which is the maximal degree of the polynomial so

%that the program doesn't run for ages.

while n <= val

difference = zeros(n,1); %difference between the aquired result and y from the task

sol = zeros(length(xi),n+1);

for i = 1:1:length(xi)

for j = 0:1:n

sol(i, j+1) = xi(i)^j;

end

end

[Q, R] = QRdec(sol);

solutions = fliplr((R\(Q'\*yi))');

result = polyval(solutions, xi);

for j = 1:n

difference(j) = abs(yi(j) - result(j));

end

error = norm(difference);

gram\_cond = cond(sol'\*sol);

%%

%following code prints coefficients, solution error and condition

%number of Gram's Matrix

fprintf('Degree of polynomial: %d\n', n);

disp(solutions);

% fprintf('Solution error:')

% disp(error);

% fprintf('Condition number of Grams matrix:');

% disp(gram\_cond);

%%

%following code draws two graphs: figure(1) with all polynomials on one

%graph and figure(2) with plot of the degree that I chose as optimal

%with one degree higher, to show that they are almost identical.

x\_fit = linspace(-5,5);

y\_fit = polyval(solutions, x\_fit);

figure(1);

subplot(1,2,1)

plot(x\_fit, y\_fit, 'DisplayName', sprintf('Poly of deg %d', n));

grid on;

xlabel('x');

ylabel('y');

title('Plots of all polynomials');

hold on

subplot(1,2,2)

plot(x\_fit, y\_fit, 'DisplayName', sprintf('Poly of deg %d', n));

xlabel('x');

ylabel('y');

xlim([-5, -3.5]);

ylim([-15, 5]);

grid on

title('Zoomed plots of all polynomials');

hold on

if n == 3

figure(2)

plot(x\_fit, y\_fit, 'm', 'DisplayName', sprintf('Poly of deg: %d', n));

hold on

grid on

xlabel('x');

ylabel('y');

legend show

end

if n == 4

figure(2)

plot(x\_fit, y\_fit, 'k', 'DisplayName', sprintf('Poly of deg: %d', n));

hold on

grid on

xlabel('x');

ylabel('y');

legend show

hold off

end

n = n + 1;

legend show

end

end

%%

%following function calculates a QR decomposition of a given matrix A.

%Outputs: Q -> orthogonal matrix

% R -> upper triangular matrix

function [Q, R] = QRdec(A)

[m, n] = size(A);

R = zeros(n,n);

Q = zeros(m,n);

d = zeros(1,n);

for i = 1:1:n

Q(:,i) = A(:,i);

R(i,i) = 1;

d(i) = Q(:,i)'\*Q(:,i);

for j = i+1:1:n

R(i,j) = (Q(:,i)' \* A(:,j))/d(i);

A(:,j) = A(:,j) - R(i,j) \* Q(:,i);

end

end

for i = 1:1:n

dd = norm(Q(:,i));

Q(:,i) = Q(:,i)/dd;

R(i,i:n) = R(i,i:n) \* dd;

end

end

# Task 2 RK4 method

function RungeKutta

%%

%declaring interval and values of x (x\_val) given in the task

interval = [0, 15];

x\_val = [0, -0.3];

optimal\_step = 0.01; %I set a new variable to make it easier to look for the step

nonoptimal\_step = 0.3; %Similar as above, I set a new variable to make it easier to draw a non-optimal step

%%

%drawing plots of two different step-sizes to show why 0.01 is the most optimal

%one

figure(1)

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

subplot(2,2,1);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

grid on

title('x1(t)');

xlim([0, 15]);

legend show

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

subplot(2,2,2);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on;

grid on;

title('x2(t)');

xlim([0, 15]);

legend show

subplot(2,2,3)

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

grid on

title('x1(t) zoomed in to make it more clear what are the steps');

xlim([4.4, 5.01]);

legend show

subplot(2,2,4)

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

grid on

title('x2(t) zoomed in to make it more clear what are the steps');

xlim([2.92, 3.45]);

legend show

%%

%x2 versus x1 on one plot for non optimal and optimal step-sizes

figure(2)

