Coursework Part 3: Data-driven Model Predictive Control

Problem Definition

In this coursework, you will design a data-driven model predictive controller (MPC) to control a multistage extraction column. You have complete freedom to create your own data-gathering routine, data-driven model (neural network, decision tree etc.) and control horizon optimisation routine which all work together to form the controller. Since the takeaway is on optimising chemical engineering systems, we operate under the following assumption:

- Evaluations are expensive, meaning that the runtime of the model training is limited by a fixed evaluation budget rather than a time budget constraint. This means that your exploration routine can only evaluate the system ***5*** times.
- The data-gathering, model training and the control horizon optimisation routine will also be time limited to a budget of 2 mins for data-gathering and model training, and 5 min for controller optimisation (time to complete one simulation with the controller active)

Your team's submission will be three functions: 1. Data-gathering routine 2. Model training 3. Controller. These form your best attempt at controlling the multistage extraction column system using a data-driven MPC.

Next steps

It is advised to read this handout thoroughly to understand the submission format and criteria. Then, to develop your understanding, it is recommended to follow the walkthrough included to gain an understanding of the script *control_task.py* and how you might want to use *evaluate.py* to test your algorithm. Furthermore, you should complete the data-driven control tutorial which will help you develop your understanding.

Installation Instructions

The installation instructions can be found on the GitHub repository: OptiMaL-PSE-Lab/DDMPC-Coursework: DDMPC Coursework for the ML4CE Course at ICL

Material Provided

your_alg.py

- This script provides an example data-driven controller with a random explorer (*see* explorer function), a linear model (*model trainer)* and the data-driven MPC.
- Your final submission must also be in the form of a *.py* file, contained in a similar function format that requires the same arguments and returns the same outputs as this script. Essentially, you may use it as a template for your algorithm.

control_task.py

- This script creates the multistage extraction columns model using the python package pc-gym. Then the three phases to the implementation of your data-driven MPC: exploration, model training and control. Finally, the evaluation which is weighted sum of the setpoint error and control effort.

evaluate.py

- This script provides an example of how your algorithm will be evaluated over multiple repetitions and setpoints.

Grading (per group):

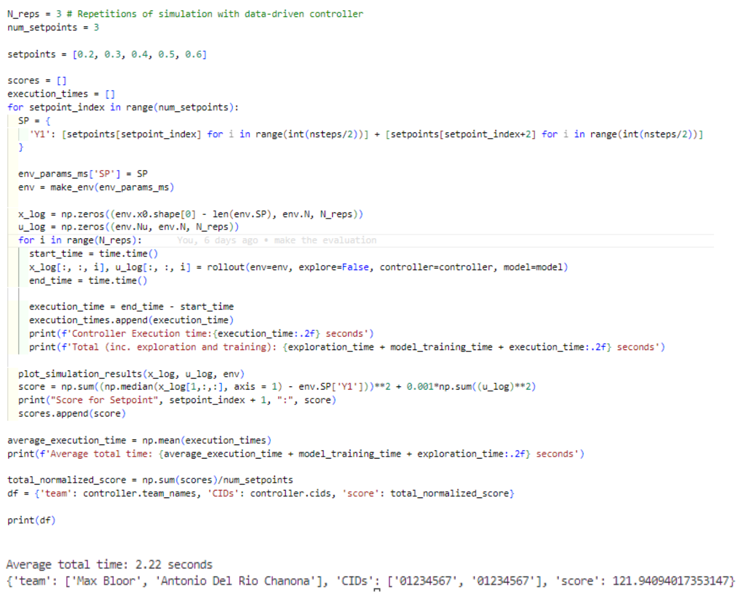
- 2 page Report on your algorithm (20%).
    - The report should have the following sections:
        - Big picture explanation and intuition behind the algorithm
        - Methodology
        - Pseudocode
        - You are allowed a figure for your algorithm which is not considered in the report length.
        - References are not considered for the length of the report.
    - The report should include a pseudocode following the format specified here.
    - The report should also explain the rational behind the algorithm and in paragraph form the main steps in the algorithm.
    - Please do not use a letter smaller than size 11 (with a decent font: e.g., Arial, Times New Roman, Calibri, Latex font), and margins no smaller than 2cm Top, Bottom, Left, Right.
    - You will be graded based on clarity of communication, creativity of your algorithm and scientific explanation of your methods.
- Your implementation will be graded using a similar script to *evaluate.py* provided (80%).
    - You will upload a single python file ".py" such that the file is *"team_name_part3.py"* and the algorithm to be called is *"algorithm_team_name"*.

