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

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:

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

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:

$$v = \kappa_\rho e_\rho, \tag{2}$$

$$\omega = \kappa_\alpha e_\alpha + \kappa_\beta e_\beta, \tag{3}$$

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:

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

2.2.3 MPC Controller

Model Predictive Control solves the finite-horizon optimization:

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

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:

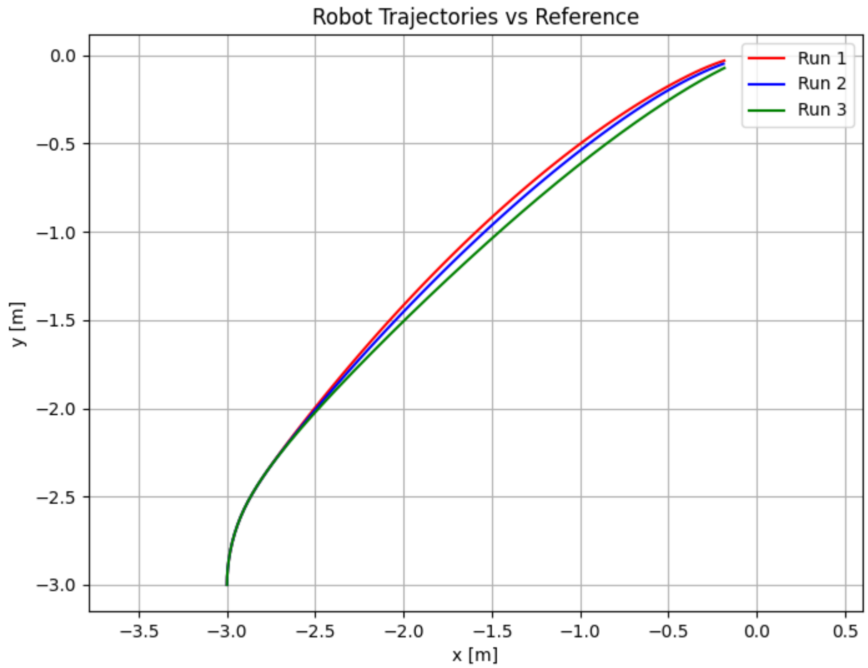

Figure 1: Trajectory Tracking Comparison for Kinematic Controller

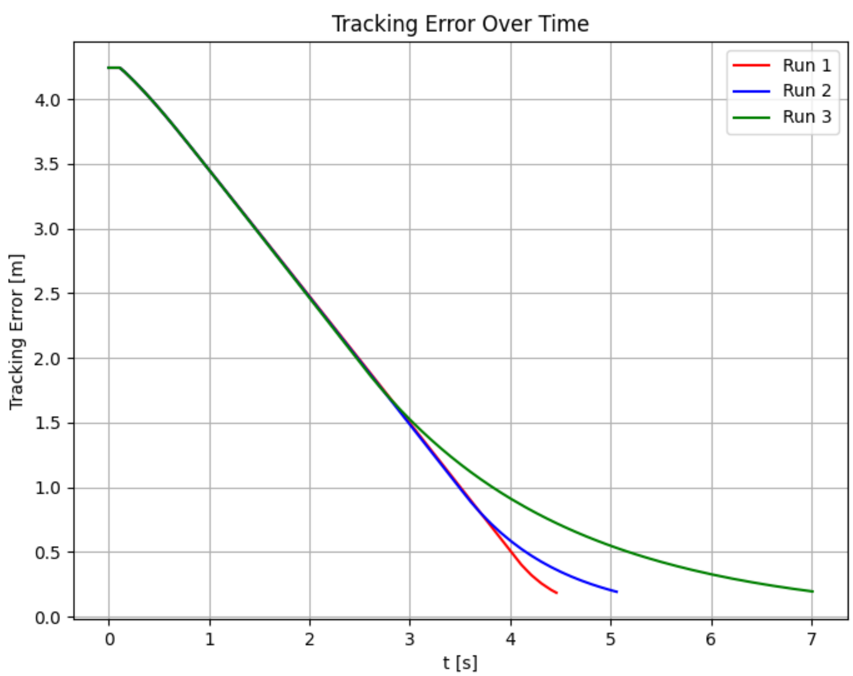

Figure 2: Tracking Errors Over Time

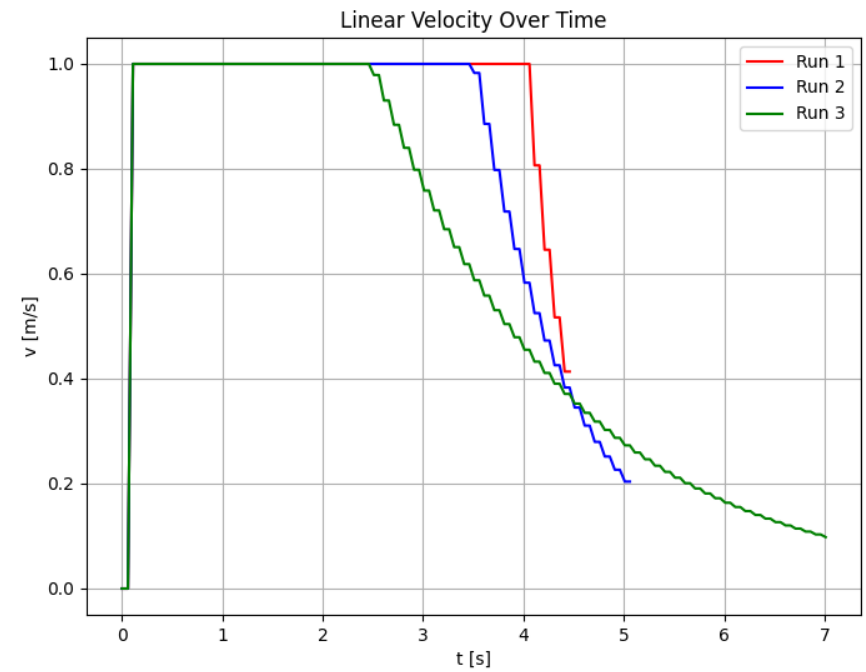

Figure 3: Linear velocity v Over Time for Kinematic Controller

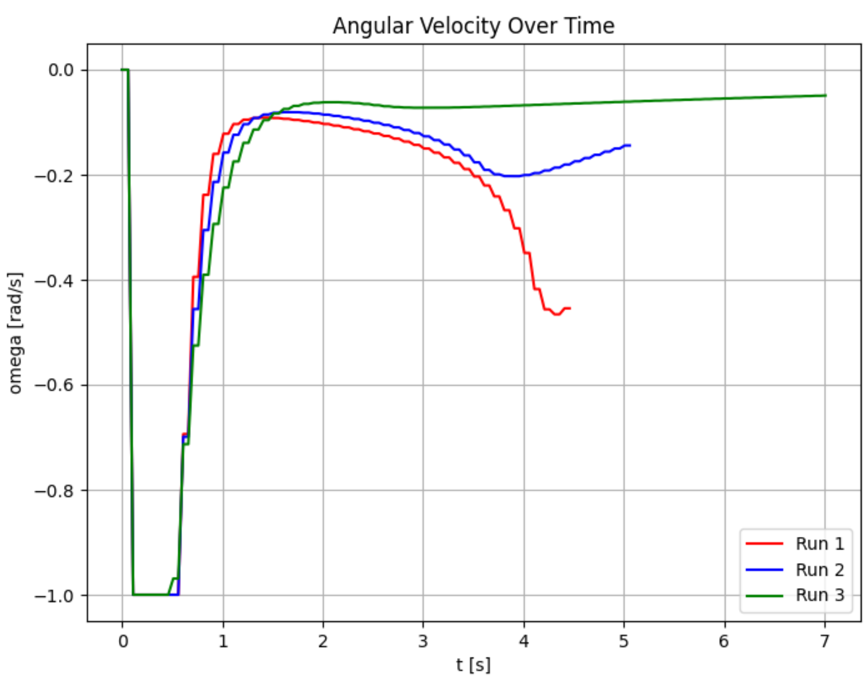

Figure 4: Angular velocity ω Over Time for Kinematic Controller

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:

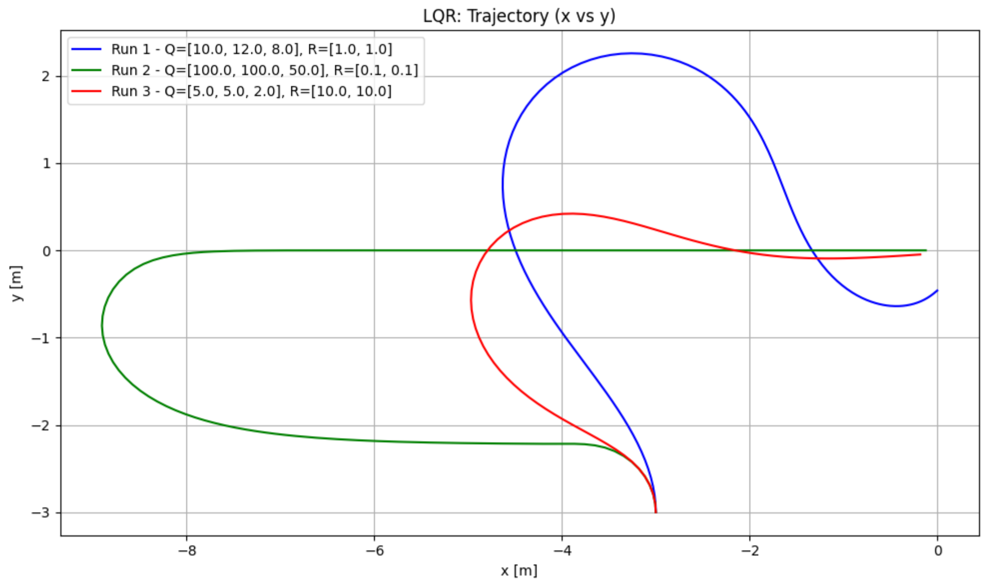

Figure 5: Trajectory Plot

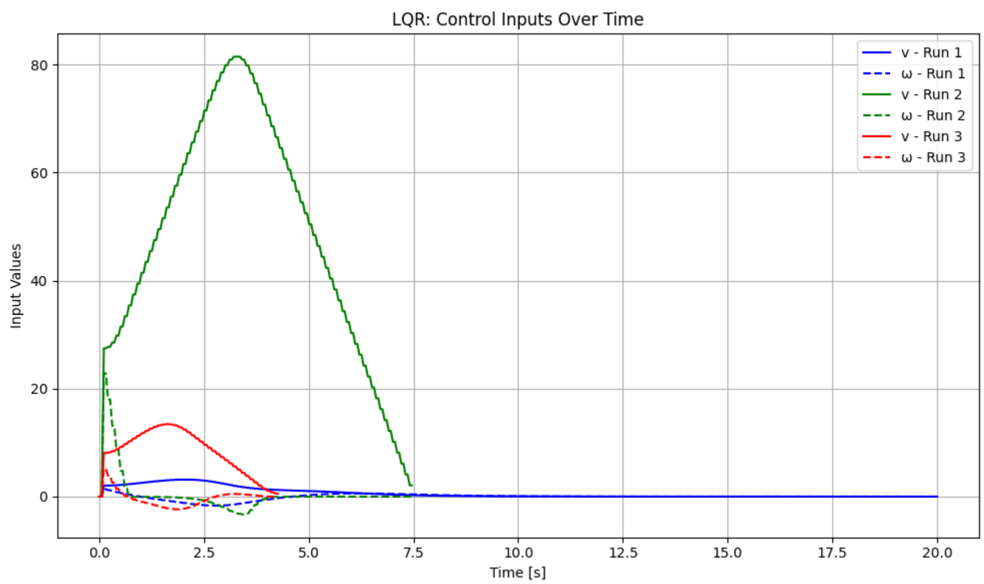

Figure 6: Control Inputs Plot

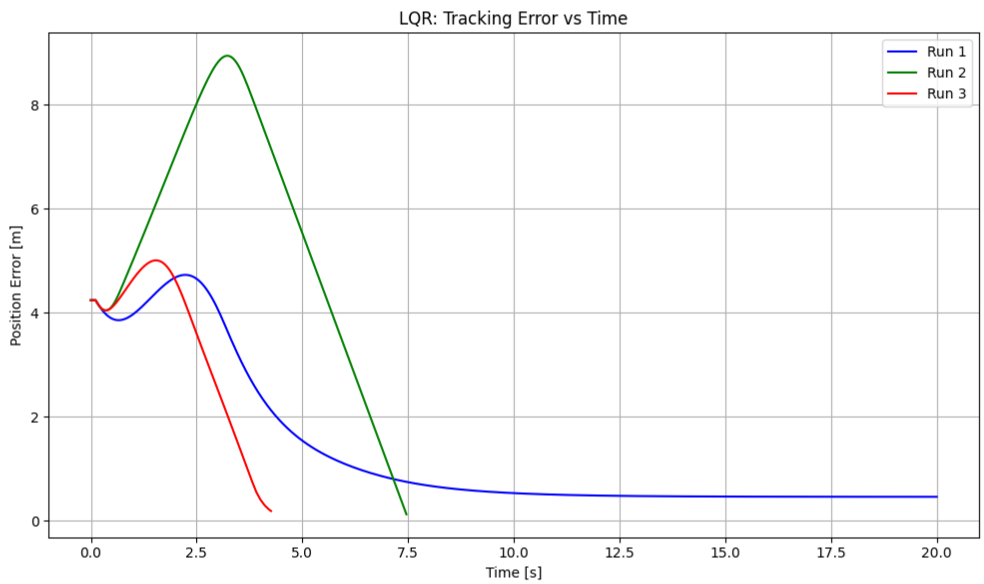

Figure 7: Tracking Error Plot

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 (≈ 0.2m).

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 < 0.05m.
- Heavy-R settles in 12s, final error ≈ 0.15m.

