COMP0128 Coursework 2

Jian Zhou

Q1:

There are two files containing code, "$Drone.m$" and "$quadcopter\_script.m$" . Drone.m is a class file, which defines the properties and methods a specific class has. While the other file is a script that helps to instantiate the class and makes the whole code work.

"$quadcopter\_script.m$"

The following picture shows the main part in file "$quadcopter\_script.m$". It calls $updates()$ method in class $Drone$ to iterate until time reaches to 8 seconds.

```
while(drones(1).time < 8)
    % clear axis
    cla(ax1);


    %% main: update and draw drones
    for i = 1:num_drones
        % Jian 15/12/2022: I changed update(drones(1)) to drones.update()
        % for better code readability
        drones.update();
    end

    %% optionally
    if(draw_ground) % draw the ground imageâ
        imagesc([-spaceDim,spaceDim],[-spaceDim,spaceDim],ground_img);
    end

    camlight %apply fancy lighting

    %update figure
    % Jian 15/12/2022: I commented the line "drawnow",
    % because, in drones.update(),there has already been a 'drones.draw'
    drawnow
end
```

Figure1: quadcopter.m

"$Drone.m$"

```
        function update(obj)
            %update simulation time
            obj.time = obj.time + obj.time_interval;

            %change position and orientation of drone
           %% input
            % 1.b.
            num_gamma=obj.m*obj.g/4/obj.k;
            inputs=[num_gamma;num_gamma;num_gamma;num_gamma];

            % 1.c.
            if (obj.time>2)&&(obj.time<=4)
                inputs=inputs*1.15;
            end
            if (obj.time>4)&&(obj.time<=8)
                inputs(4)=0;
            end
            %% update kinematicss
            kinematics(obj,inputs);
            %draw drone on figure
            draw(obj);
        end
```

Figure 2: update function

Instance Drone will response to the call of function $update()$. Firstly, it will add 1 to the timer. Then, for this task, it will set the inputs to default vector, which is [0.49;0.49;0.49;0.49]. The number of 0.49 is the square of angular velocity of a single propeller and it is the parameter that we can access to. As is shown in figure 2, the inputs are increased by 15% when timer is at the range of (2,4) seconds. For the next four seconds, the power we apply to γ_3 is removed.

In the end, we call $kinematics()$ to update the state of the model. The details of kinematics are shown below in figure 3.

```
        function T=thrust(~,inputs,k) ...

        function tau=torques(~,inputs,L,b,k) ...

        function a= acceleration(obj,inputs,angles,xdot,m,g,k,kd) ...

        function omegadot=angular_acceleration(obj,inputs,omega,I,L,b,k) ...

        function R=rotation(~,Theta) ...

        function omega=thetadot2omega(~,thetadot,theta) ...

        function thetadot=omega2thetadot(~,omega,theta) ...

        function kinematics(obj,inputs)
        %% kinematics
            dt=obj.time_interval;
            obj.omega=thetadot2omega(obj,obj.thetadot,obj.theta);
            %Compute linear and angular accelerations
            a=acceleration(obj,inputs,obj.theta,obj.xdot,obj.m,obj.g,obj.k,obj.kd);
            omegadot=angular_acceleration(obj,inputs,obj.omega,obj.I,obj.L,obj.b,obj.k);
            obj.omega=obj.omega+dt*omegadot;
            obj.thetadot=omega2thetadot(obj,obj.omega,obj.theta);
            obj.theta=obj.theta+dt*obj.thetadot;
            obj.xdot=obj.xdot+dt*a;
            obj.pos=obj.pos+dt*obj.xdot;


            obj.posRecord = [obj.posRecord , obj.pos];
            obj.orienRecord = [obj.orienRecord , obj.theta];
        end
```

