COMP0128 Robotic Control Theory and Systems
Coursework 2

Hou Nam Chiang (15055142) hou.chiang.20@ucl.ac.uk

Tim Andersson (20167996) ucabta2@ucl.ac.uk

Sarosh Habib (20096908) ucabhh0@ucl.ac.uk

7^{th} December, 2020

Question 1

(a) The advantages of using the strategy defined in [1] are as follows:

- Not using inputs directly makes it simpler for the system to be linearised around a desired equilibrium point. In this case, the paper linearises the system around $T = mg$. Linearising around any point in either thrust T or torque τ defines the other. Any desired T or τ is simply a function of the inputs γ_i, making it easy to solve for desired inputs based on desired states.
- Using this linearisation, inputs can be changed based solely on derivatives (i.e. controlling thrust based on velocity to stabilise around $T = mg$), meaning no LTI system needs to be implemented to understand the state of the motors controlling the thrust and torque, and no prior knowledge of the inner workings of each of the rotor motors is required.
- τ and T are both physical forces that relate directly to the dynamics of the system, simplifying the state space representation when used as inputs.
- In dynamic systems such as this where stability is a crucial aspect, minimising the input to be solely the aspects of the system that define stability (i.e. τ and T) is the most efficient and simple way to control it, using some sort of controller such as a PID, PI, P or LQR controller.

(b) Two examples of similar strategies are as follows:

- a DDR travelling with a fixed distance to a wall.
- An inverted pendulum being controlled with an LQR.

DDR example

The DDR example in which the DDR travels parallel to a wall at a fixed distance uses two variables in the linearised control system:

- The distance to the wall, d.

- The deviation of the angle φ from the equilibrium point, where $\varphi = 0$. φ is equal to zero when the orientation of the DDR is parallel to the of the wall.

Whilst this system is much simpler than the quadcopter control and the inverted pendulum. The linearisation around the equilibrium point is very similar, and in practice a PID controller could very easily be fit to system as is similarly done in [1].

Inverted pendulum LQR example

Much like the system for the quadcopter in [1], the inverted pendulum example in lecture 6 is a non-linear system. It uses only two variables for control and approximately linearising the system:

- The full state, derived from p, which is the position of the end of the body of the pendulum with length l. Using p, $\dot{p}$ can be derived, as can θ and $\dot{\theta}$.
- The force F perpendicular to the body of the pendulum with length l, which is the systems defined input.

This method of controlling a system uses the absolute bare minimum information needed to control the system. The state for this system could be made up of copious amounts of variables, as could the input, but similarly to the proposed control system in [1] uses the bare minimum information required to achieve a full- rank reachability matrix (dependent on the input variables). The method in [1] could also go further, similar to the LQR in lecture 6a, and reduce the state down to the minimal variables required, which would be x_1 and x_3, the x, y, z position of the quadcopter and the ϕ, θ, ψ (roll pitch and yaw) angles of the quadcopter respectively.

Question 2

(a) In the code given for this question, the approximation being done is that the dynamics and the system itself is discrete, whereas the code is updating the state as if the system was continuous. This, in a way, can work in practice but over time the error in the estimated state accumulates much greater than it would if the system were to be rightly implemented in a discrete manner.

(b) Working with this continuous approximation, the best way to minimise the approximation error is to have the smallest sampling time d_t possible. This essentially makes the system as close to continuous as possible, given:

$$d_t \overset{lim}{\rightarrow} 0$$

(c) Updating the state without using this approximation involves taking the continuous state space equation for an LTI system, shown below:

$$\dot{x} = Ax + Bu$$

Where A and B are made up of constants related to dynamics and aspects of the state x. We then discretise the system using the same parameters, using the equations below:

$$x[k+1] = A_d x[k] + B_d u[k]$$

Where:

$$A_d = e^{A\Delta t}, B_d = A^{-1}\left(e^{A\Delta t} - I\right)B$$

Giving:

$$x[k+1] = e^{A\Delta t}x[k] + A^{-1}\left(e^{A\Delta t} - I\right)Bu[k]$$

Where the state is updated at every discrete step.

