DRAFT CODE SUMMARY

AE 352: Group Project

Jadyn Chowdhury | jadynnc2

05/05/2024

Table of Contents

BASE CODE

The following code stays the same for all cases.

CONTROLLER

The controller is the feedback system for the drone that derives torque motor outputs from the EOMs. Known to work on randomly generated courses.

```
import sympy as sym
from sympy import Matrix, diag, nsimplify, lambdify
import numpy as np
import scipy.signal
import scipy.linalg

class DroneDynamics:
    def __init__(self):
        """
         Initializes the Drone Dynamics class by defining symbolic variables for
drone's position, orientation,
        velocity, angular velocity, torques, and force. It also sets the drone's
physical parameters, initializes
        matrices, computes equations of motion, and calculates matrices A, B, C,
D, K, and L required for control
        and dynamics analysis.
        """
        # Define symbols
        self.p_x, self.p_y, self.p_z = sym.symbols('p_x, p_y, p_z')
        self.psi, self.theta, self.phi = sym.symbols('psi, theta, phi')
        self.v_x, self.v_y, self.v_z = sym.symbols('v_x, v_y, v_z')
        self.w_x, self.w_y, self.w_z = sym.symbols('w_x, w_y, w_z')
        self.tau_x, self.tau_y, self.tau_z = sym.symbols('tau_x, tau_y, tau_z')
        self.f_z = sym.symbols('f_z')

        # Define parameters
        self.params = {
            'm': 0.5,
            'Jx': 0.0023,
            'Jy': 0.0023,
            'Jz': 0.0040,
            'l': 0.175,
            'g': 9.81,
        }
        self.equilibrium_values = {
            'p_xe': 0, 'p_ye': 0, 'p_ze': 0,
            'psi_e': 0, 'theta_e': 0, 'phi_e': 0,
```

```
            'v_xe': 0, 'v_ye': 0, 'v_ze': 0,
            'w_xe': 0, 'w_ye': 0, 'w_ze': 0,
            'tau_xe': 0, 'tau_ye': 0, 'tau_ze': 0,
            'f_ze': 4.905  # 0.5 * 9.81
        }
        self._init_params()

        # Define other matrices and equations
        self._init_matrices()
        self._compute_equations()

        # Calculate A, B, C, D, K, L
        self._calculate_ABCD(self.equilibrium_values)
        self._calculate_KL()

    def _init_params(self):
        """
        Initializes the drone's physical parameters like mass, moments of inertia, arm length, and gravity.
        These parameters are essential for the dynamics and control of the drone. The method also computes
        an equilibrium force required to counteract gravity based on the drone's configuration.
        """
        self.m = nsimplify(self.params['m'])
        self.Jx = nsimplify(self.params['Jx'])
        self.Jy = nsimplify(self.params['Jy'])
        self.Jz = nsimplify(self.params['Jz'])
        self.l = nsimplify(self.params['l'])
        self.g = nsimplify(self.params['g'])
        self.J = diag(self.Jx, self.Jy, self.Jz)
        self.vxe = 1.0
        self.wye = 0.0
        self.vye = 1.0
        self.wxe = 0.0
        self.phie = 0.0
        self.thetae = 0.0
        self.fze = -(self.vxe * self.wye - self.vye*self.wxe - 981 * np.cos(self.phie) * np.cos(self.thetae) / 100) / 2

    def _init_matrices(self):
        """
        Initializes rotation matrices (Rz, Ry, Rx) representing the drone's orientation in 3D space.
        """
```

```
        # rotation matrices
        self.Rz = Matrix([[sym.cos(self.psi), -sym.sin(self.psi), 0],
                     [sym.sin(self.psi), sym.cos(self.psi), 0],
                     [0, 0, 1]])
        self.Ry = Matrix([[sym.cos(self.theta), 0, sym.sin(self.theta)],
                     [0, 1, 0],
                     [-sym.sin(self.theta), 0, sym.cos(self.theta)]])
        self.Rx = Matrix([[1, 0, 0],
                     [0, sym.cos(self.phi), -sym.sin(self.phi)],
                     [0, sym.sin(self.phi), sym.cos(self.phi)]])
        self.R_body_in_world = self.Rz @ self.Ry @ self.Rx

    def _compute_equations(self):
        """
        Computes the equations of motion for the drone. This includes the calculation of linear and angular
        velocities, transformation from angular velocity to angular rates, and the forces and torques acting
        on the drone. The output is a symbolic representation of the drone's dynamics.
        """
        # components of linear velocity
        v_in_body = Matrix([self.v_x, self.v_y, self.v_z])

        # components of angular velocity
        w_in_body = Matrix([self.w_x, self.w_y, self.w_z])

        # angular velocity to angular rates
        ex = Matrix([[1], [0], [0]])
        ey = Matrix([[0], [1], [0]])
        ez = Matrix([[0], [0], [1]])
        M = sym.simplify(Matrix.hstack((self.Ry @ self.Rx).T @ ez, self.Rx.T @ ey, ex).inv(), full=True)

        # applied forces
        f_in_body = self.R_body_in_world.T @ Matrix([[0], [0], [-self.m * self.g]]) + Matrix([[0], [0], [self.f_z]])

        # applied torques
        tau_in_body = Matrix([[self.tau_x], [self.tau_y], [self.tau_z]])

        # equations of motion
        f = Matrix.vstack(
            self.R_body_in_world @ v_in_body,
            M @ w_in_body,
```

```
            (1 / self.m) * (f_in_body - w_in_body.cross(self.m * v_in_body)),
            self.J.inv() @ (tau_in_body - w_in_body.cross(self.J @ w_in_body)),
        )

        self.f = sym.simplify(f, full=True)

        # Sensor model
        p_in_world = Matrix([self.p_x, self.p_y, self.p_z])
        a_in_body = Matrix([self.l, 0, 0])  # marker on front rotor
        b_in_body = Matrix([-self.l, 0, 0]) # marker on rear rotor

        a_in_world = p_in_world + self.R_body_in_world @ a_in_body
        b_in_world = p_in_world + self.R_body_in_world @ b_in_body

        self.g = sym.simplify(Matrix.vstack(a_in_world, b_in_world))

    def _calculate_ABCD(self, equilibrium_values):
        """
       Calculates the linearized system matrices (A, B, C, D) around an equilibrium
point. These matrices are
        essential for control system design, such as in linear quadratic regulator
(LQR) and observer design,
        which rely on a linear approximation of the drone's dynamics.
        """
         p_xe, p_ye, p_ze, psi_e, theta_e, phi_e, v_xe, v_ye, v_ze, w_xe, w_ye,
w_ze, tau_xe, tau_ye, tau_ze, f_ze = (
            equilibrium_values.get(k, 0) for k in [
                'p_xe', 'p_ye', 'p_ze', 'psi_e', 'theta_e', 'phi_e',
                'v_xe', 'v_ye', 'v_ze', 'w_xe', 'w_ye', 'w_ze',
                'tau_xe', 'tau_ye', 'tau_ze', 'f_ze'
            ]
        )

        self.A_num = sym.lambdify((self.p_x, self.p_y, self.p_z,
                                    self.psi, self.theta, self.phi,
                                    self.v_x, self.v_y, self.v_z,
                                    self.w_x, self.w_y, self.w_z,
                                    self.tau_x, self.tau_y, self.tau_z, self.f_z),
                                  self.f.jacobian([self.p_x, self.p_y, self.p_z, self.psi,
self.theta,
                                                   self.phi, self.v_x, self.v_y, self.v_z,
self.w_x,
                                                 self.w_y, self.w_z]))
         latex_matrix = sym.latex(self.f.jacobian([self.p_x, self.p_y, self.p_z,
self.psi, self.theta,
```

```
                                            self.phi, self.v_x, self.v_y, self.v_z,
self.w_x,
                                          self.w_y, self.w_z]))
        #print(latex_matrix)

        self.B_num =  sym.lambdify((self.p_x, self.p_y, self.p_z,
                            self.psi, self.theta, self.phi,
                            self.v_x, self.v_y, self.v_z,
                            self.w_x, self.w_y, self.w_z,
                            self.tau_x, self.tau_y, self.tau_z, self.f_z),
                             self.f.jacobian([self.tau_x, self.tau_y, self.tau_z,
self.f_z]))
             latex_matrix  =  sym.latex(self.f.jacobian([self.tau_x,  self.tau_y,
self.tau_z, self.f_z]))
        #print(latex_matrix)

        self.A = self.A_num(p_xe, p_ye, p_ze, psi_e, theta_e, phi_e, v_xe, v_ye,
v_ze, w_xe, w_ye, w_ze, tau_xe, tau_ye, tau_ze, f_ze)
        self.B = self.B_num(p_xe, p_ye, p_ze, psi_e, theta_e, phi_e, v_xe, v_ye,
v_ze, w_xe, w_ye, w_ze, tau_xe, tau_ye, tau_ze, f_ze)


        # Jacobian matrices C and D
           self.C_num  =  sym.lambdify((self.p_x,  self.p_y,  self.p_z,  self.psi,
self.theta),
             self.g.jacobian([self.p_x, self.p_y, self.p_z, self.psi, self.theta,
self.phi,
            self.v_x, self.v_y, self.v_z, self.w_x, self.w_y, self.w_z]))
         latex_matrix = sym.latex(self.g.jacobian([self.p_x, self.p_y, self.p_z,
self.psi, self.theta, self.phi,
            self.v_x, self.v_y, self.v_z, self.w_x, self.w_y, self.w_z]))
        #print(latex_matrix)
           self.D_num  =  sym.lambdify((self.p_x,  self.p_y,  self.p_z,  self.psi,
self.theta),
            self.g.jacobian([self.tau_x, self.tau_y, self.tau_z, self.f_z]))
             latex_matrix  =  sym.latex(self.g.jacobian([self.tau_x,  self.tau_y,
self.tau_z, self.f_z]))
        #print(latex_matrix)

        self.C = self.C_num(p_xe, p_ye, p_ze, psi_e, theta_e)
        self.D = self.D_num(p_xe, p_ye, p_ze, psi_e, theta_e)

    def _lqr(self, A, B, Q, R):
        """
```

```
        Solves the continuous-time linear quadratic regulator (LQR) problem. The
LQR controller is designed to
        minimize a cost function that balances the state error and control effort.
The function returns the
        optimal gain matrix K, which is used to control the system.

        :param A: System matrix.
        :param B: Input matrix.
        :param Q: State cost matrix.
        :param R: Control cost matrix.
        :return: Optimal gain matrix K.
        """
        P = scipy.linalg.solve_continuous_are(A, B, Q, R)
        K = np.linalg.inv(R) @ B.T @ P
        return K

    def _calculate_KL(self):
        """
        Calculates the feedback gain matrix K and the observer gain matrix L. K is
calculated using pole placement
        to ensure desired closed-loop behavior. L is calculated using the LQR
approach for the dual system
        (transposed system matrices) to design an observer that estimates the
system states from outputs.
        """
        pole = np.linspace(-1, -5, 12)
        K = scipy.signal.place_poles(self.A,self.B, pole).gain_matrix

        Qo = np.identity(self.C.shape[0])
        Ro = np.identity(self.A.shape[0])

        Qinv = np.linalg.inv(Qo)
        Rinv = np.linalg.inv(Ro)

        L = self._lqr(self.A.T, self.C.T, Rinv, Qinv).T

        self.K = K
        self.L = L

    def test_stable_K(self):
        """
        Tests the stability of the feedback gain matrix K. Stability is ensured if
the real parts of all
        eigenvalues of (A - B*K) are negative. This method prints a statement
regarding the stability of K.
```

```
        """
        eigens_K = np.linalg.eigvals(self.A - self.B @ self.K)
        if np.all(np.real(eigens_K) < 0):
            print('K matrix is stable')

    def test_stable_L(self):
        """
        Tests the stability of the observer gain matrix L. Stability is ensured if
the real parts of all
       eigenvalues of (A.T - C.T*L.T) are negative. This method prints a statement
regarding the stability of L.
        """
        eigens_L = np.linalg.eigvals(self.A.T - self.C.T @ self.L.T)
        if np.all(np.real(eigens_L) < 0):
            print('L matrix is stable')
    def check_controllability(self):
        """
        Checks if the system is controllable.
        """
        n = self.A.shape[0]  # number of states
        controllability_matrix = self.B
        self.W =controllability_matrix
        for i in range(1, n):
                    controllability_matrix  =  np.hstack((controllability_matrix,
np.linalg.matrix_power(self.A, i) @ self.B))

        if np.linalg.matrix_rank(controllability_matrix) == n:
            return True
        else:
            return False
    def check_observability(self):
        """
        Checks if the system is observable.
        """
        n = self.A.shape[0]  # number of states
        observability_matrix = self.C
        self.O = observability_matrix
        for i in range(1, n):
               observability_matrix  =  np.vstack((observability_matrix,  self.C  @
np.linalg.matrix_power(self.A, i)))

        if np.linalg.matrix_rank(observability_matrix) == n:
            return True
        else:
            return False
```