Figure 3: Tool functions and kinematics function

Firstly, as we only have gyro to observe the drone, we can only access to angular velocity $(\dot{\phi}, \dot{\theta}, \dot{\psi})$. We assume the sensor is precise enough and hence can calculate theta. Basing on thetadot and theta, we can therefore calculate acceleration and omega. Using linear information, the actual position can be calculated by discrete integral. All other tool functions are shown below in figure 4 and the specific explanation can be found in [1].

```
function T=thrust(~,inputs,k)
T=[0;0;k*sum(inputs)];
end

function tau=torques(~,inputs,L,b,k)
    % Inputs are values for omega^2
    tau = [L * k * (inputs(1) - inputs(3))
           L * k * (inputs(2) - inputs(4))
           b * (inputs(1) - inputs(2) + inputs(3) - inputs(4))
           ];
end

function a= acceleration(obj,inputs,angles,xdot,m,g,k,kd)
    gravity=[0;0;-g];
    obj.R=rotation(obj,angles);
    T=obj.R*thrust(obj,inputs,k);
    Fd=-kd*xdot;
    a=gravity+1/m*T+Fd;
end

function omegadot=angular_acceleration(obj,inputs,omega,I,L,b,k)
    tau=torques(obj,inputs,L,b,k);
    omegadot=inv(I)*(tau-cross(omega,I*omega));
end

function R=rotation(~,Theta)

    Rx=[1,0          ,0           ;
        0,cos(Theta(1)),-sin(Theta(1));
        0,sin(Theta(1)),cos(Theta(1))];

    Ry=[cos(Theta(2)),0,sin(Theta(2));
        0           ,1,0           ;
        -sin(Theta(2)),0,cos(Theta(2))];
    Rz=[cos(Theta(3)),-sin(Theta(3)),0;
        sin(Theta(3)),cos(Theta(3)),0;
        0,0,1];
    R=Rz*Ry*Rx;
end

function omega=thetadot2omega(~,thetadot,theta)
         omega=[1, 0              , -sin(theta(2));
                0, cos(theta(2)) , cos(theta(2))*sin(theta(1));
                0, -sin(theta(2)),cos(theta(2))*cos(theta(1))]*thetadot;
end

function thetadot=omega2thetadot(~,omega,theta)
    thetadot  =  inv([1 , 0              , -sin(theta(2)) ;
                      0 , cos(theta(1)) , cos(theta(2))*sin(theta(1));
                      0 , -sin(theta(1)), cos(theta(2))*cos(theta(1))  ] )*omega ;

end
```

Figure 4: All other tool functions

In the end, the results are shown below.

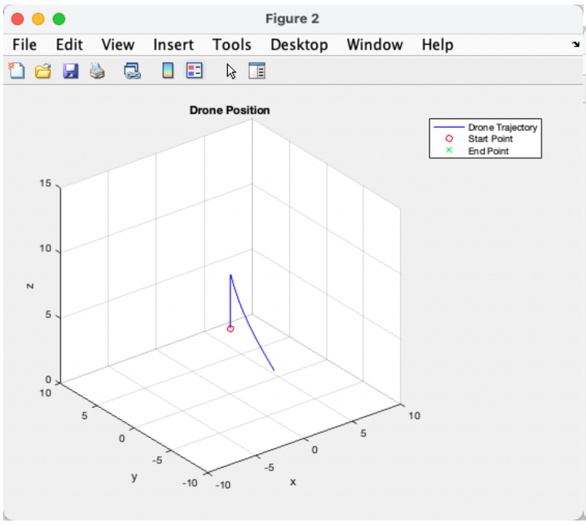


Figure 5

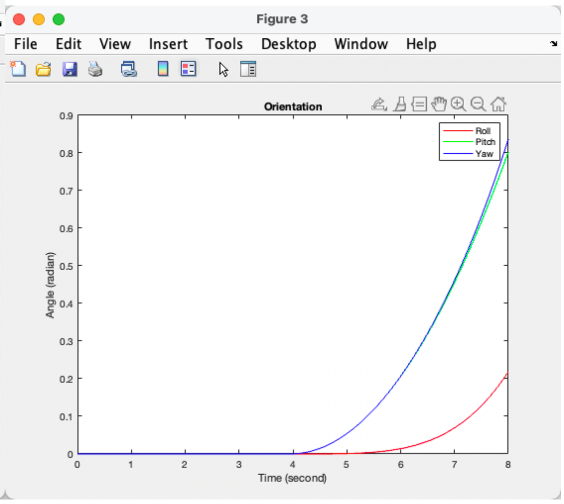


figure 6

From figure 5 and figure 6, we can find that, before the 4th second, the orientation of the drone didn't change as the inputs of it hadn't changed. After we removed the power of the third propellers, the drone started to rotate and eventually fell.

Q2:

a) To find the non-linear dynamics of the quadcopter in state-space representation. We could firstly define multiple symbols in MATLAB and then using MATLAB to perform the partial derivatives, instead of performing by hand (as is shown in figure 7).

```
function sys=getLTIPara(obj)
    syms Theta [3 1]
    syms Omega [3 1]
    syms X [4 1]
    syms Xdot [4 1]
    syms u [4 1]
    syms Pos [3 1]
    syms Vel [3 1]
    % Define some constants
    Rx=[1,0          ,0           ;
        0,cos(Theta1),-sin(Theta1);
        0,sin(Theta1),cos(Theta1)];

    Ry=[cos(Theta2),0,sin(Theta2);
        0          ,1,0          ;
        -sin(Theta2),0,cos(Theta2)];
    Rz=[cos(Theta3),-sin(Theta3),0;
        sin(Theta3),cos(Theta3),0;
        0,0,1];
    R=Rz*Ry*Rx;
    % Define the LTI system
    X1 = Pos ;
    X2 = Vel;
    X3 = Theta;
    X4 = Omega;

    Xdot1=Vel;
    Xdot2=[0;0;-obj.g]+1/obj.m*R*obj.k*[0;0;u1+u2+u3+u4]-obj.kd/obj.m*Vel;
%     Xdot2 = [0;0;-obj.g] + 1/obj.m  *(Rz*Ry*Rx)* obj.k*[0;0;(u1+u2+u3+u4)] - 1/obj.m*obj.kd* Vel;
    Xdot3=[1,0,-sin(Theta2); ...
                0,cos(Theta1),cos(Theta2)*sin(Theta1); ...
                0, -sin(Theta1),cos(Theta2)*cos(Theta1)]\Omega;
%     Xdot4=inv(obj.I)*[obj.L*obj.k,0,-obj.L*obj.k,0; ...
    Xdot4 = [(obj.L * obj.k * (u1 - u3))/(obj.I(1,1));
            (obj.L * obj.k * (u2 - u4))/(obj.I(2,2));
            (obj.b * (u1 - u2 + u3 - u4))/(obj.I(3,3))] ...
            - [(obj.I(2,2) - obj.I(3,3))/obj.I(1,1) * Omega2*Omega3;
            (obj.I(3,3) - obj.I(1,1))/obj.I(2,2) * Omega1*Omega3;
            (obj.I(1,1)-obj.I(2,2))/obj.I(3,3) * Omega1*Omega2];


    A=jacobian([Xdot1;Xdot2;Xdot3;Xdot4],[Pos;Vel;Theta;Omega]);
    B=jacobian([Xdot1;Xdot2;Xdot3;Xdot4],[u1;u2;u3;u4]);
```