Question 3

(a) Consider the state space quadcopter dynamics at the beginning of page 8 of [1]:

$$\dot{x_1} = x_2 \tag{1}$$

$$\dot{x_2} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m}RT_B + \frac{1}{m}F_D \tag{2}$$

$$\dot{x_3} = \mathbf{R}_{zyx}^{-1}x_4 \tag{3}$$

$$\dot{x_4} = \begin{bmatrix} \tau_\phi I_{xx}^{-1} \\ \tau_\phi I_{yy}^{-1}\tau_\phi I_{zz}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{xx}-I_{zz}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y \end{bmatrix} \tag{4}$$

Where $\mathbf{R}_{zyx}$ is an Euler angle zyx rotation matrix. To do these equations, the following physical constants are used[3][4]:

Constant	Value
I_{xx}	0.0221
I_{yy}	0.0259
I_{zz}	0.0354
m	1.305kg
C_D where $\phi, \theta, \varphi = 0$	0.0464
C_D where $\phi, \theta, \varphi \approx \pm 15°$	0.0613
A where $\phi, \theta, \varphi = 0$	$0.021m^2$
A where $\phi, \theta, \varphi \approx \pm 15°$	$0.029m^2$
L	$\approx 0.3678m$

m - Mass of the quadcopter
L - Distance from rotor to centre of quadcopter
C_D - Drag coefficient
I_{xx} - Moment of inertia in x axis
I_{yy} - Moment of inertia in y axis
I_{zz} - Moment of inertia in z axis
A - Area swept out by quadcopter

(b) Given:

$$\bar{\dot{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \frac{-k_d \dot{x}}{m} \\ \frac{-k_d \dot{y}}{m} \\ \frac{-k_d \dot{z}}{m} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \tau_\phi I_{xx}^{-1} \\ \tau_\theta I_{yy}^{-1} \\ \tau_\psi I_{zz}^{-1} \end{bmatrix} = Ax + Bu, \bar{x} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \phi \\ \theta \\ \psi \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{5}$$

Where:

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-k_d}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-k_d}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{-k_d}{m} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ I_{xx}^{-1} & 0 & 0 \\ 0 & I_{yy}^{-1} & 0 \\ 0 & 0 & I_{zz}^{-1} \end{bmatrix}$$

And:

$$u = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{6}$$

We can linearise this system around the equilibrium point $T = mg$, defined in state space as:

$$\dot{x}, \dot{y}, \dot{z}, \phi, \theta, \varphi = 0 \tag{7}$$

and:

$$\bar{\delta_x} = \begin{bmatrix} x \\ y \\ z \\ -\dot{x} \\ -\dot{y} \\ -\dot{z} \\ -\phi \\ -\theta \\ -\psi \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi}, \end{bmatrix}, \bar{\delta_u} = \begin{bmatrix} \frac{\tau_\phi + \tau_\theta + \tau_\varphi}{3} - \tau_\phi \\ \frac{\tau_\phi + \tau_\theta + \tau_\varphi}{3} - \tau_\theta \\ \frac{\tau_\phi + \tau_\theta + \tau_\varphi}{3} - \tau_\varphi \end{bmatrix} \tag{8}$$

Ultimately, the control system is discretised in the form $\delta_x[k+1] = A_d\delta_x[k] + B_d\delta_u[k]$, where:

$$A_d = \begin{bmatrix}
1 & 1 & 1 & e^{\Delta t} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & e^{\Delta t} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & e^{\Delta t} & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & e^{\frac{-k_d \Delta t}{m}} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & e^{\frac{-k_d \Delta t}{m}} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & e^{\frac{-k_d \Delta t}{m}} & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & e^{\Delta t} & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & e^{\Delta t} & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & e^{\Delta t} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}$$

And:

$$B_d = \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\frac{k_d|m|^2 - m|k_d|^2}{I_{xx}k_d(|k_d|^2+|m|^2)} & \frac{k_d|m|^2 - m|k_d|^2}{I_{yy}k_d(|k_d|^2+|m|^2)} & \frac{k_d|m|^2 - m|k_d|^2}{I_{zz}k_d(|k_d|^2+|m|^2)} \\
\frac{k_d|m|^2 - m|k_d|^2}{I_{xx}k_d(|k_d|^2+|m|^2)} & \frac{k_d|m|^2 - m|k_d|^2}{I_{yy}k_d(|k_d|^2+|m|^2)} & \frac{k_d|m|^2 - m|k_d|^2}{I_{zz}k_d(|k_d|^2+|m|^2)} \\
\frac{k_d|m|^2 - m|k_d|^2}{I_{xx}k_d(|k_d|^2+|m|^2)} & \frac{k_d|m|^2 - m|k_d|^2}{I_{yy}k_d(|k_d|^2+|m|^2)} & \frac{k_d|m|^2 - m|k_d|^2}{I_{zz}k_d(|k_d|^2+|m|^2)} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
e^{\Delta t}I_{xx}^{-1} & I_{yy}^{-1} & I_{zz}^{-1} \\
I_{xx}^{-1} & e^{\Delta t}I_{yy}^{-1} & I_{zz}^{-1} \\
Y_{xx}^{-1} & I_{yy}^{-1} & e^{\Delta t}I_{zz}^{-1}
\end{bmatrix}$$

