1 (b).

function [z] = q1b(t, z\_0)

% Calculate denominator

z\_de = (z\_0(1).^2 + z\_0(2).^2).^(3/2);

% Calculate z

z\_1 = z\_0(3);

z\_2 = z\_0(4);

z\_3 = (z\_0(1) / z\_de)\*(-1);

z\_4 = (z\_0(2) / z\_de)\*(-1);

% Return

z = [z\_1; z\_2; z\_3; z\_4]';

end

**q1b.m**

1 (c).

% clear figure

clf;

% initial values

t\_0 = 0;

t\_f = 2\*pi;

N = 500000;

z\_0 = [0.4 0.0 0.0 2.0];

% calculate Z

Z = ForwardEuler(@q1b, t\_0, t\_f, N, z\_0);

z\_x = Z(:, 1);

z\_y = Z(:, 2);

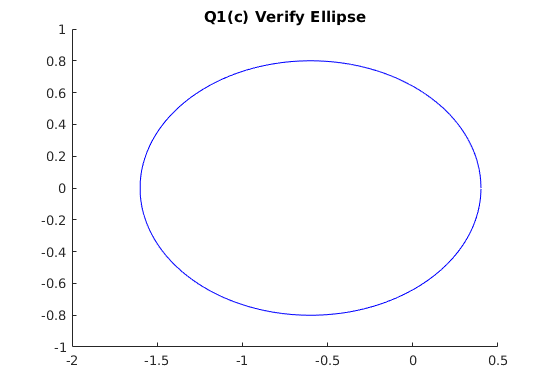
% plot

hold on;

plot(z\_x, z\_y);

title('Q1(c) Verify Ellipse');

**q1c.m**

****

So, it is indeed an ellipse.

2.

Shared Matlab code is presented here:

function [y] = q2func(t, y\_0)

y = -2 \* t \* y\_0.^2;

end

**q2func.m**

function [y] = q2func\_exact(t)

y = 1 / (1 + t.^2);

end

**q2func\_exact.m**

function [err] = q2error(f\_n, t, f\_exact)

f\_n\_exact = f\_exact(t);

err = f\_n - f\_n\_exact;

end

**q2error.m**

2 (a).

function [y] = MidpointRule (f, t\_0, t\_f, N, y\_0)

% allocate space

y = zeros(N+1, length(y\_0));

y(1, :) = y\_0';

% calculate h

h = (t\_f - t\_0) / N;

% iterate

for n = 1:N

t = t\_0 + (n-1) \* h;

k\_1 = f(t, y\_0) \* h;

k\_2 = f(t+h/2, y\_0+k\_1/2) \* h;

y\_0 = y\_0 + k\_2;

y(n+1, :) = y\_0';

end

end

**MidpointRule.m**

2 (b).

% clear figure

clf;

% initial values

t\_0 = 0;

t\_f = 4;

N = 40;

y\_0 = 1;

h = (t\_f - t\_0) / N;

% calculate Y

y = ForwardEuler(@q2func, t\_0, t\_f, N, y\_0);

t = t\_0:h:t\_f;

% plot

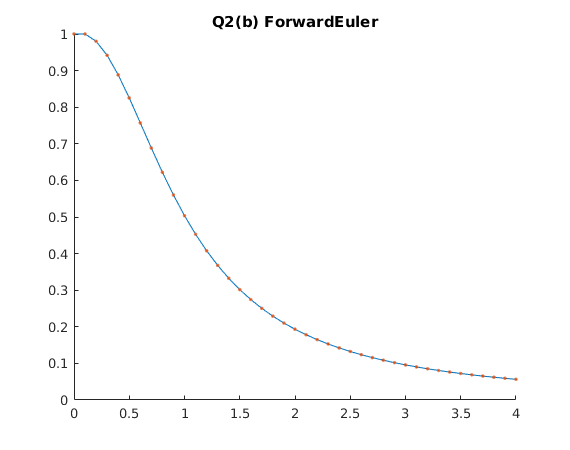
hold on;

plot(t, y);

plot(t, y, '.');

title('Q2(b) ForwardEuler');

**q2b.m**



2 (c).