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:

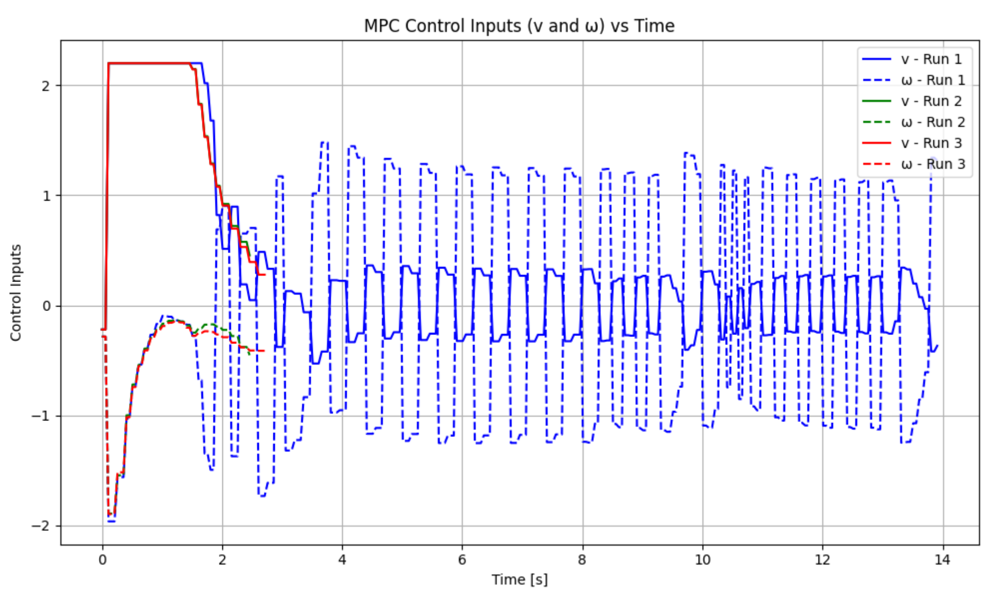

Figure 8: Control Inputs

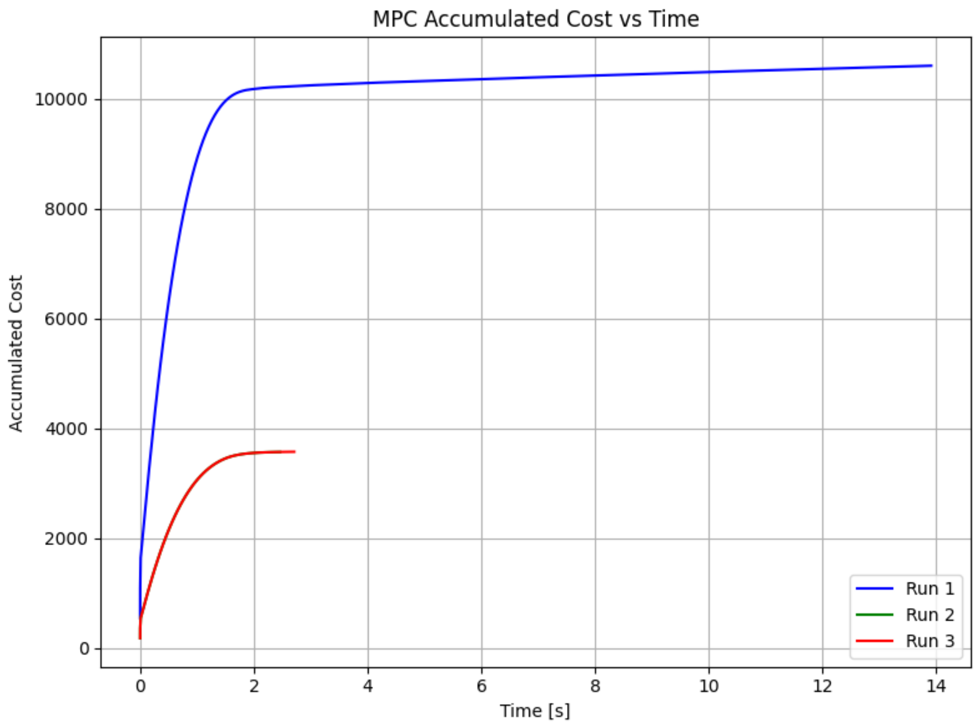

Figure 9: Accumulated Cost

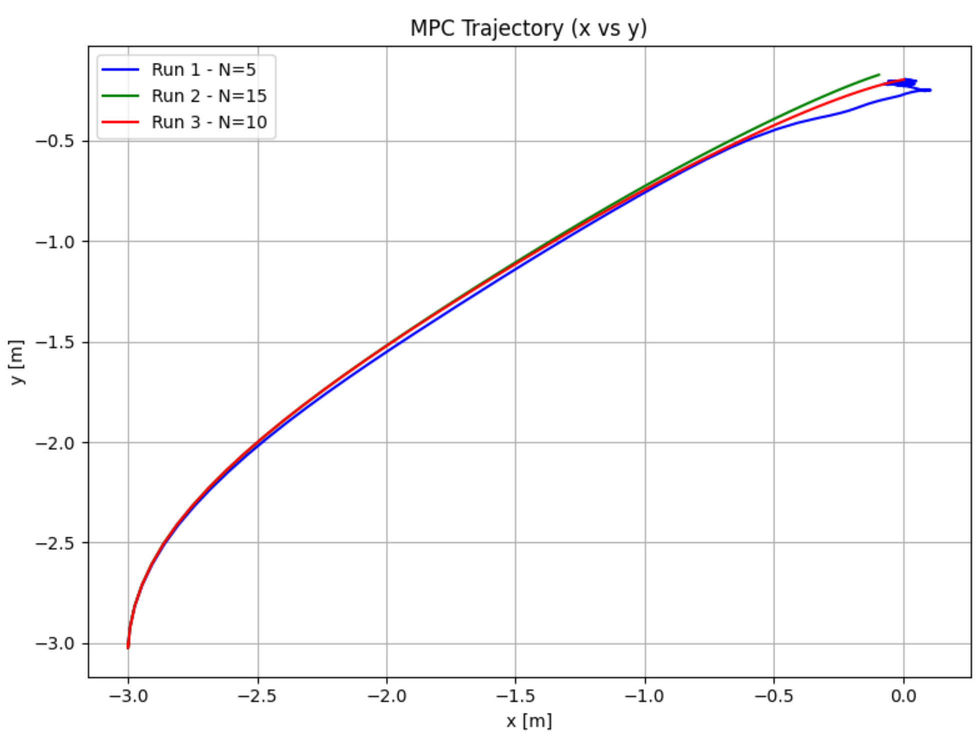

Figure 10: Trajectory

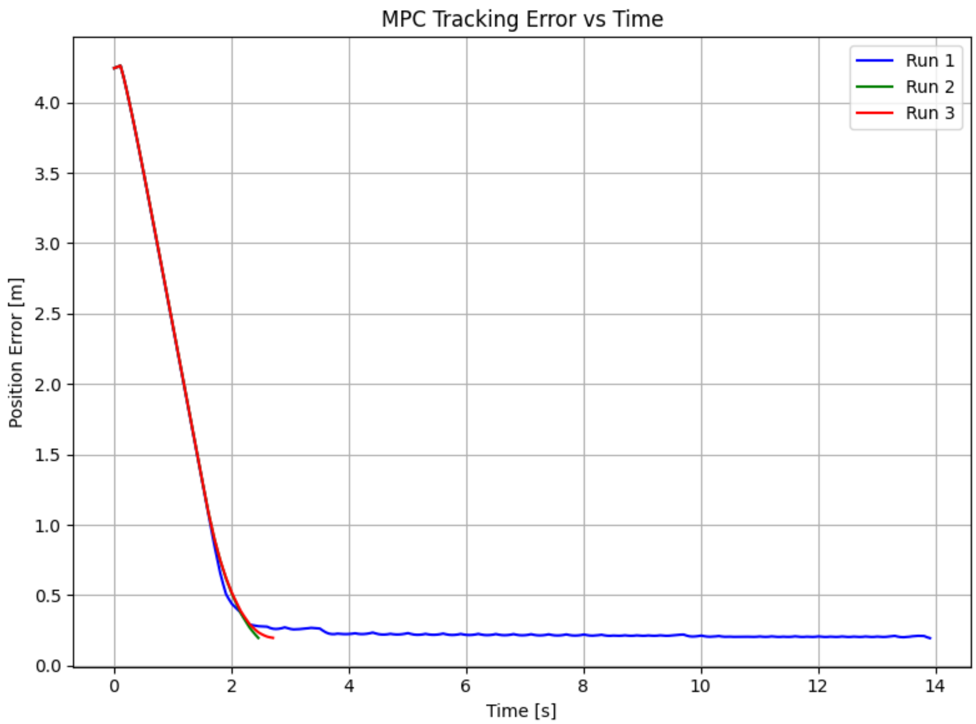

Figure 11: Tracking Error

5.4 Analysis

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

- **Trajectory**: Myopic, cuts corners, oscillatory near goal.
- **Error**: Settles in $\approx 10\,\text{s}$ with ripples $\pm 0.05\,\text{m}$.
- **Cost**: Highest accumulated cost.

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

- **Trajectory**: Smoother, fewer sharp turns.
- **Error**: Settles in $\approx 8\,\text{s}$, oscillations $< 0.02\,\text{m}$.
- **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 $\approx 6\,\mathrm{s}$, final error $< 0.01\,\mathrm{m}$.
- **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

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

```

```
Remarks:
- All vectors are treated as of type [n,]
- All buffers are treated as of type [L, n] where each row is a vector
- Buffers are updated from bottom to top
"""

import numpy as np
from numpy.random import randn

class System:
    """
    Interface class of dynamical systems a.k.a. environments.
    Concrete systems should be built upon this class.
    To design a concrete system: inherit this class, override:
        | :func:`~systems.system._state_dyn` :
        | right-hand side of system description (required)
        | :func:`~systems.system._disturb_dyn` :
        | right-hand side of disturbance model (if necessary)
        | :func:`~systems.system._ctrl_dyn` :
        | right-hand side of controller dynamical model (if necessary)
        | :func:`~systems.system.out` :
        | system out (if not overridden, output is identical to state)

    Attributes
    ----------
    sys_type : : string
        Type of system by description:
        | ``diff_eqn`` : differential equation :math:`\mathcal D state = f(state, action, disturb)`
        | ``discr_fnc`` : difference equation :math:`state^+ = f(state, action, disturb)`
        | ``discr_prob`` :  by probability distribution :math:`X^+ \sim P_X(state^+| state, action, disturb)`

    where:
```

```

        | :math:`state` : state
        | :math:`action` : input
        | :math:`disturb` : disturbance

    The time variable ``t`` is commonly used by ODE solvers, and you shouldn't have it explicitly referenced in the definition, unless your system is non-autonomous.
    For the latter case, however, you already have the input and disturbance at your disposal.

    Parameters of the system are contained in ``pars`` attribute.

    dim_state, dim_input, dim_output, dim_disturb : : integer
        System dimensions
    pars : : list
        List of fixed parameters of the system
    ctrl_bnds : : array of shape ``[dim_input, 2]``
        Box control constraints.
        First element in each row is the lower bound, the second - the upper bound.
        If empty, control is unconstrained (default)
    is_dyn_ctrl : : 0 or 1
        If 1, the controller (a.k.a. agent) is considered as a part of the full state vector
    is_disturb : : 0 or 1
        If 0, no disturbance is fed into the system
    pars_disturb : : list
        Parameters of the disturbance model

   Each concrete system must realize ``System`` and define ``name`` attribute.

    """
    def __init__(self,
                 sys_type,
                 dim_state,
                 dim_input,
```

```
                 dim_output,
                 dim_disturb,
                 pars=[],
                 ctrl_bnds=[],
                 is_dyn_ctrl=0,
                 is_disturb=0,
                 pars_disturb=[]):

        """
        Parameters
        ----------
        sys_type : : string
            Type of system by description:

            | ``diff_eqn`` : differential equation :math:`\mathcal D
                state = f(state, action, disturb)`
            | ``discr_fnc`` : difference equation :math:`state^+ = f
               (state, action, disturb)`
            | ``discr_prob`` :  by probability distribution :math:`X
               ^+ \sim P_X(state^+| state, action, disturb)`

        where:

            | :math:`state` : state
            | :math:`action` : input
            | :math:`disturb` : disturbance

        The time variable ``t`` is commonly used by ODE solvers, and
            you shouldn't have it explicitly referenced in the
           definition, unless your system is non-autonomous.
        For the latter case, however, you already have the input and
            disturbance at your disposal.

        Parameters of the system are contained in ``pars`` attribute
           .

        dim_state, dim_input, dim_output, dim_disturb : : integer
            System dimensions
```

```
        pars : : list
            List of fixed parameters of the system
        ctrl_bnds : : array of shape ``[dim_input, 2]``
            Box control constraints.
            First element in each row is the lower bound, the second - the upper bound.
            If empty, control is unconstrained (default)
        is_dyn_ctrl : : 0 or 1
            If 1, the controller (a.k.a. agent) is considered as a part of the full state vector
        is_disturb : : 0 or 1
            If 0, no disturbance is fed into the system
        pars_disturb : : list
            Parameters of the disturbance model
        """

        self.sys_type = sys_type

        self.dim_state = dim_state
        self.dim_input = dim_input
        self.dim_output = dim_output
        self.dim_disturb = dim_disturb
        self.pars = pars
        self.ctrl_bnds = ctrl_bnds
        self.is_dyn_ctrl = is_dyn_ctrl
        self.is_disturb = is_disturb
        self.pars_disturb = pars_disturb

        # Track system's state
        self._state = np.zeros(dim_state)

        # Current input (a.k.a. action)
        self.action = np.zeros(dim_input)

        if is_dyn_ctrl:
            if is_disturb:
                self._dim_full_state = self.dim_state + self.dim_disturb + self.dim_input
```

```
            else:
                self._dim_full_state = self.dim_state
        else:
            if is_disturb:
                self._dim_full_state = self.dim_state + self.
                   dim_disturb
            else:
                self._dim_full_state = self.dim_state

    def _state_dyn(self, t, state, action, disturb):
        """
        Description of the system internal dynamics.
        Depending on the system type, may be either the right-hand
           side of the respective differential or difference
           equation, or a probability distribution.
        As a probability disitribution, ``_state_dyn`` should return
            a number in :math:`[0,1]`

        """
        pass

    def _disturb_dyn(self, t, disturb):
        """
        Dynamical disturbance model depending on the system type:

        | ``sys_type = "diff_eqn"`` : :math:`\mathcal D disturb =
           f_q(disturb)`
        | ``sys_type = "discr_fnc"`` : :math:`disturb^+ = f_q(
           disturb)`
        | ``sys_type = "discr_prob"`` : :math:`disturb^+ \sim P_Q(
           disturb^+|disturb)`

