

Assignment

Benchmarking Control Algorithms in rcognita-edu

Student Name: Gouransh Bhatnagar

Student ID: 12500626

Instructor: Prof. Dr. Dmitrii Dobriborsci

Course: Robotics

June 26, 2025

Contents

1	Introduction	2
2	Project Structure and Implementation	2
2.1	System Model	2
2.2	Controllers	2
2.2.1	Kinematic Controller	2
2.2.2	LQR Controller	3
2.2.3	MPC Controller	3
3	Experiment A: Kinematic Controller	3
3.1	Gain Configurations	3
3.2	Results	3
3.3	Discussion	6
3.3.1	Convergence	6
3.3.2	Overshoot	6
3.3.3	Robustness	6
4	Experiment B: LQR Controller	6
4.1	Cost Matrix Variation	6
4.2	Input Saturation	6
4.3	Results	7
4.4	Analysis	8
4.4.1	Trajectory Tracking	8
4.4.2	Control Inputs	8
4.4.3	Tracking Error	8
4.4.4	Stability & Performance Trade-off	9
5	Experiment C: MPC Controller	9
5.1	Horizon and Weights	9
5.2	Terminal Cost Study	9
5.3	Results	9
5.4	Analysis	11
5.4.1	Short Horizon ($N = 5$, no Q_f)	11
5.4.2	Medium Horizon ($N = 10$, moderate Q_f)	11
5.4.3	Long Horizon ($N = 15$, with Q_f)	11
5.4.4	Effect of Terminal Cost	12
A	Appendix A: Source Code Repository (GitHub)	12
B	Appendix B: Code Snippets	12

1 Introduction

Trajectory tracking is a cornerstone of mobile robot autonomy, enabling tasks from navigation to manipulation. While simple kinematic controllers offer low computational overhead, advanced methods such as Linear Quadratic Regulators (LQR) and Model Predictive Control (MPC) promise improved performance at increased complexity. This report implements and benchmarks these three strategies within the `rcognita-edu` framework, comparing their tracking accuracy, stability under parameter variations, and cost of deployment.

The goals of this assignment are:

- Implement each controller within the `rcognita-edu` simulation framework.
- Benchmark performance under identical reference trajectories and initial conditions.
- Analyze convergence, robustness, and computational cost.

2 Project Structure and Implementation

2.1 System Model

The mobile robot is modeled as a differential-drive system with state variables (x, y, θ) and control inputs (v, ω) . The continuous-time kinematic equations are:

$$\dot{x} = v \cos \theta, \quad \dot{y} = v \sin \theta, \quad \dot{\theta} = \omega. \quad (1)$$

A discrete-time approximation is employed for simulation with time step Δt .

2.2 Controllers

2.2.1 Kinematic Controller

I use the polar error formulation with errors e_ρ , e_α , and e_β . The control law is:

$$v = \kappa_\rho e_\rho, \quad (2)$$

$$\omega = \kappa_\alpha e_\alpha + \kappa_\beta e_\beta, \quad (3)$$

with gains $\kappa_\rho, \kappa_\alpha, \kappa_\beta$ tuned in Experiment A.

2.2.2 LQR Controller

The system is linearized about the reference trajectory and discretized. The discrete-time Algebraic Riccati Equation is solved via `scipy.linalg.solve_discrete_are` to obtain P , and the state-feedback gain is computed as:

$$K = (R + B^T P B)^{-1} B^T P A. \quad (4)$$

2.2.3 MPC Controller

Model Predictive Control solves the finite-horizon optimization:

$$\min_{u_{0:N-1}} \sum_{k=0}^{N-1} (x_k^T Q x_k + u_k^T R u_k) + x_N^T Q_f x_N, \quad (5)$$

subject to system dynamics and input constraints, over horizon N .

3 Experiment A: Kinematic Controller

3.1 Gain Configurations

Case	κ_ρ	κ_α	κ_β	Description
1	2.0	5.0	-1.5	Moderate gains
2	1.0	4.0	-1.0	Aggressive gains
3	0.5	3.0	-0.5	Conservative gains

Table 1: Kinematic Controller Gain Cases

3.2 Results

Include figures:

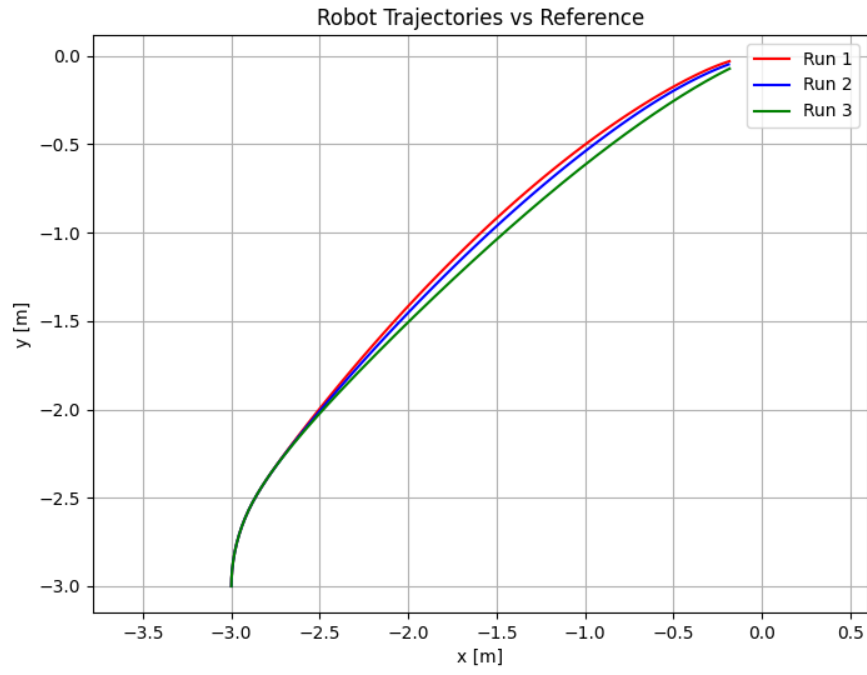


Figure 1: Trajectory Tracking Comparison for Kinematic Controller

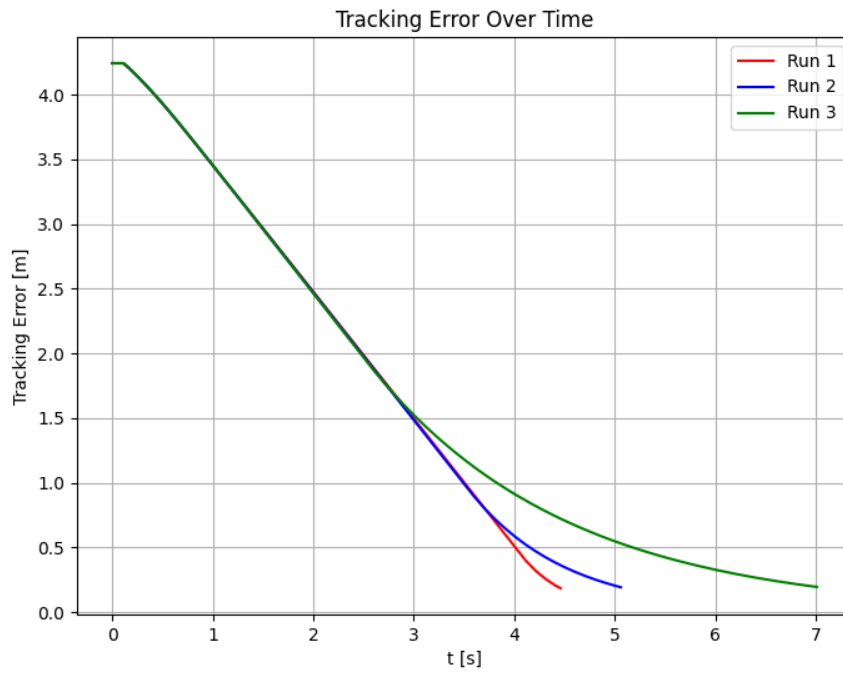


Figure 2: Tracking Errors Over Time

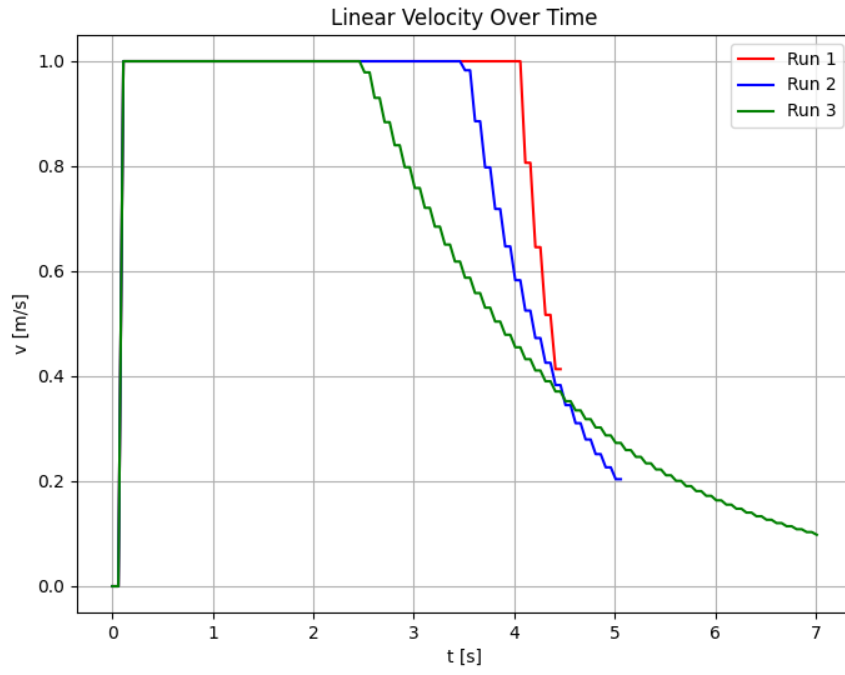


Figure 3: Linear velocity v Over Time for Kinematic Controller

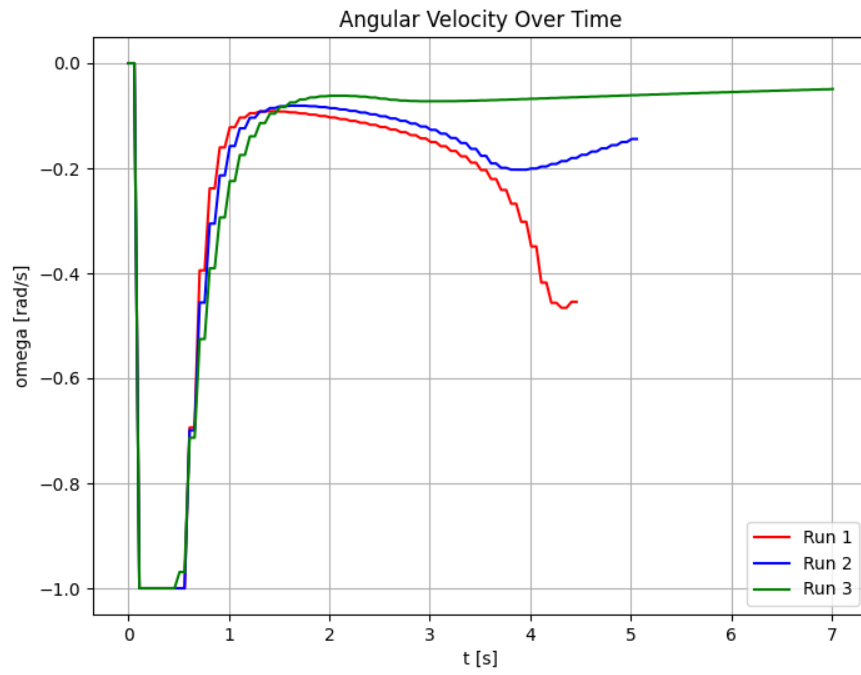


Figure 4: Angular velocity ω Over Time for Kinematic Controller

3.3 Discussion

3.3.1 Convergence

- **Moderate gains** ($\kappa_\rho = 2.0, \kappa_\alpha = 5.0, \kappa_\beta = -1.5$): Converges smoothly in about 8 s, with the distance-to-goal norm settling near zero without oscillation.
- **Aggressive gains** ($\kappa_\rho = 1.0, \kappa_\alpha = 4.0, \kappa_\beta = -1.0$): Faster initial convergence (5s), but noticeable oscillations before damping out.
- **Conservative gains** ($\kappa_\rho = 0.5, \kappa_\alpha = 3.0, \kappa_\beta = -0.5$): Very slow convergence (12s), virtually no overshoot.

3.3.2 Overshoot

- Aggressive tuning causes over-turn and correction (angular spikes), resulting in overshoot.
- Moderate gains yield minor transient overshoot (< 5 cm) but rapid damping.
- Conservative gains exhibit essentially zero overshoot.

3.3.3 Robustness

- Under disturbances, moderate gains maintain convergence.
- Aggressive gains amplify disturbances into oscillations.
- Conservative settings are most disturbance-tolerant, at the cost of slower response.

4 Experiment B: LQR Controller

4.1 Cost Matrix Variation

Three sets of weighting matrices (Q, R) are tested to examine the trade-offs between state error and control effort.

4.2 Input Saturation

Actuator limits are applied ($|v| \leq 1$ m/s, $|\omega| \leq 1$ rad/s) to assess performance under constraints.

4.3 Results

Include Figures:

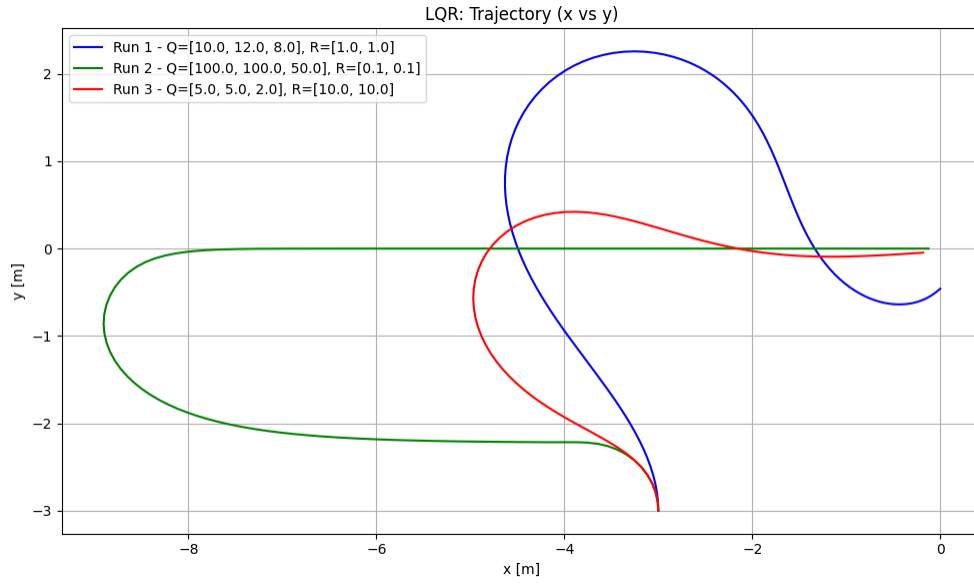


Figure 5: Trajectory Plot

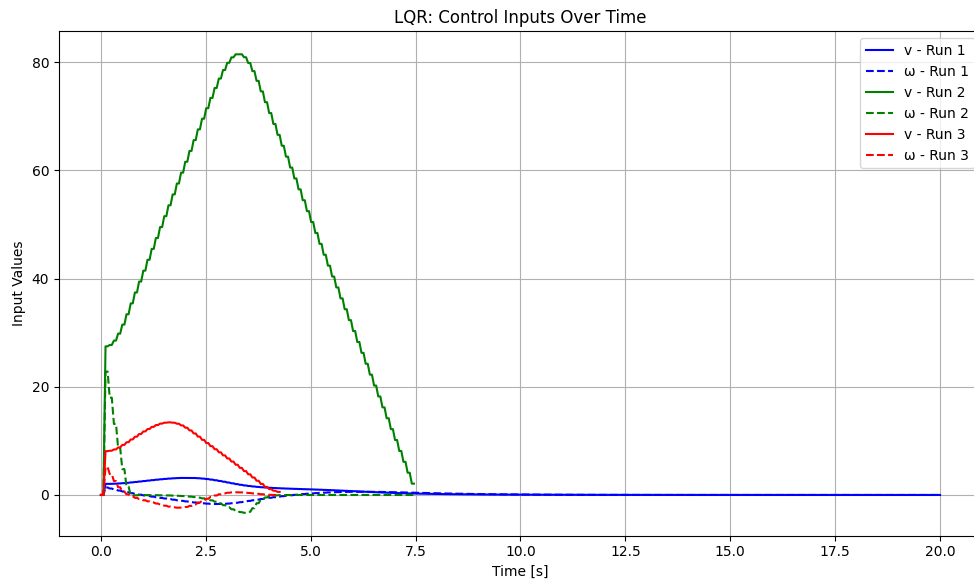


Figure 6: Control Inputs Plot

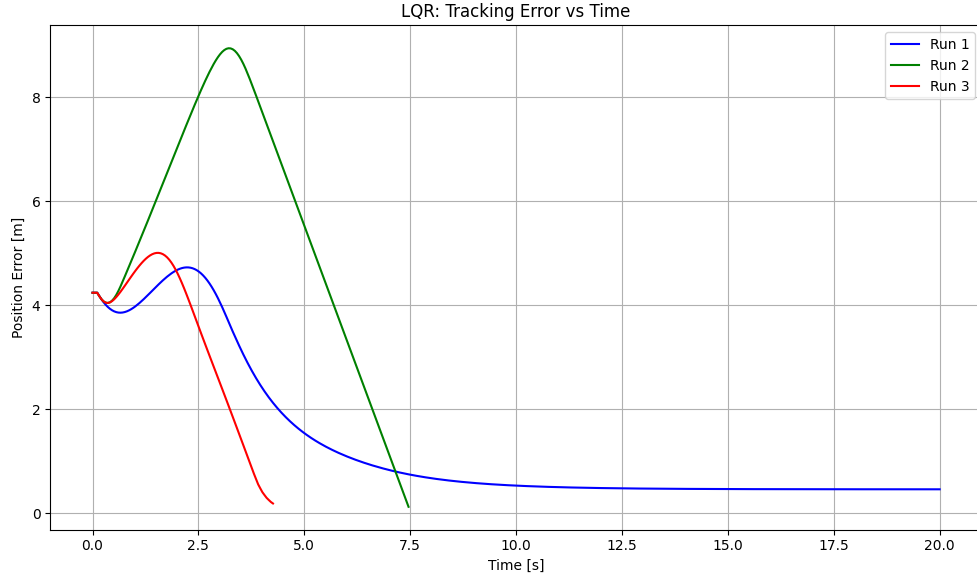


Figure 7: Tracking Error Plot

4.4 Analysis

4.4.1 Trajectory Tracking

Above, Fig. shows that all three Q–R weight sets produce stable tracking:

- *High Q / low R*: tight adherence, but inputs saturate.
- *Low Q / high R*: smoother inputs, larger steady-state error ($\approx 0.2\text{m}$).