Question 4

(a) Given:

$$y = Cx + Du$$

However in this case the output is not directly dependent on the input but directly dependent on the state, so for this purpose we will derive y as:

$$y = Cx$$

The state and output are given below:

$$\bar{x} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \phi \\ \theta \\ \psi \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}, \bar{y} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

Where the output is simply the sensor measurements, which are the angular velocities. This is a severely unobservable system; shown by the rank-deficient C matrix below:

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

in B the system is observable

(b) In this part, we have more sensors to measure the altitude and the horizontal position, so the output y will become:

$$\bar{y} = \begin{bmatrix} x \\ y \\ z \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

so in this case, the matrix C will become a 6×12 matrix:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Question 5

(a) For the non-linear quadcopter dynamics, we modeled it in the same way as described in A. Gibiansky's document using the same set of state space equations for modelling the evolution of the states. To code this up, we initialize the positions, linear velocities, roll, pitch and yaw angles as defined by the Z-Y-X extrinsic Euler representation, and the angular velocities as four 3 by 1 vectors (each element of the vectors corresponds to a x, y or z axis), namely, *obj.pos*, *obj.xdot*, *obj.theta* and *obj.omega* respectively as represented in the MATLAB script. All these four state vectors are initialized with values of zeros initially in the Drone class constructor as this corresponds to the initial pose of the drone. The physical parameters, such as the moments of inertia I, mass of the drone m, the frictional coefficient k_d and etc, are set as described in Question 3a.) of this report. To test the behaviour of our quadcopter in simulation, a button "Move!" is embedded into the figure, so that it can be pressed whenever a user wants to initiate a "move" command. This button is disabled when the drone is performing a certain action such as elevating to a height of 2.5m and moving around a circle, so as to prevent user from initiating the "move" commands multiple times when the drone is not ready yet to proceed to the next action.

To control the dynamics of the drone, we choose values for the thrust in the z-axis and the torques in the three axes respectively by using two different PID controllers. For the "move" commands' behaviour that are required for us to implement for the quadcopter, different strategies and controller gains would be used as described in the subsequent paragraphs. Regardless of the controlling strategies, the same set of equations are used to update the states of the drone given the corresponding controller inputs at every time step (i.e. whenever the update function is called on the drone) as implemented in the function *evolveStates*. Firstly, the acceleration vector (3×1) is calculated using this Newton's equation of motion: $\vec{a} = \vec{g} + \frac{1}{m} \times R \cdot \vec{T_B} - k_d \times \vec{obj.xdot}$, where $\vec{g} = \begin{bmatrix} 0 \\ 0 \\ -9.8 \end{bmatrix}$, $R = R_z(\phi)R_y(\theta)R_x(\psi)$ with each matrix being the rotation matrix for each Euler axis, $\vec{T_B} = \begin{bmatrix} 0 \\ 0 \\ T_{Bz} \end{bmatrix}$ with T_{B_z} being the input of one of the controllers. And then, this new acceleration could be used to update the linear velocities and positions of the quadcopter by using the following approximation of the non-linear dynamics:

```
1    obj.xdot = obj.xdot + obj.time_interval * a;
2    obj.pos = obj.pos + obj.time_interval * obj.xdot;
```

Similarly, the angular acceleration $\dot{\omega}$ could be calculated by using $\dot{\omega} = I^{-1}(\tau - \omega \times (I\omega))$., where I is the inertia matrix $\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$, τ is the input torques given by one of the controller as inputs and ω being the angular velocities (one of the states) of the quadcopter. The derivatives of the roll, pitch and yaw angles could also be calculated by using $\dot{\theta} = \mathbf{R}_{zyx}^{-1}\omega$. And then, with $\dot{\omega}$ and $\dot{\theta}$, the states relating to the angular velocities and the roll, pitch and yaw angles of the quadcopter could be updated with the following approximation of the non-linear dynamics:

```
obj.omega = obj.omega + obj.time_interval * omegadot;
obj.theta = obj.theta + obj.time_interval * thetadot;
```

With these states updates, the new position and orientation of the quadcopter could be shown in the simulation in each time step. As mentioned previously, all the control strategies would be utilizing these update rules for the evolution of states.

For the first "move" command as implemented in the function *altitudeController*, the thrust in the z-axis of the body frame of the quadcopter T_{Bz} is controlled using a PID controller. The error between the current quadcopter z-position and the desired z-position which is 2.5m above the ground is used to calculate the controlled thrust with a corresponding proportional gain. To avoid the quadcopter from elevating too quickly, a derivative gain is also added to control the speed in its z-direction to be as close to zero as possible. Thus, the corresponding derivative error is calculated by the error between current drone speed in z-axis and zero. With this PD control, after a few experiments, it was observed that the quadcopter can approach to the desired z-position but with a certain offset error. Therefore, an integral part is also added to the controller with the corresponding integral gain to eliminate the accumulated error over time, with this accumulated error calculated by summing over all the previous errors since the drone starts elevating multiplied by the time interval of each step. The equation for this PID controller is $T_{Bz} = K_p e(t) + K_d \dot{e(t)} + K_i \int_0^t e(t)\,dt$ with $e(t)$ being the discrepancy of the drone's current position and its desired position in the z-axis. With such a PID controller, our quadcopter could thus stabilize itself at a height of 2.5m above the ground.