        """
        pass

    def _ctrl_dyn(self, t, action, observation):
        """
        Dynamical controller. When ``is_dyn_ctrl=0``, the controller
```

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

        Depending on the system type, can be:

        | ``sys_type = "diff_eqn"`` : :math:`\mathcal D action = f_u
            (action, observation)`
        | ``sys_type = "discr_fnc"`` : :math:`action^+ = f_u(action,
            observation)`
        | ``sys_type = "discr_prob"`` : :math:`action^+ \sim P_U(
            action^+|action, observation)`

        """
        Daction = np.zeros(self.dim_input)

        return Daction

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

        See also
        --------
        :func:`~systems.system._state_dyn`

        """
        # Trivial case: output identical to state
        observation = state
        return observation
```

```python

    def receive_action(self, action):
        """
        Receive exogeneous control action to be fed into the system.
        This action is commonly computed by your controller (agent)
           using the system output :func:`~systems.system.out`.

        Parameters
        ----------
        action : : array of shape ``[dim_input, ]``
            Action

        """
        self.action = action

    def closed_loop_rhs(self, t, state_full):
        """
        Right-hand side of the closed-loop system description.
        Combines everything into a single vector that corresponds to
            the right-hand side of the closed-loop system
           description for further use by simulators.

        Attributes
        ----------
        state_full : : vector
            Current closed-loop system state

        """
        rhs_full_state = np.zeros(self._dim_full_state)

        state = state_full[0:self.dim_state]

        if self.is_disturb:
            disturb = state_full[self.dim_state:]
        else:
            disturb = []

        if self.is_dyn_ctrl:
```

```python
            action = state_full[-self.dim_input:]
            observation = self.out(state)
            rhs_full_state[-self.dim_input:] = self._ctrlDyn(t, action, observation)
        else:
            # Fetch the control action stored in the system
            action = self.action

        if self.ctrl_bnds.any():
            for k in range(self.dim_input):
                action = action.copy()
# action[k] = np.clip(action[k], self.ctrl_bnds[k, 0], self.ctrl_bnds[k, 1])

                action[k] = np.clip(action[k], self.ctrl_bnds[k, 0], self.ctrl_bnds[k, 1])

        rhs_full_state[0:self.dim_state] = self._state_dyn(t, state, action, disturb)

        if self.is_disturb:
            rhs_full_state[self.dim_state:] = self._disturb_dyn(t, disturb)

        # Track system's state
        self._state = state

        return rhs_full_state

class Sys3WRobotNI(System):
    """
    System class: 3-wheel robot with static actuators (the NI - non-holonomic integrator).


    """

    def __init__(self, *args, **kwargs):
```

```
        super().__init__(*args, **kwargs)

        self.name = '3wrobotNI'

        if self.is_disturb:
            self.sigma_disturb = self.pars_disturb[0]
            self.mu_disturb = self.pars_disturb[1]
            self.tau_disturb = self.pars_disturb[2]

    def _state_dyn(self, t, state, action, disturb=[]):
        Dstate = np.zeros(self.dim_state)
        # print(state,state.shape,action,action.shape)



        Dstate[0]=action[0]*np.cos(state[2])
        Dstate[1]=action[0]*np.sin(state[2])
        Dstate[2]=action[1]

        if disturb:
           Dstate += np.array(disturb)




        return Dstate

    def _disturb_dyn(self, t, disturb):
        """


        """
        Ddisturb = np.zeros(self.dim_disturb)

        for k in range(0, self.dim_disturb):
            Ddisturb[k] = - self.tau_disturb[k] * ( disturb[k] +
               self.sigma_disturb[k] * (randn() + self.mu_disturb[k
               ]) )
```

```python

        return Ddisturb

    def out(self, state, action=[]):
        observation = np.zeros(self.dim_output)
        observation = state
        return observation
```

Listing 2: Utilities.py

```python
"""
Contains auxiliary functions.

"""

import numpy as np
from numpy.random import rand
from numpy.matlib import repmat
import scipy.stats as st
from scipy import signal
import matplotlib.pyplot as plt

def rej_sampling_rvs(dim, pdf, M):
    """
    Random variable (pseudo)-realizations via rejection sampling.

    Parameters
    ----------
    dim : : integer
        dimension of the random variable
    pdf : : function
        desired probability density function
    M : : number greater than 1
        it must hold that :math:`\\text{pdf}_{\\text{desired}} \le M
           \\text{pdf}_{\\text{proposal}}`.
        This function uses a normal pdf with zero mean and identity
          covariance matrix as a proposal distribution.
        The smaller `M` is, the fewer iterations to produce a sample
           are expected.
```

```

    Returns
    -------
    A single realization (in general, as a vector) of the random
       variable with the desired probability density.

    """

    # Use normal pdf with zero mean and identity covariance matrix
       as a proposal distribution
    normal_RV = st.multivariate_normal(cov=np.eye(dim))

    # Bound the number of iterations to avoid too long loops
    max_iters = 1e3

    curr_iter = 0

    while curr_iter <= max_iters:
        proposal_sample = normal_RV.rvs()

        unif_sample = rand()

        if unif_sample < pdf(proposal_sample) / M / normal_RV.pdf(
           proposal_sample):
            return proposal_sample

def to_col_vec(argin):
    """
    Convert input to a column vector.

    """
    if argin.ndim < 2:
        return np.reshape(argin, (argin.size, 1))
    elif argin.ndim ==2:
        if argin.shape[0] < argin.shape[1]:
            return argin.T
        else:
            return argin
```

```python

def rep_mat(argin, n, m):
    """
    Ensures 1D result.

    """
    return np.squeeze(repmat(argin, n, m))

def push_vec(matrix, vec):
    return np.vstack([matrix[1:,:], vec])

def uptria2vec(mat):
    """
    Convert upper triangular square sub-matrix to column vector.

    """
    n = mat.shape[0]

    vec = np.zeros( (int(n*(n+1)/2)) )

    k = 0
    for i in range(n):
        for j in range(n):
            vec[j] = mat[i, j]
            k += 1

    return vec

class ZOH:
    """
    Zero-order hold.

    """
    def __init__(self, init_time=0, init_val=0, sample_time=1):
        self.time_step = init_time
        self.sample_time = sample_time
        self.currVal = init_val

```

```python
    def hold(self, signal_val, t):
        timeInSample = t - self.time_step
        if timeInSample >= self.sample_time: # New sample
            self.time_step = t
            self.currVal = signal_val

        return self.currVal

class DFilter:
    """
    Real-time digital filter.

    """
    def __init__(self, filter_num, filter_den, buffer_size=16,
       init_time=0, init_val=0, sample_time=1):
        self.Num = filter_num
        self.Den = filter_den
        self.zi = rep_mat( signal.lfilter_zi(filter_num, filter_den)
           , 1, init_val.size)

        self.time_step = init_time
        self.sample_time = sample_time
        self.buffer = rep_mat(init_val, 1, buffer_size)

    def filt(self, signal_val, t=None):
        # Sample only if time is specified
        if t is not None:
            timeInSample = t - self.time_step
            if timeInSample >= self.sample_time: # New sample
                self.time_step = t
                self.buffer = push_vec(self.buffer, signal_val)
        else:
            self.buffer = push_vec(self.buffer, signal_val)

        bufferFiltered = np.zeros(self.buffer.shape)

        for k in range(0, signal_val.size):
                bufferFiltered[k,:], self.zi[k] = signal.lfilter(
```

```
                   self.Num, self.Den, self.buffer[k,:], zi=self.zi[k, :])
        return bufferFiltered[-1,:]

def dss_sim(A, B, C, D, uSqn, x0, y0):
    """
    Simulate output response of a discrete-time state-space model.
    """
    if uSqn.ndim == 1:
        return y0, x0
    else:
        ySqn = np.zeros( [ uSqn.shape[0], C.shape[0] ] )
        xSqn = np.zeros( [ uSqn.shape[0], A.shape[0] ] )
        x = x0
        ySqn[0, :] = y0
        xSqn[0, :] = x0
        for k in range( 1, uSqn.shape[0] ):
            x = A @ x + B @ uSqn[k-1, :]
            xSqn[k, :] = x
            ySqn[k, :] = C @ x + D @ uSqn[k-1, :]

        return ySqn, xSqn

def upd_line(line, newX, newY):
    line.set_xdata( np.append( line.get_xdata(), newX) )
    line.set_ydata( np.append( line.get_ydata(), newY) )

def reset_line(line):
    line.set_data([], [])

def upd_scatter(scatter, newX, newY):
    scatter.set_offsets( np.vstack( [ scatter.get_offsets().data, np.c_[newX, newY] ] ) )

def upd_text(textHandle, newText):
    textHandle.set_text(newText)

def on_key_press(event, anm):
```

```
    """
    Key press event handler for a ``FuncAnimation`` animation object
       .

    """
    if event.key==' ':
        if anm.running:
            anm.event_source.stop()

        else:
            anm.event_source.start()
        anm.running ^= True
    elif event.key=='q':
        plt.close('all')
        raise Exception('exit')
```

Listing 3: Visuals.py

```
"""
Contains an interface class `animator` along with concrete
   realizations, each of which is associated with a corresponding
   system.

"""

import numpy as np
import numpy.linalg as la

from utilities import upd_line
from utilities import reset_line
from utilities import upd_scatter
from utilities import upd_text
from utilities import to_col_vec

import matplotlib as mpl
import matplotlib.pyplot as plt

from svgpath2mpl import parse_path
from collections import namedtuple
```

```
# from svgpathconverter import SVGPathConverter

class Animator:
    """
    Interface class of visualization machinery for simulation of system-controller loops.
    To design a concrete animator: inherit this class, override:
        | :func:`~visuals.Animator.__init__` :
        | define necessary visual elements (required)
        | :func:`~visuals.Animator.init_anim` :
        | initialize necessary visual elements (required)
        | :func:`~visuals.Animator.animate` :
        | animate visual elements (required)

    Attributes
    ----------
    objects : : tuple
        Objects to be updated within animation cycle
    pars : : tuple
        Fixed parameters of objects and visual elements

    """
    def __init__(self, objects=[], pars=[]):
        pass

    def init_anim(self):
        pass

    def animate(self, k):
        pass

    def get_anm(self, anm):
        """
        ``anm`` should be a ``FuncAnimation`` object.
        This method is needed to hand the animator access to the currently running animation, say, via ``anm.event_source.stop()``.

```

```
        """
        self.anm = anm

    def stop_anm(self):
        """
        Stops animation, provided that ``self.anm`` was defined via ``get_anm``.

        """
        self.anm.event_source.stop()


class RobotMarker:
    """
    Robot marker for visualization.

    """
    def __init__(self, angle=None, path_string=None):
        self.angle = angle or []
        self.path_string = path_string or """m 66.893258,227.10128 h 5.37899 v 0.91881 h 1.65571 l 1e-5,-3.8513 3.68556,-1e-5 v -1.43933
        l -2.23863,10e-6 v -2.73937 l 5.379,-1e-5 v 2.73938 h -2.23862 v 1.43933 h 3.68556 v 8.60486 l -3.68556,1e-5 v 1.43158
        h 2.23862 v 2.73989 h -5.37899 l -1e-5,-2.73989 h 2.23863 v -1.43159 h -3.68556 v -3.8513 h -1.65573 l 1e-5,0.91881 h -5.379 z"""

        self.path = parse_path( self.path_string )
        self.path.vertices -= self.path.vertices.mean( axis=0 )
        self.marker = mpl.markers.MarkerStyle( marker=self.path )
        self.marker._transform = self.marker.get_transform().rotate_deg(angle)

    def rotate(self, angle=0):
        self.marker._transform = self.marker.get_transform().rotate_deg(angle-self.angle)
```

```
        self.angle = angle

class Animator3WRobotNI(Animator):
    """
    Animator class for a 3-wheel robot with static actuators.

    """
    def __init__(self, objects=[], pars=[]):
        self.objects = objects
        self.pars = pars

        # Unpack entities
        self.simulator, self.sys, self.ctrl_nominal, self.
           ctrl_benchmarking,self.ctrl_lqr, self.datafiles, self.
           ctrl_selector, self.logger = self.objects

        state_init, \
        action_init, \
        t0, \
        t1, \
        state_full_init, \
        xMin, \
        xMax, \
        yMin, \
        yMax, \
        ctrl_mode, \
        action_manual, \
        v_min, \
        omega_min, \
        v_max, \
        omega_max, \
        Nruns, \
        is_print_sim_step, \
        is_log_data, \
        is_playback, \
        run_obj_init, \
        circle_coord = self.pars

