ENUME ASSIGNMENT C no.21

TAIMUR RAHMAN

Contents

Least-Squares Approximation

Theory

Linear least squares is an approach to fit mathematical or statistical model to data in cases where the idealised value provided by the model for any data point is expressed linearly in terms of the unknown parameters of the model. The resulting fitted model can be used to summarise the data, to predict unobserved values from the same system, and to understand the mechanisms that may underlie the system. Linear least squares is the problem of approximately solving an overdetermined system of linear equations, where the best approximation is defined as that which minimises the sum of squared differences between the data values and their corresponding modelled values. The approach is called linear least squares since the assumed function is linear in the parameters to be estimated. The problems are convex and have a closed-form solution that is unique, provided that the number of data points used for fitting equals or exceeds the number of unknown parameters, except in special situations. In contrast, non-linear least squares problems generally must be solved by an iterative procedure, and the problems can be non-convex with multiple optima for the objective function. If prior distributions are available, then even an underdetermined system can be solved using the Bayesian MMSE estimator.

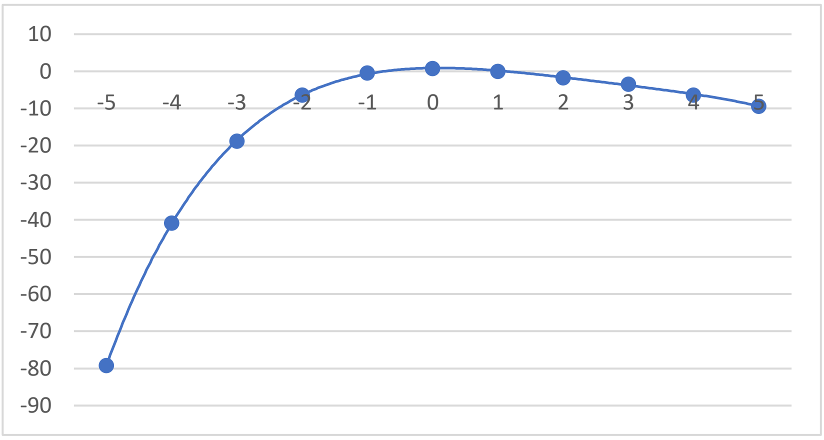
Polynomials are often used as approximating functions (let the order of the polynomial be n), The justification of this fact is the classic Weierstrass theorem about a uniform approximation of a continuous function f(x) on a closed interval [a, b] by an polynomial.

The order n of the approximating polynomial is usually much lower than the number of points, at which the values of the original function are given.

The QR Method should be better because the Gram matrix solution is ill-conditioned with bigger matrices.

Task

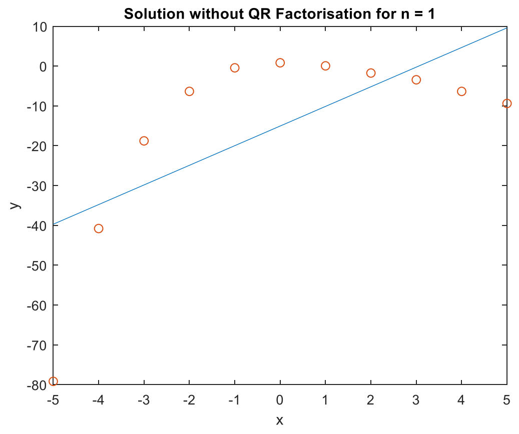
x	y
-5	-79.1639
-4	-40.7900
-3	-18.7814
-2	-6.3530
-1	-0.4392
0	0.8270
1	0.0585
2	-1.7477
3	-3.4384
4	-6.3580
5	-9.3875

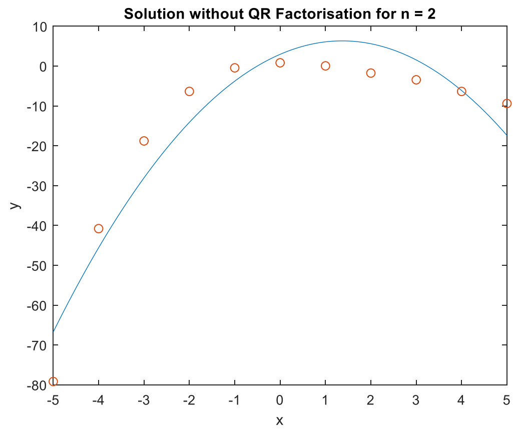


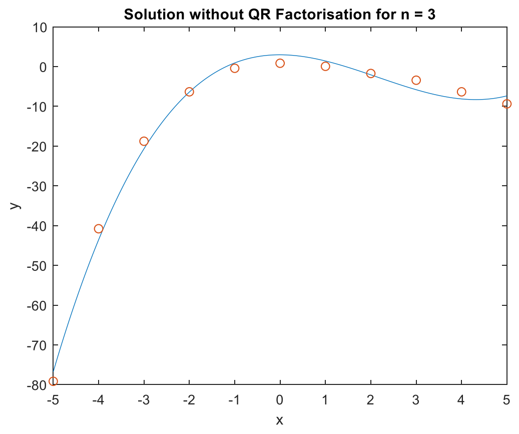
I expect the polynomial to fit the values at degree 4 and above.