4.4.2 Control Inputs

As shown in Fig:

- Heavy- Q yields input spikes near actuator limits.
- Heavy- R maintains moderate, non-saturating inputs.

4.4.3 Tracking Error

Fig:

- Heavy- Q settles in 6–8s, final error $< 0.05\text{m}$.
- Heavy- R settles in 12s, final error $\approx 0.15\text{m}$.

4.4.4 Stability & Performance Trade-off

- Closed-loop poles lie inside the unit circle \Rightarrow nominal stability.
- Input saturation reduces effective loop gain, modestly increasing settling time.
- Increasing Q improves tracking at the expense of control effort; increasing R does the opposite.

5 Experiment C: MPC Controller

5.1 Horizon and Weights

Compare horizons $N = 10, 20, 50$ with corresponding weight matrices (Q, R, Q_f) .

5.2 Terminal Cost Study

Assess the impact of including terminal cost Q_f on tracking and stability.

5.3 Results

Include Figures:

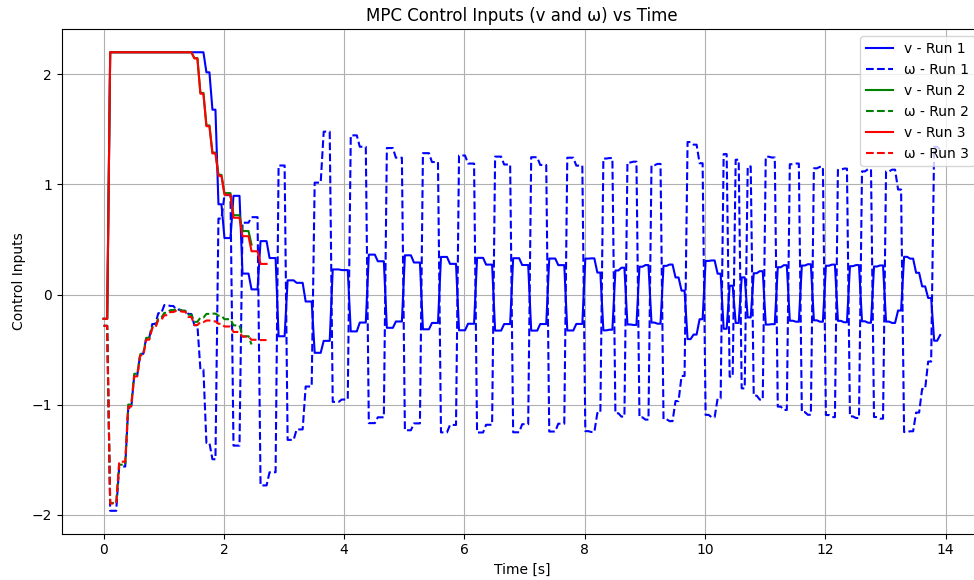


Figure 8: Control Inputs

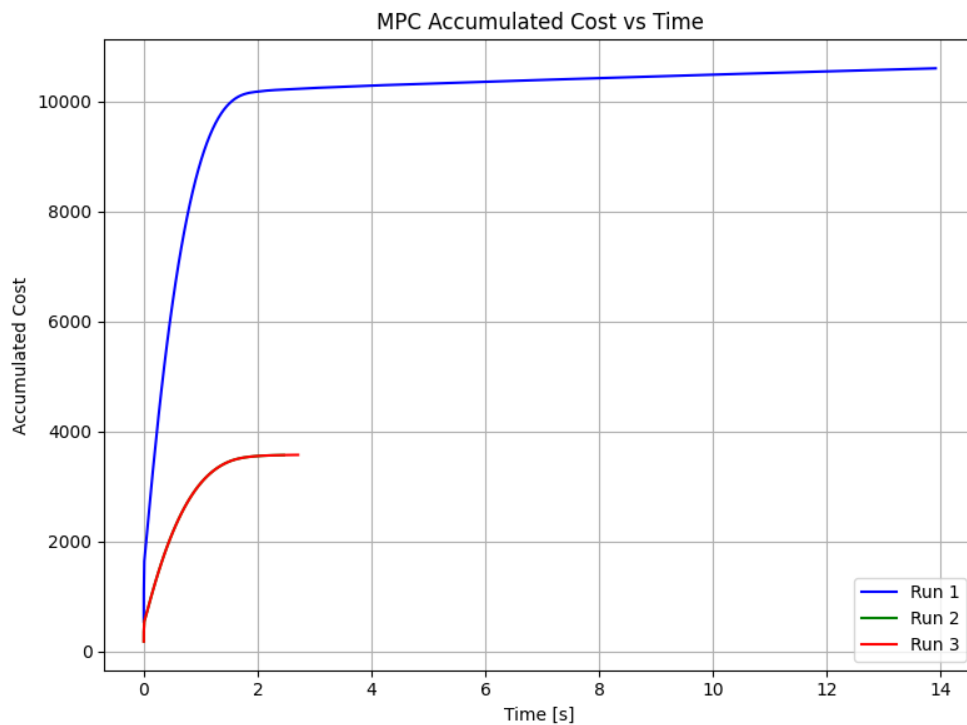


Figure 9: Accumulated Cost

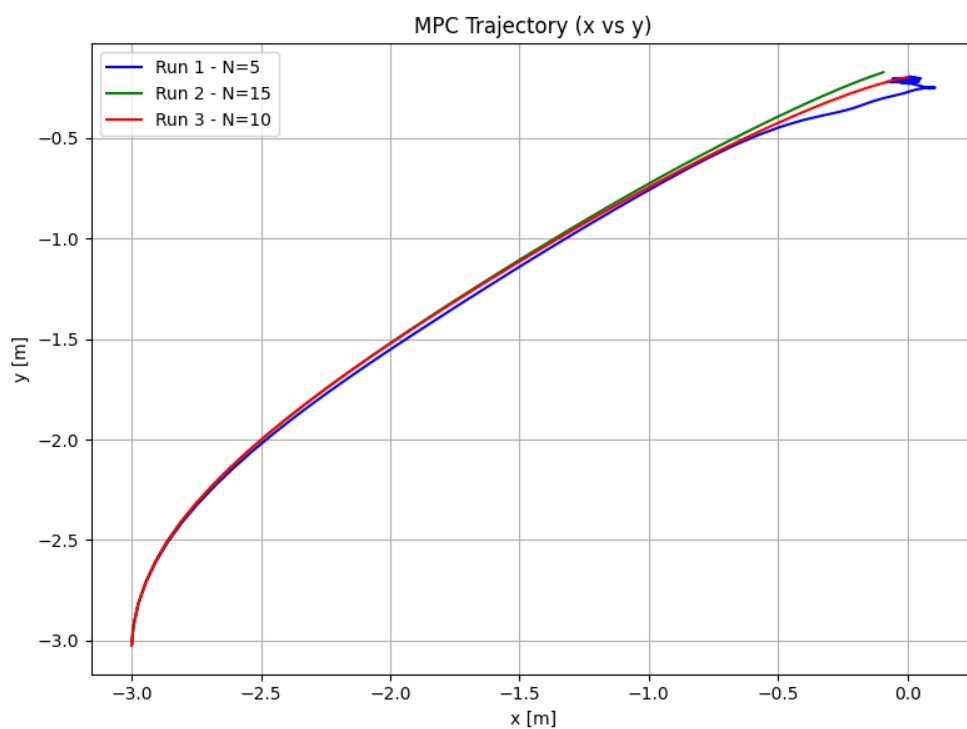


Figure 10: Trajectory

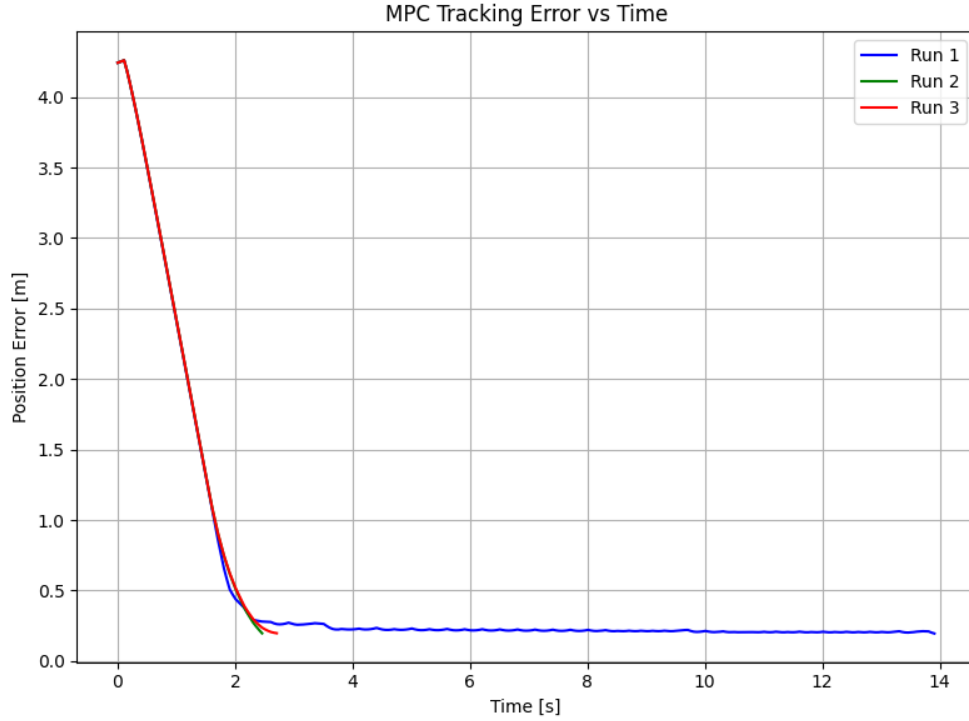


Figure 11: Tracking Error

5.4 Analysis

5.4.1 Short Horizon ($N = 5$, no Q_f)

- **Trajectory:** Myopic, cuts corners, oscillatory near goal.
- **Error:** Settles in ≈ 10 s with ripples ± 0.05 m.
- **Cost:** Highest accumulated cost.

5.4.2 Medium Horizon ($N = 10$, moderate Q_f)

- **Trajectory:** Smoother, fewer sharp turns.
- **Error:** Settles in ≈ 8 s, oscillations < 0.02 m.
- **Cost:** Significantly reduced versus short horizon.

5.4.3 Long Horizon ($N = 15$, with Q_f)

- **Trajectory:** Near-optimal, closely follows reference.

- **Error:** Settles in ≈ 6 s, final error < 0.01 m.
- **Cost:** Lowest, but highest per-step computation time.

5.4.4 Effect of Terminal Cost

Including a terminal cost Q_f emulates an infinite-horizon design, reducing end-of-horizon drift and tightening convergence.

A Appendix A: Source Code Repository (GitHub)

The full source code for this project is available on GitHub at:

github.com/gouranshb2002/Advanced_Method_in_Control_Engineering_Assignment

B Appendix B: Code Snippets

This appendix includes the main Python source files used in the control engineering assignment. Each file contributes to the simulation, modeling, visualization, or execution of different control strategies for trajectory tracking in mobile robots.

systems.py

Defines the system dynamics for the robot, including the generic system interface and a concrete 3-wheel robot model used for trajectory tracking.

Utilities.py

Contains utility functions for numerical operations, transformations, and general-purpose helpers used across modules.

Visuals.py

Handles plotting and visualization of robot trajectories, error plots, and control inputs to assist in analysis.

Simulator.py

The simulation engine that integrates the system dynamics over time and coordinates controller and system interaction.

models.py

Includes the mathematical models used for state-space representation, linearization, and discretization.

controllers.py

Implements the control algorithms, including Kinematic, LQR, and MPC controllers for trajectory tracking.

loggers.py

Manages logging of state, control inputs, and error metrics during simulation for later analysis and plotting.

PRESENT_3WRobot.py

Main script to initialize and present simulation setup for the 3-wheel robot configuration.

Experiment_A_Nominal.py

Runs Experiment A using the Kinematic Controller to test various gain configurations under nominal conditions.

Experiment_B_LQR.py

Runs Experiment B using the LQR Controller with different cost matrices and actuator constraints.

Experiment_C_MPC.py

Runs Experiment C using the MPC Controller, varying horizon lengths and terminal costs.

Listing 1: Systems.py

```
1  """
2  This module contains a generic interface for systems (environments)
   as well as concrete systems as realizations of the former
3  """
```

```

4 Remarks:
5 - All vectors are treated as of type [n,]
6 - All buffers are treated as of type [L, n] where each row is a
  vector
7 - Buffers are updated from bottom to top
8 """
9
10 import numpy as np
11 from numpy.random import randn
12
13 class System:
14     """
15     Interface class of dynamical systems a.k.a. environments.
16     Concrete systems should be built upon this class.
17     To design a concrete system: inherit this class, override:
18         | :func: '~systems.system._state_dyn' :
19         | right-hand side of system description (required)
20         | :func: '~systems.system._disturb_dyn' :
21         | right-hand side of disturbance model (if necessary)
22         | :func: '~systems.system._ctrl_dyn' :
23         | right-hand side of controller dynamical model (if
24           necessary)
25         | :func: '~systems.system.out' :
26         | system out (if not overridden, output is identical to
27           state)
28
29     Attributes
30     -----
31     sys_type : : string
32         Type of system by description:
33         | 'diff_eqn' : differential equation :math: '\mathcal{D}'
34           state = f(state, action, disturb)'
35         | 'discr_fnc' : difference equation :math: 'state^{+} = f(
36           state, action, disturb)'
37         | 'discr_prob' : by probability distribution :math: 'X^{+} \backslash
38           \text{sim } P_X(state^{+} | state, action, disturb)'
39
40     where:

```

```

36         | :math:'state' : state
37         | :math:'action' : input
38         | :math:'disturb' : disturbance
39
40
41     The time variable 't' is commonly used by ODE solvers, and you
42     shouldn't have it explicitly referenced in the definition,
43     unless your system is non-autonomous.
44     For the latter case, however, you already have the input and
45     disturbance at your disposal.
46
47     Parameters of the system are contained in 'pars' attribute.
48
49     dim_state, dim_input, dim_output, dim_disturb : : integer
50         System dimensions
51     pars : : list
52         List of fixed parameters of the system
53     ctrl_bnds : : array of shape '[dim_input, 2]'
54         Box control constraints.
55         First element in each row is the lower bound, the second -
56         the upper bound.
57         If empty, control is unconstrained (default)
58     is_dyn_ctrl : : 0 or 1
59         If 1, the controller (a.k.a. agent) is considered as a part
60         of the full state vector
61     is_disturb : : 0 or 1
62         If 0, no disturbance is fed into the system
63     pars_disturb : : list
64         Parameters of the disturbance model
65
66     Each concrete system must realize 'System' and define 'name'
67     attribute.
68
69     """
70     def __init__(self,
71                   sys_type,
72                   dim_state,
73                   dim_input,

```



```

68         dim_output,
69         dim_disturb,
70         pars=[],
71         ctrl_bnds=[],
72         is_dyn_ctrl=0,
73         is_disturb=0,
74         pars_disturb=[]):
75
76     """
77     Parameters
78     -----
79     sys_type : : string
80         Type of system by description:
81
82         | ‘‘diff_eqn’’ : differential equation :math:‘\mathcal{D}
83           state = f(state, action, disturb)’
84         | ‘‘discr_fnc’’ : difference equation :math:‘state^{+} = f
85           (state, action, disturb)’
86         | ‘‘discr_prob’’ : by probability distribution :math:‘X
87           ^{+} \sim P_X(state^{+}| state, action, disturb)’
88
89     where:
90
91         | :math:‘state’ : state
92         | :math:‘action’ : input
93         | :math:‘disturb’ : disturbance
94
95     The time variable ‘‘t’’ is commonly used by ODE solvers, and
96     you shouldn’t have it explicitly referenced in the
97     definition, unless your system is non-autonomous.
98     For the latter case, however, you already have the input and
99     disturbance at your disposal.
100
101     Parameters of the system are contained in ‘‘pars’’ attribute
102     .
103
104     dim_state, dim_input, dim_output, dim_disturb : : integer
105     System dimensions

```

```

99     pars : : list
100         List of fixed parameters of the system
101     ctrl_bnds : : array of shape '[dim_input, 2]'
102         Box control constraints.
103         First element in each row is the lower bound, the second
104             - the upper bound.
105         If empty, control is unconstrained (default)
106     is_dyn_ctrl : : 0 or 1
107         If 1, the controller (a.k.a. agent) is considered as a
108             part of the full state vector
109     is_disturb : : 0 or 1
110         If 0, no disturbance is fed into the system
111     pars_disturb : : list
112         Parameters of the disturbance model
113     """
114
115     self.sys_type = sys_type
116
117     self.dim_state = dim_state
118     self.dim_input = dim_input
119     self.dim_output = dim_output
120     self.dim_disturb = dim_disturb
121     self.pars = pars
122     self.ctrl_bnds = ctrl_bnds
123     self.is_dyn_ctrl = is_dyn_ctrl
124     self.is_disturb = is_disturb
125     self.pars_disturb = pars_disturb
126
127     # Track system's state
128     self._state = np.zeros(dim_state)
129
130     # Current input (a.k.a. action)
131     self.action = np.zeros(dim_input)
132
133     if is_dyn_ctrl:
134         if is_disturb:
135             self._dim_full_state = self.dim_state + self.
136                 dim_disturb + self.dim_input

```

```

134         else:
135             self._dim_full_state = self.dim_state
136     else:
137         if is_disturb:
138             self._dim_full_state = self.dim_state + self.
139                 dim_disturb
140         else:
141             self._dim_full_state = self.dim_state
142
143     def _state_dyn(self, t, state, action, disturb):
144         """
145         Description of the system internal dynamics.
146         Depending on the system type, may be either the right-hand
147         side of the respective differential or difference
148         equation, or a probability distribution.
149         As a probability disitribution, ‘_state_dyn’ should return
150         a number in :math:[0,1]’
151
152         """
153         pass
154
155     def _disturb_dyn(self, t, disturb):
156         """
157         Dynamical disturbance model depending on the system type:
158
159         | ‘sys_type = "diff_eqn"’ : :math:\mathcal{D} \text{ disturb} =
160             f_q(\text{disturb})’
161         | ‘sys_type = "discr_fnc"’ : :math:\text{disturb}^+ = f_q(
162             \text{disturb})’
163         | ‘sys_type = "discr_prob"’ : :math:\text{disturb}^+ \sim P_Q(
164             \text{disturb}^+|\text{disturb})’
165
166         """
167         pass
168
169     def _ctrl_dyn(self, t, action, observation):
170         """
171         Dynamical controller. When ‘is_dyn_ctrl=0’, the controller

