Numerical Methods, project C, Number 32

Krzysztof Rudnicki
Student number: 307585
Advisor: dr hab. Piotr Marusak

January 9, 2022

Contents

Chapter 1

Determine polynomial function fitting experimental data

1.1 Problem

Given following samples:

x_i	y_i
-5	-6.5743
-4	0.9765
-3	3.1026
-2	1.8572
-1	1.3165
0	-0.6144
1	0.1032
2	0.3729
3	2.5327
4	7.3857
5	9.4892

We have to determine polynomial function $y = f(x)$ that best fits this data.
We will use least-square approximation using system of normal equation with QR factorization.

1.2 Theoretical introduction

1.2.1 Linear Least Squares Problem

We have polynomial function:

$$y(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$$

where n - degree of the polynomial we use to approximate function
Given matrix $\mathbf{A} \in \mathbb{R}^{m \times n} \quad (m > n)$ and vector $\mathbf{y} \in \mathbb{R}^m$
We need to find vector $\hat{\mathrm{x}}$ such that:

$$\forall x \in \mathbb{R}^n \quad ||\mathbf{y} - \mathbf{A}\hat{x}||_2 \leq ||\mathbf{y} - \mathbf{A}x||_2$$

$\mathbf{A}\hat{x}$ is the vector of values of the approximating function at each data point and y is the vector of values of data points.
We have to minimize the Euclidean norm of difference between approximation and data points.
To obtain $\hat{x}$ we will calculate the derrivative of error function with respect to b_n and equate it to 0. This way we will get system of equations caled **normal equations** which have unique solution in form of vector $\hat{x}$

1.2.2 Gram matrix and QR decomposition

We will use Gram's matrix since it's condition number tends to be pretty high which improves the accuracy of our solution.
Gram's matrix is a matrix of set of normal equations. The simplest way to present normal equations is by using formula:

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

Which then leads to following matrix $\mathbf{A}$:

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

In our case we define matrix **A** as:

$$A = \begin{bmatrix} 1 & x_0 & (x_0)^2 & \dots & (x_0)^n \\ 1 & x_1 & (x_1)^2 & \dots & (x_1)^n \\ 1 & x_2 & (x_2)^2 & \dots & (x_2)^n \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & x_N & (x_N)^2 & \dots & (x_N)^n \end{bmatrix}$$

Where x_N is the x-axis position for the N-th data point. Then we can express system of natural equations using A in this way:

$$A^T y = (A^T A)\hat{x}$$

With condition number of A equal to square root of condition number of Gram's matrix which gives us small errors. It is also usefull for QR factoization since we can rewrite it as follows:

$$(R^T Q^T)(QR)\hat{x} = (R^T Q^T)y$$

Since determinant of $R \neq 0$ and $Q^T Q = I$ we can simplify aforementioned equation to:

$$R\hat{x} = Q^T y$$

And to optimize even further ahead insetead of using QR decomposition we can use faster $\tilde{Q}\tilde{R}$ decomposition:

$$\tilde{R}\hat{x} = (\tilde{Q}^T \tilde{Q})^{-1}(\tilde{Q}^T)$$

This is faster since $\tilde{R}$ is an upper triangular matrix, easily solved using back-substitution.

1.2.3 Gram-Schmidt algorithm

We will use Gram-Schmidt algorithm for $\tilde{Q}\tilde{R}$ decomposition. Gram-Schmidt algorithm takes non-orthogonal function basis denoted as:

$$\phi_0, \dots, \phi_n$$

and orthogonalizes it into orthogonal function basis denoted as:

$$\psi_0, \dots, \psi_n$$

Standard Gram-Schmid algorithm looks like this:

$$\psi_0 = \phi_0$$

$$\psi_1 = \phi_1 - \frac{\langle \psi_0, \phi_1 \rangle}{\langle \psi_0, \psi_0 \rangle} \psi_0$$

$$\psi_n = \phi_n - \sum_{j=0}^{i-1} \frac{\langle \psi_j, \phi_i \rangle}{\langle \psi_j, \psi_j \rangle} \psi_j$$

Where $i = 2, \ldots, n$ Then we get a very well conditioned set of normal equations:

$$a_0 \langle \psi_0, \psi_0 \rangle = \langle f, \psi_0 \rangle$$

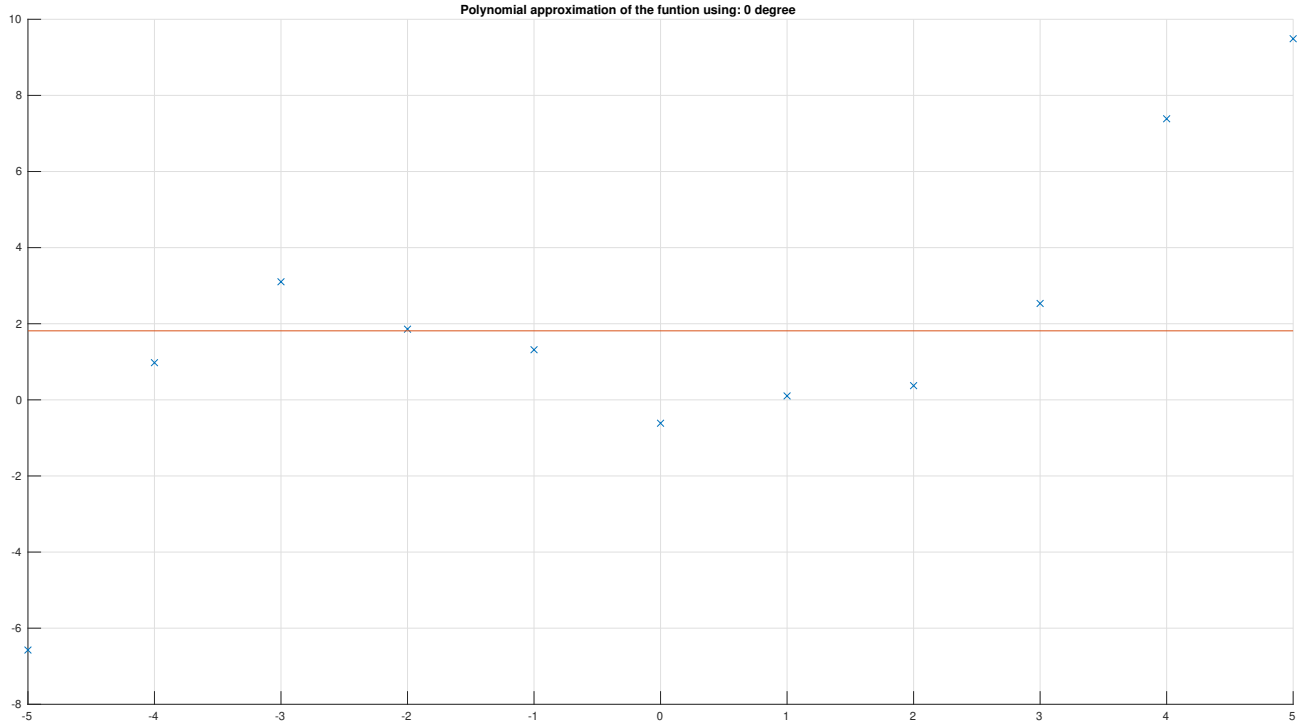
$$a_1 \langle \psi_1, \psi_1 \rangle = \langle f, \psi_1 \rangle$$

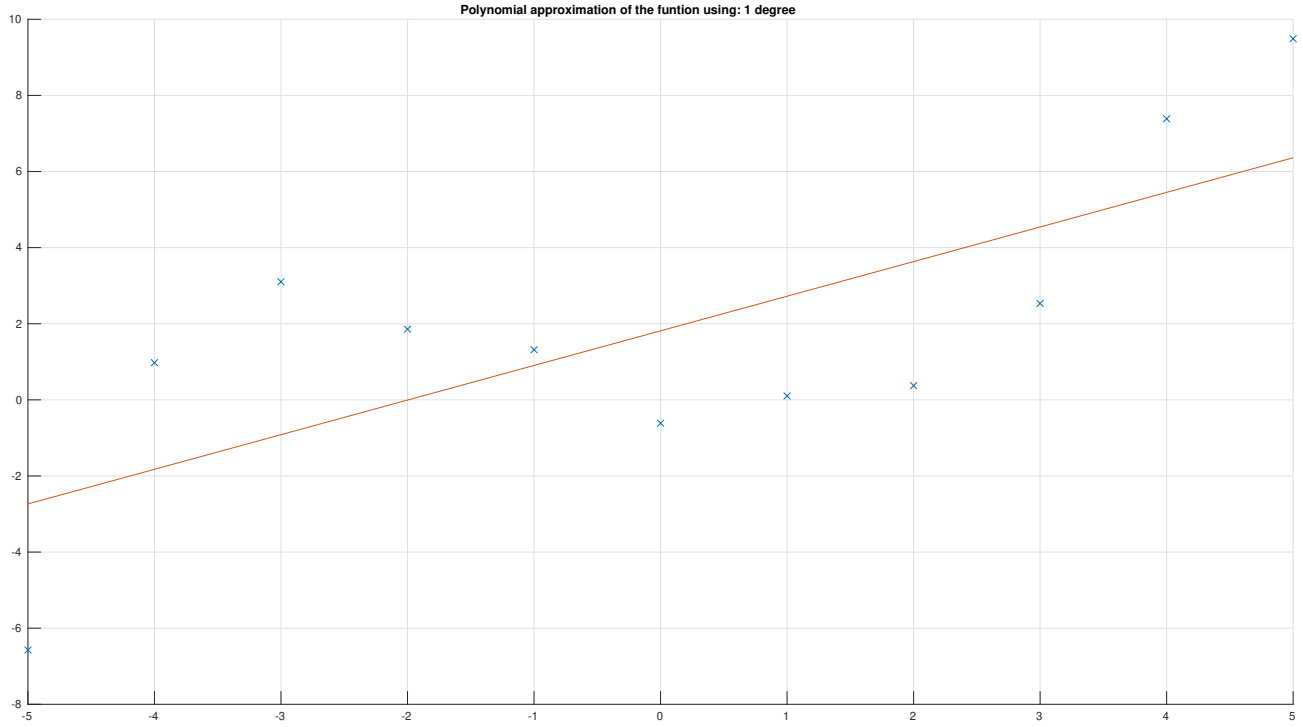
$$\vdots$$

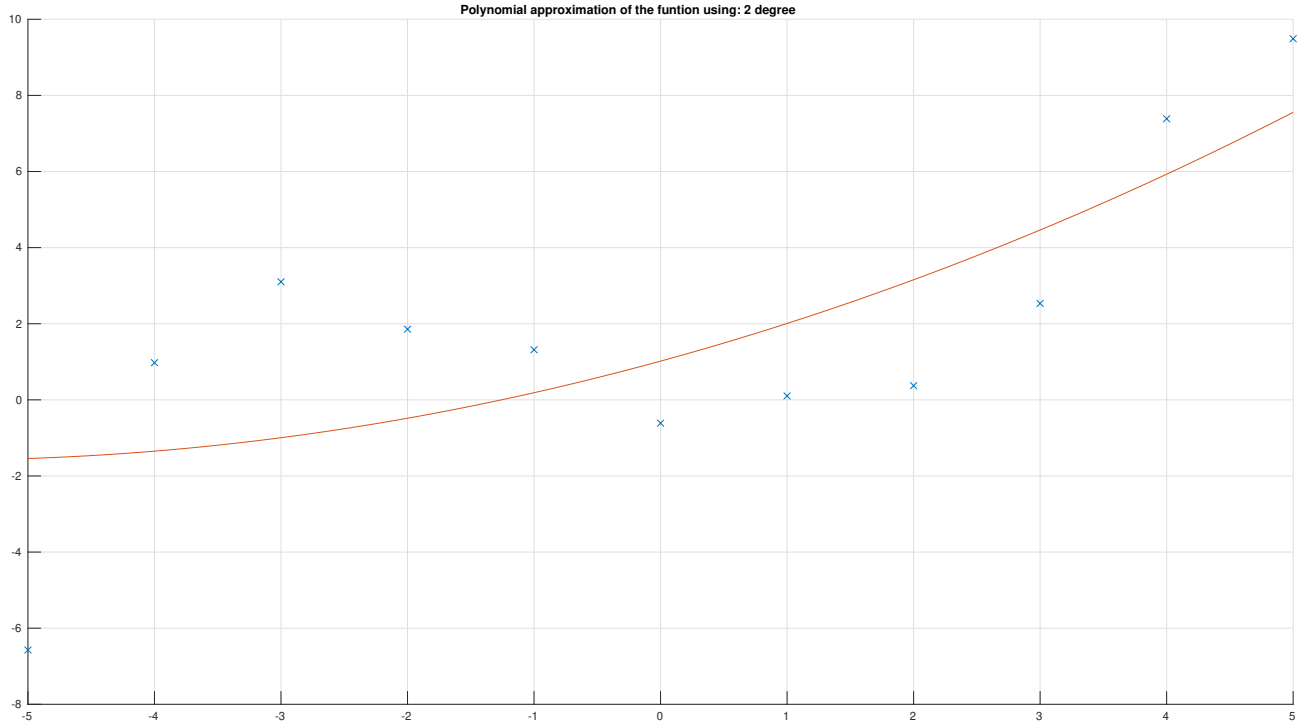
$$a_n \langle \psi_n, \psi_n \rangle = \langle f, \psi_n \rangle$$

