Principles of Robot Autonomy: Homework 2

[Chetan Reddy] [Narayanaswamy]

10/17/24

Other students worked with: None
Time spent on homework: 9 hours

Problem 1:

Part (i)

The control inputs used are linear velocity and angular velocity. In the given problem, a constant unit linear velocity is used while the angular velocity is a sinusoidal input between -1 and 1. As seen in the figure, the turtlebot points towards the positive x-axis at t=0. Since the linear velocity is +1 and angular velocity is also positive, it enters the first quadrant and both x(t) and y(t) remain mostly in the first quadrant. The orientation is a function of the angular velocity alone and shows can expected curve which has the shape $1 - cos(\frac{2\pi}{N}t)$ which is the integral of the angular velocity from t=0 to t=t.

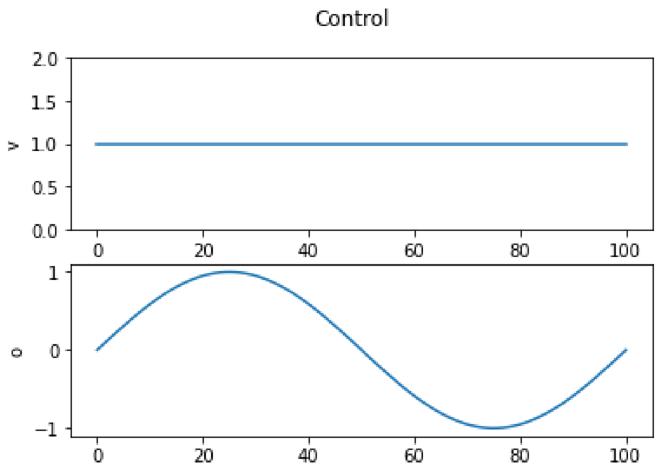

Figure 1: Control Trajectories with the Kinematic Model

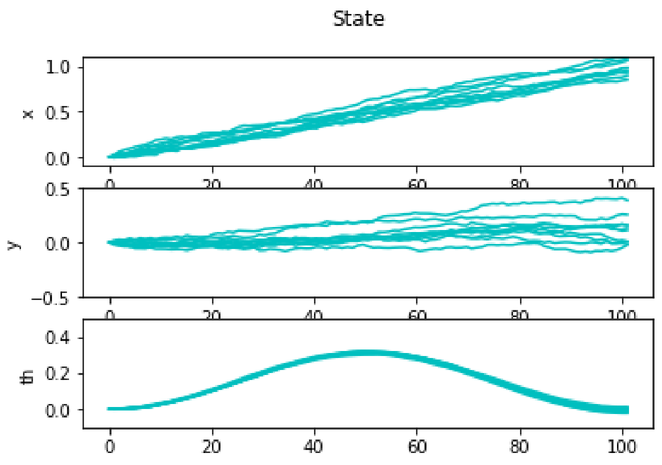

Figure 2: State Trajectories with the Kinematic Model

Part (ii)

For a single rollout:

$$x_{t+1} = Ax_t + Bu_t$$

We can vectorise the equation by stacking all the rollouts for a given timestep as a single tall vector.

$$X_{t+1} = \bar{A}X_t + \bar{B}u_t$$

where $X_t \in \mathbb{R}^{4N\times 1}, \bar{A} \in \mathbb{R}^{4N\times 4N}, \bar{B} \in \mathbb{R}^{4N\times 2}$ and $u_t \in \mathbb{R}^{2\times 1}$