Figure 7:

Following the state definition in [1], where x1 is the position, x_2 is the linear velocity, x_3 is the angular and x_4 is the angular velocity vector. The derivatives of them are shown in figure 8.

$$\dot{x_1} = x_2$$

$$\dot{x_2} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} R T_B + \frac{1}{m} F_D$$

$$\dot{x_3} = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix}^{-1} x_4$$

$$\dot{x_4} = \begin{bmatrix} \tau_\phi {I_{xx}}^{-1} \\ \tau_\theta {I_{yy}}^{-1} \\ \tau_\psi {I_{zz}}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}} \omega_y \omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}} \omega_x \omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}} \omega_x \omega_y \end{bmatrix}$$

Figure 8: State

After we get all state and statedot, we can then use MATLAB built-in function Jacobian() to get parameters of LTI system. Basis $[u1: u2: u3: u4]$ and $[Postion; Vel; Theta; Omega]$ are used as they are the input and state of the state form LTI system.

Now we have already digitalized the system. Then, substitute the initial state into the system and then use MATLAB function $c2d()$ to digitalize the system.

```
B=jacobian([Xdot1;Xdot2;Xdot3;Xdot4],[u1;u2;u3;u4]);
A=subs(A,u,[-0.49;-0.49;-0.49;-0.49]);
A=subs(A,Omega,[0;0;0]);
A=subs(A,Theta,[0;0;0]);
B=subs(B,Theta,[0;0;0]);
B=subs(B,Omega,[0;0;0]);



A=double(A);
B=double(B);
sys=c2d(ss(A,B,eye(12),zeros(12,4)),obj.time_interval,'zoh');
```

Figure 9: digitalize the system

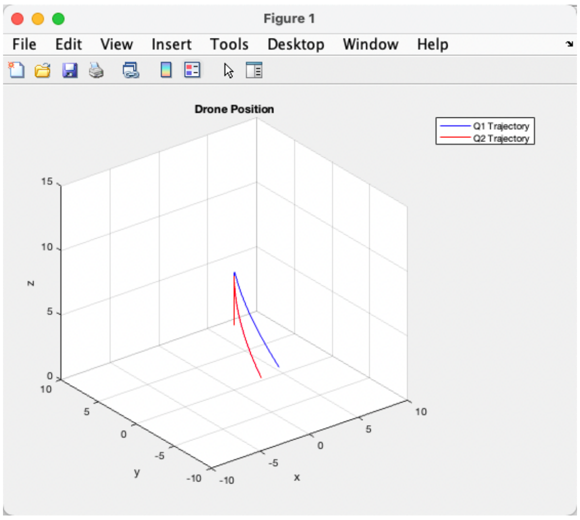
b) The core idea of state-space representation is using the formula

$$x[k+1] = A * x[k] + B * u[K]$$

```
%% kinematics
% update
obj.sys=getLTIPara(obj,input);
obj.A=obj.sys.A;
obj.B=obj.sys.B;
obj.state_current=obj.A*(obj.state_current-zeros(12,1))...
    +obj.B*(input-[num_gamma;num_gamma;num_gamma;num_gamma]);

% data process
obj.pos=obj.state_current(1:3);
obj.theta=obj.state_current(7:9);
obj.R=rotation(obj,obj.theta);
obj.posRecord = [obj.posRecord , obj.pos];
obj.orienRecord = [obj.orienRecord , obj.theta];
```

Figure 10: kinematics of state-space representation.

In figure 10, using the parameter A and B we can calculate the current state by apply the state of the last iteration. Note that we should update the parameter A and B in each iteration with up-to-date data.