CONTROLLER CLASS

Feedback loop that provides output torques from input positions. Also logs all data for graphical analysis. Is adjusted later to account for velocity restrictions.

```
class Controller:
    def __init__(self):
        """
        List all class variables you want the simulator to log. For
        example, if you want the simulator to log "self.xhat", then
        do this:

            self.variables_to_log = ['xhat']

        Similarly, if you want the simulator to log "self.xhat" and
        "self.y", then do this:

            self.variables_to_log = ['xhat', 'y']

        Etc. These variables have to exist in order to be logged.
        """

        self.variables_to_log = ['xhat','x_des', 'v_des', 'rpm1', 'rpm2', 'rpm3',
'rpm4']
        self.dt  = .04
        self.A = drone.A
        self.B = drone.B
        self.C = drone.C
        self.D = drone.D
        self.K = drone.K
        self.L = drone.L
        self.fze = drone.fze
        self.base_rpm = 5000

    def get_color(self):
        """
        If desired, change these three numbers - RGB values between
        0 and 1 - to change the color of your drone.
        """
        return [
            0., # <-- how much red (between 0 and 1)
            1., # <-- how much green (between 0 and 1)
            0., # <-- how much blue (between 0 and 1)
        ]

    def reset(
```

```
            self,
            p_x, p_y, p_z, # <-- approximate initial position of drone (meters)
            yaw,           # <-- approximate initial yaw angle of drone (radians)
        ):
        self.xhat = np.array([p_x,p_y,p_z,yaw,0,0,0,0,0,0,0,0]).astype(float)
        self.x_des = np.zeros(12)

    def compute_motor_rpms(self, F, tau_x, tau_y, tau_z):
        delta_thrust = F / 4
        delta_pitch = tau_y / 0.02
        delta_roll = tau_x / 0.02
        delta_yaw = tau_z / 0.04

        rpm1 = self.base_rpm + delta_thrust - delta_pitch + delta_roll -
delta_yaw
        rpm2 = self.base_rpm + delta_thrust + delta_pitch + delta_roll +
delta_yaw
        rpm3 = self.base_rpm + delta_thrust + delta_pitch - delta_roll -
delta_yaw
        rpm4 = self.base_rpm + delta_thrust - delta_pitch - delta_roll +
delta_yaw

        return rpm1, rpm2, rpm3, rpm4

    def run(
            self,
            pos_markers,
            pos_ring,
            dir_ring,
            is_last_ring,                  # <-- True if next ring is the last
ring, False otherwise
            pos_others,                    # <-- 2d array of size n x 3, where n
is the number
                                           #     of all *other* drones - the ith
row in this array
                                           #     has the coordinates [x_i, y_i,
z_i], in meters, of
                                           #     the ith other drone
        ):
        """
        pos_markers is a 1d array of length 6:

            [
                measured x position of marker on front rotor (meters),
                measured y position of marker on front rotor (meters),
```

```
                measured z position of marker on front rotor (meters),
                measured x position of marker on back rotor (meters),
                measured y position of marker on back rotor (meters),
                measured z position of marker on back rotor (meters),
            ]

        pos_ring is a 1d array of length 3:

            [
                x position of next ring center (meters),
                y position of next ring center (meters),
                z position of next ring center (meters),
            ]

        dir_ring is a 1d array of length 3:

            [
                x component of vector normal to next ring (meters),
                y component of vector normal to next ring (meters),
                z component of vector normal to next ring (meters),
            ]

        is_last_ring is a boolean that is True if the next ring is the
                     last ring, and False otherwise

        pos_others is a 2d array of size n x 3, where n is the number of
                   all *other* drones:

            [
                [x_1, y_1, z_1], # <-- position of 1st drone (meters)
                [x_2, y_2, z_2], # <-- position of 2nd drone (meters)

                ...

                [x_n, y_n, z_n], # <-- position of nth drone (meters)
            ]
        """
        # GENERAL
        x_des=np.zeros(12)
        pos_est=self.xhat[0:3]
        e_max=.95
        if np.linalg.norm(pos_ring-pos_est)>e_max:
            p_des=pos_est+e_max*((pos_ring-pos_est)/np.linalg.norm(pos_ring-
pos_est))
            v_des=2.15*((pos_ring-pos_est)/np.linalg.norm(pos_ring-pos_est))
```

```
        else:
            p_des=pos_ring
            v_des=1.15*((pos_ring-pos_est)/np.linalg.norm(pos_ring-pos_est))
        x_des[0:3] = p_des
        x_des[6:9] = v_des
        self.x_des = x_des
        y = pos_markers
        u = -self.K@(self.xhat-self.x_des)
        self.xhat += self.dt*(self.A@self.xhat+self.B@u -
self.L@(self.C@self.xhat-y))

        self.v_des = v_des

        tau_x = u[0]
        tau_y = u[1]
        tau_z = u[2]
        f_z = u[3]+self.fze



        self.rpm1, self.rpm2, self.rpm3, self.rpm4 = self.compute_motor_rpms(f_z,
tau_x, tau_y, tau_z)

        return tau_x, tau_y, tau_z, f_z
```

HOVERING

Code related to task of hovering 1m above the ground.