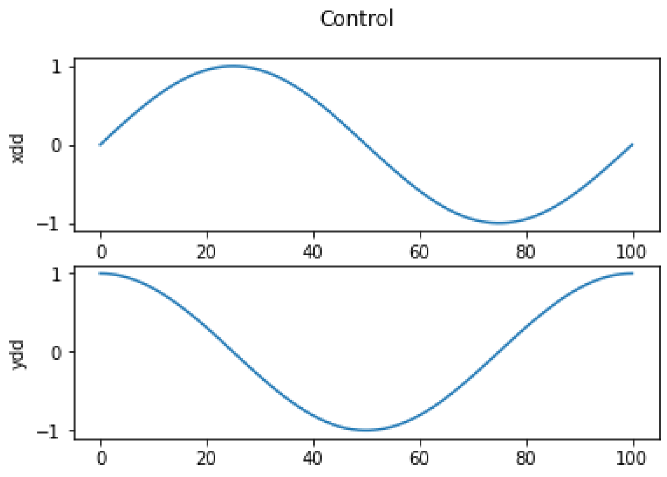

Figure 3: Control Trajectories with the Dynamic Model

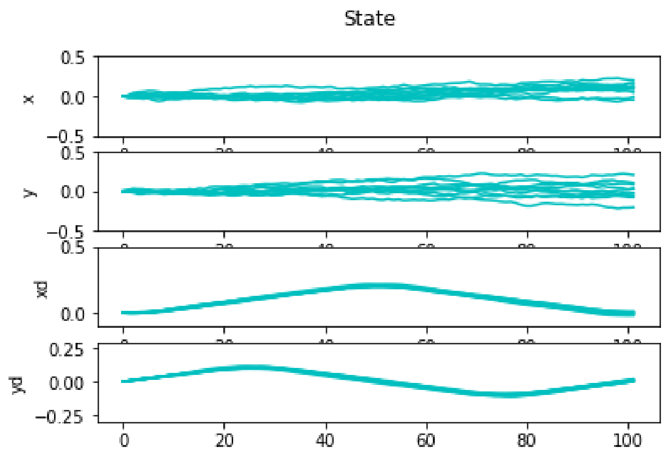

Figure 4: State Trajectories with the Dynamic Model

Similarities/Differences between Kinematic and Dynamic Model:

1. The states are only (x, y, θ) in the kinematic model while it is $(x, y, \dot{x}, \dot{y})$ in the dynamic model
2. For the kinematic model, we provide velocities (linear and angular) as the control input while we give accelerations as the control input in the dynamic model
3. The accelerations seem to obey circular motion in the dynamic model and we see that the x and y values oscillate around the zero value.

Problem 2:

Part (i)

The trajectory for the x and y component is expressed using a polynomial basis expansion, where the basis functions $\psi_i(t)$ are chosen as powers of t upto degree 3. The general form for $x(t), y(t)$ is:

$$x(t) = x_1\psi_1(t) + x_2\psi_2(t) + x_3\psi_3(t) + x_4\psi_4(t)$$
$$y(t) = y_1\psi_1(t) + y_2\psi_2(t) + y_3\psi_3(t) + y_4\psi_4(t)$$

Substituting the basis functions $\psi_1(t) = 1$, $\psi_2(t) = t$, $\psi_3(t) = t^2$, and $\psi_4(t) = t^3$ and also differentiating once wrt time, we have the expanded form of the equations:

$$\begin{aligned}
x(t) &= x_1 + x_2 t + x_3 t^2 + x_4 t^3 \\
y(t) &= y_1 + y_2 t + y_3 t^2 + y_4 t^3 \\
\dot{x}(t) &= 0 \cdot x_1 + x_2 + 2x_3 t + 3x_4 t^2 = v(t)cos(\theta(t)) \\
\dot{y}(t) &= 0 \cdot y_1 + y_2 + 2y_3 t + 3y_4 t^2 = v(t)sin(\theta(t))
\end{aligned}$$

Substituting Initial and Final Conditions

$$\begin{aligned}
x(0) &= x_1 + x_2 \cdot 0 + x_3 \cdot 0^2 + x_4 \cdot 0^3 && = 0\\
x(25) &= x_1 + x_2 \cdot 25 + x_3 \cdot 25^2 + x_4 \cdot 25^3 && = 5\\
\dot{x}(0) &= 0x_1 + x_2 + 2x_3 \cdot 0 + 3x_4 \cdot 0^2 && = 0\\
\dot{x}(25) &= 0x_1 + x_2 + 2x_3 \cdot 25 + 3x_4 \cdot 25^2 && = 0\\
y(0) &= y_1 + y_2 \cdot 0 + y_3 \cdot 0^2 + y_4 \cdot 0^3 && = 0\\
y(25) &= y_1 + y_2 \cdot 25 + y_3 \cdot 25^2 + y_4 \cdot 25^3 && = 5\\
\dot{y}(0) &= 0y_1 + y_2 + 2y_3 \cdot 0 + 3y_4 \cdot 0^2 && = -0.5\\
\dot{y}(25) &= 0y_1 + y_2 + 2y_3 \cdot 25 + 3y_4 \cdot 25^2 && = -0.5
\end{aligned}$$

Solving the system of equations, we get the values of the coefficients as follows:
$x_1 = 0,\ x_2 = 0,\ x_3 = 0.024,\ x_4 = -0.00064$ and $y_1 = 0,\ y_2 = -0.4933,\ y_3 = 0.0835,\ y_4 = -0.0022$

Part (ii)

The velocity cannot be set to zero because that would create a singularity in the jacobian matrix and it's inverse cannot be found.