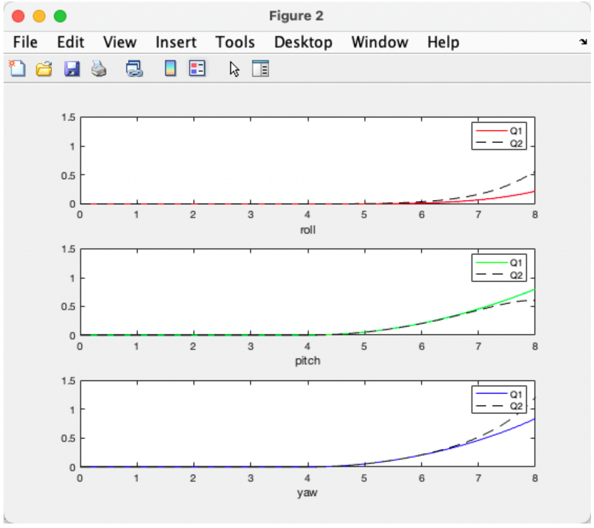


Figure 11

Figure 12

The comparison of results from Q1 and Q2 are shown in figure 11 and 12. The difference between trajectories are noticeable. And thus, roll, pitch and yaw all have difference non-negligible numeral difference. This error may be generated when we linearize the model. As we used Tylor series to expand the origin formula and only kept the first two terms for calculation. In this case, the abandoned terms cause noticeable error.

Q3

a) Two control a quadcopter basing on the model in [1], we can apply two controllers to the model. The first one is used to control the altitude of the aircraft. For a quadcopter with four propellers, the thrust on z axis should compensate the gravity and friction. Fortunately, we can increase and decrease the power supply of the propellers to control the quadcopter. Therefore, the first PID controller has been created in figure 13.

```
%% The first controller
Kp1=0.06;
Ki1=0.01;
Kd1=1;

ep1=pos_ref(3)-obj.pos(3);
ei1=obj.ei1_last+ep1*obj.time_interval;
ed1=(ep1-obj.ep1_last)/obj.time_interval;

thrust_additional= Kp1*ep1 + Ki1*ei1 + Kd1*ed1;

obj.ep1_last=ep1;
obj.ei1_last=obj.ei1_last+ep1*obj.time_interval;

Thrust_next=obj.m*obj.g/obj.k/cos(obj.theta(1))/...
    cos(obj.theta(2))+thrust_additional;
```

Figure 13: the first controller

The input of the controller is distance vector between target and quadcopter and the output is the additional thrust that will be add to total thrust in the end before kinematics step.

The other controller is responded to control the angular of the quadcopter to make the aircraft move in x-y plane. Code is shown below in figure 14.

```
%% The second controller
pos_vector=pos_ref-obj.pos;

if obj.a == zeros(3,1)
   a_Vector = [0;0;1];
else
    a_Vector =[obj.a(1);obj.a(2);1];
end
rAngle = vrrotvec( a_Vector,pos_vector);


Kp2=80;
Ki2=6;
Kd2=10;

ep2=obj.theta;
ei2=obj.ei2_last+obj.time_interval*ep2;
ed2=(ep2-obj.ep2_last)/obj.time_interval;

e=obj.I*(Kp2*ep2+Ki2*ei2+Kd2*ed2)-transpose(rAngle(1:3));

obj.ei2_last=ei2;
obj.ep2_last=ep2;
```

Figure 14: The second controller

The input of the controller is the angular of quadcopter and the output is the error which is proportional to the differences between desired trajectory and observed trajectory and its derivatives. However, we can only access to the angular velocity as we just own a gyro. Therefore, we have to use to use the relationship below, which assume the gyro can precisely record angular velocity.

$$\gamma_1 = \frac{mg}{4k\cos\theta\cos\phi} - \frac{2be_\phi I_{xx} + e_\psi I_{zz}kL}{4bkL}$$

$$\gamma_2 = \frac{mg}{4k\cos\theta\cos\phi} + \frac{e_\psi I_{zz}}{4b} - \frac{e_\theta I_{yy}}{2kL}$$

$$\gamma_3 = \frac{mg}{4k\cos\theta\cos\phi} - \frac{-2be_\phi I_{xx} + e_\psi I_{zz}kL}{4bkL}$$

$$\gamma_4 = \frac{mg}{4k\cos\theta\cos\phi} + \frac{e_\psi I_{zz}}{4b} + \frac{e_\theta I_{yy}}{2kL}$$