```

```
        # Store some parameters for later use
        self.t0 = t0
        self.state_full_init = state_full_init
        self.t1 = t1
        self.ctrl_mode = ctrl_mode
        self.action_manual = action_manual
        self.Nruns = Nruns
        self.is_print_sim_step = is_print_sim_step
        self.is_log_data = is_log_data
        self.is_playback = is_playback

        xCoord0 = state_init[0]
        yCoord0 = state_init[1]
        alpha0 = state_init[2]
        alpha_deg0 = alpha0/2*np.pi

        plt.close('all')

        # print(v_min, v_max, omega_min, omega_max)
    #
        self.fig_sim = plt.figure(figsize=(10,10))

        # xy plane
        self.axs_xy_plane = self.fig_sim.add_subplot(221, autoscale_on=False, xlim=(xMin,xMax), ylim=(yMin,yMax),
                                                      xlabel='x [m]', ylabel='y [m]', title='Pause - space, q - quit, click - data cursor')
        self.axs_xy_plane.set_aspect('equal', adjustable='box')
        self.axs_xy_plane.plot([xMin, xMax], [0, 0], 'k--', lw=0.75)   # Help line
        self.axs_xy_plane.plot([0, 0], [yMin, yMax], 'k--', lw=0.75)   # Help line
        self.line_traj, = self.axs_xy_plane.plot(xCoord0, yCoord0, 'b--', lw=0.5)
```

```


        cirlce_target = plt.Circle((0, 0), radius=0.2, color='y', fill=True, lw=2)
        self.axs_xy_plane.add_artist(cirlce_target)
        self.text_target_handle = self.axs_xy_plane.text(0.88, 0.9, 'Target',
                                                     horizontalalignment='left', verticalalignment='center', transform=self.axs_xy_plane.transAxes)



        self.robot_marker = RobotMarker(angle=alpha_deg0)
        text_time = 't = {time:2.3f}'.format(time = t0)
        self.text_time_handle = self.axs_xy_plane.text(0.05, 0.95, text_time,
                                                     horizontalalignment='left', verticalalignment='center', transform=self.axs_xy_plane.transAxes)
        self.axs_xy_plane.format_coord = lambda state,observation: '%2.2f, %2.2f' % (state,observation)

        # Solution
        sol_max_on_plot = 2*np.max( np.array( [np.abs( xMin), np.abs( yMin), np.abs( yMin), np.abs( yMax)] ) )
        self.axs_sol = self.fig_sim.add_subplot(222, autoscale_on=False, xlim=(t0,t1), ylim=( -1.1*sol_max_on_plot, 1.1*sol_max_on_plot ), xlabel='t [s]')
        # 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'$\Vert(x,y)\Vert$ [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-',
```

```
            lw=0.5, label='Run. obj.')
        self.line_accum_obj, = self.axs_cost.plot(t0, 0, 'g-', lw=0.5, label=r'$\int \mathrm{Run.\,obj.} \,\mathrm{d}t$')
        self.axs_cost.legend(fancybox=True, loc='upper right')

        # Control
        self.axs_ctrl = self.fig_sim.add_subplot(224, autoscale_on=False, xlim=(t0,t1), ylim=(1.1*np.min([v_min, omega_min]), 1.1*np.max([v_max, omega_max])), xlabel='t [s]')
        self.axs_ctrl.plot([t0, t1], [0, 0], 'k--', lw=0.75)   # Help line
        self.lines_ctrl = self.axs_ctrl.plot(t0, to_col_vec(action_init).T, lw=0.5)
        # self.axs_ctrl.legend(iter(self.lines_ctrl), ('v [m/s]', r'$\omega$ [rad/s]'), fancybox=True, loc='upper right')
        self.axs_ctrl.legend(self.lines_ctrl, ('v [m/s]', r'$\omega$ [rad/s]'), fancybox=True, loc='upper right')



        # Pack all lines together
        cLines = namedtuple('lines', ['line_traj', 'line_norm', 'line_alpha', 'line_run_obj', 'line_accum_obj', 'lines_ctrl'])
        self.lines = cLines(line_traj=self.line_traj,
                            line_norm=self.line_norm,
                            line_alpha=self.line_alpha,
                            line_run_obj=self.line_run_obj,
                            line_accum_obj=self.line_accum_obj,
                            lines_ctrl=self.lines_ctrl)

        self.AAA = []
        self.index_prev = 0

    def set_sim_data(self, ts, xCoords, yCoords, alphas, rs, accum_objs, vs, omegas):
        """
        This function is needed for playback purposes when
```

```
            simulation data were generated elsewhere.
        It feeds data into the animator from outside.
        The simulation step counter ``curr_step`` is reset accordingly.

        """
        self.ts, self.xCoords, self.yCoords, self.alphas = ts, xCoords, yCoords, alphas
        self.rs, self.accum_objs, self.vs, self.omegas = rs, accum_objs, vs, omegas
        self.curr_step = 0

    def upd_sim_data_row(self):
        self.t = self.ts[self.curr_step]
        self.state_full = np.array([self.xCoords[self.curr_step], self.yCoords[self.curr_step], self.alphas[self.curr_step]])
        self.run_obj = self.rs[self.curr_step]
        self.accum_obj = self.accum_objs[self.curr_step]
        self.action = np.array([self.vs[self.curr_step], self.omegas[self.curr_step]])

        self.curr_step = self.curr_step + 1

    def init_anim(self):
        state_init, *_ = self.pars

        xCoord0 = state_init[0]
        yCoord0 = state_init[1]

        self.scatter_sol = self.axs_xy_plane.scatter(xCoord0, yCoord0, marker=self.robot_marker.marker, s=400, c='b')
        self.run_curr = 1
        self.datafile_curr = self.datafiles[0]

    def animate(self, k):

        if self.is_playback:
```

```
            self.upd_sim_data_row()
            t = self.t
            state_full = self.state_full
            action = self.action
            run_obj = self.run_obj
            accum_obj = self.accum_obj

        else:
            self.simulator.sim_step()

            t, state, observation, state_full = self.simulator.get_sim_step_data()

            while observation[2] > np.pi:
                    observation[2] -= 2 * np.pi
            while observation[2] < -np.pi:
                    observation[2] += 2 * np.pi

            action = self.ctrl_selector(t, observation, self.action_manual, self.ctrl_nominal, self.ctrl_benchmarking,self.ctrl_lqr, self.ctrl_mode)

            self.sys.receive_action(action)
            self.ctrl_benchmarking.receive_sys_state(self.sys._state)
            self.ctrl_benchmarking.upd_accum_obj(observation, action)

            run_obj = self.ctrl_benchmarking.run_obj(observation, action)
            accum_obj = self.ctrl_benchmarking.accum_obj_val

        xCoord = state_full[0]
        yCoord = state_full[1]
        alpha = state_full[2]
        alpha_deg = alpha/np.pi*180

        if self.is_print_sim_step:
```

```
            self.logger.print_sim_step(t, xCoord, yCoord, alpha, run_obj, accum_obj, action)

        if self.is_log_data:
            self.logger.log_data_row(self.datafile_curr, t, xCoord, yCoord, alpha, run_obj, accum_obj, action)

        # xy plane
        text_time = 't = {time:2.3f}'.format(time = t)
        upd_text(self.text_time_handle, text_time)
        upd_line(self.line_traj, xCoord, yCoord)  # Update the robot's track on the plot


        self.robot_marker.rotate(alpha_deg)    # Rotate the robot on the plot
        self.scatter_sol.remove()
        self.scatter_sol = self.axs_xy_plane.scatter(xCoord, yCoord, marker=self.robot_marker.marker, s=400, c='b')

        # # Solution
        upd_line(self.line_norm, t, la.norm([xCoord, yCoord]))
        upd_line(self.line_alpha, t, alpha)

        # Cost
        upd_line(self.line_run_obj, t, run_obj)
        upd_line(self.line_accum_obj, t, accum_obj)
        text_accum_obj = r'$\int \mathrm{{Run.\,obj.}} \,\mathrm{{d}}t$ = {accum_obj:2.1f}'.format(accum_obj = accum_obj)
        upd_text(self.text_accum_obj_handle, text_accum_obj)

        # Control
        for (line, action_single) in zip(self.lines_ctrl, action):
            upd_line(line, t, action_single)


        # Run done
        if t >= self.t1 or np.linalg.norm(observation[:2]) < 0.2:
```

```
            if self.is_print_sim_step:
                print('....................................Run {run:2d} done....................................'.format(run = self.run_curr))

            self.run_curr += 1

            if self.is_log_data:
                print("Data file:     ", self.datafile_curr)
                print(str(self.datafile_curr) + ".svg" + " saved")
                plt.savefig(str(self.datafile_curr) + ".svg")

            if self.run_curr > self.Nruns:
                print('Animation done...')
                # print(self.AAA)
                self.stop_anm()
                # plt.close('all')
                # print("HERE OK")
                return

            if self.is_log_data:
                self.datafile_curr = self.datafiles[self.run_curr-1]

            # Reset simulator
            self.simulator.reset()

            # Reset controller
            if self.ctrl_mode != 'nominal':
                self.ctrl_benchmarking.reset(self.t0)
            else:
                self.ctrl_nominal.reset(self.t0)


            accum_obj = 0

            reset_line(self.line_norm)
            reset_line(self.line_alpha)
            reset_line(self.line_run_obj)
```

```
            reset_line(self.line_accum_obj)
            reset_line(self.lines_ctrl[0])
            reset_line(self.lines_ctrl[1])
            reset_line(self.line_traj)

            upd_line(self.line_traj, np.nan, np.nan)
```

Listing 4: Simulator.py

```
"""
Contains one single class that simulates controller-system (agent-environment) loops.
The system can be of three types:

- discrete-time deterministic
- continuous-time deterministic or stochastic
- discrete-time stochastic (to model Markov chains)

"""

import numpy as np
import scipy as sp

from utilities import rej_sampling_rvs

class Simulator:
    """
    Class for simulating closed loops (system-controllers).

    Attributes
    ----------
    sys_type : : string
        Type of system by description:

        | ``diff_eqn`` : differential equation :math:`\mathcal D state = f(state, u, q)`
        | ``discr_fnc`` : difference equation :math:`state^+ = f(state, u, q)`
        | ``discr_prob`` :  by probability distribution :math:`X^+ \
```

```
        sim P_X(state^+| state, u, q)`

    where:

        | :math:`state` : state
        | :math:`u` : input
        | :math:`q` : disturbance

    closed_loop_rhs : : function
        Right-hand side description of the closed-loop system.
        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``,
        this could be just ``my_sys.closed_loop_rhs``.

    sys_out : : function
        System output function.
        Same as above, this could be, say, ``my_sys.out``.

    is_dyn_ctrl : : 0 or 1
        If 1, the controller (a.k.a. agent) is considered as a part
           of the full state vector.

    state_init, disturb_init, action_init : : vectors
        Initial values of the (open-loop) system state, disturbance
           and input.

    t0, t1, dt : : numbers
        Initial, final times and time step size

    max_step, first_step, atol, rtol : : numbers
        Parameters for an ODE solver (used if ``sys_type`` is ``
           diff_eqn``).

    See also
    --------

    ``systems`` module
```

```
    """

    def __init__(self, sys_type,
                 closed_loop_rhs,
                 sys_out,
                 state_init,
                 disturb_init=[],
                 action_init=[],
                 t0=0,
                 t1=1,
                 dt=1e-2,
                 max_step=0.5e-2,
                 first_step=1e-6,
                 atol=1e-5,
                 rtol=1e-3,
                 is_disturb=0,
                 is_dyn_ctrl=0):

        """
        Parameters
        ----------
        sys_type : : string
            Type of system by description:

            | ``diff_eqn`` : differential equation :math:`\mathcal D state = f(state, u, q)`
            | ``discr_fnc`` : difference equation :math:`state^+ = f(state, u, q)`
            | ``discr_prob`` :  by probability distribution :math:`X^+ \sim P_X(state^+| state, u, q)`

        where:

            | :math:`state` : state
            | :math:`u` : input
            | :math:`q` : disturbance

```

```
        closed_loop_rhs : : function
            Right-hand side description of the closed-loop system.
            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``,
            this could be just ``my_sys.closed_loop_rhs``.

        sys_out : : function
            System output function.
            Same as above, this could be, say, ``my_sys.out``.

        is_dyn_ctrl : : 0 or 1
            If 1, the controller (a.k.a. agent) is considered as a
               part of the full state vector.

        state_init, disturb_init, action_init : : vectors
            Initial values of the (open-loop) system state,
               disturbance and input.

        t0, t1, dt : : numbers
            Initial, final times and time step size

        max_step, first_step, atol, rtol : : numbers
            Parameters for an ODE solver (used if ``sys_type`` is ``
               diff_eqn``).
        """

        self.sys_type = sys_type
        self.closed_loop_rhs = closed_loop_rhs
        self.sys_out = sys_out
        self.dt = dt

        # Build full state of the closed-loop
        if is_dyn_ctrl:
            if is_disturb:
                state_full_init = np.concatenate([state_init,
                   disturb_init, action_init])
            else:
```

```python
                state_full_init = np.concatenate([state_init, action_init])
        else:
            if is_disturb:
                state_full_init = np.concatenate([state_init, disturb_init])
            else:
                state_full_init = state_init

        self.state_full = state_full_init

        self.t = t0
        self.state = state_init
        self.dim_state = state_init.shape[0]
        self.observation = self.sys_out(state_init)

        if sys_type == "diff_eqn":

            # Store these for reset purposes
            self.state_full_init = state_full_init
            self.t0 = t0
            self.t1 = t1
            self.max_step = dt/2
            self.first_step = first_step
            self.atol = atol
            self.rtol = rtol

            # self.ODE_solver = sp.integrate.RK45(closed_loop_rhs, t0, state_full_init, t1, max_step = dt/2, first_step=first_step, atol=atol, rtol=rtol)
            self.ODE_solver = sp.integrate.RK45(self.closed_loop_rhs,
                                                self.t0,
                                                self.state_full_init,
                                                self.t1,
                                                max_step=self.dt/2,
                                                first_step=self.
```

```
                                                    first_step,
                                                 atol=self.atol,
                                                 rtol=self.rtol
                                                 )

    def sim_step(self):
        """
        Do one simulation step and update current simulation data (
           time, system state and output).

        """
        if self.sys_type == "diff_eqn":
            self.ODE_solver.step()

            self.t = self.ODE_solver.t
            self.state_full = self.ODE_solver.y

            self.state = self.state_full[0:self.dim_state]
            self.observation = self.sys_out(self.state)

        elif self.sys_type == "discr_fnc":
            self.t = self.t + self.dt
            self.state_full = self.closed_loop_rhs(self.t, self.
               state_full)

            self.state = self.state_full[0:self.dim_state]
            self.observation = self.sys_out(self.state)

        elif self.sys_type == "discr_prob":
            self.state_full = rej_sampling_rvs(self.dim_state, self.
               closed_loop_rhs, 10)

            self.t = self.t + self.dt

            self.state = self.state_full[0:self.dim_state]
            self.observation = self.sys_out(self.state)
        else:
            raise ValueError('Invalid system description')
```

```
    def get_sim_step_data(self):
        """
        Collect current simulation data: time, system state and
           output, and, for completeness, full closed-loop state.

        """

        t, state, observation, state_full = self.t, self.state, self
           .observation, self.state_full

        return t, state, observation, state_full

    def reset(self):
        if self.sys_type == "diff_eqn":
            self.ODE_solver = sp.integrate.RK45(self.closed_loop_rhs
               ,
                                                self.t0,
                                                self.state_full_init,
                                                self.t1,
                                                max_step=self.dt/2,
                                                first_step=self.first_step,
                                                atol=self.atol,
                                                rtol=self.rtol
                                                )

        else:
            self.t = self.t0
            self.ODE_solver.y = self.state_full_init
```

Listing 5: Models.py

```
"""
Contains classes to be used in fitting system models.
Includes both State-Space and placeholder Neural Network models.
"""

import numpy as np # Importing numpy for matrix operations

```

```python
class ModelSS:
    """
    Discrete-Time State-Space Model

    \[
    \begin{array}{ll}
        \hat{x}_{k+1} &= A \hat{x}_k + B u_k \\
        y_k &= C \hat{x}_k + D u_k
    \end{array}
    \]

    Attributes:
    -----------
    A, B, C, D : ndarray
        State-space matrices defining system dynamics.
    x0set : ndarray
        Initial condition (state estimate) for simulation.
    """

    def __init__(self, A, B, C, D, x0est):  # Consistent naming for initial state
        self.A = A  # State transition matrix
        self.B = B  # Input matrix
        self.C = C  # Output matrix
        self.D = D  # Feedforward matrix
        self.x0set = x0est  # Consistent naming for initial state

    def upd_pars(self, Anew, Bnew, Cnew, Dnew):
        """Update system matrices with new values."""
        self.A = Anew   # Update state transition matrix
        self.B = Bnew   # Update input matrix
        self.C = Cnew   # Update output matrix
        self.D = Dnew   # Update feedforward matrix

    def updateIC(self, x0setNew):
        """Update initial condition/state estimate."""
        self.x0set = x0setNew   # Consistent naming for initial state
```

```python

    def predict(self, x, u):
        """
        Perform one-step forward simulation using current model.

        Parameters:
        -----------
        x : ndarray
            Current state.
        u : ndarray
            Current control input.

        Returns:
        --------
        x_next : ndarray
            Predicted next state.
        y : ndarray
            Output signal.
        """
        x_next = self.A @ x + self.B @ u    # State update equation
        y = self.C @ x + self.D @ u   # Output equation
        return x_next, y


class ModelNN:
    """
    Placeholder for Neural Network Model.
    Intended to be implemented in the future for learning nonlinear
       dynamics.
    """

    def __init__(self, *args, **kwargs):
        raise NotImplementedError(f"Class {self.__class__.__name__}
           is not yet implemented.")
        # Placeholder for future implementation
        # This will raise an error if instantiated, indicating that
           the class is not yet ready for use.
    def predict(self, x, u):
```

```
        """
        Placeholder for prediction method.
        Intended to be implemented in the future for learning nonlinear dynamics.

        Parameters:
        -----------
        x : ndarray
            Current state.
        u : ndarray
            Current control input.

        Returns:
        --------
        NotImplementedError
            Indicates that this method is not yet implemented.
        """
        raise NotImplementedError(f"Method {self.predict.__name__} in class {self.__class__.__name__} is not yet implemented.")
        # Placeholder for future implementation
        # This will raise an error if called, indicating that the method is not yet ready for use.
```

Listing 6: Loggers.py

```
"""
Logger Module for Simulation Data
=================================

This file provides logging functionality for simulations, including both console output and CSV file logging.
Supports a generic base Logger class and a concrete implementation for a 3-wheel robot (Sys3WRobotNI).
"""

import csv
from tabulate import tabulate