Part (iii)

The value of the orientation can be found from the x and y components of the velocity. θ is found using the following formula (More precisely by considering the signs of the velocity):

$$\theta = arctan(\frac{v_y}{v_x})$$

The plotted trajectory for the state perfectly obeys the constraints on x, y, θ. We see that the orientation is $-\frac{\pi}{2}$ in the initial and final state. The generated control inputs in V and ω ensure that the trajectory is followed perfectly (when there are no distrubances)

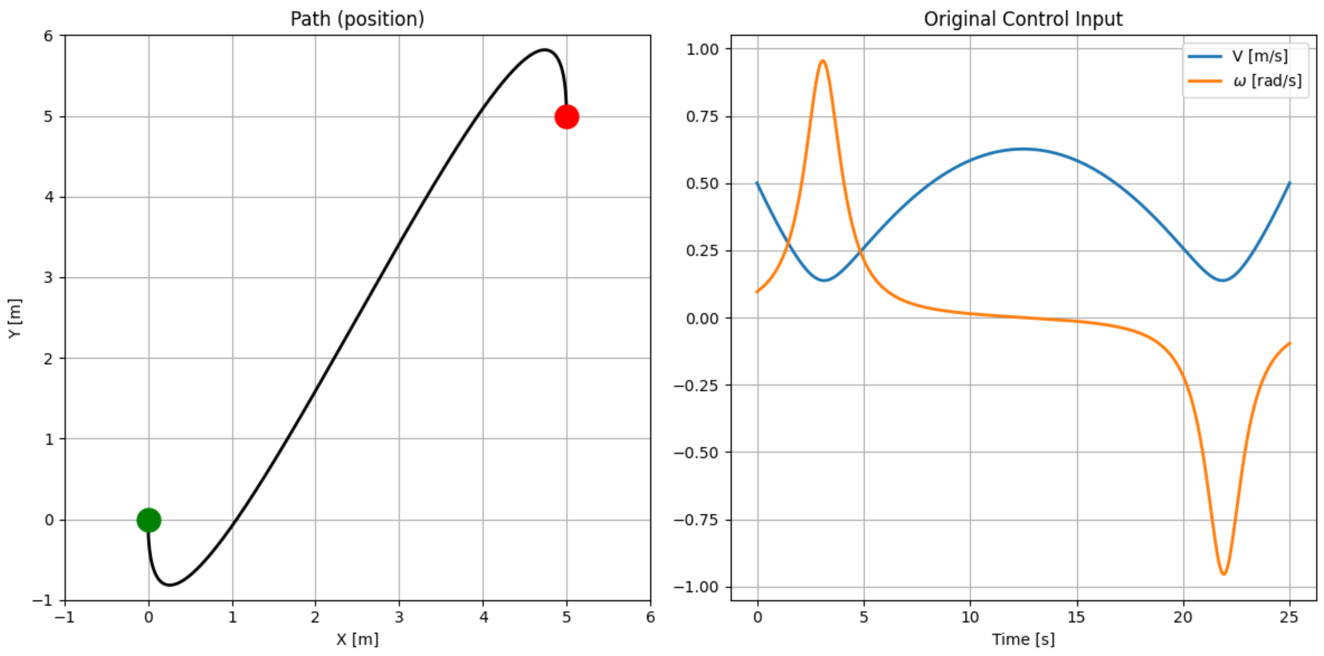

Figure 5: Computed Controls and Path (Differential Flatness)

Part (iv)