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(x(1,:), x(2,:),'r', 'DisplayName', sprintf('Step = %0.5f', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(x(1,:), x(2,:),'b', 'DisplayName', sprintf('Step = %0.5f', optimal\_step));

hold on

grid on

title(sprintf('x2 versus x1'));

legend show

%%

%comparing x1(t) and x2(t) graphs for optimal and non-optimal step-sizes to built-in matlab function

%ode45

figure(3)

%optimal stepsize obtained by trial and error is 0.01

[t45, x45] = ode45(@fun\_val, interval, x\_val);

subplot(2,2,1);

%Here I draw two graphs: one for a non-optimal step-size and one for an

%optimal step-size

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(t45, x45(:,1),'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(1,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(1,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

grid on

title('x1(t) calculated with ode45');

legend show

subplot(2,2,2);

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(t45, x45(:,2), 'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(2,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

hold on

%Reseting the step size to show that 0.02 fits worse than 0.01

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(2,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

grid on

title('x2(t) calculated with ode45');

legend show

subplot(2,2,3);

%Here I draw two graphs: one for a non-optimal step-size and one for an

%optimal step-size

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(t45, x45(:,1),'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(1,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

hold on

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(1,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

grid on

title('Zoomed in graph to make the differences clearer');

xlim([4.28, 5.13]);

legend show

subplot(2,2,4);

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(t45, x45(:,2), 'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(2,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

hold on

%Reseting the step size to show that 0.02 fits worse than 0.01

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(t, x(2,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

grid on

title('Zoomed in graph to make the differences clearer');

xlim([2.76, 3.6]);

legend show

%%

%comparing x2 versus x1 of the selfmade ode calculator to the one built in

%matlab

figure(4);

[x,t] = Calc(nonoptimal\_step, interval, x\_val);

plot(x45(:,1),x45(:,2),'b', 'DisplayName', sprintf('ode45'));

hold on

plot(x(1,:), x(2,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

hold on

%Reseting the step size to show that 0.02 fits worse than 0.01

[x,t] = Calc(optimal\_step, interval, x\_val);

plot(x(1,:), x(2,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

grid on

title('x2 versus x1 calculated with ode45 with x2 vesus x1 calculated with our method');

legend show

end

%%

%function that stores both ODE in one array for convinience.

function [out] = fun\_val(t, x)

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

%%

%function that calculates x and t values using RK4 method

function [y, t] = Calc(step, interval, x\_val)

steps = floor(abs(interval(2) - interval(1))/abs(step));

index = 1;

t = zeros(1, steps + 1);

t(1) = interval(1);

y(:,1) = x\_val;

for i = interval(1) + step:step:interval(2)

k(:,1) = fun\_val(t(index), y(:,index));

k(:,2) = fun\_val(t(index), y(:,index)+step\*k(:,1)/2);

k(:,3) = fun\_val(t(index), y(:,index)+step\*k(:,2)/2);

k(:,4) = fun\_val(t(index), y(:,index)+step\*k(:,3));

%evaluating new y for next iteration

index = index + 1;

t(index) = i;

y(:,index) = y(:,index-1) + (step/6)\*(k(:,1)+2\*k(:,2)+2\*k(:,3)+k(:,4));

end

end

%%

# Task 2 Adam’s Predictor-Corrector method

function Adams\_PC

%%

interval = [0, 15]; % interval set as in the task

%following values are used to find optimal step by trial and error and also

%compare it to a non-optimal one

optimal\_step = 0.01;

nonoptimal\_step = 0.3;

x\_val = [0, -0.3]; %values of x as set in the task

%%

%rest of the function plots the graphs required to detemine whether the

%chosen step is the optimal one

figure(1)

[t,x] = Calc(interval, x\_val, optimal\_step);

subplot(2,2,1);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

grid on

title('x1(t)');

xlim([0, 15]);

legend show

subplot(2,2,2);

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on;

grid on;

title('x2(t)');

xlim([0, 15]);

legend show

subplot(2,2,3)

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(1,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

grid on

title('x1(t) zoomed in to make it clearer what are the differences');

xlim([4.5, 5.1]);

legend show

subplot(2,2,4)

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

% [t,x] = Calc(interval, x\_val, nonoptimal\_step);

% plot(t, x(2,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

% hold on

% grid on

% title('x2(t) zoomed in to make it clearer what are the differences');

% xlim([3, 3.6]);

