## 24784 Project 2

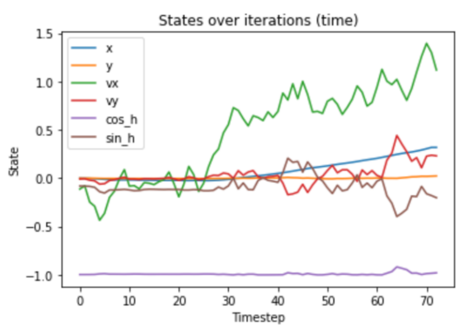
## Alonso Buitano Tang

### Exercise 1. Understanding the Parking-v0 environment.

1. Run one episode with random actions. Check the observation dictionary returned by the environment. Store the observations (states) and rewards over time! What is your target state (desired goal) s∗? What is your final state sn at the 100-th time step? Plot the values of all the states over time!

```
  logger.warn(

Observation format: OrderedDict([('observation', array([ 0.31849487,  0.022848  ,  1.11902165,  0.2309222 , -0.9793
6439,
       -0.20210242])), ('achieved_goal', array([ 0.31849487,  0.022848  ,  1.11902165,  0.2309222 , -0.97936439,
       -0.20210242])), ('desired_goal', array([-1.400000e-01,  1.400000e-01,  0.000000e+00,  0.000000e+00,
        6.123234e-17,  1.000000e+00]))])
****************************************************************************************************
s_n and s^* are stored in obs variable, final state after 73 steps is:
[ 0.31849487  0.022848    1.11902165  0.2309222  -0.97936439 -0.20210242]
desired_goal (target) is:
[-1.400000e-01  1.400000e-01  0.000000e+00  0.000000e+00  6.123234e-17
  1.000000e+00]
```

In [7]:
```
states_labels = env.config['observation']['features'] #the labels for the states

plt.plot(states)
plt.title("States over iterations (time)")
plt.xlabel("Timestep")
plt.ylabel("State")
plt.legend(states_labels)
plt.show()
```



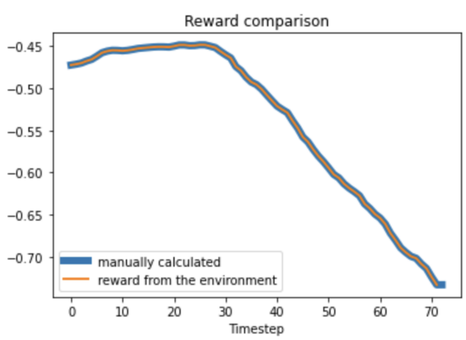
2. Calculate the reward of the environment manually. The reward rt is defined as the negative of the weighted Euclidean norm of |st − s∗|. (Hint:
Compare your manual calculation with the environment reward to check if you are correct!)

In [8]:

```
#Negative reward
states_labels = env.config['observation']['features'] #the labels for the states
states_scales = env.config['observation']['scales'] #the scales for the states (if needed for plotting)

print(states_labels)
print(states_scales)

weights = np.array([1, 0.3, 0, 0, 0.02, 0.02]) #weights
weighted_norm = - np.sqrt(np.sum(np.abs(states - obs["desired_goal"])*weights, axis=-1)) # REPLACE THE T

#plot the weighted norm over time
# REPLACE THE THREE DOTS WITH YOUR OWN CODE
plt.plot(weighted_norm)
plt.title("Reward over time")
plt.xlabel("Timestep")
plt.ylabel("Reward")
plt.show()

```

```
['x', 'y', 'vx', 'vy', 'cos_h', 'sin_h']
[100, 100, 5, 5, 1, 1]
```


Reward comparison:



3. Plot the positions (x and y coordinate) of the vehicle! What is the reward at the final time step in the episode? Show the rendered image of the environment at the final time step in the episode!