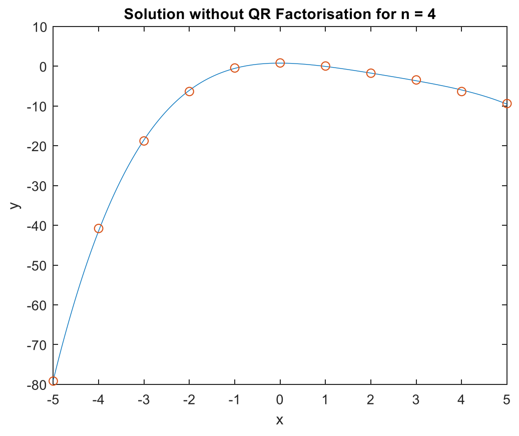
Result

Solution without QR factorisation

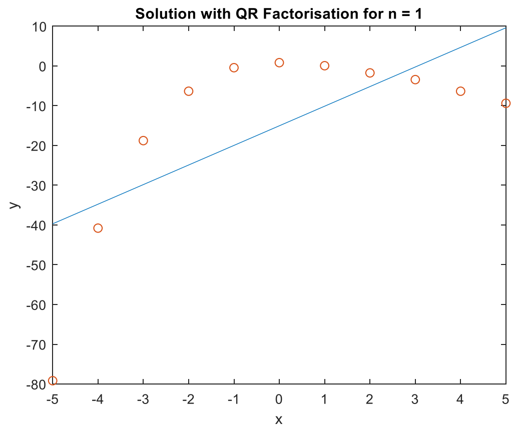


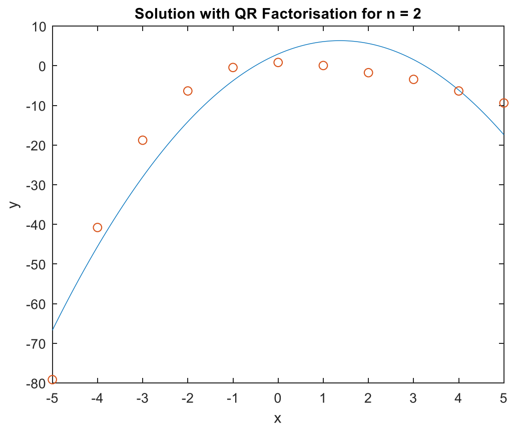


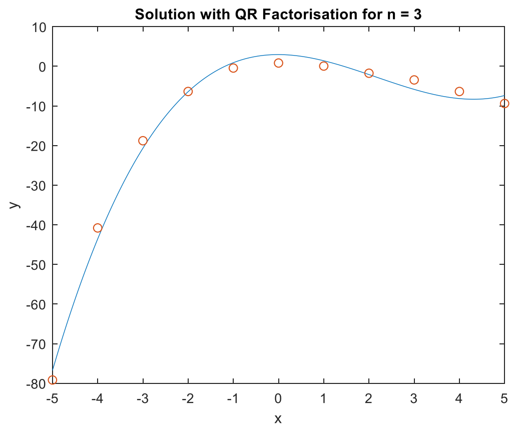


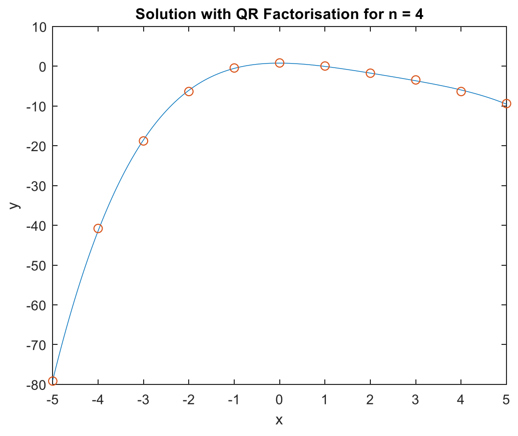


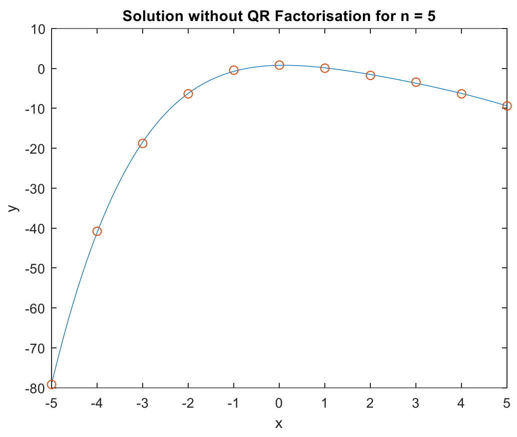
Solution with QR factorisation

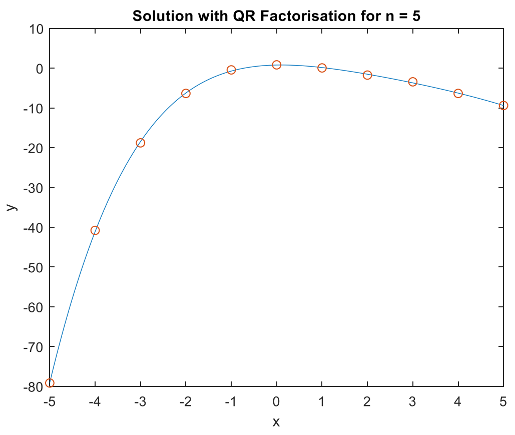


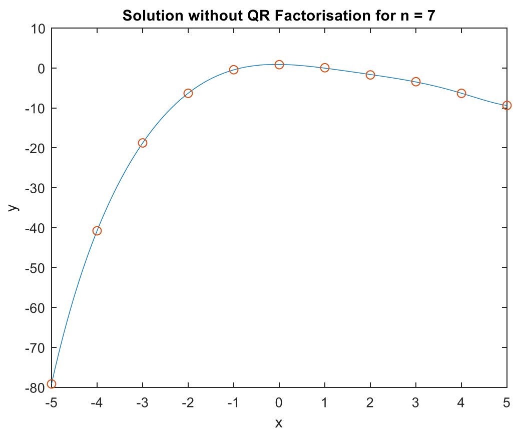


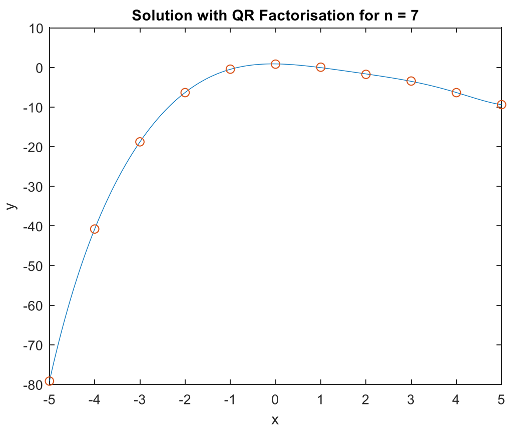


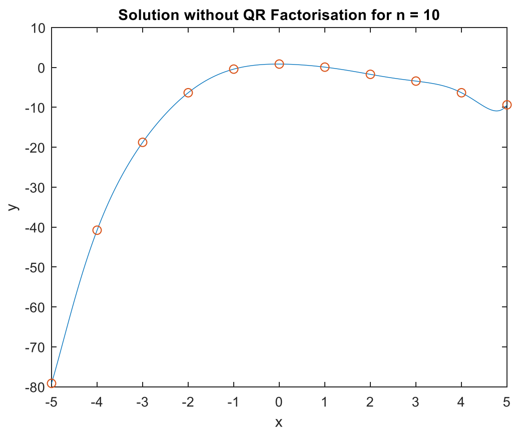


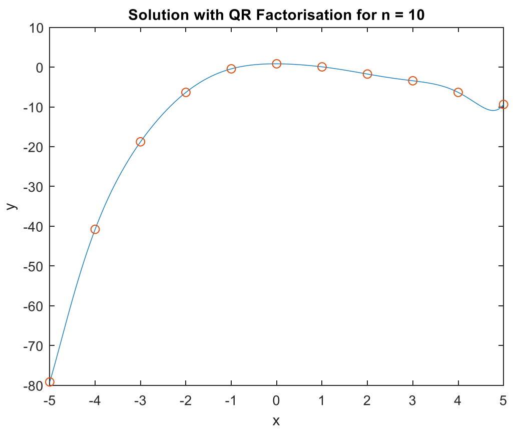

Solution without QR Factorisation for n = 5
y
x

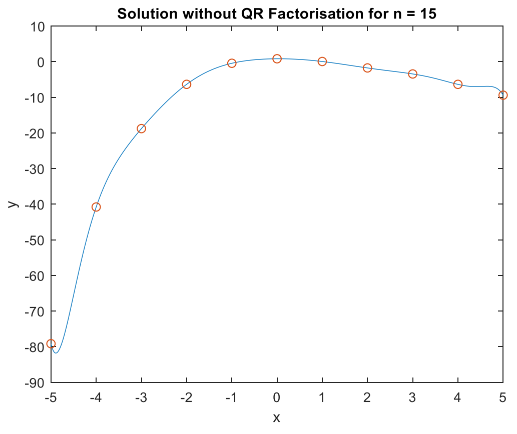

Solution with QR Factorisation for n = 5
y
x

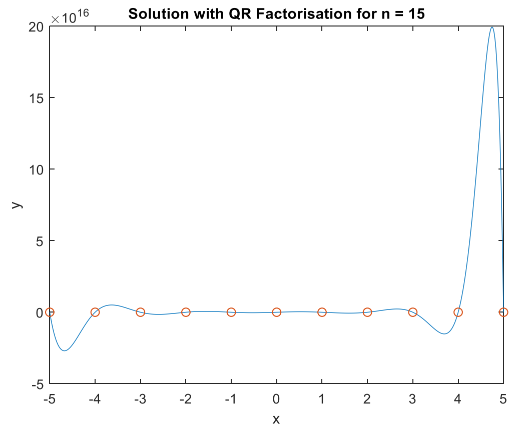

Solution without QR Factorisation for n = 7
y
x

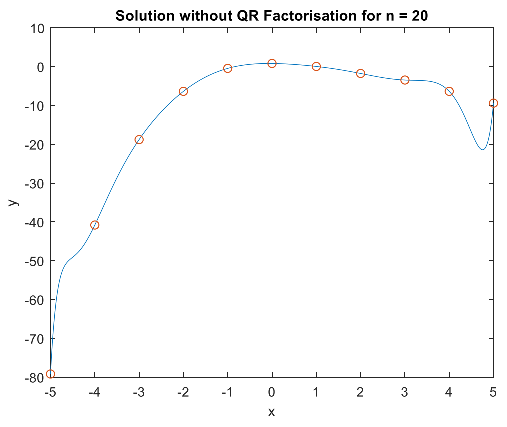

Solution with QR Factorisation for n = 7
y
x

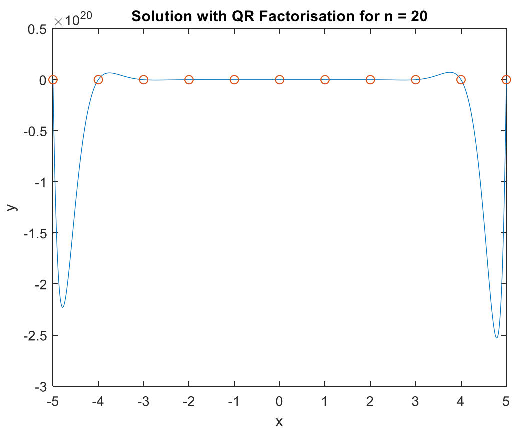

Solution without QR Factorisation for n = 10
y
x


Solution with QR Factorisation for n = 10
y
x


Solution without QR Factorisation for n = 15
y
x


Solution with QR Factorisation for n = 15
×10^16
y
x





	Without QR		**With QR**	
n	**P**	**error**	**P**	**error**
1	4.9304 -15.0521	0	4.9304 -15.0521	1.1369e-13
2	-1.8009 4.9304 2.9569	2.8422e-14	-1.8009 4.9304 2.9569	1.1719e-13
3	0.2790 -1.8009 -0.0357 2.9569	2.8422e-14	0.2790 -1.8009 -0.0357 2.9569	1.8228e-12
4	-0.0302 0.2790 -1.0465 -0.0357 0.7843	2.8422e-14	-0.0302 0.2790 -1.0465 -0.0357 0.7843	1.4729e-11
5	0.0013 -0.0302 0.2394 -1.0465 0.2028 0.7843	1.8192e-12	0.0013 -0.0302 0.2394 -1.0465 0.2028 0.7843	3.2618e-11
7	0.0002 -0.0002 -0.0088 -0.0210 0.3592 -1.1295 -0.1381 0.8901	2.9104e-11	0.0002 -0.0002 -0.0088 -0.0210 0.3592 -1.1295 -0.1381 0.8901	1.8626e-09
10	0.0000 0.0000 -0.0010 -0.0004 0.0167 0.0009 -0.1318 0.3030 -0.9013 -0.0547 0.8270	3.5771e-07	0.0000 0.0000 -0.0010 -0.0004 0.0167 0.0009 -0.1318 0.3030 -0.9013 -0.0547 0.8270	6.3665e-08