% legend show

%%

figure(2)

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(x(1,:), x(2,:), 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(x(1,:), x(2,:), 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

grid on

title('x2 versus x1');

legend show

%%

figure (3)

[t45, x45] = ode45(@fun\_val, interval, x\_val);

[t,x] = Calc(interval, x\_val, optimal\_step);

subplot(2,2,1);

plot(t45, x45(:,1),'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(1,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(1,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

grid on

title('x1(t) calculated with ode45');

legend show

subplot(2,2,2);

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(t45, x45(:,2), 'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(2,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(2,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

grid on

title('x2(t) calculated with ode45');

legend show

subplot(2,2,3)

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(t45, x45(:,1), 'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(1,:), 'r', 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(1,:), 'g', 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

grid on

title('Graphs zoomed in to make it clearer what are the differences');

xlim([4.2, 5.1]);

legend show

subplot(2,2,4)

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(t45, x45(:,2), 'b', 'DisplayName', sprintf('ode45'));

hold on

plot(t, x(2,:),'r', 'DisplayName', sprintf('Current step = %0.5f', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(t, x(2,:), 'g', 'DisplayName', sprintf('Current step = %0.5f', nonoptimal\_step));

hold on

grid on

title('Graphs zoomed in to make it clearer what are the differences');

xlim([2.7, 3.6]);

legend show

%%

figure(4);

[t,x] = Calc(interval, x\_val, optimal\_step);

plot(x45(:,1),x45(:,2),'b', 'DisplayName', sprintf('ode45'));

hold on

plot(x(1,:), x(2,:), 'r', 'DisplayName', sprintf('Selfmade ode for step %d', optimal\_step));

hold on

[t,x] = Calc(interval, x\_val, nonoptimal\_step);

plot(x(1,:), x(2,:), 'g', 'DisplayName', sprintf('Selfmade ode for step %d', nonoptimal\_step));

grid on

title('x2 versus x1 calculated with ode45 with x2 vesus x1 calculated with our method');

legend show

end

%%

function [out] = fun\_val(t, x)

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

%%

function [t, y] = Calc(interval, x0, step)

%Allocation and initial values

steps = floor(abs(interval(2) - interval(1))/abs(step));

y(:,1) = x0;

t = zeros(1, steps + 1); %time

t(1) = interval(1);

temp\_step = step;

new\_index = 5;

%Beta parameters taken from the book

Beta\_explicit = [1901/720, -2774/720, 2616/720, -1274/720, 251/720];

Beta\_implicit = [475/1440, 1427/1440, -798/1440, 482/1440, -173/1440, 27/1440];

for index = 2:5

%using an algorithm of RK4 to calculate the initial values of the

%function to later use them in Adam's PC method. It's ran for times in

%order to reduce any possible errors.

t(index) = temp\_step;

temp\_step = temp\_step + step;

k(:,1) = fun\_val(t(index-1), y(:,index-1));

k(:,2) = fun\_val(t(index-1), y(:,index-1)+step\*k(:,1)/2);

k(:,3) = fun\_val(t(index-1), y(:,index-1)+step\*k(:,2)/2);

k(:,4) = fun\_val(t(index-1), y(:,index-1)+step\*k(:,3));

y(:,index) = y(:,index-1) + (step/6)\*(k(:,1)+2\*k(:,2)+2\*k(:,3)+k(:,4));

end

for i = temp\_step:step:interval(2)

new\_index = new\_index + 1;

t(new\_index) = i;

sum = 0;

%Prediction

for j = 1:1:5

if new\_index - j < 1

break;

end

sum = sum + Beta\_explicit(j) \* fun\_val(t(new\_index-j), y(:, new\_index-j));

end

xstep = y(:,new\_index-1) + step \* sum;

%Correction

sum = 0;

for j = 1:1:5

if new\_index - j < 1

break;

end

sum = sum + Beta\_implicit(j+1) \* fun\_val(t(new\_index-j), y(:,new\_index-j));

end

y(:,new\_index) = y(:,new\_index-1) + step \* sum + step \* Beta\_implicit(1) \* fun\_val(t(new\_index-1), xstep);

end

end

# Task 2b RK4 with automatically adjusted step-size

%%

function task2b

interval = [0,15]; %interval set in the task

init\_step = 4; %initial step. I chose 4 becuase it gives a very smooth result

%both errors were chosen to be very small but not smaller than machine

%epsilon

relative = 1e-10;

absolute = 1e-10;