In [10]:
```
# final reward
print(f"Final reward at {len(rewards)} timesteps: {rewards[-1]} (collision)")
print(f"Reward at {len(rewards)-1} timesteps: {rewards[-2]}") # REPLACE THE THREE DOTS WITH YOUR OWN CODE

# Plot the trajectory of the vehicle (x and y position over timestep)
plt.plot(np.array(states)[:,0], np.array(states)[:,1]) # REPLACE THE THREE DOTS WITH YOUR OWN CODE
plt.xlabel("x")
plt.ylabel("y")
plt.title("x and y position over timestep")
# plt.xlim(-10, 10)
# plt.ylim(-5, 5)
plt.show()
```

```
Final reward at 73 timesteps: -500.73298690378186 (collision)
Reward at 72 timesteps: -0.7330224497721212
```

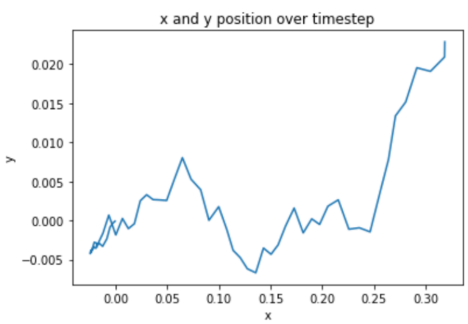
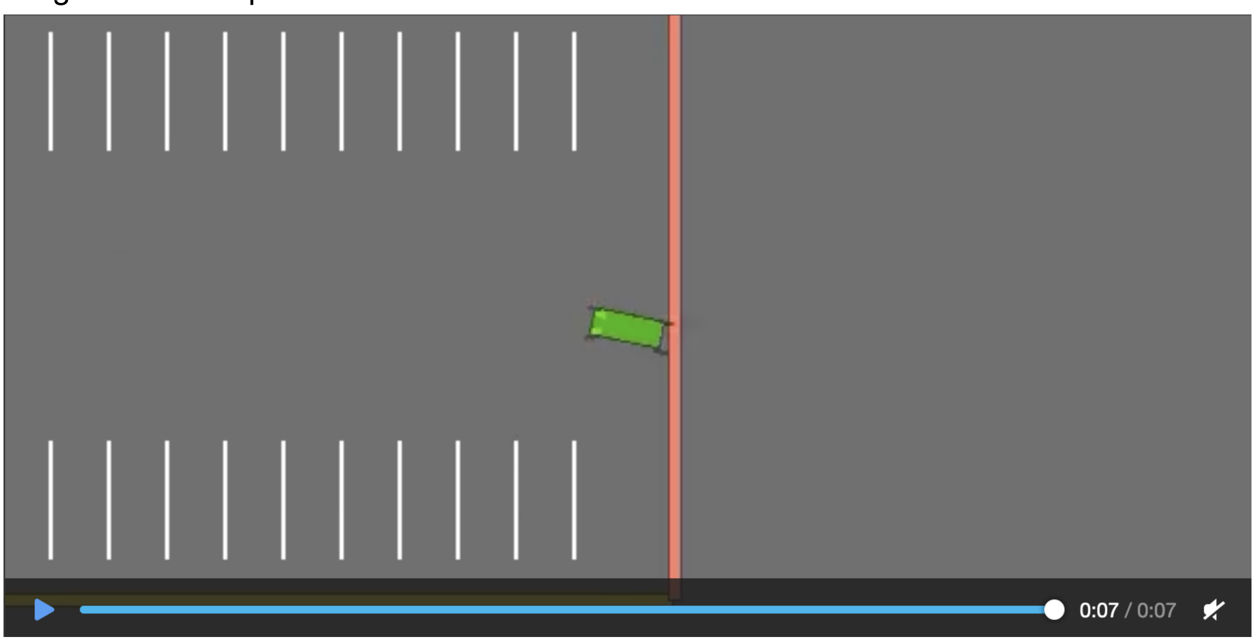