% clear figure

clf;

% initial values

t\_0 = 0;

t\_f = 4;

N = 40;

y\_0 = 1;

h = (t\_f - t\_0) / N;

% calculate Y

y = MidpointRule(@q2func, t\_0, t\_f, N, y\_0);

t = t\_0:h:t\_f;

% plot

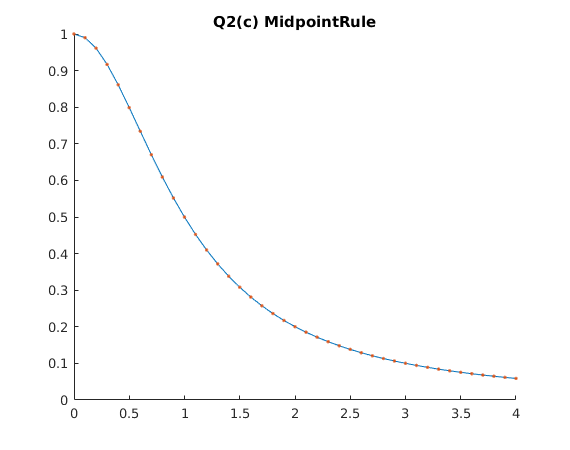
hold on;

plot(t, y);

plot(t, y, '.');

title('Q2(c) MidpointRule');

**q2c.m**

****

2 (d).

% initial values

y\_0 = 1;

t\_0 = 0;

t\_f = 4;

N = zeros(11, 1);

for i = 0:10

N(i+1) = 2.^i\*10;

end

% calculate errors

error = zeros(11, 1);

for i = 1:11

y = MidpointRule(@q2func, t\_0, t\_f, N(i), y\_0);

f\_n = y(N(i) + 1);

error(i) = q2error(f\_n, t\_f, @q2func\_exact);

end

disp('Error:');

disp(error);

% calculate error ratios & p-values

error\_ratio = zeros(10, 1);

p\_val = zeros(10, 1);

for i = 1:10

error\_ratio(i) = error(i + 1) / error(i);

p\_val(i) = log2(1 / error\_ratio(i));

end

disp('Error Ratio:');

disp(error\_ratio);

disp('p:')

disp(p\_val);

**q2d.m**

Experiment evidences:

|  |  |  |  |
| --- | --- | --- | --- |
| **i** | **Error** | **Error ratio** | **p** |
| 0 | -0.009195833455462 | 0.503344940427460 | 0.990380682354790 |
| 1 | -0.004628676242821 | 0.500494440818975 | 0.998574050296498 |
| 2 | -0.002316626727883 | 0.500091451785889 | 0.999736150052849 |
| 3 | -0.001158525223593 | 0.500022110403486 | 0.999936204271617 |
| 4 | -0.000579288227257 | 0.500006927655467 | 0.999980011150101 |
| 5 | -0.000289648126738 | 0.500002646196381 | 0.999992364711411 |
| 6 | -0.000144824829835 | 0.500001145002375 | 0.999996696225288 |
| 7 | -0.000072412580742 | 0.500000531238298 | 0.999998467171098 |
| 8 | -0.000036206328839 | 0.500000255706248 | 0.999999262187917 |
| 9 | -0.000018103173678 | 0.500000125432825 | 0.999999638077417 |
| 10 | -0.000009051589110 | 0.503344940427460 | 0.990380682354790 |

So, p approximately equals to 1.

Forward Euler method is first order method.

2 (e).

% initial values

y\_0 = 1;

t\_0 = 0;

t\_f = 4;

N = zeros(11, 1);

for i = 0:10

N(i+1) = 2.^i\*10;

end

% calculate errors

error = zeros(11, 1);

for i = 1:11

y = MidpointRule(@q2func, t\_0, t\_f, N(i), y\_0);

f\_n = y(N(i) + 1);

error(i) = q2error(f\_n, t\_f, @q2func\_exact);

end

disp('Error:');

disp(error);