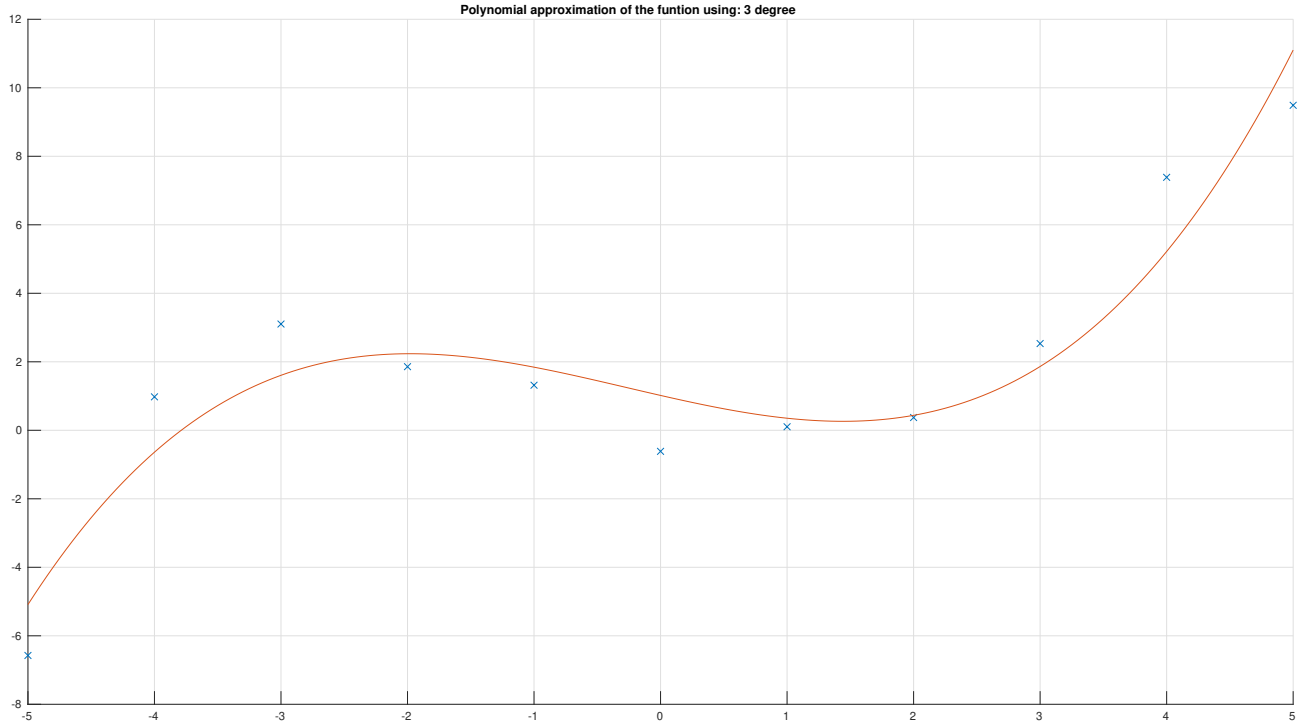
1.3 Results

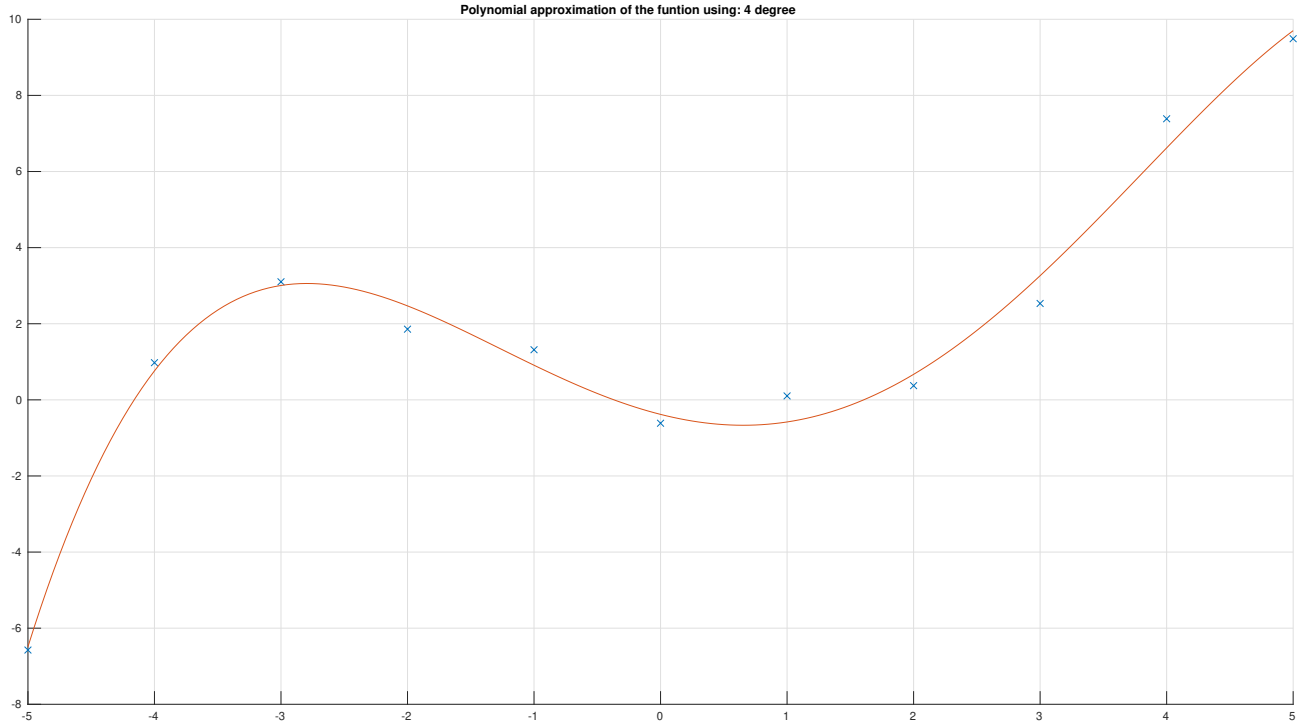
I decided to simulate polynomials up to $\mathbf{10^{th}}$ degree.

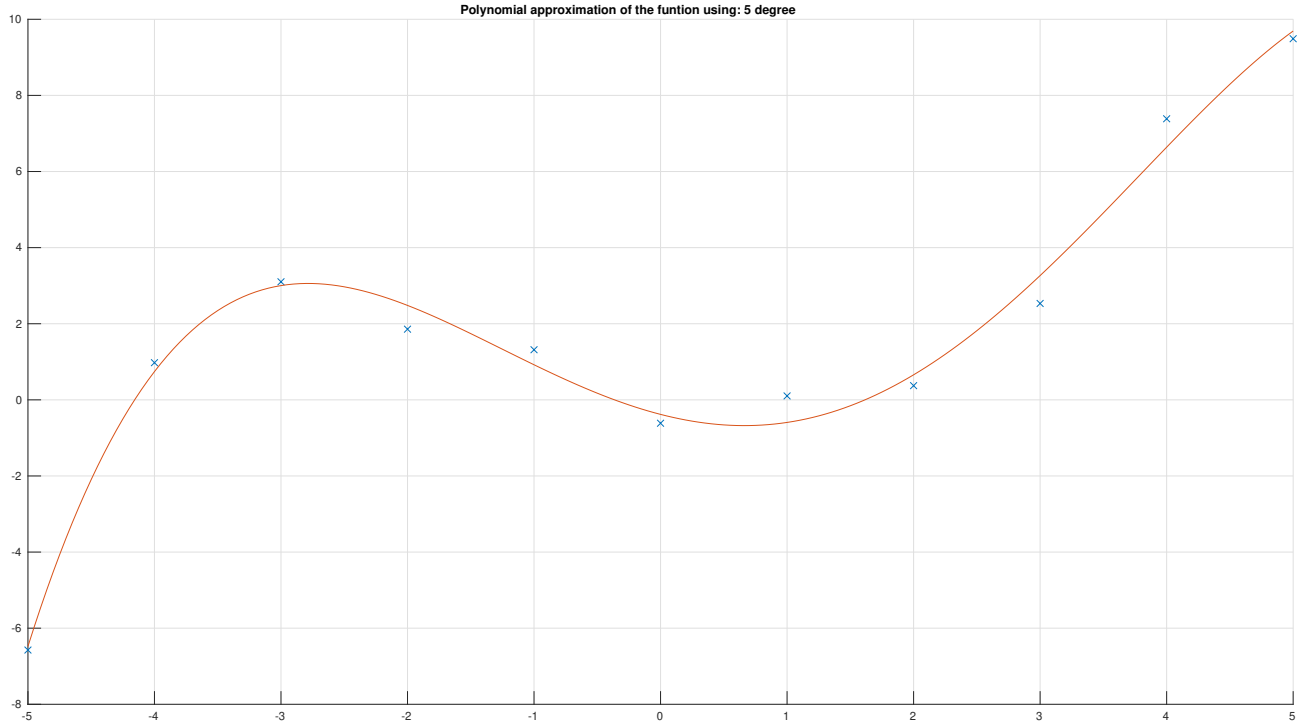

Polynomial approximation of the funtion using: 0 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5

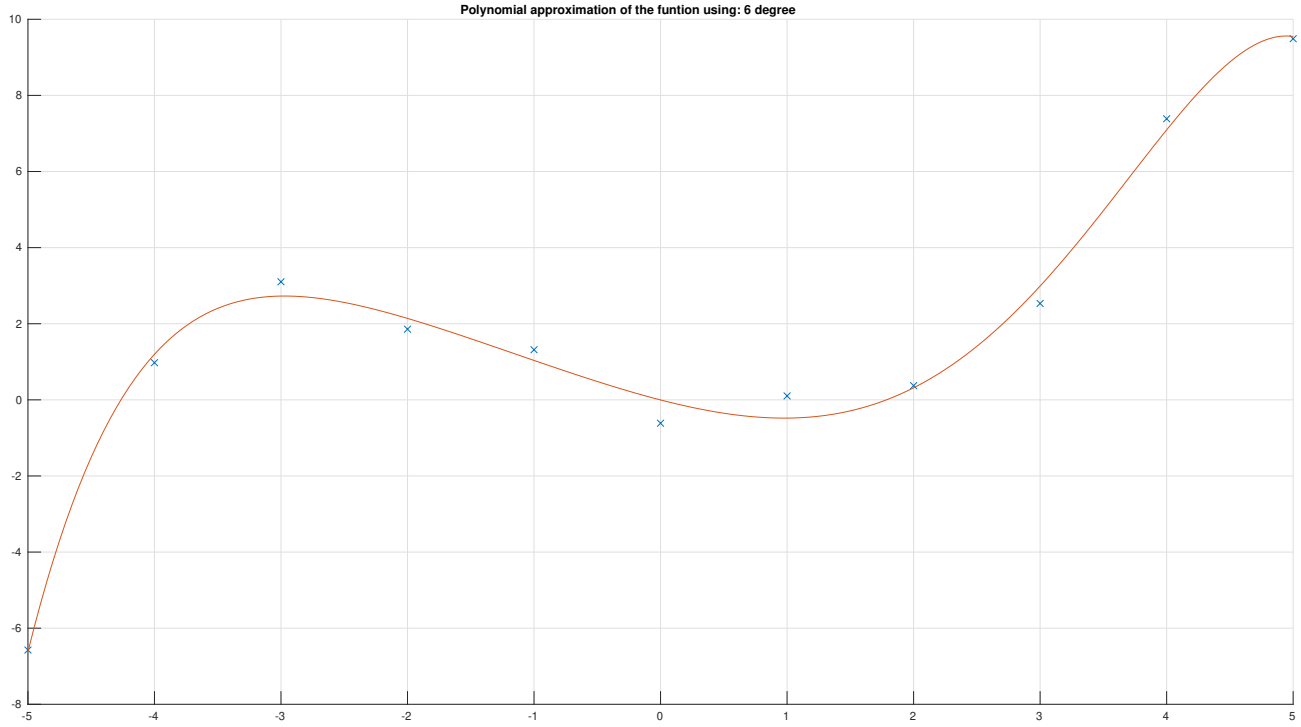

Polynomial approximation of the funtion using: 1 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5

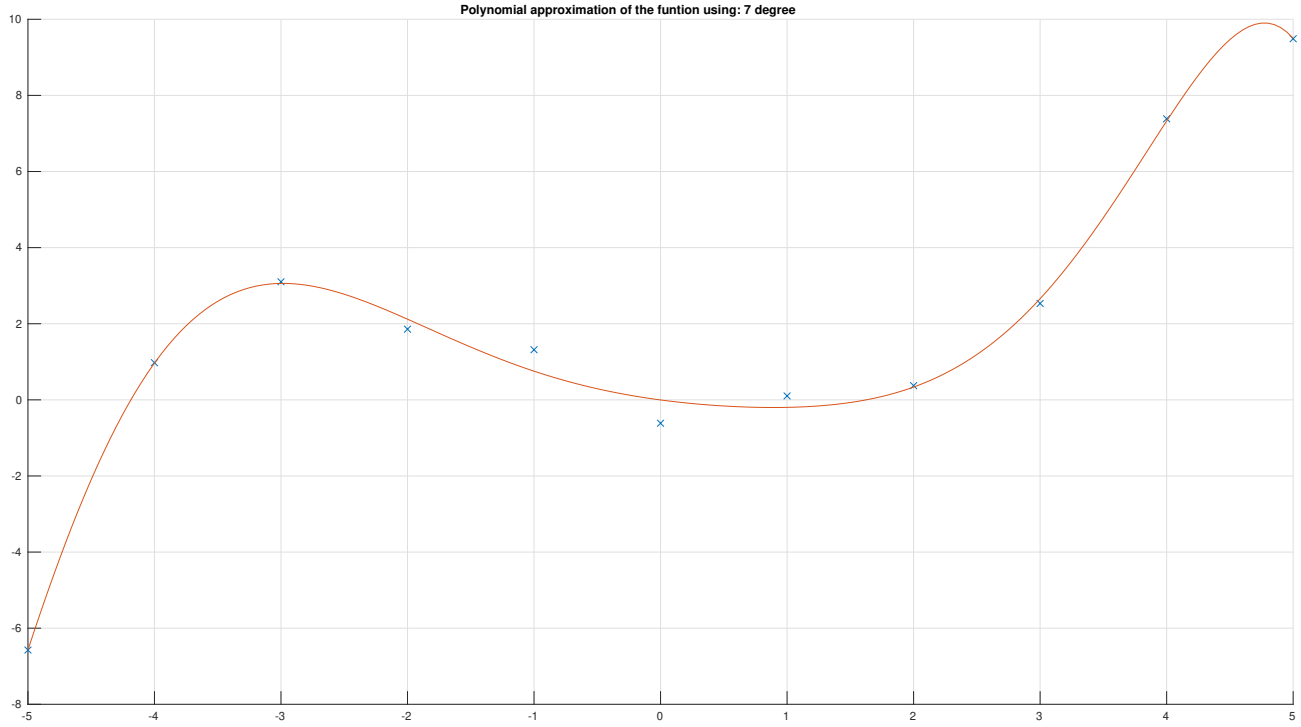

Polynomial approximation of the funtion using: 2 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5