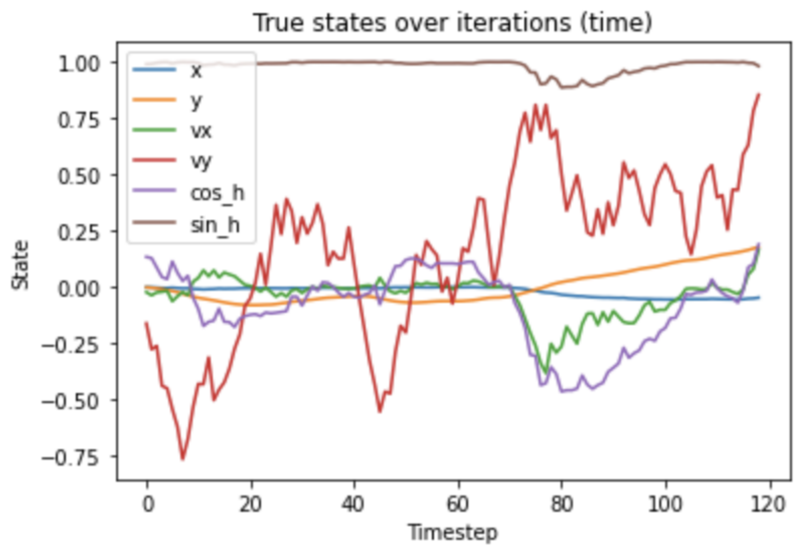

Image at end of episode:

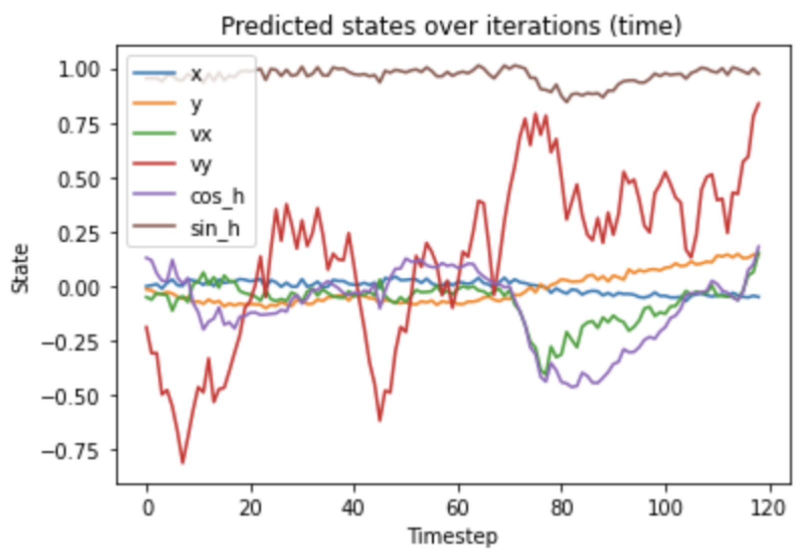


**Exercise 2. Model Based Reinforcement Learning with Neural Network.**

1. Construct a NN model with one hidden layer with random initialization (the structure is given in the Colab notebook). Implement the forward pass. Perform one run of episode and plot the untrained NN predictions and the true values of all the states over time!

```
plt.ylabel("State")
plt.legend(states_labels)
plt.show()
```



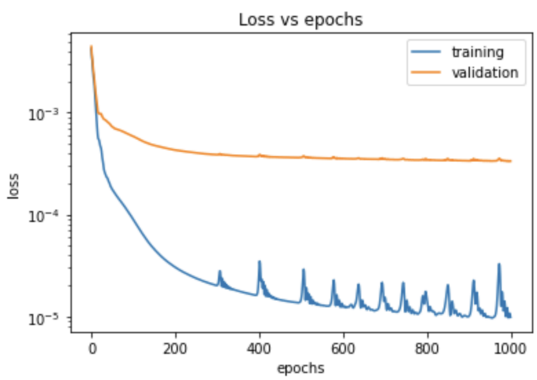



2. Create a batch of experiences dataset D with n = 2000 using the provided function. Train a NN model that consists of one hidden layer with 128 nodes and ReLU activation (the structure is already provided in the Colab notebook). Use this dataset with training and validation ratio 70%-30% using Adam optimizer with learning rate 0.01 and training epoch 1,000. Plot the MSE loss for training and validation set!