% calculate error ratios & p-values

error\_ratio = zeros(10, 1);

p\_val = zeros(10, 1);

for i = 1:10

error\_ratio(i) = error(i + 1) / error(i);

p\_val(i) = log2(1 / error\_ratio(i));

end

disp('Error Ratio:');

disp(error\_ratio);

disp('p:')

disp(p\_val);

**q2e.m**

Experiment evidences:

|  |  |  |  |
| --- | --- | --- | --- |
| **i** | **Error** | **Error ratio** | **p** |
| 0 | 0.002477321222346 | 0.195801910441708 | 2.352533253483385 |
| 1 | 0.000485064228113 | 0.224080254106996 | 2.157912570345280 |
| 2 | 0.000108693315494 | 0.237558895391946 | 2.073642865416771 |
| 3 | 0.000025821063965 | 0.243912664051205 | 2.035563429456995 |
| 4 | 0.000006298084500 | 0.246989210288278 | 2.017480075764028 |
| 5 | 0.000001555558917 | 0.248502731007491 | 2.008666388012017 |
| 6 | 0.000000386560639 | 0.249253382802279 | 2.004315010248027 |
| 7 | 0.000000096351547 | 0.249627193799754 | 2.002152988319292 |
| 8 | 0.000000024051966 | 0.249813718695707 | 2.001075389155217 |
| 9 | 0.000000006008511 | 0.249906923892663 | 2.000537221765148 |
| 10 | 0.000000001501569 | 0.195801910441708 | 2.352533253483385 |

So, p approximately equals to 2.

Midpoint Rule method is second order method.

5.

Shared Matlab code is presented here:

function theta\_dir = dire(x\_t, y\_t, x\_p, y\_p, theta\_p)

%input:

% x\_p x-coord pursuer

% y\_p y-coord pursuer

% theta\_p direction of motion pursuer

% (angle to x - axis, radians)

% x\_t x-coord target

% y\_t y-coord target

%output:

% theta\_dir direction from pursure to target

% (angle to x - axis, radians)

x\_diff = x\_t - x\_p;

y\_diff = y\_t - y\_p;

theta\_dir = atan2( y\_diff, x\_diff); %range [-pi, +pi]

if( theta\_dir < 0.0)

theta\_dir = theta\_dir + 2.\*pi; % now range [0,2\*pi]

end

theta\_dir = theta\_dir + 2\*pi\*floor( theta\_p/(2.\*pi) );

% now, theta\_dir is in the same interval as theta\_pursuer,

% k\*2\*pi <= theta\_pursuer <= (k+1)\*2\*pi

% k\*2\*pi <= theta\_dir <= (k+1)\*2\*pi

%

% make sure that we turn in the shortest

% way to point in the right direction, i.e. we don't

% turn the "long way round"

if( (theta\_p - theta\_dir) > pi)

theta\_dir = theta\_dir + 2\*pi;

elseif( theta\_dir - theta\_p > pi)

theta\_dir = theta\_dir - 2\*pi;

end

**dire.m**

5 (a).

global METHOD RT SP ST D HD AST

% global values

METHOD = 1; % 1 => ode15s; otherwise => ode45

RT = 0.25;

SP = 2.0;

ST = 1.0;

D = 0.001;

HD = 0.01;

AST = 0.1;

% local values

y\_0 = [0.0; 0.0; -pi; 5.0; 5.0; 0.0]; % pursuer | target

t\_0 = 0.0;

t\_f = 20.0;

tol = 0.000001;

% ode solver options

options = odeset('AbsTol', tol,'RelTol',tol,'MaxOrder',5,'Stats','on',...

'events', 'on', 'Refine',4);

% solve odes

if METHOD == 1

[t,y] = ode15s('q5a\_fun',[t\_0,t\_f],y\_0,options);

else

[t,y] = ode45('q5a\_fun',[t\_0,t\_f],y\_0,options);

end;

% plot the trajectory of the pursuer

figure(2);

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

title('The Trajectory of the Pursuer');

xlabel('x(t)'); % x-axis label

ylabel('y(t)'); % y-axis label

% plot the trajectory of the target

figure(3);

plot(y(:, 4), y(:, 5));

title('The Trajectory of the Target');

xlabel('x(t)'); % x-axis label

ylabel('y(t)'); % y-axis label

% show hitting time

disp('Hitting time: ');

disp(t(length(t)));

