Principles of Robot Autonomy: Homework 02

Purushotham Mani
10/14/24

Other students worked with: None
Time spent on homework: approx. 4.5 Hrs excluding report and Problem 4

Problem 1: Numpy and Class Inheritance

1. Non-Vectorized Implementation:

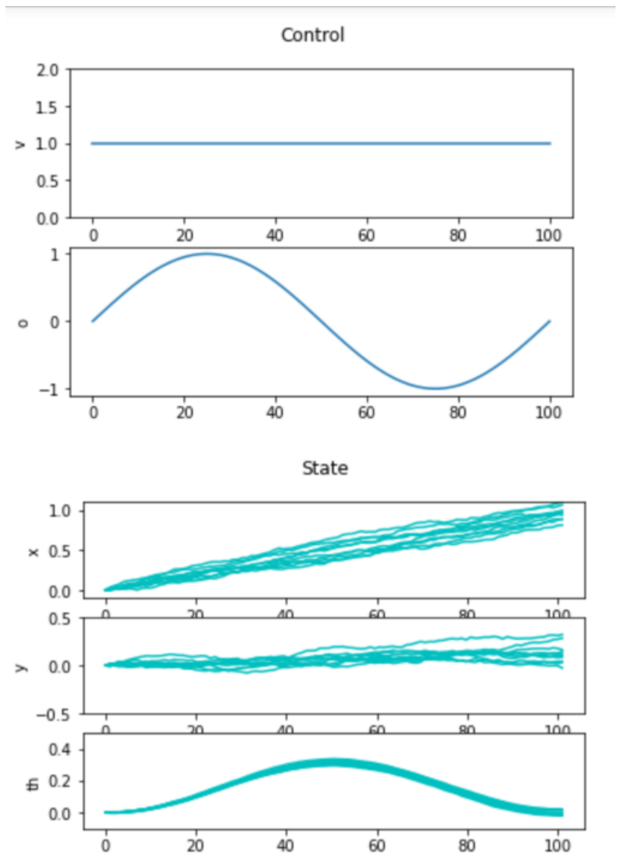

Figure 1: Control and State trajectory Plot

The initial state of the robot (x, y, θ) = (0,0,0).
Therefore, with v=1, you can see that the robot moves more in the x-direction with oscillation and just oscillates about the y-axis.
You can also see that because of sine like ω you get a -ve cosine like theta.

2. $X_{t+1} = \overline{A}X_t + \overline{B}U_t$
where X_t and U_t are the stacked state variable and stacked control variable at timestep t with state/control variables for each rollout stacked one after the other.