Training the model

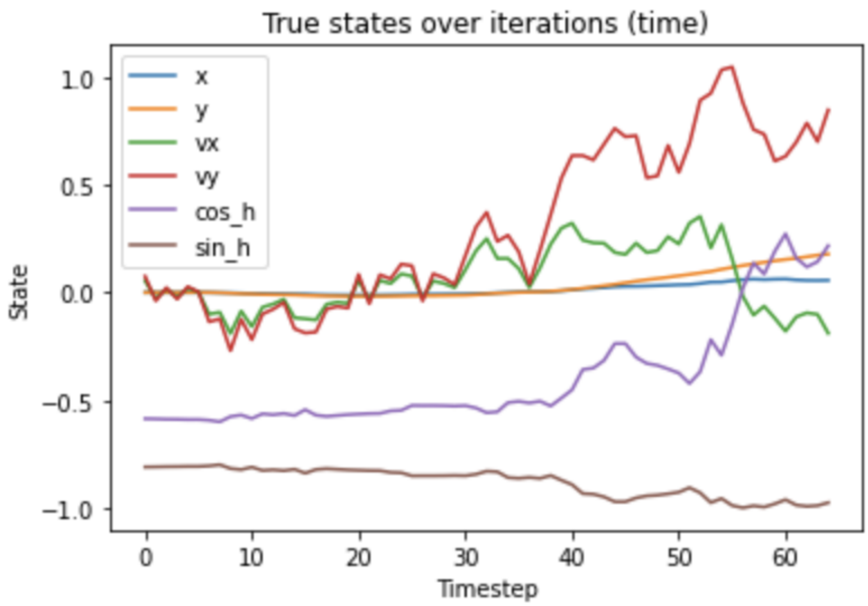
In [21]:
```
num_epochs = 1000 # REPLACE THE THREE DOTS WITH YOUR OWN CODE
learning_rate = 0.01 # REPLACE THE THREE DOTS WITH YOUR OWN CODE
train(dynamics_nn, train_data, validation_data, epochs=num_epochs, learning_rate = learning_rate)
```

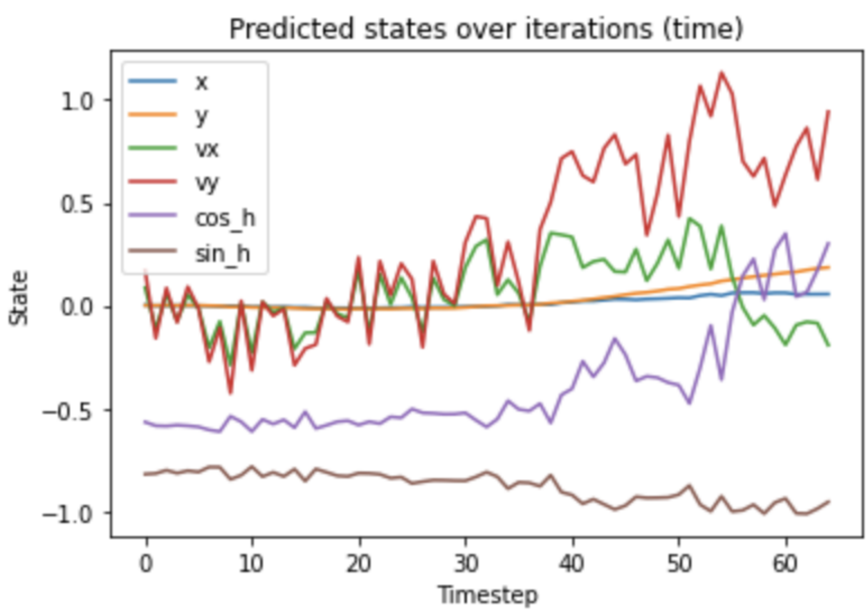
```
0%|          | 0/1000 [00:00<?, ?it/s]
```



3. Perform one run of episode again and plot the trained NN predictions and the true values of all the states over time!

```
plt.xlabel("Timestep")
plt.ylabel("State")
plt.legend(states_labels)
plt.show()
```