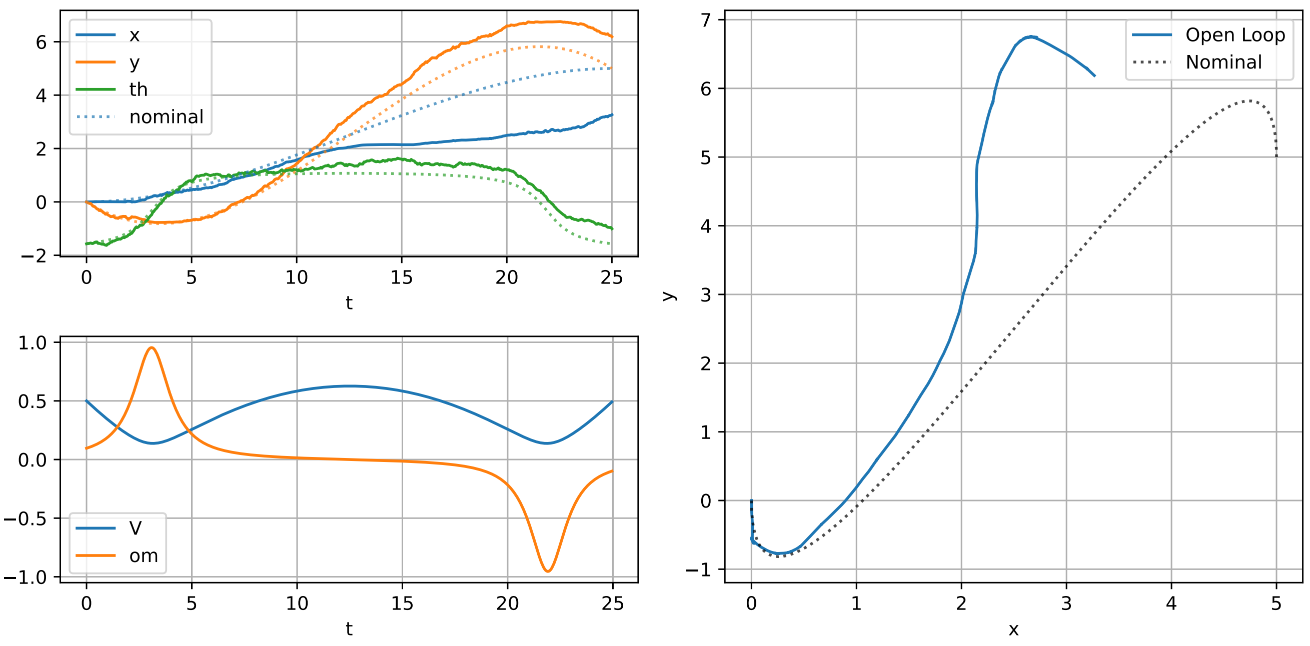

Figure 6: Simulated Trajectory in Open Loop (With Distrubances)

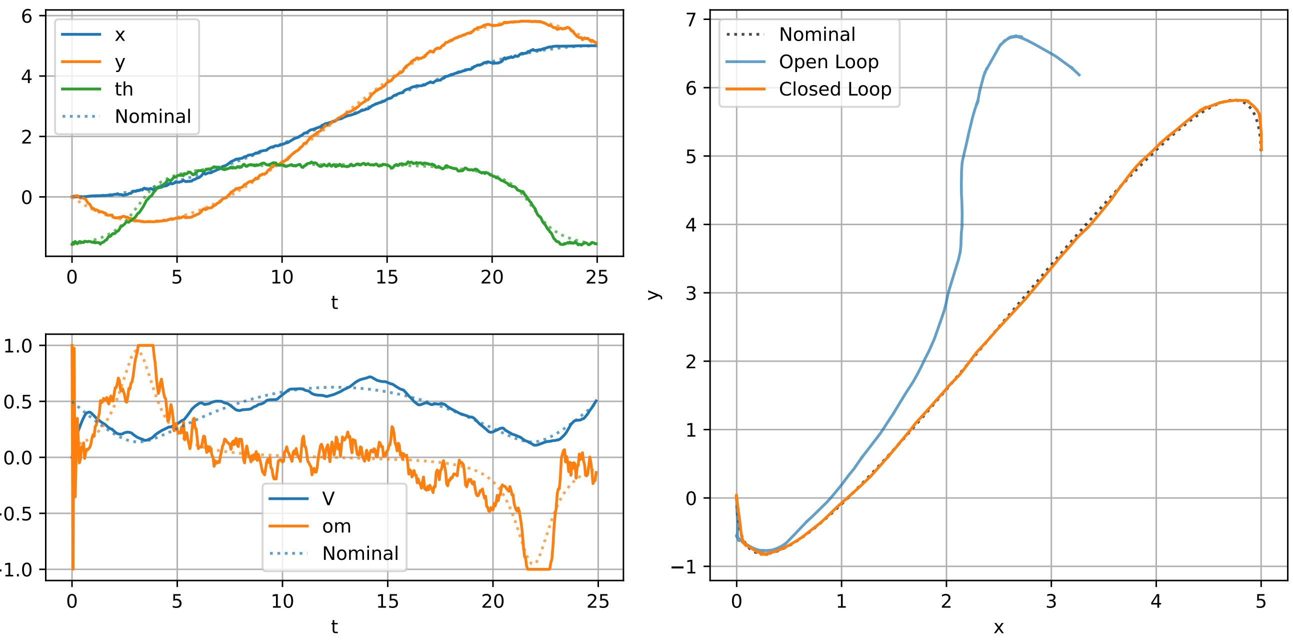

Figure 7: Simulated Trajectory in Closed Loop (with Disturbances)