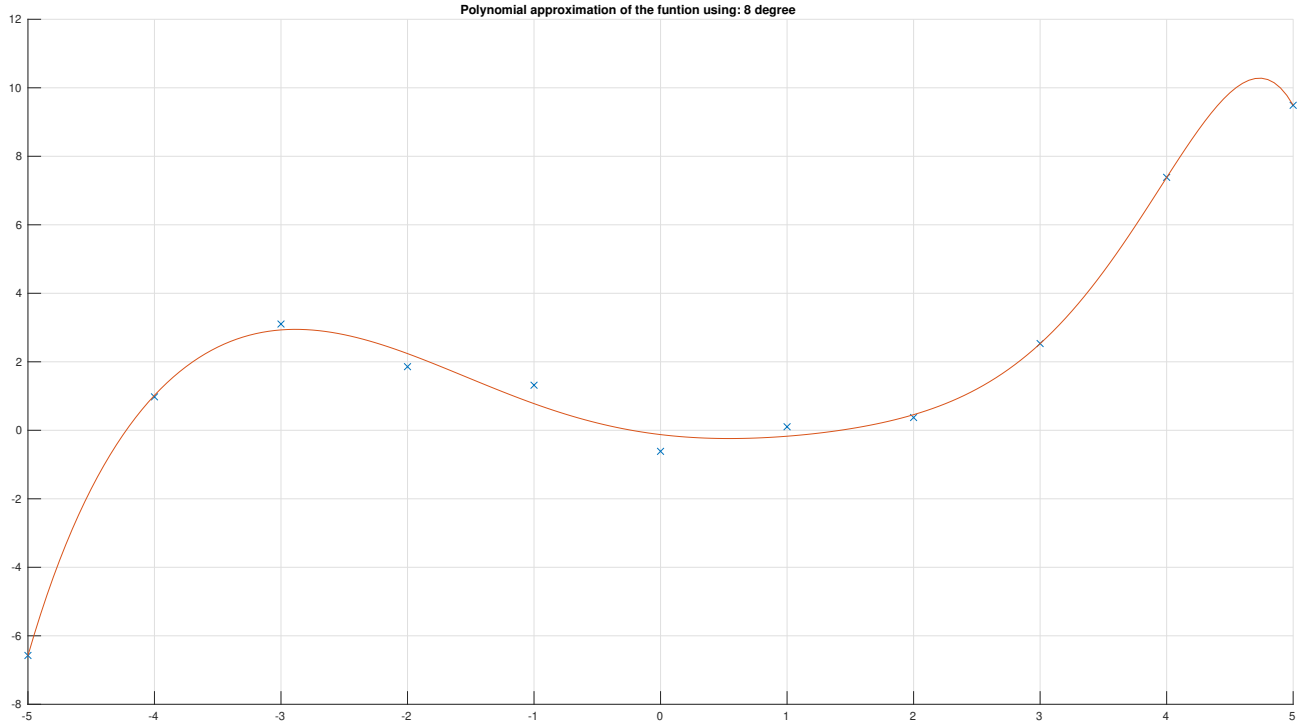

Polynomial approximation of the funtion using: 3 degree
12
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5

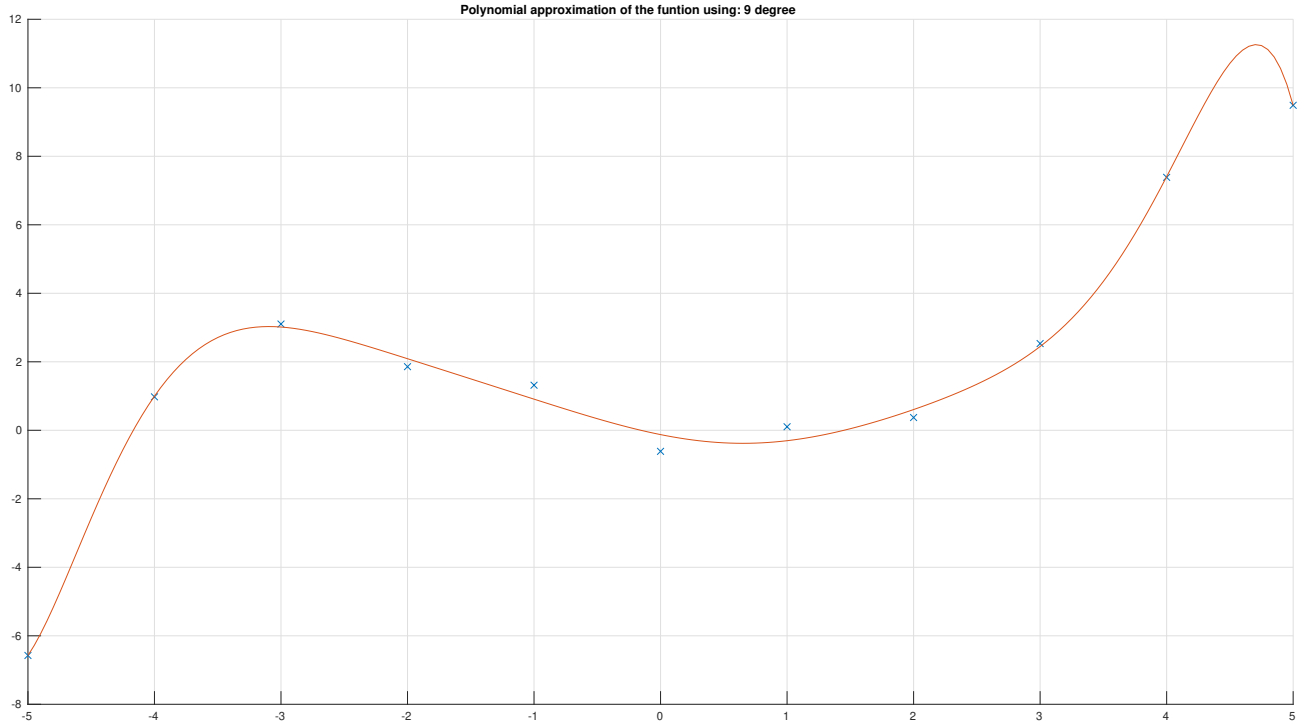

Polynomial approximation of the funtion using: 4 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5

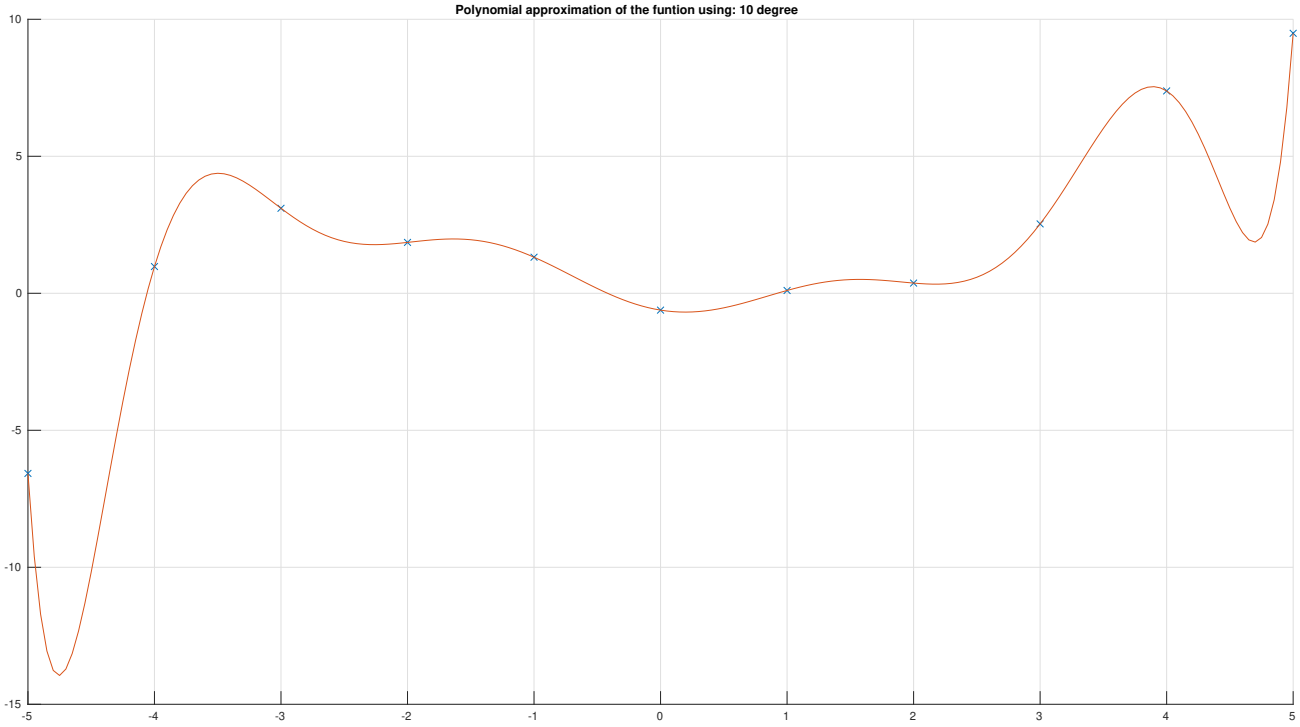

Polynomial approximation of the funtion using: 5 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5

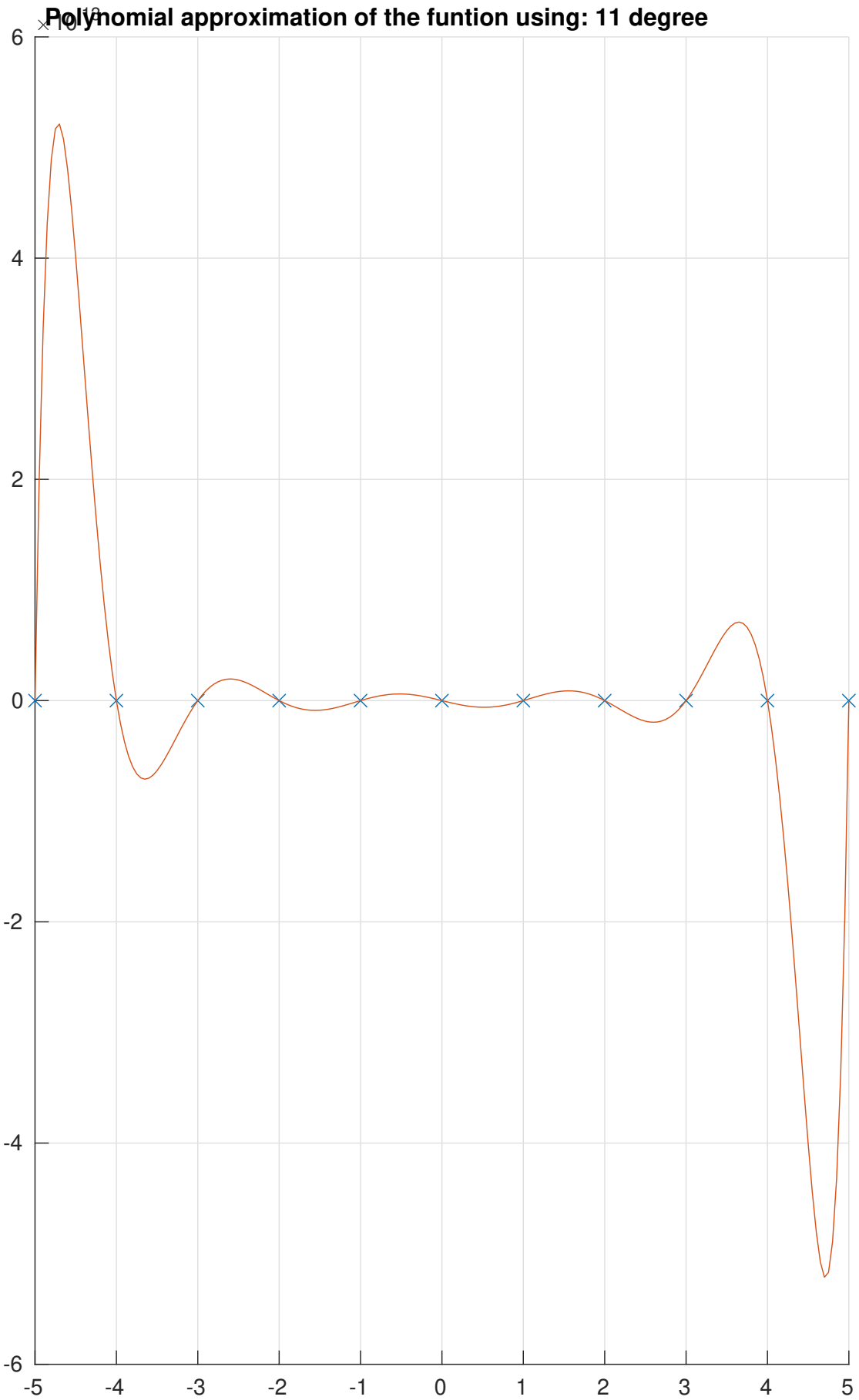

Polynomial approximation of the funtion using: 6 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5


Polynomial approximation of the funtion using: 7 degree
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5


Polynomial approximation of the funtion using: 8 degree
12
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5


Polynomial approximation of the funtion using: 9 degree
12
10
8
6
4
2
0
-2
-4
-6
-8
-5
-4
-3
-2
-1
0
1
2
3
4
5


Polynomial approximation of the funtion using: 10 degree
10
5
0
-5
-10
-15
-5
-4
-3
-2
-1
0
1
2
3
4
5


Polynomial approximation of the funtion using: 11 degree
×10^13
6
4
2
0
-2
-4
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5

Degree	Error	Condition number of Gram's Matrix
0	13.2040	1
1	9.1281	10
2	8.8257	408.7796
3	4.2365	8.5584e+03
4	1.5418	3.1798e+05
5	1.5413	7.4675e+06
6	1.1689	2.8316e+08
7	0.9345	7.6462e+09
8	0.8883	3.3055e+11
9	0.8334	1.5167e+13
10	6.3645e-13	9.2930e+14
11	2.2390	8.7763e+17

1.4 Discussion of results

As expected, with the increase in polynomial degree error of the solution decreased and condion number of Gram's matrix increased. The most interesting part is how for polynomial of degree **10** the error got significantly lower, 10^{-12} times lower than for polynomial of lower degree, with only around 100 times bigger Gram's matrix condition number, and then for polynomial of degree 11, the error become much much higher and so did Gram's matrix. I suppose that since we get 11 data points, approaching number of data points with the degree of the polynomial lowers the erorr until we have the same polynomial degree as data points at which point the error increases again.