%the result should be much grater than min\_step so if the result is

%smaller then something went wrong and the program should give throw an

%error

min\_step = 1e-6;

x\_val = [0, -0.3]; %initial values set in the task

%using Runge-Kutta method with variable step for values described above

[x ,t, error\_error, step] = RungeKuttaVar(init\_step, relative, absolute, min\_step, interval, x\_val);

%%

%the rest of the function draws all the graphs

figure(1)

plot(t, error\_error(1,:), 'DisplayName', sprintf('Error estiamtion for x1'));

hold on

plot(t, error\_error(2,:), 'DisplayName', sprintf('Error estiamtion for x2'));

grid on

title('Error plot with respect to time');

xlabel('t');

ylabel('Error');

legend show

%%

figure(2)

subplot(2,1,1);

plot(t, x(1,:));

grid on

title('x1(t)');

subplot(2,1,2)

plot(t, x(2,:));

grid on

title('x2(t)');

%%

figure(3)

plot(x(1,:), x(2,:));

grid on

ylabel('x2');

xlabel('x1');

title('x2 versus x1');

%%

figure(4)

plot(t, step);

grid on

xlabel('t');

ylabel('step');

title('Step plot with respect to time');

end

%%

%the following function uses Runge-Kutta implementation from the previous

%task with a change so that it can adjust the step on its own

function [x ,t, error\_error, step] = RungeKuttaVar(init\_step, relative, absolute, min\_step, interval, x\_val)

T = 5;

different\_step = 0.9; %taken from the flowchart from the report

step(1) = init\_step; %seting the initial value of the step

error\_error(:,1) = [0,0]; %preallocating data for later use for the error;

n = 1;

steps = floor(abs(interval(2) - interval(1))/abs(step)); %calculating the number of iterations

t = zeros(1, steps+1);

t(1) = interval(1);

x(:, 1) = x\_val;

while t(n) + step(n) < interval(2)

%calculating Runge-Kutta for a fullstep

k(:,1) = fun\_val(t(n), x(:,n));

k(:,2) = fun\_val(t(n), x(:,n)+step(n)\*k(:,1)/2);

k(:,3) = fun\_val(t(n), x(:,n)+step(n)\*k(:,2)/2);

k(:,4) = fun\_val(t(n), x(:,n)+step(n)\*k(:,3));

full\_step = x(:, n) + step(n)/6\*(k(:, 1)+2\*k(:, 2)+2\*k(:, 3)+k(:, 4));

step2 = step(n)/2;

half\_step = x(:,n);

x(:,n+1) = full\_step;

%calculating the Runge-Kutta again, this time with half of the original

%step-size

for i = 1:1:2

k(:,1) = fun\_val(t(n), half\_step);

k(:,2) = fun\_val(t(n), half\_step + step2\*k(:,1)/2);

k(:,3) = fun\_val(t(n), half\_step + step2\*k(:,2)/2);

k(:,4) = fun\_val(t(n), half\_step + step2\*k(:,3));

half\_step = half\_step + step2/6 \* (k(:,1) + 2 \* k(:,2) + 2 \* k(:,3) + k(:,4));

end

%estimating the error of the calculations

error\_estim = (half\_step - full\_step)/(2^4 - 1);

error\_error(:,n+1) = error\_estim;

ei = abs(half\_step)\*relative + absolute;

alpha = min((ei./abs(error\_estim)).^(1/5));

%calculating the new step-size according to the flowchart from the

%report

step\_p = different\_step \* alpha \* step(n); %value of the step calculated according to the formula on the flowchart

if different\_step\*alpha >= 1

if t(n) + step(n) == interval(2)

return

end

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

step(n+1) = min([step\_p,T\*step(n),interval(2)-t(n)]);

n = n+1;

else

if step\_p < min\_step

error("Solution not possible with chosen accuracy ");

else

step(n) = step\_p;

end

end

end

end

%Here I combine two functions given in the task into one for convenience

function [out] = fun\_val(t, x)

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

# Sources

* Numerical Methods – Piotr Tatjewski