15	-0.0000 0.0000 -0.0000 0.0000 0.0000 -0.0000 0.0002 0.0003 -0.0067 0.0068 0.0578 -0.1043 0.1343 -0.9201 0.0633 0.8270	0.0020	-0.0000 -0.0000 -0.0000 0.0001 0.0002 -0.0026 -0.0018 0.0224 -0.0268 -0.0882 0.4150 0.1469 -1.4196 -0.0787 1.0330 0.0000	8.0042e+13
20	-0.0000 0.0000 0.0000 -0.0000 0.0000 -0.0000 0.0000 0.0000 0.0000 0.0000 -0.0000 -0.0001 -0.0001 0.0028 0.0099 -0.0295 -0.1135 0.3971 -0.9137 -0.1215 0.8270	1.2567	0.0000 0.0000 -0.0000 -0.0000 -0.0000 -0.0000 -0.0006 0.0015 0.0313 -0.0121 -0.1839 -0.0192 -0.5230 0.4118 2.6232 -0.1466 3.8113 -4.0100 -5.7583 3.7747 0.0000	3.7841e+20

Conclusion

With and without QR method, it is safe to assume that the degree of the polynomial is 8. However, with QR factorization, the error increases drastically with larger degree of the polynomial (when it diverges from the optimal degree which is 8)

Runge-Kutta Method

Task

$$x_1' = x_2 + x_1(0.5 - x_1^2 - x_2^2)$$
$$x_2' = -x_1 + x_2(0.5 - x_1^2 - x_2^2)$$
$$interval = [0,15] \qquad x_1(0) = 8, \qquad x_2(0) = 8$$

a) **Runge-Kutta Method of 4th order and Adams PC (P_5EC_5E) with different constant step sizes until an optimal step size is found**. Discussion of step sizes illustrated by:

1) two solution curves x2 versus x2 on one plot: first for the optimal step size selection and second for the larger step size (for which the solution visibly differs from the first one)
2) problem solution versus time, obtained for same optimal and larger step sizes.

b) **Runge-Kutta Method of 4th order with variable step size automatically adjusted by algorithm, making error estimation according to step doubling rule**. Discussion of the choses value of minimal step size, absolute and relative tolerances, and the following plots:

1) x2 versus x1
2) problem solution versus time
3) step size versus time
4) error estimate versus time

A flow diagram of the algorithm should also be attached.