Chapter 2

Determine trajectory of the motion

2.1 a) Runge-Kutta method of 4^{th} order and Adams PC

2.1.1 Problem

We are given following equations:

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

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

And we have to determine the trajectory of the motion on interval $[0, 15]$ with following initial conditions: $x_1(0) = 8; x_2(0) = 9$ In this section we will use Runge-Kutta method of 4^{th} order and Adams PC with different step-sizes until we find an optimal constant step size - when the decrease of the step size does not influence the solution significantly.

2.1.2 Theoretical Introdution

Determining the trajectory of the motion in this example can be mathematically described as solving a system of first order ordinary differential equations with given initical conditions. That is why we use Runge-Kutta method of order 4 which deals with exactly this kind of problem.

Runge-Kutta method of order 4

Any single step method are defined by the following formula, where h is a fixed step-size:

$$y_{n+1} = y_n + \Phi_f(x_n, y_n, h)$$

where:

$$x_n = x_0 + nh$$

$$y(x_0) = y_0 = y_a$$

where y_a is given. Runge-Kutta method of order 4 - RK4 method looks like this:

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

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

$$k_3 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1)$$

$$k_2 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2)$$

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

2.1.3 Adams PC method

Adams method with P_kEC_kE algorithm has the following form:
P:

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

E:

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

C:

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

E:

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

When compared to explicit and implicit adams method PC metho has smaller absolute stability intervals than implicit method, while having significant advantage in calculations cost.

2.2 b) Runge-Kutta method of 4^{th} order with variable step size automatically adjusted

2.2.1 Problem

We are given following equations:

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

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

And we have to determine the trajectory of the motion on interval $[0, 15]$ with following initial conditions: $x_1(0) = 8; x_2(0) = 9$ In this section we will use Runge-Kutta method of 4^{th} order with step size automatically adjusted by the algorithm, with error estimation made according to the step-doubling rule.

2.2.2 Theoretical Introdution

Since Runge-Kutta method was explained in previous task I will focus on step-doubling rule. Everytime we increase step size we receive less accurate results at the benefit of less calculations cost, this means that we need to have reasonable approach to setting step size to maximise accuracy and minimize calculations cost. We will use step-doubling rule in this example.

Error estimation using step doubling-approach

For every step of size h we perform two additional steps of size $\frac{h}{2}$. We denote:
y_n^1 as a new point obtained using the original step-size h
y_n^2 as a new point obtained using steps of the size $\frac{h}{2}$
$r^{(1)}$ - approximation error affter the single step h
$r^{(2)}$ - summed approximation errors after the two smaller steps of length $\frac{h}{2}$ each.
We have:

After a single step:

$$y(x_n + h) = y_n^{(1)} + \frac{r_n^{p+1}(0)}{(p+1)!}h^{p+1} + O(h^{p+2})$$

After a double step:

$$y(x_n + h) \simeq y_n^{(2)} + 2\frac{r_n^{p+1}(0)}{(p+1)!}(\frac{h}{2})^{p+1} + O(h^{p+2})$$

We evaluate unknown coefficient from the first equation $\frac{r_n^{p+1}(0)}{(p+1)!}$ and insert it into second one:

$$y(x_n + h) = y_n^{(2)} + \frac{h^{p+1}}{2^p}\frac{(x_n + h) - y_n^{(1)}}{h^{p+1} + O(h^{p+2})}$$

We continue further and receive:

$$y(x_n + h)(1 - \frac{1}{2^p}) = y_n^{(2)}(1 - \frac{1}{2^p}) + \frac{y_n^2}{2^p} - \frac{y_n^{(1)}}{2^p} + O(h^{p+2})$$

We multiply by $\frac{2^p}{2^p-1}$ and get:

$$y(x_n + h) = y_n^{(2)} + \frac{y_n^{(2)} - y_n^{(1)}}{2^p - 1} + O(h^{p+2}) \tag{2.1}$$

We also obtain for first equation in a similar way:

$$y(x_n + h) = y_n^{(1)} + 2^p\frac{y_n^{(2)} - y_n^{(1)}}{2^p - 1} + O(h^{p+2})$$

Assuming we take the main part of the error we get error estimate:

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

But from the equation (2.1) we can treat expression:

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

as en estimate of the error of two consecutive steps of size $\frac{h}{2}$

Actual correction of the step size

In general formula for main part of the approximation error where:
h - length of a step

$$\gamma = \frac{r_n^{(p+1)}(0)}{(p+1)!}$$

Looks like this:

$$\delta_n(h) = \gamma h^{p+1}$$

If we change step-size to αh then we get:

$$\delta_n(\alpha h) = \gamma(\alpha h)^{p+1}$$

Then:

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

We assume tolerance ϵ:

$$|\delta_n(\alpha h)| = \epsilon$$

We get:

$$\alpha^{p+1}|\delta_n(h)| = \epsilon$$

Coefficient α for the step-size correction can be evaluated in such a way:

$$\alpha = (\frac{\epsilon}{|\delta_n(h)|})^{\frac{1}{p+1}}$$

This method is also correct for RK4 method when $\delta_n(h) = \delta_n(2 \times \frac{h}{2})$ We should also take into considertaion lack of accuracy we do so by using safety factor s:

$$h_{n+1} = s\alpha h_n$$

where $s < 1$ For RK4 we use $s \approx 0.9$
We define accuracy parameters in such a way:

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

where ϵ_r is relative tolerance and ϵ_a is absolute tolerance.
In case of set of more than on differential equations we must use worst case approach so the equations that need smallest step-size define step-size for all other equations.

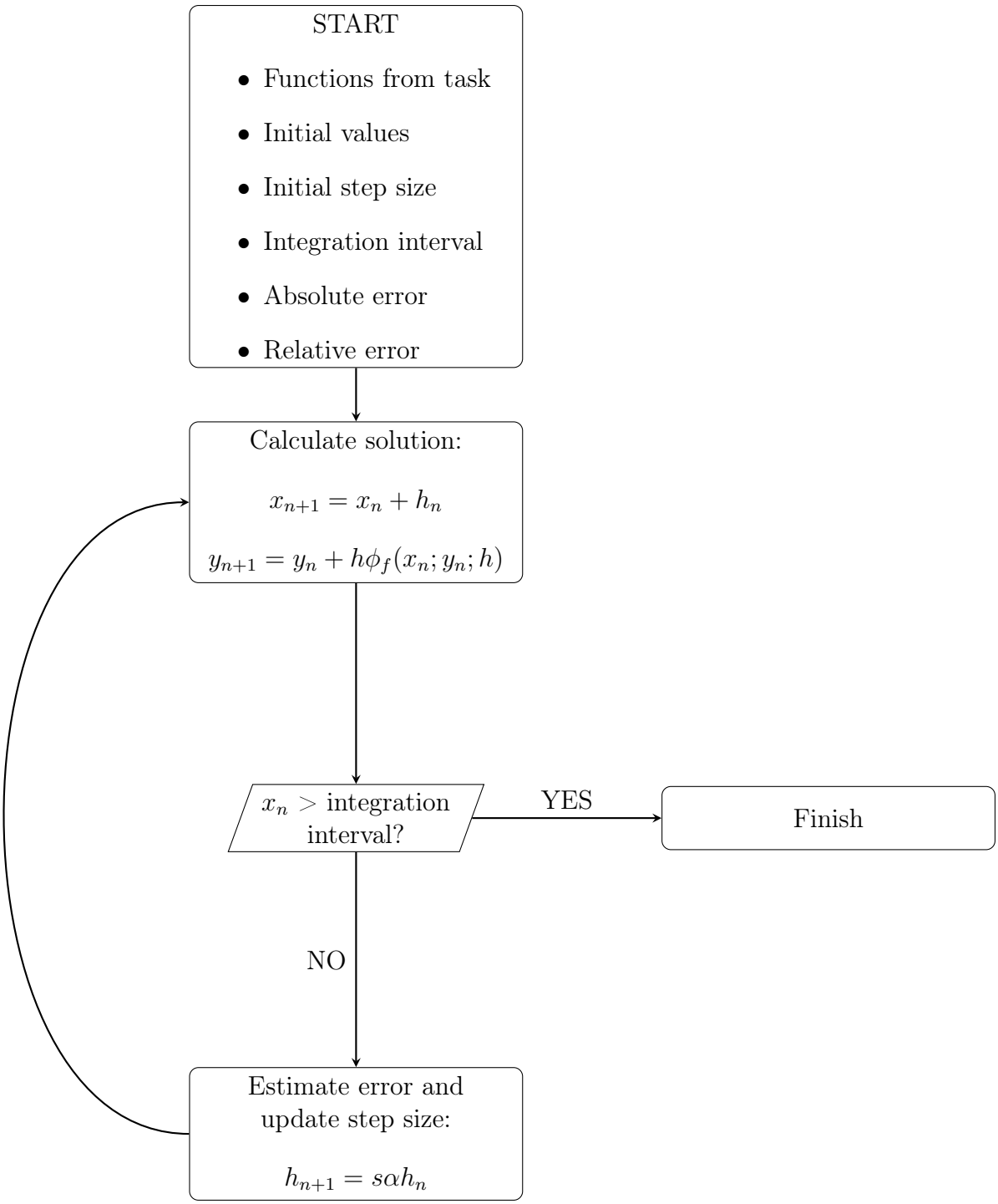
For RK methods we have following parameters:

$$\delta_n(h)_i = \frac{(y_i)_n^{(2)} - (y_i)_n^{(1)}}{2^p - 1}$$

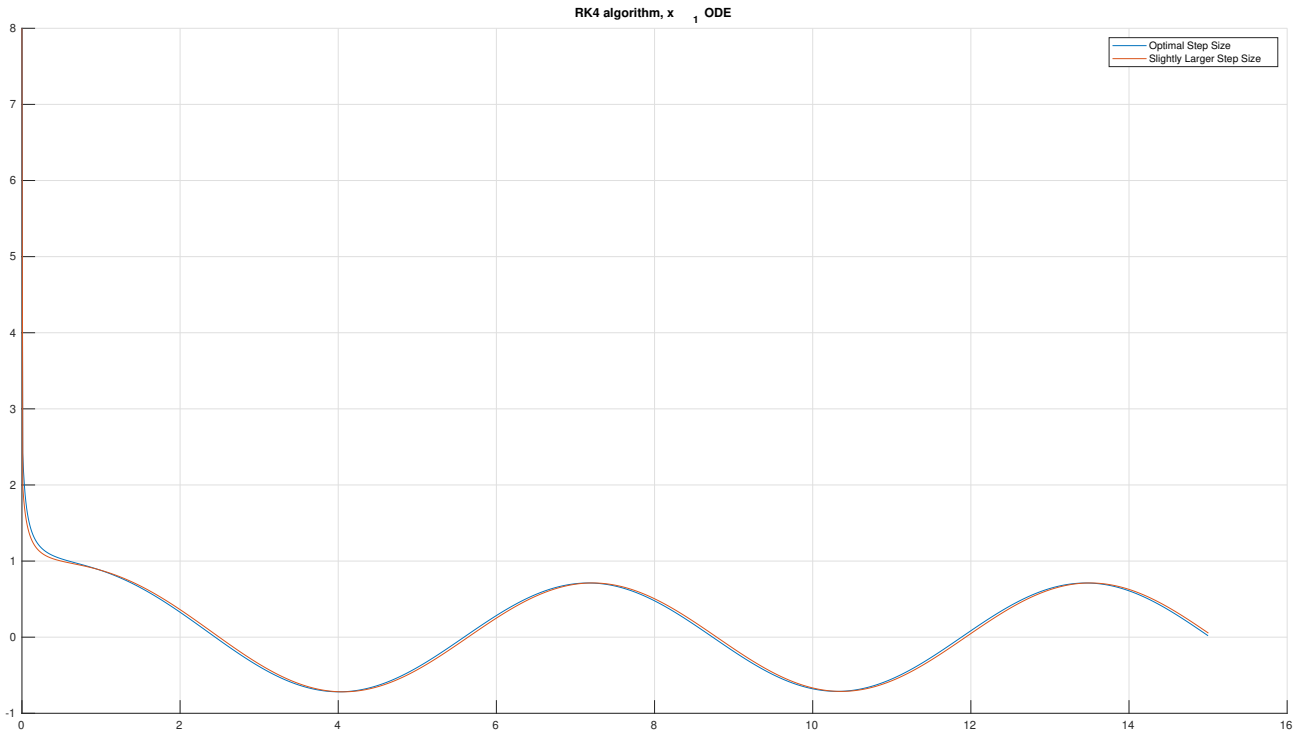
$$\epsilon_i = |(y_i)_n^{(2)}|\epsilon_r + \epsilon_a$$

