Numerical methods

Project C No.25

AAYUSH GUPTA

309601

Problem 1

For the following experimental measurements (sample)

x_i	y_i
-5	-13.0376
-4	-6.5256
-3	-2.9949
-2	-1.2815
-1	-0.0683
0	-0.5885
1	-0.5269
2	-2.1830
3	-7.7977
4	-20.1130
5	-41.4544

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.

Solution:

Theory:

Linear least square fits a polynomial function to a given data. Given data of the form (xi,yi), we can define a fitting function of the following form:

$$F(x) = \sum_{i=0}^{n} a_i \emptyset_i(x)$$

In approximation, the model function does not necessarily pass exactly at the data points. Hence there is a measurable error between the data samples and the model function. From the given data samples with the relation yi=f(xi), then the error can be defined to be equal to

$$E = f(xi) - \sum_{i=0}^{n} a_i \emptyset_i(x)$$

Linear Least Square (LLS) seeks to minimize the error E between the actual data samples and the model function. Minimizing the square of the error is similar to minimizing the error itself. The square error is defined as follows.

$$SE = \left[f(xi) - \sum_{i=0}^{n} a_i \emptyset_i(x) \right]^2$$

The minimum of SE is derived by equating its derivative to zero and solve for the parameters ai. This is a sufficient condition to get the minimum since SE is convex. Apply the derivative's chain rule and product rule to get the following

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

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

Rearranging the above equation and dividing by 2 we get

$$-\sum_{j=0}^{N} f(x_j)\cdot \phi_k(x_j) + \sum_{j=0}^{N}\left[\sum_{i=0}^{n} a_i\phi_i(x_j)\right]\cdot \phi_k(x_j) = 0, \qquad k = 0, \dots, n$$

$$\sum_{j=0}^{N}\left[\sum_{i=0}^{n} a_i\phi_i(x_j)\right]\cdot \phi_k(x_j) = \sum_{j=0}^{N} f(x_j)\cdot \phi_k(x_j), \qquad k = 0, \dots, n$$

Divide through by $\phi_k(x_j)$, to get the following

$$\sum_{j=0}^{N}\left[\sum_{i=0}^{n} a_i\phi_i(x_j)\right] = \sum_{j=0}^{N} f(x_j)$$

This represents a set of normal equations with the unknown parameters being a0, a1,…,an. In matrix form this becomes

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_n(x_N) \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

In a simpler notation form the set of normal equations can be written as:

$$\boldsymbol{A}a = y$$

Where

$$\boldsymbol{A} = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_n(x_N) \end{bmatrix}, \qquad a = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}, \qquad y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The solution of the system can be obtained by first forming Gram's matrix of A. the equivalent set of equation would be equal to;

$$\boldsymbol{A^T A}a = A^T y.$$

The Gram's matrix $\boldsymbol{G} = \boldsymbol{A^T A}$ is non-singular since A has a full rank, however, for large sizes of **G**, the matrix can be ill-conditioned. This would cause huge error when solving the system. A measure on the level of il-condition of Gram's matrix is given by the condition number. In equation form it is given by.

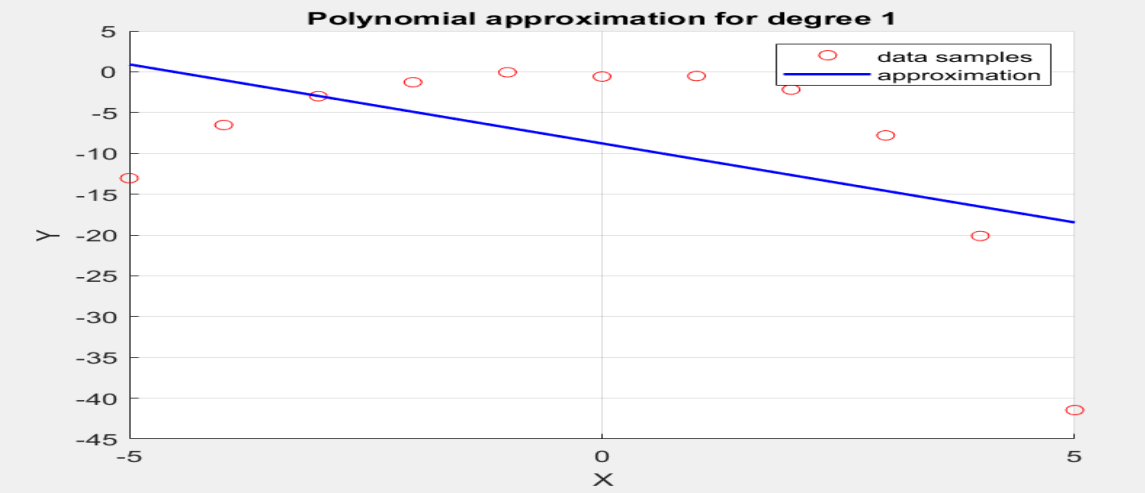
$$Condition\ Number = norm\big(inv(G)\big) * norm(G)$$

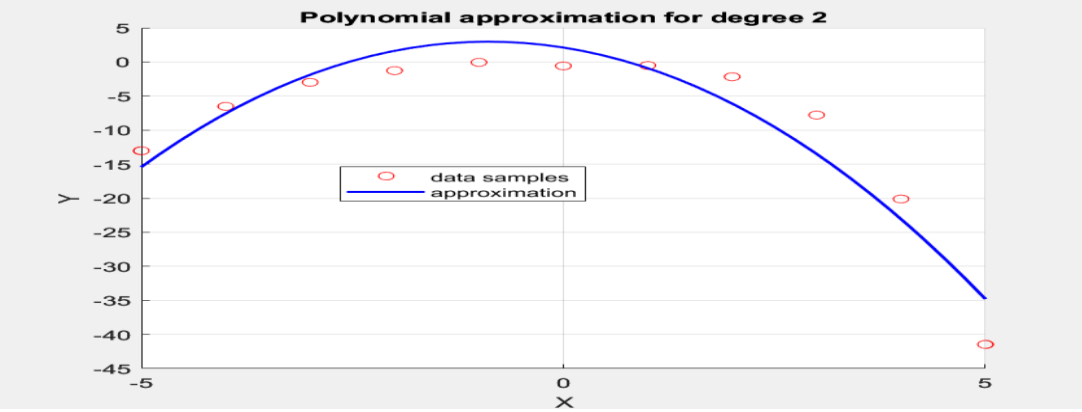
A large condition number implied a large relative error on the results. A good approach to solve this problem is to use QR factorization of matrix A. the set of equations is re-written using the Q and R matrices as follows:

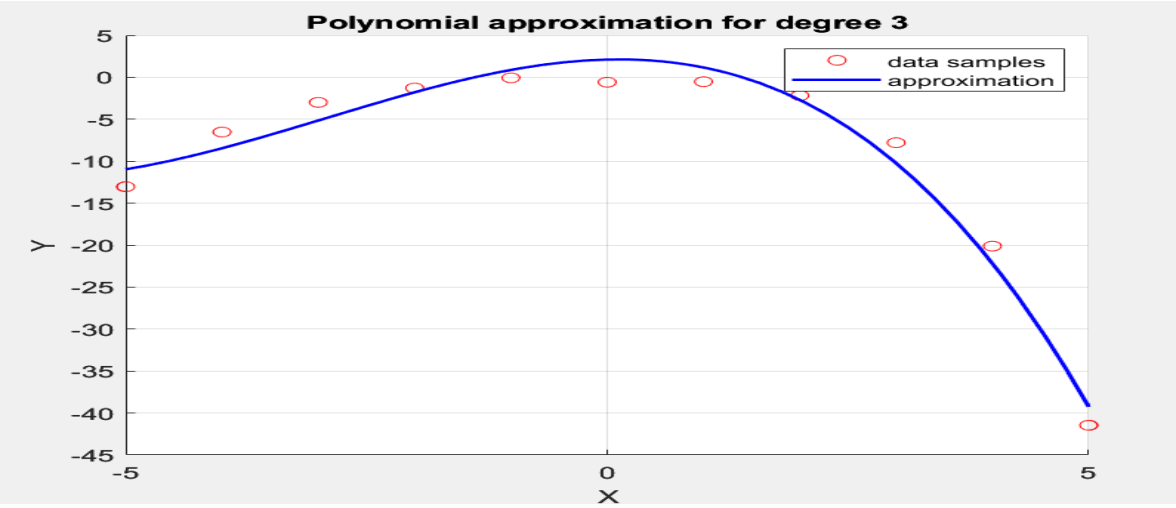
$$\boldsymbol{R}a = \boldsymbol{Q}^T y$$

