1 (b).

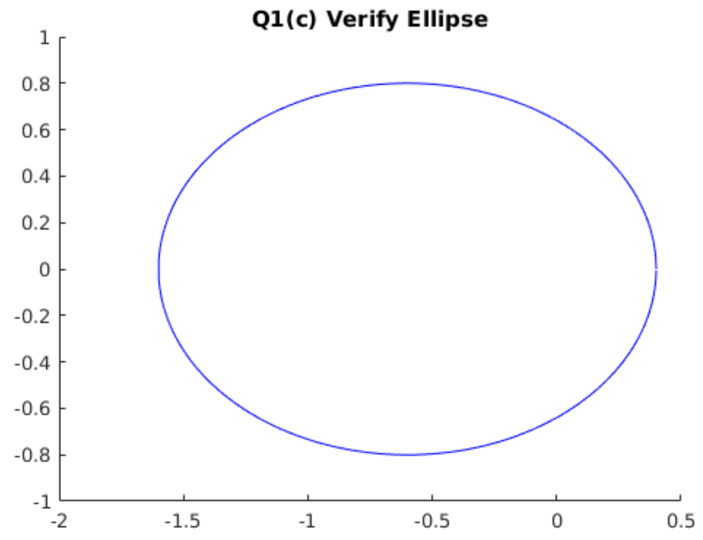
```
function [z] = q1b(t, z_0)
	% Calculate denominator
	z_de = (z_0(1).^2 + z_0(2).^2).^(3/2);
	% Calculate z
	z_1 = z_0(3);
	z_2 = z_0(4);
	z_3 = (z_0(1) / z_de)*(-1);
	z_4 = (z_0(2) / z_de)*(-1);
	% Return
	z = [z_1; z_2; z_3; z_4]';
end
```

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).

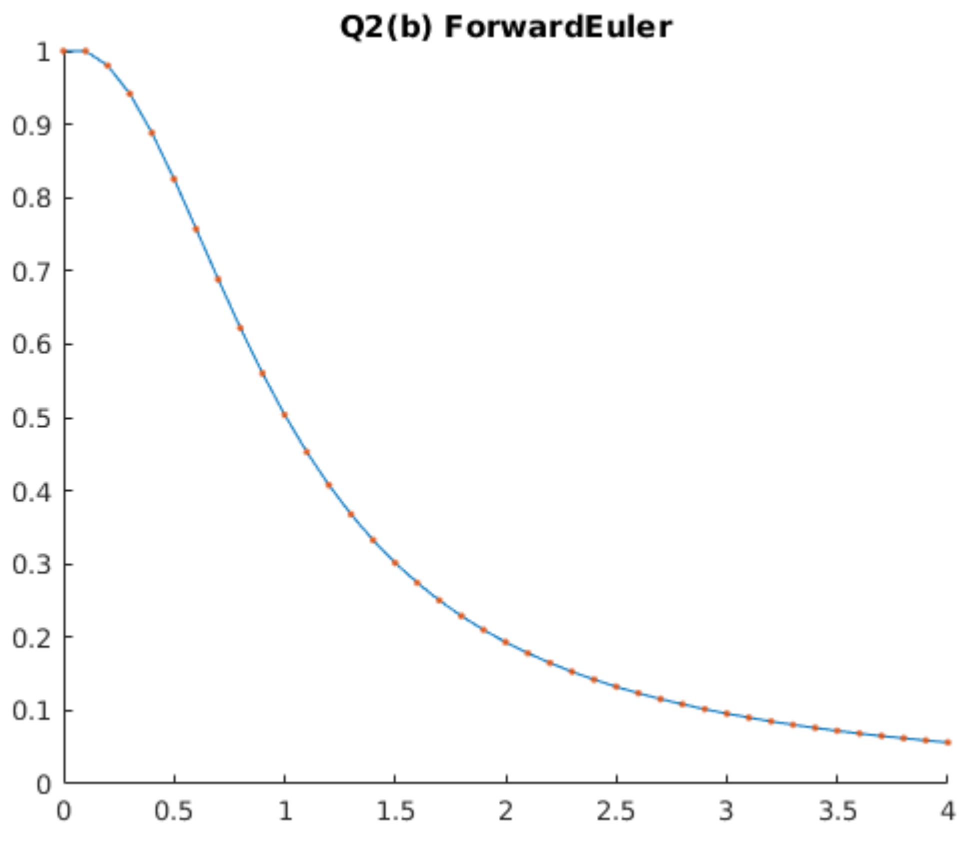
```
function [y] = MidpointRule (f, t_0, t_f, N, y_0)
        % allocate space
        y = zeros(N+1, length(y_0));
        y(1, :) = y_0';
        % calculate h
        h = (t_f - t_0) / N;
        % iterate
        for n = 1:N
                t = t_0 + (n-1) * h;
                k_1 = f(t, y_0) * h;
                 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');
```