Theory

Runge-Kutta Methods

The Runge-Kutta Methods are derived from appropriate Taylor Method in such a way that the final global error is of order O. A trade off is made to perform several function evaluations at each step and eliminate the necessity to compute higher order derivatives. These methods can be constructed for any order N. The Runge-Kutta method of order N = 4 is the most popular. Going higher orders is not necessary because the increased accuracy is offset by additional computation effort. If more accuracy is required, then either a smaller step size or an adaptive method should be used. The RK4 Method simulates the accuracy of the Taylor series method of order N = 4.

A family of Runge-Kutta methods can be defined by the following formula:

$$y_{n+1} = y_n + h\sum_{i=1}^{m} w_i k_i$$
$$k_1 = f(x_n, y_n)$$
$$k_i = f\left(x_n + c_i h, y_n + h\sum_{j=1}^{i-1} a_{ij} k_j\right), \qquad i = 2,3,\dots,m$$
$$\sum_{j=1}^{i-1} a_{ij} = c_i, \qquad i = 2,3,\dots,m$$

To perform a single step of the method the values of the right-hand sides of the equations must be calculated in precisely m times. The coefficients are usually chosen in a way assuring a high order of the method for a given m. Let p(m) denote the maximal possible order of the Runge-Kutta method, then it can be shown that:

$$p(m) = \begin{cases} = m \; m = 1, 2, 3, 4 \\ = m - 1 \; m = 5, 6, 7 \\ \leq m - 2 \; m \geq 8 \end{cases}$$

The methods with m = 4 (and in extension, p(m) = 4 are most used as they provide a good tradeoff between accuracy and computation time. The Runge-Kutta method of order 4 is defined as follows:

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

Step Size Selection

An issue with the practical implementation of numerical methods for solving differential equations is an appropriate procedure for the selection or the correction the step size, h. There are two balancing counteracting factors here:

If step size becomes smaller, then the approximation error of the method becomes smaller. However, it also increases the number of steps necessary to find a solution in the interval. And since the number of arithmetic calculations increases with number of steps, numerical errors also increase.

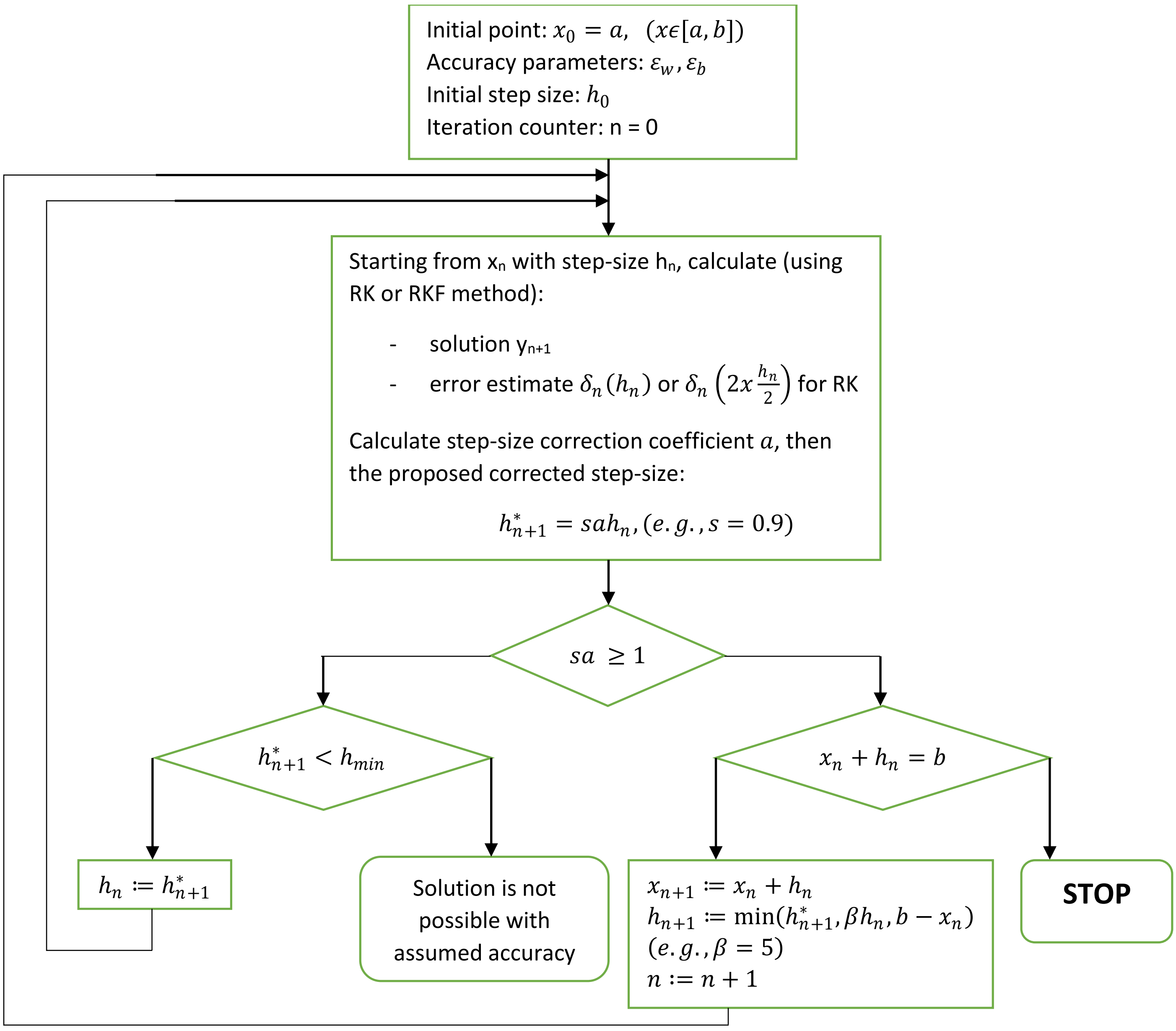
Therefore, small step sizes are not recommended but step size should be sufficiently small enough to perform calculations with desired accuracy. So, a fixed step size could be adequate for solutions varying similarly over an interval.

Error Estimation

In order to find approximation error, for every step of the size h, two additional steps of size h/2 are performed parallelly. We denote $y_n(1)$ as the new point obtained using step size h, and $y_n(2)$ as the new point obtained using step size h/2. Following a number of equations, we are able to derive this formula for error estimate:

$$\begin{aligned} &\delta_n(h) = \frac{2^P}{2^P - 1}\left(y_n^{(2)} - y_n^{(1)}\right) \\ &\delta_n\left(2x\frac{h}{2}\right) = \frac{y_n^{(2)} - y_n^{(1)}}{2^P - 1} \end{aligned}$$

Step Size Correction Algorithm Diagram



Initial point: $x_0 = a, \quad (x\epsilon[a,b])$
Accuracy parameters: $\varepsilon_w, \varepsilon_b$
Initial step size: h_0
Iteration counter: n = 0
Starting from x_n with step-size h_n, calculate (using RK or RKF method):
- solution y_{n+1}
- error estimate $\delta_n(h_n)$ or $\delta_n\left(2x\frac{h_n}{2}\right)$ for RK
Calculate step-size correction coefficient a, then the proposed corrected step-size:
$h^*_{n+1} = sah_n, (e.g., s = 0.9)$
$sa \geq 1$
$h^*_{n+1} < h_{min}$
$x_n + h_n = b$
$h_n := h^*_{n+1}$
Solution is not possible with assumed accuracy
$x_{n+1} := x_n + h_n$
$h_{n+1} := \min(h^*_{n+1}, \beta h_n, b - x_n)$
$(e.g., \beta = 5)$
$n := n + 1$
STOP

Adams PC Method

The Adams method is a numerical method for solving linear first-order ordinary differential equations of the form: y' = f(x,y).