IMPORTANT:

- Good coding practice: as aforementioned, make sure your function takes the same inputs and outputs as the example functions provided. This is **very important**, as otherwise, we have no way to test your algorithms and provide a mark. This is easily mitigated by making sure your submission runs in *evaluate.py* and ensuring it follows the format of the *your_alg.py* script (parameters and return values).
- Stochasticity: Do not set random seeds to your algorithm.
- Packages: please restrict to use numpy, scipy, sobol-seq, random, time, pc-gym and scikit-learn. You are allowed to use scipy.optimize to train any surrogates if you wish, but you are not allowed to optimize f() directly. In other words, you are not allowed to use the optimization routines of python packages directly on the functions. **Make absolutely sure** that your code runs in an environment where **only** numpy, scipy, sobol-seq, random, time, pcgym and scikit-learn packages are available.
- Ensure you add your team's names and CIDs to your function, as shown in the example provided.

Algorithm Grading

To help you evaluate your algorithm’s performance and guide its development, you have been provided with a simplified version of the script (*evaluate.py*) that will be used to assess your final algorithm submission.



```python
N_reps = 3 # Repetitions of simulation with data-driven controller
num_setpoints = 3

setpoints = [0.2, 0.3, 0.4, 0.5, 0.6]

scores = []
execution_times = []
for setpoint_index in range(num_setpoints):
  SP = {
    'Y1': [setpoints[setpoint_index] for i in range(int(nsteps/2))] + [setpoints[setpoint_index+2] for i in range(int(nsteps/2))]
  }

  env_params_ms['SP'] = SP
  env = make_env(env_params_ms)

  x_log = np.zeros((env.x0.shape[0] - len(env.SP), env.N, N_reps))
  u_log = np.zeros((env.Nu, env.N, N_reps))
  for i in range(N_reps):        You, 6 days ago • make the evaluation
    start_time = time.time()
    x_log[:, :, i], u_log[:, :, i] = rollout(env=env, explore=False, controller=controller, model=model)
    end_time = time.time()

    execution_time = end_time - start_time
    execution_times.append(execution_time)
    print(f'Controller Execution time:{execution_time:.2f} seconds')
    print(f'Total (inc. exploration and training): {exploration_time + model_training_time + execution_time:.2f} seconds')

  plot_simulation_results(x_log, u_log, env)
  score = np.sum((np.median(x_log[1,:,:], axis = 1) - env.SP['Y1']))**2 + 0.001*np.sum((u_log)**2)
  print("Score for Setpoint", setpoint_index + 1, ":", score)
  scores.append(score)

average_execution_time = np.mean(execution_times)
print(f'Average total time: {average_execution_time + model_training_time + exploration_time:.2f} seconds')

total_normalized_score = np.sum(scores)/num_setpoints
df = {'team': controller.team_names, 'CIDs': controller.cids, 'score': total_normalized_score}

print(df)
```

```
Average total time: 2.22 seconds
{'team': ['Max Bloor', 'Antonio Del Rio Chanona'], 'CIDs': ['01234567', '01234567'], 'score': 121.94094017353147}
```

This is the code block which evaluates the controller above which is the same as the *control_task.py*.

The code then cycles through three different setpoint combinations to fully evaluate the controller. For each setpoint the controller is simulated three times to record an average performance. Your complete algorithm is repeated five times and the best (smallest) score is return

There is an example output from *evaluate.py*. Please remember to add your team’s names and

Your grade will be determined by the score your algorithm achieves on a similar script. The actual grading script will change the setpoints of the Multistage Extraction Column such that no algorithm can achieve an unfair advantage from over-tuning to a specific instance.