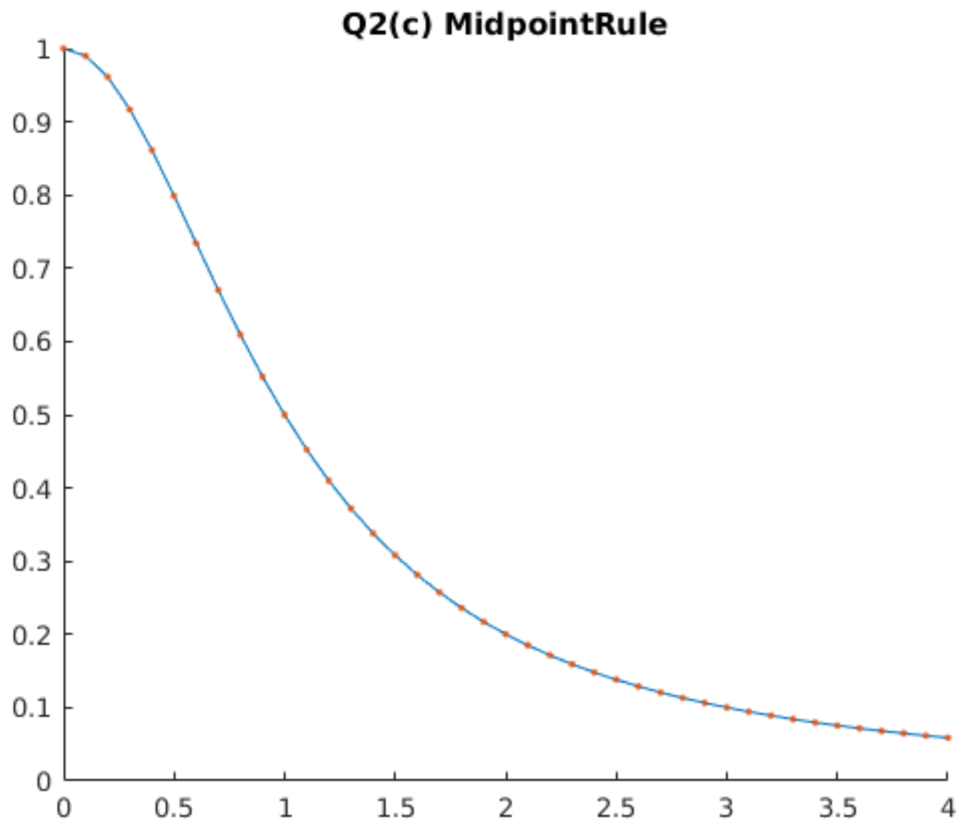
q2b.m



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)));
```

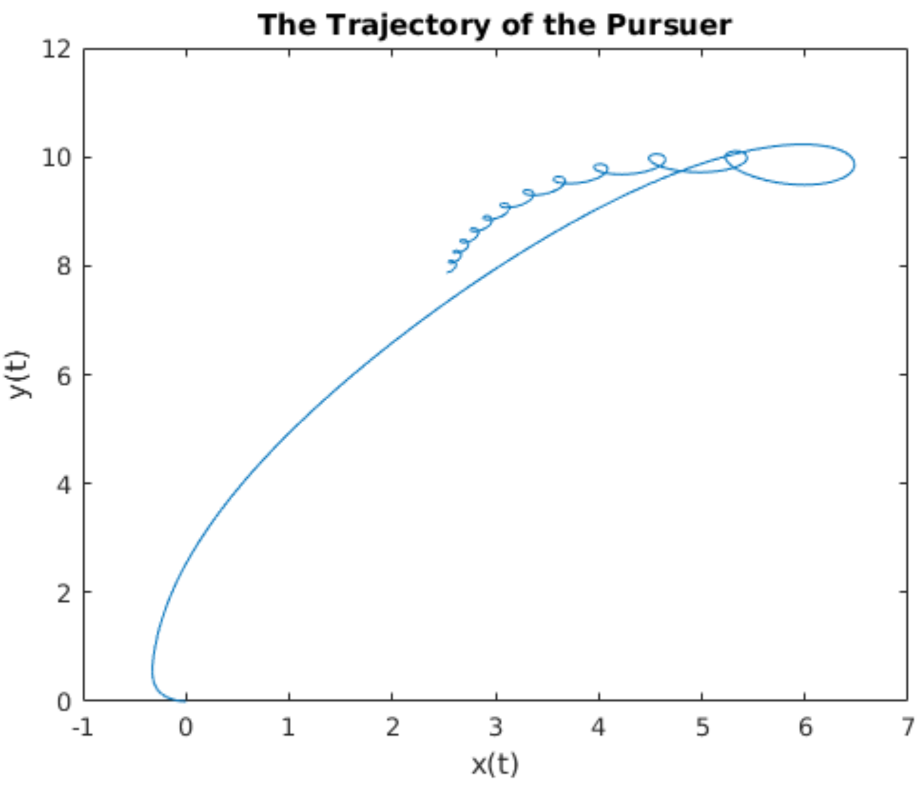
q5a.m

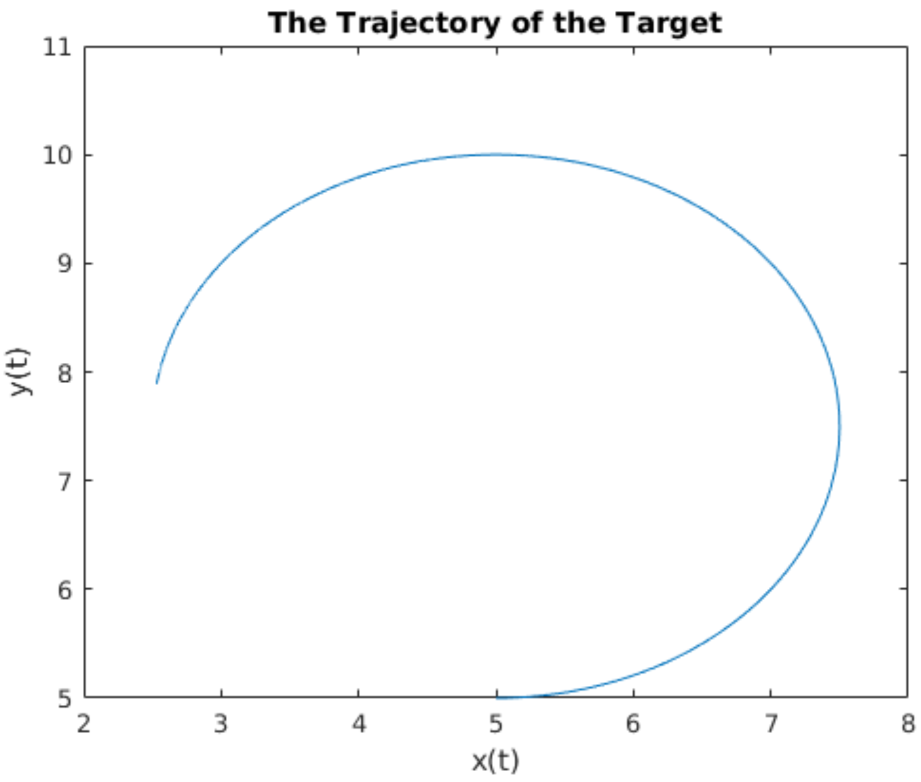
```
function [dP,halt,direction] = q5a_fun(t,P,flag)
% P is position 6-d vector [x_p, y_p, theta_p, x_t, y_t, theta_t]
% dP is the change of P

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.


The Trajectory of the Pursuer
y(t)
x(t)
12
10
8
6
4
2
0
-1
0
1
2
3
4
5
6
7


The Trajectory of the Target
y(t)
x(t)
11
10
9
8
7
6
5
2
3
4
5
6
7
8

5 (b).

tol	Hitting Time	Number of Function Evaluations
10^{-3}	11.2298	897
10^{-4}	11.2228	1040
10^{-5}	11.3910	1402
10^{-6}	11.3895	1874
10^{-7}	11.3895	2402
10^{-8}	11.3895	3140
10^{-9}	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
10^{-4}	11.3749	1207
10^{-5}	11.5500	1675
10^{-6}	11.5510	2455
10^{-7}	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.