For the P_5EC_5E we first need 5 values calculated from the Runge-Kutta algorithm, then for the next points we use the predictor formula:

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

And we can find β_j from the table below:

k	*β₁*	*β₂*	*β₃*	*β₄*	*β₅*
5	$\frac{1901}{720}$	$-\frac{2774}{720}$	$\frac{2616}{720}$	$-\frac{1274}{720}$	$\frac{251}{720}$

Then we use error correction which defined as follows:

$$r_n(h) \stackrel{\text{def}}{=} \left[\sum_{j=1}^{k} \alpha_j y(x_{n-j}) + h\sum_{j=0}^{k} \beta_j f(x_{n-j}, y(x_{n-j}))\right] - y(x_n)$$
$$y_n - y(x_n) = h\beta_0[f(x_n, y_n) - f(x_n, y(x_n))] + r_n(h)$$

And we can find β_j from the table below:

k	β_0^*	β_1^*	β_2^*	β_3^*	β_4^*	β_5^*
5	$\frac{475}{1440}$	$\frac{1427}{1440}$	$-\frac{798}{1440}$	$\frac{482}{1440}$	$-\frac{173}{1440}$	$\frac{27}{1440}$

We can simplify it into four equations.

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: $f_n = f(x_n, y_n)$

A multistep method is explicit if $\beta_0 = 0$, in this case y_n depends explicitly on values of the solution y and its derivative f(x, y) taken at previously calculated points only.

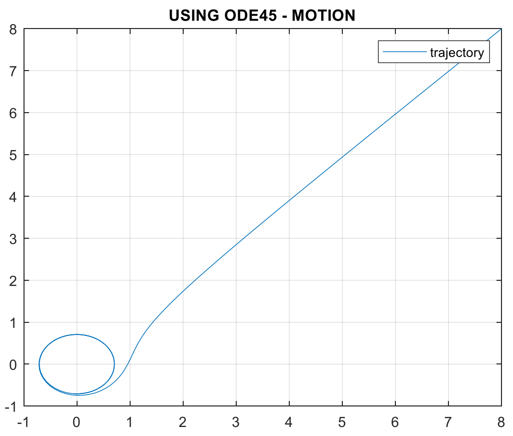
A multistep method is implicit if $\beta_0 \neq 0$, in this case y_n depends on not only previous values but also on $f_n = f(x_n, y_n)$.

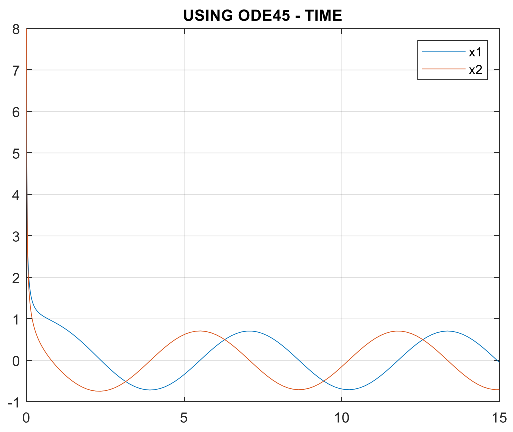
A multistep method is convergent if and only if it is stable at least of order 1. A multistep method with the following properties would be most useful:

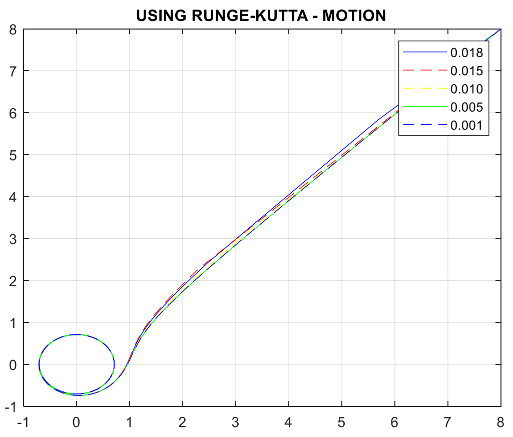
1. A high order and a small error constant.
2. A large set of the absolute stability.
3. A small number of arithmetic operations performed during one iteration.

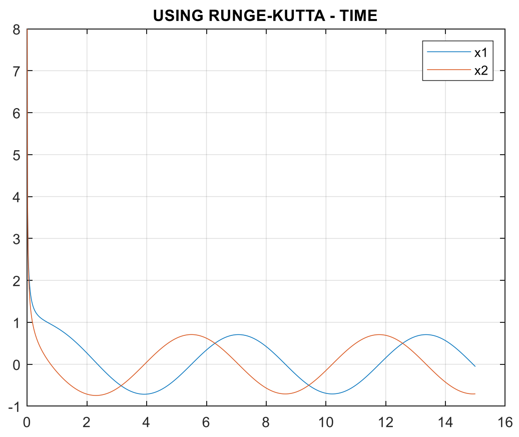
The explicit methods fulfil the last property, but not the first two. The implicit methods fulfil the first two properties, but not the last one.