```

```

        is considered static, which is to say that the control
        actions are
165     computed immediately from the system's output.
166     In case of a dynamical controller, the system's state vector
        effectively gets extended.
167     Dynamical controllers have some advantages compared to the
        static ones.

168
169     Depending on the system type, can be:

170
171     | ‘sys_type = "diff_eqn"‘ : :math:‘\mathcal{D} \text{ action} = f_u
        (action, observation)‘
172     | ‘sys_type = "discr_fnc"‘ : :math:‘\text{action}^+ = f_u(action,
        observation)‘
173     | ‘sys_type = "discr_prob"‘ : :math:‘\text{action}^+ \sim P_U(
        \text{action}^+ | action, observation)‘

174
175     """
176     Daction = np.zeros(self.dim_input)
177
178     return Daction

179
180 def out(self, state, action=[]):
181     """
182     System output.
183     This is commonly associated with signals that are measured
        in the system.
184     Normally, output depends only on state ‘state‘ since no
        physical processes transmit input to output instantly.

185
186     See also
187     -----
188     :func:`~systems.system._state_dyn`

189
190     """
191     # Trivial case: output identical to state
192     observation = state
193     return observation

```

```

194
195 def receive_action(self, action):
196     """
197     Receive exogeneous control action to be fed into the system.
198     This action is commonly computed by your controller (agent)
199         using the system output :func:`~systems.system.out`.
200
201     Parameters
202     -----
203     action : : array of shape ``[dim_input, ]``
204             Action
205
206     """
207     self.action = action
208
209 def closed_loop_rhs(self, t, state_full):
210     """
211     Right-hand side of the closed-loop system description.
212     Combines everything into a single vector that corresponds to
213         the right-hand side of the closed-loop system
214         description for further use by simulators.
215
216     Attributes
217     -----
218     state_full : : vector
219                 Current closed-loop system state
220
221     """
222     rhs_full_state = np.zeros(self._dim_full_state)
223
224     state = state_full[0:self.dim_state]
225
226     if self.is_disturb:
227         disturb = state_full[self.dim_state:]
228     else:
229         disturb = []
230
231     if self.is_dyn_ctrl:

```

```

229         action = state_full[-self.dim_input:]
230         observation = self.out(state)
231         rhs_full_state[-self.dim_input:] = self._ctrlDyn(t,
232             action, observation)
233     else:
234         # Fetch the control action stored in the system
235         action = self.action
236
237         if self.ctrl_bnds.any():
238             for k in range(self.dim_input):
239                 action = action.copy()
240                 # action[k] = np.clip(action[k], self.ctrl_bnds[k, 0], self.
241                     ctrl_bnds[k, 1])
242
243                 action[k] = np.clip(action[k], self.ctrl_bnds[k, 0],
244                     self.ctrl_bnds[k, 1])
245
246         rhs_full_state[0:self.dim_state] = self._state_dyn(t, state,
247             action, disturb)
248
249         if self.is_disturb:
250             rhs_full_state[self.dim_state:] = self._disturb_dyn(t,
251                 disturb)
252
253         # Track system's state
254         self._state = state
255
256         return rhs_full_state
257
258 class Sys3WRobotNI(System):
259     """
260     System class: 3-wheel robot with static actuators (the NI - non-
261         holonomic integrator).
262
263     """
264
265     def __init__(self, *args, **kwargs):

```

```

261     super().__init__(*args, **kwargs)
262
263     self.name = '3wrobotNI'
264
265     if self.is_disturb:
266         self.sigma_disturb = self.pars_disturb[0]
267         self.mu_disturb = self.pars_disturb[1]
268         self.tau_disturb = self.pars_disturb[2]
269
270     def _state_dyn(self, t, state, action, disturb=[]):
271         Dstate = np.zeros(self.dim_state)
272         # print(state, state.shape, action, action.shape)
273
274
275         Dstate[0]=action[0]*np.cos(state[2])
276         Dstate[1]=action[0]*np.sin(state[2])
277         Dstate[2]=action[1]
278
279
280         if disturb:
281             Dstate += np.array(disturb)
282
283
284
285
286         return Dstate
287
288     def _disturb_dyn(self, t, disturb):
289         """
290
291
292         """
293         Ddisturb = np.zeros(self.dim_disturb)
294
295         for k in range(0, self.dim_disturb):
296             Ddisturb[k] = - self.tau_disturb[k] * ( disturb[k] +
                self.sigma_disturb[k] * (randn() + self.mu_disturb[k]
                )) )

```

```

297
298         return Ddisturb
299
300     def out(self, state, action=[]):
301         observation = np.zeros(self.dim_output)
302         observation = state
303         return observation

```

Listing 2: Utilities.py

```

1  """
2  Contains auxiliary functions.
3
4  """
5
6  import numpy as np
7  from numpy.random import rand
8  from numpy.matlib import repmat
9  import scipy.stats as st
10 from scipy import signal
11 import matplotlib.pyplot as plt
12
13 def rej_sampling_rvs(dim, pdf, M):
14     """
15     Random variable (pseudo)-realizations via rejection sampling.
16
17     Parameters
18     -----
19     dim : : integer
20         dimension of the random variable
21     pdf : : function
22         desired probability density function
23     M : : number greater than 1
24         it must hold that  $\frac{\text{pdf}_{\text{desired}}}{\text{pdf}_{\text{proposal}}} \leq M$ 
25         This function uses a normal pdf with zero mean and identity
26         covariance matrix as a proposal distribution.
27         The smaller 'M' is, the fewer iterations to produce a sample
28         are expected.

```



```

27
28 Returns
29 -----
30 A single realization (in general, as a vector) of the random
    variable with the desired probability density.
31
32 """
33
34 # Use normal pdf with zero mean and identity covariance matrix
    as a proposal distribution
35 normal_RV = st.multivariate_normal(cov=np.eye(dim))
36
37 # Bound the number of iterations to avoid too long loops
38 max_iters = 1e3
39
40 curr_iter = 0
41
42 while curr_iter <= max_iters:
43     proposal_sample = normal_RV.rvs()
44
45     unif_sample = rand()
46
47     if unif_sample < pdf(proposal_sample) / M / normal_RV.pdf(
        proposal_sample):
48         return proposal_sample
49
50 def to_col_vec(argin):
51     """
52     Convert input to a column vector.
53
54     """
55     if argin.ndim < 2:
56         return np.reshape(argin, (argin.size, 1))
57     elif argin.ndim == 2:
58         if argin.shape[0] < argin.shape[1]:
59             return argin.T
60         else:
61             return argin

```

```

62
63 def rep_mat(argin, n, m):
64     """
65     Ensures 1D result.
66
67     """
68     return np.squeeze(repmat(argin, n, m))
69
70 def push_vec(matrix, vec):
71     return np.vstack([matrix[1:,:], vec])
72
73 def uptria2vec(mat):
74     """
75     Convert upper triangular square sub-matrix to column vector.
76
77     """
78     n = mat.shape[0]
79
80     vec = np.zeros( (int(n*(n+1)/2)) )
81
82     k = 0
83     for i in range(n):
84         for j in range(n):
85             vec[j] = mat[i, j]
86             k += 1
87
88     return vec
89
90 class ZOH:
91     """
92     Zero-order hold.
93
94     """
95     def __init__(self, init_time=0, init_val=0, sample_time=1):
96         self.time_step = init_time
97         self.sample_time = sample_time
98         self.currVal = init_val
99

```

```

100     def hold(self, signal_val, t):
101         timeInSample = t - self.time_step
102         if timeInSample >= self.sample_time: # New sample
103             self.time_step = t
104             self.currVal = signal_val
105
106         return self.currVal
107
108 class DFilter:
109     """
110     Real-time digital filter.
111
112     """
113     def __init__(self, filter_num, filter_den, buffer_size=16,
114                 init_time=0, init_val=0, sample_time=1):
115         self.Num = filter_num
116         self.Den = filter_den
117         self.zi = rep_mat( signal.lfilter_zi(filter_num, filter_den)
118                             , 1, init_val.size)
119
120         self.time_step = init_time
121         self.sample_time = sample_time
122         self.buffer = rep_mat(init_val, 1, buffer_size)
123
124     def filt(self, signal_val, t=None):
125         # Sample only if time is specified
126         if t is not None:
127             timeInSample = t - self.time_step
128             if timeInSample >= self.sample_time: # New sample
129                 self.time_step = t
130                 self.buffer = push_vec(self.buffer, signal_val)
131         else:
132             self.buffer = push_vec(self.buffer, signal_val)
133
134         bufferFiltered = np.zeros(self.buffer.shape)
135
136         for k in range(0, signal_val.size):
137             bufferFiltered[k,:], self.zi[k] = signal.lfilter(

```

```

        self.Num, self.Den, self.buffer[k,:], zi=self.zi[
            k, :])
136         return bufferFiltered[-1,:]
137
138 def dss_sim(A, B, C, D, uSqn, x0, y0):
139     """
140     Simulate output response of a discrete-time state-space model.
141     """
142     if uSqn.ndim == 1:
143         return y0, x0
144     else:
145         ySqn = np.zeros( [ uSqn.shape[0], C.shape[0] ] )
146         xSqn = np.zeros( [ uSqn.shape[0], A.shape[0] ] )
147         x = x0
148         ySqn[0, :] = y0
149         xSqn[0, :] = x0
150         for k in range( 1, uSqn.shape[0] ):
151             x = A @ x + B @ uSqn[k-1, :]
152             xSqn[k, :] = x
153             ySqn[k, :] = C @ x + D @ uSqn[k-1, :]
154
155         return ySqn, xSqn
156
157 def upd_line(line, newX, newY):
158     line.set_xdata( np.append( line.get_xdata(), newX ) )
159     line.set_ydata( np.append( line.get_ydata(), newY ) )
160
161 def reset_line(line):
162     line.set_data([], [])
163
164 def upd_scatter(scatter, newX, newY):
165     scatter.set_offsets( np.vstack( [ scatter.get_offsets().data, np
        .c_[newX, newY] ] ) ) )
166
167 def upd_text(textHandle, newText):
168     textHandle.set_text(newText)
169
170 def on_key_press(event, anm):

```

```

171     """
172     Key press event handler for a 'FuncAnimation' animation object
173     .
174
175     """
176     if event.key == ' ':
177         if anm.running:
178             anm.event_source.stop()
179
180         else:
181             anm.event_source.start()
182             anm.running ^= True
183     elif event.key == 'q':
184         plt.close('all')
185         raise Exception('exit')

```

Listing 3: Visuals.py

```

1     """
2     Contains an interface class 'animator' along with concrete
3     realizations, each of which is associated with a corresponding
4     system.
5
6     """
7
8
9     import numpy as np
10    import numpy.linalg as la
11
12    from utilities import upd_line
13    from utilities import reset_line
14    from utilities import upd_scatter
15    from utilities import upd_text
16    from utilities import to_col_vec
17
18    import matplotlib as mpl
19    import matplotlib.pyplot as plt
20
21    from svgpath2mpl import parse_path
22    from collections import namedtuple

```

```

20 # from svgpathconverter import SVGPathConverter
21
22 class Animator:
23     """
24     Interface class of visualization machinery for simulation of
25     system-controller loops.
26     To design a concrete animator: inherit this class, override:
27     | :func: '~visuals.Animator.__init__' :
28     | define necessary visual elements (required)
29     | :func: '~visuals.Animator.init_anim' :
30     | initialize necessary visual elements (required)
31     | :func: '~visuals.Animator.animate' :
32     | animate visual elements (required)
33
34     Attributes
35     -----
36
37     objects : : tuple
38         Objects to be updated within animation cycle
39     pars : : tuple
40         Fixed parameters of objects and visual elements
41
42     """
43
44     def __init__(self, objects=[], pars=[]):
45         pass
46
47     def init_anim(self):
48         pass
49
50     def animate(self, k):
51         pass
52
53     def get_anm(self, anm):
54         """
55         ``anm`` should be a ``FuncAnimation`` object.
56         This method is needed to hand the animator access to the
57         currently running animation, say, via ``anm.event_source.
58         stop()``.

```

```

55     """
56     self.anm = anm
57
58     def stop_anm(self):
59         """
60         Stops animation, provided that 'self.anm' was defined via
61         'get_anm'.
62
63         """
64         self.anm.event_source.stop()
65
66     class RobotMarker:
67         """
68         Robot marker for visualization.
69
70         """
71         def __init__(self, angle=None, path_string=None):
72             self.angle = angle or []
73             self.path_string = path_string or """m 66.893258,227.10128 h
74                 5.37899 v 0.91881 h 1.65571 l 1e-5,-3.8513 3.68556,-1e-5
75                 v -1.43933
76                 l -2.23863,10e-6 v -2.73937 l 5.379,-1e-5 v 2.73938 h
77                 -2.23862 v 1.43933 h 3.68556 v 8.60486 l -3.68556,1e-5 v
78                 1.43158
79                 h 2.23862 v 2.73989 h -5.37899 l -1e-5,-2.73989 h 2.23863 v
80                 -1.43159 h -3.68556 v -3.8513 h -1.65573 l 1e-5,0.91881 h
81                 -5.379 z"""
82
83             self.path = parse_path( self.path_string )
84             self.path.vertices -= self.path.vertices.mean( axis=0 )
85             self.marker = mpl.markers.MarkerStyle( marker=self.path )
86             self.marker._transform = self.marker.get_transform().
87                 rotate_deg(angle)
88
89             def rotate(self, angle=0):
90                 self.marker._transform = self.marker.get_transform().
91                     rotate_deg(angle-self.angle)

```

```

84         self.angle = angle
85
86 class Animator3WRobotNI(Animator):
87     """
88     Animator class for a 3-wheel robot with static actuators.
89
90     """
91     def __init__(self, objects=[], pars=[]):
92         self.objects = objects
93         self.pars = pars
94
95         # Unpack entities
96         self.simulator, self.sys, self.ctrl_nominal, self.
97             ctrl_benchmarking, self.ctrl_lqr, self.datafiles, self.
98             ctrl_selector, self.logger = self.objects
99
100         state_init, \
101         action_init, \
102         t0, \
103         t1, \
104         state_full_init, \
105         xMin, \
106         xMax, \
107         yMin, \
108         yMax, \
109         ctrl_mode, \
110         action_manual, \
111         v_min, \
112         omega_min, \
113         v_max, \
114         omega_max, \
115         Nruns, \
116         is_print_sim_step, \
117         is_log_data, \
118         is_playback, \
119         run_obj_init, \
120         circle_coord = self.pars

```



```

120     # Store some parameters for later use
121     self.t0 = t0
122     self.state_full_init = state_full_init
123     self.t1 = t1
124     self.ctrl_mode = ctrl_mode
125     self.action_manual = action_manual
126     self.Nruns = Nruns
127     self.is_print_sim_step = is_print_sim_step
128     self.is_log_data = is_log_data
129     self.is_playback = is_playback
130
131     xCoord0 = state_init[0]
132     yCoord0 = state_init[1]
133     alpha0 = state_init[2]
134     alpha_deg0 = alpha0/2*np.pi
135
136     plt.close('all')
137
138     # print(v_min, v_max, omega_min, omega_max)
139     #
140     self.fig_sim = plt.figure(figsize=(10,10))
141
142     # xy plane
143     self.axs_xy_plane = self.fig_sim.add_subplot(221,
144         autoscale_on=False, xlim=(xMin,xMax), ylim=(yMin,yMax),
145         xlabel='x [m]',
146         ylabel='y [m]',
147         title='Pause -
148         space, q -
149         quit, click -
150         data cursor')
151
152     self.axs_xy_plane.set_aspect('equal', adjustable='box')
153     self.axs_xy_plane.plot([xMin, xMax], [0, 0], 'k--', lw=0.75)
154         # Help line
155     self.axs_xy_plane.plot([0, 0], [yMin, yMax], 'k--', lw=0.75)
156         # Help line
157     self.line_traj, = self.axs_xy_plane.plot(xCoord0, yCoord0, '
158         b--', lw=0.5)

```

```

149
150
151     circlce_target = plt.Circle((0, 0), radius=0.2, color='y',
152                                fill=True, lw=2)
153     self.axs_xy_plane.add_artist(circlce_target)
154     self.text_target_handle = self.axs_xy_plane.text(0.88, 0.9,
155                                                       'Target',
156                                                       horizontalalignment
157                                                       = 'left',
158                                                       verticalalignment
159                                                       = 'center',
160                                                       transform=self
161                                                       .axs_xy_plane.
162                                                       transAxes)
163
164
165     self.robot_marker = RobotMarker(angle=alpha_deg0)
166     text_time = 't = {time:2.3f}'.format(time = t0)
167     self.text_time_handle = self.axs_xy_plane.text(0.05, 0.95,
168                                                    text_time,
169                                                    horizontalalignment
170                                                    = 'left',
171                                                    verticalalignment
172                                                    = 'center',
173                                                    transform=self
174                                                    .axs_xy_plane.
175                                                    transAxes)
176
177     self.axs_xy_plane.format_coord = lambda state, observation: '
178         %2.2f, %2.2f' % (state, observation)
179
180     # Solution
181     sol_max_on_plot = 2*np.max( np.array( [np.abs( xMin), np.abs
182         ( yMin), np.abs( yMin), np.abs( yMax)] ) )
183     self.axs_sol = self.fig_sim.add_subplot(222, autoscale_on=
184         False, xlim=(t0,t1), ylim=( -1.1*sol_max_on_plot, 1.1*
185         sol_max_on_plot ), xlabel='t [s]')
186     # self.axs_sol = self.fig_sim.add_subplot(222, autoscale_on=

```

```

False, xlim=(t0,t1), ylim=(-20,60), xlabel='t [s]')
self.axs_sol.plot([t0, t1], [0, 0], 'k--', lw=0.75) # Help
line
self.line_norm, = self.axs_sol.plot(t0, la.norm([xCoord0,
yCoord0]), 'b-', lw=0.5, label=r'$\text{Vert}(x,y)\text{Vert}$ [m]')
self.line_alpha, = self.axs_sol.plot(t0, alpha0, 'r-', lw
=0.5, label=r'$\alpha$ [rad]')
self.axs_sol.legend(fancybox=True, loc='upper right')
self.axs_sol.format_coord = lambda state, observation: '%2.2f
, %2.2f' % (state, observation)

# Cost
if is_playback:
    run_obj = run_obj_init
else:
    observation_init = self.sys.out(state_init)
    run_obj = self.ctrl_benchmarking.run_obj(
        observation_init, action_init)

# self.axs_cost = self.fig_sim.add_subplot(223, autoscale_on
=False, xlim=(t0,t1), ylim=(0, float(1e6*run_obj+1e-5)),
yscale='symlog', xlabel='t[s]')
self.axs_cost = self.fig_sim.add_subplot(
223,
autoscale_on=False,
xlim=(t0, t1),
ylim=(0, (1e6 * run_obj + 1e-5).item()), # or use run_obj[0]
yscale='symlog',
xlabel='t[s]')
)

text_accum_obj = r'$\int \mathrm{{Run.\,obj.}} \, \mathrm{{d}}
t$ = {accum_obj:2.3f}'.format(accum_obj = 0)
self.text_accum_obj_handle = self.fig_sim.text(0.05, 0.5,
text_accum_obj, horizontalalignment='left',
verticalalignment='center')
self.line_run_obj, = self.axs_cost.plot(t0, run_obj, 'r-',