Problem Walkthrough

1. Multistage Extraction Model & Simulation

The multistage extraction model describes a counter-current liquid-gas extraction process with five stages. It captures the mass transfer of a solute between the liquid and gas phases in each stage. The model consists of ten ordinary differential equations (ODEs) representing the concentration dynamics of the solute in both phases for each stage. The model is setup using the pc-gym package as shown below which includes the code to simulate the model.

$$\frac{dX_n}{dt} = \frac{1}{V_l}\left(L(X_{n-1} - X_n) - Q_n\right)$$

$$\frac{dY_n}{dt} = \frac{1}{V_g}\left(G(Y_{n-1} - Y_n) + Q_n\right)$$

$$X_{n,eq} = \frac{Y_n^e}{m}$$

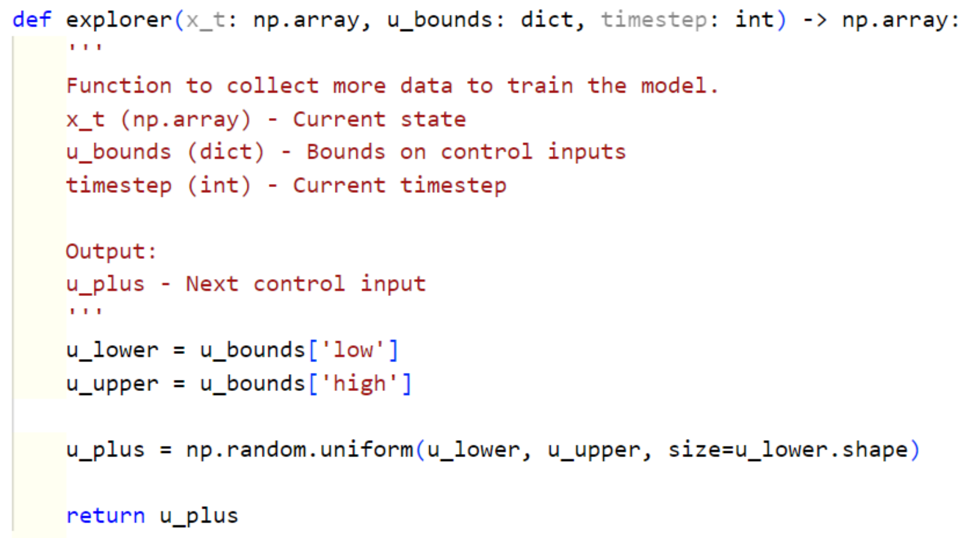
$$Q_n = K_{la}\left(X_n - X_{n,eq}\right)V_l$$

```python
##########################################
#  Multistage Extraction Column model #
##########################################
T = 100
nsteps = 60
SP = {
        'Y1': [0.5 for i in range(int(nsteps/2))] + [0.7 for i in range(int(nsteps/2))]
    }

action_space = {
    'low': np.array([5, 10]),
    'high':np.array([500, 1000])
}

observation_space = {
    'low' : np.array([0]*10+[0.3]+[0.3]),
    'high' : np.array([1]*10+[0.4]+[0.4])
}

env_params_ms = {
    'N': nsteps,
    'tsim':T,
    'SP':SP,
    'o_space' : observation_space,
    'a_space' : action_space,
    'x0': np.array([0.55, 0.3, 0.45, 0.25, 0.4, 0.20, 0.35, 0.15, 0.25, 0.1,0.3,0.3]),
    'model': 'multistage_extraction',
    'noise':True, #Add noise to the states
    'noise_percentage':0.01,
    'integration_method': 'casadi',
    'custom_reward': reward_fn,
    'normalise_o': False,
    'normalise_a':False,
}

env = make_env(env_params_ms)
```