Then, as the output of the PID is proportional to Tau_B, which means we can directly add our desired gesture information into the system with the Tau_B. Therefore, the rotation between the desired gesture and current gesture can represented by the rotation between current acceleration and desired gesture. Desired gesture can be calculated by differences to target and current acceleration can be found as is shown in Q1. The corresponding code is shown below.

```
%% The second controller
pos_vector=pos_ref-obj.pos;

if obj.a == zeros(3,1)
   a_Vector = [0;0;1];
else
    a_Vector =[obj.a(1);obj.a(2);1];
end
if obj.time==122
    rr=1;
end
rAngle = vrrotvec( a_Vector,pos_vector);


Kp2=80;
Ki2=6;
Kd2=10;

ep2=obj.theta;
ei2=obj.ei2_last+obj.time_interval*ep2;
ed2=(ep2-obj.ep2_last)/obj.time_interval;

e=obj.I*(Kp2*ep2+Ki2*ei2+Kd2*ed2)-transpose(rAngle(1:3));

obj.ei2_last=ei2;
obj.ep2_last=ep2;


inputs=error2inputs(obj,e,Thrust_next);
```

Figure 15: The second controller

Beside controller, the pos_ref is determined by logic control. Several tasks' functions are created to control the main thread of the code. I defined some flags to indicate the progress of the code. And the five tasks have been combined into three.

```
%% firstly, get pos_ref basing on current step
if obj.flags==[0;0;0;0]
    pos_ref=taskABC(obj,pos);
elseif obj.flags==[1;0;0;0]
    pos_ref=taskABC(obj,pos);
elseif obj.flags==[1;1;0;0]
    pos_ref=taskD(obj);
elseif obj.flags==[1;1;1;0]
    pos_ref=taskE(obj);
else
    aaa=0;
    pos_ref=[5;5;0];
end
```

Figure 16

```
function pos_ref=taskABC(obj,pos)
    pos_ref=[5;5;5];
    % requirements check
    if all(pos>=5-obj.tolerance) && all(pos<=5+obj.tolerance)
        if obj.timmer_in_use==0 % means the first time we enter this if
        % initialize a timmer
        obj.timmer_in_use=1;
        obj.current_step=0;
        %set flag
        obj.flags(1)=1;
        end
        obj.current_step=obj.current_step+1;
        if obj.current_step>=250
            obj.flags(2)=1;
            obj.timmer_in_use=0;

        end

    end

end
```

Figure 17

Figure 16 shows how the program know the progress of itself according to the flags we set. And Figure 17 ,18 and 19 are the code for each task. It's safe for them to use same timer because we are using single thread and single progress and the tasks are performed sequentially.

```
function pos_ref=taskD(obj)
if obj.timmer_in_use==0
    % initialize a timmer
        obj.timmer_in_use=1;
        obj.current_step=0;
end
%suppose we use 80 seconds to finish the circular, thus 1000steps
theta_circles=obj.current_step*pi/30;
pos_ref=[2.5+2.5*cos(theta_circles);5+2.5*sin(theta_circles);5];
diff=sqrt((obj.pos(1)-pos_ref(1))^2+(obj.pos(2)-pos_ref ...
    (2))^2);

if all(obj.pos>=pos_ref-obj.tolerance) && all(obj.pos<=pos_ref+obj.tolerance)
    obj.current_step=obj.current_step+1;
end
if obj.current_step==60
    obj.flags(3)=1;
    obj.timmer_in_use=0;
    obj.current_step=0;
    pos_ref=[5;5;5];
end
end
```

Figure 18

```
function pos_ref=taskE(obj)
if obj.timmer_in_use==0
    % initialize a timmer
        obj.timmer_in_use=1;
        obj.current_step=0;
end
pos_ref=[5;5;5*(30-obj.current_step)/30];


if all(obj.pos>=pos_ref-obj.tolerance) ...
        && all(obj.pos<=pos_ref+obj.tolerance)...
        && obj.time-obj.last_time>=2
       obj.last_time=obj.time;
       obj.current_step=obj.current_step+1;

end
if obj.current_step==30
    obj.flags(4)=1;
    obj.timmer_in_use=0;
    obj.current_step=0;
    pos_ref=[5;5;0];

end
end
```

Figure 19

For both task D and task E, we divide the whole desired trajectory into several checkpoints.