**q5a.m**

function [dP,halt,direction] = q5a\_fun(t,P,flag)

% P is position 6-d vector [x\_p, y\_p, theta\_p, x\_t, y\_t, theta\_t]

% dP is the change of P

global RT SP ST D HD AST

pp = [P(1); P(2); P(3)]; % pursuer position

tp = [P(4); P(5); P(6)]; % target position

% calculate distance between pursuer and target

dp = tp - pp;

distance = norm([dp(1) dp(2)], 2);

if nargin<3 | isempty(flag)

% calculate theta\_d

theta\_d = dire(P(4), P(5), P(1), P(2), P(3));

% calculate pursuer position change

dP(1) = SP\*cos(P(3));

dP(2) = SP\*sin(P(3));

dP(3) = SP\*((theta\_d - P(3) / (abs(theta\_d - P(3)) + D)));

% calculate target position change

dP(4) = ST\*cos(P(6));

dP(5) = ST\*sin(P(6));

dP(6) = ST/RT\*AST;

dP = dP';

elseif strcmp(flag,'events')

dP = distance-HD;

halt = 1;

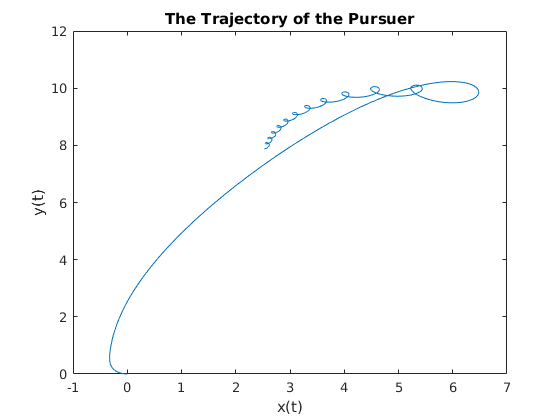
direction = 0;

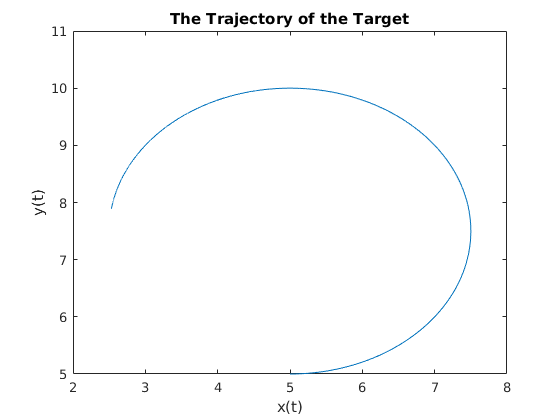
end;

**q5a\_fun.m**

The hitting time for 5(a) is **11.3895**;

Plots are showed in the next page.

****

****

5 (b).

|  |  |  |
| --- | --- | --- |
| **tol** | **Hitting Time** | **Number of Function Evaluations** |
|  | 11.2298 | 897 |
|  | 11.2228 | 1040 |
|  | 11.3910 | 1402 |
|  | 11.3895 | 1874 |
|  | 11.3895 | 2402 |
|  | 11.3895 | 3140 |
|  | 11.3895 | 4307 |

With decreasing error tolerance, hitting time is approaching to a fixed value “11.3895”, and the number of function evaluations is increasing.

From that, it is able to conclude that smaller error tolerance make hitting time closer to the real value, but this needs more number of function evaluations.

5 (c).

|  |  |  |
| --- | --- | --- |
| **tol** | **Hitting Time** | **Number of Function Evaluations** |
|  | 11.3749 | 1207 |
|  | 11.5500 | 1675 |
|  | 11.5510 | 2455 |
|  | 11.3895 | 3571 |

Compared with the data with using *ode15s*, *ode45* uses larger number of function evaluations under same error tolerances. Also, *ode45* needs smaller error tolerance than *ode15s* to get to stable hitting time “11.3895”.

From that, it is able to conclude that *ode15s* is more accurate than *ode45* when evaluating the ODEs of this question.