2. Explorer

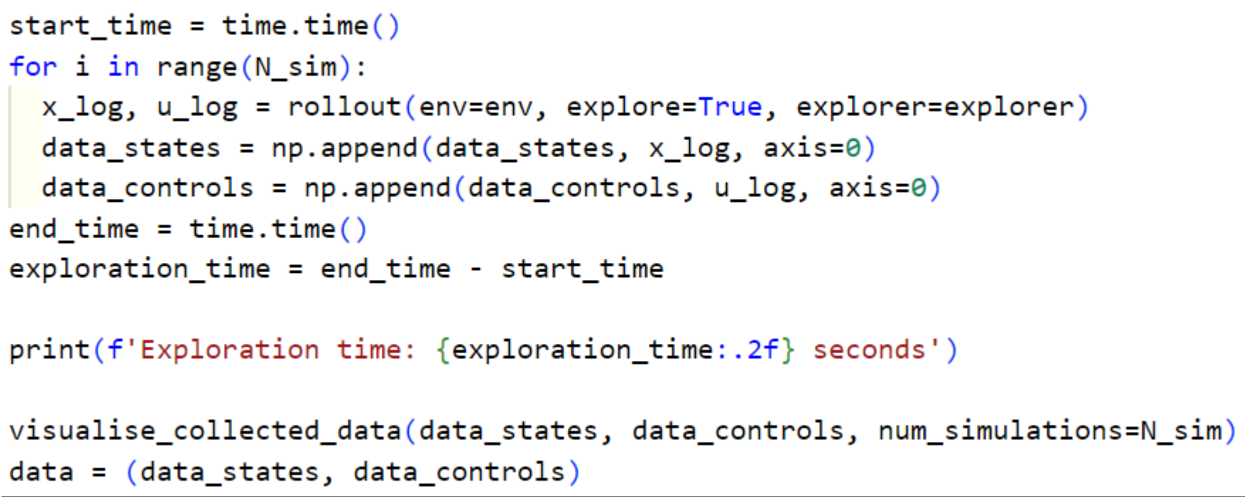
During the first phase of your algorithm, additional data from the system can be collected to improve the model accuracy and subsequent control in the following phases. This is ***important*** to your model and subsequent controller's performance. In the example code below, the control bounds are sampled uniformly at each timestep to produce the next control input this process is repeated for 5 separate simulations each of 60 timesteps. This data is combined with the provided data found in the "time_series" folder to form the dataset for training your model.

In the example algorithm (your_alg.py):



```python
def explorer(x_t: np.array, u_bounds: dict, timestep: int) -> np.array:
    '''
    Function to collect more data to train the model.
    x_t (np.array) - Current state
    u_bounds (dict) - Bounds on control inputs
    timestep (int) - Current timestep

    Output:
    u_plus - Next control input
    '''
    u_lower = u_bounds['low']
    u_upper = u_bounds['high']

    u_plus = np.random.uniform(u_lower, u_upper, size=u_lower.shape)

    return u_plus
```

In the evaluation (evaluate.py):



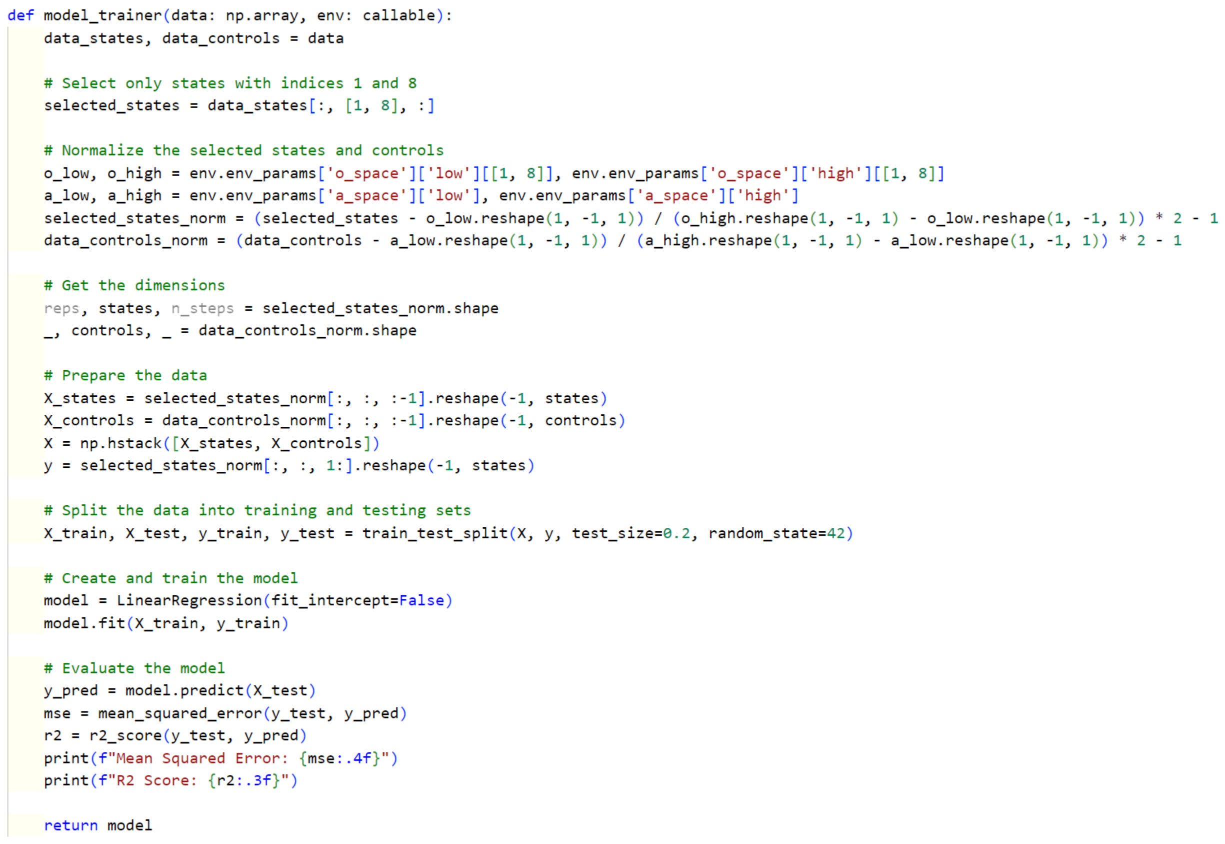
```python
start_time = time.time()
for i in range(N_sim):
  x_log, u_log = rollout(env=env, explore=True, explorer=explorer)
  data_states = np.append(data_states, x_log, axis=0)
  data_controls = np.append(data_controls, u_log, axis=0)
end_time = time.time()
exploration_time = end_time - start_time

print(f'Exploration time: {exploration_time:.2f} seconds')

visualise_collected_data(data_states, data_controls, num_simulations=N_sim)
data = (data_states, data_controls)
```