Results

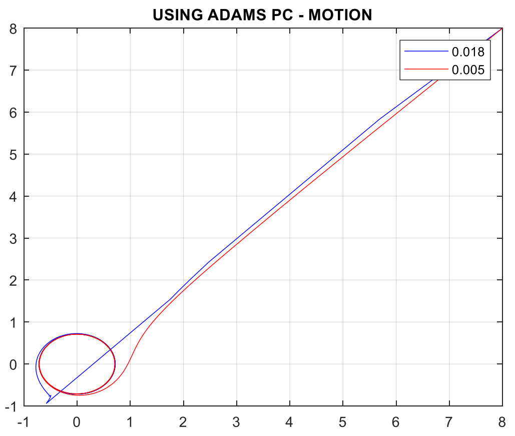


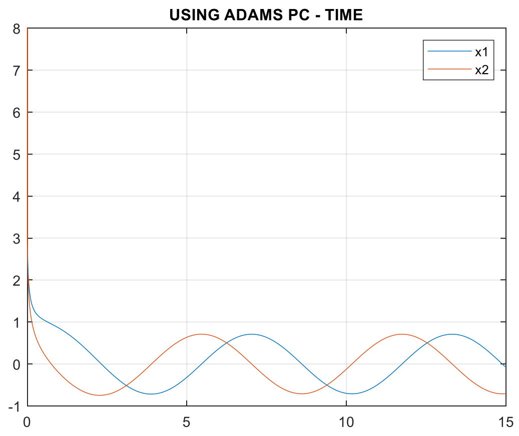






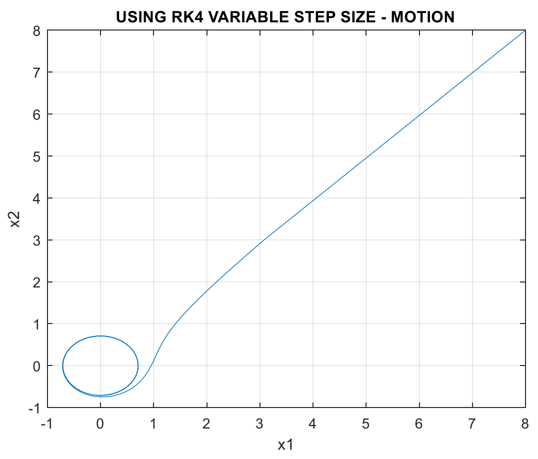
If we go above step-size h = 0.018, the method becomes unstable. And as seen in the graph, step-size of h = 0.005 is optimal as decreasing it to 0.001 shows no improvement in accuracy. Both ode45 and Runge-Kutta method with constant step-size show similar results. For the Adam's PC method, I decided to only use two step-sizes, h = 0.018 and h = 0.005 as they can show the difference between the step sizes far better. And for the time graphs I use the optimal step-size h = 0.005.

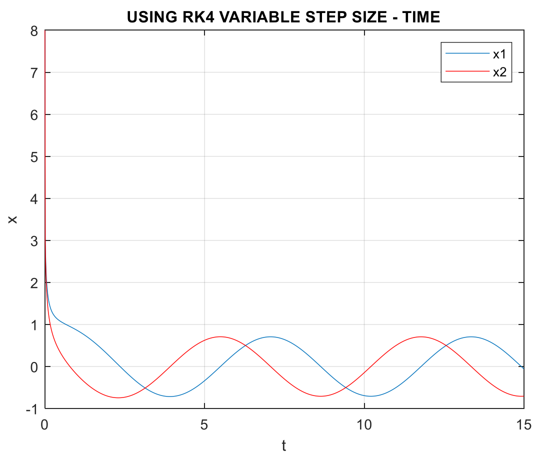


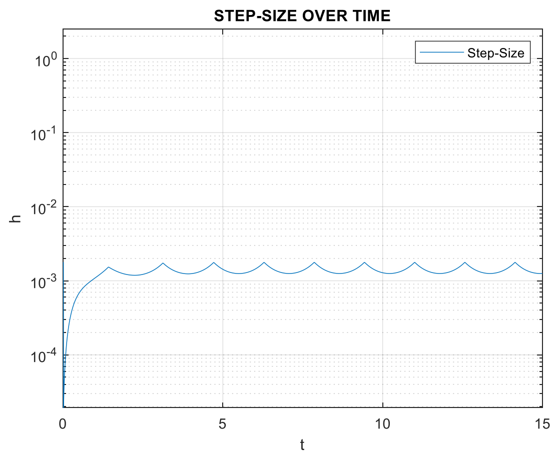


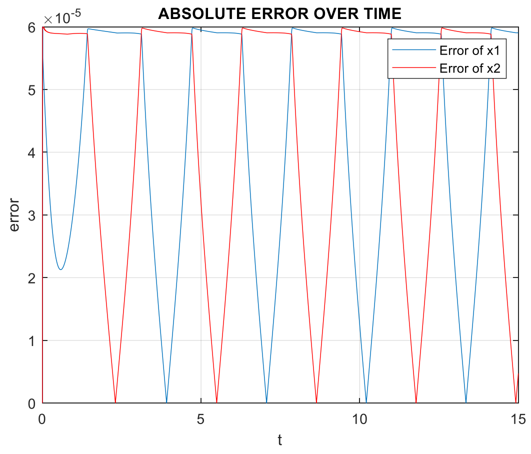
We can see that for greater step-sizes, the Adam's PC method is much more sensitive while the RK4 method is more accurate.

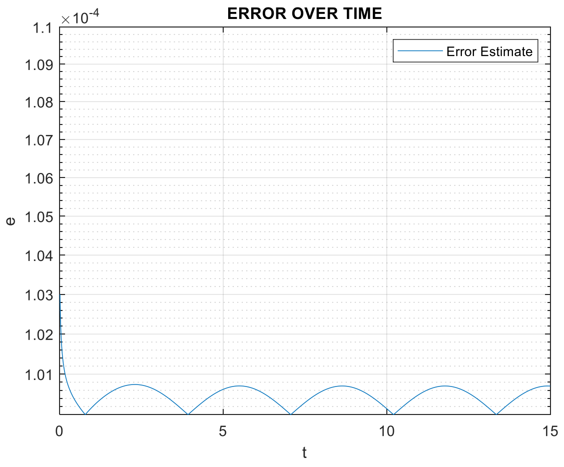
Runge-Kutta with variable step size:











Conclusion

Runge-Kutta method of 4th order with constant step-size is the fastest method. It is also very easy to implement, and I have found similar results with Adam's PC Method as well as MATLAB's inbuilt ode45 function and with variable step sizes. However, RK4 loses precision over time. Therefore, a better solution might be using Runge-Kutta with variable step-size. The issue with variable step-size is that the time of calculations and complexity make it difficult to use in practice. Extra function calls (due to step doubling) can be compensated when the trajectory becomes stable (by increased step-size). But it is worth considering how to iterate, for example the recalculation of the step-size, when chosen wrong, can generate far more function-calls therefore making it more complex.

Appendix

All the MATLAB codes shown on the report were used after `clear` and `clc` commands were used.

Task I: Least Square Approximation

MATLAB code

```
%initialising
clc;
clear;
format short;
%interval
a = -5;
b = 5;
x = (a : b)';
%values of y
y = [   -79.1639
        -40.7900
        -18.7814
        -6.3530
        -0.4392
        0.8270
        0.0585
        -1.7477
        -3.4384
        -6.3580
        -9.3875];
X = a : 0.01 : b;
%different values of polynomial
for n = [1, 2, 3, 4, 5, 7, 10, 15, 20]
    %print degree of polynomial
    display(n);
    % plot graph for method without QR
    display('Method without QR');
    [P, err] = leastSquare(x, y, n);
    display(P);
    display(err);
    figure;
    plot(X, polyval(P,X),x,y,'o');
    xlabel('x');
    ylabel('y');
    title(['Solution without QR Factorisation for n = ' int2str(n)]);
    %plot graph for method with QR
    display('Method with QR');
    [P, err] = QRleastSquare(x,y,n);
    display(P);
    display(err);
    figure;
    plot(X, polyval(P,X),x,y,'o');
    xlabel('x');
    ylabel('y');
    title(['Solution with QR Factorisation for n = ' int2str(n)]);
end
```