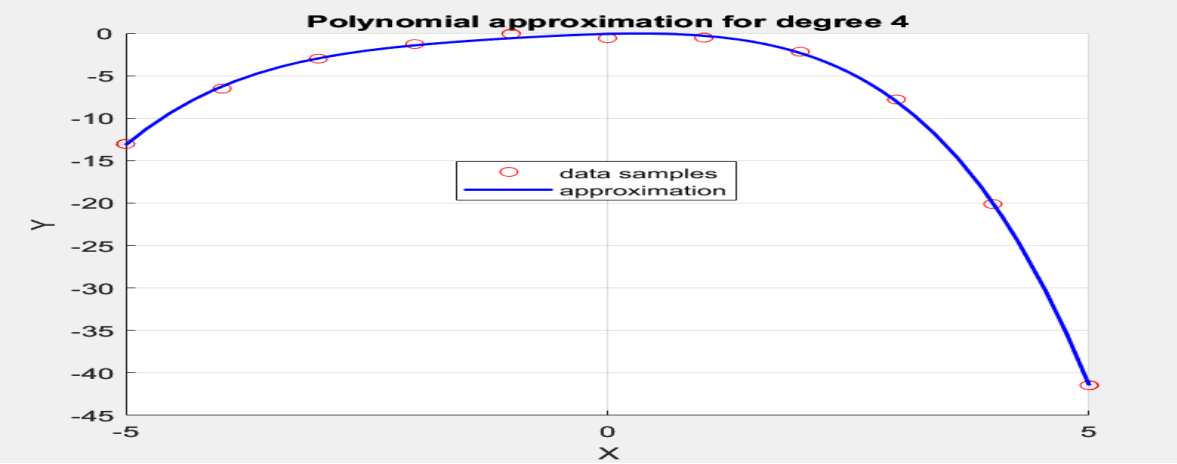
Since R is a triangular matrix, the solution parameters ‘a’ can be obtained through back substitution. This kind of system requires computations. More importantly is that matrix R is always in good condition to guarantee higher accuracy.

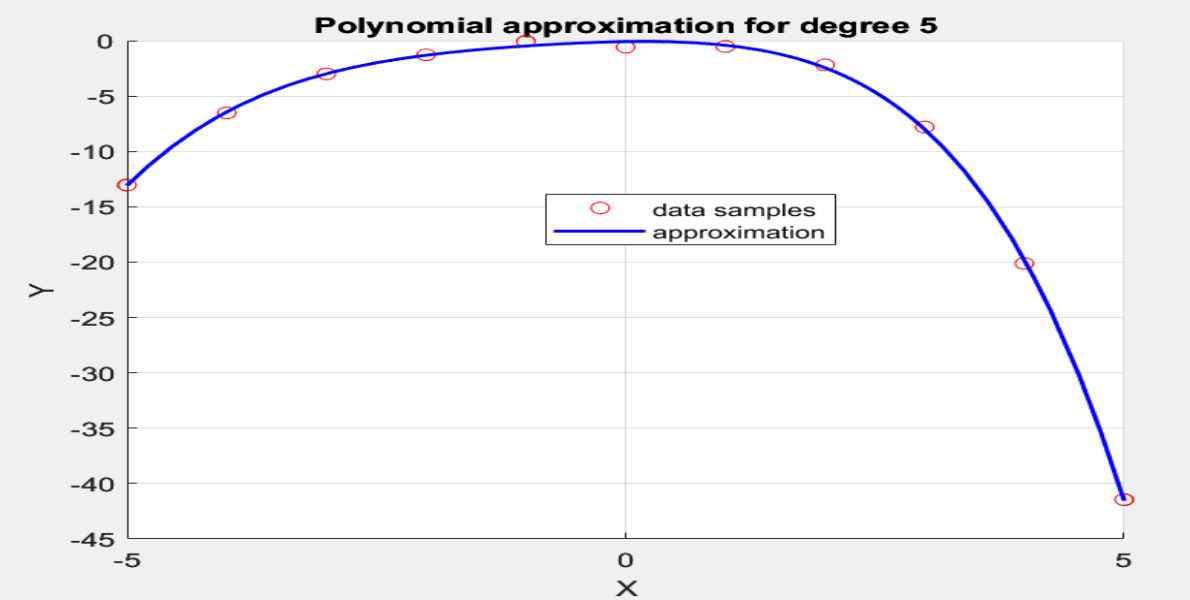
Result and calculations:

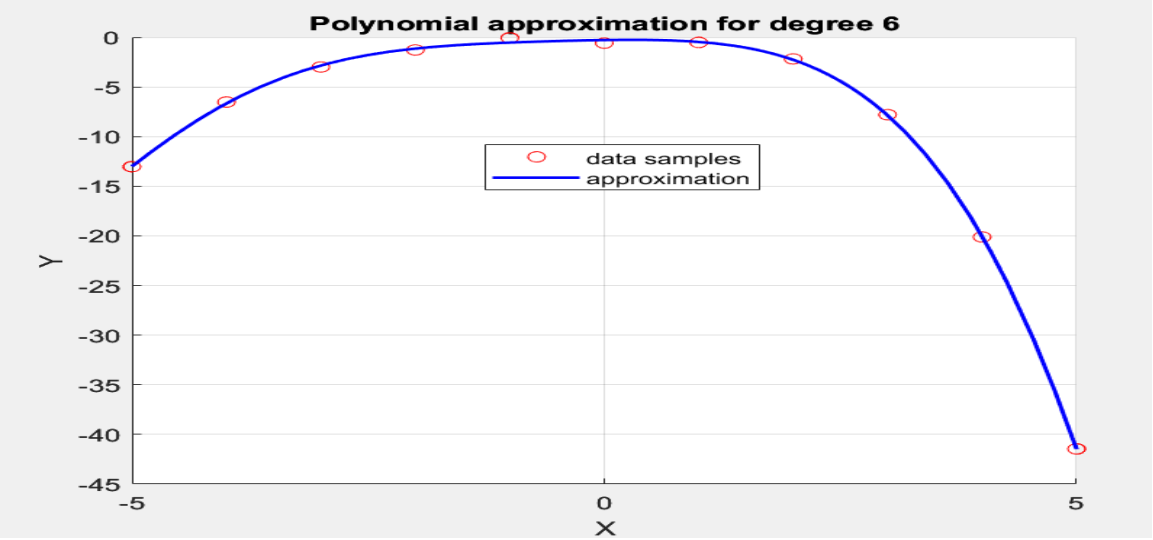


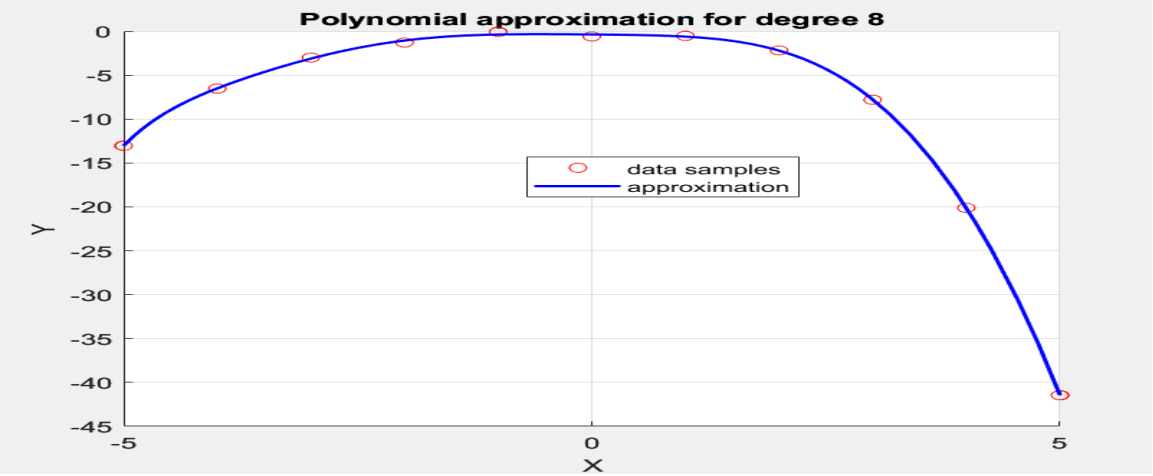


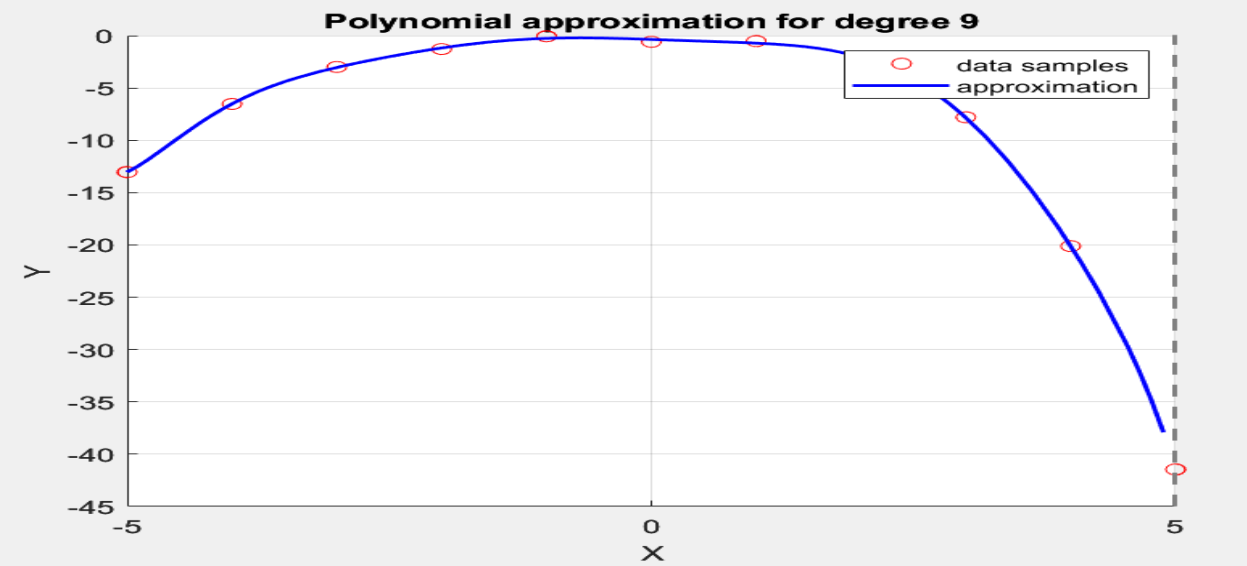


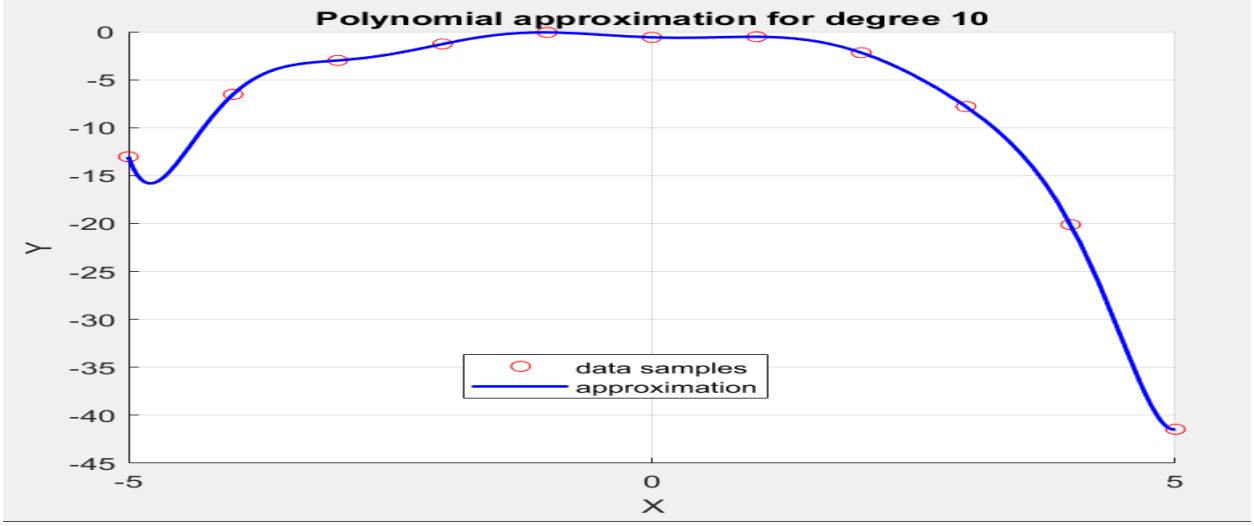

Polynomial approximation for degree 4
data samples
approximation
Y
X
0
-5
-10
-15
-20
-25
-30
-35
-40
-45
-5
0
5


Polynomial approximation for degree 5
data samples
approximation
Y
X
0
-5
-10
-15
-20
-25
-30
-35
-40
-45
-5
0
5


Polynomial approximation for degree 6
data samples
approximation
Y
X
0
-5
-10
-15
-20
-25
-30
-35
-40
-45
-5
0
5


Polynomial approximation for degree 8
data samples
approximation
Y
X
0
-5
-10
-15
-20
-25
-30
-35
-40
-45
-5
0
5


Polynomial approximation for degree 9
data samples
approximation
Y
X
0
-5
-10
-15
-20
-25
-30
-35
-40
-45
-5
0
5