For the second "move" command as implemented in the function *circleController*, to get the quadcopter to move in a circle of radius 2.5, a equation of circle is devised. This circle is discretized into a series of points (e.g. 100 points in our controller) so that our quadcopter could move to each point on this circle sequentially which results in the final circular motion (Figure 2). In each time step, an angle α to the next point is calculated and then the next point (x- and y- coordinates) to travel to is calculated with this angle by the equation $[(\cos(\alpha) * 2.5) - 2.5; (\sin(\alpha) * 2.5)]$. Then the distance in the x- and y-directions between the next point and current points are calculated. These two distances are

After the quadcopter completed a round of circular motion returning to the starting point, it is then made to return to the ground by using the same controller as in the first "move" command but with a desired z-position of 0m instead of 2.5m from the ground. In order to make the simulation more realistic, we set the position and speed of the drone in the z-axis to be zero whenever its position and speed in the z-axis is negative, due to the fact that there is the ground which stops the drone from penetrating any further into the ground. Besides, to stop the quadcopter from moving any further after completion of the entire command and to maintain it on the ground, the thrust is set to zero when it is very close to the ground (i.e. 0.05m from the ground) so as to simulate power off of the drone.

To complete a full circle, the circle is discretised into 100 points. The circle is treated as having the centre being the origin, and sub-sequentially transformed by -2.5 in the x axis. The calculation of each discrete point is shown in the following code:

```
point = [(cos(obj.first_ang) * 2.5) - 2.5; (sin(obj.first_ang) *
    2.5)];
```

```
2    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%
3    % code in between not relevant to this paragraph
4    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~%
5
6  if (overall_distance < 0.25)
7      if (obj.count2 < 100)
8          obj.first_ang = obj.first_ang + 2*pi / 100;
9          obj.count2 = obj.count2 + 1;
10       end
11 end
```

Where the overall distance is the magnitude of the distance between the drone and the next position on the circle. The first angle is initialised to be $\frac{2\pi}{100}$ and then is subsequentially incremented by that same value each time to create a smooth velocity around the circle. The full set of 100 discrete points around the perimeter of the circle travelled by the drone can be seen below:

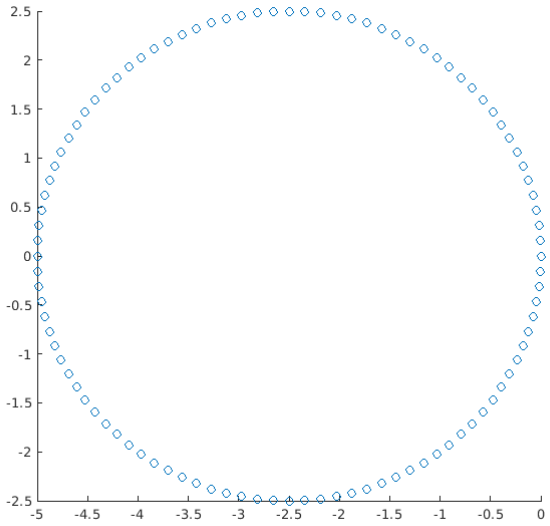

Figure 1: The target points the drone sequentially follows to complete a full circle, starting at (0,0) and going counterclockwise.

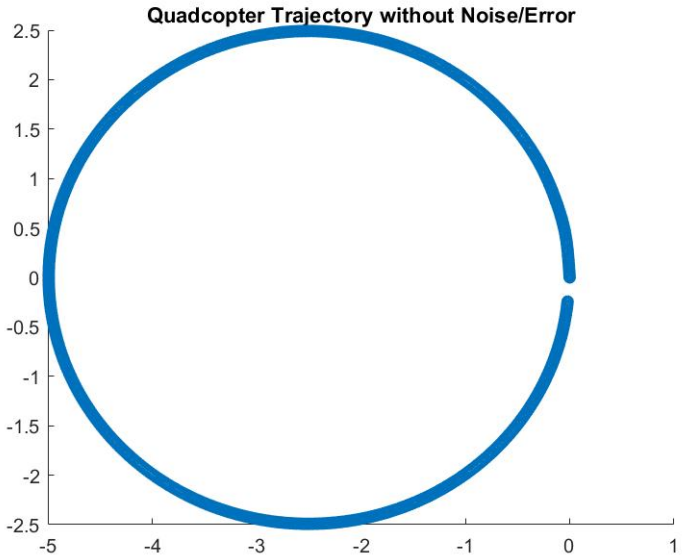

Figure 2: Motion of the Quadcopter Moving Around a Circle Without Sensor Noise/Error