3. Vectorized Implementation:

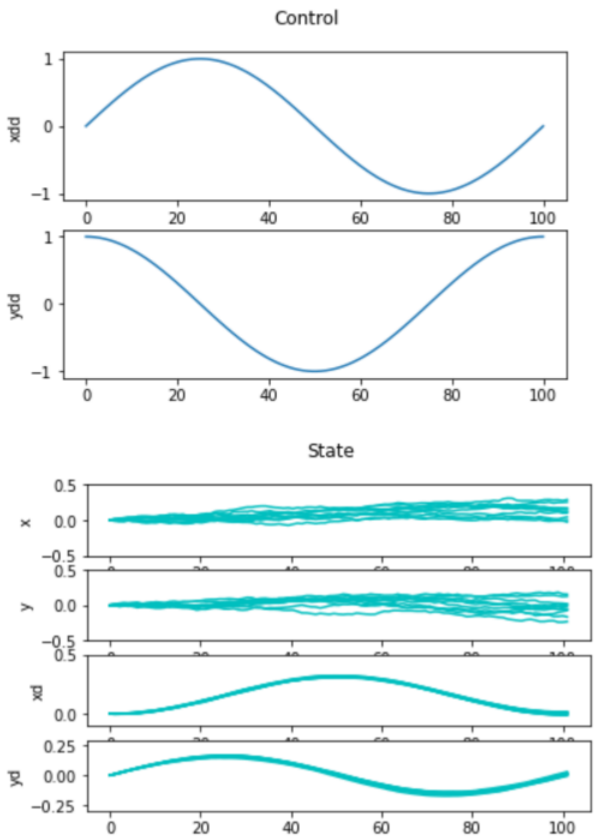

Figure 2: Control and State trajectory Plot

While the first approach used a kinematic model the second one uses a double integrator dynamics model. Because of these the state and form of control differ in both both approaches.
First approach:
state: (x, y, θ), control: $(V, \omega) = (1, \sin t)$ which corresponds to $(\ddot{x}, \ddot{y}) = (-\omega \sin \omega t, \omega \cos \omega t)$
Second approach:
state: $(x, y, \dot{x}, \dot{y})$, control: $(\ddot{x}, \ddot{y}) = (\sin t, \cos t)$
The plots of the state trajectory are also different because the control inputs are different.

Problem 2: Trajectory Generation using Differential Flatness

1. Spline:

$$\begin{bmatrix} 0 & 0 \\ 0 & -0.5 \\ 5 & 5 \\ 0 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix}$$

where $x_{1:4} = [0, 0, 0.024, -0.00064]$ and $y_{1:4} = [0, -0.5, 0.084, -0.00224]$ are coefficients when $t_f = 25$

2. We can't set $v(t) = 0$ because then matrix J loses rank and becomes non-invertible.

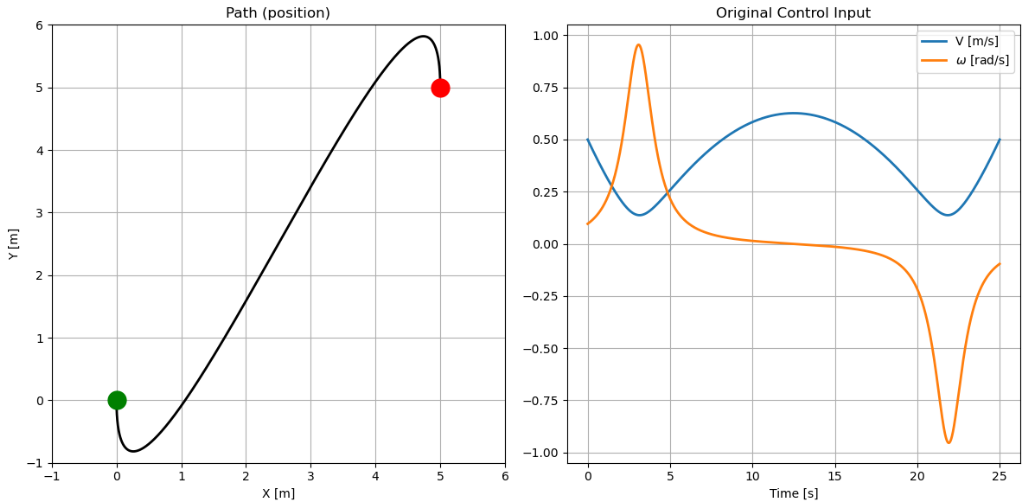

Figure 3: Differential Flatness - Open-Loop (Without disturbances)

3. (a) At every timestep t, I'd use the coefficients $x_{1:4}, y_{1:4}$ to compute state $(x, \dot{x}, \ddot{x}, y, \dot{y}, \ddot{y})$ and $\theta = \arctan(\dot{y}, \dot{x})$

   (b) Initial conditions: $x(0) = 0, y(0) = 0, v(0) = 0.5, \theta(0) = -\pi/2$
   Final conditions: $x(t_f) = 5, y(t_f) = 5, v(t_f) = 0.5, \theta(t_f) = -\pi/2$
   As one can see from the plotted trajectory, the green and red point do indeed correspond to the intial and final conditions respectively.