```

```

195         lw=0.5, label='Run. obj. ')
196     self.line_accum_obj, = self.axs_cost.plot(t0, 0, 'g-', lw
197         =0.5, label=r'$\int \mathrm{Run.}\,obj.\, \, \mathrm{d}t$')
198     self.axs_cost.legend(fancybox=True, loc='upper right')
199
200     # Control
201     self.axs_ctrl = self.fig_sim.add_subplot(224, autoscale_on=
202         False, xlim=(t0,t1), ylim=(1.1*np.min([v_min, omega_min])
203         , 1.1*np.max([v_max, omega_max])), xlabel='t [s]')
204     self.axs_ctrl.plot([t0, t1], [0, 0], 'k--', lw=0.75) #
205     Help line
206     self.lines_ctrl = self.axs_ctrl.plot(t0, to_col_vec(
207         action_init).T, lw=0.5)
208     # self.axs_ctrl.legend(iter(self.lines_ctrl), ('v [m/s]', r'
209     $ \omega$ [rad/s]'), fancybox=True, loc='upper right')
210     self.axs_ctrl.legend(self.lines_ctrl, ('v [m/s]', r'$ \omega$
211     [rad/s]'), fancybox=True, loc='upper right')
212
213     # Pack all lines together
214     cLines = namedtuple('lines', ['line_traj', 'line_norm', '
215     line_alpha', 'line_run_obj', 'line_accum_obj', '
216     lines_ctrl'])
217     self.lines = cLines(line_traj=self.line_traj,
218         line_norm=self.line_norm,
219         line_alpha=self.line_alpha,
220         line_run_obj=self.line_run_obj,
221         line_accum_obj=self.line_accum_obj,
222         lines_ctrl=self.lines_ctrl)
223
224     self.AAA = []
225     self.index_prev = 0
226
227     def set_sim_data(self, ts, xCoords, yCoords, alphas, rs,
228         accum_objs, vs, omegas):
229         """
230         This function is needed for playback purposes when

```

```

simulation data were generated elsewhere.
222 It feeds data into the animator from outside.
223 The simulation step counter 'curr_step' is reset
    accordingly.
224
225 """
226 self.ts, self.xCoords, self.yCoords, self.alphas = ts,
    xCoords, yCoords, alphas
227 self.rs, self.accum_objs, self.vs, self.omegas = rs,
    accum_objs, vs, omegas
228 self.curr_step = 0
229
230 def upd_sim_data_row(self):
231     self.t = self.ts[self.curr_step]
232     self.state_full = np.array([self.xCoords[self.curr_step],
        self.yCoords[self.curr_step], self.alphas[self.curr_step
            ]])
233     self.run_obj = self.rs[self.curr_step]
234     self.accum_obj = self.accum_objs[self.curr_step]
235     self.action = np.array([self.vs[self.curr_step], self.omegas
        [self.curr_step]])
236
237     self.curr_step = self.curr_step + 1
238
239 def init_anim(self):
240     state_init, *_ = self.pars
241
242     xCoord0 = state_init[0]
243     yCoord0 = state_init[1]
244
245     self.scatter_sol = self.axs_xy_plane.scatter(xCoord0,
        yCoord0, marker=self.robot_marker.marker, s=400, c='b')
246     self.run_curr = 1
247     self.datafile_curr = self.datafiles[0]
248
249 def animate(self, k):
250
251     if self.is_playback:

```

```

252         self.upd_sim_data_row()
253         t = self.t
254         state_full = self.state_full
255         action = self.action
256         run_obj = self.run_obj
257         accum_obj = self.accum_obj
258
259     else:
260         self.simulator.sim_step()
261
262         t, state, observation, state_full = self.simulator.
            get_sim_step_data()
263
264         while observation[2] > np.pi:
265             observation[2] -= 2 * np.pi
266         while observation[2] < -np.pi:
267             observation[2] += 2 * np.pi
268
269         action = self.ctrl_selector(t, observation, self.
            action_manual, self.ctrl_nominal, self.
            ctrl_benchmarking, self.ctrl_lqr, self.ctrl_mode)
270
271         self.sys.receive_action(action)
272         self.ctrl_benchmarking.receive_sys_state(self.sys._state
            )
273         self.ctrl_benchmarking.upd_accum_obj(observation, action
            )
274
275         run_obj = self.ctrl_benchmarking.run_obj(observation,
            action)
276         accum_obj = self.ctrl_benchmarking.accum_obj_val
277
278         xCoord = state_full[0]
279         yCoord = state_full[1]
280         alpha = state_full[2]
281         alpha_deg = alpha/np.pi*180
282
283         if self.is_print_sim_step:

```

```

284         self.logger.print_sim_step(t, xCoord, yCoord, alpha,
285                                     run_obj, accum_obj, action)
286
287     if self.is_log_data:
288         self.logger.log_data_row(self.datafile_curr, t, xCoord,
289                                 yCoord, alpha, run_obj, accum_obj, action)
290
291     # xy plane
292     text_time = 't = {time:2.3f}'.format(time = t)
293     upd_text(self.text_time_handle, text_time)
294     upd_line(self.line_traj, xCoord, yCoord) # Update the robot
295                                             's track on the plot
296
297     self.robot_marker.rotate(alpha_deg) # Rotate the robot on
298                                         the plot
299     self.scatter_sol.remove()
300     self.scatter_sol = self.axs_xy_plane.scatter(xCoord, yCoord,
301                                                  marker=self.robot_marker.marker, s=400, c='b')
302
303     # # Solution
304     upd_line(self.line_norm, t, la.norm([xCoord, yCoord]))
305     upd_line(self.line_alpha, t, alpha)
306
307     # Cost
308     upd_line(self.line_run_obj, t, run_obj)
309     upd_line(self.line_accum_obj, t, accum_obj)
310     text_accum_obj = r'$\int \mathrm{{Run.\,obj.}} \,\mathrm{{d}}$ = {accum_obj:2.1f}'.format(accum_obj = accum_obj)
311     upd_text(self.text_accum_obj_handle, text_accum_obj)
312
313     # Control
314     for (line, action_single) in zip(self.lines_ctrl, action):
315         upd_line(line, t, action_single)
316
317     # Run done
318     if t >= self.t1 or np.linalg.norm(observation[:2]) < 0.2:

```

```

316         if self.is_print_sim_step:
317             print('.....Run {run
                 :2d} done.....').
                 format(run = self.run_curr))
318
319         self.run_curr += 1
320
321         if self.is_log_data:
322             print("Data file: ", self.datafile_curr)
323             print(str(self.datafile_curr) + ".svg" + " saved")
324             plt.savefig(str(self.datafile_curr) + ".svg")
325
326         if self.run_curr > self.Nruns:
327             print('Animation done...')
328             # print(self.AAA)
329             self.stop_anm()
330             # plt.close('all')
331             # print("HERE OK")
332             return
333
334         if self.is_log_data:
335             self.datafile_curr = self.datafiles[self.run_curr-1]
336
337         # Reset simulator
338         self.simulator.reset()
339
340         # Reset controller
341         if self.ctrl_mode != 'nominal':
342             self.ctrl_benchmarking.reset(self.t0)
343         else:
344             self.ctrl_nominal.reset(self.t0)
345
346
347         accum_obj = 0
348
349         reset_line(self.line_norm)
350         reset_line(self.line_alpha)
351         reset_line(self.line_run_obj)

```



```

352         reset_line(self.line_accum_obj)
353         reset_line(self.lines_ctrl[0])
354         reset_line(self.lines_ctrl[1])
355         reset_line(self.line_traj)
356
357         upd_line(self.line_traj, np.nan, np.nan)

```

Listing 4: Simulator.py

```

1  """
2  Contains one single class that simulates controller-system (agent-
3  environment) loops.
4
5  The system can be of three types:
6
7  - discrete-time deterministic
8  - continuous-time deterministic or stochastic
9  - discrete-time stochastic (to model Markov chains)
10
11  """
12
13  import numpy as np
14  import scipy as sp
15
16  from utilities import rej_sampling_rvs
17
18  class Simulator:
19      """
20      Class for simulating closed loops (system-controllers).
21
22      Attributes
23      -----
24      sys_type : : string
25          Type of system by description:
26
27          | 'diff_eqn' : differential equation :math:'\mathcal{D}
28              state = f(state, u, q)'
29          | 'discr_fnc' : difference equation :math:'state^{+} = f(
30              state, u, q)'
31          | 'discr_prob' : by probability distribution :math:'X^{+} \

```

```

28         sim P_X(state^+| state, u, q)'
29
29 where:
30
31     | :math:'state' : state
32     | :math:'u' : input
33     | :math:'q' : disturbance
34
35 closed_loop_rhs : : function
36     Right-hand side description of the closed-loop system.
37     Say, if you instantiated a concrete system (i.e., as an
38         instance of a subclass of ''system'' class with concrete
39         ''closed_loop_rhs'' method) as ''my_sys'',
40         this could be just ''my_sys.closed_loop_rhs''.
41
42 sys_out : : function
43     System output function.
44     Same as above, this could be, say, ''my_sys.out''.
45
46 is_dyn_ctrl : : 0 or 1
47     If 1, the controller (a.k.a. agent) is considered as a part
48     of the full state vector.
49
50 state_init, disturb_init, action_init : : vectors
51     Initial values of the (open-loop) system state, disturbance
52     and input.
53
54 t0, t1, dt : : numbers
55     Initial, final times and time step size
56
57 max_step, first_step, atol, rtol : : numbers
58     Parameters for an ODE solver (used if ''sys_type'' is ''
59     diff_eqn'').
60
61 See also
62 -----
63
64 ''systems'' module

```

```

60
61     """
62
63     def __init__(self, sys_type,
64                   closed_loop_rhs,
65                   sys_out,
66                   state_init,
67                   disturb_init=[],
68                   action_init=[],
69                   t0=0,
70                   t1=1,
71                   dt=1e-2,
72                   max_step=0.5e-2,
73                   first_step=1e-6,
74                   atol=1e-5,
75                   rtol=1e-3,
76                   is_disturb=0,
77                   is_dyn_ctrl=0):
78
79     """
80     Parameters
81     -----
82     sys_type : : string
83               Type of system by description:
84
85               | ‘‘diff_eqn’’ : differential equation :math:‘\mathcal{D}
86                   state = f(state, u, q)’
87               | ‘‘discr_fnc’’ : difference equation :math:‘state^{+} = f
88                   (state, u, q)’
89               | ‘‘discr_prob’’ : by probability distribution :math:‘X
90                   ^{+} \sim P_X(state^{+}| state, u, q)’
91
92     where:
93
94         | :math:‘state’ : state
95         | :math:‘u’ : input
96         | :math:‘q’ : disturbance

```

```

95     closed_loop_rhs : : function
96         Right-hand side description of the closed-loop system.
97         Say, if you instantiated a concrete system (i.e., as an
          instance of a subclass of ‘‘System’’ class with
          concrete ‘‘closed_loop_rhs’’ method) as ‘‘my_sys’’,
98         this could be just ‘‘my_sys.closed_loop_rhs’’.
99
100     sys_out : : function
101         System output function.
102         Same as above, this could be, say, ‘‘my_sys.out’’.
103
104     is_dyn_ctrl : : 0 or 1
105         If 1, the controller (a.k.a. agent) is considered as a
          part of the full state vector.
106
107     state_init, disturb_init, action_init : : vectors
108         Initial values of the (open-loop) system state,
          disturbance and input.
109
110     t0, t1, dt : : numbers
111         Initial, final times and time step size
112
113     max_step, first_step, atol, rtol : : numbers
114         Parameters for an ODE solver (used if ‘‘sys_type’’ is ‘‘
          diff_eqn’’).
115     """
116
117     self.sys_type = sys_type
118     self.closed_loop_rhs = closed_loop_rhs
119     self.sys_out = sys_out
120     self.dt = dt
121
122     # Build full state of the closed-loop
123     if is_dyn_ctrl:
124         if is_disturb:
125             state_full_init = np.concatenate([state_init,
          disturb_init, action_init])
126         else:

```

```

127         state_full_init = np.concatenate([state_init,
128                                           action_init])
129
130     else:
131         if is_disturb:
132             state_full_init = np.concatenate([state_init,
133                                               disturb_init])
134         else:
135             state_full_init = state_init
136
137     self.state_full = state_full_init
138
139     self.t = t0
140     self.state = state_init
141     self.dim_state = state_init.shape[0]
142     self.observation = self.sys_out(state_init)
143
144     if sys_type == "diff_eqn":
145
146         # Store these for reset purposes
147         self.state_full_init = state_full_init
148         self.t0 = t0
149         self.t1 = t1
150         self.max_step = dt/2
151         self.first_step = first_step
152         self.atol = atol
153         self.rtol = rtol
154
155         # self.ODE_solver = sp.integrate.RK45(closed_loop_rhs,
156         #                                     t0, state_full_init, t1, max_step = dt/2, first_step=
157         #                                     first_step, atol=atol, rtol=rtol)
158         self.ODE_solver = sp.integrate.RK45(self.closed_loop_rhs
159
160                                             ,
161                                             self.t0,
162                                             self.state_full_init
163
164                                             ,
165                                             self.t1,
166                                             max_step=self.dt/2,
167                                             first_step=self.

```

```

159         first_step,
160         atol=self.atol,
161         rtol=self.rtol
162     )
163
164     def sim_step(self):
165         """
166         Do one simulation step and update current simulation data (
167         time, system state and output).
168
169         """
170         if self.sys_type == "diff_eqn":
171             self.ODE_solver.step()
172
173             self.t = self.ODE_solver.t
174             self.state_full = self.ODE_solver.y
175
176             self.state = self.state_full[0:self.dim_state]
177             self.observation = self.sys_out(self.state)
178
179         elif self.sys_type == "discr_fnc":
180             self.t = self.t + self.dt
181             self.state_full = self.closed_loop_rhs(self.t, self.
182             state_full)
183
184             self.state = self.state_full[0:self.dim_state]
185             self.observation = self.sys_out(self.state)
186
187         elif self.sys_type == "discr_prob":
188             self.state_full = rej_sampling_rvs(self.dim_state, self.
189             closed_loop_rhs, 10)
190
191             self.t = self.t + self.dt
192
193             self.state = self.state_full[0:self.dim_state]
194             self.observation = self.sys_out(self.state)
195         else:
196             raise ValueError('Invalid system description')

```

```

193
194     def get_sim_step_data(self):
195         """
196         Collect current simulation data: time, system state and
197             output, and, for completeness, full closed-loop state.
198         """
199
200         t, state, observation, state_full = self.t, self.state, self
201             .observation, self.state_full
202
203         return t, state, observation, state_full
204
205     def reset(self):
206         if self.sys_type == "diff_eqn":
207             self.ODE_solver = sp.integrate.RK45(self.closed_loop_rhs
208                 ,
209                 self.t0,
210                 self.state_full_init,
211                 self.t1,
212                 max_step=self.dt/2,
213                 first_step=self.first_step,
214                 atol=self.atol,
215                 rtol=self.rtol
216             )
217
218         else:
219             self.t = self.t0
220             self.ODE_solver.y = self.state_full_init

```

Listing 5: Models.py

```

1  """
2  Contains classes to be used in fitting system models.
3  Includes both State-Space and placeholder Neural Network models.
4  """
5
6  import numpy as np # Importing numpy for matrix operations
7

```

```

8 class ModelSS:
9     """
10     Discrete-Time State-Space Model
11
12     \[
13     \begin{array}{ll}
14         \hat{x}_{k+1} &= A \hat{x}_k + B u_k \\
15         y_k &= C \hat{x}_k + D u_k
16     \end{array}
17     \]
18
19     Attributes:
20     -----
21     A, B, C, D : ndarray
22         State-space matrices defining system dynamics.
23     x0set : ndarray
24         Initial condition (state estimate) for simulation.
25     """
26
27     def __init__(self, A, B, C, D, x0est): # Consistent naming for
28         initial state
29         self.A = A # State transition matrix
30         self.B = B # Input matrix
31         self.C = C # Output matrix
32         self.D = D # Feedforward matrix
33         self.x0set = x0est # Consistent naming for initial state
34
35     def upd_pars(self, Anew, Bnew, Cnew, Dnew):
36         """Update system matrices with new values."""
37         self.A = Anew # Update state transition matrix
38         self.B = Bnew # Update input matrix
39         self.C = Cnew # Update output matrix
40         self.D = Dnew # Update feedforward matrix
41
42     def updateIC(self, x0setNew):
43         """Update initial condition/state estimate."""
44         self.x0set = x0setNew # Consistent naming for initial
45         state

```



```

44
45     def predict(self, x, u):
46         """
47         Perform one-step forward simulation using current model.
48
49         Parameters:
50         -----
51         x : ndarray
52             Current state.
53         u : ndarray
54             Current control input.
55
56         Returns:
57         -----
58         x_next : ndarray
59             Predicted next state.
60         y : ndarray
61             Output signal.
62         """
63         x_next = self.A @ x + self.B @ u    # State update equation
64         y = self.C @ x + self.D @ u    # Output equation
65         return x_next, y
66
67
68     class ModelNN:
69         """
70         Placeholder for Neural Network Model.
71         Intended to be implemented in the future for learning nonlinear
72         dynamics.
73         """
74
75         def __init__(self, *args, **kwargs):
76             raise NotImplementedError(f"Class {self.__class__.__name__}
77                                     is not yet implemented.")
78             # Placeholder for future implementation
79             # This will raise an error if instantiated, indicating that
80             # the class is not yet ready for use.
81
82         def predict(self, x, u):

```

```

79     """
80     Placeholder for prediction method.
81     Intended to be implemented in the future for learning
82         nonlinear dynamics.
83
84     Parameters:
85     -----
86     x : ndarray
87         Current state.
88     u : ndarray
89         Current control input.
90
91     Returns:
92     -----
93     NotImplementedError
94         Indicates that this method is not yet implemented.
95     """
96     raise NotImplementedError(f"Method {self.predict.__name__}
97         in class {self.__class__.__name__} is not yet implemented
98         .")
99
100     # Placeholder for future implementation
101     # This will raise an error if called, indicating that the
102         method is not yet ready for use.