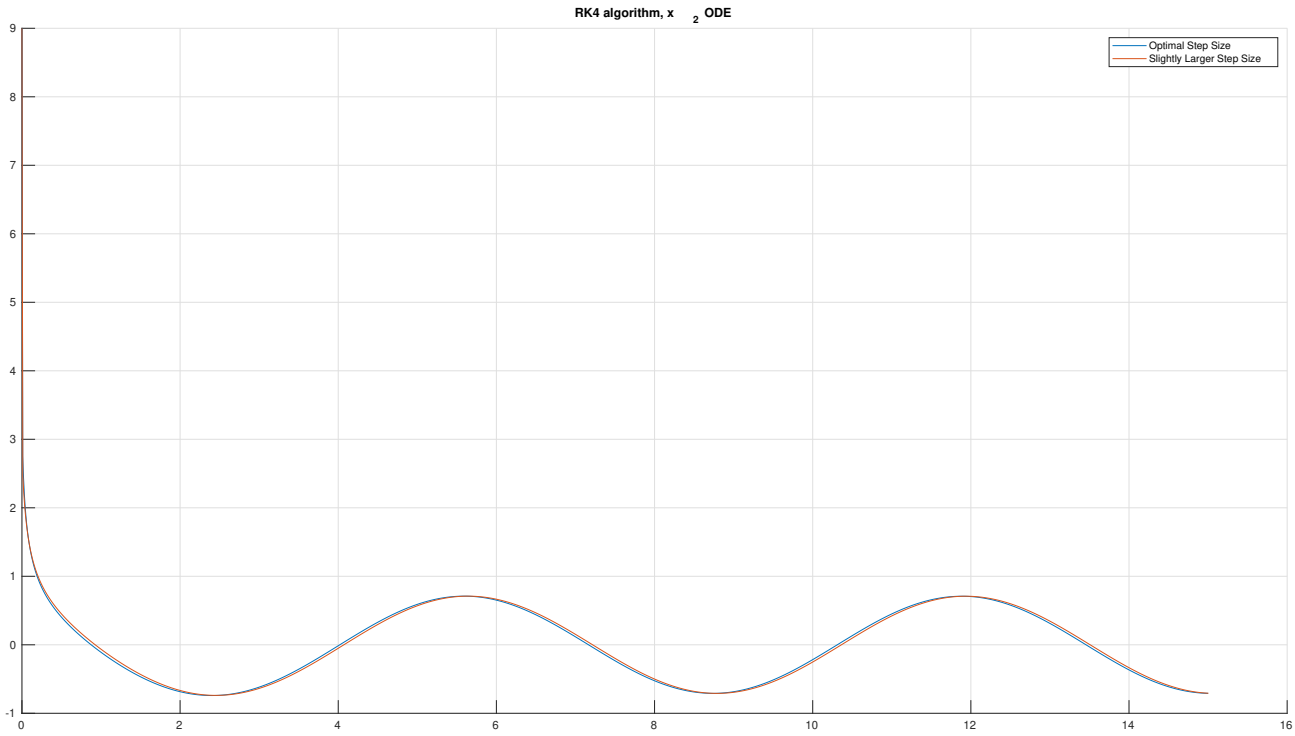
$$\alpha = \min_{1 \le i \le m} (\frac{\epsilon_i}{|\delta_n(h)_i|})^{\frac{1}{p+1}}$$

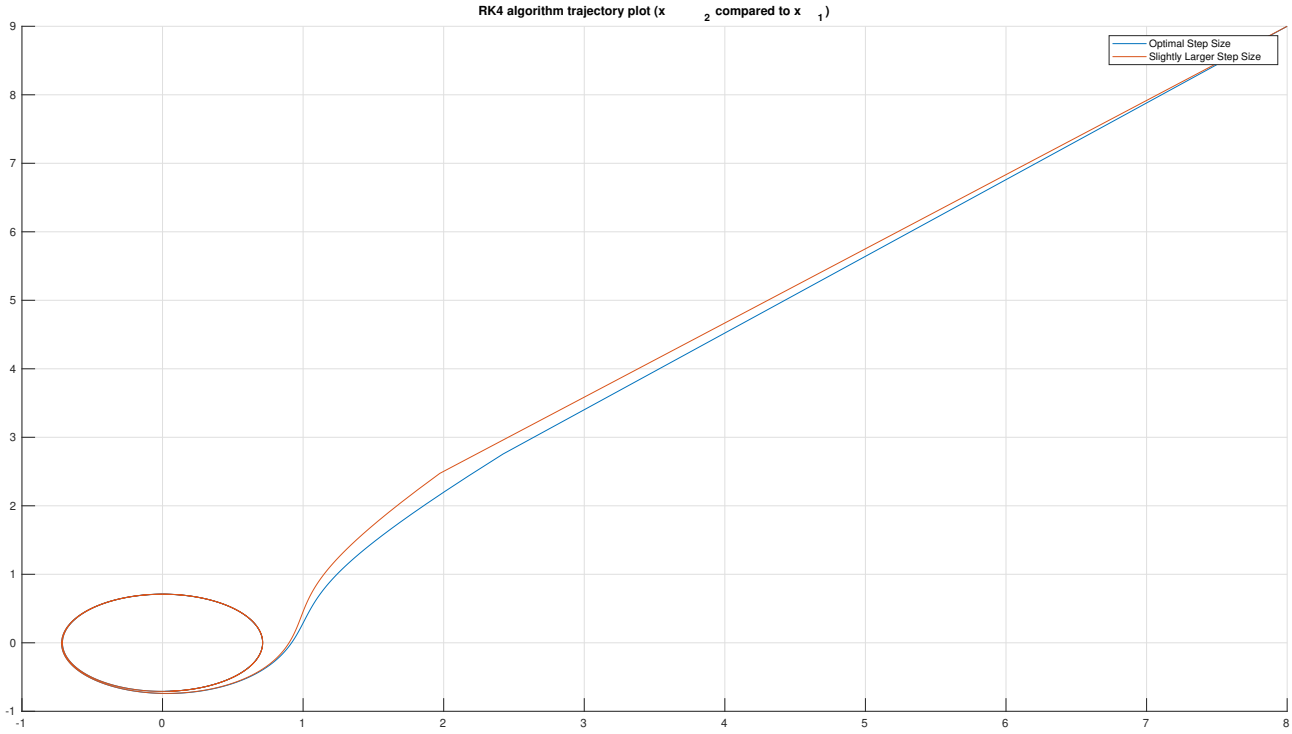
2.2.3 Flow diagram

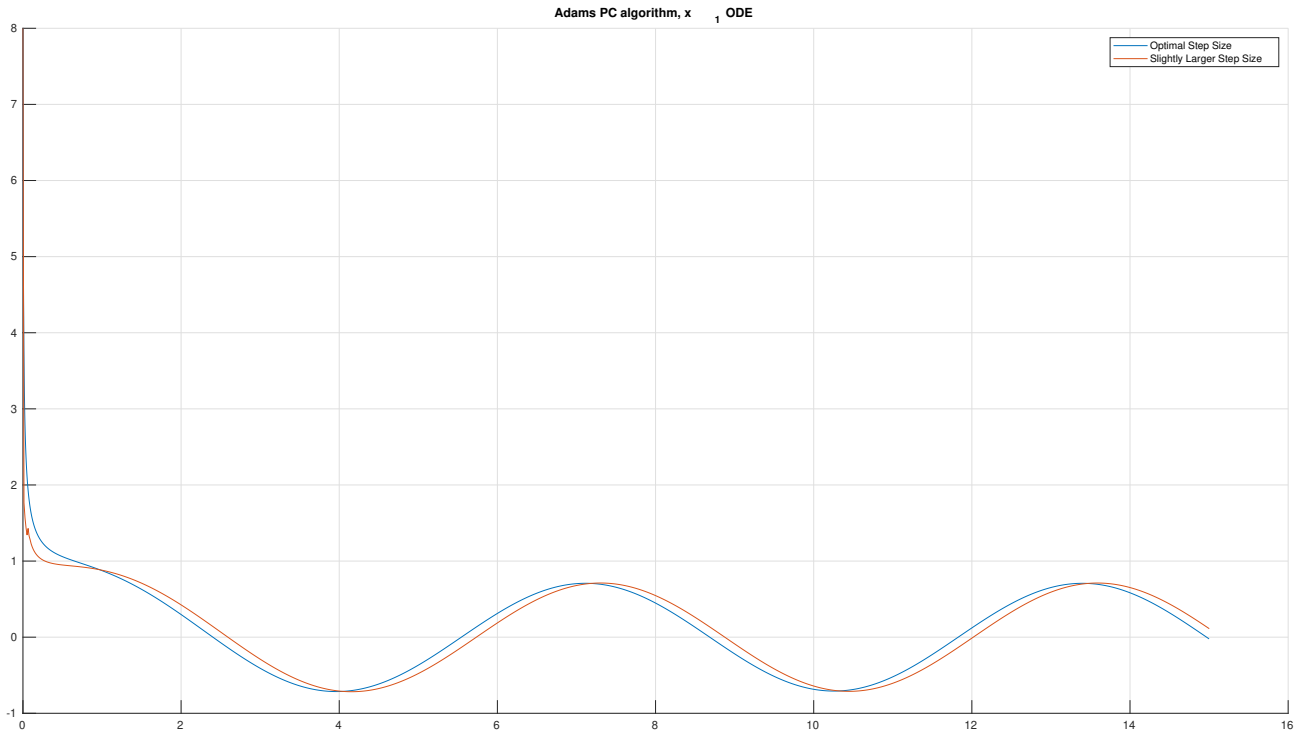


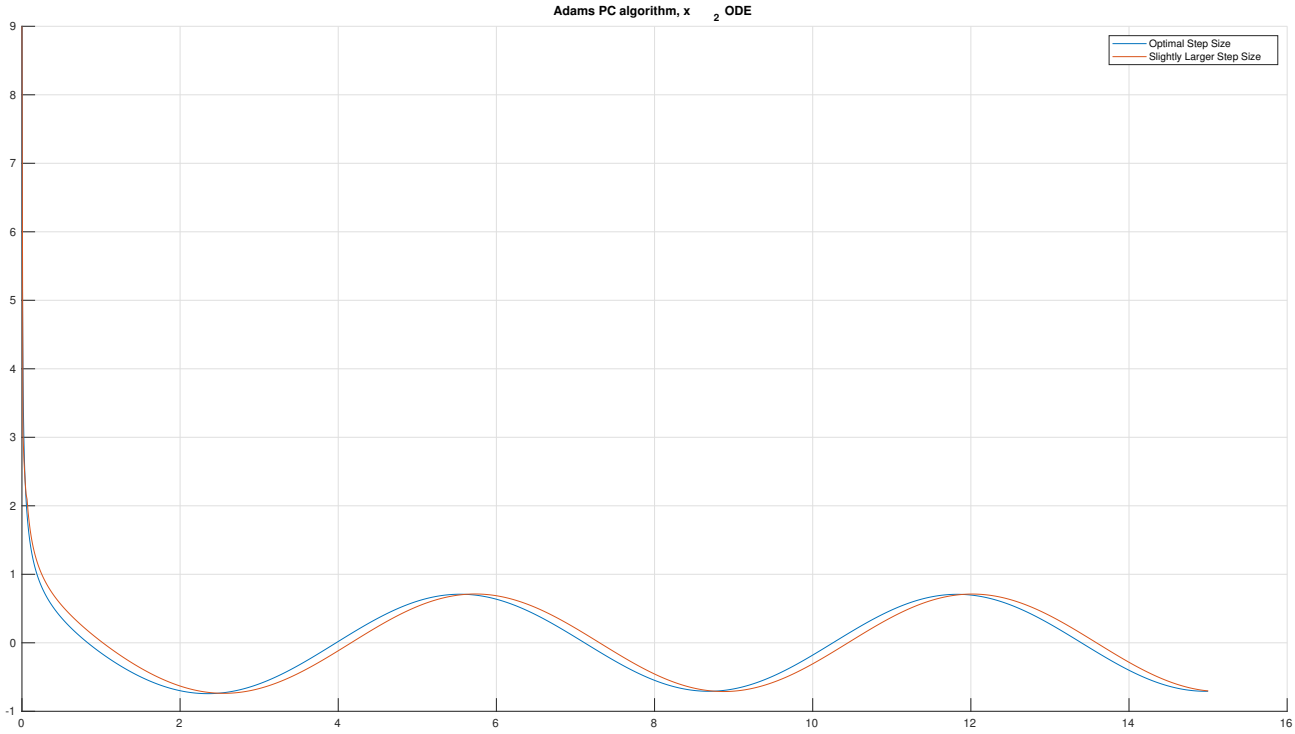
2.3 Results

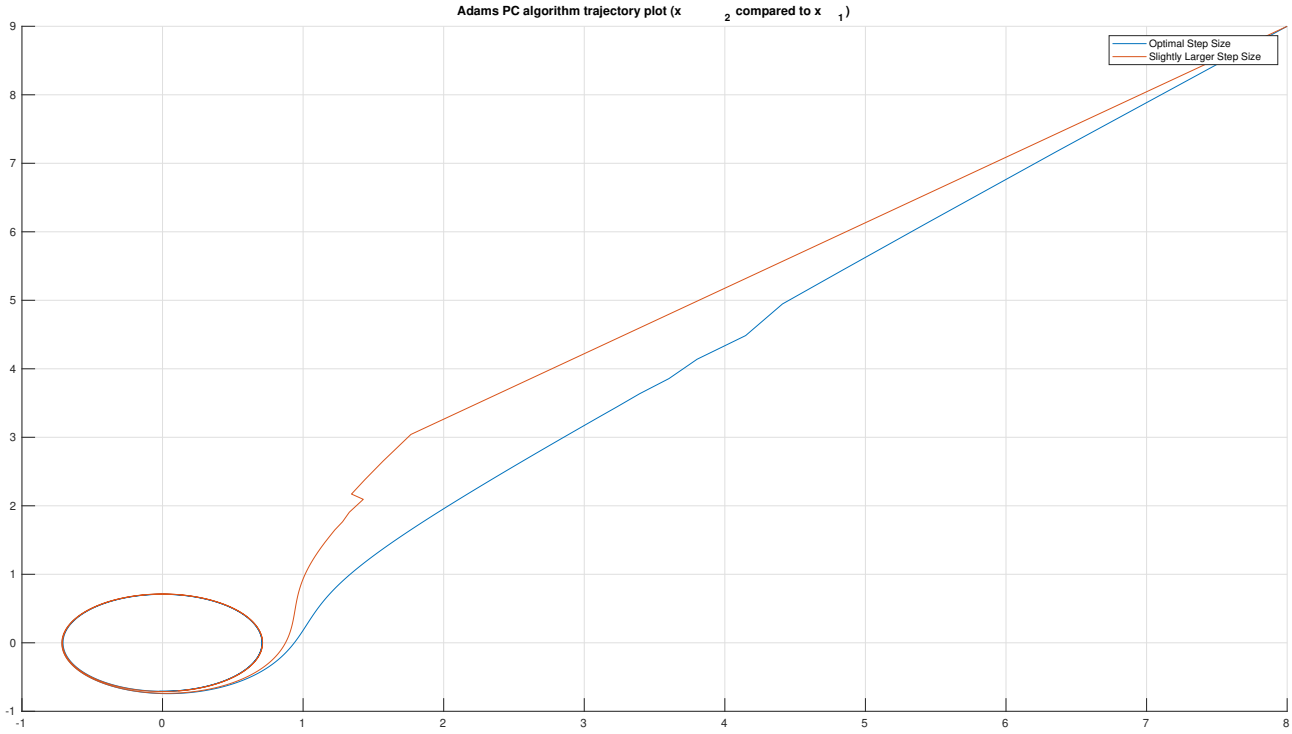

RK4 algorithm, x 1 ODE
Optimal Step Size
Slightly Larger Step Size
8
7
6
5
4
3
2
1
0
-1
0
2
4
6
8
10
12
14
16