4. Open-Loop (With disturbances):

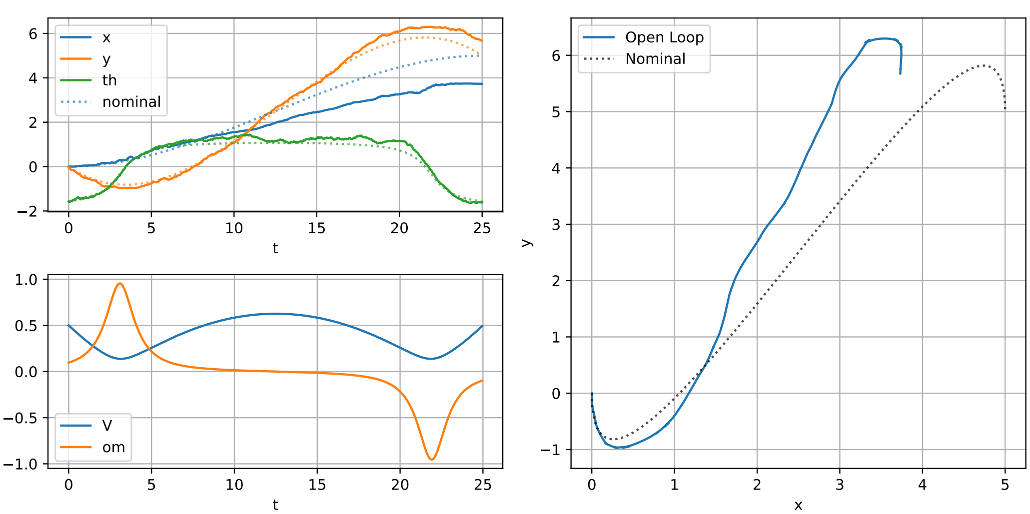

Figure 4: Trajectory Tracking without Feedback

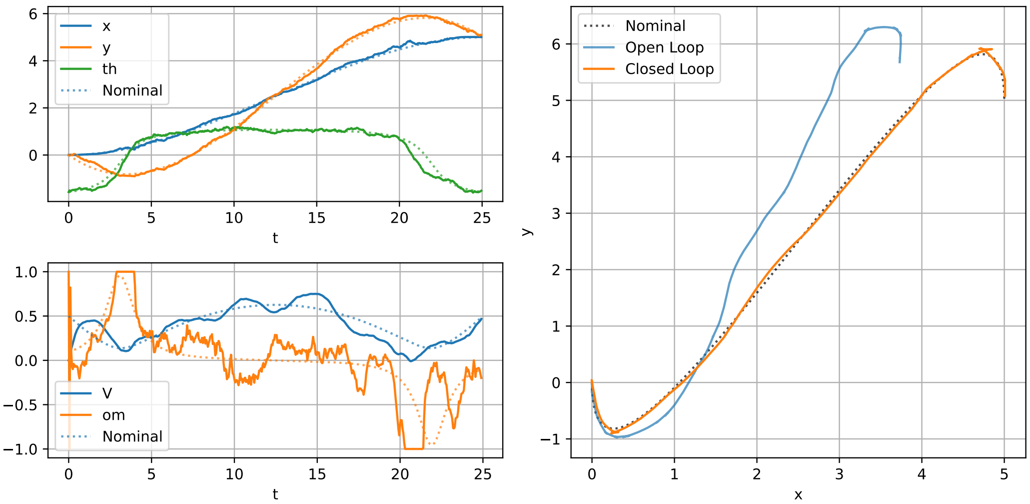

Figure 5: Closed-Loop (With disturbances)

5. Virtual Control to True Control:

$$\begin{bmatrix} \dot{v} \\ \omega \end{bmatrix} = \underbrace{\begin{bmatrix} \cos\theta & -v(t)\sin\theta \\ \sin\theta & v(t)\cos\theta \end{bmatrix}^{-1}}_{:=J^{-1}} \begin{bmatrix} \ddot{x}(t) \\ \ddot{y}(t) \end{bmatrix}$$

$$v(t+1) = v(t) + \dot{v}dt$$

6. (a) J can be used as shown above
   (b) The closed loop trajectory is better at tracking the trajectory in presence of disturbance compared to open-loop trajectory as can be seen from the orange vs. blue line in Fig. 5.

Problem 3: Gain-Scheduled LQR

1. The state-space of the quadrotor has a dimension of 6 represented by $(x, \dot{x}, y, \dot{y}, \phi, \dot{\phi})$. where $(x, \dot{x}, y, \dot{y})$ represents the global position of the quadrotor and their derivatives (global velocities) and $(\phi, \dot{\phi})$ represent quadrotors pitch and pitch-rate.