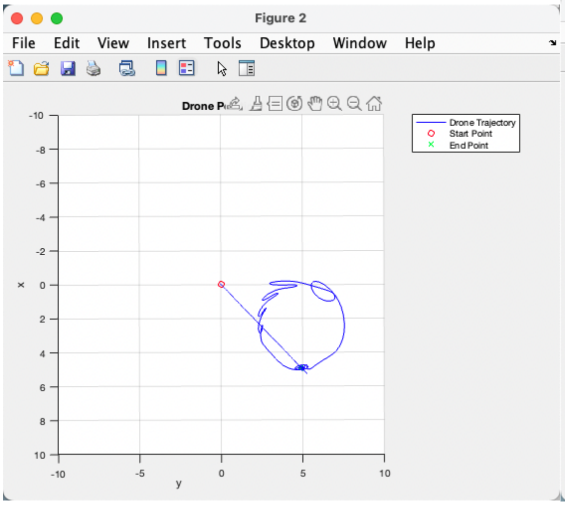


Figure 20

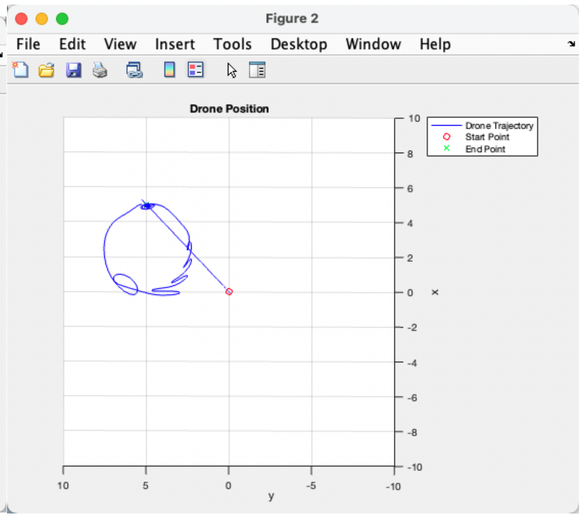


Figure 21

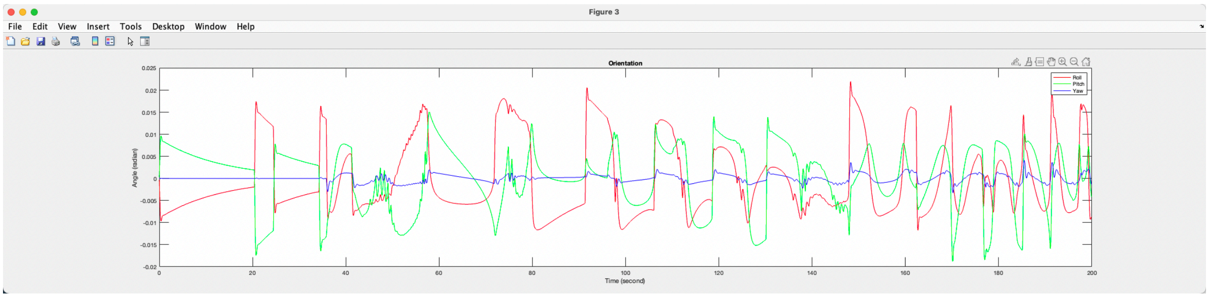


Figure 22

The results are shown above. The basic tasks are completed successfully. However, it is noticeable that the circle is not smooth and has many vibrations in it. One of the possible reasons is that the parameters of the PID controller are not tuned to an optimization, if I had enough time, I could use automatic PID tuning method to help find the best PID parameters. Another possible reason could be that the error calculation model for the pid is wrong, I refer exactly to the method in [1], although I still have doubts about it. I think it would be more appropriate to use the pos obtained by theta calculation as the error. But for time reasons, I will have to do it later when I have the chance.

b)

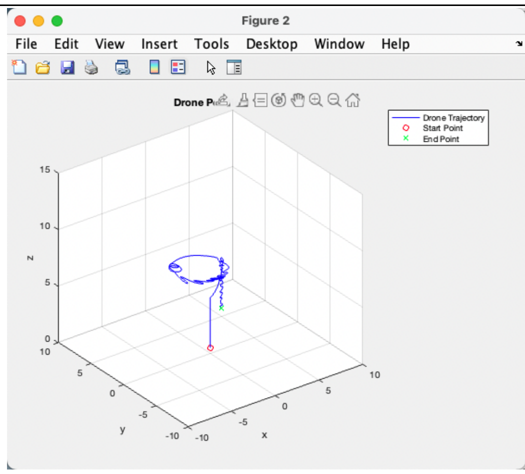
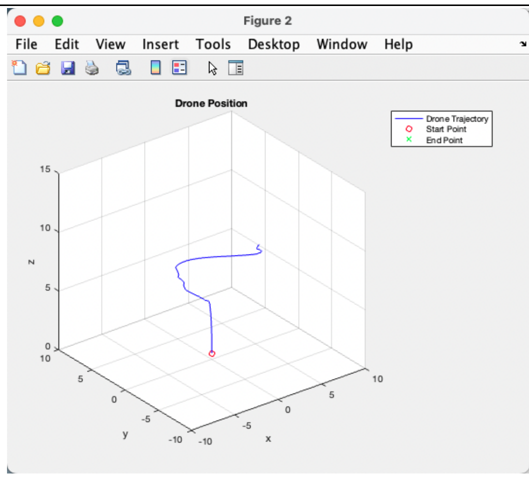
Since we only has gyro that can only observe angular velocity, therefore, we just simply need to add a gaussian noise to $obj.theta$ dot before each iteration. The code is shown below and figures with different means and variances are shown in table 1.