(b) From the gyroscope, altitude and GPS, we can obtain the measurements of the derivatives of roll, pitch and yaw angles, the positions of the quadcopter in z, x and y-axis respectively. To use the same non-linear dynamics and controller as in Question 5a, we need to calculate all the four states, namely the position, linear velocities, roll, pitch and yaw angles and the angular velocities, from these sensor measurements. This sensing capability is implemented in the function *sense* inside the Drone class where the positions state of the drone are directly obtained from the newest values from the evolution of non-linear dynamics, and similarly, the derivatives of roll, pitch and yaw angles are obtained from the newest angular velocities values from the evolution of non-linear dynamics multiplied by the rotation matrix. This way of implementing the sensing capability of the drone is appropriate as the latest values obtained from the evolution of non-linear dynamics represent the same information obtained by the sensors without any noise or error.

This is handled by the function *process_sensor_values*. Firstly, the 3D position of the quadcopter could be obtained from the sensors GPS and altitude. Then, its linear velocities could be obtained by storing the previous 3D positions and calculating its difference with the current 3D positions divided by the time interval of each time step, i.e. $\frac{current\_position - previous\_position}{time\_interval}$, giving us the approximation of the derivatives of positions with respect to time which is equivalent to the linear velocities. For calculating the roll, pitch and yaw angles, we can calculate the integral of the derivatives of roll, pitch and yaw angles (which we obtained from the gyroscope) with respect to time, i.e. $\int_0^t \dot{\theta}\, dt$, where θ is the 3×1 vector of roll, pitch and yaw angles. To calculate this integral to obtain the roll, pitch and yaw angles, we approximated it by using the MATLAB built-in trapezoidal numerical integration *trapz* by feeding in the past derivatives of θ(as an array of the 3×1 vectors) and the time interval in between. With the calculated derivative of θ, the angular velocity vector ω can then be obtained simply by multiplying the derivatives of θ by the rotation matrix.

With these calculated states from the sensor values, they could then be fed into those controllers introduced in Question 5a.) and then the evolution of states and etc. From these, we obtained a very similar result as in Question 5a.) which is supposed to be the

case because there is no noise or error in the sensor values.

(c) With the way the sensing capability (function *sense*) is implemented in the drone, the Gaussian noise can simply be added to the sensors by directly adding this noise to the corresponding sense positions, altitude and derivative of θ. This noise is calculated by using the built-in *randn*, which gives the normally distributed random numbers, multiplied by the square root of the variation of distribution and then added by the mean of distribution, i.e. $noise = \sqrt{variation} \times randn + mean$. In our case, the variation is set to be 1 and mean to be 0 for simplicity sake.

To find how much noise our controller can handle, the magnitude of the noise is tuned by multiplying the Gaussian noise by a defined magnitude constant. We tried different values of the magnitude constant, such as 0.005, 0.01, 0.015 and 0.02. For noise magnitude constant of less than 0.005, the trajectory of the quadcopter does not have noticeably difference to the case without noise (as manifested as the smoothness of the trajectory in Figure 2). For the values of 0.01 and 0.015, the quadcopter starts to wiggles but still able to complete the desired behaviour (manifested as the roughness of the trajectory in Figure 3) despite with a bit of time delay. For a magnitude constant larger than 0.02 (and on a few occassions for 0.015), the wiggling of the quadcopter starts to affect its trajectory significantly to the point that it stuck at some point and thus cannot complete the desired behaviour while turning around a circle (Figure 4).

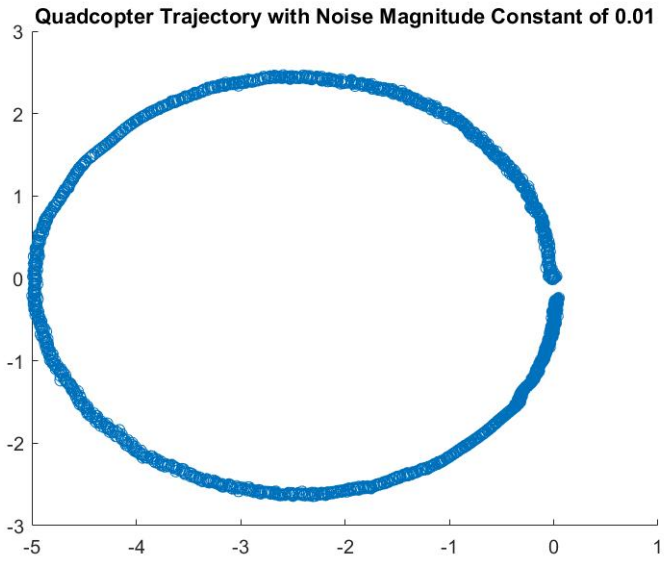

Figure 3: Motion of the Quadcopter Moving Around a Circle With a Sensor Gaussian Noise Magnitude of 0.01