```

Listing 6: Loggers.py

```

1  """
2  Logger Module for Simulation Data
3  =====
4
5  This file provides logging functionality for simulations, including
6  both console output and CSV file logging.
7  Supports a generic base Logger class and a concrete implementation
8  for a 3-wheel robot (Sys3WRobotNI).
9  """
10
11  import csv
12  from tabulate import tabulate

```

```

12 class Logger:
13     """
14     Abstract base class for data loggers.
15
16     Subclasses must override:
17         - print_sim_step: prints simulation step data to console.
18         - log_data_row: writes simulation step data to CSV.
19     """
20
21     def print_sim_step(self, *args, **kwargs):
22         """Prints one step of simulation data to the console (must
23         override)."""
24         raise NotImplementedError("Subclasses must implement
25         print_sim_step")
26
27     def log_data_row(self, *args, **kwargs):
28         """Logs one row of data to a CSV file (must override)."""
29         raise NotImplementedError("Subclasses must implement
30         log_data_row")
31
32 class Logger3WRobotNI(Logger):
33     """
34     Logger for a 3-wheel robot (non-holonomic integrator).
35
36     Fields logged:
37         - Time (t [s])
38         - x position (x [m])
39         - y position (y [m])
40         - Orientation angle (alpha [rad])
41         - Instantaneous cost (run_obj)
42         - Accumulated cost (accum_obj)
43         - Linear velocity (v [m/s])
44         - Angular velocity (omega [rad/s])
45     """
46
47     def print_sim_step(self, t, xCoord, yCoord, alpha, run_obj,
48         accum_obj, action):

```

```

46     """
47     Prints one simulation step as a formatted table to the
        console.
48     """
49     row_header = [
50         't [s]', 'x [m]', 'y [m]', 'alpha [rad]',
51         'run_obj', 'accum_obj', 'v [m/s]', 'omega [rad/s]'
52     ]
53
54     row_data = [t, xCoord, yCoord, alpha, run_obj, accum_obj,
55                 action[0], action[1]]
56
57     row_format = (
58         '8.3f', '8.3f', '8.3f', '8.3f',
59         '8.1f', '8.1f', '8.3f', '8.3f'
60     )
61
62     table = tabulate(
63         [row_header, row_data],
64         floatfmt=row_format,
65         headers='firstrow',
66         tablefmt='grid'
67     )
68
69     print(table)
70
71 def log_data_row(self, datafile, t, xCoord, yCoord, alpha,
72                 run_obj, accum_obj, action):
73     """
74     Appends a row of simulation data to the specified CSV file.
75
76     Parameters:
77         datafile (str): Path to CSV file.
78         t (float): Time.
79         xCoord (float): X-position.
80         yCoord (float): Y-position.
81         alpha (float): Orientation.
82         run_obj (float): Instantaneous cost.

```

```

81         accum_obj (float): Accumulated cost.
82         action (list): Control input [v, omega].
83     """
84     with open(datafile, 'a', newline='') as outfile:
85         writer = csv.writer(outfile)
86         writer.writerow([t, xCoord, yCoord, alpha, run_obj,
                           accum_obj, action[0], action[1]])

```

Listing 7: Controllers.py

```

1  """
2  Contains controllers a.k.a. agents.
3
4  """
5
6  from utilities import dss_sim
7  from utilities import rep_mat
8  from utilities import uptria2vec
9  from utilities import push_vec
10 import models
11 import numpy as np
12 import scipy as sp
13 from scipy.linalg import solve_discrete_are
14 from numpy.random import rand
15 from scipy.optimize import minimize
16 from scipy.optimize import basinhopping
17 from scipy.optimize import NonlinearConstraint
18 from scipy.stats import multivariate_normal
19 from numpy.linalg import lstsq
20 from numpy import reshape
21 import warnings
22 import math
23 # For debugging purposes
24 from tabulate import tabulate
25 import os
26
27 def ctrl_selector(t, observation, action_manual, ctrl_nominal,
28                 ctrl_benchmarking, ctrl_lqr, mode):
29     """

```

```

29     Main interface for various controllers.
30
31     Parameters
32     -----
33     mode : : string
34           Controller mode as acronym of the respective control method.
35
36     Returns
37     -----
38     action : : array of shape '[dim_input, ]'.
39             Control action.
40
41     """
42
43     if mode=='manual':
44         action = action_manual
45     elif mode=='Nominal':
46         action = ctrl_nominal.compute_action(t, observation)
47     elif mode=='lqr':
48         action=ctrl_lqr.compute_action(t,observation)
49     else: # Controller for benchmakring
50         action = ctrl_benchmarking.compute_action(t, observation)
51
52     return action
53
54
55 class ControllerOptimalPredictive:
56     """
57     Class of predictive optimal controllers, primarily model-
58         predictive control and predictive reinforcement learning,
59         that optimize a finite-horizon cost.
60
61     Currently, the actor model is trivial: an action is generated
62         directly without additional policy parameters.
63
64     Attributes
65     -----
66     dim_input, dim_output : : integer

```

```

64         Dimension of input and output which should comply with the
           system-to-be-controlled.
65     mode : : string
66         Controller mode. Currently available (:math:'\\rho' is the
           running objective, :math:'\\gamma' is the discounting
           factor):
67
68     .. list-table:: Controller modes
69        :widths: 75 25
70        :header-rows: 1
71
72        * - Mode
73          - Cost function
74        * - 'MPC' - Model-predictive control (MPC)
75          - :math:'J_a \\left( y_1, \\{action\\}_1^{N_a} \\right)'
            = \\sum_{k=1}^{N_a} \\gamma^{k-1} \\rho(y_k, u_k)'
76        * - 'RQL' - RL/ADP via :math:'N_a-1' roll-outs of :math:
            : '\\rho'
77          - :math:'J_a \\left( y_1, \\{action\\}_{1}^{N_a} \\right)'
            = \\sum_{k=1}^{N_a-1} \\gamma^{k-1} \\rho(y_k, u_k)
            + \\hat{Q}^{\\theta}(y_{N_a}, u_{N_a})'
78        * - 'SQL' - RL/ADP via stacked Q-learning
79          - :math:'J_a \\left( y_1, \\{action\\}_1^{N_a} \\right)'
            = \\sum_{k=1}^{N_a-1} \\hat{\\gamma}^{k-1} Q^{\\theta}(y_{N_a}, u_{N_a})'
80
81         Here, :math:'\\theta' are the critic parameters (neural
           network weights, say) and :math:'y_1' is the current
           observation.
82
83         *Add your specification into the table when customizing the
           agent*.
84
85     ctrl_bnds : : array of shape '[dim_input, 2]'
86         Box control constraints.
87         First element in each row is the lower bound, the second -
           the upper bound.
88         If empty, control is unconstrained (default).

```

```

89     action_init : : array of shape '[dim_input, ]'
90         Initial action to initialize optimizers.
91     t0 : : number
92         Initial value of the controller's internal clock.
93     sampling_time : : number
94         Controller's sampling time (in seconds).
95     Nactor : : natural number
96         Size of prediction horizon :math:'N_a' .
97     pred_step_size : : number
98         Prediction step size in :math:'J_a' as defined above (in
99         seconds). Should be a multiple of 'sampling_time'.
100         Commonly, equals it, but here left adjustable for
101         convenience. Larger prediction step size leads to longer
102         factual horizon.
103     sys_rhs, sys_out : : functions
104         Functions that represent the right-hand side, resp., the
105         output of the exogenously passed model.
106         The latter could be, for instance, the true model of the
107         system.
108         In turn, 'state_sys' represents the (true) current state
109         of the system and should be updated accordingly.
110         Parameters 'sys_rhs, sys_out, state_sys' are used in those
111         controller modes which rely on them.
112     buffer_size : : natural number
113         Size of the buffer to store data.
114     gamma : : number in (0, 1]
115         Discounting factor.
116         Characterizes fading of running objectives along horizon.
117     Ncritic : : natural number
118         Critic stack size :math:'N_c' . The critic optimizes the
119         temporal error which is a measure of critic's ability to
120         capture the
121         optimal infinite-horizon cost (a.k.a. the value function).
122         The temporal errors are stacked up using the said buffer.
123     critic_period : : number
124         The critic is updated every 'critic_period' units of time.
125     critic_struct : : natural number
126         Choice of the structure of the critic's features.

```



```

117
118     Currently available:
119
120     .. list-table:: Critic structures
121        :widths: 10 90
122        :header-rows: 1
123
124        * - Mode
125          - Structure
126        * - 'quad-lin'
127          - Quadratic-linear
128        * - 'quadratic'
129          - Quadratic
130        * - 'quad-nomix'
131          - Quadratic, no mixed terms
132        * - 'quad-mix'
133          - Quadratic, no mixed terms in input and output, i.e.,
134            :math:'w_1 y_1^2 + \dots w_p y_p^2 + w_{p+1} y_1
135              u_1 + \dots w_{\bullet} u_1^2 + \dots',
136            where :math:'w' is the critic's weight vector
137
138        *Add your specification into the table when customizing the
139          critic*.
140
141 run_obj_struct : : string
142     Choice of the running objective structure.
143
144     Currently available:
145
146     .. list-table:: Critic structures
147        :widths: 10 90
148        :header-rows: 1
149
150        * - Mode
151          - Structure
152        * - 'quadratic'
153          - Quadratic :math:'\chi^{\text{top}}_{R_1} \chi', where :math
154            :'\chi = [\text{observation}, \text{action}]', ''run_obj_pars''
155            should be ''[R1]''

```

```

150     * - 'biquadratic'
151     - 4th order :math: '\left( \chi^{\top} \right)^2 R_2'
      \left( \chi \right)^2 + \chi^{\top} R_1 \chi',
      where :math: '\chi = [\text{observation}, \text{action}]', ''
      run_obj_pars''
152     should be ''[R1, R2]''
153
154     *Pass correct run objective parameters in* ''run_obj_pars''
      *(as a list)*
155
156     *When customizing the running objective, add your
      specification into the table above*
157
158 References
159 -----
160 .. [1] Osinenko, Pavel, et al. "Stacked adaptive dynamic
      programming with unknown system model." IFAC-PapersOnLine
      50.1 (2017): 4150-4155
161
162 """
163 def __init__(self,
164             dim_input,
165             dim_output,
166             mode='MPC',
167             ctrl_bnds=[],
168             action_init = [],
169             t0=0,
170             sampling_time=0.1,
171             Nactor=1,
172             pred_step_size=0.1,
173             sys_rhs=[],
174             sys_out=[],
175             state_sys=[],
176             buffer_size=20,
177             gamma=1,
178             Ncritic=4,
179             critic_period=0.1,
180             critic_struct='quad-nomix',

```

```

181         run_obj_struct='quadratic',
182         run_obj_pars=[],
183         observation_target=[],
184         state_init=[],
185         obstacle=[],
186         seed=1):
187
188     """
189
190     Parameters
191     -----
192
193     dim_input, dim_output : : integer
194         Dimension of input and output which should comply
195         with the system-to-be-controlled.
196
197     mode : : string
198         Controller mode. Currently available (:math:`\rho`
199         is the running objective, :math:`\gamma` is the
200         discounting factor):
201
202
203     .. list-table:: Controller modes
204        :widths: 75 25
205        :header-rows: 1
206
207        * - Mode
208          - Cost function
209        * - 'MPC' - Model-predictive control (MPC)
210          - :math:`J_a \left( y_1, \{action\}_1^{N_a} \right) = \sum_{k=1}^{N_a} \gamma^{k-1} \rho(y_k, u_k)`
211        * - 'RQL' - RL/ADP via :math:`N_a-1` roll-outs of :
212          :math:`\rho`
213          - :math:`J_a \left( y_1, \{action\}_1^{N_a} \right) = \sum_{k=1}^{N_a-1} \gamma^{k-1} \rho(y_k, u_k) + \hat{Q}^{\theta}(y_{N_a}, u_{N_a})`
214        * - 'SQL' - RL/ADP via stacked Q-learning
215          - :math:`J_a \left( y_1, \{action\}_1^{N_a} \right) = \sum_{k=1}^{N_a-1} \gamma^{k-1} \hat{Q}^{\theta}(y_{N_a}, u_{N_a})`

```

```

208         Here,  $\theta$  are the critic parameters (
           neural network weights, say) and  $y_1$  is
           the current observation.

209
210         *Add your specification into the table when
           customizing the agent* .

211
212     ctrl_bnds : : array of shape '[dim_input, 2]'
           Box control constraints.
213
           First element in each row is the lower bound, the
214             second - the upper bound.
           If empty, control is unconstrained (default).
215
           action_init : : array of shape '[dim_input, ]'
           Initial action to initialize optimizers.
216
           t0 : : number
           Initial value of the controller's internal clock
217
           sampling_time : : number
           Controller's sampling time (in seconds)
218
           Nactor : : natural number
           Size of prediction horizon  $N_a$ 
219
           pred_step_size : : number
           Prediction step size in  $J$  as defined above (
220             in seconds). Should be a multiple of '
           sampling_time'. Commonly, equals it, but here
221             left adjustable for
           convenience. Larger prediction step size leads to
222             longer factual horizon.
           sys_rhs, sys_out : : functions
223
           Functions that represent the right-hand side, resp.,
           the output of the exogenously passed model.
224
           The latter could be, for instance, the true model of
           the system.
225
           In turn, 'state_sys' represents the (true) current
           state of the system and should be updated
           accordingly.
226
           Parameters 'sys_rhs, sys_out, state_sys' are used
           in those controller modes which rely on them.
227
           buffer_size : : natural number

```

```

233         Size of the buffer to store data.
234     gamma : : number in (0, 1]
235         Discounting factor.
236         Characterizes fading of running objectives along
            horizon.
237     Ncritic : : natural number
238         Critic stack size :math:'N_c' . The critic optimizes
            the temporal error which is a measure of critic's
            ability to capture the
239         optimal infinite-horizon cost (a.k.a. the value
            function). The temporal errors are stacked up
            using the said buffer.
240     critic_period : : number
241         The critic is updated every ''critic_period'' units
            of time.
242     critic_struct : : natural number
243         Choice of the structure of the critic's features.
244
245         Currently available:
246
247         .. list-table:: Critic feature structures
248            :widths: 10 90
249            :header-rows: 1
250
251            * - Mode
252              - Structure
253            * - 'quad-lin'
254              - Quadratic-linear
255            * - 'quadratic'
256              - Quadratic
257            * - 'quad-nomix'
258              - Quadratic, no mixed terms
259            * - 'quad-mix'
260              - Quadratic, no mixed terms in input and output,
                i.e., :math:'w_1 y_1^2 + \dots w_p y_p^2 +
                w_{p+1} y_1 u_1 + \dots w_{\bullet} u_1^2 +
                \dots',
261              where :math:'w' is the critic's weights

```

```

262         *Add your specification into the table when
263         customizing the critic*.
264     run_obj_struct : : string
265         Choice of the running objective structure.
266
267     Currently available:
268
269     .. list-table:: Running objective structures
270        :widths: 10 90
271        :header-rows: 1
272
273        * - Mode
274          - Structure
275        * - 'quadratic'
276          - Quadratic :math:'\\chi^{\\top} R_1 \\chi', where
277            :math:'\\chi = [observation, action]', ''
278              run_obj_pars'' should be ''[R1]''
279        * - 'biquadratic'
280          - 4th order :math:'\\left( \\chi^{\\top} \\right)
281            ^2 R_2 \\left( \\chi \\right)^2 + \\chi^{\\top}
282            R_1 \\chi', where :math:'\\chi = [
283            observation, action]', ''run_obj_pars''
284              should be ''[R1, R2]''
285
286     """
287
288     np.random.seed(seed)
289     # print(seed)
290
291     self.dim_input = dim_input
292     self.dim_output = dim_output
293
294     self.mode = mode
295
296     self.ctrl_clock = t0
297     self.sampling_time = sampling_time
298
299     # Controller: common

```

```

294     self.Nactor = Nactor
295     self.pred_step_size = pred_step_size
296
297     self.action_min = np.array( ctrl_bnds[:,0] )
298     self.action_max = np.array( ctrl_bnds[:,1] )
299     self.action_sqn_min = rep_mat(self.action_min, 1, Nactor)
300     self.action_sqn_max = rep_mat(self.action_max, 1, Nactor)
301     self.action_sqn_init = []
302     self.state_init = []
303
304     if len(action_init) == 0:
305         self.action_curr = self.action_min/10
306         self.action_sqn_init = rep_mat( self.action_min/10 , 1,
307             self.Nactor)
308         self.action_init = self.action_min/10
309     else:
310         self.action_curr = action_init
311         self.action_sqn_init = rep_mat( action_init , 1, self.
312             Nactor)
313
314     self.action_buffer = np.zeros( [buffer_size, dim_input] )
315     self.observation_buffer = np.zeros( [buffer_size, dim_output
316         ] )
317     self.t=0
318
319     # Exogeneous model's things
320     self.sys_rhs = sys_rhs
321     self.sys_out = sys_out
322     self.state_sys = state_sys
323
324     # Learning-related things
325     self.buffer_size = buffer_size
326     self.critic_clock = t0
327     self.gamma = gamma
328     self.Ncritic = Ncritic
329     self.Ncritic = np.min([self.Ncritic, self.buffer_size-1]) #
330         Clip critic buffer size

```

```

328     self.critic_period = critic_period
329     self.critic_struct = critic_struct
330     self.run_obj_struct = run_obj_struct
331     self.run_obj_pars = run_obj_pars
332     self.observation_target = observation_target
333
334     self.accum_obj_val = 0
335     # print('---Critic structure---', self.critic_struct)
336
337     if self.critic_struct == 'quad-lin':
338         self.dim_critic = int( ( ( self.dim_output + self.
339             dim_input ) + 1 ) * ( self.dim_output + self.
340             dim_input )/2 + (self.dim_output + self.dim_input) )
341         self.Wmin = -1e3*np.ones(self.dim_critic)
342         self.Wmax = 1e3*np.ones(self.dim_critic)
343     elif self.critic_struct == 'quadratic':
344         self.dim_critic = int( ( ( self.dim_output + self.
345             dim_input ) + 1 ) * ( self.dim_output + self.
346             dim_input )/2 )
347         self.Wmin = np.zeros(self.dim_critic)
348         self.Wmax = 1e3*np.ones(self.dim_critic)
349     elif self.critic_struct == 'quad-nomix':
350         self.dim_critic = self.dim_output + self.dim_input
351         self.Wmin = np.zeros(self.dim_critic)
352         self.Wmax = 1e3*np.ones(self.dim_critic)
353     elif self.critic_struct == 'quad-mix':
354         self.dim_critic = int( self.dim_output + self.dim_output
355             * self.dim_input + self.dim_input )
356         self.Wmin = -1e3*np.ones(self.dim_critic)
357         self.Wmax = 1e3*np.ones(self.dim_critic)
358     elif self.critic_struct == 'poly3':
359         self.dim_critic = int( ( ( self.dim_output + self.
360             dim_input ) + 1 ) * ( self.dim_output + self.
361             dim_input ) )
362         self.Wmin = -1e3*np.ones(self.dim_critic)
363         self.Wmax = 1e3*np.ones(self.dim_critic)
364     elif self.critic_struct == 'poly4':
365         self.dim_critic = int( ( ( self.dim_output + self.