```

```
class Logger:
    """
    Abstract base class for data loggers.

    Subclasses must override:
        - print_sim_step: prints simulation step data to console.
        - log_data_row: writes simulation step data to CSV.
    """

    def print_sim_step(self, *args, **kwargs):
        """Prints one step of simulation data to the console (must
           override)."""
        raise NotImplementedError("Subclasses must implement
           print_sim_step")

    def log_data_row(self, *args, **kwargs):
        """Logs one row of data to a CSV file (must override)."""
        raise NotImplementedError("Subclasses must implement
           log_data_row")


class Logger3WRobotNI(Logger):
    """
    Logger for a 3-wheel robot (non-holonomic integrator).

    Fields logged:
        - Time (t [s])
        - x position (x [m])
        - y position (y [m])
        - Orientation angle (alpha [rad])
        - Instantaneous cost (run_obj)
        - Accumulated cost (accum_obj)
        - Linear velocity (v [m/s])
        - Angular velocity (omega [rad/s])
    """

    def print_sim_step(self, t, xCoord, yCoord, alpha, run_obj,
       accum_obj, action):
```

```
        """
        Prints one simulation step as a formatted table to the
           console.
        """
        row_header = [
            't [s]', 'x [m]', 'y [m]', 'alpha [rad]',
            'run_obj', 'accum_obj', 'v [m/s]', 'omega [rad/s]'
        ]

        row_data = [t, xCoord, yCoord, alpha, run_obj, accum_obj,
           action[0], action[1]]

        row_format = (
            '8.3f', '8.3f', '8.3f', '8.3f',
            '8.1f', '8.1f', '8.3f', '8.3f'
        )

        table = tabulate(
            [row_header, row_data],
            floatfmt=row_format,
            headers='firstrow',
            tablefmt='grid'
        )

        print(table)

    def log_data_row(self, datafile, t, xCoord, yCoord, alpha,
       run_obj, accum_obj, action):
        """
        Appends a row of simulation data to the specified CSV file.

        Parameters:
            datafile (str): Path to CSV file.
            t (float): Time.
            xCoord (float): X-position.
            yCoord (float): Y-position.
            alpha (float): Orientation.
            run_obj (float): Instantaneous cost.
```

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

Listing 7: Controllers.py

```
"""
Contains controllers a.k.a. agents.

"""

from utilities import dss_sim
from utilities import rep_mat
from utilities import uptria2vec
from utilities import push_vec
import models
import numpy as np
import scipy as sp
from scipy.linalg import solve_discrete_are
from numpy.random import rand
from scipy.optimize import minimize
from scipy.optimize import basinhopping
from scipy.optimize import NonlinearConstraint
from scipy.stats import multivariate_normal
from numpy.linalg import lstsq
from numpy import reshape
import warnings
import math
# For debugging purposes
from tabulate import tabulate
import os

def ctrl_selector(t, observation, action_manual, ctrl_nominal,
   ctrl_benchmarking,ctrl_lqr, mode):
    """
```

```
    Main interface for various controllers.

    Parameters
    ----------
    mode : : string
        Controller mode as acronym of the respective control method.

    Returns
    -------
    action : : array of shape ``[dim_input, ]``.
        Control action.

    """

    if mode=='manual':
        action = action_manual
    elif mode=='Nominal':
        action = ctrl_nominal.compute_action(t, observation)
    elif mode=='lqr':
        action=ctrl_lqr.compute_action(t,observation)
    else: # Controller for benchmakring
        action = ctrl_benchmarking.compute_action(t, observation)

    return action


class ControllerOptimalPredictive:
    """
    Class of predictive optimal controllers, primarily model-predictive control and predictive reinforcement learning, that optimize a finite-horizon cost.

    Currently, the actor model is trivial: an action is generated directly without additional policy parameters.

    Attributes
    ----------
    dim_input, dim_output : : integer
```

```
        Dimension of input and output which should comply with the
           system-to-be-controlled.
    mode : : string
        Controller mode. Currently available (:math:`\\rho` is the
           running objective, :math:`\\gamma` is the discounting
           factor):

        .. list-table:: Controller modes
           :widths: 75 25
           :header-rows: 1

           * - Mode
             - Cost function
           * - 'MPC' - Model-predictive control (MPC)
             - :math:`J_a \\left( y_1, \\{action\\}_1^{N_a} \\right)
                = \\sum_{k=1}^{N_a} \\gamma^{k-1} \\rho(y_k, u_k)`
           * - 'RQL' - RL/ADP via :math:`N_a-1` roll-outs of :math
              :`\\rho`
             - :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})`
           * - 'SQL' - RL/ADP via stacked Q-learning
             - :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})`

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

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

    ctrl_bnds : : array of shape ``[dim_input, 2]``
        Box control constraints.
        First element in each row is the lower bound, the second -
           the upper bound.
        If empty, control is unconstrained (default).
```

```
    action_init : : array of shape ``[dim_input, ]``
        Initial action to initialize optimizers.
    t0 : : number
        Initial value of the controller's internal clock.
    sampling_time : : number
        Controller's sampling time (in seconds).
    Nactor : : natural number
        Size of prediction horizon :math:`N_a`.
    pred_step_size : : number
        Prediction step size in :math:`J_a` as defined above (in seconds). Should be a multiple of ``sampling_time``. Commonly, equals it, but here left adjustable for
        convenience. Larger prediction step size leads to longer factual horizon.
    sys_rhs, sys_out : : functions
        Functions that represent the right-hand side, resp., the output of the exogenously passed model.
        The latter could be, for instance, the true model of the system.
        In turn, ``state_sys`` represents the (true) current state of the system and should be updated accordingly.
        Parameters ``sys_rhs, sys_out, state_sys`` are used in those controller modes which rely on them.
    buffer_size : : natural number
        Size of the buffer to store data.
    gamma : : number in (0, 1]
        Discounting factor.
        Characterizes fading of running objectives along horizon.
    Ncritic : : natural number
        Critic stack size :math:`N_c`. The critic optimizes the temporal error which is a measure of critic's ability to capture the
        optimal infinite-horizon cost (a.k.a. the value function). The temporal errors are stacked up using the said buffer.
    critic_period : : number
        The critic is updated every ``critic_period`` units of time.
    critic_struct : : natural number
        Choice of the structure of the critic's features.
```

```
        Currently available:

        .. list-table:: Critic structures
           :widths: 10 90
           :header-rows: 1

           * - Mode
             - Structure
           * - 'quad-lin'
             - Quadratic-linear
           * - 'quadratic'
             - Quadratic
           * - 'quad-nomix'
             - Quadratic, no mixed terms
           * - 'quad-mix'
             - 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`,
             where :math:`w` is the critic's weight vector

        *Add your specification into the table when customizing the
           critic*.
    run_obj_struct : : string
        Choice of the running objective structure.

        Currently available:

        .. list-table:: Critic structures
           :widths: 10 90
           :header-rows: 1

           * - Mode
             - Structure
           * - 'quadratic'
             - Quadratic :math:`\\chi^\\top R_1 \\chi`, where :math
               :`\\chi = [observation, action]`, ``run_obj_pars``
               should be ``[R1]``
```

```
            * - 'biquadratic'
              - 4th order :math:`\\left( \\chi^\\top \\right)^2 R_2 \\left( \\chi \\right)^2 + \\chi^\\top R_1 \\chi`, where :math:`\\chi = [observation, action]`, ``run_obj_pars``
                should be ``[R1, R2]``

        *Pass correct run objective parameters in* ``run_obj_pars`` *(as a list)*

        *When customizing the running objective, add your specification into the table above*

    References
    ----------
    .. [1] Osinenko, Pavel, et al. "Stacked adaptive dynamic programming with unknown system model." IFAC-PapersOnLine 50.1 (2017): 4150-4155

    """
    def __init__(self,
                 dim_input,
                 dim_output,
                 mode='MPC',
                 ctrl_bnds=[],
                 action_init = [],
                 t0=0,
                 sampling_time=0.1,
                 Nactor=1,
                 pred_step_size=0.1,
                 sys_rhs=[],
                 sys_out=[],
                 state_sys=[],
                 buffer_size=20,
                 gamma=1,
                 Ncritic=4,
                 critic_period=0.1,
                 critic_struct='quad-nomix',
```

```
                         run_obj_struct='quadratic',
                         run_obj_pars=[],
                         observation_target=[],
                         state_init=[],
                         obstacle=[],
                         seed=1):
        """
            Parameters
            ----------
            dim_input, dim_output : : integer
                Dimension of input and output which should comply with the system-to-be-controlled.
            mode : : string
                Controller mode. Currently available (:math:`\\rho` is the running objective, :math:`\\gamma` is the discounting factor):

                .. list-table:: Controller modes
                :widths: 75 25
                :header-rows: 1

                * - Mode
                    - Cost function
                * - 'MPC' - Model-predictive control (MPC)
                    - :math:`J_a \\left( y_1, \\{action\\}_1^{N_a} \\right)= \\sum_{k=1}^{N_a} \\gamma^{k-1} \\rho(y_k, u_k)`
                * - 'RQL' - RL/ADP via :math:`N_a-1` roll-outs of :math:`\\rho`
                    - :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})`
                * - 'SQL' - RL/ADP via stacked Q-learning
                    - :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})`

```

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

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

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

```
                Size of the buffer to store data.
            gamma : : number in (0, 1]
                Discounting factor.
                Characterizes fading of running objectives along horizon.
            Ncritic : : natural number
                Critic stack size :math:`N_c`. The critic optimizes the temporal error which is a measure of critic's ability to capture the
                optimal infinite-horizon cost (a.k.a. the value function). The temporal errors are stacked up using the said buffer.
            critic_period : : number
                The critic is updated every ``critic_period`` units of time.
            critic_struct : : natural number
                Choice of the structure of the critic's features.

                Currently available:

                .. list-table:: Critic feature structures
                :widths: 10 90
                :header-rows: 1

                * - Mode
                    - Structure
                * - 'quad-lin'
                    - Quadratic-linear
                * - 'quadratic'
                    - Quadratic
                * - 'quad-nomix'
                    - Quadratic, no mixed terms
                * - 'quad-mix'
                    - 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`,
                    where :math:`w` is the critic's weights
```

```

            *Add your specification into the table when
               customizing the critic*.
        run_obj_struct : : string
            Choice of the running objective structure.

            Currently available:

            .. list-table:: Running objective structures
            :widths: 10 90
            :header-rows: 1

            * - Mode
                - Structure
            * - 'quadratic'
                - Quadratic :math:`\\chi^\\top R_1 \\chi`, where
                   :math:`\\chi = [observation, action]`, ``
                  run_obj_pars`` should be ``[R1]``
            * - 'biquadratic'
                - 4th order :math:`\\left( \\chi^\\top \\right)
                  ^2 R_2 \\left( \\chi \\right)^2 + \\chi^\\top
                   R_1 \\chi`, where :math:`\\chi = [
                  observation, action]`, ``run_obj_pars``
                should be ``[R1, R2]``
        """

    np.random.seed(seed)
    # print(seed)

    self.dim_input = dim_input
    self.dim_output = dim_output

    self.mode = mode

    self.ctrl_clock = t0
    self.sampling_time = sampling_time

    # Controller: common
```

```
        self.Nactor = Nactor
        self.pred_step_size = pred_step_size

        self.action_min = np.array( ctrl_bnds[:,0] )
        self.action_max = np.array( ctrl_bnds[:,1] )
        self.action_sqn_min = rep_mat(self.action_min , 1, Nactor)
        self.action_sqn_max = rep_mat(self.action_max , 1, Nactor)
        self.action_sqn_init = []
        self.state_init = []

        if len(action_init) == 0:
            self.action_curr = self.action_min/10
            self.action_sqn_init = rep_mat( self.action_min/10 , 1, self.Nactor)
            self.action_init = self.action_min/10
        else:
            self.action_curr = action_init
            self.action_sqn_init = rep_mat( action_init , 1, self.Nactor)


        self.action_buffer = np.zeros( [buffer_size , dim_input] )
        self.observation_buffer = np.zeros( [buffer_size , dim_output ] )
        self.t=0

        # Exogeneous model's things
        self.sys_rhs = sys_rhs
        self.sys_out = sys_out
        self.state_sys = state_sys

        # Learning-related things
        self.buffer_size = buffer_size
        self.critic_clock = t0
        self.gamma = gamma
        self.Ncritic = Ncritic
        self.Ncritic = np.min([self.Ncritic , self.buffer_size-1]) # Clip critic buffer size
```

```
        self.critic_period = critic_period
        self.critic_struct = critic_struct
        self.run_obj_struct = run_obj_struct
        self.run_obj_pars = run_obj_pars
        self.observation_target = observation_target

        self.accum_obj_val = 0
        # print('---Critic structure---', self.critic_struct)

        if self.critic_struct == 'quad-lin':
            self.dim_critic = int( ( ( self.dim_output + self.dim_input ) + 1 ) * ( self.dim_output + self.dim_input )/2 + (self.dim_output + self.dim_input) )
            self.Wmin = -1e3*np.ones(self.dim_critic)
            self.Wmax = 1e3*np.ones(self.dim_critic)
        elif self.critic_struct == 'quadratic':
            self.dim_critic = int( ( ( self.dim_output + self.dim_input ) + 1 ) * ( self.dim_output + self.dim_input )/2 )
            self.Wmin = np.zeros(self.dim_critic)
            self.Wmax = 1e3*np.ones(self.dim_critic)
        elif self.critic_struct == 'quad-nomix':
            self.dim_critic = self.dim_output + self.dim_input
            self.Wmin = np.zeros(self.dim_critic)
            self.Wmax = 1e3*np.ones(self.dim_critic)
        elif self.critic_struct == 'quad-mix':
            self.dim_critic = int( self.dim_output + self.dim_output * self.dim_input + self.dim_input )
            self.Wmin = -1e3*np.ones(self.dim_critic)
            self.Wmax = 1e3*np.ones(self.dim_critic)
        elif self.critic_struct == 'poly3':
            self.dim_critic = int( ( ( self.dim_output + self.dim_input ) + 1 ) * ( self.dim_output + self.dim_input ) )
            self.Wmin = -1e3*np.ones(self.dim_critic)
            self.Wmax = 1e3*np.ones(self.dim_critic)
        elif self.critic_struct == 'poly4':
            self.dim_critic = int( ( ( self.dim_output + self.
```

```
dim_input ) + 1 ) * ( self.dim_output + self.dim_input )/2 * 3)
            self.Wmin = np.zeros(self.dim_critic)
            self.Wmax = np.ones(self.dim_critic)
        self.N_CTRL = N_CTRL(k_rho=1.0, k_alpha=4.0, k_beta=-1.5, # Example gains, vary for Experiment A
        target_x=0.0, target_y=0.0, target_theta=0.0,
        sampling_time=sampling_time,
        dim_input=dim_input, dim_output=dim_output)


    def reset(self,t0):
        """
        Resets agent for use in multi-episode simulation.
        Only internal clock, value and current actions are reset.
        All the learned parameters are retained.

        """

        # Controller: common

        if len(self.action_init) == 0:
            self.action_curr = self.action_min/10
            self.action_sqn_init = rep_mat( self.action_min/10 , 1, self.Nactor)
            self.action_init = self.action_min/10
        else:
            self.action_curr = self.action_init
            self.action_sqn_init = rep_mat( self.action_init , 1, self.Nactor)

        self.action_buffer = np.zeros( [self.buffer_size, self.dim_input] )
        self.observation_buffer = np.zeros( [self.buffer_size, self.dim_output] )

        self.critic_clock = t0
        self.ctrl_clock = t0
```

```
    def receive_sys_state(self, state):
        """
        Fetch exogenous model state. Used in some controller modes.
           See class documentation.

        """
        self.state_sys = state

    def upd_accum_obj(self, observation, action):
        """
        Sample-to-sample accumulated (summed up or integrated)
           running objective. This can be handy to evaluate the
           performance of the agent.
        If the agent succeeded to stabilize the system, ``accum_obj
           `` would converge to a finite value which is the
           performance mark.
        The smaller, the better (depends on the problem
           specification of course - you might want to maximize cost
            instead).

        """
        self.accum_obj_val += self.run_obj(observation, action)*self
           .sampling_time

    def run_obj(self, observation, action,terminal=False):
        """
        Running (equivalently, instantaneous or stage) objective.
           Depending on the context, it is also called utility,
           reward, running cost etc.

        See class documentation.
        """
        observation_arr = np.array(observation)
        action_arr = np.array(action)

        # Handle observation target: calculate error if a target is
           provided
```

```python
        self.observation_target=np.array([0.0,0.0,0.0])
        observation_err = observation_arr - self.observation_target

        # Form the combined state-action vector chi
        chi = np.concatenate((observation_err, action_arr))
        run_obj=0



        # if self.t==self.Nactor-1:
        #     terminal=True

        if self.run_obj_struct == 'quadratic':
            # run_obj_pars should be [R1] where R1 is a matrix for chi.T @ R1 @ chi
            if not self.run_obj_pars or len(self.run_obj_pars) < 1:
                warnings.warn("run_obj_pars is empty or malformed for 'quadratic' mode. Returning 0 for running objective.")
                return 0.0 # Return a float
            R1 = np.array(self.run_obj_pars[0])
            # print(R1)
            # Check dimensions for compatibility: R1 must be square and match chi's dimension
            expected_dim = chi.shape[0]
            if R1.shape != (expected_dim, expected_dim):
                warnings.warn(f"R1 matrix dimension mismatch. Expected {expected_dim}x{expected_dim}, got {R1.shape}. Returning 0 for running objective.")
                return 0.0 # Return a float
            run_obj = chi.T @ R1 @ chi
            self.t+=1
            # print(self.t)

            # Add terminal cost if requested
            if terminal:
                if len(self.run_obj_pars) < 2:
                    warnings.warn("Qf not provided in run_obj_pars
```

```
for terminal cost. Skipping terminal cost.")
                else:
                    Qf = np.array(self.run_obj_pars[1])
                    obs_dim = observation_err.shape[0]
                    if Qf.shape != (obs_dim, obs_dim):
                        warnings.warn(f"Qf matrix dimension mismatch. Expected {obs_dim}x{obs_dim}, got {Qf.shape}. Skipping terminal cost.")
                    else:
                        run_obj += observation_err.T @ Qf @ observation_err



        return run_obj

    def _actor_cost(self, action_sqn, observation):
        """
        See class documentation.

        Customization
        -------------

        Introduce your mode and the respective actor loss in this method. Don't forget to provide description in the class documentation.

        """

        my_action_sqn = np.reshape(action_sqn, [self.Nactor, self.dim_input])

        observation_sqn = np.zeros([self.Nactor, self.dim_output])

        # System observation prediction
        observation_sqn[0, :] = observation
        state = self.state_sys
        for k in range(1, self.Nactor):
```

```
            state = state + self.pred_step_size * self.sys_rhs(0.01, state, my_action_sqn[k-1, :])  # Euler scheme

            observation_sqn[k, :] = self.sys_out(state)

        J = 0
        if self.mode=='MPC':
            for k in range(self.Nactor):

                if k==self.Nactor-1:
                    J+=self.gamma**k * self.run_obj(observation_sqn[k, :], my_action_sqn[k, :],terminal=True)

                J += self.gamma**k * self.run_obj(observation_sqn[k, :], my_action_sqn[k, :])

        return J

    def _actor_optimizer(self, observation):
        """
        This method is merely a wrapper for an optimizer that minimizes :func:`~controllers.ControllerOptimalPredictive._actor_cost`.
        See class documentation.

        Customization
        -------------

        This method normally should not be altered, adjust :func:`~controllers.ControllerOptimalPredictive._actor_cost` instead.
        The only customization you might want here is regarding the optimization algorithm.



        # For direct implementation of state constraints, this needs `partial` from `functools`
```

```
        # See [here](https://stackoverflow.com/questions/27659235/adding-multiple-constraints-to-scipy-minimize-autogenerate-constraint-dictionar)
        # def state_constraint(action_sqn, idx):

        #     my_action_sqn = np.reshape(action_sqn, [N, self.dim_input])

        #     observation_sqn = np.zeros([idx, self.dim_output])

        #     # System output prediction
        #     if (mode==1) or (mode==3) or (mode==5):    # Via exogenously passed model
        #         observation_sqn[0, :] = observation
        #         state = self.state_sys
        #         Y[0, :] = observation
        #         x = self.x_s
        #         for k in range(1, idx):
        #             # state = get_next_state(state, my_action_sqn[k-1, :], delta)
        #             state = state + delta * self.sys_rhs([], state, my_action_sqn[k-1, :], [])  # Euler scheme
        #             observation_sqn[k, :] = self.sys_out(state)

        #     return observation_sqn[-1, 1] - 1

        # my_constraints=[]
        # for my_idx in range(1, self.Nactor+1):
        #     my_constraints.append({'type': 'eq', 'fun': lambda action_sqn: state_constraint(action_sqn, idx=my_idx)})

        # my_constraints = {'type': 'ineq', 'fun': state_constraint}

        # Optimization method of actor
        # Methods that respect constraints: BFGS, L-BFGS-B, SLSQP, trust-constr, Powell
        # actor_opt_method = 'SLSQP' # Standard
        """
```

```
        actor_opt_method = 'SLSQP'
        if actor_opt_method == 'trust-constr':
            actor_opt_options = {'maxiter': 40, 'disp': False} #'disp': True, 'verbose': 2}
        else:
            actor_opt_options = {'maxiter': 40, 'maxfev': 60, 'disp': False, 'adaptive': True, 'xatol': 1e-3, 'fatol': 1e-3}

        isGlobOpt = 0

        my_action_sqn_init = np.reshape(self.action_sqn_init, [self.Nactor*self.dim_input,])

        bnds = sp.optimize.Bounds(self.action_sqn_min, self.action_sqn_max, keep_feasible=True)

        try:
            if isGlobOpt:
                minimizer_kwargs = {'method': actor_opt_method, 'bounds': bnds, 'tol': 1e-3, 'options': actor_opt_options}
                action_sqn = basinhopping(lambda action_sqn: self._actor_cost(action_sqn, observation),
                                          my_action_sqn_init,
                                          minimizer_kwargs=minimizer_kwargs,
                                          niter = 10).x
            else:
                action_sqn = minimize(lambda action_sqn: self._actor_cost(action_sqn, observation),
                                      my_action_sqn_init,
                                      method=actor_opt_method,
                                      tol=1e-3,
                                      bounds=bnds,
                                      options=actor_opt_options).x