Part (v)

The relation between virtual inputs and true inputs is derived below

$$\ddot{x}(t) = a\cos(\theta) - \omega v(t)\sin(\theta) = u_1 \tag{1}$$
$$\ddot{y}(t) = a\sin(\theta) + \omega v(t)\cos(\theta) = u_2 \tag{2}$$

Solving for a and ω, we get the following expression:

$$a = u_2\sin(\theta) + u_1\cos(\theta) = \dot{v}$$
$$\omega = (u_2\cos(\theta) - u_1\sin(\theta))/v(t)$$

(v, ω) can be found by using the following expressions in discrete time intervals of dt:

$$\omega = (u_2\cos(\theta) - u_1\sin(\theta))/v_{prev}$$
$$v = v_{prev} + \dot{v}dt$$

Part (vi)

(a) Using the jacobian, equations 1 and 2 could have been expressed as a single matrix equation and the inverse of the jacobian could have directly the values of a and ω

$$\begin{bmatrix} a \\ \omega \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -v\sin(\theta) \\ \sin(\theta) & v\cos(\theta) \end{bmatrix}^{-1} \begin{bmatrix} \ddot{x}(t) \\ \ddot{y}(t) \end{bmatrix}$$

$$\begin{bmatrix} a \\ \omega \end{bmatrix} = J^{-1} \begin{bmatrix} \ddot{x}(t) \\ \ddot{y}(t) \end{bmatrix}$$

(b) We see that the closed loop trajectory (in orange) is robust to disturbances and closely follows the reference nominal path unlike the open loop trajectory (in blue) in the right image of Fig.7

Problem 3:

Part (i)

The dimensionality of the state space is 6. The state is given by the position in the vertical plane (x, y), translational velocity (v_x, v_y), pitch ϕ, and pitch rate ω.

Part (ii)

The dimensionality of the action space is 2. The controls are the thrusts (T_1, T_2) for the left and right prop respectively.

Part (iii)

The notebook uses a non-linear trajectory optimisation algorithm by defining a cost on the controls constraints on the dynamics and obstacle avoidance.

Part (iv)

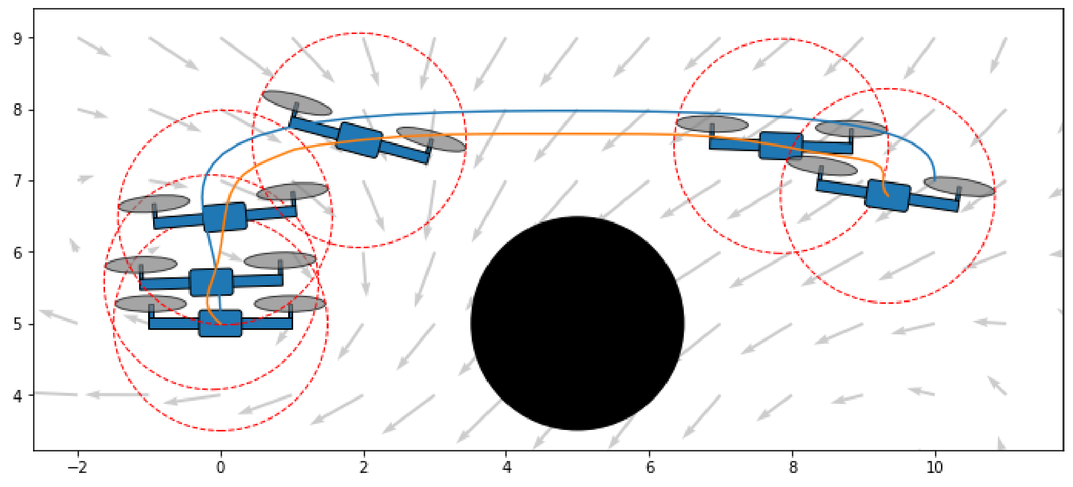

Figure 8: Trajectory of the Drone

(a) The dimensions of the gain matrix are (number of controls, number of states) and this is 2x6 i.e $K_i \in \mathbb{R}^{2\times 6}$

(b) R and Q matrices can be modified to improve the gain correction. They determine the cost and the right choice of the values can help the drone track the trajectory better.