3. <u>Model Trainer</u>

In this phase, a model of the multistage extraction column is trained on the combined dataset which was created in the previous phase. In the example algorithm specific selected states (gas concentration in the first stage and liquid in the last stage) are normalised then used to create a linear model W. The model created takes the current state x_t and control input u_t as inputs and outputs the next state x_{t+1}:

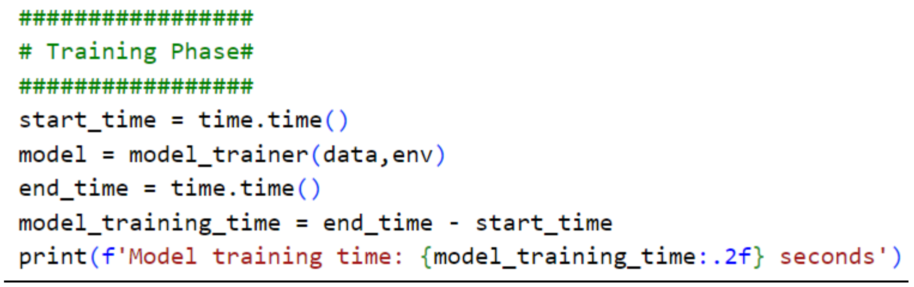
$$x_{t+1} = W(x_t, u_t)$$

<u>In the example algorithm (your_alg.py):</u>



```python
def model_trainer(data: np.array, env: callable):
    data_states, data_controls = data

    # Select only states with indices 1 and 8
    selected_states = data_states[:, [1, 8], :]

    # Normalize the selected states and controls
    o_low, o_high = env.env_params['o_space']['low'][[1, 8]], env.env_params['o_space']['high'][[1, 8]]
    a_low, a_high = env.env_params['a_space']['low'], env.env_params['a_space']['high']
    selected_states_norm = (selected_states - o_low.reshape(1, -1, 1)) / (o_high.reshape(1, -1, 1) - o_low.reshape(1, -1, 1)) * 2 - 1
    data_controls_norm = (data_controls - a_low.reshape(1, -1, 1)) / (a_high.reshape(1, -1, 1) - a_low.reshape(1, -1, 1)) * 2 - 1

    # Get the dimensions
    reps, states, n_steps = selected_states_norm.shape
    _, controls, _ = data_controls_norm.shape

    # Prepare the data
    X_states = selected_states_norm[:, :, :-1].reshape(-1, states)
    X_controls = data_controls_norm[:, :, :-1].reshape(-1, controls)
    X = np.hstack([X_states, X_controls])
    y = selected_states_norm[:, :, 1:].reshape(-1, states)

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Create and train the model
    model = LinearRegression(fit_intercept=False)
    model.fit(X_train, y_train)

    # Evaluate the model
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print(f"Mean Squared Error: {mse:.4f}")
    print(f"R2 Score: {r2:.3f}")

    return model
```