Least Square Approximation – `leastSquare.m`

```
%LEAST SQUARE PROBLEM
function [P, err] = leastSquare(x, y, n)
n = n + 1; %for the gram function

M = gram(x,n); %getting matrix formed from gram
A = M' * M; %positive definite symmetric matrix
b = M' * y; %MT with vector y gives solution vector b
P = gauss(A,b)'; %solve the equation with gaussian elimination
err = norm(A * P' - b, 2); %error

end
```

Least Square with QR Factorisation – `QRleastSquare.m`

```
%LEAST SQUARE WITH QR FACTORISATION
function [P, err] = QRleastSquare(x, y, n)
n = n + 1;
M = gram(x, n);
[Q,R] = QR(M); %QR factorization from QR function
b = Q' * y;
P = zeros(n, 1);
for i = n : -1 : 1 %from n to 1
    P(i) = b(i);
    P(i) = P(i) - R(i,(i+1):n) * P((i+1):n);
    P(i) = P(i)/R(i,i);
end
err = norm(M' * M * P - M' * y, 2);
P = P';

end
```

Gram Matrix – `gram.m`

```
%GRAM MATRIX FUNCTION
function M = gram(x,n)
m = length(x);
M = zeros(m,n); %zero matrix thats 11 x degree of polynomial (+1)
v = ones(m,1); %column vector filled with ones with 11 elements
for i = n : -1 : 1 %counts down from degree of polynomial (+1) to 1
    M(:, i) = v; %replaces zero column with v
    v = v .* x; %v is then multiplied with x
end

end
```

Gauss function – `gauss.m`

```
function b = gauss(A,b)
%gauss function
n = length(b);
for i = 1 : (n-1)
    [d, maxi] = max(abs(A(i:n,i))); %maximum value on ith column from ith
value to nth value
    maxi = maxi + i - 1;
    if d < 1e-15
        b = NaN; %if max accuracy is reached b is NaN
        return
    end
    %switching ith row with maxith row
    A([i;maxi],:) = A([maxi;i],:);
    b([i;maxi]) = b([maxi;i]);
    for j = (i+1) : n %row substitution
        d = A(j,i)/A(i,i); %row j by row i
        A(j,:) = A(j,:) - d * A(i,:); %row j = row j - d * row i
        b(j) = b(j) - d * b(i); %solution vector also gets decreased
    end
end
for i = n : -1 : 2 %from n to 2
    b(i) = b(i) / A(i,i);
    j = 1 : (i-1);
    b(j) = b(j) - A(j,i) * b(i);
end
b(1) = b(1) / A(1,1);
end
```

QR Method – `QR.m`

```
%QR Function
function [Q,R] = QR(M)
[m,n] = size(M); %dimensions of the matrix M
Q = zeros(m,n);
R = zeros(n,n);
for i = 1:n
    Q(:,i) = M(:,i);
    R(i,i) = 1;
    N = Q(:,i)' * Q(:,i);
    for j = (i+1):n
        R(i,j) = (Q(:,i)' * M(:,j))/N; %rij = ((ith column of q)' * jth col-
umn of m)/N
        M(:,j) = M(:,j) - R(i,j) * Q(:,i); %j column M = Mj - Rij*ith columnQ
    end
    N = norm(Q(:,i),2);
    Q(:,i) = Q(:,i)/N;
    R(i,i:n) = N * R(i,i:n);
end
end
```

Task II a: Runge-Kutta method of 4th order with constant step size and Adam’s PC

MATLAB code

```
%initialising
clc;clear;
%declaring the set of ordinary differential equations
f1 = @(t,x1,x2) (x2+(x1*(0.5-(x1)^2-(x2)^2)));
f2 = @(t,x1,x2) (-x1+(x2*(0.5-(x1)^2-(x2)^2)));
%initial conidtions
x0 = [8, 8];
a = 0;b = 15;
interval = [a, b];
%using ODE45---------------------------
func=@(t,y) [y(2)+y(1)*(0.5-y(1)^2-y(2)^2);-y(1)+y(2)*(0.5-y(1)^2-y(2)^2)];
[t, y] = ode45(func, interval, x0);
%plotting trajectory
figure(1);
plot(y(:,1),y(:,2));
hold on;grid on;
title('USING ODE45 - MOTION');
legend ('trajectory');
%plotting time
figure(2);
plot(t,y);hold on;grid on;
title('USING ODE45 - TIME');legend('x1','x2');
%plotting for RK4 different constant step sizes------------------------
figure(3);
for h = [0.018,0.015,0.01,0.005,0.001]
    [x1,x2] = RK4CStep(a,b,h,f1,f2,x0(1),x0(2));
    switch h
        case 0.018
            plot(x1,x2,'b');
        case 0.015
            plot(x1,x2,'r--');
        case 0.010
            plot(x1,x2,'y--');
        case 0.005
            plot(x1,x2,'g');
        case 0.001
            plot(x1,x2,'b--');
    end
    grid on;hold on;
end
legend('0.018','0.015','0.010','0.005','0.001');
title('USING RUNGE-KUTTA - MOTION');
%plotting with time
figure(4);
[x1,x2, t] = RK4CStep(a,b,0.005,f1,f2,x0(1),x0(2));
plot(t,x1);hold on;grid on;plot(t,x2);
title('USING RUNGE-KUTTA - TIME');legend ('x1','x2');



%continued on next page
```

```
%using adams PC------------------------------------
figure(5);
for h = [0.018,0.005]
    [x1,x2] = AdamsPC(a,b,h,f1,f2,x0(1),x0(2));
    switch h
        case 0.018
            plot(x1,x2,'b');
        case 0.005
            plot(x1,x2,'r');
    end
    grid on; hold on;
end
legend('0.018','0.005');
title('USING ADAMS PC - MOTION');
%plotting with time
figure(6);
[x1,x2, t] = AdamsPC(a,b,0.005,f1,f2,x0(1),x0(2));
plot(t,x1);
hold on;grid on;
plot(t,x2);
title('USING ADAMS PC - TIME');
legend ('x1','x2');
```