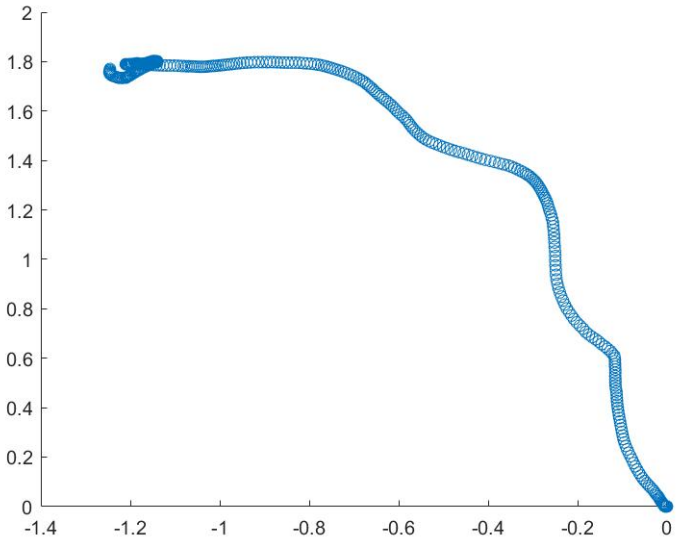

Figure 4: Quadcopter stuck at 1/4 of a Circle While Moving With a Sensor Gaussian Noise Magnitude of 0.02 or above

(d) For this part, we have to add a wind model to our simulation, which includes a mean field and a turbulence field. The mean field is expressed as follows[2]:

$$w_c = \begin{bmatrix} u_{20}\left(\frac{\log\left(-\frac{p_z}{0.15}\right)}{\log\left(\frac{20}{0.15}\right)}\right) \\ 0 \\ 0 \end{bmatrix}$$

where:

- u_{20} is the wind magnitude at 20 feet (6.096m) from the ground.
- $-p_z$ is the altitude of the drone.

The corresponding function is as shown:

```
function w_c = WindMeanField(u_20, p_z)
    w_c = [u_20*(log(-p_z/0.15)/log(20/0.15)); 0; 0];
end
```

And the turbulence field is expressed as follows: Turbulence scale lengths on u,v,w direction:

$$L_u = L_v = -\frac{p_z}{(0.177 + 0.000823h)^{1.2}}$$
$$L_w = -p_z$$

Turbulence Intensities on u,v,w direction:

$$\sigma_u = \sigma_v = \frac{\sigma_w}{(0.177 + 0.000823h)^{0.4}}$$
$$\sigma_w = 0.1u_{20}$$

With the above, the wind strength at time t+T:

$$w_c^{t+T} = \begin{bmatrix} w_u^{t+T} \\ w_v^{t+T} \\ w_w^{t+T} \end{bmatrix} = \begin{bmatrix} \left(1 - \frac{\mathrm{VT}}{L_u}\right) w_u^t + \varepsilon_u \sqrt{2\frac{\mathrm{VT}}{L_u}} \\ \left(1 - \frac{\mathrm{VT}}{L_v}\right) w_v^t + \varepsilon_v \sqrt{2\frac{\mathrm{VT}}{L_v}} \\ \left(1 - \frac{\mathrm{VT}}{L_w}\right) w_w^t + \varepsilon_w \sqrt{2\frac{\mathrm{VT}}{L_w}} \end{bmatrix}$$

where:

- T is the discretization period
- ε_u, ε_v and ε_w are white noise perturbations with variances σ_u^2, σ_v^2 and σ_w^2
- Wind strength at time t w_u^t, w_v^t, w_w^t
- h and $-p_z$ are drone altitude