```

```
        except ValueError:
            print('Actor''s optimizer failed. Returning default
               action')
            action_sqn = self.action_curr

        ##print(action_sqn)

        return action_sqn[:self.dim_input]    # Return first action

    def compute_action(self, t, observation):
        """
        Main method. See class documentation.

        Customization
        -------------

        Add your modes, that you introduced in :func:`~controllers.
           ControllerOptimalPredictive._actor_cost`, here.

        """

        time_in_sample = t - self.ctrl_clock

        if time_in_sample >= self.sampling_time: # New sample
            # Update controller's internal clock
            self.ctrl_clock = t

            action = None

            if self.mode == 'MPC':

                action = self._actor_optimizer(observation)
                # print(action.shape)

            elif self.mode == "Nominal":

                action = self.N_CTRL.compute_action(observation)
            if action is not None:
```

```
                self.action_curr = action


            return action

        else:
            return self.action_curr



# New/Updated N_CTRL Class
class N_CTRL:
    def __init__(self, k_rho=1.0, k_alpha=2.0, k_beta=-1.5,
                 target_x=0.0, target_y=0.0, target_theta=0.0,
                 sampling_time=0.01, dim_input=2, dim_output=3): #
                    Added dim_input/output for consistency
        self.k_rho = k_rho
        self.k_alpha = k_alpha
        self.k_beta = k_beta
        self.dt = sampling_time # Use sampling_time for internal dt
            consistency

        self.target_x = target_x
        self.target_y = target_y
        self.target_theta = target_theta

        self.ctrl_clock = 0.0 # Initial time for the controller's
            internal clock
        self.sampling_time = sampling_time
        self.action_curr = np.array([0.0, 0.0]) # Initialize current
             action to zero
        self.accum_obj_val = 0.0 # Initialize accumulated objective
            value
        self.dim_input = dim_input
        self.dim_output = dim_output

        # Dummy attributes for compatibility with logger/visualizer
        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)

```

```python
            # Debug prints - these are very useful!
            print(f"t: {t:.2f}, Obs: ({x:.2f}, {y:.2f}, {th:.2f})")
            print(f"Target: ({x_f:.2f}, {y_f:.2f}, {th_f:.2f})")
            # print(f"Errors: rho={rho:.4f}, alpha={alpha:.4f}, beta={beta:.4f}")

            # Define a small tolerance for "at target"
            pos_tolerance = 0.05 # meters
            angle_tolerance = 0.05 # radians

            if rho < pos_tolerance:
                if abs(self.wrap_angle(th_f - th)) < angle_tolerance:
                    v = 0.0
                    w = 0.0
                # print("--> At target, stopping. v=0, w=0")
            else:
                v = self.k_rho * rho
                w = self.k_alpha * alpha + self.k_beta * beta
                # print(f"--> Control action: v={v:.4f}, w={w:.4f}")

            v = np.clip(v, -1, 1)
            w = np.clip(w, -1.0, 1.0)


            self.action_curr = np.array([v, w])
            return self.action_curr
        else:
            return self.action_curr # Return the last computed action if not a new sample

    def reset(self, t0):
        """
        Resets controller for new episode.
        """
        self.ctrl_clock = t0
        self.action_curr = np.array([0.0, 0.0])
        self.accum_obj_val = 0.0 # Reset accumulated objective
```

```

    def receive_sys_state(self, state):
        """
        Placeholder method for compatibility with the simulation framework.
        N_CTRL doesn't actively use the full system state for its internal logic,
        but the main simulation loop might try to pass it.
        """
        self.state_sys = state # Store it just in case, or do nothing if truly not needed.

    def upd_accum_obj(self, observation, action):
        """
        Accumulates the running objective for performance evaluation.
        """
        self.accum_obj_val += self.run_obj(observation, action) * self.sampling_time

    def run_obj(self, observation, action):
        """
        Calculates a running objective (cost) for the N_CTRL.
        This is primarily for logging and visualization, not for the controller's internal logic.
        """
        x, y, th = observation
        x_f = self.target_x
        y_f = self.target_y
        th_f = self.target_theta

        # Cost components: position error squared and orientation error squared, and control effort.
        pos_cost = (x - x_f)**2 + (y - y_f)**2
        # Use a proper angle difference for orientation cost
        # orientation_cost = self.wrap_angle(th - th_f)**2

        # Control effort cost (e.g., squared velocities)
```

```
        # v, w = action
        # control_effort_cost = 0.1 * (v**2 + w**2) # Small weight
           on control effort

        # You can tune these weights
        total_cost = pos_cost
        return total_cost

class LQR:
    def __init__(self,A,B,Q,R,obs_target=[0.0,0.0,0.0],sampling_time
       =0.01,):
        self.A=A
        self.B=B
        self.Q=Q
        self.R=R
        self.observation_target=obs_target
        self.ctrl_clock = 0.0
        self.sampling_time =sampling_time
        self.action_curr = np.array([0.0, 0.0]) # Initialize current
            action to zero
        self.accum_obj_val = 0.0

    def compute_gain(self):

        # Solve the Discrete Algebraic Riccati Equation (DARE)
        P = solve_discrete_are(self.A, self.B, self.Q, self.R)

        K=np.linalg.inv(self.R+self.B.T@P@self.B)@self.B.T@P@self.A

        return K

    def compute_action(self,t,observation):
        # 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(observation)
            # observation_target=[]

            state_err = observation - self.observation_target
            # print(state_err.shape)

            self.K=self.compute_gain()


            u=-self.K@state_err
            # print(u)

            # x=self.A@state_err +self.B@u

            self.action_curr = u

            # print(x)
            return self.action_curr
        else:
            return self.action_curr #


    def reset(self, t0):
        """
        Resets controller for new episode.
        """
        self.ctrl_clock = t0
        self.action_curr = np.array([0.0, 0.0])
        self.accum_obj_val = 0.0 # Reset accumulated objective

    def receive_sys_state(self, state):
        """
        Placeholder method for compatibility with the simulation framework.
        N_CTRL doesn't actively use the full system state for its internal logic,
        but the main simulation loop might try to pass it.
```

```python
        """
        self.state_sys = state # Store it just in case, or do nothing if truly not needed.

    def upd_accum_obj(self, observation, action):
        """
        Accumulates the running objective for performance evaluation.
        """
        self.accum_obj_val += self.run_obj(observation, action) * self.sampling_time

    def run_obj(self, observation, action):
        """
        Calculates a running objective (cost) for the N_CTRL.
        This is primarily for logging and visualization, not for the controller's internal logic.
        """
        state_error = np.diag(observation - self.observation_target)
        action=np.diag(action)
        running_cost = state_error.T @ self.Q @ state_error
        # action.T @ self.R@ action

        # print(running_cost.shape)
        return np.linalg.det(running_cost)
```