<u>In the evaluation (evaluate.py)</u>

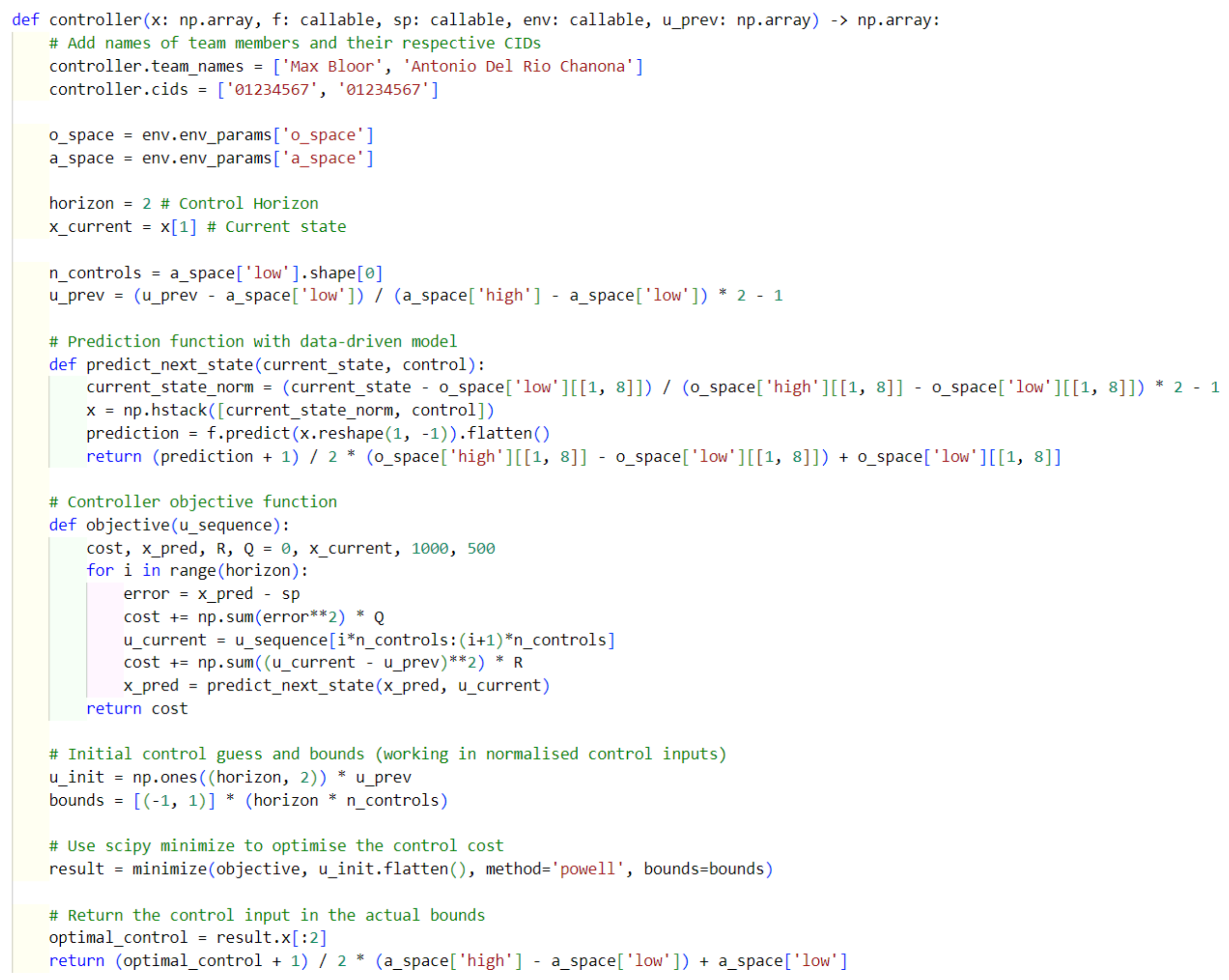


```python
##################
# Training Phase#
##################
start_time = time.time()
model = model_trainer(data,env)
end_time = time.time()
model_training_time = end_time - start_time
print(f'Model training time: {model_training_time:.2f} seconds')
```

4. <u>Controller</u>

Once the model has been trained, we can use this to create a controller for the system. The below example controller starts with adding the team member names and CIDs as attributes of the function. Then we create a function to automatically construct the inputs to the model and return the prediction. Then we define the controller's objective function which is a combination of setpoint error and control effort. This is then minimized, and the resulting control input is returned. The controller is sensitive to the parameters *horizon*, *R,* and *Q*.