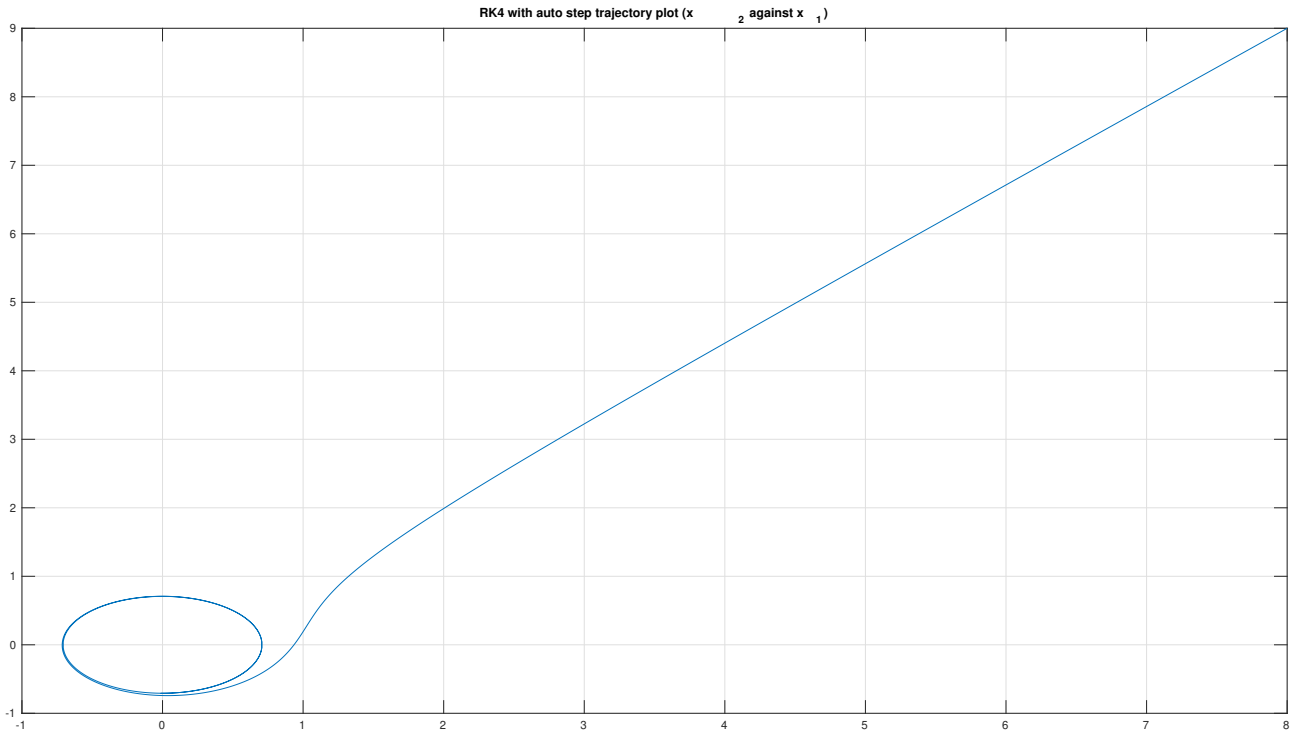

RK4 algorithm, x 2 ODE
Optimal Step Size
Slightly Larger Step Size
9
8
7
6
5
4
3
2
1
0
-1
0
2
4
6
8
10
12
14
16

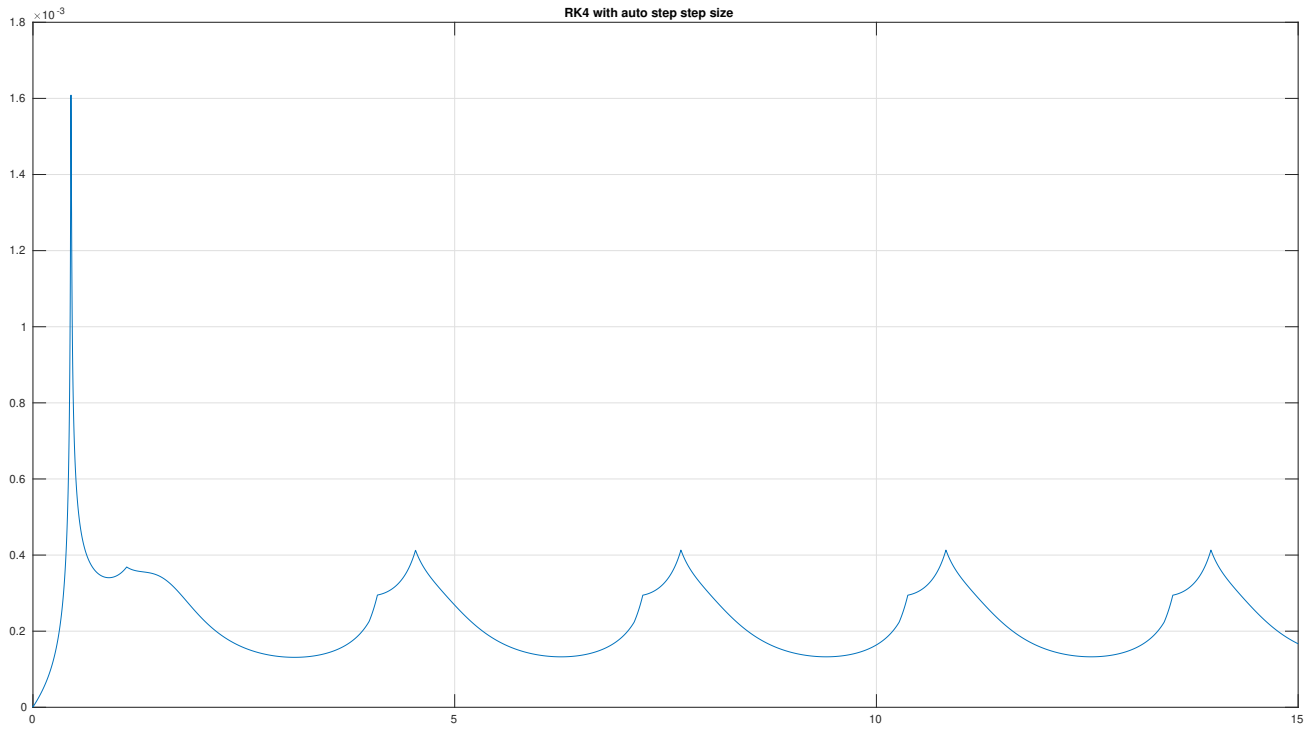

RK4 algorithm trajectory plot (x 2 compared to x 1)
Optimal Step Size
Slightly Larger Step Size
9
8
7
6
5
4
3
2
1
0
-1
-1
0
1
2
3
4
5
6
7
8

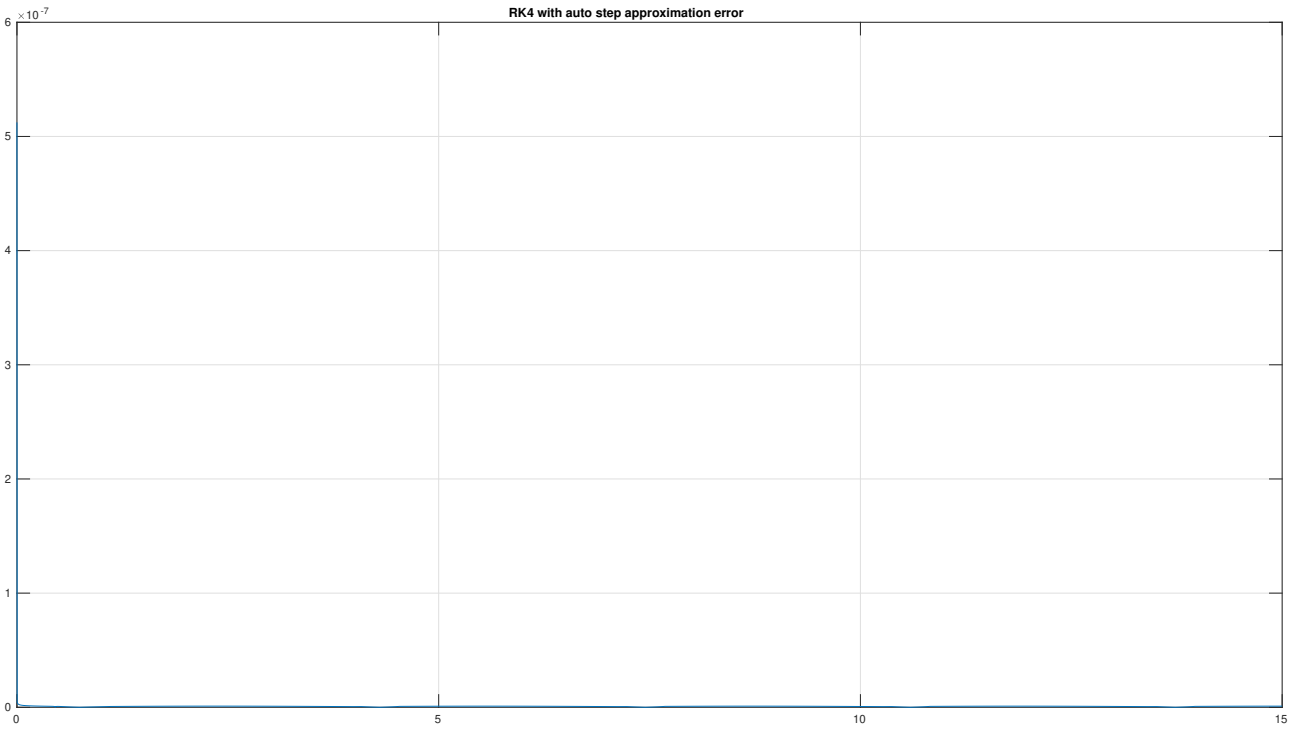

Adams PC algorithm, x 1 ODE
Optimal Step Size
Slightly Larger Step Size
8
7
6
5
4
3
2
1
0
-1
0
2
4
6
8
10
12
14
16

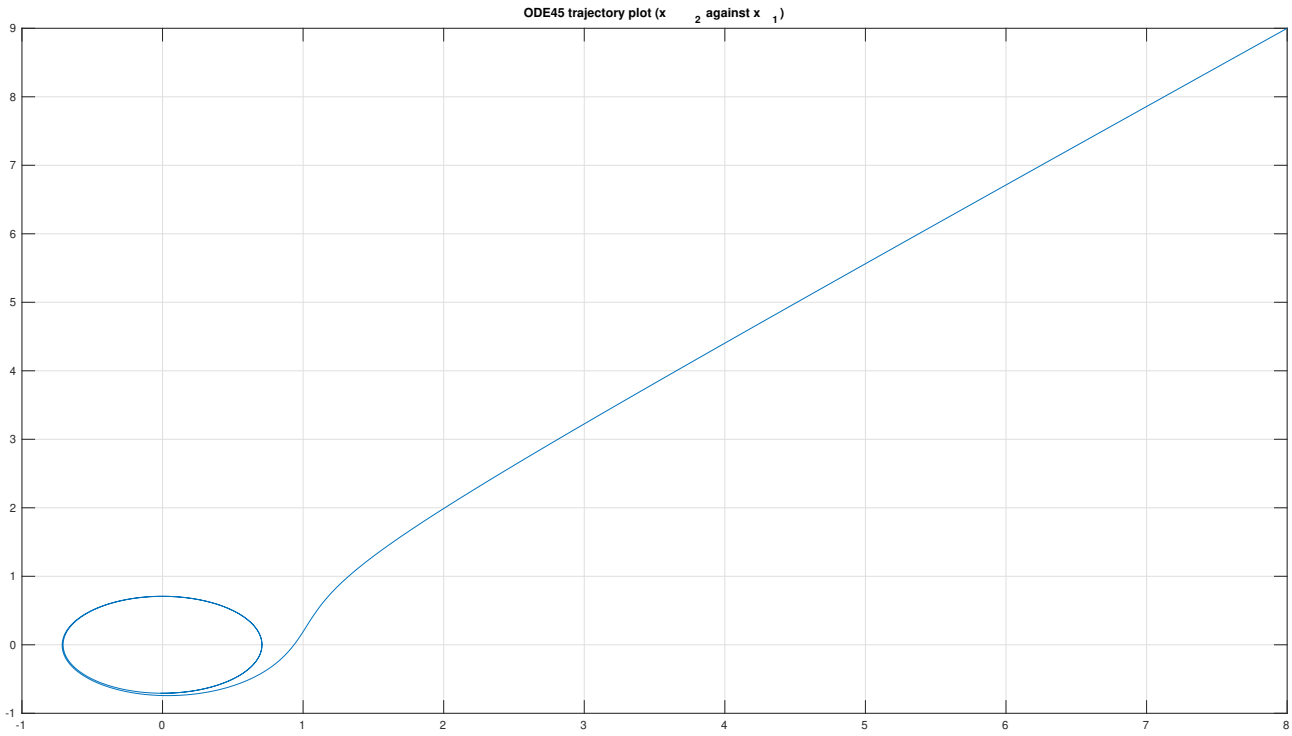

Adams PC algorithm, x 2 ODE
Optimal Step Size
Slightly Larger Step Size
9
8
7
6
5
4
3
2
1
0
-1
0
2
4
6
8
10
12
14
16