Listing 8: Present$_3Wrobot_NI.py$

```python
"""
Preset: a 3-wheel robot (kinematic model a. k. a. non-holonomic integrator).

"""

import pathlib
import os
import warnings
import csv
from datetime import datetime
```

```
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np

import systems
import simulator
import controllers
import loggers
import visuals
from utilities import on_key_press

import argparse

#----------------------------------------Set up dimensions
dim_state = 3
dim_input = 2
dim_output = dim_state
dim_disturb = 0

dim_R1 = dim_output + dim_input
dim_R2 = dim_R1

description = "Agent-environment preset: a 3-wheel robot (kinematic
   model a.k.a. non-holonomic integrator)."

parser = argparse.ArgumentParser(description=description)

parser.add_argument('--ctrl_mode', metavar='ctrl_mode', type=str,
                    choices=['MPC',
                             "Nominal",
                             "lqr"],
                    default="MPC",
                    help='Control mode. Currently available: ' +
                    '----manual: manual constant control specified
                       by action_manual; ' +
                    '----nominal: nominal controller, usually used
                       to benchmark optimal controllers;' +
                    '----MPC:model-predictive control; ' +
```

```
                    '----RQL: Q-learning actor-critic with Nactor-1 roll-outs of running objective; ' +
                    '----SQL: stacked Q-learning; ' +
                    '----RLStabLyap: (experimental!) learning agent with Lyapunov-like stabilizing contraints.')
parser.add_argument('--dt', type=float, metavar='dt',
                    default=0.1,
                    help='Controller sampling time.' )
parser.add_argument('--t1', type=float, metavar='t1',
                    default=20,
                    help='Final time of episode.' )
parser.add_argument('--Nruns', type=int,
                    default=1,
                    help='Number of episodes. Learned parameters are not reset after an episode.')
parser.add_argument('--is_log_data', type=int,
                    default=1,
                    help='Flag to log data into a data file. Data are stored in simdata folder.')
parser.add_argument('--is_visualization', type=int,
                    default=1
                    ,
                    help='Flag to produce graphical output.')
parser.add_argument('--is_print_sim_step', type=int,
                    default=1,
                    help='Flag to print simulation data into terminal.')
parser.add_argument('--action_manual', type=float,
                    default=[-5, -3], nargs='+',
                    help='Manual control action to be fed constant, system-specific!')
parser.add_argument('--Nactor', type=int,
                    default=15,
                    help='Horizon length (in steps) for predictive controllers.')
parser.add_argument('--pred_step_size_multiplier', type=float,
                    default=5.0,
                    help='Size of each prediction step in seconds is
```

```
                        a pred_step_size_multiplier multiple of controller sampling time dt.')
parser.add_argument('--buffer_size', type=int,
                    default=25,
                    help='Size of the buffer (experience replay) for model estimation, agent learning etc.')
parser.add_argument('--run_obj_struct', type=str,
                    default='quadratic',
                    choices=['quadratic',
                             'biquadratic'],
                    help='Structure of running objective function.')
parser.add_argument('--Q', type=float, nargs='+',
                    default=[60,60,60])
parser.add_argument('--R', type=float,nargs='+',
                    default=[1, 1])
parser.add_argument('--R1_diag', type=float, nargs='+',
                    default=[105, 105, 10, 10, 5],
                    help='Parameter of running objective function. Must have proper dimension. ' +
                    'Say, if chi = [observation, action], then a quadratic running objective reads chi.T diag(R1) chi, where diag() is transformation of a vector to a diagonal matrix.')
parser.add_argument('--Qf', type=float, nargs='+',
                    default=[10,10,10])

parser.add_argument('--R2_diag', type=float, nargs='+',
                    default=[1, 10, 1, 0, 0],
                    help='Parameter of running objective function . Must have proper dimension. ' +
                    'Say, if chi = [observation, action], then a bi-quadratic running objective reads chi**2.T diag(R2) chi**2 + chi.T diag(R1) chi, ' +
                    'where diag() is transformation of a vector to a diagonal matrix.')
parser.add_argument('--Ncritic', type=int,
                    default=25,
                    help='Critic stack size (number of temporal
```

```
difference terms in critic cost).')
parser.add_argument('--gamma', type=float,
                    default=0.9,
                    help='Discount factor.')
parser.add_argument('--critic_period_multiplier', type=float,
                    default=1.0,
                    help='Critic is updated every critic_period_multiplier times dt seconds.')
parser.add_argument('--critic_struct', type=str,
                    default='quadratic', choices=['quad-lin',
                                                  'quadratic',
                                                  'quad-nomix',
                                                  'quad-mix',
                                                  'poly3',
                                                  'poly4'],
                    help='Feature structure (critic). Currently available: ' +
                    '----quad-lin: quadratic-linear; ' +
                    '----quadratic: quadratic; ' +
                    '----quad-nomix: quadratic, no mixed terms; ' +
                    '----quad-mix: quadratic, mixed observation-action terms (for, say, Q or advantage function approximations); ' +
                    '----poly3: 3-order model, see the code for the exact structure; ' +
                    '----poly4: 4-order model, see the code for the exact structure. '
                    )
parser.add_argument('--actor_struct', type=str,
                    default='quad-nomix', choices=['quad-lin',
                                                   'quadratic',
                                                   'quad-nomix'],
                    help='Feature structure (actor). Currently available: ' +
                    '----quad-lin: quadratic-linear; ' +
                    '----quadratic: quadratic; ' +
                    '----quad-nomix: quadratic, no mixed terms.')
parser.add_argument('--init_robot_pose_x', type=float,
```

```
                    default=-3.0,
                    help='Initial x-coordinate of the robot pose.')
parser.add_argument('--init_robot_pose_y', type=float,
                    default=-3.0,
                    help='Initial y-coordinate of the robot pose.')
parser.add_argument('--init_robot_pose_theta', type=float,
                    default=1.57,
                    help='Initial orientation angle (in radians) of the robot pose.')
parser.add_argument('--distortion_x', type=float,
                    default=-0.6,
                    help='X-coordinate of the center of distortion.')
parser.add_argument('--distortion_y', type=float,
                    default=-0.5,
                    help='Y-coordinate of the center of distortion.')
parser.add_argument('--distortion_sigma', type=float,
                    default=0.1,
                    help='Standard deviation (sigma) of distortion.')
parser.add_argument('--seed', type=int,
                    default=1,
                    help='Seed for random number generation.')

args = parser.parse_args()
Nactor = int(os.getenv("NACTOR", args.Nactor))
r1_str = os.getenv("R1_DIAG")
qf_str = os.getenv("QF")
q_str = os.getenv("Q")
r_str = os.getenv("R")

# MPC: R1 (running objective)
if r1_str:
    R1 = np.diag([float(x) for x in r1_str.split()])
else:
    R1 = np.diag(args.R1_diag)

```

```
# MPC: Qf (terminal cost)
if qf_str:
    Qf = np.diag([float(x) for x in qf_str.split()])
else:
    Qf = np.diag(args.Qf)

# LQR: Q matrix (state error)
if q_str:
    Q = np.diag([float(x) for x in q_str.split()])
else:
    Q = np.diag(args.Q)

# LQR: R matrix (control effort)
if r_str:
    R = np.diag([float(x) for x in r_str.split()])
else:
    R = np.diag(args.R)

seed=args.seed
print(seed)

xdistortion_x = args.distortion_x
ydistortion_y = args.distortion_y
distortion_sigma = args.distortion_sigma

x = args.init_robot_pose_x
y = args.init_robot_pose_y
theta = args.init_robot_pose_theta

while theta > np.pi:
        theta -= 2 * np.pi
while theta < -np.pi:
        theta += 2 * np.pi

state_init = np.array([x, y, theta])

args.action_manual = np.array(args.action_manual)

```

```
pred_step_size = args.dt * args.pred_step_size_multiplier
critic_period = args.dt * args.critic_period_multiplier

Nactor = int(os.getenv("NACTOR", args.Nactor))

r1_str = os.getenv("R1_DIAG")
qf_str = os.getenv("QF")

if r1_str:
    R1 = np.diag([float(x) for x in r1_str.split()])
else:
    R1 = np.diag(np.array(args.R1_diag))

if qf_str:
    Qf = np.diag([float(x) for x in qf_str.split()])
else:
    Qf = np.diag(np.array(args.Qf))

assert args.t1 > args.dt > 0.0
assert state_init.size == dim_state

globals().update(vars(args))

#----------------------------------------Fixed settings
is_disturb = 0
is_dyn_ctrl = 0

t0 = 0

action_init = 0 * np.ones(dim_input)

# Solver
atol = 1e-3
rtol = 1e-2

# xy-plane
xMin = -4#-1.2
xMax = 2
```

```
yMin = -4#-1.2
yMax = 2

# Control constraints
v_min = -0.22 *10
v_max = 0.22 *10
omega_min = -2.84
omega_max = 2.84

ctrl_bnds=np.array([[v_min, v_max], [omega_min, omega_max]])
# ctrl_bnds=np.zeros((2,2))

#----------------------------------------Initialization : : system
my_sys = systems.Sys3WRobotNI(sys_type="diff_eqn",
                                     dim_state=dim_state,
                                     dim_input=dim_input,
                                     dim_output=dim_output,
                                     dim_disturb=dim_disturb,
                                     pars=[],
                                     ctrl_bnds=ctrl_bnds,
                                     is_dyn_ctrl=is_dyn_ctrl,
                                     is_disturb=is_disturb,
                                     pars_disturb=[])

observation_init = my_sys.out(state_init)

xCoord0 = state_init[0]
yCoord0 = state_init[1]
alpha0 = state_init[2]
alpha_deg_0 = alpha0/2*np.pi

#----------------------------------------Initialization : : model

#----------------------------------------Initialization : :
   controller

target_x=0.0
target_y=0.0
```

```
target_theta=0.0
dt=0.1
# my_ctrl_nominal = controllers.N_CTRL(k_rho=1.0, k_alpha=4.0, k_beta=-1.5, # Example gains, vary for Experiment A
#          target_x=target_x, target_y=target_y, target_theta=target_theta,
#          sampling_time=dt,
#          dim_input=dim_input, dim_output=dim_output
#     )


#  Nominal forward velocity for linearization
v_nom =0.001

# Linearized discrete system (unit time step)
A = np.array([
    [1, 0, -v_nom * dt*np.sin(0)],
    [0, 1,  v_nom *dt* np.cos(0)],
    [0, 0, 1]
])

# print(A)

B = np.array([
    [dt*np.cos(0), 0],
    [dt*np.sin(0), 0],
    [0, dt]
])

# ===============================
# 2. Define cost matrices
# ===============================

# Q penalizes state error (x, y, theta)
Q = np.diag(Q)

# R penalizes control effort (v, omega)
R = np.diag(R)
```

```
print(Q)
print(R)
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)


# Predictive optimal controller
my_ctrl_opt_pred = controllers.ControllerOptimalPredictive(dim_input,
                                                           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,
```

```
                                                                ydistortion_y,
                                                                distortion_sigma],
                                                            seed=seed)


my_ctrl_benchm = my_ctrl_opt_pred

#----------------------------------------Initialization : : simulator
my_simulator = simulator.Simulator(sys_type = "diff_eqn",
                                   closed_loop_rhs = my_sys.closed_loop_rhs,
                                   sys_out = my_sys.out,
                                   state_init = state_init,
                                   disturb_init = [],
                                   action_init = action_init,
                                   t0 = t0,
                                   t1 = t1,
                                   dt = dt,
                                   max_step = dt,
                                   first_step = 1e-4,
                                   atol = atol,
                                   rtol = rtol,
                                   is_disturb = is_disturb,
                                   is_dyn_ctrl = is_dyn_ctrl)

#----------------------------------------Initialization : : logger
date = datetime.now().strftime("%Y-%m-%d")
time = datetime.now().strftime("%Hh%Mm%Ss")
datafiles = [None] * Nruns

data_folder = 'simdata/' + ctrl_mode + "/Init_angle_{}_seed_{}_Nactor_{}".format(str(state_init[2]), seed, Nactor)

if is_log_data:
    pathlib.Path(data_folder).mkdir(parents=True, exist_ok=True)

for k in range(0, Nruns):
```

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

    if is_log_data:
        print('Logging data to:     ' + datafiles[k])

        with open(datafiles[k], 'w', newline='') as outfile:
            writer = csv.writer(outfile)
            writer.writerow(['System', my_sys.name ] )
            writer.writerow(['Controller', ctrl_mode ] )
            writer.writerow(['dt', str(dt) ] )
            writer.writerow(['state_init', str(state_init) ] )
            writer.writerow(['Nactor', str(Nactor) ] )
            writer.writerow(['pred_step_size_multiplier', str(
                pred_step_size_multiplier) ] )
            writer.writerow(['buffer_size', str(buffer_size) ] )
            writer.writerow(['run_obj_struct', str(run_obj_struct) ]
                 )
            writer.writerow(['R1_diag', str(R1_diag) ] )
            writer.writerow(['R2_diag', str(R2_diag) ] )
            writer.writerow(['Ncritic', str(Ncritic) ] )
            writer.writerow(['gamma', str(gamma) ] )
            writer.writerow(['critic_period_multiplier', str(
                critic_period_multiplier) ] )
            writer.writerow(['critic_struct', str(critic_struct) ] )
            writer.writerow(['actor_struct', str(actor_struct) ] )
            writer.writerow(['t [s]', 'x [m]', 'y [m]', 'alpha [rad]
                ', 'run_obj', 'accum_obj', 'v [m/s]', 'omega [rad/s]'
                ] )

# Do not display annoying warnings when print is on
if is_print_sim_step:
    warnings.filterwarnings('ignore')

k_rho = 2.0
k_alpha = 5.0
k_beta = -1.5
```

```python


my_logger = loggers.Logger3WRobotNI()
my_ctrl_nominal=controllers.N_CTRL(k_rho=k_rho, k_alpha=k_alpha,
    k_beta=k_beta, # Example gains, vary for Experiment A
        target_x=0.0, target_y=0.0, target_theta=0.0,
        sampling_time=dt,
        dim_input=dim_input, dim_output=dim_output)

# my_ctrl_lqr=controllers.LQR(A=np.diag([2.0,3.0,4.0]),B=np.diag
    ([2.0,3.0]),Q=np.diag([1.0,2.0,3.0]),R=np.diag([3.0,4.0]),
    obs_target=np.array([0.0,0.0,0.0]))

#----------------------------------------Main loop
state_full_init = my_simulator.state_full

if is_visualization:
     my_animator = visuals.Animator3WRobotNI(objects=(my_simulator,
                                                      my_sys,
                                                      my_ctrl_nominal
                                                         ,
                                                      my_ctrl_benchm,
                                                      my_ctrl_lqr,
                                                      datafiles,
                                                      controllers.
                                                         ctrl_selector
                                                         ,
                                                      my_logger),
                                               pars=(state_init,
                                                     action_init,
                                                     t0,
                                                     t1,
                                                     state_full_init,
                                                     xMin,
                                                     xMax,
                                                     yMin,
                                                     yMax,
                                                     ctrl_mode,
```

```
                                                                action_manual,
                                                                v_min,
                                                                omega_min,
                                                                v_max,
                                                                omega_max,
                                                                Nruns,
                                                                is_print_sim_step,
                                                                    is_log_data,
                                                                   0, [], [
                                                                   xdistortion_x,
                                                                   ydistortion_y,
                                                                   distortion_sigma
                                                                   ]))

    anm = animation.FuncAnimation(my_animator.fig_sim,
                                  my_animator.animate,
                                  init_func=my_animator.init_anim,
                                  blit=False, interval=dt/1e6,
                                     repeat=True)
    print("ALSO GOOD")
    my_animator.get_anm(anm)

    cId = my_animator.fig_sim.canvas.mpl_connect('key_press_event',
       lambda event: on_key_press(event, anm))

    anm.running = True

    my_animator.fig_sim.tight_layout()

    plt.show()

else:
    run_curr = 1
    datafile = datafiles[0]

    while True:

        my_simulator.sim_step()
```

```python

        t, state, observation, state_full = my_simulator.get_sim_step_data()

        action = controllers.ctrl_selector(t, observation, action_manual, my_ctrl_nominal, my_ctrl_benchm, my_ctrl_lqr, ctrl_mode)
        print("action: ", action,ctrl_mode)

        my_sys.receive_action(action)
        my_ctrl_benchm.receive_sys_state(my_sys._state)
        my_ctrl_benchm.upd_accum_obj(observation, action)

        xCoord = state_full[0]
        yCoord = state_full[1]
        alpha = state_full[2]
        print("sample")

        run_obj = my_ctrl_benchm.run_obj(observation, action)
        accum_obj = my_ctrl_benchm.accum_obj_val

        # count_CALF = my_ctrl_benchm.D_count()
        # count_N_CTRL = my_ctrl_benchm.get_N_CTRL_count()

        if is_print_sim_step:
            my_logger.print_sim_step(t, xCoord, yCoord, alpha, run_obj, accum_obj, action)

        if is_log_data:
            my_logger.log_data_row(datafile, t, xCoord, yCoord, alpha, run_obj, accum_obj, action)


        if t >= t1 or np.linalg.norm(observation[:2]) < 0.2:

            # Reset simulator
            my_simulator.reset()

```

```python
            if ctrl_mode != 'MPC':
                my_ctrl_benchm.reset(t0)
            elif ctrl_mode=='lqr':
                my_ctrl_lqr.reset(t0)
            else:
                my_ctrl_nominal.reset(t0)

            accum_obj = 0

            if is_print_sim_step:
                print('.....................................Run {run
                   :2d} done.....................................'.
                   format(run = run_curr))

            run_curr += 1

            if run_curr > Nruns:
                plt.close('all')
                break

            if is_log_data:
                datafile = datafiles[run_curr-1]
```

Listing 9: Experiment$_{AN}ominal.py$

```python
# IMPORT REQUIRED LIBRARIES
import pandas as pd                   # For reading and handling CSV
   files
import matplotlib.pyplot as plt       # For plotting data
import numpy as np                    # For numerical operations

# DEFINE GAIN SETS
gain_sets = [
    {"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)
    {"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)
```

```
    {"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)
]

# DEFINE FILE PATHS FOR CSV FILES GENERATED FROM SIMULATION
csv_files = ['simdata/Nominal/Init_angle_1.57_seed_1_Nactor_10/3wrobotNI_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']

# DEFINE COLORS FOR EACH RUN
colors = ['red', 'blue', 'green']

# FUNCTION TO READ CSV WHILE SKIPPING METADATA COMMENTS
def smart_read_csv(file_path):
    with open(file_path, 'r') as f:
        lines = f.readlines()
    # Identify the header line (starts with "t [s],")
    for i, line in enumerate(lines):
        if line.startswith("t [s],"):
            header_index = i
            break
    return pd.read_csv(file_path, skiprows=header_index)

# PLOT 1: ROBOT TRAJECTORY VS REFERENCE (x vs y)
plt.figure(figsize=(8, 6))
for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
    data = smart_read_csv(file)
    data.columns = [col.strip() for col in data.columns]

    # Plot the actual robot trajectory
    plt.plot(data['x [m]'], data['y [m]'], label=f'Run {i}', color=color)
```

```python

plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.title('Robot Trajectories (Kinematic Controller)')
plt.legend()
plt.grid(True)
plt.axis('equal')  # x-y plot, so equal scaling makes sense
plt.savefig('simdata/Robot_Trajectories_Kinematics.png')
plt.close()

# PLOT 2: LINEAR VELOCITY v(t)
plt.figure(figsize=(8, 6))
for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
    data = smart_read_csv(file)
    data.columns = [col.strip() for col in data.columns]

    # Plot v [m/s] over time
    plt.plot(data['t [s]'], data['v [m/s]'], label=f'Run {i}', color
       =color)

plt.xlabel('t [s]')
plt.ylabel('v [m/s]')
plt.title('Linear Velocity Over Time')
plt.legend()
plt.grid(True)
plt.savefig('simdata/Linear_Velocity_Kinematics.png')
plt.close()

# PLOT 3: ANGULAR VELOCITY omega(t)
plt.figure(figsize=(8, 6))
for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
    data = smart_read_csv(file)
    data.columns = [col.strip() for col in data.columns]

    # Plot omega [rad/s] over time
    plt.plot(data['t [s]'], data['omega [rad/s]'], label=f'Run {i}',
        color=color)

```

```
plt.xlabel('t [s]')
plt.ylabel('omega [rad/s]')
plt.title('Angular Velocity Over Time')
plt.legend()
plt.grid(True)
plt.savefig('simdata/Angular_Velocity_Kinematics.png')
plt.close()

# PLOT 4: TRACKING ERROR VS TIME (Using fixed goal)
plt.figure(figsize=(8, 6))

# Define your fixed goal location (set your actual goal here)
x_goal, y_goal = 0.0, 0.0

for i, (file, color) in enumerate(zip(csv_files, colors), start=1):
    data = smart_read_csv(file)
    data.columns = [col.strip() for col in data.columns]

    # Compute distance to fixed goal at every time step
    error = np.sqrt((data['x [m]'] - x_goal)**2 + (data['y [m]'] -
       y_goal)**2)

    # Plot tracking error over time
    plt.plot(data['t [s]'], error, label=f'Run {i}', color=color)

plt.xlabel('t [s]')
plt.ylabel('Tracking Error [m]')
plt.title('Tracking Error Over Time')
plt.legend()
plt.grid(True)
plt.savefig('simdata/Tracking_Error_Over_Time.png')
plt.close()
# PRINT SUCCESS MESSAGE
print("Plots Saved:")
print("      Robot_Trajectories_Kinematics.png")
print("      Linear_Velocity_Kinematics.png")
print("      Angular_Velocity_Kinematics.png")
print("      Tracking_Error_Over_Time.png")
```

Listing 10: Experiment$_{BL}QR.py$

```python
# IMPORT REQUIRED LIBRARIES
import os
import subprocess
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from datetime import datetime
import glob

# DEFINE COST MATRICES Q AND R FOR TESTING
cost_sets = [
    {"Q": [10.0, 12.0, 8.0], "R": [1.0, 1.0]},
    {"Q": [100.0, 100.0, 50.0], "R": [0.1, 0.1]},
    {"Q": [5.0, 5.0, 2.0], "R": [10.0, 10.0]}
]

# LOG FOLDER SETUP
log_folder = "simdata/lqr/Init_angle_1.57_seed_1_Nactor_10/"
os.makedirs(log_folder, exist_ok=True)

# RUN SIMULATIONS WITH VARYING Q AND R
for i, cost in enumerate(cost_sets):
    print(f"\n Running LQR Simulation {i+1} with Q={cost['Q']} and R={cost['R']}")

    os.environ["Q_VALS"] = " ".join(str(x) for x in cost["Q"])
    os.environ["R_VALS"] = " ".join(str(x) for x in cost["R"])
    print("Environment variables set:", os.environ["Q_VALS"], os.environ["R_VALS"])

    subprocess.run([
        "python3", "PRESET_3wrobot_NI.py",
        "--ctrl_mode", "lqr",
        "--Nruns", "1",
        "--t1", "20",
        "--is_visualization", "0",
        "--is_log_data", "1",
```

```
        "--Q", *map(str, cost["Q"]),
        "--R", *map(str, cost["R"]),
        "--v_max", "1.0",               # Actuator limit: max linear
            velocity
        "--omega_max", "1.0"            # Actuator limit: max
           angular velocity
    ])

# FIND LATEST LQR LOG FILES
def find_latest_lqr_csvs(folder, prefix="3wrobotNI_lqr_", run_suffix
   ="__run01.csv", count=3):
    pattern = os.path.join(folder, f"{prefix}*{run_suffix}")
    all_files = glob.glob(pattern)
    all_files.sort(key=os.path.getmtime, reverse=True)
    return all_files[:count]

print("\n Plotting Results")

# READ THE CSV FILES
csvs = find_latest_lqr_csvs(log_folder)

if not csvs:
    print("No CSV files found for plotting.")
    exit()

# FUNCTION TO READ CSV SKIPPING HEADER LINES BEFORE ACTUAL DATA
def smart_read_csv(file_path):
    with open(file_path, 'r') as f:
        lines = f.readlines()
    for i, line in enumerate(lines):
        if line.startswith("t [s],"):
            header_index = i
            break
    return pd.read_csv(file_path, skiprows=header_index)

# READ AND STORE DATAFRAMES FROM CSV FILES
dfs = [smart_read_csv(f) for f in csvs]
print(f"Found {len(dfs)} CSV files for plotting.")
```

```python

# DEFINE COLORS FOR EACH RUN
colors = ['b', 'g', 'r']

# PLOT 1: TRAJECTORIES (x vs y)
plt.figure(figsize=(10, 6))
for i, df in enumerate(dfs):
    plt.plot(df['x [m]'], df['y [m]'], label=f"Run {i+1} - Q={cost_sets[i]['Q']}, R={cost_sets[i]['R']}", color=colors[i])
plt.title("LQR: Trajectory (x vs y)")
plt.xlabel("x [m]")
plt.ylabel("y [m]")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("simdata/lqr/Trajectory_Plot.png")
plt.close()

# PLOT 2: TRACKING ERROR VS TIME (distance to origin)
plt.figure(figsize=(10, 6))
for i, df in enumerate(dfs):
    error = np.sqrt(df['x [m]']**2 + df['y [m]']**2)
    plt.plot(df['t [s]'], error, label=f"Run {i+1}", color=colors[i])

    # Highlight runs with large steady-state error
    if error.iloc[-1] > 0.5:
        print(f" Run {i+1} may not converge properly: final error = {error.iloc[-1]:.2f} m")

plt.title("LQR: Tracking Error vs Time")
plt.xlabel("Time [s]")
plt.ylabel("Position Error [m]")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("simdata/lqr/Tracking_Error_Plot.png")
plt.close()
```

```

# PLOT 3: CONTROL INPUTS (v and omega vs time)
plt.figure(figsize=(10, 6))
for i, df in enumerate(dfs):
    plt.plot(df['t [s]'], df['v [m/s]'], label=f"v - Run {i+1}",
        color=colors[i])
    plt.plot(df['t [s]'], df['omega [rad/s]'], '--', label=f"   -
        Run {i+1}", color=colors[i])
plt.title("LQR: Control Inputs Over Time")
plt.xlabel("Time [s]")
plt.ylabel("Input Values")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("simdata/lqr/Control_Inputs_Plot.png")
plt.close()
# PRINT SUCCESS MESSAGE
print(" Plots Saved:")
print("      Trajectory_Plot.png")
print("      Tracking_Error_Plot.png")
print("      Control_Inputs_Plot.png")
```

Listing 11: Experiment$_{CM}PC.py$

```
# IMPORT REQUIRED LIBRARIES
import os
import subprocess
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import glob

# DEFINE MPC CONFIGURATION SET WITH VARIATIONS
mpc_configs = [
    {"Nactor": 5, "R1_diag": [100, 100, 10, 10, 5], "Qf": [0, 0,
        0]},        # No terminal cost, short horizon
    {"Nactor": 15, "R1_diag": [100, 100, 10, 10, 5], "Qf": [100,
        100, 10]},  # Long horizon + terminal cost
    {"Nactor": 10, "R1_diag": [300, 300, 30, 5, 1], "Qf": [150, 150,
```

```
        15]},   # Medium horizon, high R weight
]

# LOG FOLDER SETUP
log_folder = "simdata/MPC/"
os.makedirs(log_folder, exist_ok=True)

# RUM SIMULATIONS USING SUBPROCESS CALL
for i, config in enumerate(mpc_configs):
    print(f"\n Running MPC Simulation {i+1} with Nactor={config['
        Nactor']}, Qf={config['Qf']}")

    # EXPORT VARIABLES (IN CASE THE CONTROLLER READS FROM ENV)
    os.environ["NACTOR"] = str(config["Nactor"])
    os.environ["R1_DIAG"] = " ".join(str(x) for x in config["R1_diag
        "])
    os.environ["QF"] = " ".join(str(x) for x in config["Qf"])

    # RUN SIMULATION SCRIPT
    subprocess.run([
        "python3", "PRESET_3wrobot_NI.py",
        "--ctrl_mode", "MPC",
        "--Nruns", "1",
        "--t1", "20",
        "--is_visualization", "0",
        "--is_log_data", "1",
        "--Nactor", str(config["Nactor"]),
        "--R1_diag", *map(str, config["R1_diag"]),
        "--Qf", *map(str, config["Qf"]),
    ])

# GET LATEST LOG FILES FROM EACH RUN
def get_latest_csv_from_each_subfolder(main_folder, pattern="3
    wrobotNI_MPC_*__run01.csv"):
    """Find the latest CSV file matching pattern in each subfolder.
        """
    csv_paths = []
    subfolders = [os.path.join(main_folder, d) for d in os.listdir(
```

```
        main_folder)
                    if os.path.isdir(os.path.join(main_folder, d))]
    subfolders.sort()  # Sort for consistency
    subfolders = subfolders[:3]  # Limit to 3 experiments

    for folder in subfolders:
        match = glob.glob(os.path.join(folder, pattern))
        if match:
            match.sort(key=os.path.getmtime, reverse=True)
            csv_paths.append(match[0])
        else:
            print(f"No matching CSV in: {folder}")

    return csv_paths

csvs = get_latest_csv_from_each_subfolder(log_folder)

if not csvs:
    print("No CSV files found for plotting.")
    exit()

# DEFINE HELPER TO READ CSV DATA SKIPPING COMMENT LINES
def smart_read_csv(file_path):
    with open(file_path, 'r') as f:
        lines = f.readlines()
    header_index = next(i for i, line in enumerate(lines) if line.
        startswith("t [s],"))
    return pd.read_csv(file_path, skiprows=header_index)

# READ DATA FROM CSV FILES
dfs = [smart_read_csv(f) for f in csvs]

# DEFINE COLORS FOR EACH RUN
colors = ['b', 'g', 'r']

# PLOT 1: PLOT TRAJECTORY (x vs y)
plt.figure(figsize=(8, 6))
for i, df in enumerate(dfs):
```

```
    plt.plot(df['x [m]'], df['y [m]'], label=f"Run {i+1} - N={mpc_configs[i]['Nactor']}", color=colors[i])
plt.title("MPC Trajectory (x vs y)")
plt.xlabel("x [m]")
plt.ylabel("y [m]")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("simdata/MPC/MPC_Trajectory.png")
plt.close()

# PLOT 2: TRACKING ERROR VS TIME
plt.figure(figsize=(8, 6))
for i, df in enumerate(dfs):
    error = np.sqrt(df['x [m]']**2 + df['y [m]']**2)
    plt.plot(df['t [s]'], error, label=f"Run {i+1}", color=colors[i])
plt.title("MPC Tracking Error vs Time")
plt.xlabel("Time [s]")
plt.ylabel("Position Error [m]")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("simdata/MPC/MPC_Tracking_Error.png")
plt.close()

# PLOT 3: CONTROL INPUTS (V) VS OMEGA OVER TIME
plt.figure(figsize=(10, 6))
for i, df in enumerate(dfs):
    plt.plot(df['t [s]'], df['v [m/s]'], label=f"v - Run {i+1}", color=colors[i])
    plt.plot(df['t [s]'], df['omega [rad/s]'], '--', label=f"  - Run {i+1}", color=colors[i])
plt.title("MPC Control Inputs (v and  ) vs Time")
plt.xlabel("Time [s]")
plt.ylabel("Control Inputs")
plt.legend()
plt.grid(True)
```

```
plt.tight_layout()
plt.savefig("simdata/MPC/MPC_Control_Inputs.png")
plt.close()

# PLOT 4: ACCUMULATED COST OVER TIME
plt.figure(figsize=(8, 6))
for i, df in enumerate(dfs):
    if 'accum_obj' in df.columns:
        plt.plot(df['t [s]'], df['accum_obj'], label=f"Run {i+1}",
            color=colors[i])
plt.title("MPC Accumulated Cost vs Time")
plt.xlabel("Time [s]")
plt.ylabel("Accumulated Cost")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("simdata/MPC/MPC_Accumulated_Cost.png")
plt.close()

print("Plots saved:")
print("      MPC_Trajectory.png")
print("      MPC_Tracking_Error.png")
print("      MPC_Control_Inputs.png")
print("      MPC_Accumulated_Cost.png")
```