CHECK RING

Simulation code that is adjusted to not require the drone to finish, instead can also allow for a max time to be reached. Needed to make sure drone operates for as long as necessary.

```
def check_ring(self, drone):
                                          position,         orientation         =
self.bullet_client.getBasePositionAndOrientation(drone['id'])
        position = np.array(position)

        # Check if drone['cur_ring'] is valid before accessing the ring
        if drone['cur_ring'] >= len(self.rings):
            drone['cur_ring'] = 0  # Reset to the first ring or handle as needed
            return False

        # Check if the drone is inside any ring and whether it's the final ring
```

```
        if self.is_inside_ring(self.rings[drone['cur_ring']], position):
            # If it's the final ring and time is less than max, delay finishing
            if drone['cur_ring'] == len(self.rings) - 1:
                if self.t < self.max_time_steps:
                    return False
                else:
                    drone['finish_time'] = self.t
                    drone['state'] = 'finished'
                    return True
            drone['cur_ring'] += 1  # Increment the current ring if not the final
or max time reached
        return False
```

RESET CODE

Used to set initial conditions and sensitivity.

```
# simulator.reset()
simulator.reset(
    initial_conditions={
        'template': {
            'p_x': 0.,
            'p_y': 0.,
            'p_z': 0.,
            'yaw': 0.,
            'pitch': 0.,
            'roll': 0.,
            'v_x': 0.,
            'v_y': 0.,
            'v_z': 0.,
            'w_x': 0.,
            'w_y': 0.,
            'w_z': 0.,
            'p_x_meas': 0.,
            'p_y_meas': 0.,
            'p_z_meas': 0.0000000000000001,
            'yaw_meas': 0.,
        },
    },
)
```

SIM CODE

Code to manually generate a course. In this case consisting of one ring 1m above the ground.

```
# PLACE FOR 1M ABOVE GROUND
    def place_rings(self):
        # Remove all existing rings
        for ring in self.rings:
            self.bullet_client.removeBody(ring['id'])
        self.meshcat_clear_rings()
        self.rings = []
        self.num_rings = 0

        # Define the position for a single hovering task
        hover_height = 1.  # 1 meter above ground
        ring_position = np.array([0., 0., hover_height])

        # Place a single ring at the hover position
          self.add_ring(ring_position, [0., np.pi / 2., 0.], 2.5, 0.5, 'big-
ring.urdf')

        # Set contact parameters for the ring
        for ring in self.rings:
            self.bullet_client.changeDynamics(ring['id'], -1,
                    lateralFriction=1.0,
                    spinningFriction=0.0,
                    rollingFriction=0.0,
                    restitution=0.5,
                    contactDamping=-1,
                    contactStiffness=-1)

        # Add the ring to meshcat
        self.meshcat_add_rings()
```

CONTROLLER CLASS

Set a limit on velocity to restrict sensitivity by avoiding larger output torques to improve stability.

```
class Controller:
    def __init__(self):
        """
        List all class variables you want the simulator to log. For
        example, if you want the simulator to log "self.xhat", then
        do this:

            self.variables_to_log = ['xhat']

        Similarly, if you want the simulator to log "self.xhat" and
        "self.y", then do this:
```

```
            self.variables_to_log = ['xhat', 'y']

        Etc. These variables have to exist in order to be logged.
        """

        self.variables_to_log = ['xhat','x_des', 'v_des', 'rpm1', 'rpm2', 'rpm3',
'rpm4']
        self.dt  = .04
        self.A = drone.A
        self.B = drone.B
        self.C = drone.C
        self.D = drone.D
        self.K = drone.K
        self.L = drone.L
        self.fze = drone.fze
        self.base_rpm = 5000

    def get_color(self):
        """
        If desired, change these three numbers - RGB values between
        0 and 1 - to change the color of your drone.
        """
        return [
            0., # <-- how much red (between 0 and 1)
            1., # <-- how much green (between 0 and 1)
            0., # <-- how much blue (between 0 and 1)
        ]

    def reset(
            self,
            p_x, p_y, p_z, # <-- approximate initial position of drone (meters)
            yaw,           # <-- approximate initial yaw angle of drone (radians)
        ):
        self.xhat = np.array([p_x,p_y,p_z,yaw,0,0,0,0,0,0,0,0]).astype(float)
        self.x_des = np.zeros(12)

    def compute_motor_rpms(self, F, tau_x, tau_y, tau_z):
        delta_thrust = F / 4
        delta_pitch = tau_y / 0.02
        delta_roll = tau_x / 0.02
        delta_yaw = tau_z / 0.04

        rpm1 = self.base_rpm + delta_thrust - delta_pitch + delta_roll -
delta_yaw
```

```
        rpm2 = self.base_rpm + delta_thrust + delta_pitch + delta_roll +
delta_yaw
        rpm3 = self.base_rpm + delta_thrust + delta_pitch - delta_roll -
delta_yaw
        rpm4 = self.base_rpm + delta_thrust - delta_pitch - delta_roll +
delta_yaw

        return rpm1, rpm2, rpm3, rpm4

    def run(
            self,
            pos_markers,
            pos_ring,
            dir_ring,
            is_last_ring,                 # <-- True if next ring is the last
ring, False otherwise
            pos_others,                   # <-- 2d array of size n x 3, where n
is the number
                                          #     of all *other* drones - the ith
row in this array
                                          #     has the coordinates [x_i, y_i,
z_i], in meters, of
                                          #     the ith other drone
        ):
        """
        pos_markers is a 1d array of length 6:

            [
                measured x position of marker on front rotor (meters),
                measured y position of marker on front rotor (meters),
                measured z position of marker on front rotor (meters),
                measured x position of marker on back rotor (meters),
                measured y position of marker on back rotor (meters),
                measured z position of marker on back rotor (meters),
            ]

        pos_ring is a 1d array of length 3:

            [
                x position of next ring center (meters),
                y position of next ring center (meters),
                z position of next ring center (meters),
            ]

        dir_ring is a 1d array of length 3:
```

```
        [
            x component of vector normal to next ring (meters),
            y component of vector normal to next ring (meters),
            z component of vector normal to next ring (meters),
        ]

    is_last_ring is a boolean that is True if the next ring is the
                 last ring, and False otherwise

    pos_others is a 2d array of size n x 3, where n is the number of
               all *other* drones:

        [
            [x_1, y_1, z_1], # <-- position of 1st drone (meters)
            [x_2, y_2, z_2], # <-- position of 2nd drone (meters)

            ...

            [x_n, y_n, z_n], # <-- position of nth drone (meters)
        ]
    """
    # MATCH SPECIFC VELOCITY
    x_des = np.zeros(12)
    pos_est = self.xhat[0:3]

    # Calculate the vector pointing from the current position to the ring
    direction_to_target = pos_ring - pos_est

    # Normalize the direction vector
    unit_direction_to_target = direction_to_target /
np.linalg.norm(direction_to_target)

    # Set the desired speed to 0.5 m/s
    desired_speed = 0.01  # m/s

    # Compute desired velocity vector with the magnitude of desired_speed
    v_des = unit_direction_to_target * desired_speed

    # Use the position of the ring directly if within allowable error margin
    e_max = 0.95
    if np.linalg.norm(direction_to_target) > e_max:
        p_des = pos_est + e_max * unit_direction_to_target
    else:
        p_des = pos_ring
```

```
        x_des[0:3] = p_des
        x_des[6:9] = v_des
        self.x_des = x_des
        self.v_des = v_des  # Log desired velocity for verification

        # Use the updated desired velocity in control calculations
        y = pos_markers
        u = -self.K @ (self.xhat - self.x_des)
        self.xhat += self.dt * (self.A @ self.xhat + self.B @ u - self.L @
(self.C @ self.xhat - y))

        # Compute torques and thrusts
        tau_x = u[0]
        tau_y = u[1]
        tau_z = u[2]
        f_z = u[3] + self.fze

        self.rpm1, self.rpm2, self.rpm3, self.rpm4 = self.compute_motor_rpms(f_z,
tau_x, tau_y, tau_z)

        return tau_x, tau_y, tau_z, f_z
```

GRAPHS

```
simulator.run(
    max_time=120.,       # <-- if None, then simulation will run until all drones fail or finish
    print_debug=True,   # <-- if False, then nothing will be printed (good for data collection)
)
✓ 2m 0.0s

Simulated 3001 time steps in 120.0454 seconds (24.9989 time steps per second)
```

Figure 1: Simulation completion time proof (2 minutes)