The corresponding function is as shown:

```
function w_c_t = WindTurbulenceField(V, u_20, p_z, h, T, w_u_t,
    w_v_t, w_w_t)

    % Turbulence scale lengths on u,v,w direction:
    L_u = -p_z/(0.177 + 0.000823*h)^1.2;
    L_v = L_u;
    L_w = -p_z;

    % Turbulence Intensities on u,v,w direction (standard
       deviation, get variance by squaring
    % them):
    sigma_w = 0.1*u_20;
    sigma_u = sigma_w/(0.177 + 0.000823*h)^0.4;
    sigma_v = sigma_u;

    % White noise:
    epsilon_u = sigma_w*randn(1);
    epsilon_v = sigma_u*randn(1);
    epsilon_w = sigma_v*randn(1);

    % wind strength at time t+T:
    w_c_t = [WindTurbulenceField_Singular(V, T, L_u, w_u_t,
       epsilon_u);
             WindTurbulenceField_Singular(V, T, L_v, w_v_t,
                epsilon_v);
             WindTurbulenceField_Singular(V, T, L_w, w_w_t,
                epsilon_w)];
end


function w = WindTurbulenceField_Singular(V, T, L_n, w_n,
    epsilon_n)
     w = (1 - (V*T/L_n))*w_n + epsilon_n*sqrt(2*V*T/L_n);
end
```

Adding these two fields together gives a wind blowing at random direction which dominantly at x- direction. With this wind, we have explored that when $u_{20} > 0.000057$ 5, our drone will start to stuck at some point when it tried to perform the circle. Therefore, our controller can handle the wind strength up to $u_{20} = 0.000057$ (Figure 6). The function combining the two field is as shown, and the resultant wind strength w is added to the linear velocities in the drone state.

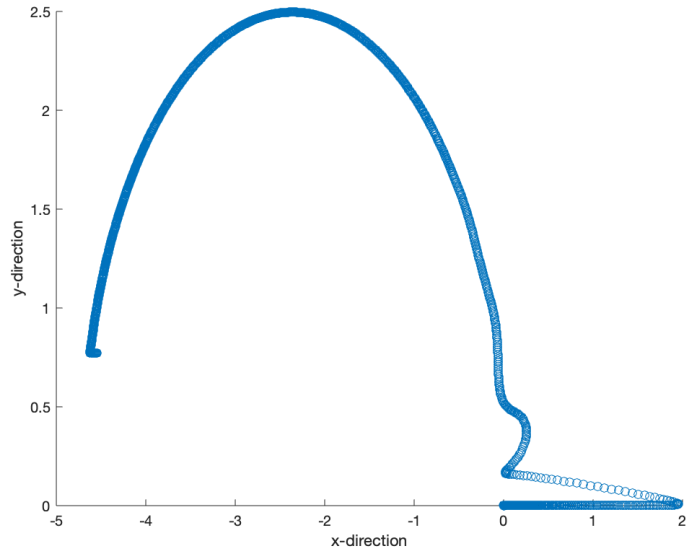

Figure 5: Motion under the wind model with $u_{20} = 0.000065$

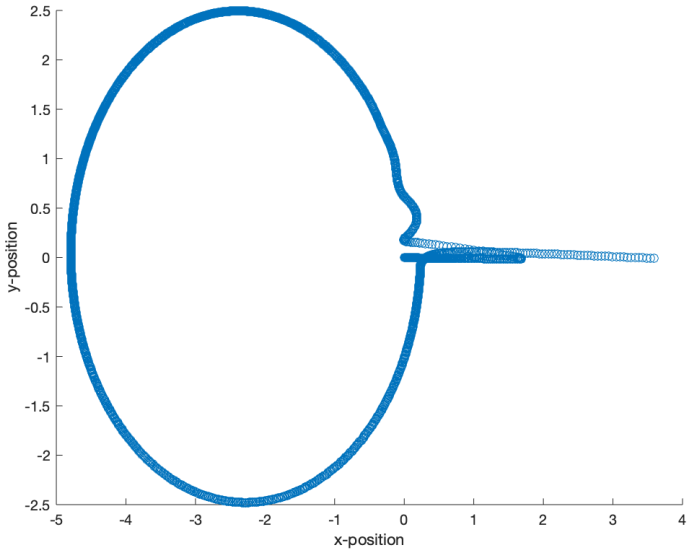

Figure 6: Motion under the wind model with $u_{20} = 0.000057$

```
function [w] = Wind(drone_velocities, drone_z_pos, w_old,
    time_interval)
    u_20     = 0.000057;
    p_z      = -drone_z_pos;
    T        = time_interval;
    V        = norm(drone_velocities);
```

```
6
7       w_c   = WindMeanField(u_20, p_z);
8       w_c_t = WindTurbulenceField(V, u_20, p_z, drone_z_pos, T,
            w_old(1), w_old(2), w_old(3));
9       w     = w_c + w_c_t;
10  end
```