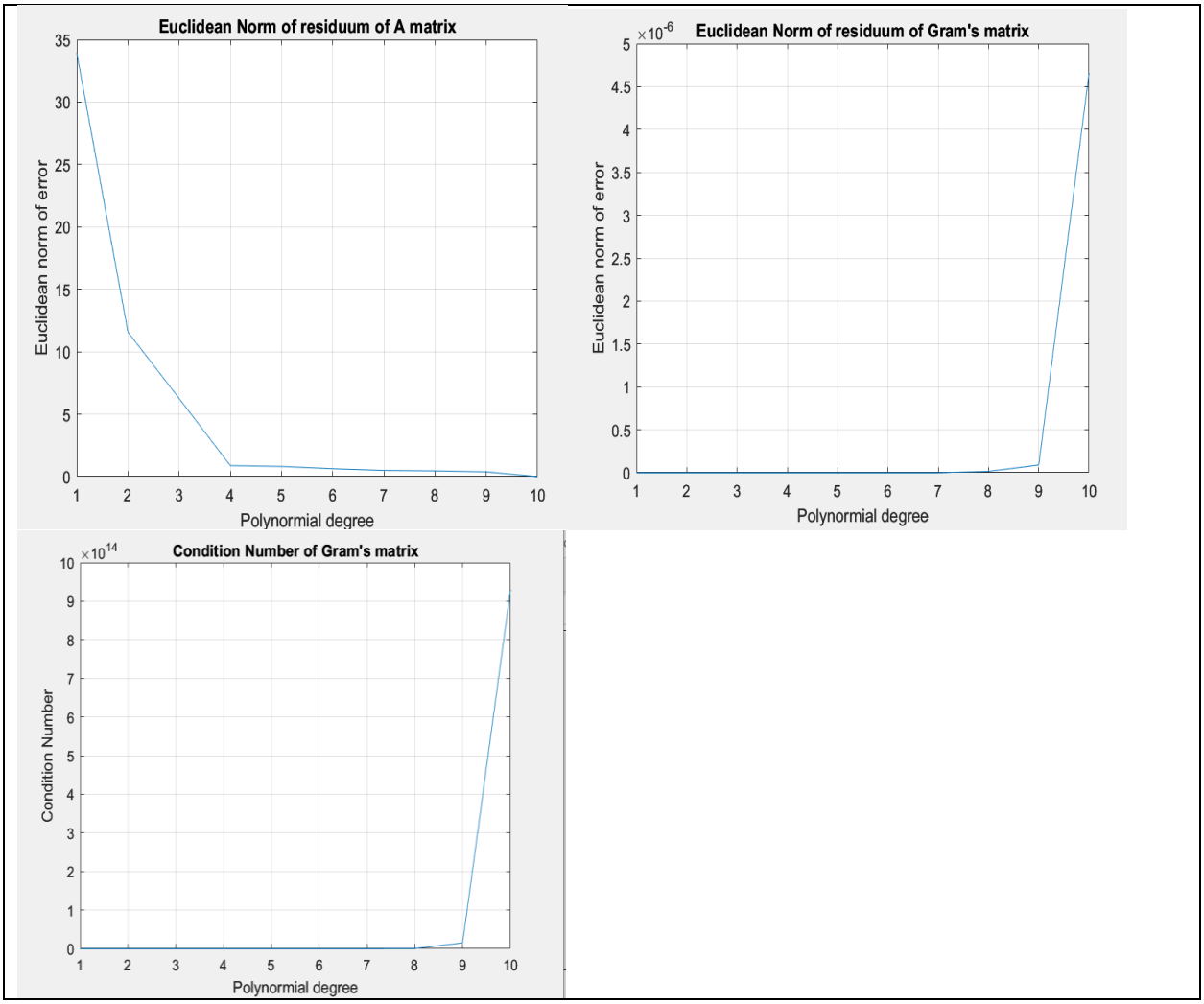


As the polynomial order increases, the accuracy of the approximation increases. The polynomial The Euclidean norm of residuum of A matrix decreases with increasing degree of the polynomial. At 10^{th} degree polynomial we get the smallest Euclidean norm of residuum of A matrix.

Euclidean Norm of residuum of A matrix and Gram's Matrix

n	EnormA
1	33.937
2	11.581
3	6.2604
4	0.88371
5	0.81172
6	0.62977
7	0.50132
8	0.46272
9	0.38451
10	9.504e-10

n	condG	EnormG
1	10	5.8593e-14
2	408.78	4.5586e-13
3	8558.4	9.3921e-13
4	3.1798e+05	9.4178e-13
5	7.4675e+06	1.4582e-11
6	2.8316e+08	2.3295e-10
7	7.6462e+09	4.8021e-10
8	3.3055e+11	1.5018e-08
9	1.5167e+13	8.9719e-08
10	9.293e+14	4.6541e-06



The solution error goes down as the polynomial's degree goes up. This function approximates the polynomial of sixth degree because the solution error is equal to zero. Similar curves exist for polynomials of degree 4 and higher, and these curves are close to the experimental data points. The solution error decreases dramatically from degree (4). The solution for the 8th degree polynomial is tiny, but not sufficiently so. Since there aren't many points in the experimental data, the error can always be reduced. (As in the instance of a polynomial of degree 10 when it equals 0). As predicted, using polynomials with degrees 5 and higher causes the condition number of Gram's matrix to increase dramatically.

Problem 2:

A motion of a point is given by equations:

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

$$\frac{dx2}{dt} = -x_1 + x_2(0.5 - x_1^2 - x_2^2)$$

Determine the trajectory of the motion on the interval [0, 15] for the following initial conditions: x1(0) = 0.5, x2(0) = 0.3. 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.

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

Solution:

Theory:

Differential equation is an equation that determines the relationship between an unknown function and its derivatives.

The solution of the ordinary differential equation (the equation in which we are looking for a function of one variable) is to find a function y(x) that satisfies this equation.

There are no known methods of determining analytical solutions to differential equations - the only way is to determine the solutions using numerical methods.

Frequently used groups of methods for numerical solving of ordinary differential equations are the so-called one-step methods and multi-step methods.

Let the function f be continuous on the set

$$D = \{(x,y) : a \le x \le b\}, \qquad y \in R^m$$

and satisfies a Lipschitz's condition

$$\exists L > 0 \; \forall x \in [a,b] \; \forall y, \bar{y} \in R^m \; \left||f(x,y) - f(x,\bar{y})|\right| \le L \le ||y - \bar{y}||$$

For every starting conditions y a there is exactly one function y(x) which fulfill

$$y'(x) = f(x,y), \qquad y(a) = y_a, \qquad x \in [a,b]$$

Numerical methods of solving systems of differential equations are called differential methods. In those methods, the approximate value of the solution is calculated in following, discrete points xn,

$$a = x_0 \le x_1 \le \cdots \le x_n < \cdots b,$$

where the successive steps of method $h_n = x_{n+1} - x_n$ can be predefined or changeable.

The differential method converges, if for every system of differential equations having y(x) solution, there is

$$\lim_{h_n \to 0} y(x_n; h_n) \to y(x)$$

where y(xn; hn) is an approximate solution which was obtained by this method.

One-step differential methods:

- Runge-Kutta methods (RK),
- Runge-Kutta-Fehlberg methods (RKF)

There is a general formula describing single step hn of the one-step differential method:

$$y_{n+1} = y_n + h * \varphi_f(x_n, y_n; h)$$

where h is a constant,
$y(x_0) = y_0 = y_a,\ x_n = x_0 + n * h,\ n = 0, 1, \ldots$
and $\varphi_f(x_n, y_n; h)$ is a function defining the method: $\varphi_f(x_n, y_n; h) = \frac{y_{n+1} - y_n}{h}$

Taking Equation 12 and that the method converges when h → 0, the approximation condition can be described as:

$$\varphi_f(x_n, y_n; 0) = f(x, y)$$

This equation is necessary and sufficient condition for the convergence of the one-step method, if the assumptions Equation 8 and Equation 9 are satisfied and ϕf(xn, ·; h) satisfies a Lipschitz's condition. Approximation error (local error of method) is an error created in one step:

$$r_n(h) \stackrel{def}{=} y(x_n + h) - [y(x_n) + h * \varphi_f(x_n, y_n; h)]$$

where $y(x_n + h)$ is a solution which cross through the points $y(x_n) = y_n$.