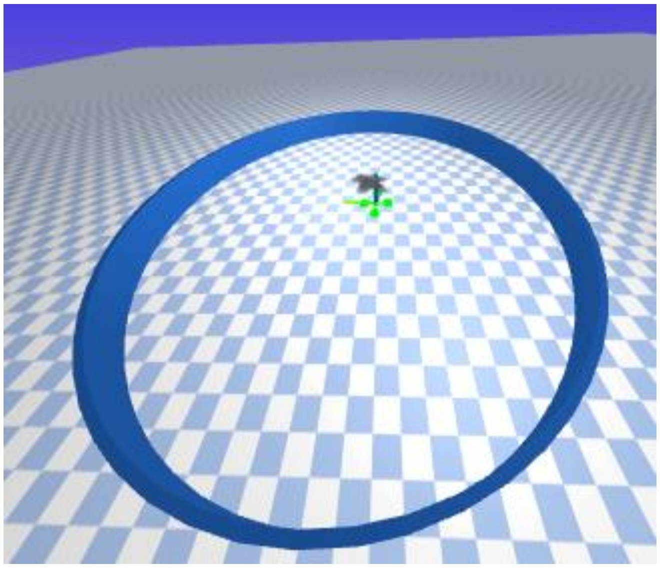

Figure 2: Screen capture of simulation (one course ring)

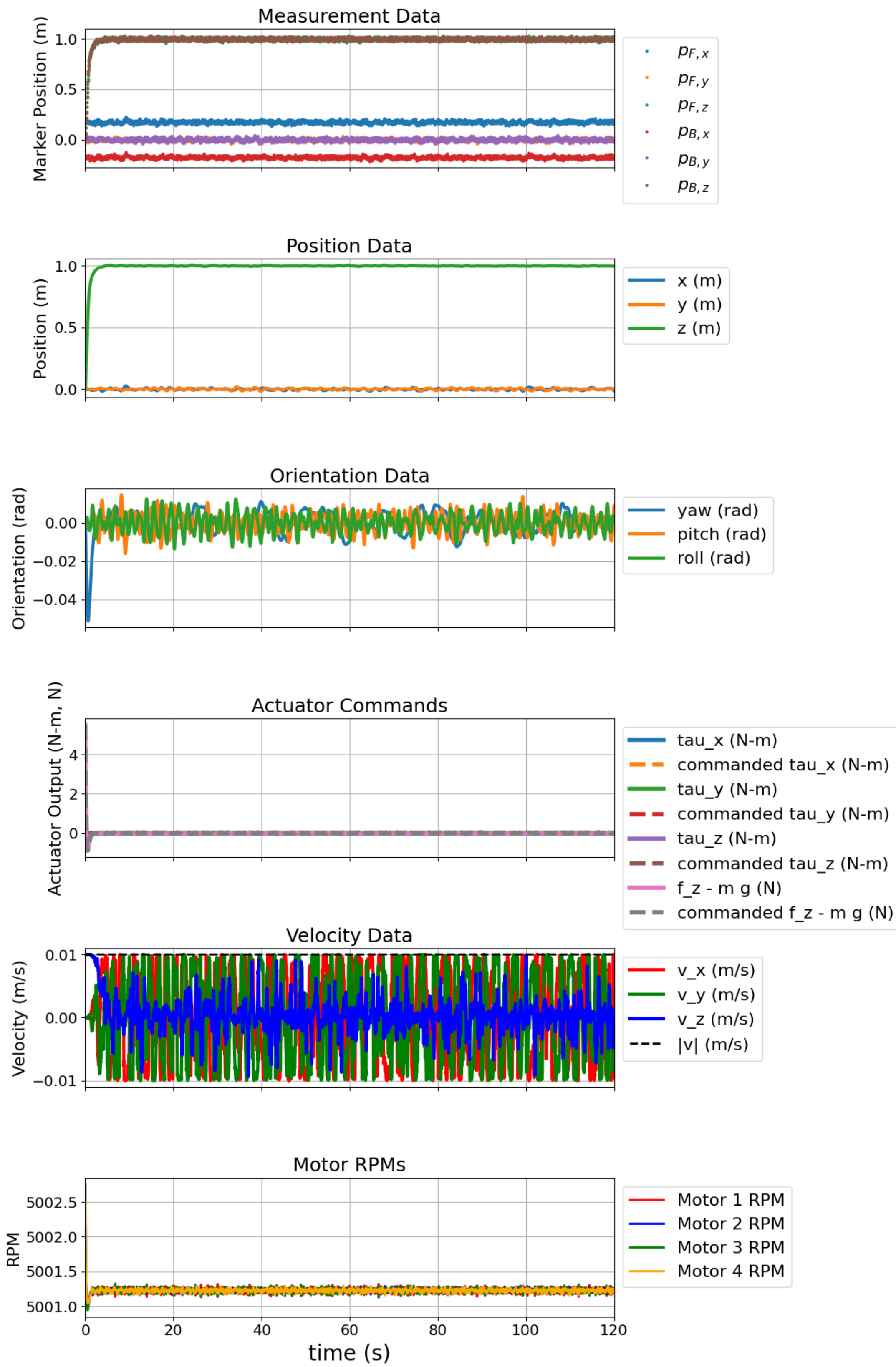

Figure 3: Graph of simulated variables. Area of focus here is z, fluctuating at 1m

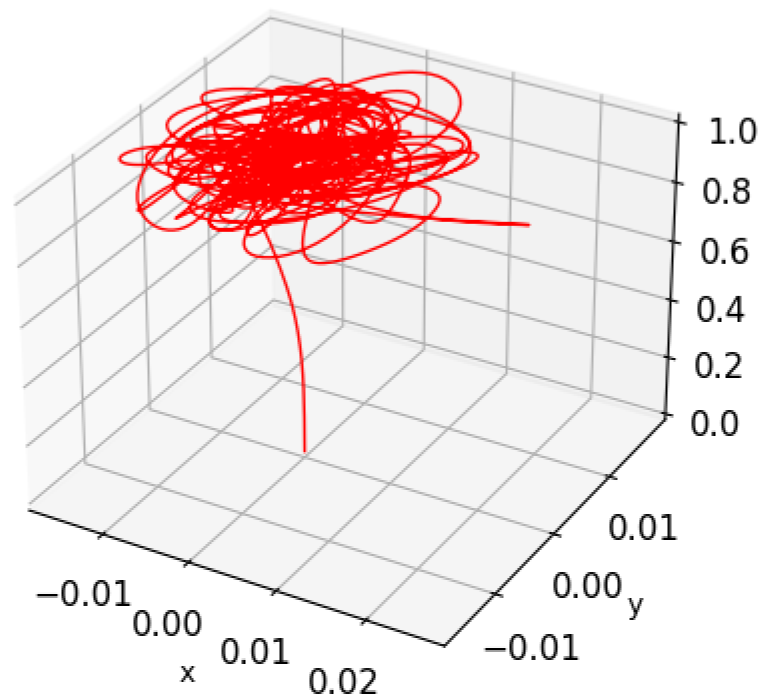

Figure 4: Plot of drone path in 3D space

EXPLANATIONS

The code works by creating a custom course with one ring. The drone starts at rest on the surface. Once the simulation starts the drone immediately accelerates upwards to 1m and fluctuates in its height by a small (insignificant, <0.01m) amount. The drones x and y positions vary in an oscillating sine function. Reasons are unknown exactly but presumed as a result of the sensitivity of the controller. This fluctuation was also insignificant, +-0.01m as per the trajectory graph. Overall success, Drone hovered at 1m for 2 minutes.

CIRCULAR PATH

Code needed for drone to circle in a radius of 2m at 1m above the ground.

CHECK RING

Similar to above but also rests current ring when drone finishes loop to ensure infinite cycles until time completed.

```
# CHECK RING FOR CIRCULAR PATH
    def check_ring(self, drone):
        position, orientation =
self.bullet_client.getBasePositionAndOrientation(drone['id'])
```

```
        position = np.array(position)

        # Wrap around the ring index to create a continuous loop
        if self.is_inside_ring(self.rings[drone['cur_ring']], position):
            # Increment the ring index or wrap it around if it's the last ring
            drone['cur_ring'] = (drone['cur_ring'] + 1) % len(self.rings)
            # If it's the last ring in the list, check if simulation should end
            if drone['cur_ring'] == 0 and self.t >= self.max_time_steps:
                drone['finish_time'] = self.t
                drone['running'] = False
                return True
        return False
```

RESET CODE

Same reasons as above. Setting to be within a ring point.

```
# simulator.reset()
simulator.reset(
    initial_conditions={
        'template': {
            'p_x': 1.,
            'p_y': 1.,
            'p_z': 0.5,
            'yaw': 0.,
            'pitch': 0.,
            'roll': 0.,
            'v_x': 0.,
            'v_y': 0.,
            'v_z': 0.,
            'w_x': 0.,
            'w_y': 0.,
            'w_z': 0.,
            'p_x_meas': 1.,
            'p_y_meas': 1.,
            'p_z_meas': 0.5,
            'yaw_meas': 0.,
        },
    },
)
```