Appendix A: Code Submission

Problem 1

```python
class TurtleBotDynamics(Dynamics):

  def __init__(self) -> None:
    super().__init__()
    self.n = 3  # length of state
    self.m = 2  # length of control

  def feed_forward(self, state:TurtleBotState, control:TurtleBotControl):
    # Define Gaussian Noise
    if self.noisy == False:
      var = np.array([0.0, 0.0, 0.0])
    elif self.noisy == True:
      var = np.array([0.01, 0.01, 0.001])
    w = np.random.normal(loc=np.array([0.0, 0.0, 0.0]), scale=var)


    state_new = TurtleBotState()
    ############################## Code starts here ##############################
    """
    Populate "state_new" by applying discrete time dynamics equations. Use "self.dt"
    from the Dynamics base class.
    """
    state_new.x = state.x + control.v*np.cos(state.th)*self.dt
    state_new.y = state.y + control.v*np.sin(state.th)*self.dt
    state_new.th = state.th + control.o*self.dt
    ############################## Code ends here ##############################

    # Add noise
    state_new.x  = state_new.x  + w[0]
    state_new.y  = state_new.y  + w[1]
    state_new.th = state_new.th + w[2]
    return state_new

  def rollout(self, state_init, control_traj, num_rollouts):
    num_steps = control_traj.shape[1]

    state_traj_rollouts = np.zeros((self.n*num_rollouts, num_steps+1))
    ############################## Code starts here ##############################
    """
    Use two for-loops to loop through "num_rollouts" and "num_steps" to populate "
    state_traj_rollouts". Use the "feed_forward" function above.
    """

    # Shape of state_traj_rollouts is (3*num_rollouts,num_steps+1)
    for rollout in range(num_rollouts):
      state = state_init
      x,y,th = state.x,state.y,state.th
      state_traj_rollouts[rollout*self.n:rollout*self.n+self.n,0] = np.array([x,y,th])
      for step in range(1,num_steps+1):
        # Control input at t
        controls_step = control_traj[:,step-1]
        control_obj = TurtleBotControl()
        control_obj.v = controls_step[0]
        control_obj.o = controls_step[1]
        state = self.feed_forward(state,control_obj)

        x,y,th = state.x,state.y,state.th
        state_traj_rollouts[rollout*self.n:rollout*self.n+self.n,step] = np.array([x,y,
    th])

    ############################## Code ends here ##############################

    return state_traj_rollouts
```

```
class DoubleIntegratorDynamics(Dynamics):

  def __init__(self) -> None:
    super().__init__()
    self.xdd_max = 0.5 # m/s^2
    self.ydd_max = 0.5 # m/s^2
    self.n = 4
    self.m = 2

  def feed_forward(self, state:np.array, control:np.array):

    num_rollouts = int(state.shape[0] / self.n)

    # Define Gaussian Noise
    if self.noisy == False:
      var = np.array([0.0, 0.0, 0.0, 0.0])
    elif self.noisy == True:
      var = np.array([0.01, 0.01, 0.001, 0.001])

    var_stack = np.tile(var, (num_rollouts))
    w = np.random.normal(loc=np.zeros(state.shape), scale=var_stack)

    # State space dynamics
    A = np.array([[1.0, 0.0, self.dt, 0.0],
                  [0.0, 1.0, 0.0, self.dt],
                  [0.0, 0.0, 1.0, 0.0],
                  [0.0, 0.0, 0.0, 1.0]])

    B = np.array([[0.0, 0.0],
                  [0.0, 0.0],
                  [self.dt, 0.0],
                  [0.0, self.dt]])

    # Stack to parallelize trajectories
    A_stack = np.kron(np.eye(num_rollouts), A) # Imagine it like [[A 0 0 0....],[0 A 0
    0...],[0 0 A 0 ....]]
    B_stack = np.tile(B, (num_rollouts, 1))

    ############################## Code starts here ##############################
    """
    Construct "state_new" by applying discrete time dynamics vectorized equations. Will
    require use of "A_stack" and "B_stack".
    """
    # Shape of state is num_rollouts*4
    # Shape of control is (2,1)
```

```python

    # Clip control inputs based on xdd_max and ydd_max
    u1 = np.clip(control[0], -self.xdd_max, self.xdd_max)
    u2 = np.clip(control[1], -self.ydd_max, self.ydd_max)

    control_vec = np.array([u1, u2])

    # Update state based on dynamics
    state_new = A_stack @ state + B_stack @ control_vec

    ############################## Code ends here ##############################
    # Add noise
    state_new = state_new + w

    return state_new

  def rollout(self, state_init, control_traj, num_rollouts):

    num_steps = control_traj.shape[1]

    state_traj = np.zeros((self.n*num_rollouts, num_steps+1))
    state_traj[:,0] = np.tile(state_init, num_rollouts) # RHS generates a shape of (4*num_rollouts,)
    print(state_traj.shape)
    ############################## Code starts here ##############################
    """
    Populate "state_traj" using only one for-loop, along with the "feed_forward" function above.
    """
    for time_step in range(num_steps):

      control_step = control_traj[:, time_step]

      # Get current state
      current_state = state_traj[:, time_step]

      # Compute next state using feed_forward method
      next_state = self.feed_forward(current_state, control_step)

      # Update state trajectory
      state_traj[:, time_step + 1] = next_state

    ############################## Code ends here ##############################
    return state_traj
```