Runge-Kutta Method of 4th Order with Constant Step Size – `RK4CStep.m`

```
%runge kutta constant step
%start, stop = interval
%h = step size
%f1, f2 = function 1 and 2
%y1, y2 = initial conditions
%x1, x2 = values of x1, x2
%t = time
function [x1, x2, t] = RK4CStep(start,stop,h,f1,f2,y1,y2)

%number of steps in interval
iter = ceil(stop/h);

%preallocation
t = zeros(1,iter);
x1 = zeros(1,iter);
x2 = zeros(1,iter);

%initial conditions
t(1) = start;
x1(1) = y1;
x2(1) = y2;

%loops through all points
for i = 1:iter
    t(i+1) = t(i) + h;
    %runge kutta of 4th order
    k11 = f1(t(i),x1(i),x2(i));
    k21 = f2(t(i),x1(i),x2(i));
    k12 = f1(t(i)+h/2,x1(i)+(h/2)*k11,x2(i)+(h/2)*k21);
    k22 = f2(t(i)+h/2,x1(i)+(h/2)*k11,x2(i)+(h/2)*k21);
    k13 = f1(t(i)+h/2,x1(i)+(h/2)*k12,x2(i)+(h/2)*k22);
    k23 = f2(t(i)+h/2,x1(i)+(h/2)*k12,x2(i)+(h/2)*k22);
    k14 = f1(t(i)+h,x1(i)+h*k13,x2(i)+h*k23);
    k24 = f2(t(i)+h,x1(i)+h*k13,x2(i)+h*k23);

    x1(i+1) = x1(i) + (h/6)*(k11 + 2*k12 + 2*k13 + k14);
    x2(i+1) = x2(i) + (h/6)*(k21 + 2*k22 + 2*k23 + k24);
end

end
```

Adam's PC Method – `AdamsPC.m`

```
%Adams PC Method
%start, stop = interval
%h = step size
%f1, f2 = function 1 and 2
%y1, y2 = initial conditions
%x1, x2 = values of x1, x2
%t = time
function [x1, x2, t] = AdamsPC(start, stop, h, f1, f2, y1, y2)
B0 = 475/1440; %B0 from table
k = 5; %P5EC5E, k = 5
tfin = (k*h)-start; %for 5point runge-kutta approximation
iter = ceil((stop - start)/h);
t = zeros(1,iter);%preallocation
[x1, x2] = RK4CStep(start,tfin,h,f1,f2,y1,y2);%initial points from RK4
for i=6:iter
    t(1+i) = t(i)+h;
    %prediction - explicit sum
    x1(i+1) = x1(i) + h*explicitSum(k,f1,t,i,x1,x2);
    x2(i+1) = x2(i) + h*explicitSum(k,f2,t,i,x1,x2);
    %evaluation
    fn1 = f1(t(i+1),x1(i+1),x2(i+1));
    fn2 = f2(t(i+1),x1(i+1),x2(i+1));
    %correction - implicit sum
    x1(i+1) = x1(i) + h*implicitSum(k,f1,t,i,x1,x2)+h*B0*fn1;
    x2(i+1) = x2(i) + h*implicitSum(k,f2,t,i,x1,x2)+h*B0*fn2;
end
end
```

Explicit Sum – `explicitSum.m`

```
function sum = explicitSum(k,f,t,i,x1,x2)
sum = 0;
B = [1901, -2774, 2616, -1274, 251];
B = B/720;
for j=1:k
    sum = sum + B(j)*f(t(i-j),x1(i-j),x2(i-j));
end
end
```

Imlicit Sum – `implicitSum.m`

```
function sum = implicitSum(k, f,t,i,x1,x2)
sum = 0;
B = [1427, -798, 482, -173, 27];
B = B/1440;
for j=1:k
    sum = sum + B(j) * f(t(i-j),x1(i-j),x2(i-j));
end
end
```

Task II b: Runge-Kutta method of 4th order with variable step size

MATLAB code

```
%initialising
clc;clear;
%initial conidtions
x0 = [8, 8];
a = 0;b = 15;
interval = [a, b];
h0 = 0.01; %initial step
%declaring the functions
func=@(t,y) [y(2)+y(1)*(0.5-y(1)^2-y(2)^2);-y(1)+y(2)*(0.5-y(1)^2-y(2)^2)];
%error tolerances
eps_rel = 10^-6;%relative tolerance
eps_abs = 10^-4;%absolute tolerance
hmin = 10^-7;%minimum step size
h(2) = h0;
t(1) = a;
n = (b-a)/h(2);
x = x0';
t(2) = t(1) + h(2);
xtmp(:,1)=x(:,1);
i = 2;
while t<= b %while we are in range keep looping algorithm
    incrh = true;
    %Runge kutta method
    tj = t(i);
    k1 = h(i)*feval(func,tj,x(:,i-1));
    k2 = h(i)*feval(func,tj+h(i)/2,x(:,i-1)+k1/2);
    k3 = h(i)*feval(func,tj+h(i)/2,x(:,i-1)+k2/2);
    k4 = h(i)*feval(func,tj+h(i),x(:,i-1)+k3);
    x(:,i) = x(:,i-1) + (k1+2*k2+2*k3+k4)/6;
    %error calculation
    d1(i) = (x(1,i)-xtmp(1,1))/((2^4)-1);
    d2(i) = (x(2,i)-xtmp(2,1))/((2^4)-1);
    xtmp(:,1) = x(:,i);
    %calculation sa1 = s*alpha
    e(i) = (abs(x(1,i)))*eps_rel+eps_abs;
    sa1 = (e(i)/abs(d1(i)))^(1/5)*0.9;
    e(i) = (abs(x(2,i)))*eps_rel+eps_abs;
    sa2 = (e(i)/abs(d2(i)))^(1/5)*0.9;
    %selecting smaller sa as the new step size
    if sa1 < sa2
        nh = h(i)*sa1;
    else
        nh = h(i)*sa2;
    end
    %increasing until sa is smaller than 1
    while((sa1<1)&&(sa2<1))
        incrh = false;
        if nh>hmin
            h(i) = nh;
            break;
        else %if step size is smaller than hmin
            disp('Accuracy not possible to satisfy');
            break;
        end
    end
%continued on next page...
```

```
    if incrh == true %assigning new step sizes
        h(i+1) = min(nh,3*h(i));
    else
        h(i+1) = h(i);
    end
    if t(i)+h(i+1)>b
        break %if interval end is met then break
    end
    i = i+1;
    t(i) = t(i-1)+h(i);
end
%All figures

figure(1);
plot(t,x(1,:));
axis([0 15 -1 8])
hold on;grid on;
title('USING RK4 VARIABLE STEP SIZE - TIME');
xlabel('t');ylabel('x');
plot(t,x(2,:),'r');
legend('x1','x2');

figure(2);
plot(x(1,:),x(2,:));
hold on;grid on;
xlabel('x1'); ylabel('x2');
title('USING RK4 VARIABLE STEP SIZE - MOTION');

figure(3);
semilogy(t,h(1:length(t)));
axis([0 15 0 2.5])
grid on
title('STEP-SIZE OVER TIME');
xlabel('t');ylabel('h');
legend('Step-Size');

figure(4);
semilogy(t,e(1:length(t)));
axis([0 15 0 1.1*10^(-4)])
grid on
title('ERROR OVER TIME');
xlabel('t');ylabel('e');
legend('Error Estimate');

figure(5);
plot(t,abs(d1));
axis([0 15 0 6*10^(-5)])
hold on
grid on
title('ABSOLUTE ERROR OVER TIME');
xlabel('t');ylabel('error');
plot(t,abs(d2),'r');
legend('Error of x1','Error of x2');
```

Bibliography

Numerical Methods – Piotr Tatjewski

MATLAB Primer

MATLAB Documentation

Wikipedia