SIM CODE

Manually create circular course using polar coordinate, parametrized equation for a circle at specific time intervals. Using tunnel function to create smooth continuous curves. Ensures that the path is cyclical.

```
# PLACE RINGS FOR CIRCLE
    def place_rings(self):
        # Remove all existing rings
        for ring in self.rings:
            self.bullet_client.removeBody(ring['id'])
        self.meshcat_clear_rings()
        self.rings = []
        self.num_rings = 0

        # Parameters for the circular course
        radius = 2.0  # radius of the circle in meters
        num_rings = 20  # number of rings to place around the circle
        height = 1.0  # height of the rings from the ground

        # Calculate positions for each ring on the circle
        angles = np.linspace(0, 2 * np.pi, num_rings, endpoint=False)  # Close
loop without duplicating first point

        points = []
        for angle in angles:
            x = radius * np.cos(angle)  # x position of the ring
            y = radius * np.sin(angle)  # y position of the ring
            z = height  # z position is constant at 1 meter
            points.append(np.array([x, y, z]))

        # Connect all points with tunnels, ensuring cyclic continuation
        for i in range(len(points)):
            start = points[i]
            end = points[(i + 1) % len(points)]  # Wrap around using modulo for
cyclic behavior
            next_point = points[(i + 2) % len(points)]

            self.add_tunnel(start, end, next_point)

        # Set contact parameters for all rings
        for ring in self.rings:
            self.bullet_client.changeDynamics(ring['id'], -1,
                                                lateralFriction=1.0,
                                                spinningFriction=0.0,
                                                rollingFriction=0.0,
                                                restitution=0.5,
                                                contactDamping=-1,
                                                contactStiffness=-1)

        # Add all rings to visualization
```

```
        self.meshcat_add_rings()
```

CONTROLLER CLASS

This implements a restriction on the desired velocity of the drone in tandem with the movement through the rings.

```
class Controller:
    def __init__(self):
        """
        List all class variables you want the simulator to log. For
        example, if you want the simulator to log "self.xhat", then
        do this:

            self.variables_to_log = ['xhat']

        Similarly, if you want the simulator to log "self.xhat" and
        "self.y", then do this:

            self.variables_to_log = ['xhat', 'y']

        Etc. These variables have to exist in order to be logged.
        """

        self.variables_to_log = ['xhat','x_des', 'v_des', 'rpm1', 'rpm2', 'rpm3',
'rpm4']
        self.dt  = .04
        self.A = drone.A
        self.B = drone.B
        self.C = drone.C
        self.D = drone.D
        self.K = drone.K
        self.L = drone.L
        self.fze = drone.fze
        self.base_rpm = 5000

    def get_color(self):
        """
        If desired, change these three numbers - RGB values between
        0 and 1 - to change the color of your drone.
        """
        return [
            0., # <-- how much red (between 0 and 1)
            1., # <-- how much green (between 0 and 1)
            0., # <-- how much blue (between 0 and 1)
```

```
        ]

    def reset(
            self,
            p_x, p_y, p_z, # <-- approximate initial position of drone (meters)
            yaw,           # <-- approximate initial yaw angle of drone (radians)
        ):
        self.xhat = np.array([p_x,p_y,p_z,yaw,0,0,0,0,0,0,0,0]).astype(float)
        self.x_des = np.zeros(12)

    def compute_motor_rpms(self, F, tau_x, tau_y, tau_z):
        delta_thrust = F / 4
        delta_pitch = tau_y / 0.02
        delta_roll = tau_x / 0.02
        delta_yaw = tau_z / 0.04

        rpm1 = self.base_rpm + delta_thrust - delta_pitch + delta_roll -
delta_yaw
        rpm2 = self.base_rpm + delta_thrust + delta_pitch + delta_roll +
delta_yaw
        rpm3 = self.base_rpm + delta_thrust + delta_pitch - delta_roll -
delta_yaw
        rpm4 = self.base_rpm + delta_thrust - delta_pitch - delta_roll +
delta_yaw

        return rpm1, rpm2, rpm3, rpm4

    def run(
            self,
            pos_markers,
            pos_ring,
            dir_ring,
            is_last_ring,                # <-- True if next ring is the last
ring, False otherwise
            pos_others,                  # <-- 2d array of size n x 3, where n
is the number
                                         #     of all *other* drones - the ith
row in this array
                                         #     has the coordinates [x_i, y_i,
z_i], in meters, of
                                         #     the ith other drone
        ):
        """
        pos_markers is a 1d array of length 6:
```

```
    [
        measured x position of marker on front rotor (meters),
        measured y position of marker on front rotor (meters),
        measured z position of marker on front rotor (meters),
        measured x position of marker on back rotor (meters),
        measured y position of marker on back rotor (meters),
        measured z position of marker on back rotor (meters),
    ]

pos_ring is a 1d array of length 3:

    [
        x position of next ring center (meters),
        y position of next ring center (meters),
        z position of next ring center (meters),
    ]

dir_ring is a 1d array of length 3:

    [
        x component of vector normal to next ring (meters),
        y component of vector normal to next ring (meters),
        z component of vector normal to next ring (meters),
    ]

is_last_ring is a boolean that is True if the next ring is the
             last ring, and False otherwise

pos_others is a 2d array of size n x 3, where n is the number of
           all *other* drones:

    [
        [x_1, y_1, z_1], # <-- position of 1st drone (meters)
        [x_2, y_2, z_2], # <-- position of 2nd drone (meters)

        ...

        [x_n, y_n, z_n], # <-- position of nth drone (meters)
    ]
"""

# MATCH SPECIFC VELOCITY
x_des = np.zeros(12)
pos_est = self.xhat[0:3]
```

```
        # Calculate the vector pointing from the current position to the ring
        direction_to_target = pos_ring - pos_est

        # Normalize the direction vector
        unit_direction_to_target = direction_to_target /
np.linalg.norm(direction_to_target)

        # Set the desired speed to 0.5 m/s
        desired_speed = 0.5  # m/s

        # Compute desired velocity vector with the magnitude of desired_speed
        v_des = unit_direction_to_target * desired_speed

        # Use the position of the ring directly if within allowable error margin
        e_max = 0.95
        if np.linalg.norm(direction_to_target) > e_max:
            p_des = pos_est + e_max * unit_direction_to_target
        else:
            p_des = pos_ring

        x_des[0:3] = p_des
        x_des[6:9] = v_des
        self.x_des = x_des
        self.v_des = v_des  # Log desired velocity for verification

        # Use the updated desired velocity in control calculations
        y = pos_markers
        u = -self.K @ (self.xhat - self.x_des)
        self.xhat += self.dt * (self.A @ self.xhat + self.B @ u - self.L @
(self.C @ self.xhat - y))

        # Compute torques and thrusts
        tau_x = u[0]
        tau_y = u[1]
        tau_z = u[2]
        f_z = u[3] + self.fze

        self.rpm1, self.rpm2, self.rpm3, self.rpm4 = self.compute_motor_rpms(f_z,
tau_x, tau_y, tau_z)

        return tau_x, tau_y, tau_z, f_z
```

GRAPHS

```
simulator.run(
    max_time=120.,        # <-- if None, then simulation will run until all drones fail or finish
    print_debug=True,  # <-- if False, then nothing will be printed (good for data collection)
)
```

✓ 2m 0.0s

```
Simulated 3001 time steps in 120.0570 seconds (24.9965 time steps per second)
```

Figure 5: Simulation completion time stamp (2 minutes)