```



```

        dim_input ) + 1 ) * ( self.dim_output + self.
        dim_input )/2 * 3)
359         self.Wmin = np.zeros(self.dim_critic)
360         self.Wmax = np.ones(self.dim_critic)
361 self.N_CTRL = N_CTRL(k_rho=1.0, k_alpha=4.0, k_beta=-1.5, #
        Example gains, vary for Experiment A
362 target_x=0.0, target_y=0.0, target_theta=0.0,
363 sampling_time=sampling_time,
364 dim_input=dim_input, dim_output=dim_output)
365
366
367 def reset(self,t0):
368     """
369     Resets agent for use in multi-episode simulation.
370     Only internal clock, value and current actions are reset.
371     All the learned parameters are retained.
372
373     """
374
375     # Controller: common
376
377     if len(self.action_init) == 0:
378         self.action_curr = self.action_min/10
379         self.action_sqn_init = rep_mat( self.action_min/10 , 1,
            self.Nactor)
380         self.action_init = self.action_min/10
381     else:
382         self.action_curr = self.action_init
383         self.action_sqn_init = rep_mat( self.action_init , 1,
            self.Nactor)
384
385     self.action_buffer = np.zeros( [self.buffer_size, self.
        dim_input] )
386     self.observation_buffer = np.zeros( [self.buffer_size, self.
        dim_output] )
387
388     self.critic_clock = t0
389     self.ctrl_clock = t0

```

```

390
391 def receive_sys_state(self, state):
392     """
393     Fetch exogenous model state. Used in some controller modes.
394     See class documentation.
395     """
396     self.state_sys = state
397
398 def upd_accum_obj(self, observation, action):
399     """
400     Sample-to-sample accumulated (summed up or integrated)
401     running objective. This can be handy to evaluate the
402     performance of the agent.
403     If the agent succeeded to stabilize the system, ‘‘accum_obj
404     ‘‘ would converge to a finite value which is the
405     performance mark.
406     The smaller, the better (depends on the problem
407     specification of course - you might want to maximize cost
408     instead).
409     """
410     self.accum_obj_val += self.run_obj(observation, action)*self
411     .sampling_time
412
413 def run_obj(self, observation, action, terminal=False):
414     """
415     Running (equivalently, instantaneous or stage) objective.
416     Depending on the context, it is also called utility,
417     reward, running cost etc.
418     See class documentation.
419     """
420     observation_arr = np.array(observation)
421     action_arr = np.array(action)
422
423     # Handle observation target: calculate error if a target is
424     provided

```

```

417     self.observation_target=np.array([0.0,0.0,0.0])
418     observation_err = observation_arr - self.observation_target
419
420     # Form the combined state-action vector chi
421     chi = np.concatenate((observation_err, action_arr))
422     run_obj=0
423
424
425
426     # if self.t==self.Nactor-1:
427     #     terminal=True
428
429     if self.run_obj_struct == 'quadratic':
430         # run_obj_pars should be [R1] where R1 is a matrix for
431         # chi.T @ R1 @ chi
432         if not self.run_obj_pars or len(self.run_obj_pars) < 1:
433             warnings.warn("run_obj_pars is empty or malformed
434                           for 'quadratic' mode. Returning 0 for running
435                           objective.")
436             return 0.0 # Return a float
437         R1 = np.array(self.run_obj_pars[0])
438         # print(R1)
439         # Check dimensions for compatibility: R1 must be square
440         # and match chi's dimension
441         expected_dim = chi.shape[0]
442         if R1.shape != (expected_dim, expected_dim):
443             warnings.warn(f"R1 matrix dimension mismatch.
444                           Expected {expected_dim}x{expected_dim}, got {R1.
445                           shape}. Returning 0 for running objective.")
446             return 0.0 # Return a float
447         run_obj = chi.T @ R1 @ chi
448         self.t+=1
449         # print(self.t)
450
451         # Add terminal cost if requested
452         if terminal:
453             if len(self.run_obj_pars) < 2:
454                 warnings.warn("Qf not provided in run_obj_pars

```

```

449         for terminal cost. Skipping terminal cost.")
450     else:
451         Qf = np.array(self.run_obj_pars[1])
452         obs_dim = observation_err.shape[0]
453         if Qf.shape != (obs_dim, obs_dim):
454             warnings.warn(f"Qf matrix dimension mismatch
455                             . Expected {obs_dim}x{obs_dim}, got {Qf.
456                                 shape}. Skipping terminal cost.")
457         else:
458             run_obj += observation_err.T @ Qf @
459                 observation_err
460
461     return run_obj
462
463 def _actor_cost(self, action_sqn, observation):
464     """
465     See class documentation.
466
467     Customization
468     -----
469
470     Introduce your mode and the respective actor loss in this
471     method. Don't forget to provide description in the class
472     documentation.
473
474     """
475
476     my_action_sqn = np.reshape(action_sqn, [self.Nactor, self.
477         dim_input])
478
479     observation_sqn = np.zeros([self.Nactor, self.dim_output])
480
481     # System observation prediction
482     observation_sqn[0, :] = observation
483     state = self.state_sys
484     for k in range(1, self.Nactor):

```

```

480         state = state + self.pred_step_size * self.sys_rhs(0.01,
481             state, my_action_sqn[k-1, :]) # Euler scheme
482
483         observation_sqn[k, :] = self.sys_out(state)
484
485     J = 0
486     if self.mode=='MPC':
487         for k in range(self.Nactor):
488
489             if k==self.Nactor-1:
490                 J+=self.gamma**k * self.run_obj(observation_sqn[
491                     k, :], my_action_sqn[k, :],terminal=True)
492
493                 J += self.gamma**k * self.run_obj(observation_sqn[k,
494                     :], my_action_sqn[k, :])
495
496         return J
497
498     def _actor_optimizer(self, observation):
499         """
500         This method is merely a wrapper for an optimizer that
501         minimizes :func:`~controllers.ControllerOptimalPredictive
502             ._actor_cost`.
503         See class documentation.
504
505         Customization
506         -----
507
508         This method normally should not be altered, adjust :func:`~
509             controllers.ControllerOptimalPredictive._actor_cost`
510         instead.
511
512         The only customization you might want here is regarding the
513         optimization algorithm.
514
515         # For direct implementation of state constraints, this needs
516             'partial' from 'functools'

```

```

509     # See [here](https://stackoverflow.com/questions/27659235/
        adding-multiple-constraints-to-scipy-minimize-
        autogenerate-constraint-dictionar)
510     # def state_constraint(action_sqn, idx):
511
512     #     my_action_sqn = np.reshape(action_sqn, [N, self.
        dim_input])
513
514     #     observation_sqn = np.zeros([idx, self.dim_output])
515
516     #     # System output prediction
517     #     if (mode==1) or (mode==3) or (mode==5):      # Via
        exogenously passed model
518     #         observation_sqn[0, :] = observation
519     #         state = self.state_sys
520     #         Y[0, :] = observation
521     #         x = self.x_s
522     #         for k in range(1, idx):
523     #             # state = get_next_state(state, my_action_sqn[
        k-1, :], delta)
524     #             state = state + delta * self.sys_rhs([], state
        , my_action_sqn[k-1, :], []) # Euler scheme
525     #             observation_sqn[k, :] = self.sys_out(state)
526
527     #     return observation_sqn[-1, 1] - 1
528
529     # my_constraints=[]
530     # for my_idx in range(1, self.Nactor+1):
531     #     my_constraints.append({'type': 'eq', 'fun': lambda
        action_sqn: state_constraint(action_sqn, idx=my_idx)})
532
533     # my_constraints = {'type': 'ineq', 'fun': state_constraint}
534
535     # Optimization method of actor
536     # Methods that respect constraints: BFGS, L-BFGS-B, SLSQP,
        trust-constr, Powell
537     # actor_opt_method = 'SLSQP' # Standard
538     """

```

```

539
540 actor_opt_method = 'SLSQP'
541 if actor_opt_method == 'trust-constr':
542     actor_opt_options = {'maxiter': 40, 'disp': False} #'
543     disp': True, 'verbose': 2}
544 else:
545     actor_opt_options = {'maxiter': 40, 'maxfev': 60, 'disp'
546     : False, 'adaptive': True, 'xatol': 1e-3, 'fatol': 1e
547     -3}
548
549 isGlobOpt = 0
550
551 my_action_sqn_init = np.reshape(self.action_sqn_init, [self.
552     Nactor*self.dim_input,])
553
554 bnds = sp.optimize.Bounds(self.action_sqn_min, self.
555     action_sqn_max, keep_feasible=True)
556
557 try:
558     if isGlobOpt:
559         minimizer_kwargs = {'method': actor_opt_method, '
560         bounds': bnds, 'tol': 1e-3, 'options':
561         actor_opt_options}
562         action_sqn = basinhopping(lambda action_sqn: self.
563             _actor_cost(action_sqn, observation),
564             my_action_sqn_init,
565             minimizer_kwargs=
566                 minimizer_kwargs,
567             niter = 10).x
568     else:
569         action_sqn = minimize(lambda action_sqn: self.
570             _actor_cost(action_sqn, observation),
571             my_action_sqn_init,
572             method=actor_opt_method,
573             tol=1e-3,
574             bounds=bnds,
575             options=actor_opt_options).x
576

```

```

567     except ValueError:
568         print('Actor''s optimizer failed. Returning default
569               action')
570         action_sqn = self.action_curr
571
572     ##print(action_sqn)
573
574     return action_sqn[:self.dim_input]      # Return first action
575
576 def compute_action(self, t, observation):
577     """
578     Main method. See class documentation.
579
580     Customization
581     -----
582
583     Add your modes, that you introduced in :func:`~controllers.
584     ControllerOptimalPredictive._actor_cost`, here.
585
586     """
587
588     time_in_sample = t - self.ctrl_clock
589
590     if time_in_sample >= self.sampling_time: # New sample
591         # Update controller's internal clock
592         self.ctrl_clock = t
593
594         action = None
595
596         if self.mode == 'MPC':
597
598             action = self._actor_optimizer(observation)
599             # print(action.shape)
600
601         elif self.mode == "Nominal":
602
603             action = self.N_CTRL.compute_action(observation)
604         if action is not None:

```



```

603         self.action_curr = action
604
605
606         return action
607
608     else:
609         return self.action_curr
610
611
612
613 # New/Updated N_CTRL Class
614 class N_CTRL:
615     def __init__(self, k_rho=1.0, k_alpha=2.0, k_beta=-1.5,
616                 target_x=0.0, target_y=0.0, target_theta=0.0,
617                 sampling_time=0.01, dim_input=2, dim_output=3): #
618         Added dim_input/output for consistency
619
620         self.k_rho = k_rho
621         self.k_alpha = k_alpha
622         self.k_beta = k_beta
623         self.dt = sampling_time # Use sampling_time for internal dt
624         consistency
625
626         self.target_x = target_x
627         self.target_y = target_y
628         self.target_theta = target_theta
629
630         self.ctrl_clock = 0.0 # Initial time for the controller's
631         internal clock
632         self.sampling_time = sampling_time
633         self.action_curr = np.array([0.0, 0.0]) # Initialize current
634         action to zero
635         self.accum_obj_val = 0.0 # Initialize accumulated objective
636         value
637         self.dim_input = dim_input
638         self.dim_output = dim_output
639
640         # Dummy attributes for compatibility with logger/visualizer
641         self.state_sys = np.zeros(dim_output) # N_CTRL doesn't use

```

```

        this directly, but visualizer might expect it.
# It will be updated by
    receive_sys_state
    if called.

def wrap_angle(self, angle):
    while angle > math.pi:
        angle -= 2 * math.pi
    while angle <= -math.pi:
        angle += 2 * math.pi
    return angle

def compute_action(self, t, observation):
    """
    Computes the control action for the 3-wheel robot.
    """
    time_in_sample = t - self.ctrl_clock
    # print("lqr")

    if t >= self.ctrl_clock + self.sampling_time - 1e-6:

        self.ctrl_clock = t # Update controller's internal clock
        print("nominal")

        # print(observation)

        x, y, th = observation
        x_f = self.target_x
        y_f = self.target_y
        th_f = self.target_theta

        dx = x_f - x
        dy = y_f - y
        rho = math.sqrt(dx**2 + dy**2)

        alpha = self.wrap_angle(-th + math.atan2(dy, dx))
        beta = self.wrap_angle((th_f - th) - alpha)

```

```

671         # Debug prints - these are very useful!
672         print(f"t: {t:.2f}, Obs: ({x:.2f}, {y:.2f}, {th:.2f})")
673         print(f"Target: ({x_f:.2f}, {y_f:.2f}, {th_f:.2f})")
674         # print(f"Errors: rho={rho:.4f}, alpha={alpha:.4f}, beta
           = {beta:.4f}")
675
676         # Define a small tolerance for "at target"
677         pos_tolerance = 0.05 # meters
678         angle_tolerance = 0.05 # radians
679
680         if rho < pos_tolerance:
681             if abs(self.wrap_angle(th_f - th)) < angle_tolerance
               :
682                 v = 0.0
683                 w = 0.0
684                 # print("--> At target, stopping. v=0, w=0")
685             else:
686                 v = self.k_rho * rho
687                 w = self.k_alpha * alpha + self.k_beta * beta
688                 # print(f"--> Control action: v={v:.4f}, w={w:.4f}")
689
690                 v = np.clip(v, -1, 1)
691                 w = np.clip(w, -1.0, 1.0)
692
693
694                 self.action_curr = np.array([v, w])
695                 return self.action_curr
696             else:
697                 return self.action_curr # Return the last computed
           action if not a new sample
698
699     def reset(self, t0):
700         """
701         Resets controller for new episode.
702         """
703         self.ctrl_clock = t0
704         self.action_curr = np.array([0.0, 0.0])
705         self.accum_obj_val = 0.0 # Reset accumulated objective

```

```

706
707 def receive_sys_state(self, state):
708     """
709     Placeholder method for compatibility with the simulation
710     framework.
711     N_CTRL doesn't actively use the full system state for its
712     internal logic,
713     but the main simulation loop might try to pass it.
714     """
715     self.state_sys = state # Store it just in case, or do
716     nothing if truly not needed.
717
718
719 def upd_accum_obj(self, observation, action):
720     """
721     Accumulates the running objective for performance evaluation
722     .
723     """
724     self.accum_obj_val += self.run_obj(observation, action) *
725     self.sampling_time
726
727
728 def run_obj(self, observation, action):
729     """
730     Calculates a running objective (cost) for the N_CTRL.
731     This is primarily for logging and visualization, not for the
732     controller's internal logic.
733     """
734     x, y, th = observation
735     x_f = self.target_x
736     y_f = self.target_y
737     th_f = self.target_theta
738
739     # Cost components: position error squared and orientation
740     error squared, and control effort.
741     pos_cost = (x - x_f)**2 + (y - y_f)**2
742     # Use a proper angle difference for orientation cost
743     # orientation_cost = self.wrap_angle(th - th_f)**2
744
745     # Control effort cost (e.g., squared velocities)

```

```

737     # v, w = action
738     # control_effort_cost = 0.1 * (v**2 + w**2) # Small weight
       on control effort
739
740     # You can tune these weights
741     total_cost = pos_cost
742     return total_cost
743
744 class LQR:
745     def __init__(self,A,B,Q,R,obs_target=[0.0,0.0,0.0],sampling_time
       =0.01,):
746         self.A=A
747         self.B=B
748         self.Q=Q
749         self.R=R
750         self.observation_target=obs_target
751         self.ctrl_clock = 0.0
752         self.sampling_time =sampling_time
753         self.action_curr = np.array([0.0, 0.0]) # Initialize current
       action to zero
754         self.accum_obj_val = 0.0
755
756     def compute_gain(self):
757
758         # Solve the Discrete Algebraic Riccati Equation (DARE)
759         P = solve_discrete_are(self.A, self.B, self.Q, self.R)
760
761         K=np.linalg.inv(self.R+self.B.T@P@self.B)@self.B.T@P@self.A
762
763         return K
764
765     def compute_action(self,t,observation):
766         # time_in_sample = t - self.ctrl_clock
767         print("lqr")
768
769         if t >= self.ctrl_clock + self.sampling_time - 1e-6:
770
771             self.ctrl_clock = t # Update controller's internal clock

```

```

772         # print(observation)
773         # observation_target=[]
774
775
776         state_err = observation - self.observation_target
777         # print(state_err.shape)
778
779         self.K=self.compute_gain()
780
781
782         u=-self.K@state_err
783         # print(u)
784
785         # x=self.A@state_err +self.B@u
786
787         self.action_curr = u
788
789         # print(x)
790         return self.action_curr
791     else:
792         return self.action_curr #
793
794
795     def reset(self, t0):
796         """
797         Resets controller for new episode.
798         """
799         self.ctrl_clock = t0
800         self.action_curr = np.array([0.0, 0.0])
801         self.accum_obj_val = 0.0 # Reset accumulated objective
802
803     def receive_sys_state(self, state):
804         """
805         Placeholder method for compatibility with the simulation
            framework.
806         N_CTRL doesn't actively use the full system state for its
            internal logic,
807         but the main simulation loop might try to pass it.