Problem 2

P2_differential_flatness.py

```python
def compute_traj_coeffs(initial_state: State, final_state: State, tf: float) -> np.ndarray:
    """
    Inputs:
        initial_state (State)
        final_state (State)
        tf (float) final time
    Output:
        coeffs (np.array shape [8]), coefficients on the basis functions

    Hint: Use the np.linalg.solve function.
    """
    ########## Code starts here ##########
    A = np.array([[1,0,0,0],
                  [1,tf,tf**2,tf**3],
                  [0,1,0,0],
                  [0,1,2*tf,3*tf**2]])

    bx = np.array([[initial_state.x],
                   [final_state.x],
                   [initial_state.V*np.cos(initial_state.th)],
                   [final_state.V*np.cos(final_state.th)]])

    by = np.array([[initial_state.y],
                   [final_state.y],
                   [initial_state.V*np.sin(initial_state.th)],
                   [final_state.V*np.sin(final_state.th)]])

    x_coeff = np.linalg.solve(A,bx)
    y_coeff = np.linalg.solve(A,by)
    coeffs = np.vstack([x_coeff,y_coeff])

    ########## Code ends here ##########
    return coeffs

def compute_traj(coeffs: np.ndarray, tf: float, N: int) -> T.Tuple[np.ndarray, np.
    ndarray]:
    """
    Inputs:
        coeffs (np.array shape [8]), coefficients on the basis functions
        tf (float) final_time
        N (int) number of points
    Output:
        t (np.array shape [N]) evenly spaced time points from 0 to tf
        traj (np.array shape [N,7]), N points along the trajectory, from t=0
            to t=tf, evenly spaced in time
    """
    t = np.linspace(0, tf, N) # generate evenly spaced points from 0 to tf
    traj = np.zeros((N, 7))
    ########## Code starts here ##########
    def get_PHI_t(t):
        return np.array([[1, t, t**2,t**3],
                         [0, 1, 2*t, 3*t**2],
                         [0, 0, 2, 6*t]])

    for ti,time_step in enumerate(t):
        PHI_t = get_PHI_t(time_step)
        x,x_dot,x_dotdot = PHI_t@coeffs[:4]
        y,y_dot,y_dotdot = PHI_t@coeffs[4:]
        theta = np.arctan2(y_dot,x_dot)
        traj[ti,:] = np.array([x,y,theta,x_dot,y_dot,x_dotdot,y_dotdot]).reshape(-1)

    ########## Code ends here ##########

    return t, traj

def compute_controls(traj: np.ndarray) -> T.Tuple[np.ndarray, np.ndarray]:
    """
    Input:
        traj (np.array shape [N,7])
    Outputs:
        V (np.array shape [N]) V at each point of traj
        om (np.array shape [N]) om at each point of traj
    """
    ########## Code starts here ##########
    N = traj.shape[0]
    V = np.zeros(N)
    om = np.zeros(N)
    for i, traj_point in enumerate(traj):
        x,y,theta,x_dot,y_dot,x_dotdot,y_dotdot = traj_point
        v = (x_dot**2 + y_dot**2)**0.5
        J = np.array([[np.cos(theta), -v*np.sin(theta)],
                      [np.sin(theta),v*np.cos(theta)]])

        a,omega = np.linalg.inv(J)@np.array([[x_dotdot],
                                              [y_dotdot]])

        V[i] = v
        om[i] = omega

    ########## Code ends here ##########

    return V, om
```