The method is of pth order if:

$$r_n(0) = 0, r_n'^{(0)} = 0, \ldots, r_n^{(p)}(0) = 0, r_n^{(p+1)}(0) \neq 0$$

Then the approximation error is equal to:

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

To perform one step of the RK method, it is needed to calculate the values of the right sides of differential equations exactly m times.

These methods can be defined by the following formulas:

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

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

$$k_i = f\left(x_n + c_i * h, y_n + h * \sum_{j=1}^{i-1} a_{ij} * k_j\right), \qquad i = 2, 3, \dots, m$$

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

Parameters wi, aij, ci are set so that the order of the method is as high as possible.

The methods with m = 4 and order p = 4 are of greatest practical importance - a compromise between accuracy (order of the method) and the computational effort.

Formulas for RK4 method

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

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

$$k_2 = f\left(x_n + \frac{1}{2} * h, y_n + \frac{1}{2} * h * k_1\right),$$

$$k_3 = f\left(x_n + \frac{1}{2} * h, y_n + \frac{1}{2} * h * k_2\right),$$

$$k_4 = f(x_n + h, y_n + h * k_3)$$

Determining the length of step hn:

- If the step hn gets smaller, the approximation error is also getting smaller. In convergent method, the error decreases to zero with h tending to zero.
- If the step hn decreases, the number of iterations increases.

When choosing the optimal step, both of the above facts have to be taken into account.

The length of step hn can be estimated automatically. The most important information, needed for this, is the estimated approximation error in a given step.

In order to estimate the error, beside taking step hn, two additional steps with length 0.5*hn are also performed. Formula for the errors for the steps of the length hn and 0.5*hn:

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

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

where $y_n^{(1)}$ is a new point gained in the h_n step,
$y_n^{(2)}$ is a new point gained in two steps $0.5*h_n$

The step hn don't have to be always constant. To modify the length of step the coefficient α of modification is needed:

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

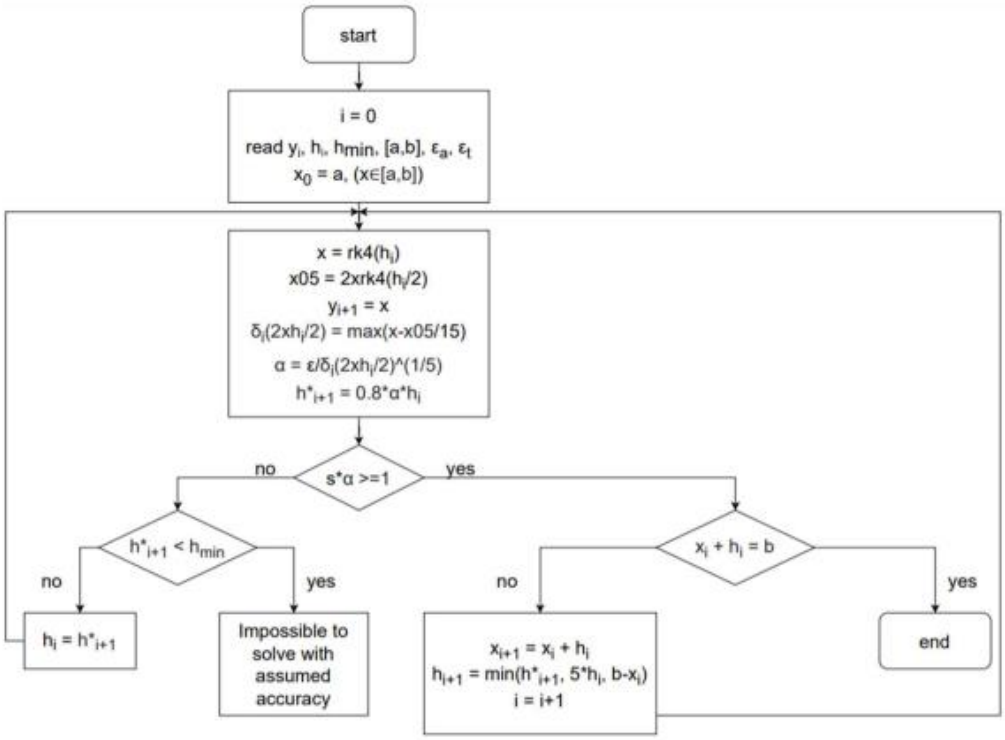
where ε is an assumed accuracy of calculations $\varepsilon = |y_n|^{(2)} * \varepsilon_{relative} + \varepsilon_{absolute}$

Changed step h_n can be calculated by:

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

where $s < 1$ in a safety factor (for RK method $s \approx 0.9$)

Algorithm of RK method with a variable step-size automatically adjusted by the algorithm, making error estimation according to the step-doubling rule:



General formula for an iteration in a k-step method with a constant step h:

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

where $y_0 = y(x_0) = y_a$,
$x_n = x_0 + n*h$,
$x_0 = a$, $x \in [a, b]$

If $\beta_0 = 0$, then the method is explicit. Otherwise, it's an implicit method.

β is a parameter depending on the number of iterations and the type of method. It can be read from some tables.

Approximation error of multi-step differential method in one step can be described as:

$$r_n(h) \stackrel{\text{def}}{=} \left[\sum_{j=1}^{k} \alpha_j * y(x_{n-j}) + h * \sum_{j=0}^{k} \beta_j * f(x_{n-j}, y(x_{n-j}))\right] - y(x_n)$$

The method is of p^{th} order if:

$$c_0 = 0, c_1 = 0, \ldots, c_p = 0, c_{p+1} \neq 0$$

Then the approximation error is equal to:

$$r_n(h) = c_{p+1} * h^{(p+1)} * (y(x_n))^{(p+1)} + O(h^{(p+2)})$$

where c_{p+1} is an error constant

The multi-step method is stable if

- for h = 0 there is a relationship described as formula:

$$-y_n + \sum_{j=1}^{k} \alpha_j * y_{n-j} = -y_n + \alpha_1 * y_{n-1} + \cdots + \alpha_{k-1} * y_{n-k-1} + \alpha_k * y_{n-k} = 0$$

which is stable.

- for h > 0 there is a stable relationship

$$\sum_{j=0}^{k} (\alpha_j + h * \lambda * \beta_j) * y_{n-j} = 0$$

where $a_0 = -1$

For h > 0 method can be absolutely stable from q to 0. Values of q can be found in some tables.

Approximation condition: K-step method is convergent if, and only if, it is stable and at least of first order. Conditions for the best multi-step method:

- high order and small error constant,
- great field of absolute stability,
- as little calculations for one iteration as possible

Explicit methods fulfill all above conditions but the first two are not satisfied as much as in case of implicit methods which not fulfill last condition at all.

That is why the predictor-corrector (PC) methods exist as a connection of explicit and implicit methods.