```

```

808     """
809     self.state_sys = state # Store it just in case, or do
      nothing if truly not needed.
810
811     def upd_accum_obj(self, observation, action):
812         """
813         Accumulates the running objective for performance evaluation
814         .
815         """
816         self.accum_obj_val += self.run_obj(observation, action) *
      self.sampling_time
817
818     def run_obj(self, observation, action):
819         """
820         Calculates a running objective (cost) for the N_CTRL.
821         This is primarily for logging and visualization, not for the
822         controller's internal logic.
823         """
824         state_error = np.diag(observation - self.observation_target)
825         action=np.diag(action)
826         running_cost = state_error.T @ self.Q @ state_error
827         # action.T @ self.R @ action
828
829         # print(running_cost.shape)
830         return np.linalg.det(running_cost)

```

Listing 8: `Present3WrobotNI.py`

```

1  """
2  Preset: a 3-wheel robot (kinematic model a. k. a. non-holonomic
      integrator).
3
4  """
5
6  import pathlib
7  import os
8  import warnings
9  import csv
10 from datetime import datetime

```

```

11 import matplotlib.animation as animation
12 import matplotlib.pyplot as plt
13 import numpy as np
14
15 import systems
16 import simulator
17 import controllers
18 import loggers
19 import visuals
20 from utilities import on_key_press
21
22 import argparse
23
24 #-----Set up dimensions
25 dim_state = 3
26 dim_input = 2
27 dim_output = dim_state
28 dim_disturb = 0
29
30 dim_R1 = dim_output + dim_input
31 dim_R2 = dim_R1
32
33 description = "Agent-environment preset: a 3-wheel robot (kinematic
    model a.k.a. non-holonomic integrator)."
34
35 parser = argparse.ArgumentParser(description=description)
36
37 parser.add_argument('--ctrl_mode', metavar='ctrl_mode', type=str,
38                     choices=['MPC',
39                             "Nominal",
40                             "lqr"],
41                     default="MPC",
42                     help='Control mode. Currently available: ' +
43                         '----manual: manual constant control specified
44                             by action_manual; ' +
45                         '----nominal: nominal controller, usually used
46                             to benchmark optimal controllers;' +
47                         '----MPC:model-predictive control; ' +

```



```

46         '----RQL: Q-learning actor-critic with Nactor-1
47         roll-outs of running objective; ' +
48         '----SQL: stacked Q-learning; ' +
49         '----RLStabLyap: (experimental!) learning agent
50         with Lyapunov-like stabilizing constraints.')
51 parser.add_argument('--dt', type=float, metavar='dt',
52                     default=0.1,
53                     help='Controller sampling time.' )
54 parser.add_argument('--t1', type=float, metavar='t1',
55                     default=20,
56                     help='Final time of episode.' )
57 parser.add_argument('--Nruns', type=int,
58                     default=1,
59                     help='Number of episodes. Learned parameters are
60                     not reset after an episode.')
61 parser.add_argument('--is_log_data', type=int,
62                     default=1,
63                     help='Flag to log data into a data file. Data
64                     are stored in simdata folder.')
65 parser.add_argument('--is_visualization', type=int,
66                     default=1,
67                     ,
68                     help='Flag to produce graphical output.')
69 parser.add_argument('--is_print_sim_step', type=int,
70                     default=1,
71                     help='Flag to print simulation data into
72                     terminal.')
73 parser.add_argument('--action_manual', type=float,
74                     default=[-5, -3], nargs='+',
75                     help='Manual control action to be fed constant,
76                     system-specific!')
77 parser.add_argument('--Nactor', type=int,
78                     default=15,
79                     help='Horizon length (in steps) for predictive
80                     controllers.')
81 parser.add_argument('--pred_step_size_multiplier', type=float,
82                     default=5.0,
83                     help='Size of each prediction step in seconds is

```

```

77         a pred_step_size_multiplier multiple of
78         controller sampling time dt.')
```

```

79 parser.add_argument('--buffer_size', type=int,
80                     default=25,
81                     help='Size of the buffer (experience replay) for
82                          model estimation, agent learning etc.')
```

```

83 parser.add_argument('--run_obj_struct', type=str,
84                     default='quadratic',
85                     choices=['quadratic',
86                             'biquadratic'],
87                     help='Structure of running objective function.')
```

```

88 parser.add_argument('--Q', type=float, nargs='+',
89                     default=[60,60,60])
90 parser.add_argument('--R', type=float, nargs='+',
91                     default=[1, 1])
92 parser.add_argument('--R1_diag', type=float, nargs='+',
93                     default=[105, 105, 10, 10, 5],
94                     help='Parameter of running objective function.
95                          Must have proper dimension. ' +
96                          'Say, if chi = [observation, action], then a
97                          quadratic running objective reads chi.T diag(
98                          R1) chi, where diag() is transformation of a
99                          vector to a diagonal matrix.')
```

```

100 parser.add_argument('--Qf', type=float, nargs='+',
101                     default=[10,10,10])
102 parser.add_argument('--R2_diag', type=float, nargs='+',
103                     default=[1, 10, 1, 0, 0],
104                     help='Parameter of running objective function .
105                          Must have proper dimension. ' +
106                          'Say, if chi = [observation, action], then a bi-
107                          quadratic running objective reads chi**2.T
108                          diag(R2) chi**2 + chi.T diag(R1) chi, ' +
109                          'where diag() is transformation of a vector to a
110                          diagonal matrix.')
```

```

111 parser.add_argument('--Ncritic', type=int,
112                     default=25,
113                     help='Critic stack size (number of temporal
```

```

        difference terms in critic cost).')
104 parser.add_argument('--gamma', type=float,
105                       default=0.9,
106                       help='Discount factor.')
107 parser.add_argument('--critic_period_multiplier', type=float,
108                       default=1.0,
109                       help='Critic is updated every
        critic_period_multiplier times dt seconds.')
110 parser.add_argument('--critic_struct', type=str,
111                       default='quadratic', choices=['quad-lin',
112                                                       'quadratic',
113                                                       'quad-nomix',
114                                                       'quad-mix',
115                                                       'poly3',
116                                                       'poly4'],
117                       help='Feature structure (critic). Currently
        available: ' +
118                       '----quad-lin: quadratic-linear; ' +
119                       '----quadratic: quadratic; ' +
120                       '----quad-nomix: quadratic, no mixed terms; ' +
121                       '----quad-mix: quadratic, mixed observation-
        action terms (for, say, Q or advantage
        function approximations); ' +
122                       '----poly3: 3-order model, see the code for the
        exact structure; ' +
123                       '----poly4: 4-order model, see the code for the
        exact structure. '
124                       )
125 parser.add_argument('--actor_struct', type=str,
126                       default='quad-nomix', choices=['quad-lin',
127                                                       'quadratic',
128                                                       'quad-nomix'],
129                       help='Feature structure (actor). Currently
        available: ' +
130                       '----quad-lin: quadratic-linear; ' +
131                       '----quadratic: quadratic; ' +
132                       '----quad-nomix: quadratic, no mixed terms.')
133 parser.add_argument('--init_robot_pose_x', type=float,

```

```

134         default=-3.0,
135         help='Initial x-coordinate of the robot pose.')
136 parser.add_argument('--init_robot_pose_y', type=float,
137                     default=-3.0,
138                     help='Initial y-coordinate of the robot pose.')
139 parser.add_argument('--init_robot_pose_theta', type=float,
140                     default=1.57,
141                     help='Initial orientation angle (in radians) of
142                           the robot pose.')
143 parser.add_argument('--distortion_x', type=float,
144                     default=-0.6,
145                     help='X-coordinate of the center of distortion.')
146 parser.add_argument('--distortion_y', type=float,
147                     default=-0.5,
148                     help='Y-coordinate of the center of distortion.')
149 parser.add_argument('--distortion_sigma', type=float,
150                     default=0.1,
151                     help='Standard deviation (sigma) of distortion.')
152 parser.add_argument('--seed', type=int,
153                     default=1,
154                     help='Seed for random number generation.')
155
156 args = parser.parse_args()
157 Nactor = int(os.getenv("NACTOR", args.Nactor))
158 r1_str = os.getenv("R1_DIAG")
159 qf_str = os.getenv("QF")
160 q_str = os.getenv("Q")
161 r_str = os.getenv("R")
162
163 # MPC: R1 (running objective)
164 if r1_str:
165     R1 = np.diag([float(x) for x in r1_str.split()])
166 else:
167     R1 = np.diag(args.R1_diag)

```

```

168 # MPC: Qf (terminal cost)
169 if qf_str:
170     Qf = np.diag([float(x) for x in qf_str.split()])
171 else:
172     Qf = np.diag(args.Qf)
173
174 # LQR: Q matrix (state error)
175 if q_str:
176     Q = np.diag([float(x) for x in q_str.split()])
177 else:
178     Q = np.diag(args.Q)
179
180 # LQR: R matrix (control effort)
181 if r_str:
182     R = np.diag([float(x) for x in r_str.split()])
183 else:
184     R = np.diag(args.R)
185
186 seed=args.seed
187 print(seed)
188
189 xdistortion_x = args.distortion_x
190 ydistortion_y = args.distortion_y
191 distortion_sigma = args.distortion_sigma
192
193 x = args.init_robot_pose_x
194 y = args.init_robot_pose_y
195 theta = args.init_robot_pose_theta
196
197 while theta > np.pi:
198     theta -= 2 * np.pi
199 while theta < -np.pi:
200     theta += 2 * np.pi
201
202 state_init = np.array([x, y, theta])
203
204 args.action_manual = np.array(args.action_manual)
205

```

```

206 pred_step_size = args.dt * args.pred_step_size_multiplier
207 critic_period = args.dt * args.critic_period_multiplier
208
209 Nactor = int(os.getenv("NACTOR", args.Nactor))
210
211 r1_str = os.getenv("R1_DIAG")
212 qf_str = os.getenv("QF")
213
214 if r1_str:
215     R1 = np.diag([float(x) for x in r1_str.split()])
216 else:
217     R1 = np.diag(np.array(args.R1_diag))
218
219 if qf_str:
220     Qf = np.diag([float(x) for x in qf_str.split()])
221 else:
222     Qf = np.diag(np.array(args.Qf))
223
224 assert args.t1 > args.dt > 0.0
225 assert state_init.size == dim_state
226
227 globals().update(vars(args))
228
229 #-----Fixed settings
230 is_disturb = 0
231 is_dyn_ctrl = 0
232
233 t0 = 0
234
235 action_init = 0 * np.ones(dim_input)
236
237 # Solver
238 atol = 1e-3
239 rtol = 1e-2
240
241 # xy-plane
242 xMin = -4#-1.2
243 xMax = 2

```

```

244 yMin = -4#-1.2
245 yMax = 2
246
247 # Control constraints
248 v_min = -0.22 *10
249 v_max = 0.22 *10
250 omega_min = -2.84
251 omega_max = 2.84
252
253 ctrl_bnds=np.array([[v_min, v_max], [omega_min, omega_max]])
254 # ctrl_bnds=np.zeros((2,2))
255
256 #-----Initialization : : system
257 my_sys = systems.Sys3WRobotNI(sys_type="diff_eqn",
258                                dim_state=dim_state,
259                                dim_input=dim_input,
260                                dim_output=dim_output,
261                                dim_disturb=dim_disturb,
262                                pars=[],
263                                ctrl_bnds=ctrl_bnds,
264                                is_dyn_ctrl=is_dyn_ctrl,
265                                is_disturb=is_disturb,
266                                pars_disturb=[])
267
268 observation_init = my_sys.out(state_init)
269
270 xCoord0 = state_init[0]
271 yCoord0 = state_init[1]
272 alpha0 = state_init[2]
273 alpha_deg_0 = alpha0/2*np.pi
274
275 #-----Initialization : : model
276
277 #-----Initialization : :
278     controller
279
280 target_x=0.0
281 target_y=0.0

```

```

281 target_theta=0.0
282 dt=0.1
283 # my_ctrl_nominal = controllers.N_CTRL(k_rho=1.0, k_alpha=4.0,
      k_beta=-1.5, # Example gains, vary for Experiment A
284 #         target_x=target_x, target_y=target_y, target_theta=
      target_theta,
285 #         sampling_time=dt,
286 #         dim_input=dim_input, dim_output=dim_output
287 #     )
288
289
290 # Nominal forward velocity for linearization
291 v_nom =0.001
292
293 # Linearized discrete system (unit time step)
294 A = np.array([
295     [1, 0, -v_nom * dt*np.sin(0)],
296     [0, 1,  v_nom *dt* np.cos(0)],
297     [0, 0, 1]
298 ])
299
300 # print(A)
301
302 B = np.array([
303     [dt*np.cos(0), 0],
304     [dt*np.sin(0), 0],
305     [0, dt]
306 ])
307
308 # =====
309 # 2. Define cost matrices
310 # =====
311
312 # Q penalizes state error (x, y, theta)
313 Q = np.diag(Q)
314
315 # R penalizes control effort (v, omega)
316 R = np.diag(R)

```



```

317
318 print(Q)
319 print(R)
320 my_ctrl_lqr=controllers.LQR(A=A,B=B,Q=Q,R=R,obs_target=np.array
    ([0.0,0.0,0.0]),sampling_time=dt)
321
322
323 # Predictive optimal controller
324 my_ctrl_opt_pred = controllers.ControllerOptimalPredictive(dim_input
    ,
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
    dim_output,
    ctrl_mode,
    ctrl_bnds = ctrl_bnds,
    action_init = [],
    t0 = t0,
    sampling_time = dt,
    Nactor = Nactor,
    pred_step_size =
        pred_step_size,
    sys_rhs = my_sys.
        _state_dyn,
    sys_out = my_sys.out,
    state_sys = state_init,
    buffer_size = buffer_size
    ,
    gamma = gamma,
    Ncritic = Ncritic,
    critic_period =
        critic_period,
    critic_struct =
        critic_struct,
    run_obj_struct =
        run_obj_struct,
    run_obj_pars = [R1,Qf],
    observation_target =
        [0.0,0.0,0.0],
    state_init=state_init,
    obstacle=[xdistortion_x,

```

```

346         ydistortion_y,
347         distortion_signal],
348         seed=seed)
349 my_ctrl_benchm = my_ctrl_opt_pred
350
351 #-----Initialization : :
352     simulator
353 my_simulator = simulator.Simulator(sys_type = "diff_eqn",
354                                     closed_loop_rhs = my_sys.
355                                         closed_loop_rhs,
356                                     sys_out = my_sys.out,
357                                     state_init = state_init,
358                                     disturb_init = [],
359                                     action_init = action_init,
360                                     t0 = t0,
361                                     t1 = t1,
362                                     dt = dt,
363                                     max_step = dt,
364                                     first_step = 1e-4,
365                                     atol = atol,
366                                     rtol = rtol,
367                                     is_disturb = is_disturb,
368                                     is_dyn_ctrl = is_dyn_ctrl)
369
370 #-----Initialization : : logger
371 date = datetime.now().strftime("%Y-%m-%d")
372 time = datetime.now().strftime("%Hh%Mm%Ss")
373 datafiles = [None] * Nruns
374
375 data_folder = 'simdata/' + ctrl_mode + "/Init_angle_{}_seed_{}_
376               _Nactor{}".format(str(state_init[2]), seed, Nactor)
377
378 if is_log_data:
379     pathlib.Path(data_folder).mkdir(parents=True, exist_ok=True)
380
381 for k in range(0, Nruns):

```

```

379     datafiles[k] = data_folder + '/' + my_sys.name + '_' + ctrl_mode
        + '_' + date + '_' + time + '__run{run:02d}.csv'.format(run=
            k+1)

380
381     if is_log_data:
382         print('Logging data to:      ' + datafiles[k])
383
384         with open(datafiles[k], 'w', newline='') as outfile:
385             writer = csv.writer(outfile)
386             writer.writerow(['System', my_sys.name ] )
387             writer.writerow(['Controller', ctrl_mode ] )
388             writer.writerow(['dt', str(dt) ] )
389             writer.writerow(['state_init', str(state_init) ] )
390             writer.writerow(['Nactor', str(Nactor) ] )
391             writer.writerow(['pred_step_size_multiplier', str(
                pred_step_size_multiplier) ] )
392             writer.writerow(['buffer_size', str(buffer_size) ] )
393             writer.writerow(['run_obj_struct', str(run_obj_struct) ]
                )
394             writer.writerow(['R1_diag', str(R1_diag) ] )
395             writer.writerow(['R2_diag', str(R2_diag) ] )
396             writer.writerow(['Ncritic', str(Ncritic) ] )
397             writer.writerow(['gamma', str(gamma) ] )
398             writer.writerow(['critic_period_multiplier', str(
                critic_period_multiplier) ] )
399             writer.writerow(['critic_struct', str(critic_struct) ] )
400             writer.writerow(['actor_struct', str(actor_struct) ] )
401             writer.writerow(['t [s]', 'x [m]', 'y [m]', 'alpha [rad]
                ', 'run_obj', 'accum_obj', 'v [m/s]', 'omega [rad/s]'
                ] )
402
403     # Do not display annoying warnings when print is on
404     if is_print_sim_step:
405         warnings.filterwarnings('ignore')
406
407     k_rho = 2.0
408     k_alpha = 5.0
409     k_beta = -1.5

```

```

410
411
412 my_logger = loggers.Logger3WRobotNI()
413 my_ctrl_nominal=controllers.N_CTRL(k_rho=k_rho, k_alpha=k_alpha,
414     k_beta=k_beta, # Example gains, vary for Experiment A
415     target_x=0.0, target_y=0.0, target_theta=0.0,
416     sampling_time=dt,
417     dim_input=dim_input, dim_output=dim_output)
418
419 # my_ctrl_lqr=controllers.LQR(A=np.diag([2.0,3.0,4.0]),B=np.diag
420     ([2.0,3.0]),Q=np.diag([1.0,2.0,3.0]),R=np.diag([3.0,4.0]),
421     obs_target=np.array([0.0,0.0,0.0]))
422
423 #-----Main loop
424 state_full_init = my_simulator.state_full
425
426 if is_visualization:
427     my_animator = visuals.Animator3WRobotNI(objects=(my_simulator,
428         my_sys,
429         my_ctrl_nominal,
430         my_ctrl_benchm,
431         my_ctrl_lqr,
432         datafiles,
433         controllers,
434         ctrl_selector,
435         my_logger),
436         pars=(state_init,
437             action_init,
438             t0,
439             t1,
440             state_full_init,
441             xMin,
442             xMax,
443             yMin,
444             yMax,
445             ctrl_mode,

```

```

442         action_manual,
443         v_min,
444         omega_min,
445         v_max,
446         omega_max,
447         Nruns,
448         is_print_sim_step,
            is_log_data,
            0, [], [
            xdistortion_x,
            ydistortion_y,
            distortion_sigma
            ]))

449
450     anm = animation.FuncAnimation(my_animator.fig_sim,
451                                   my_animator.animate,
452                                   init_func=my_animator.init_anim,
453                                   blit=False, interval=dt/1e6,
                                   repeat=True)

454     print("ALSO GOOD")
455     my_animator.get_anm(anm)
456
457     cId = my_animator.fig_sim.canvas.mpl_connect('key_press_event',
458           lambda event: on_key_press(event, anm))
459
460
461     anm.running = True
462
463     my_animator.fig_sim.tight_layout()
464
465     plt.show()
466
467 else:
468     run_curr = 1
469     datafile = datafiles[0]
470
471     while True:
472
473         my_simulator.sim_step()

```

```

472     t, state, observation, state_full = my_simulator.
473         get_sim_step_data()
474
475     action = controllers.ctrl_selector(t, observation,
476         action_manual, my_ctrl_nominal, my_ctrl_benchm,
477         my_ctrl_lqr, ctrl_mode)
478     print("action: ", action, ctrl_mode)
479
480     my_sys.receive_action(action)
481     my_ctrl_benchm.receive_sys_state(my_sys._state)
482     my_ctrl_benchm.upd_accum_obj(observation, action)
483
484     xCoord = state_full[0]
485     yCoord = state_full[1]
486     alpha = state_full[2]
487     print("sample")
488
489     run_obj = my_ctrl_benchm.run_obj(observation, action)
490     accum_obj = my_ctrl_benchm.accum_obj_val
491
492     # count_CALF = my_ctrl_benchm.D_count()
493     # count_N_CTRL = my_ctrl_benchm.get_N_CTRL_count()
494
495     if is_print_sim_step:
496         my_logger.print_sim_step(t, xCoord, yCoord, alpha,
497             run_obj, accum_obj, action)
498
499     if is_log_data:
500         my_logger.log_data_row(datafile, t, xCoord, yCoord,
501             alpha, run_obj, accum_obj, action)
502
503     if t >= t1 or np.linalg.norm(observation[:2]) < 0.2:
504
505         # Reset simulator
506         my_simulator.reset()