Adams PC algorithm trajectory plot (x 2 compared to x 1)
Optimal Step Size
Slightly Larger Step Size
9
8
7
6
5
4
3
2
1
0
-1
-1
0
1
2
3
4
5
6
7
8


RK4 with auto step trajectory plot (x 2 against x 1)
9
8
7
6
5
4
3
2
1
0
-1
-1
0
1
2
3
4
5
6
7
8


RK4 with auto step step size
×10^-3
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
5
10
15


RK4 with auto step approximation error
×10^-7
6
5
4
3
2
1
0
0
5
10
15


ODE45 trajectory plot (x 2 against x 1)
9
8
7
6
5
4
3
2
1
0
-1
-1
0
1
2
3
4
5
6
7
8

2.4 Discussion of results

As we can see using RK4 algorithm with auto step size results in best solutions error-wise, trajectory plot is almost the same as the one generated by matlab function. Especially graph of step approximation error for RK4 auto step algorithm looks very good with errors as low as 10^{-7}. Adam PC method on the other hand suffers from bigger accuarcy loss though the results are still satisfactory while requiring much less work on the coding part.

Chapter 3

Code appendix

3.1 functionDataPoints.m

3.1.1 functionDataPoints

```
function dataPoints = functionDataPoints()
    dataPoints = (-5:5)';
    dataPoints(:, 2) = [
        -6.5743
        0.9765
        3.1026
        1.8572
        1.3165
        -0.6144
        0.1032
        0.3729
        2.5327
        7.3857
        9.4892
    ];
end
```

3.2 task1.m

3.2.1 task1

```
function task1(polynomialDegree)
    for currentPolynomialDegree = 0 : polynomialDegree
        dataPoints = functionDataPoints();
        % obtain LSPSolutions of approximating polynomial
        [LSPSolutions, error, conditionNumber] = approximate(
            dataPoints, currentPolynomialDegree);
        displayInfo(currentPolynomialDegree, error,
            conditionNumber);
        plotGraph(currentPolynomialDegree, dataPoints,
            LSPSolutions)
    end
end
```

3.2.2 displayInfo

```
function displayInfo(currentPolynomialDegree, error,
   conditionNumber)
    disp(Current Polynomial Degree:);
    disp(currentPolynomialDegree);
    disp(Error:  );
    disp(error);
    disp(Condition number of Gram's Matrix:  );
    disp(conditionNumber);
end
```

3.2.3 plotGraph

```
function plotGraph(currentPolynomialDegree, dataPoints,
   LSPSolutions)
    figure;
    grid on;
    hold on;
```

```
    plotDataPoints(currentPolynomialDegree, dataPoints);
    plotApproximation(dataPoints, LSPSolutions);
    hold off;
end
```

3.2.4 plotDataPoints

```
function plotDataPoints(currentPolynomialDegree, dataPoints)
    title(['Polynomial approximation of the funtion using: ',
       num2str(currentPolynomialDegree), ' degree']);
    scatter(dataPoints(:, 1), dataPoints(:, 2), 72, 'x');
end
```

3.2.5 plotApproximation

```
function plotApproximation(dataPoints, LSPSolutions)
    x = dataPoints(1):0.05:dataPoints(end, 1);
    y = valueApproximationAtx(LSPSolutions, x);
    plot(x, y);
end
```

3.2.6 approximate

```
% find the approximating polynomial of the given degree
function [LSPSolutions, error, conditionNumber] = approximate(
   dataPoints, polydeg)
    % initialize A matrix
    A = zeros(size(dataPoints, 1), polydeg + 1);

    A = calculateACells(dataPoints, A);
    LSPSolutions = solveLSP(A, dataPoints);

    % Calculate error of solution
    error = norm(dataPoints(:, 2) - A * LSPSolutions);
    % Calculate condition number of Gram's matrix
```

```
12      GramsMatrix = A' * A;
13      conditionNumber = cond(GramsMatrix);
14  end
```

3.2.7 calculateACells

```
1   function Matrix = calculateACells(dataPoints, Matrix)
2       for row = 1 : size(Matrix, 1)
3           for column = 1 : size(Matrix, 2)
4               Matrix(row, column) = dataPoints(row, 1) ^ (column
                    - 1);
5           end
6       end
7   end
```

3.2.8 solveLSP

```
1   function solutions = solveLSP(Matrix, dataPoints)
2       [Q, eqsys, invqtq] = QRDecomposition(Matrix);
3       eqsys(:, end + 1) = invqtq * Q' * dataPoints(:, 2);
4       solutions = backSubstitution(eqsys);
5   end
```

3.2.9 valueApproximationAtx

```
1   function arrayOfValues = valueApproximationAtx(LSPSolutions,
        arrayOfArguments)
2       arrayOfValues = zeros(1, size(arrayOfArguments, 2));
3       arrayOfValues = calculateArrayofValues(arrayOfArguments,
            arrayOfValues, LSPSolutions);
4   end
```

3.2.10 calculateArrayofValues

```
function arrayOfValues = calculateArrayofValues(
   arrayOfArguments, arrayOfValues, LSPSolutions)
    for x = 1 : size(arrayOfArguments, 2)
        for i = 1 : size(LSPSolutions, 1)
            arrayOfValues(x) = arrayOfValues(x) + LSPSolutions(
               i) * arrayOfArguments(x) ^ (i - 1);
        end
    end
end
```

3.3 task2.m

3.3.1 task2

```
function task2()
    [ordinaryDifferentialEquations, initialValues, interval,
       algorithms] = initialize();
    solveAndPrintODEs(algorithms, ordinaryDifferentialEquations
       , initialValues, interval);
    [result, sizes, errors] = solveAndPlotODEAutomatic(
       ordinaryDifferentialEquations, initialValues, interval);
    plotStatistics(result, sizes, errors);
    plotComparisonToMatlabFunction(
       ordinaryDifferentialEquations, interval, initialValues);
end
```

3.3.2 initialize

```
function [ordinaryDifferentialEquations, initialValues,
   interval, algorithms] = initialize()
    ordinaryDifferentialEquations = {
        @(x) x(2) + x(1) * (0.5 - x(1)^2 - x(2)^2);
        @(x) -x(1) + x(2) * (0.5 - x(1)^2 - x(2)^2)
```

```
    };
    initialValues = [8; 9];
    interval = [0; 15];

    algorithms = {
        'RK4 algorithm', @RK4, [0.01, 0.011];
        'Adams PC algorithm', @AdamsPCMethod, [0.002, 0.013]
    };
end
```

3.3.3 solveAndPrintODEs

```
function solveAndPrintODEs(algorithms,
   ordinaryDifferentialEquations, initialValues, interval)
    for algorithm = 1 : 2
        [algorithmName, algorithmFunction, stepSizes] =
           algorithms{algorithm, :};
        stepResults = solveForEachStep(stepSizes,
           ordinaryDifferentialEquations, initialValues,
           interval, algorithmFunction);
        plotAgainstTime(ordinaryDifferentialEquations,
           algorithmName, stepResults);
        plotAgainst(algorithmName, stepResults);

    end
end
```

3.3.4 solveForEachStep

```
function stepResults = solveForEachStep(stepSizes,
   ordinaryDifferentialEquations, initialValues, interval,
   algorithmFunction)
    stepResults = cell(size(stepSizes, 2), 3);
    stepNames = {'Optimal Step Size', 'Slightly Larger Step
       Size'};
    for stepNumber = 1:size(stepSizes, 2)
```

```
        result = algorithmFunction(
            ordinaryDifferentialEquations, initialValues,
            interval, stepSizes(stepNumber));
        stepResults(stepNumber, :) = {stepSizes(stepNumber),
            stepNames{stepNumber}, result};
    end
end
```

3.3.5 beginPlot

```
function beginPlot()
    figure;
    grid on;
    hold on;
end
```

3.3.6 plotAgainstTime

```
function plotAgainstTime(ordinaryDifferentialEquations,
    algorithmName, stepResults)
    for equationNumber = 1:size(ordinaryDifferentialEquations,
        1)
        beginPlot();

        title([algorithmName, ', x_', num2str(equationNumber),
            ' ODE']);

        for stepresult = stepResults'
            plot(stepresult{3}(1, :), stepresult{3}(
                equationNumber + 1, :));
        end

        hold off;
        legend(stepResults{:, 2});
    end
end
```

3.3.7 plotAgainst

```
function plotAgainst(algorithmName, stepResults)
    beginPlot();
    title([algorithmName, ' trajectory plot (x_2 compared to
       x_1)']);
    for stepresult = stepResults'
        plot(stepresult{3}(2, :), stepresult{3}(3, :));
    end
    hold off;
    legend(stepResults{:, 2});
%    %print(['report/', func2str(algfunc), 'traj'], '-dpdf');
end
```

3.3.8 solveAndPlotODEAutomatic

```
function [result, sizes, errors] = solveAndPlotODEAutomatic(
   ordinaryDifferentialEquations, initialValues, interval)
    [result, sizes, errors] = solveRKAutomatic(
       ordinaryDifferentialEquations, initialValues, interval);
    plotTrajectory(result);
end
```

3.3.9 solveRKAutomatic

```
function [result, sizes, errors] = solveRKAutomatic(
   ordinaryDifferentialEquations, initialValues, interval)
    initialStepSize = 1e-5;
    relativeEpsilon = 1e-9;
    absoluteEpsilon = 1e-9;
    [result, sizes, errors] = RK4Automatic(
       ordinaryDifferentialEquations, initialValues, interval,
       initialStepSize, relativeEpsilon, absoluteEpsilon);
end
```

3.3.10 plotTrajectory

```
function plotTrajectory(result)
    figure;
    plot(result(2, :), result(3, :));
    grid on;
    title('RK4 with auto step trajectory plot (x_2 against x_1)
       ');
end
```

3.3.11 plotStatistics

```
function plotStatistics(result, sizes, errors)
    stats = {
        RK4 with auto step step size, rk4sizes, sizes;
        RK4 with auto step approximation
           error, rk4errors, errors
    };
    for stat = stats'
        figure;
        plot(result(1, 2:(end - 1)), stat{3});
        grid on;
        title(stat{1});
    end
end
```

3.3.12 plotComparisonToMatlabFunction

```
function plotComparisonToMatlabFunction(
   ordinaryDifferentialEquations, interval, initialValues)
    % compare results with ODE45
    odefun = @(t, x) [ ordinaryDifferentialEquations{1}(x);
       ordinaryDifferentialEquations{2}(x) ];
    odeoptions = odeset('RelTol', 10e-10, 'AbsTol', 10e-10);
    [~, x] = ode45(odefun, interval, initialValues, odeoptions)
       ;
```

```
    figure;
    plot(x(:, 1), x(:, 2));
    grid on;
    title('ODE45 trajectory plot (x_2 against x_1)');
end
```

3.4 AdamsPCMethod.m

3.4.1 AdamsPCMethod

```
function x = AdamsPCMethod(functions, initialValues, interval,
   stepSize)
    [x, derrivatives, explicitCoefficients,
       implicitCoefficients, stepCount] = initialize(functions,
        initialValues, interval, stepSize);
    x = adamsPcLoop(x, derrivatives, explicitCoefficients,
       implicitCoefficients, stepCount, functions, stepSize);
    % append arguments to output
    x = [interval(1):stepSize:(stepCount * stepSize); x];
end
```

3.4.2 initialize

```
function [x, derrivatives, explicitCoefficients,
   implicitCoefficients, stepCount] = initialize(functions,
   initialValues, interval, stepSize)
    % obtain first five steps from RK4
    [x, derrivatives] = RK4(functions, initialValues, interval,
        stepSize, 5);
    x = x(2:end, :);

    % define coefficient
    explicitCoefficients = [1901, -2774, 1616, -1274, 251] /
       720;
    % Constants that can  be found on the Internet or in Mister
        Tatjewski
```

```
    % book on page 177, notice how beta(3) = 1616 instead of 2616
    % I have found on the internet different value for this parameter and
    % got better results with it so I decided to stick with it
    implicitCoefficients = [475, 1427, -798, 482, -173, 27] / 1440;
    % Constants that can  be found on the Internet or in Mister Tatjewski
    % book on page 178
    % build output based on preceding values
    stepCount = ceil((interval(2) - interval(1)) / stepSize);
end
```

3.4.3 adamsPcLoop

```
function x = adamsPcLoop(x, derrivatives, explicitCoefficients, implicitCoefficients, stepCount, functions, stepSize)
    for step = 6: (stepCount + 1)
        [x, derrivatives] = PECE(x, derrivatives, explicitCoefficients, implicitCoefficients, step, functions, stepSize);
    end
end
```