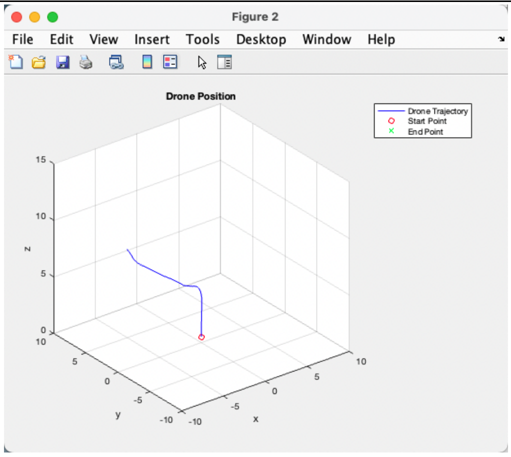
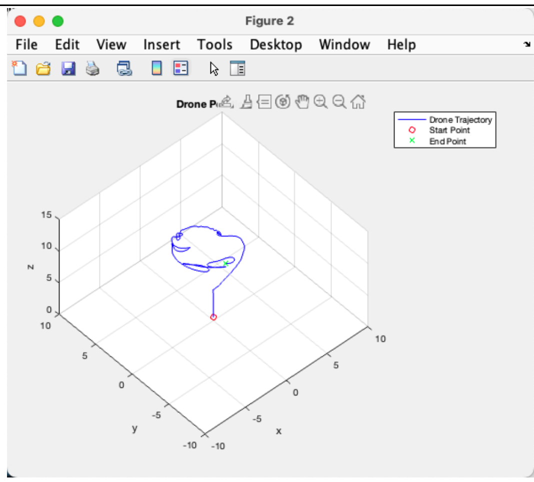
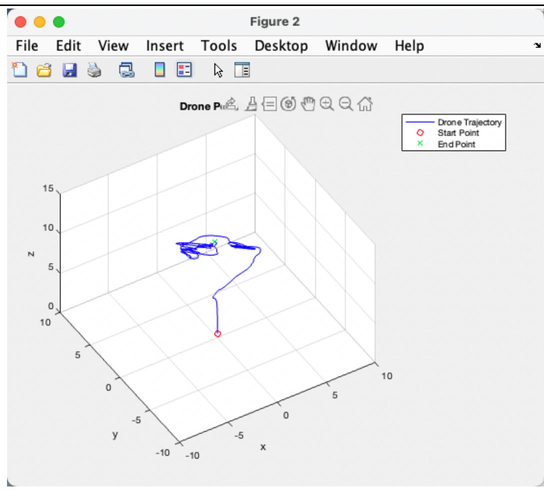
```
function noise=get_noice(~,mu,sigma)
    noise = mu + (sqrt(sigma) * randn);
    noise = noise * ones(3,1);

end
```

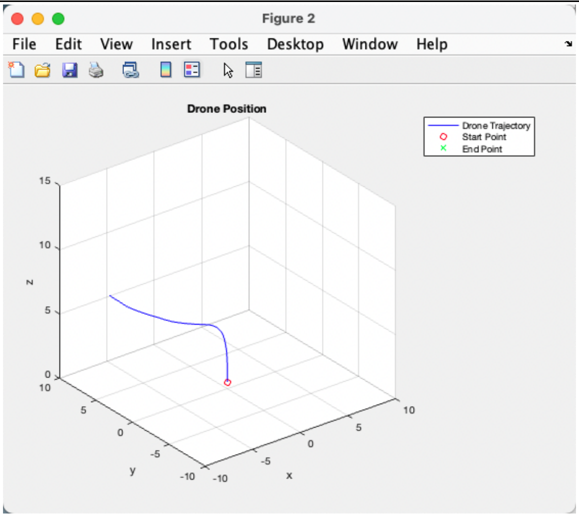
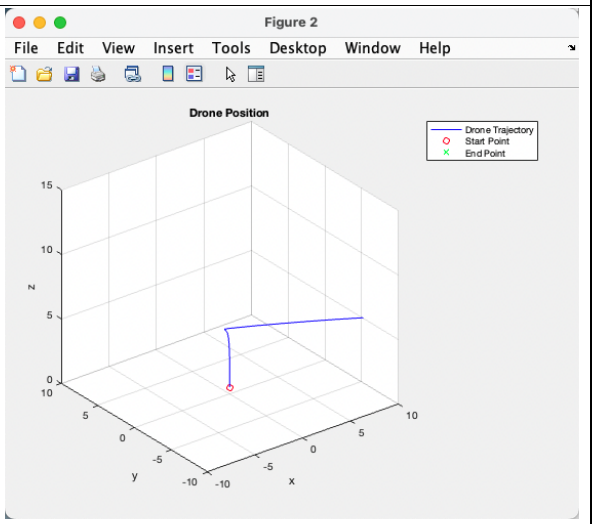
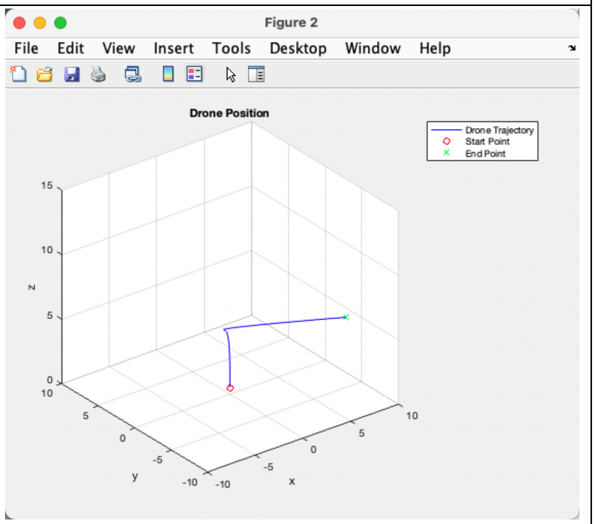
Figure: 23