(e) For the 'follow the leader' esque control of this question, it was found that rather than implementing a controller that follows the other drone specifically based on distance, the control and behaviour of the 'follower' drone was much smoother when it simply followed the other drone with the same velocity and control. This created much less unstable behaviour and was overall more reliable. The controller is functionally the same as the controller in part 5a of this report, but the desired x and y positions are simply the x and y positions of the drone that is to be followed, rather than a point on a circle. Please see the function for controlling the follower drone below (please be aware of the fact this is not the whole function, this is just the parts of the function that differ from that of the other movement command functions):

```
1   function obj = fllo_move(obj, other_drone) % The first part of
        the function is absent as it is the same as other functions
        previously used
2       if (obj.time > 10 && sqrt(other_drone.pos(1)^2 + other_drone
            .pos(2)^2) < 0.3)
3           point = [0.3; 0];
4       else
5           point = [other_drone.pos(1); other_drone.pos(2)];
6       end
7
8       % difference in position between the follower drone and
            followed drone in x and y
9       diff_x = (point(1)+20) - (obj.pos(1)+20);
10      diff_y = (point(2)+20) - (obj.pos(2)+20);
11
12      % absolute distance
13      drone_dist = sqrt((other_drone.pos(1) - obj.pos(1))^2 + (
            other_drone.pos(2) - obj.pos(2))^2);
14      display(drone_dist)
15
16      obj.distances = [obj.distances, drone_dist];
17
18
19      % T_B and T_B_z are thrust in body frame, change here
20      integral_z_pos = obj.integral_err_z_pos + (obj.pos(3) -
            desired_z_pos) * obj.time_interval;
21      T_B_z = -Kp1*(obj.pos(3) - desired_z_pos) - Kd1*(obj.xdot(3)
            - desired_z_dot) - Ki1*integral_z_pos;
22
23      obj.integral_err_z_pos = integral_z_pos;
24      T_B = [0; 0; T_B_z];
25
26      a = g + (1 / m) * (R_mat*(T_B)) - kd .* obj.xdot;
```

```
27
28      mid_val = diff_x;
29      end_val = -diff_y;
30
31      t1 = (0.8 * mid_val);
32      t2 = (0.8 * end_val);
33
34      desired_theta = [0; t1 - 0.5 * obj.xdot(1); t2 + 0.5 * obj.
            xdot(2)];
35
36      desired_omega = [0; 0; 0];
37
38      % the rest of the function beyond this point is absent as it
            is the same as previous functions mentioned
39  end
```

A plot for the distance over time is shown below:

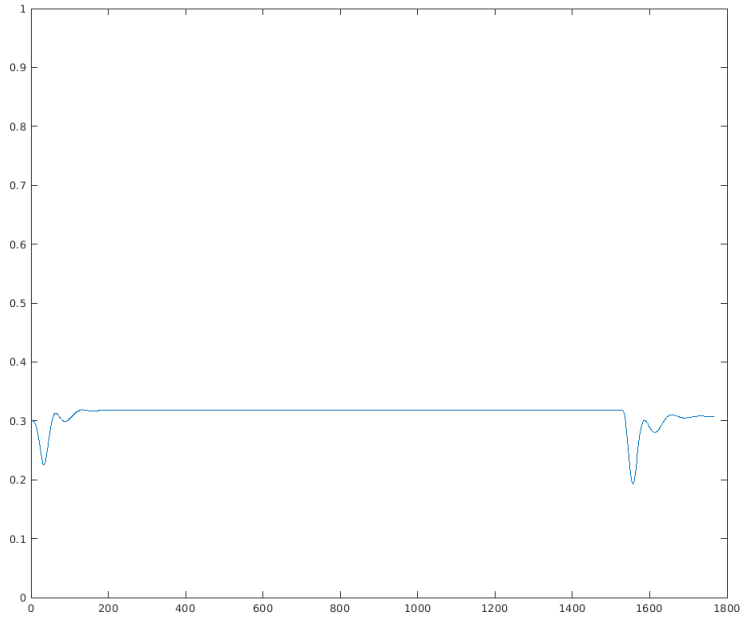

Figure 7: Distance between drones (y) over samples (x)

References

[1] Andrew Gibiansky. 2020. *Quadcopter Dynamics, Simulation And Control*. [online] andrew.gibiansky.com. Available at: [https://andrew.gibiansky.com/downloads/pdf/] [Accessed 30 November 2020].

[2] Andrew Symington, Renzo De Nardi, Simon Julier, Stephen Hailes. 2014. *Simulating quadrotor UAVs in outdoor scenarios*. Published by IEEE.

[3] Raimundo Felismina, Artur Mateus, Cândida Malça. 2017. *Study on the aerodynamic behavior of a UAV with an applied seeder for agriculturalpractices*. Published by AIP.

[4] Matija Krznar, Denis Kotarski, Petar Piljek, Danijel Pavkovic. 2018. *ON-LINE INERTIA MEASUREMENT OF UNMANNED AERIAL VEHICLES USING ON-BOARD SENSORS AND BIFILAR PENDULUM*. Published by INDECS.