3.4.4 PECE

```
function [x, derrivatives] = PECE(x, derrivatives, explicitCoefficients, implicitCoefficients, step, functions, stepSize)
    % P
    predictionOfX = adamsPredict(x, step, stepSize, explicitCoefficients, derrivatives);

    % E
    derrivativePrediction = zeros(size(functions, 1), 1);
```

```
    derrivativePrediction = adamsEvaluate(functions,
        predictionOfX, derrivativePrediction);

    % C
    x = adamsCorrect(step, functions, stepSize,
        implicitCoefficients, derrivatives,
        derrivativePrediction, x);

    % E
    derrivatives = adamsEvaluateTwo(functions, step, x,
        derrivatives);
end
```

3.4.5 adamsPredict

```
function predictionOfX = adamsPredict(x, step, stepSize,
    explicitCoefficients, derrivatives)
    % predict
    predictionOfX = x(:, step - 1);
    for equationNumber = 1 : 2
        for previous = 1:5
            predictionOfX(equationNumber) = predictionOfX(
                equationNumber) ...
                + stepSize * explicitCoefficients(previous) *
                    derrivatives(equationNumber, step - previous
                    );
        end
    end
end
```

3.4.6 adamsEvaluate

```
function derrivativePrediction = adamsEvaluate(functions,
    predictionOfX, derrivativePrediction)
    for equationNumber = 1:size(functions, 1)
```

```
        derrivativePrediction(equationNumber) = functions{
            equationNumber}(predictionOfX);
    end
end
```

3.4.7 adamsCorrect

```
function x = adamsCorrect(step, functions, stepSize,
    implicitCoefficients, derrivatives, derrivativePrediction, x
    )
    x(:, step) = x(:, step - 1);
    for equationNumber = 1:size(functions, 1)
        for previous = 1:5
            x(equationNumber, step) = x(equationNumber, step) +
                stepSize * implicitCoefficients(previous + 1) *
                derrivatives(equationNumber, step - previous);
        end
        x(equationNumber, step) = x(equationNumber, step) +
            stepSize * implicitCoefficients(1) *
            derrivativePrediction(equationNumber);
    end
end
```

3.4.8 adamsEvaluateTwo

```
function derrivatives = adamsEvaluateTwo(functions, step, x,
    derrivatives)
    for equationNumber = 1:size(functions, 1)
        derrivatives(equationNumber, step) = functions{
            equationNumber}(x(:, step));
    end
end
```

3.5 QRDecomposition.m

3.5.1 QRDecomposition

```
function [Q, R, QTQInverse] = QRDecomposition(A)
    [Q, R, QTQInverse, upperLoopLimit] = initialize(A);
    [Q, R, QTQInverse] = GramSchmidtAlgorithm(A, Q, R,
       QTQInverse, upperLoopLimit);
end
```

3.5.2 initialize

```
function [Q, R, QTQInverse, upperLoopLimit] = initialize(Matrix
   )
    Q = zeros(size(Matrix));
    upperLoopLimit = size(Matrix, 2);
    R = eye(upperLoopLimit);
    QTQInverse = zeros(upperLoopLimit);
end
```

3.5.3 GramSchmidtAlgorithm

```
function [Q, R, QTQInverse] = GramSchmidtAlgorithm(A, Q, R,
   QTQInverse, upperLoopLimit)
    for column = 1 : upperLoopLimit
        [Q, R, QTQInverse, A] = GramSchmidtAlgorithmOuterLoop(
           upperLoopLimit, A, Q, R, QTQInverse, column);
    end
end
```

3.5.4 GramSchmidtAlgorithmOuterLoop

```
function [Q, R, QTQInverse, A] = GramSchmidtAlgorithmOuterLoop(
   upperLoopLimit, A, Q, R, QTQInverse, column)

```

```
    [columnDotProduct, QTQInverse, Q] =
        calculateColumnDotProduct(A, column, Q, QTQInverse);
    [R, A] = orthogonalizeFurther(column, A, columnDotProduct,
        Q, R, upperLoopLimit);
end
```

3.5.5 calculateColumnDotProduct

```
function [columnDotProduct, QTQInverse, Q] =
    calculateColumnDotProduct(A, column, Q, QTQInverse)
    Q(:, column) = A(:, column);

    columnDotProduct = dot(Q(:, column), Q(:, column));
    QTQInverse(column, column) = 1 / columnDotProduct;
end
```

3.5.6 orthogonalizeFurther

```
function [R, A] = orthogonalizeFurther(column, A,
    columnDotProduct, Q, R, upperLoopLimit)
    for next = (column + 1): upperLoopLimit
        R(column, next) = dot(Q(:, column), A(:, next)) /
            columnDotProduct;
        A(:, next) = A(:, next) - R(column, next) * Q(:, column
            );
    end
end
```

3.6 RK4.m

3.6.1 RK4

```
function [x, derivativesTable] = RK4(equations, initialValues,
    interval, stepSize, maxSteps)
```

```
    x = initialValues;

    derivativesTable = buildDerivatiesTable(x, equations);

    % Calculate stepCount
    stepCount = ceil((interval(2) - interval(1)) / stepSize);
    if nargin == 5
        stepCount = min(stepCount, maxSteps - 1);
    end % IF we include max steps in our function input
    % (nargin is number of arguments in input)
    % then choose smaller number between maxSteps and stepCount
        and choose
    % it for stepCount

    [x, derivativesTable] = rk4Loop(x, stepCount, stepSize,
       equations, derivativesTable);


    % append arguments to output
    x = [interval(1):stepSize:(stepCount * stepSize); x];
end
```

3.6.2 buildDerivatiesTable

```
function derivativesTable = buildDerivatiesTable(x, equations)
    derivativesTable = zeros(size(x));
    for eqnum = 1:size(equations, 1)
        derivativesTable(eqnum, 1) = equations{eqnum}(x(:, 1));
    end
end
```

3.6.3 rk4Loop

```
function [x, derivativesTable] = rk4Loop(x, stepCount, stepSize
   , equations, derivativesTable)
    for step = 1 : stepCount
```

```
        [x, derivativesTable] = rk4stepLoop(x, step, equations,
            stepSize, derivativesTable);
    end
end
```

3.6.4 rk4stepLoop

```
function [x, derivativesTable] = rk4stepLoop(x, step, equations
   , stepSize, derivativesTable)
    stepValue = x(:, step);
    [x, derivativesTable] = equationsLoop(x, equations,
       stepValue, stepSize, step, derivativesTable);
end
```

3.6.5 equationsLoop

```
function [x, derivativesTable] = equationsLoop(x, equations,
   stepValue, stepSize, step, derivativesTable)
    for equationNumber = 1 : 2
        Phi = RK4Phi(equations{equationNumber}, stepValue,
           stepSize);
        x(equationNumber, step + 1) = x(equationNumber, step) +
            stepSize * Phi;

        derivativesTable(equationNumber, step + 1) = equations{
           equationNumber}(x(:, step + 1));
    end
end
```

3.7 RK4Automatic.m

3.7.1 RK4Automatic

```
function [x, sizes, errors] = RK4Automatic(equations,
   initialValues, interval, initialStepSize, relativeEpsilon,
   absoluteEpsilon)
    [functionArguments, x, sizes, errors, stepSize, step, flag]
        = Initialize(interval, initialValues, initialStepSize);
    [functionArguments, x, sizes, errors] = RK4AutomaticLoop(
       functionArguments, x, sizes, errors, stepSize, step,
       equations, relativeEpsilon, absoluteEpsilon, interval,
       flag);
    x = [functionArguments; x];
end
```

3.7.2 Initialize

```
function [functionArguments, x, sizes, errors, stepSize, step,
   flag] = Initialize(interval, initialValues, initialStepSize)
    functionArguments = interval(1);
    x = initialValues;

    sizes = double.empty();
    errors = double.empty();

    stepSize = initialStepSize;
    step = 0;
    flag = 1;
end
```

3.7.3 RK4AutomaticLoop

```
function [functionArguments, x, sizes, errors] =
   RK4AutomaticLoop(functionArguments, x, sizes, errors,
   stepSize, step, equations, relativeEpsilon, absoluteEpsilon,
    interval, flag)
   while flag
       [functionArguments, x, sizes, errors, stepSize, step,
          interval, flag] = insideWhileLoop(functionArguments,
```

```
            x, sizes, errors, stepSize, step, equations,
            relativeEpsilon, absoluteEpsilon, interval, flag);
    end
end
```

3.7.4 insideWhileLoop

```
function [functionArguments, x, sizes, errors, stepSize, step,
    interval, flag] = insideWhileLoop(functionArguments, x,
    sizes, errors, stepSize, step, equations, relativeEpsilon,
    absoluteEpsilon, interval, flag)
     [step, stepValue, x, functionArguments, flag] =
         calculateXandFunctionArguments(step, x, equations,
         stepSize, functionArguments, interval);
     if ~flag
         return;
     end
     [stepSize, errors] = calculateStepAndErrors(equations,
         stepValue, stepSize, x, step, relativeEpsilon,
         absoluteEpsilon, errors);
     sizes(step) = stepSize;

end
```

3.7.5 calculateXandFunctionArguments

```
function [step, stepValue, x, functionArguments, flag] =
    calculateXandFunctionArguments(step, x, equations, stepSize,
     functionArguments, interval)
     [step, stepValue] = initializeSteps(step, x);
     x = equationsLoop(x, equations, stepValue, stepSize, step);
     [flag, functionArguments] = stopAlgorithm(functionArguments
         , stepSize, step, interval);
end
```

3.7.6 calculateStepAndErrors

```
function [stepSize, errors] = calculateStepAndErrors(equations,
    stepValue, stepSize, x, step, relativeEpsilon,
   absoluteEpsilon, errors)
    stepValue = calculateNextStep(equations, stepValue,
       stepSize);
    [stepCorrectionFactor, errors] = calculateStepCorrection(
       stepValue, x, step, relativeEpsilon, absoluteEpsilon,
       errors);
    stepSize = 0.9 * stepCorrectionFactor * stepSize;
end
```

3.7.7 initializeSteps

```
function [step, stepValue] = initializeSteps(step, x)
    step = step + 1;
    stepValue = x(:, step);
end
```

3.7.8 equationsLoop

```
function x = equationsLoop(x, equations, stepValue, stepSize,
   step)
    for equationNumber = 1 : 2
        Phi = RK4Phi(equations{equationNumber}, stepValue,
           stepSize);
        x(equationNumber, step + 1) = x(equationNumber, step) +
            stepSize * Phi;
    end
end
```

3.7.9 stopAlgorithm

```
function [flag, functionArguments] = stopAlgorithm(
   functionArguments, stepSize, step, interval)
    functionArguments(step + 1) = functionArguments(step) +
       stepSize;
    if functionArguments(end) >= interval(2)
        flag = 0;
    else
        flag = 1;
    end
end
```

3.7.10 calculateNextStep

```
function stepValue  = calculateNextStep(equations, stepValue,
   stepSize)
    for halfStep = 1 : 2
        for equationsNumber = 1:size(equations, 1)
            Phi = RK4Phi(equations{equationsNumber}, stepValue,
                stepSize / 2);
            stepValue(equationsNumber) = stepValue(
               equationsNumber) + (stepSize / 2) * Phi;
        end
    end
end
```

3.7.11 calculateStepCorrection

```
function [stepCorrectionFactor, errors] =
   calculateStepCorrection(stepValue, x, step, relativeEpsilon,
    absoluteEpsilon, errors)
    stepCorrectionFactor = Inf; % Initialize
       stepCorrectionFactor
    [stepCorrectionFactor, errors] =
       calculateStepCorrectionLoop(stepValue, x, step,
       relativeEpsilon, absoluteEpsilon, errors,
       stepCorrectionFactor);
```

```
    stepCorrectionFactor = stepCorrectionFactor ^ (1/5);
end
```

3.7.12 calculateStepCorrectionLoop

```
function [stepCorrectionFactor, errors] =
   calculateStepCorrectionLoop(stepValue, x, step,
   relativeEpsilon, absoluteEpsilon, errors,
   stepCorrectionFactor)
    for equationsNumber = 1 : 2
        approximationError = abs(stepValue(equationsNumber) - x
           (equationsNumber, step + 1)) / 15;
        errors(step) = approximationError;

        epsilon = abs(stepValue(equationsNumber)) *
           relativeEpsilon + absoluteEpsilon;
        equationAlpha = epsilon / approximationError;

        if equationAlpha < stepCorrectionFactor
            stepCorrectionFactor = equationAlpha;
        end
    end
end
```

3.8 backSubstitution.m

3.8.1 backSubstitution

```
function solution = backSubstitution(Matrix)
    Columns = size(Matrix, 1);
    Matrix = backSubstitutionOuterLoop(Matrix, Columns);
    % rightmost column of the Matrix is now our result
    solution = Matrix(:, size(Matrix, 2));
end
```

3.8.2 backSubstitutionOuterLoop

```
function Matrix = backSubstitutionOuterLoop(Matrix, Columns)
    for k = Columns : -1 : 1
        % Diagonal coefficients of matrix need to be equal to 1
        Matrix(k, :) = Matrix(k, :) / Matrix(k, k);
        Matrix = eliminateFactors(Matrix, k);
    end
end
```

3.8.3 eliminateFactors

```
function Matrix = eliminateFactors(Matrix, k)
    for row = (k - 1) : -1 : 1
        Matrix(row, :) = Matrix(row, :) - Matrix(k, :) * (
            Matrix(row, k) / Matrix(k, k));
    end
end
```

3.9 RK4Phi.m

3.9.1 RK4Phi

```
function Phi = RK4Phi(algorithm, stepValue, stepSize)
    k1 = algorithm(stepValue);
    k2 = algorithm(stepValue + 0.5 * stepSize * k1);
    k3 = algorithm(stepValue + 0.5 * stepSize * k2);
    k4 = algorithm(stepValue + stepSize * k3);
    Phi = (k1 + 2 * k2 + 2 * k3 + k4) / 6;
end
% calculates the Phi used in RK4 algorithms
```

Bibliography

[1] Piotr Tatjewski (2014) *Numerical Methods*, Oficyna Wydawnicza Politechniki Warszawskiej