```

```

505         if ctrl_mode != 'MPC':
506             my_ctrl_benchm.reset(t0)
507         elif ctrl_mode=='lqr':
508             my_ctrl_lqr.reset(t0)
509         else:
510             my_ctrl_nominal.reset(t0)
511
512         accum_obj = 0
513
514         if is_print_sim_step:
515             print('.....Run {run
                    :2d} done.....'.
                    format(run = run_curr))
516
517         run_curr += 1
518
519         if run_curr > Nruns:
520             plt.close('all')
521             break
522
523         if is_log_data:
524             datafile = datafiles[run_curr-1]

```

Listing 9: Experiment_{ANominal}.py

```

1  # IMPORT REQUIRED LIBRARIES
2  import pandas as pd                # For reading and handling CSV
   files
3  import matplotlib.pyplot as plt    # For plotting data
4  import numpy as np                # For numerical operations
5
6  # DEFINE GAIN SETS
7  gain_sets = [
8      {"k_rho": 2.0, "k_alpha": 5.0, "k_beta": -1.5}, # k_rho > 0,
   k_alpha - k_rho > 0, k_beta < 0 (Controller Law for Control
   Design)
9      {"k_rho": 1.0, "k_alpha": 4.0, "k_beta": -1.0}, # k_rho > 0,
   k_alpha - k_rho > 0, k_beta < 0 (Controller Law for Control
   Design)

```

```

10     {"k_rho": 0.5, "k_alpha": 3.0, "k_beta": -0.5}, # k_rho > 0,
        k_alpha - k_rho > 0, k_beta < 0 (Controller Law for Control
        Design)
11 ]
12
13 # DEFINE FILE PATHS FOR CSV FILES GENERATED FROM SIMULATION
14 csv_files = ['simdata/Nominal/Init_angle_1.57_seed_1_Nactor_10/3
        wrobotNI_Nominal_2025-06-22_17h52m11s__run01.csv', '/home/
        gouransh/Desktop/Control Assignment/rcognita-edu-main/simdata/
        Nominal/Init_angle_1.57_seed_1_Nactor_10/3wrobotNI_Nominal_2025
        -06-22_17h32m42s__run01.csv', '/home/gouransh/Desktop/Control
        Assignment/rcognita-edu-main/simdata/Nominal/Init_angle_1.57
        _seed_1_Nactor_10/3wrobotNI_Nominal_2025-06-22_17h35m33s__run01.
        csv']
15
16 # DEFINE COLORS FOR EACH RUN
17 colors = ['red', 'blue', 'green']
18
19 # FUNCTION TO READ CSV WHILE SKIPPING METADATA COMMENTS
20 def smart_read_csv(file_path):
21     with open(file_path, 'r') as f:
22         lines = f.readlines()
23         # Identify the header line (starts with "t [s],")
24         for i, line in enumerate(lines):
25             if line.startswith("t [s],"):
26                 header_index = i
27                 break
28         return pd.read_csv(file_path, skiprows=header_index)
29
30 # PLOT 1: ROBOT TRAJECTORY VS REFERENCE (x vs y)
31 plt.figure(figsize=(8, 6))
32 for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
33     data = smart_read_csv(file)
34     data.columns = [col.strip() for col in data.columns]
35
36     # Plot the actual robot trajectory
37     plt.plot(data['x [m]'], data['y [m]'], label=f'Run {i}', color=
        color)

```



```

38
39 plt.xlabel('x [m]')
40 plt.ylabel('y [m]')
41 plt.title('Robot Trajectories (Kinematic Controller)')
42 plt.legend()
43 plt.grid(True)
44 plt.axis('equal') # x-y plot, so equal scaling makes sense
45 plt.savefig('simdata/Robot_Trajectories_Kinematics.png')
46 plt.close()
47
48 # PLOT 2: LINEAR VELOCITY v(t)
49 plt.figure(figsize=(8, 6))
50 for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
51     data = smart_read_csv(file)
52     data.columns = [col.strip() for col in data.columns]
53
54     # Plot v [m/s] over time
55     plt.plot(data['t [s]'], data['v [m/s]'], label=f'Run {i}', color
              =color)
56
57 plt.xlabel('t [s]')
58 plt.ylabel('v [m/s]')
59 plt.title('Linear Velocity Over Time')
60 plt.legend()
61 plt.grid(True)
62 plt.savefig('simdata/Linear_Velocity_Kinematics.png')
63 plt.close()
64
65 # PLOT 3: ANGULAR VELOCITY omega(t)
66 plt.figure(figsize=(8, 6))
67 for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
68     data = smart_read_csv(file)
69     data.columns = [col.strip() for col in data.columns]
70
71     # Plot omega [rad/s] over time
72     plt.plot(data['t [s]'], data['omega [rad/s]'], label=f'Run {i}',
              color=color)
73

```

```

74 plt.xlabel('t [s]')
75 plt.ylabel('omega [rad/s]')
76 plt.title('Angular Velocity Over Time')
77 plt.legend()
78 plt.grid(True)
79 plt.savefig('simdata/Angular_Velocity_Kinematics.png')
80 plt.close()
81
82 # PLOT 4: TRACKING ERROR VS TIME (Using fixed goal)
83 plt.figure(figsize=(8, 6))
84
85 # Define your fixed goal location (set your actual goal here)
86 x_goal, y_goal = 0.0, 0.0
87
88 for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
89     data = smart_read_csv(file)
90     data.columns = [col.strip() for col in data.columns]
91
92     # Compute distance to fixed goal at every time step
93     error = np.sqrt((data['x [m]'] - x_goal)**2 + (data['y [m]'] -
94         y_goal)**2)
95
96     # Plot tracking error over time
97     plt.plot(data['t [s]'], error, label=f'Run {i}', color=color)
98
99 plt.xlabel('t [s]')
100 plt.ylabel('Tracking Error [m]')
101 plt.title('Tracking Error Over Time')
102 plt.legend()
103 plt.grid(True)
104 plt.savefig('simdata/Tracking_Error_Over_Time.png')
105 plt.close()
106 # PRINT SUCCESS MESSAGE
107 print("Plots Saved:")
108 print("    Robot_Trajectories_Kinematics.png")
109 print("    Linear_Velocity_Kinematics.png")
110 print("    Angular_Velocity_Kinematics.png")
111 print("    Tracking_Error_Over_Time.png")

```

Listing 10: Experiment_{BLQR}.py

```

1 # IMPORT REQUIRED LIBRARIES
2 import os
3 import subprocess
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import pandas as pd
7 from datetime import datetime
8 import glob
9
10 # DEFINE COST MATRICES Q AND R FOR TESTING
11 cost_sets = [
12     {"Q": [10.0, 12.0, 8.0], "R": [1.0, 1.0]},
13     {"Q": [100.0, 100.0, 50.0], "R": [0.1, 0.1]},
14     {"Q": [5.0, 5.0, 2.0], "R": [10.0, 10.0]}
15 ]
16
17 # LOG FOLDER SETUP
18 log_folder = "simdata/lqr/Init_angle_1.57_seed_1_Nactor_10/"
19 os.makedirs(log_folder, exist_ok=True)
20
21 # RUN SIMULATIONS WITH VARYING Q AND R
22 for i, cost in enumerate(cost_sets):
23     print(f"\n Running LQR Simulation {i+1} with Q={cost['Q']} and R
24           ={{cost['R']}}")
25
26     os.environ["Q_VALS"] = " ".join(str(x) for x in cost["Q"])
27     os.environ["R_VALS"] = " ".join(str(x) for x in cost["R"])
28     print("Environment variables set:", os.environ["Q_VALS"], os.
29           environ["R_VALS"])
30
31     subprocess.run([
32         "python3", "PRESET_3wrobot_NI.py",
33         "--ctrl_mode", "lqr",
34         "--Nruns", "1",
35         "--t1", "20",
36         "--is_visualization", "0",
37         "--is_log_data", "1",

```

```

36         "--Q", *map(str, cost["Q"]),
37         "--R", *map(str, cost["R"]),
38         "--v_max", "1.0",                # Actuator limit: max linear
            velocity
39         "--omega_max", "1.0"            # Actuator limit: max
            angular velocity
40     ])
41
42 # FIND LATEST LQR LOG FILES
43 def find_latest_lqr_csvs(folder, prefix="3wrobotNI_lqr_", run_suffix
    = "__run01.csv", count=3):
44     pattern = os.path.join(folder, f"{prefix}*{run_suffix}")
45     all_files = glob.glob(pattern)
46     all_files.sort(key=os.path.getmtime, reverse=True)
47     return all_files[:count]
48
49 print("\n Plotting Results")
50
51 # READ THE CSV FILES
52 csvs = find_latest_lqr_csvs(log_folder)
53
54 if not csvs:
55     print("No CSV files found for plotting.")
56     exit()
57
58 # FUNCTION TO READ CSV SKIPPING HEADER LINES BEFORE ACTUAL DATA
59 def smart_read_csv(file_path):
60     with open(file_path, 'r') as f:
61         lines = f.readlines()
62         for i, line in enumerate(lines):
63             if line.startswith("t [s],"):
64                 header_index = i
65                 break
66         return pd.read_csv(file_path, skiprows=header_index)
67
68 # READ AND STORE DATAFRAMES FROM CSV FILES
69 dfs = [smart_read_csv(f) for f in csvs]
70 print(f"Found {len(dfs)} CSV files for plotting.")

```

```

71
72 # DEFINE COLORS FOR EACH RUN
73 colors = ['b', 'g', 'r']
74
75 # PLOT 1: TRAJECTORIES (x vs y)
76 plt.figure(figsize=(10, 6))
77 for i, df in enumerate(dfs):
78     plt.plot(df['x [m]'], df['y [m]'], label=f"Run {i+1} - Q={
79         cost_sets[i]['Q']}, R={cost_sets[i]['R']}", color=colors[i])
80 plt.title("LQR: Trajectory (x vs y)")
81 plt.xlabel("x [m]")
82 plt.ylabel("y [m]")
83 plt.legend()
84 plt.grid(True)
85 plt.tight_layout()
86 plt.savefig("simdata/lqr/Trajectory_Plot.png")
87 plt.close()
88
89 # PLOT 2: TRACKING ERROR VS TIME (distance to origin)
90 plt.figure(figsize=(10, 6))
91 for i, df in enumerate(dfs):
92     error = np.sqrt(df['x [m]']**2 + df['y [m]']**2)
93     plt.plot(df['t [s]'], error, label=f"Run {i+1}", color=colors[i
94 ])
95
96     # Highlight runs with large steady-state error
97     if error.iloc[-1] > 0.5:
98         print(f" Run {i+1} may not converge properly: final error =
99             {error.iloc[-1]:.2f} m")
100
101 plt.title("LQR: Tracking Error vs Time")
102 plt.xlabel("Time [s]")
103 plt.ylabel("Position Error [m]")
104 plt.legend()
105 plt.grid(True)
106 plt.tight_layout()
107 plt.savefig("simdata/lqr/Tracking_Error_Plot.png")
108 plt.close()

```

```

106
107 # PLOT 3: CONTROL INPUTS (v and omega vs time)
108 plt.figure(figsize=(10, 6))
109 for i, df in enumerate(dfs):
110     plt.plot(df['t [s]'], df['v [m/s]'], label=f"v - Run {i+1}",
111             color=colors[i])
112     plt.plot(df['t [s]'], df['omega [rad/s]'], '--', label=f" -
113             Run {i+1}", color=colors[i])
114 plt.title("LQR: Control Inputs Over Time")
115 plt.xlabel("Time [s]")
116 plt.ylabel("Input Values")
117 plt.legend()
118 plt.grid(True)
119 plt.tight_layout()
120 plt.savefig("simdata/lqr/Control_Inputs_Plot.png")
121 plt.close()
122 # PRINT SUCCESS MESSAGE
123 print(" Plots Saved:")
124 print("     Trajectory_Plot.png")
125 print("     Tracking_Error_Plot.png")
126 print("     Control_Inputs_Plot.png")

```

Listing 11: Experiment_{CMPC}.py

```

1 # IMPORT REQUIRED LIBRARIES
2 import os
3 import subprocess
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import pandas as pd
7 import glob
8
9 # DEFINE MPC CONFIGURATION SET WITH VARIATIONS
10 mpc_configs = [
11     {"Nactor": 5, "R1_diag": [100, 100, 10, 10, 5], "Qf": [0, 0,
12     0]}, # No terminal cost, short horizon
13     {"Nactor": 15, "R1_diag": [100, 100, 10, 10, 5], "Qf": [100,
14     100, 10]}, # Long horizon + terminal cost
15     {"Nactor": 10, "R1_diag": [300, 300, 30, 5, 1], "Qf": [150, 150,

```

```

14         15]],      # Medium horizon, high R weight
15     ]
16
17 # LOG FOLDER SETUP
18 log_folder = "simdata/MPC/"
19 os.makedirs(log_folder, exist_ok=True)
20
21 # RUN SIMULATIONS USING SUBPROCESS CALL
22 for i, config in enumerate(mpc_configs):
23     print(f"\n Running MPC Simulation {i+1} with Nactor={config['
24         Nactor']}, Qf={config['Qf']}")
25
26     # EXPORT VARIABLES (IN CASE THE CONTROLLER READS FROM ENV)
27     os.environ["NACTOR"] = str(config["Nactor"])
28     os.environ["R1_DIAG"] = " ".join(str(x) for x in config["R1_diag
29         "])
30     os.environ["QF"] = " ".join(str(x) for x in config["Qf"])
31
32     # RUN SIMULATION SCRIPT
33     subprocess.run([
34         "python3", "PRESET_3wrobot_NI.py",
35         "--ctrl_mode", "MPC",
36         "--Nruns", "1",
37         "--t1", "20",
38         "--is_visualization", "0",
39         "--is_log_data", "1",
40         "--Nactor", str(config["Nactor"]),
41         "--R1_diag", *map(str, config["R1_diag"]),
42         "--Qf", *map(str, config["Qf"]),
43     ])
44
45 # GET LATEST LOG FILES FROM EACH RUN
46 def get_latest_csv_from_each_subfolder(main_folder, pattern="3
47     wrobotNI_MPC_*_run01.csv"):
48     """Find the latest CSV file matching pattern in each subfolder.
49     """
50     csv_paths = []
51     subfolders = [os.path.join(main_folder, d) for d in os.listdir(

```

```

    main_folder)
47         if os.path.isdir(os.path.join(main_folder, d))]
48     subfolders.sort() # Sort for consistency
49     subfolders = subfolders[:3] # Limit to 3 experiments
50
51     for folder in subfolders:
52         match = glob.glob(os.path.join(folder, pattern))
53         if match:
54             match.sort(key=os.path.getmtime, reverse=True)
55             csv_paths.append(match[0])
56         else:
57             print(f"No matching CSV in: {folder}")
58
59     return csv_paths
60
61 csvs = get_latest_csv_from_each_subfolder(log_folder)
62
63 if not csvs:
64     print("No CSV files found for plotting.")
65     exit()
66
67 # DEFINE HELPER TO READ CSV DATA SKIPPING COMMENT LINES
68 def smart_read_csv(file_path):
69     with open(file_path, 'r') as f:
70         lines = f.readlines()
71         header_index = next(i for i, line in enumerate(lines) if line.
72                             startswith("#t [s],"))
73         return pd.read_csv(file_path, skiprows=header_index)
74
75 # READ DATA FROM CSV FILES
76 dfs = [smart_read_csv(f) for f in csvs]
77
78 # DEFINE COLORS FOR EACH RUN
79 colors = ['b', 'g', 'r']
80
81 # PLOT 1: PLOT TRAJECTORY (x vs y)
82 plt.figure(figsize=(8, 6))
83 for i, df in enumerate(dfs):

```



```

83     plt.plot(df['x [m]'], df['y [m]'], label=f"Run {i+1} - N={
        mpc_configs[i]['Nactor']}", color=colors[i])
84 plt.title("MPC Trajectory (x vs y)")
85 plt.xlabel("x [m]")
86 plt.ylabel("y [m]")
87 plt.legend()
88 plt.grid(True)
89 plt.tight_layout()
90 plt.savefig("simdata/MPC/MPC_Trajectory.png")
91 plt.close()
92
93 # PLOT 2: TRACKING ERROR VS TIME
94 plt.figure(figsize=(8, 6))
95 for i, df in enumerate(dfs):
96     error = np.sqrt(df['x [m]']**2 + df['y [m]']**2)
97     plt.plot(df['t [s]'], error, label=f"Run {i+1}", color=colors[i
        ])
98 plt.title("MPC Tracking Error vs Time")
99 plt.xlabel("Time [s]")
100 plt.ylabel("Position Error [m]")
101 plt.legend()
102 plt.grid(True)
103 plt.tight_layout()
104 plt.savefig("simdata/MPC/MPC_Tracking_Error.png")
105 plt.close()
106
107 # PLOT 3: CONTROL INPUTS (V) VS OMEGA OVER TIME
108 plt.figure(figsize=(10, 6))
109 for i, df in enumerate(dfs):
110     plt.plot(df['t [s]'], df['v [m/s]'], label=f"v - Run {i+1}",
        color=colors[i])
111     plt.plot(df['t [s]'], df['omega [rad/s]'], '--', label=f" -
        Run {i+1}", color=colors[i])
112 plt.title("MPC Control Inputs (v and ) vs Time")
113 plt.xlabel("Time [s]")
114 plt.ylabel("Control Inputs")
115 plt.legend()
116 plt.grid(True)

```

```

117 plt.tight_layout()
118 plt.savefig("simdata/MPC/MPC_Control_Inputs.png")
119 plt.close()
120
121 # PLOT 4: ACCUMULATED COST OVER TIME
122 plt.figure(figsize=(8, 6))
123 for i, df in enumerate(dfs):
124     if 'accum_obj' in df.columns:
125         plt.plot(df['t [s]'], df['accum_obj'], label=f"Run {i+1}",
126                 color=colors[i])
127 plt.title("MPC Accumulated Cost vs Time")
128 plt.xlabel("Time [s]")
129 plt.ylabel("Accumulated Cost")
130 plt.legend()
131 plt.grid(True)
132 plt.tight_layout()
133 plt.savefig("simdata/MPC/MPC_Accumulated_Cost.png")
134 plt.close()
135
136 print("Plots saved:")
137 print("    MPC_Trajectory.png")
138 print("    MPC_Tracking_Error.png")
139 print("    MPC_Control_Inputs.png")
140 print("    MPC_Accumulated_Cost.png")

```