2. The control space of the quadrotor has a dimension of 2 and they represent individual thrusts of each rotor.

3. We use dicrete-time open-loop trajectory optimization using direct methods as discussed in class. We first discretize the formulation using 1st order Euler discretization and then minimize our objective function subject to contraints.

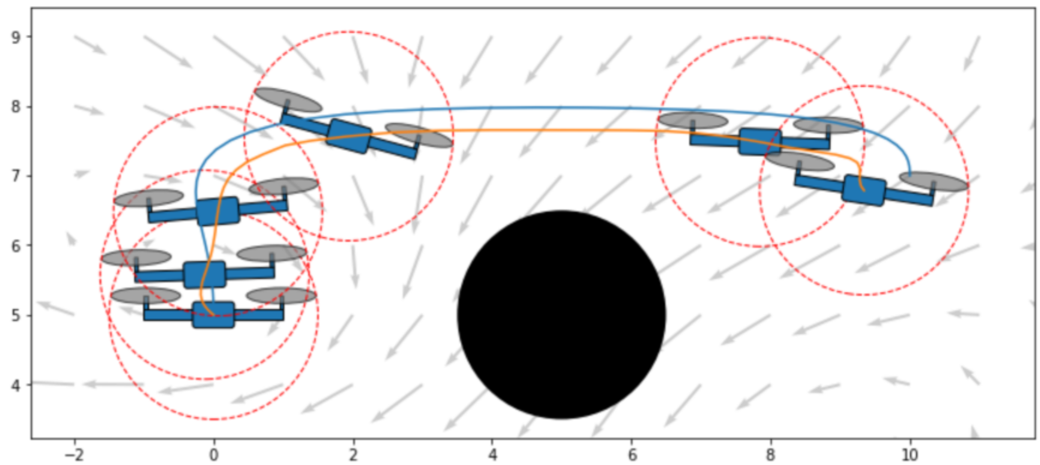

Figure 6: Quadrotor Simulation

4. (a) Dimension of gain matrix K_i: $Dim(K_i) = 2 \times 6$
   (b) We can tune Q and R matrices to track our nominal trajectories more precisely.

Appendix A: Code Submission

0.1 Problem 1: Non-Vectorized Implementation

```
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 ##############################
    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 ##############################
    for i in range(num_rollouts):
        state_traj_rollouts[3*i:3*(i+1),0] = np.array([state_init.x, state_init.y, state_init.th])
    for i in range (num_rollouts):
        for j in range(1, num_steps+1):
            control = TurtleBotControl(control_traj[0][j-1],control_traj[1][j-1])
            state = TurtleBotState(state_traj_rollouts[3*i:3*(i+1),j-1][0], state_traj_rollouts[3*i:3*(i+1),j-1][1], state_traj_rollouts[3*i:3*(i+1),j-1][2])
            state_new = self.feed_forward(state,control)
            state_traj_rollouts[3*i:3*(i+1),j] = np.array([state_new.x, state_new.y, state_new.th])
    ############################## Code ends here ##############################

    return state_traj_rollouts
```

0.2 Problem 1: Vectorized Implementation

```python
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)
    B_stack = np.tile(B, (num_rollouts, 1))
    ############################## Code starts here ##############################
    state_new = np.matmul(A_stack,state) + np.matmul(B_stack,control)
    ############################## 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)
    ############################## Code starts here ##############################
    for i in range(1,num_steps+1):
        state_traj[:,i] = self.feed_forward(state_traj[:,i-1],control_traj[:,i-1])
    ############################## Code ends here ##############################

    return state_traj