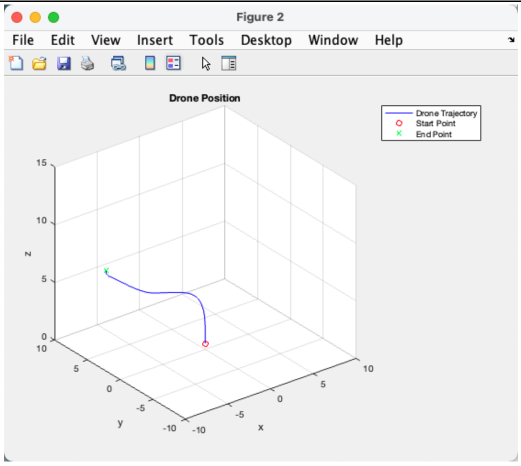
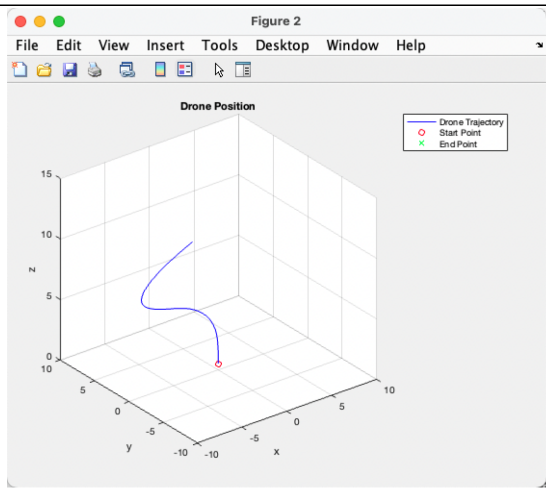
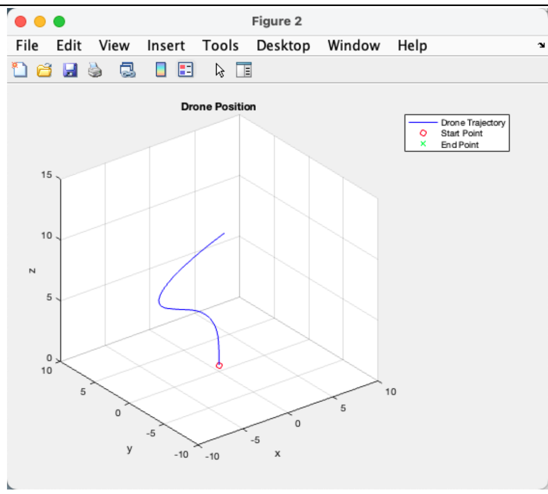
```
%% get input
% add noise
noise=get_noice(obj,0.01,0.001);
obj.thetadot=obj.thetadot+noise;


num_gamma=obj.m*obj.g/4/obj.k;
inputs=controller(obj,obj.pos);

kinematics(obj,inputs)
```

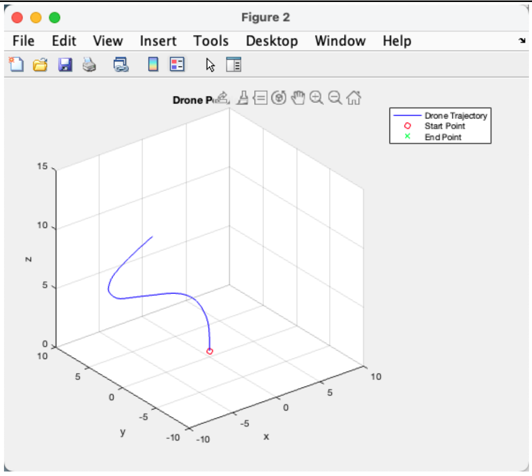
Figure 24

Table 1 : Multiple combinations of mu and sigma			
	Mu	Sigma	Figure
1	0.001	0.001	
2	0.001	0.1	

3	0.001	1	
4	0.01	0.001	
5	0.01	0.1	

6	0.01	1	
7	0.1	0.001	
8	0.1	0.1	

9	0.1	1	
10	1	0.001	
11	1	0.1	

12	1	1	

Basing on the figures in table 1, we can find that the model can be easily affect by noise. It is also possible that the noise of the setting is not within a reasonable range. As for the answer to this question, more in-depth research is needed

Reference

[1] A. Gibiansky, Quadcopter Dynamics, Simulation, and Control. An approximated dynamic model of a quadcopter and some potential control strategies are described in the following document.