trajectory_tracking.py

```
def compute_control(self, x: float, y: float, th: float, t: float) -> T.Tuple[float,
    float]:
        """
        Inputs:
            x,y,th: Current state
            t: Current time
        Outputs:
            V, om: Control actions
        """

        dt = t - self.t_prev
        x_d, xd_d, xdd_d, y_d, yd_d, ydd_d = self.get_desired_state(t)

        ########## Code starts here ##########
        v = self.V_prev
        x_dot = v*np.cos(th)
        y_dot = v*np.sin(th)
        u1 = xdd_d + self.kpx*(x_d - x) + self.kdx*(xd_d - x_dot)
        u2 = ydd_d + self.kpy*(y_d - y) + self.kdy*(yd_d - y_dot)


        if abs(v)<V_PREV_THRES:
            if v==0:
                v = V_PREV_THRES
            else:
                v = np.sign(v)*V_PREV_THRES


        J = np.array([[np.cos(th), -v*np.sin(th)],
                      [np.sin(th), v*np.cos(th)]])

        a = u2*np.sin(th) + u1*np.cos(th)
        om = (u2*np.cos(th) - u1*np.sin(th))/v

        V = self.V_prev + a*dt

        ########## Code ends here ##########

        # apply control limits
        V = np.clip(V, -self.V_max, self.V_max)
        om = np.clip(om, -self.om_max, self.om_max)

        # save the commands that were applied and the time
        self.t_prev = t
        self.V_prev = V
        self.om_prev = om

        return V, om
```

Problem 3

```
def find_closest_nominal_state(current_state):
    ################ Your code here #############################################
    # Hint: This shouldn't take more than a couple lines
    distances = np.linalg.norm(nominal_states - current_state, axis=1)

    # Find the index of the minimum distance
    closest_state_idx = np.argmin(distances)
    ############################################################################
    return closest_state_idx

from scipy.linalg import solve_continuous_are as ricatti_solver
# We will now create a gain schedule for each point in our trajectory
gains_lookup = {} # This dictionary should map each time index of the state to a gains matrix
Q = 100 * np.diag([1., 0.1, 1., 0.1, 0.1, 0.1])
R = 1e0 * np.diag([1., 1.])

for i in range(len(nominal_states)):
    #################### Your code here #########################################
    # Hints:
    # This problem very closely follows the lecture notes! We highly recommend
    # going through them before attempting the problem if you haven't already
    # done so
    # 1. Use planar_quad.get_continuous_jacobians() to calculate the jacobians of the dynamics
    # 2. Use the import ricatti_solver function to get P. Note that this function
    # actually returns the transpose of P
    # 3. Find the gains and update the gains lookup dictionary with it
    # 4. Nominal controls are not defined for the last state. Set these to zero.

    if i == len(nominal_states) - 1:
        K = np.zeros((R.shape[0], Q.shape[0]))

    else:

        state_nominal = nominal_states[i]
        control_nominal = nominal_controls[i]

        A,B  = planar_quad.get_continuous_jacobians(state_nominal,control_nominal)
        P = ricatti_solver(A, B, Q, R)
        K = np.linalg.inv(R) @ B.T @ P.T

    ##############################################################################
    gains_lookup[i] = K

def simulate_closed_loop(initial_state, nominal_controls):
    states = [initial_state]
    for k in range(N):
        #################### Your code here ####################################
        # Add code to compute the new controls using the LQR controller
        # Hints:
        # 1. Find the closest nominal state to the current state and lookup
        # the corresponding gain matrix
        # 2. Use the closest nominal state, its corresponding control, and the
        # gain matrix to compute the adjusted controls for the current state
        current_state = states[k]
        state_closest_idx = find_closest_nominal_state(current_state)
        K = gains_lookup[state_closest_idx]

        nominal_state = nominal_states[state_closest_idx]
        nominal_control = nominal_controls[state_closest_idx]
        control = nominal_control - K@(current_state - nominal_state)
        ######################################################################
        control = np.clip(control, planar_quad.min_thrust_per_prop, planar_quad.
    max_thrust_per_prop)
        next_state = planar_quad.discrete_step(states[k], control, dt)
        next_state = apply_wind_disturbance(next_state, dt)
        states.append(next_state)
    return np.array(states)
```