<u>In the example algorithm (your_alg.py):</u>



```python
def controller(x: np.array, f: callable, sp: callable, env: callable, u_prev: np.array) -> np.array:
    # Add names of team members and their respective CIDs
    controller.team_names = ['Max Bloor', 'Antonio Del Rio Chanona']
    controller.cids = ['01234567', '01234567']

    o_space = env.env_params['o_space']
    a_space = env.env_params['a_space']

    horizon = 2 # Control Horizon
    x_current = x[1] # Current state

    n_controls = a_space['low'].shape[0]
    u_prev = (u_prev - a_space['low']) / (a_space['high'] - a_space['low']) * 2 - 1

    # Prediction function with data-driven model
    def predict_next_state(current_state, control):
        current_state_norm = (current_state - o_space['low'][[1, 8]]) / (o_space['high'][[1, 8]] - o_space['low'][[1, 8]]) * 2 - 1
        x = np.hstack([current_state_norm, control])
        prediction = f.predict(x.reshape(1, -1)).flatten()
        return (prediction + 1) / 2 * (o_space['high'][[1, 8]] - o_space['low'][[1, 8]]) + o_space['low'][[1, 8]]

    # Controller objective function
    def objective(u_sequence):
        cost, x_pred, R, Q = 0, x_current, 1000, 500
        for i in range(horizon):
            error = x_pred - sp
            cost += np.sum(error**2) * Q
            u_current = u_sequence[i*n_controls:(i+1)*n_controls]
            cost += np.sum((u_current - u_prev)**2) * R
            x_pred = predict_next_state(x_pred, u_current)
        return cost

    # Initial control guess and bounds (working in normalised control inputs)
    u_init = np.ones((horizon, 2)) * u_prev
    bounds = [(-1, 1)] * (horizon * n_controls)

    # Use scipy minimize to optimise the control cost
    result = minimize(objective, u_init.flatten(), method='powell', bounds=bounds)

    # Return the control input in the actual bounds
    optimal_control = result.x[:2]
    return (optimal_control + 1) / 2 * (a_space['high'] - a_space['low']) + a_space['low']
```

The controller is then used within the evaluation to control a range of setpoints which are repeated to produce a normalised score.

In the evaluation (evaluate.py):

```
N_reps = 3 # Repetitions of simulation with data-driven controller
num_setpoints = 3

setpoints = [0.2, 0.3, 0.4, 0.5, 0.6]

scores = []
execution_times = []
for setpoint_index in range(num_setpoints):
  SP = {
    'Y1': [setpoints[setpoint_index] for i in range(int(nsteps/2))] + [setpoints[setpoint_index+2] for i in range(int(nsteps/2))]
  }

  env_params_ms['SP'] = SP
  env = make_env(env_params_ms)

  x_log = np.zeros((env.x0.shape[0] - len(env.SP), env.N, N_reps))
  u_log = np.zeros((env.Nu, env.N, N_reps))
  for i in range(N_reps):
    start_time = time.time()
    x_log[:, :, i], u_log[:, :, i] = rollout(env=env, explore=False, controller=controller, model=model)
    end_time = time.time()

    execution_time = end_time - start_time
    execution_times.append(execution_time)
    print(f'Controller Execution time:{execution_time:.2f} seconds')
    print(f'Total (inc. exploration and training): {exploration_time + model_training_time + execution_time:.2f} seconds')

  plot_simulation_results(x_log, u_log, env)
  score = np.sum((np.median(x_log[1,:,:], axis = 1) - env.SP['Y1'])**2 + 0.001*np.sum((u_log)**2)
  print("Score for Setpoint", setpoint_index + 1, ":", score)
  scores.append(score)

average_execution_time = np.mean(execution_times)
print(f'Average total time: {average_execution_time + model_training_time + exploration_time:.2f} seconds')

total_normalized_score = np.sum(scores)/num_setpoints
df = {'team': controller.team_names, 'CIDs': controller.cids, 'score': total_normalized_score}
```