Algorithm of P_kEC_kE method:

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

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

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

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

Algorithm of P_kEC_kE for Adam's methods:

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

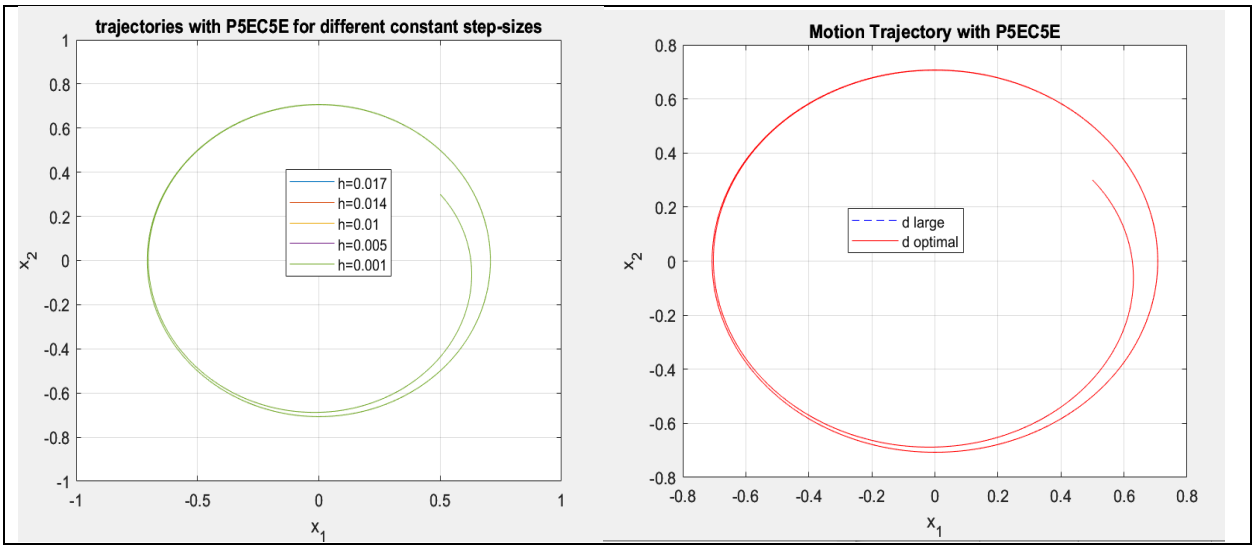
E: $f_n^{[0]} = f(x_n, y_n^{[0]})$

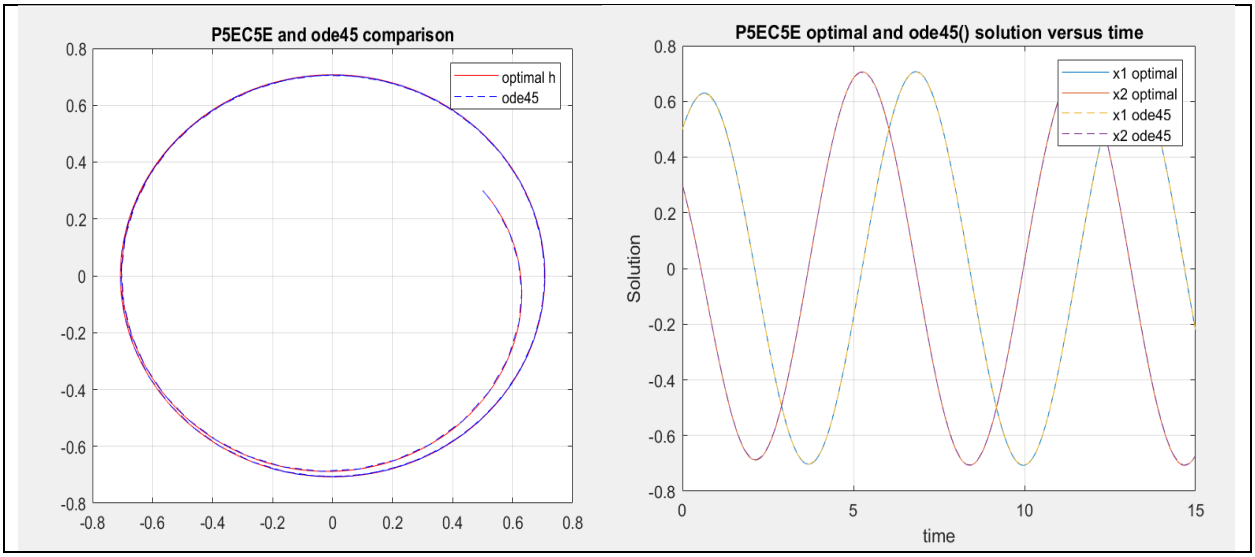
C: $y_n = y_{n-1} + h * \sum_{j=1}^{k} \beta_j^* * f_{n-j} + h * \beta_0^* * f_n^{[0]}$

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

Result and calculations:

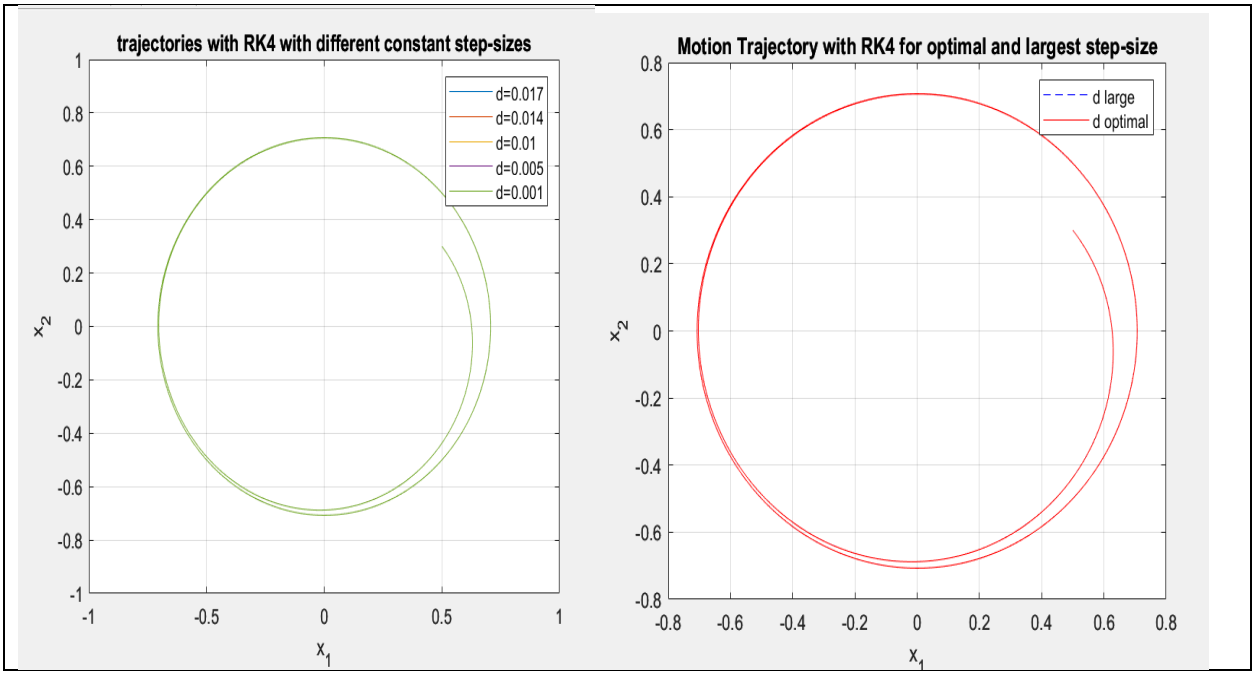
TASK II_a- Adam PC (P_5EC_5E)