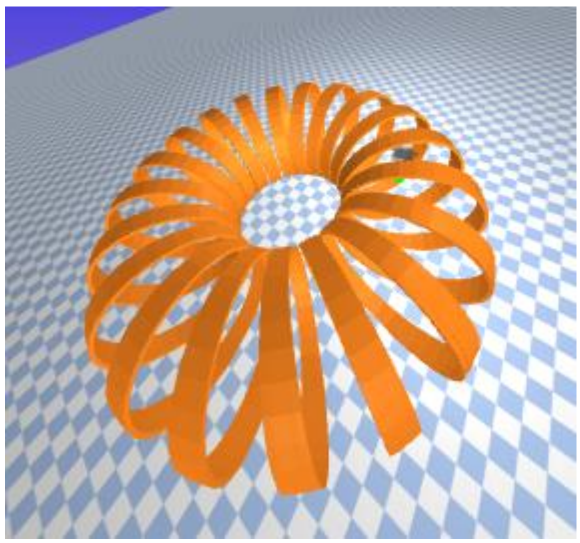

Figure 6: Screen Capture of simulation setup

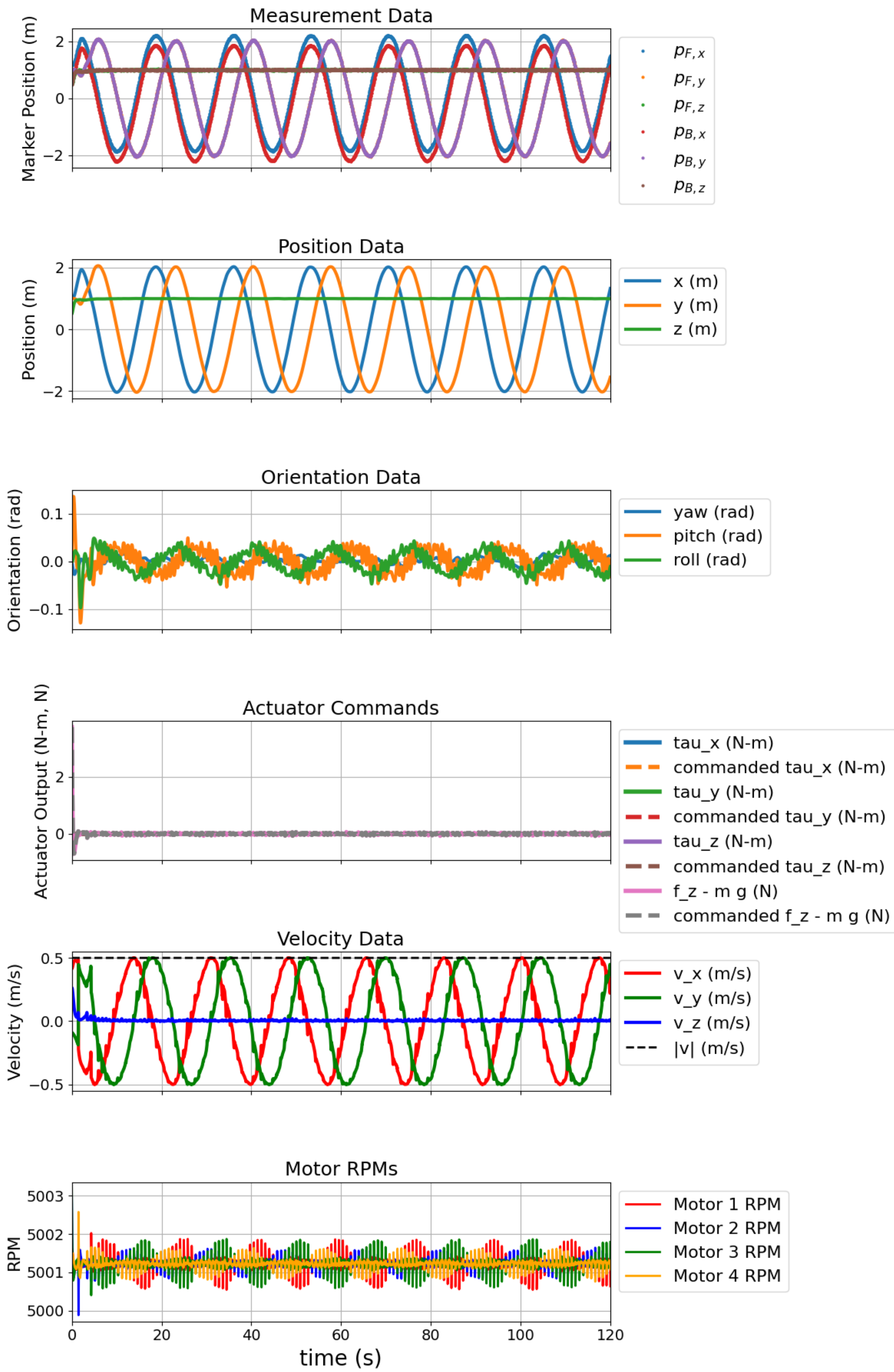

Figure 7: Graph of simulation variables. x, y, z of note

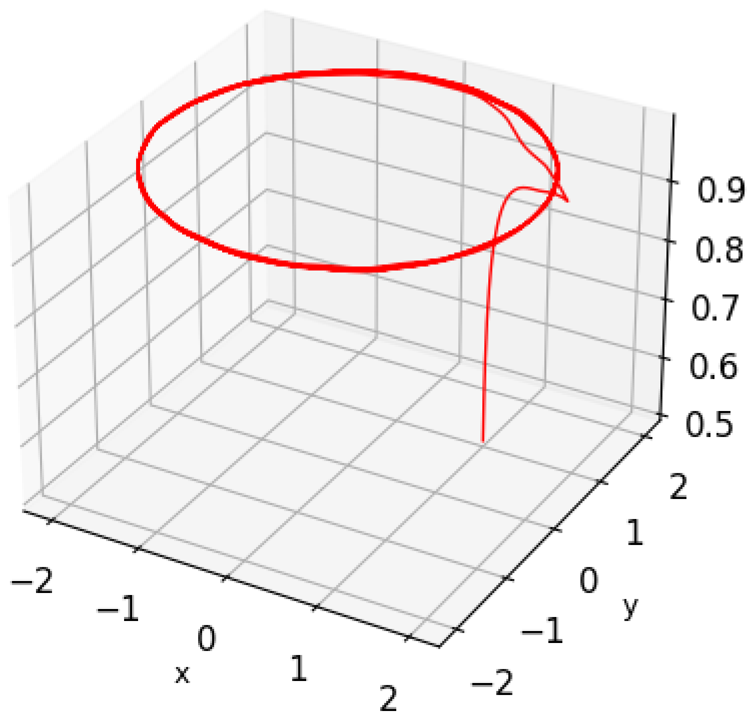

Figure 8: Plot in 3D space of drone trajectory

EXPLANATION

Drone starts at rest on the surface. It rises to 1m and begins circular path. It is very accurate as evident from sine curves of x and cosine of y of amplitudes 2 to -2 meaning circular motion. Z also stays stable at 1m. Simulation is run for 2 minutes. Double required time of 1 minute. Magnitude of velocities is shown to be constant at 0.5m/s.

SPECIFIC ROUTE

CHECK RING

Check ring function same as hover. Manages drones relative position to rings.

```
def check_ring(self, drone):
        position, orientation = 
self.bullet_client.getBasePositionAndOrientation(drone['id'])
        position = np.array(position)

        # Check if drone['cur_ring'] is valid before accessing the ring
        if drone['cur_ring'] >= len(self.rings):
            drone['cur_ring'] = 0  # Reset to the first ring or handle as needed
            return False
```

```
        # Check if the drone is inside any ring and whether it's the final ring
        if self.is_inside_ring(self.rings[drone['cur_ring']], position):
            # If it's the final ring and time is less than max, delay finishing
            if drone['cur_ring'] == len(self.rings) - 1:
                if self.t < self.max_time_steps:
                    return False
                else:
                    drone['finish_time'] = self.t
                    drone['state'] = 'finished'
                    return True
            drone['cur_ring'] += 1  # Increment the current ring if not the final
or max time reached
        return False
```

RESET CODE

Initialises drone position on the ground and sets the error parameters.

```
# simulator.reset()
simulator.reset(
    initial_conditions={
        'template': {
            'p_x': 0.,
            'p_y': 0.,
            'p_z': 0.,
            'yaw': 0.,
            'pitch': 0.,
            'roll': 0.,
            'v_x': 0.,
            'v_y': 0.,
            'v_z': 0.,
            'w_x': 0.,
            'w_y': 0.,
            'w_z': 0.,
            'p_x_meas': 0.0000000000000001,
            'p_y_meas': 0.0000000000000001,
            'p_z_meas': 0.0000000000000001,
            'yaw_meas': 0.0000000000000001,

        },
    },
)
```

SIM CODE

Is designed to create a custom course that has rings going forward for 5m, then left for another 5.

```
def place_rings(self):
        # Remove all existing rings
        for ring in self.rings:
            self.bullet_client.removeBody(ring['id'])
        self.meshcat_clear_rings()
        self.rings = []
        self.num_rings = 0


        # Move 5m above the ground x
        hover_height = 1.0  # 1 meter above the ground
        start_x = 0  # Starting x-coordinate
        # Place rings along a straight line, 1 meter apart
        num_rings = 6  # Total of 6 rings along 5 meters including origin
        for i in range(num_rings):
            x_position = start_x + i  # Calculate the x position for each ring
            ring_position = np.array([x_position, 0., hover_height])  # y is 0
since it's a straight line along x-axis

            # Place a ring at this position
            self.add_ring(ring_position, [0., 0., 0.], 2.5, 0.5, 'big-ring.urdf')

        # Move 5m above the ground y from x
        start_y = 1  # Starting y-coordinate offset from end of prev route
        for i in range(num_rings - 1):
            y_position = start_y + i  # Calculate the y position (to the left)
for each ring
            ring_position = np.array([5, y_position, hover_height])  # x is 5
since we start from the end of the previous route

            # Place a ring at this position
            self.add_ring(ring_position, [0., 0., -np.pi/2], 2.5, 0.5, 'big-
ring.urdf')



        # Set contact parameters for all rings
        for ring in self.rings:
            self.bullet_client.changeDynamics(ring['id'], -1,
                    lateralFriction=1.0,
                    spinningFriction=0.0,
                    rollingFriction=0.0,
```

```
                    restitution=0.5,
                    contactDamping=-1,
                    contactStiffness=-1)

        # Add all rings to visualization
        self.meshcat_add_rings()
```