```

0.3 Problem 2: Open-Loop

```python
def compute_traj_coeffs(initial_state: State, final_state: State, tf: float) -> np.ndarray:
    ########## Code starts here ##########
    t0 = 0
    M = np.array([[1, t0, t0**2, t0**3],
                  [0, 1, 2*t0, 3*(t0**2)],
                  [1, tf, tf**2, tf**3],
                  [0, 1, 2*tf, 3*(tf**2)]])
    coeffs = np.zeros(8)
    x = np.array([initial_state.x, initial_state.xd, final_state.x, final_state.xd]).T
    y = np.array([initial_state.y, initial_state.yd, final_state.y, final_state.yd]).T
    coeffs[0:4] = np.linalg.solve(M,x)
    coeffs[4:] = np.linalg.solve(M,y)
    ########## Code ends here ##########

    return coeffs

def compute_traj(coeffs: np.ndarray, tf: float, N: int) -> T.Tuple[np.ndarray, np.ndarray]:
    t = np.linspace(0, tf, N) # generate evenly spaced points from 0 to tf
    traj = np.zeros((N, 7))
    ########## Code starts here ##########
    for i in range(N):
        M = np.array([[1, t[i], t[i]**2, t[i]**3],
                      [0, 1, 2*t[i], 3*(t[i]**2)],
                      [0, 0, 2, 6*t[i]]])
        x = np.matmul(M,coeffs[0:4])
        y = np.matmul(M,coeffs[4:])
        traj[i,:] = np.array([x[0], y[0], np.arctan2(y[1],x[1]), x[1], y[1], x[2], y[2]])
    ########## Code ends here ##########

    return t, traj

def compute_controls(traj: np.ndarray) -> T.Tuple[np.ndarray, np.ndarray]:
    V = np.sqrt(traj[:,3]**2 + traj[:,4]**2)
    om = np.zeros(traj.shape[0])
    ########## Code starts here ##########
    for i in range(traj.shape[0]):
        th = traj[i,2]
        M = np.array([[np.cos(th), -V[i]*np.sin(th)],
                      [np.sin(th), V[i]*np.cos(th)]])
        control = np.linalg.solve(M, traj[i,5:])
        om[i] = control[1]
    ########## Code ends here ##########

    return V, om
```

0.4 Problem 2: Closed-Loop

```
def compute_control(self, x: float, y: float, th: float, t: float) -> T.Tuple[float, float]:
    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 ##########
    self.V_prev = max(self.V_prev, V_PREV_THRES)
    u1 = xdd_d + self.kpx*(x_d-x) + self.kdx*(xd_d-(self.V_prev*np.cos(th)))
    u2 = ydd_d + self.kpy*(y_d-y) + self.kdy*(yd_d-(self.V_prev*np.sin(th)))

    U = np.array([u1, u2])
    M = np.array([[np.cos(th), -self.V_prev*np.sin(th)],
                [np.sin(th), self.V_prev*np.cos(th)]])
    control = np.linalg.solve(M, U)

    V = self.V_prev + control[0]*dt
    om = control[1]
    ########## 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
```

0.5 Problem 3: Gain-Scheduled LQR

```
def find_closest_nominal_state(current_state):
    ################ Your code here ############################################
    closest_state_idx = np.argmin(np.linalg.norm(current_state-nominal_states,axis=1))
    ############################################################################
    return closest_state_idx

from scipy.linalg import solve_continuous_are as ricatti_solver
gains_lookup = {}
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 ########################################
    if i==len(nominal_states)-1:
        nominal_control = np.array([0.0, 0.0])
        A, B = planar_quad.get_continuous_jacobians(nominal_states[i],nominal_control)
        P = ricatti_solver(A, B, Q, R)
        K = np.matmul(np.linalg.inv(R),np.matmul(B.T,P.T))
    else:
        A, B = planar_quad.get_continuous_jacobians(nominal_states[i],nominal_controls[i])
        P = ricatti_solver(A, B, Q, R)
        K = np.matmul(np.linalg.inv(R),np.matmul(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 ####################################
        j = find_closest_nominal_state(states[-1])
        if j==len(nominal_states)-1:
            nominal_control = np.array([0.0, 0.0])
            control = nominal_control-np.matmul(gains_lookup[j],(states[-1]-nominal_states[j]))
        else:
            control = nominal_controls[j]-np.matmul(gains_lookup[j],(states[-1]-nominal_states[j]))
        ######################################################################
        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)
```