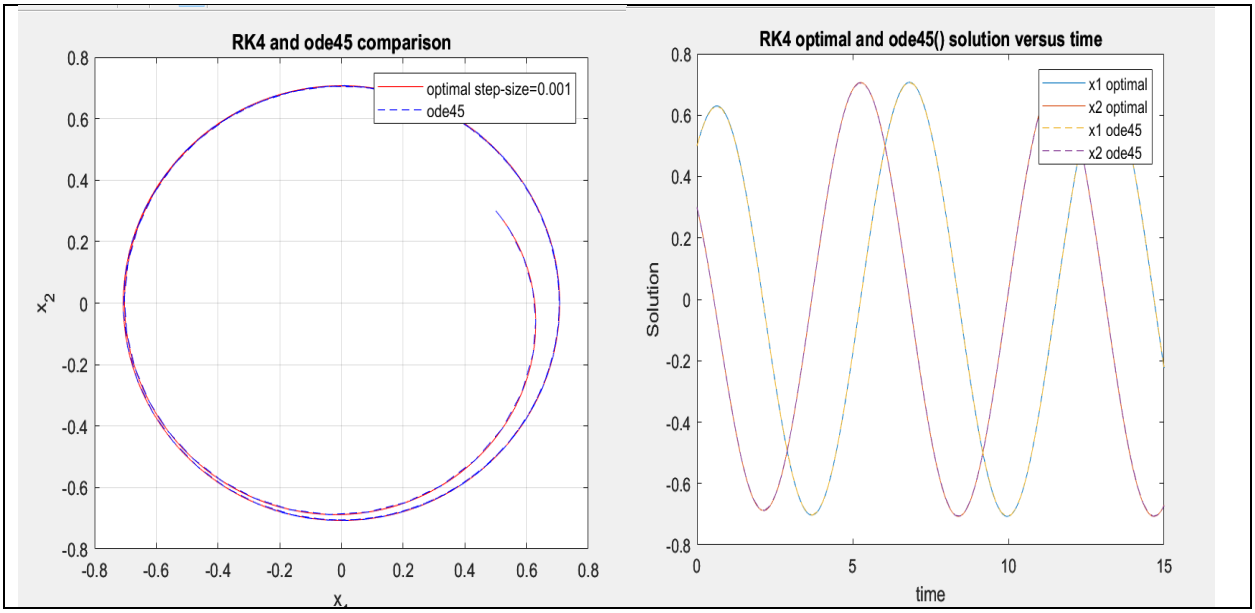




Analyzing the solution of the system of ODE using Adam PC method with different step sizes.

TASK II_a- Runge-Kutta method of 4th order (RK4





According to the aforementioned figure, h = 0.001 is the ideal step size. Lowering the step size did not result in any appreciable improvement. A larger step size, however, reveals some obvious alterations. On a single plot, two solution curves are drawn: one for the ideal constant step size and the other for a greater step size (for which the solution visibly differs from the first one). The following figure depicts this.

The number of steps and accuracy should be considered while determining the step size. It appears that both requirements are met when h = 0.01. Lowering h could improve the accuracy of the solutions, but the execution time would increase. In comparison to the MATLAB ode45 function, my Runge-Kutta method of fourth order and Adams PC method implementations are successful. The execution times of those approaches are comparable for my data and implementations, but it can be argued that the RK4 method is a little bit quicker than the Adams PC method.

TASK II,a- Runge-Kutta method of 4th order (RK4) with a variable step size automatically adjusted.

Appendix:

.1.1 P3_1st.m

```
clear all
%%task 1

%% Measurement data
xi = [-5.000 -4.000 -3.000 -2.000 -1.000 0.000 1.000 2.000 3.000 4.000 5.000];
yi = [-13.0376 -6.5256 -2.9949 -1.2815 -0.0683 -0.5885 -0.5269 -2.1830 -7.7977 -
20.1130 -41.4544]

%% Linear-Least-Squares
Nmax = 10; % maximum polynomial degree
EnormA = zeros(Nmax,1); % Initialize Euclidean Norm of residuum of A matrix
EnormG = zeros(Nmax,1); % Initialize Euclidean Norm of residuum of G matrix
condG = zeros(Nmax,1); % Initialize Condition number of Gram's matrix'

for N=1:Nmax % Iterate for all possible polynomials
    % Linear-Least-Squares
    [A,coeff] = L_L_S(xi,yi,N);
    G = A'*A ;% Gram's matrix

    % Figure of polynomial approximation for different order
    figure()
    hold on
    plot(xi,yi,'or');
    grid on
    fplot(poly2sym(coeff), [xi(1),xi(end)],'-b','linewidth',1.5);
    st = sprintf('Polynomial approximation for degree %d',N);
    xlabel('X');
    ylabel('Y');
    title(st);
    legend('data samples', 'approximation');

    % Euclidean Norm of residuum of A matrix
    EnormA(N) = norm((A*coeff) - yi(:));

    % Euclidean Norm of residuum of Gram's matrix
    EnormG(N) = norm(A'*yi(:) - G*coeff);

    % Condition number of Gram's matrix'
    condG(N) = norm(G)*norm(inv(G));
end

% Plot Euclidean Norm of residuum of A matrix
figure()
n = (1:Nmax)';
plot(n,EnormA), grid on
xlabel('Polynormial degree')
```

```
ylabel('Euclidean norm of error')
title('Euclidean Norm of residuum of A matrix')

% Plot Euclidean Norm of residuum of Gram's matrix
figure()
plot(n,EnormG), grid on
xlabel('Polynormial degree')
ylabel('Euclidean norm of error')
title('Euclidean Norm of residuum of Gram''s matrix')

% Plot Condition Number of Gram's matrix
figure()
plot(n,condG), grid on
xlabel('Polynormial degree')
ylabel('Condition Number')
title("Condition Number of Gram's matrix")

% Show Euclidean Norms and Condition Number in a table
table(n,EnormA)
table(n,condG, EnormG)
```

.1.2 QR.m

```
function [Q, R] = QR(A)
%=========================================================================
[m,n] = size(A);
%=========================================================================
% Initialization
z = zeros(n,1);
Q = zeros(m,n);
R = zeros(n,n);

%=========================================================================
% Main Program for QR Algorithm
%=========================================================================
% Initializing the Q vector
for i = 1:m
    Q(i,1) = A(i,1) / norm(A(:,1));
end

% Iterative Loop
for i = 2:n

    sum = zeros(m,1);

    for k = 1:i-1
          sum = sum + (A(:,i)'*Q(:,k))*Q(:,k);
    end
```

```
    z = A(:,i) - sum;

    for j = 1:m
        Q(j,i) = z(j)/norm(z);
    end
end

% Calculation for R
for i = 1:m
    for j = i:n
        R(i,j) = Q(:,i)'*A(:,j);
    end
end
```

1.3 L_L_S.m

```
function [A,x]=L_L_S(X,Y,N)

% INPUTS: [X,Y] is the input data samples
%             N is the degree of the approximating polynomial
%
% OUTPUT: A -- the matrix of the coefficients of the normal equations
%         x -- solution of the system: AX=Y
%
% Calculating matrix A.
for i=1:length(X)
    for j=1:(N+1)
        A(i, j) = X(i)^(j-1);
    end
end
A = flip(A,2);
% Calucalting QR decomposition of matrix A.
[Q, R] = QR(A);

% Calculating the coefficients of the polynomial approximation
B = Q'*Y(:);
n = length(B);
x = zeros(n, 1);
x(n, :) = B(n) / R(n,n);
i = n-1;

% Back substitution
while i > 0
    x(i) = (B(i) - R(i, i+1:n) * x(i+1:n)) / R(i,i);
    i = i - 1;
end
end
```