ADD RING

Add rings function needed to be updated to disable collision detection since the done would collide with the two routes preventing it from finishing.

```
def add_ring(self, pos, ori, radius, width, urdf):
        if len(ori) == 3:
            # ori is rpy
            q = self.bullet_client.getQuaternionFromEuler(ori)
        elif len(ori) == 4:
            # ori is q
            q = ori
        else:
            raise Exception(f'ori must have length 3 or 4: {ori}')
        R = np.reshape(self.bullet_client.getMatrixFromQuaternion(q), (3, 3))

        # Load the URDF
        id = self.bullet_client.loadURDF(str(Path(f'./urdf/{urdf}')),
                        basePosition=pos,
                        baseOrientation=q,
                        useFixedBase=1)

        # Disable collision for this object
        self.bullet_client.setCollisionFilterGroupMask(id, -1, 0, 0)

        self.rings.append({
            'id': id,
            'p': np.array(pos),
            'R': R,
            'q': q,
            'radius': radius,
            'width': width,
        })
```

CONTROLLER CLASS

The run function in the Controller class manages the drone's navigation through checkpoints by calculating its proximity to specific targets, managing hover states at these checkpoints, and resuming normal flight with adjusted velocities and orientations post-hover. It updates the drone's desired state (position and velocity) continuously and calculates control inputs to align the drone's actual state with these targets, updating motor speeds accordingly.

```
class Controller:
    def __init__(self):
        """
        List all class variables you want the simulator to log. For
        example, if you want the simulator to log "self.xhat", then
        do this:

            self.variables_to_log = ['xhat']

        Similarly, if you want the simulator to log "self.xhat" and
        "self.y", then do this:

            self.variables_to_log = ['xhat', 'y']

        Etc. These variables have to exist in order to be logged.
        """

        self.variables_to_log = ['xhat','x_des', 'v_des', 'rpm1', 'rpm2', 'rpm3',
'rpm4']
        self.dt  = .04
        self.A = drone.A
        self.B = drone.B
        self.C = drone.C
        self.D = drone.D
        self.K = drone.K
        self.L = drone.L
        self.fze = drone.fze
        self.base_rpm = 5000
        self.hover_start_time = 0  # Time when hover started
        self.is_hovering = False   # Flag to indicate if currently hovering
        self.yaw_target = np.pi / 2
        self.current_checkpoint = 0


    def get_color(self):
        """
        If desired, change these three numbers - RGB values between
        0 and 1 - to change the color of your drone.
        """
        return [
            0., # <-- how much red (between 0 and 1)
            1., # <-- how much green (between 0 and 1)
            0., # <-- how much blue (between 0 and 1)
```

```
        ]

    def reset(
            self,
            p_x, p_y, p_z, # <-- approximate initial position of drone (meters)
            yaw,           # <-- approximate initial yaw angle of drone (radians)
        ):
        self.xhat = np.array([p_x,p_y,p_z,yaw,0,0,0,0,0,0,0,0]).astype(float)
        self.x_des = np.zeros(12)

    def compute_motor_rpms(self, F, tau_x, tau_y, tau_z):
        delta_thrust = F / 4
        delta_pitch = tau_y / 0.02
        delta_roll = tau_x / 0.02
        delta_yaw = tau_z / 0.04

        rpm1 = self.base_rpm + delta_thrust - delta_pitch + delta_roll -
delta_yaw
        rpm2 = self.base_rpm + delta_thrust + delta_pitch + delta_roll +
delta_yaw
        rpm3 = self.base_rpm + delta_thrust + delta_pitch - delta_roll -
delta_yaw
        rpm4 = self.base_rpm + delta_thrust - delta_pitch - delta_roll +
delta_yaw

        return rpm1, rpm2, rpm3, rpm4

    def run(
            self,
            pos_markers,
            pos_ring,
            dir_ring,
            is_last_ring,                # <-- True if next ring is the last
ring, False otherwise
            pos_others,                  # <-- 2d array of size n x 3, where n
is the number
            time,                        #     of all *other* drones - the ith
row in this array
                                         #     has the coordinates [x_i, y_i,
z_i], in meters, of
                                         #     the ith other drone

        ):
        """
        pos_markers is a 1d array of length 6:
```

```
    [
        measured x position of marker on front rotor (meters),
        measured y position of marker on front rotor (meters),
        measured z position of marker on front rotor (meters),
        measured x position of marker on back rotor (meters),
        measured y position of marker on back rotor (meters),
        measured z position of marker on back rotor (meters),
    ]

pos_ring is a 1d array of length 3:

    [
        x position of next ring center (meters),
        y position of next ring center (meters),
        z position of next ring center (meters),
    ]

dir_ring is a 1d array of length 3:

    [
        x component of vector normal to next ring (meters),
        y component of vector normal to next ring (meters),
        z component of vector normal to next ring (meters),
    ]

is_last_ring is a boolean that is True if the next ring is the
             last ring, and False otherwise

pos_others is a 2d array of size n x 3, where n is the number of
           all *other* drones:

    [
        [x_1, y_1, z_1], # <-- position of 1st drone (meters)
        [x_2, y_2, z_2], # <-- position of 2nd drone (meters)

        ...

        [x_n, y_n, z_n], # <-- position of nth drone (meters)
    ]
"""
p_des = pos_ring
v_des = np.zeros(3)

# Checkpoints and landing point
```

```
        checkpoint1 = np.array([5, 0, 1])
        checkpoint2 = np.array([5, 5, 1])
        landing_point = np.array([5, 5, 0])  # target z is 0 for landing

        pos_est = self.xhat[0:3]
        distance_to_checkpoint1 = np.linalg.norm(checkpoint1 - pos_est)
        distance_to_checkpoint2 = np.linalg.norm(checkpoint2 - pos_est)
        distance_to_ground = pos_est[2]  # z-coordinate for height above ground

        # Handling hovering and landing
        if distance_to_checkpoint1 < 0.1 and not self.is_hovering:
            self.is_hovering = True
            self.current_checkpoint = 1

        if distance_to_checkpoint2 < 0.1 and not self.is_hovering:
            self.is_hovering = True
            self.current_checkpoint = 2

        if self.is_hovering:
            if self.current_checkpoint == 1:
                if time <= 11:
                    p_des = checkpoint1
                    v_des = np.zeros(3)
                else:
                    self.is_hovering = False
                    self.x_des[3] += np.pi / 2  # Yaw change after first hover
            elif self.current_checkpoint == 2:
                if time <= 26:
                    p_des = checkpoint2
                    v_des = np.zeros(3)
                else:
                    self.is_hovering = False
        else:

            # Normal flight control
            direction_to_target = pos_ring - pos_est
            unit_direction_to_target = direction_to_target /
np.linalg.norm(direction_to_target)
            desired_speed = 0.5 if self.current_checkpoint == 1 else 1.0
            v_des = unit_direction_to_target * desired_speed
            p_des = pos_est + direction_to_target * 0.95 if
np.linalg.norm(direction_to_target) > 0.95 else pos_ring

        # Landing phase after checkpoint 2 hover
        if time > 26 and distance_to_ground >= 0:
```

```
            p_des = landing_point
            v_des = np.array([0, 0, 0.01])  # descending slowly
            if distance_to_ground <= 0.02:
                v_des = np.zeros(3)

        # Control updates
        x_des = np.zeros(12)
        x_des[0:3] = p_des
        x_des[6:9] = v_des
        self.x_des = x_des
        self.v_des = v_des

        y = pos_markers
        u = -self.K @ (self.xhat - self.x_des)
        self.xhat += self.dt * (self.A @ self.xhat + self.B @ u - self.L @
(self.C @ self.xhat - y))

        tau_x = u[0]
        tau_y = u[1]
        tau_z = u[2]
        if time > 26:
            f_z = 1
        else:
            f_z = u[3] + self.fze
        f_z = u[3] + self.fze
        # Motor RPM calculation
        self.rpm1, self.rpm2, self.rpm3, self.rpm4 = self.compute_motor_rpms(f_z,
tau_x, tau_y, tau_z)

        return tau_x, tau_y, tau_z, f_z
```

GRAPHS

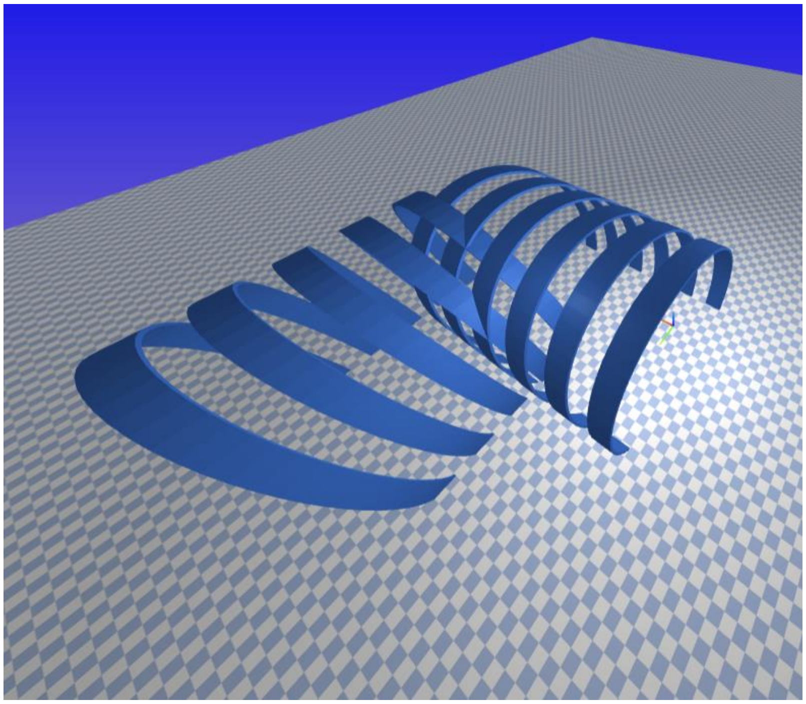

Figure 9 Special route course

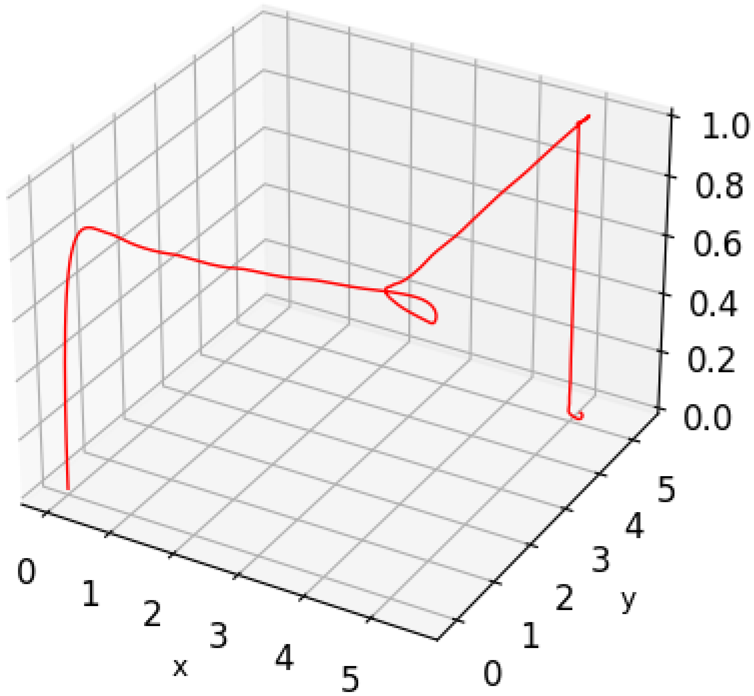

Figure 10 Drone trajectory for special route

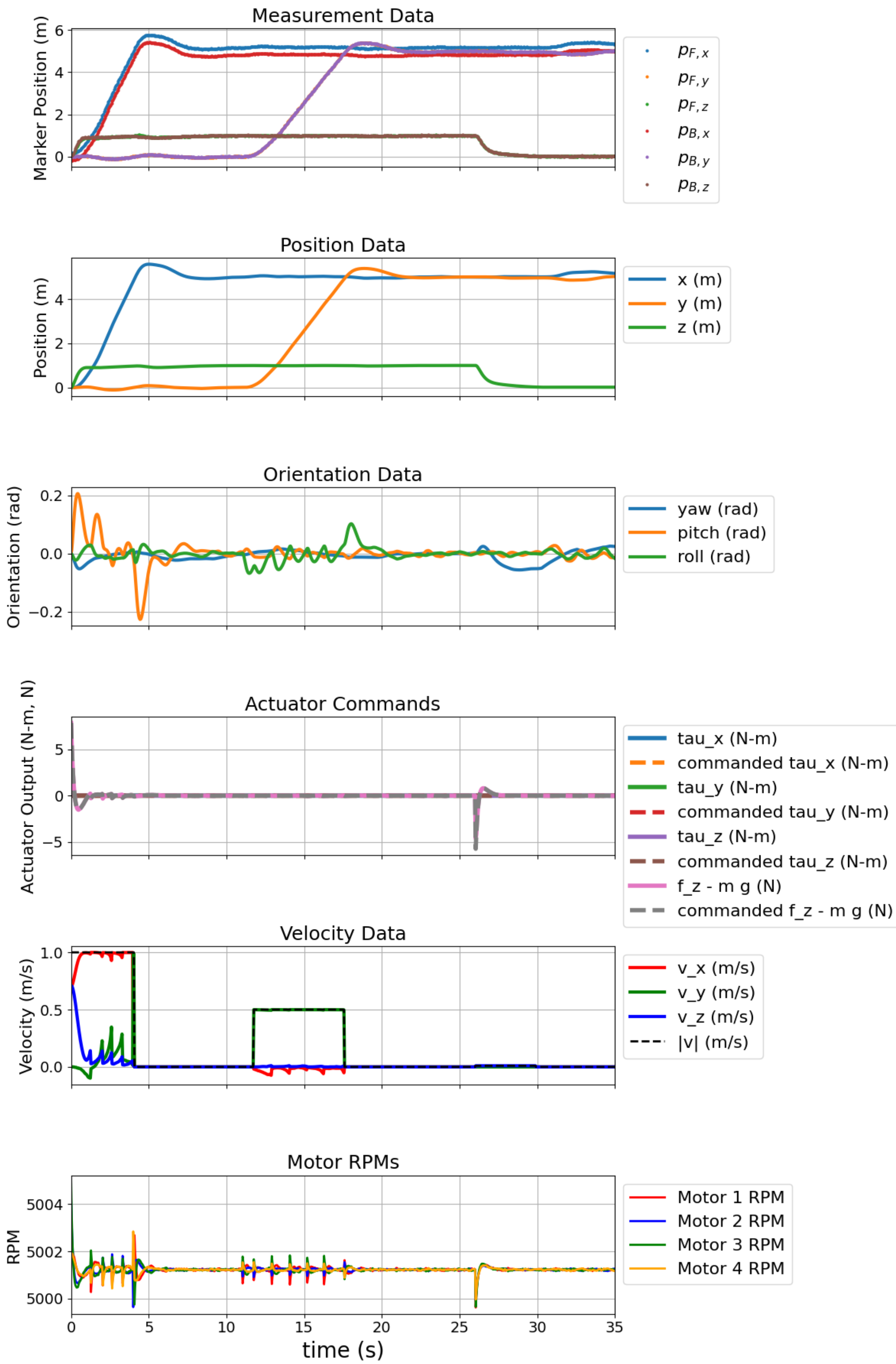

Figure 11 Special route data plots

EXPLANATION

If you look at the trajectory and position data, you see that the drone starts at the origin. The z axis almost immediately goes to 1 (drone climbed to 1m), then the x slowly increases indicating the drone is moving forward. The y stays at 0 at this point, indicating the drone is moving in a straight line. As the drone reaches 5m, it overshoots slightly and returns back to 5m, where it hovers in place at 1m above the ground. After a few seconds, the y axis begins to climb whilst the x stays constant. This shows the drone is now moving left on the same line it was hovering and at the same height of 1m. When the drone reaches 5m in this direction, it again overshoots but returns to 5m and hovers. After a few seconds, the x and y positions are constant, whilst the z drops quickly and then slowly plateauing at 0. This indicates that the drone is descending vertically and slowing as it nears the surface before landing.

If you compare these sections to the velocity diagram, you can see that as the drone moves to the first 5m mark, it travels at 1m/s. It's velocity is then 0 as it hovers, it then immediately accelerates to 0.5m/s as it starts moving left. The velocity then drops to 0 again as it hovers. Although difficult to see in the graphs, if you zoom in, you can see a small curve of velocity as the drone descends, at around 0.01m/s before reaching 0 as it lands.

These factors quantitively define that the goal was almost entirely achieved.

The yaw rotation was not possible to practically implement given the controller setup. Equilibrium, points are defined in the LQR matrices. These points have the drone tend to a specific value for its yaw, 0. Although this is relative to the ring, implementing direct yaw commands would be either overwritten by the controller, or cause it to break, as it would keep trying to correct it otherwise. In future it may be possible to implement dynamic systems assuming certain optimizations to the running of the simulation that would allow for varying commands related to the drones dynamics. As it stands, the drone responds autonomously to its environment and attempts to reach goals in the easiest way possible. Indeed, this approach seems more practical and relevant in the context of drone simulation and EOMs. It does more than what a person with a remote controller giving it direction commands could do, rather it acts independent of user control outside of environment setup.