TASK II-a

2.1 .1 AdamPC_init.m

```
%% Numerical Methods, PROJECT No. 3
% TASK 2 (a) - Aayush Gupta
% Adam P5EC5E
clc;
clear all;
close all
%-------------------------------------------------------------------------
%Set the constant parameters
Initcond = [0.5 0.3];  % Initial conditions: x1(0) = 0.5, x2(0) = 0.3
time = [0,15];    % time interval

%-------------------------------------------------------------------------
% Test different step sizes
st_sz = [0.017,0.014,0.01,0.005,0.001];

figure()
for i=1:length(st_sz)
     [ T, X ] = P5EC5E(@fun,time, Initcond,st_sz(i));
    plot(X(:,1),X(:,2))
    hold on
end
grid on
axis([-1 1 -1 1])
legend('h=0.017','h=0.014','h=0.01','h=0.005','h=0.001')
title('trajectories with P5EC5E for different constant step-sizes')
xlabel('x_1')
ylabel('x_2')

%-------------------------------------------------------------------------
% Plot Trajectories: for the optimal and largest step-size
hLarge = 0.017;
[Tlarge,Xlarge] = P5EC5E(@fun,time, Initcond,hLarge);
hOptimal = 0.001;
[Topt,Xopt] = P5EC5E(@fun,time, Initcond,hOptimal);
figure()
plot(Xlarge(:,1),Xlarge(:,2),'--b')
hold on
plot(Xopt(:,1),Xopt(:,2),'-r')
grid on
legend('d large','d optimal')
title('Motion Trajectory with P5EC5E')
xlabel('x_1')
ylabel('x_2')

%-------------------------------------------------------------------------
% Compare optimal solution with ode45(solver) solution
[t_45,x_45] = ode45(@fun, time, Initcond);
figure()
plot(Xopt(:,1),Xopt(:,2),'-r')
hold on
plot(x_45(:,1),x_45(:,2),'--b')
```

```
grid on
legend('optimal h', 'ode45');
title('P5EC5E and ode45 comparison');

%-----------------------------------------------------------------------
% Compare optimal solution vs time with ode45() solver solution
figure();
plot(Topt, Xopt);
hold on
plot(t_45, x_45, '--');
grid on
xlabel('time');
ylabel('Solution');
legend('x1 optimal', 'x2 optimal', 'x1 ode45', 'x2 ode45');
title('P5EC5E optimal and ode45() solution versus time');

%% ODE equations of the motion
function dxdt = fun(t,x)
    dxdt = zeros(2,1);
    dxdt(1) = x(2) + x(1) * (0.5 - x(1)^2 - x(2)^2);
    dxdt(2) = -x(1) + x(2) * (0.5 - x(1)^2 - x(2)^2);
end
```

2.1.2 P5EC5E.m

```
function [T,YY] = P5EC5E(fun,time, init, h)
k = 5 ; % Number of steps
t(1) = time(1); % initial time
YY(1, :) = init;
for j=1:k-1
    t(j+1) = t(j) + h;
end
% Get the first 5 values from Runge-Kutta method of order 4
[ ~, Y5 ] = RK4(fun, t(1), t(end), init, h);

YY(1:5, :)= Y5;
yn0 = @(y, f, h) (y + h*1/720*(1901*f(end,:) - 2774*f(end-1,:) + 2616*f(end-2,:)
- 1274*f(end-3,:) + 251*f(end-4,:)));

% fn0 = f(xn, yn0);
yn = @(y, f, h, fn0) (y + h*1/1440*(1427*f(end,:) - 798*f(end-1,:) + 482*f(end -
2,:) - 173*f(end-3,:) + 27*f(end-4,:)) + h*475/1440*fn0);

% fn = f(xn, yn);

f = zeros(length(YY), 2);
N = (time(2) - time(1)) / h;
```

```
for i = k+1:N
    t(i) = t(i-1) + h;
    % P
    P = yn0(YY(i-1, :), f(1:i-1, :), h);

    % E
    f(i,:) = fun(t(i),P)';

    % C
    C = yn(YY(i-1, :), f(1:i-1, :), h, f(i, :));

    % E
    f(i,:) = fun(t(i),C)';

    YY(i, :) = C;
end
T = t(:);
end
```

2.1.3 RK4A.m

```
%% Numerical Methods, PROJECT No. 3
% TASK 2 (a) - Aayush Gupta
% Runge-Kutta method of 4th order (RK4)
clc;
clear all;
%-------------------------------------------------------------------
%% Constant parameters

InitCond = [0.5 0.3]; % Initial conditions: x1(0) = 0.5, x2(0) = 0.3
time = [0,15];    % Time

%-------------------------------------------------------------------
%% Test different step sizes

st_sz = [0.017,0.014,0.01,0.005,0.001];

figure()
for i=1:length(st_sz)
    [T,X] = RK4(@func, time(1), time(2), InitCond, st_sz(i));
    plot(X(:,1),X(:,2))
    hold on
end
grid on
hold off
axis([-1 1 -1 1])
legend('d=0.017','d=0.014','d=0.01','d=0.005','d=0.001')
title('trajectories with RK4 with different constant step-sizes')
xlabel('x_1')
```

```
ylabel('x_2')

%-------------------------------------------------------------------------
%% Plot Trajectories: for the optimal constant and largest step-size

d_Large = 0.017;
[ Tlarge, Xlarge ] = RK4(@func, time(1), time(2), InitCond, d_Large);
d_Optimal = 0.001;
[ Topt, Xopt ] = RK4(@func, time(1), time(2), InitCond, d_Optimal);

figure()
plot(Xlarge(:,1),Xlarge(:,2),'--b')
hold on
plot(Xopt(:,1),Xopt(:,2),'-r')
grid on
legend('d large','d optimal')
title('Motion Trajectory with RK4 for optimal and largest step-size')
xlabel('x_1')
ylabel('x_2')

%-------------------------------------------------------------------------
%% Compare optimal solution with ode45(solver)

[t_ode45,x_ode45] = ode45(@func, time, InitCond);
figure()
plot(Xopt(:,1),Xopt(:,2),'-r')
hold on
plot(x_ode45(:,1),x_ode45(:,2),'--b')
grid on
legend('optimal step-size=0.001', 'ode45');
title('RK4 and ode45 comparison');
xlabel('x_1')
ylabel('x_2')

%-------------------------------------------------------------------------
%% Compare optimal solution with ode45()solver vs time
figure();
plot(Topt, Xopt);
hold on
plot(t_ode45, x_ode45, '--');
xlabel('time');
ylabel('Solution');
legend('x1 optimal', 'x2 optimal', 'x1 ode45', 'x2 ode45');
title('RK4 optimal and ode45() solution versus time');

%% ODE equations of the motion
function dxdt = func(t,x)
    dxdt = zeros(2,1);
    dxdt(1) = x(2) + x(1) * (0.5 - x(1)^2 - x(2)^2);
    dxdt(2) = -x(1) + x(2) * (0.5 - x(1)^2 - x(2)^2);
end
```

2.1.4 RK4.m

```
function [ T, Y ] = RK4( f, t0, tf, y0, d)

T = (t0 : d : tf)';             % time vector
N = length(T);                  % Number of sample points
dim = length(y0);               % Dimension of the solution
Y = zeros(N,dim);               % Initialize the solution
Y(1,1) = y0(1);Y(1,2) = y0(2);  % Append the initial conditions

% RK method of order 4:
for i = 2 : length(T)
    % k1 = f (xn, yn) ,
    k1 = f(T(i-1),Y(i-1,:))';

    % k2 = f(xn + 1/2h, yn+1/2 hk1)
    k2 = f(T(i-1)+(1/2)*d,Y(i-1,:)+(1/2)*d*k1)';

    % k3 = f(xn + 1/2h, yn+1/2hk2)
    k3 = f(T(i-1)+(1/2)*d,Y(i-1,:)+(1/2)*d*k2)';

    % k4 = f(xn + d, yn+hk3)
    k4 = f(T(i-1)+d,Y(i-1,:)+d*k3)';

    %yn+1 = yn +(1/6)d(k1 + 2k2 + 2k3 + k4)
    Y(i,:) = Y(i-1,:) + (1/6)*d*(k1 + 2*k2 + 2*k3 + k4);
end

end
```