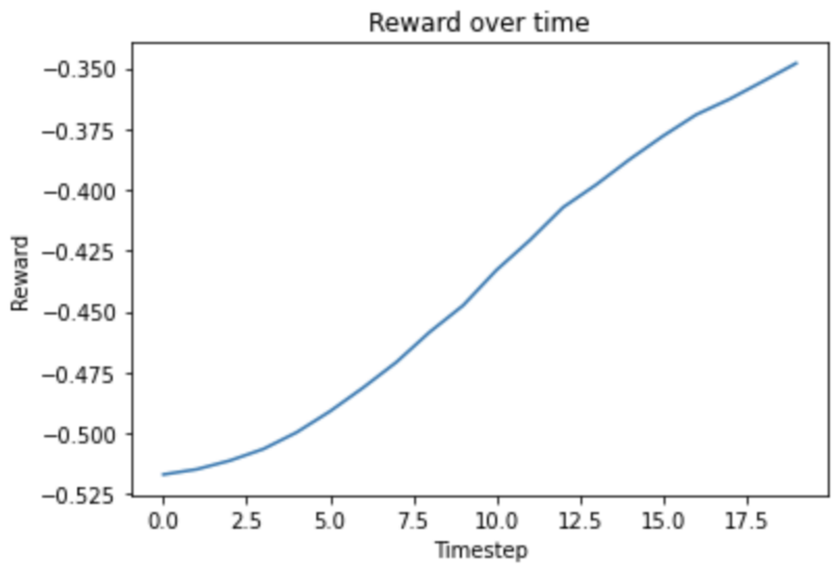


4. Implement the CEM planner. Combine the NN model with CEM planner. Evaluate the performance of your model when planning using a time horizon H = 5 and K = 10 possible action sequences only for n = 20 timesteps. Plot the rewards over time!

In [28]:

```
# Plot the rewards over time step

plt.plot(reward_data_nn)
plt.title("Reward over time")
plt.xlabel("Timestep")
plt.ylabel("Reward")
plt.show()
```



**Exercise 3. Model Based Reinforcement Learning with Gaussian Process.**

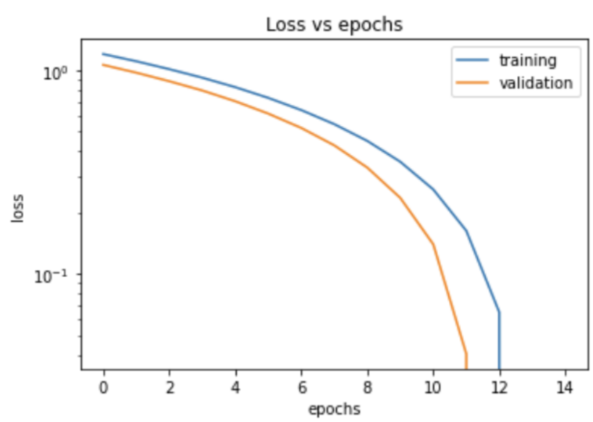
1. Construct and train the provided GP model. Train the model using dataset D from 2.2 with training and validation ratio 20%-80% using Adam optimizer with learning rate 0.2 and training epoch 15. Note that GP is more sample-efficient than NN, so we need fewer training samples for GP. More training samples will also slow down the speed. Plot the NLL loss for training and validation set!

```
    losses[i] = [loss.detach().numpy(), validation_loss.detach().numpy()] #st

    print('Iter %d/%d - Loss: %.3f - Val Los: %.3f' % (i + 1, epochs, loss.ite
```

```
  0%|          | 0/15 [00:00<?, ?it/s]

Iter 1/15 - Loss: 1.203 - Val Los: 1.064
Iter 2/15 - Loss: 1.109 - Val Los: 0.975
Iter 3/15 - Loss: 1.015 - Val Los: 0.885
Iter 4/15 - Loss: 0.921 - Val Los: 0.797
Iter 5/15 - Loss: 0.828 - Val Los: 0.706
Iter 6/15 - Loss: 0.733 - Val Los: 0.614
Iter 7/15 - Loss: 0.639 - Val Los: 0.522
Iter 8/15 - Loss: 0.545 - Val Los: 0.429
Iter 9/15 - Loss: 0.451 - Val Los: 0.333
Iter 10/15 - Loss: 0.356 - Val Los: 0.237
Iter 11/15 - Loss: 0.260 - Val Los: 0.140
Iter 12/15 - Loss: 0.163 - Val Los: 0.041
Iter 13/15 - Loss: 0.065 - Val Los: -0.058
Iter 14/15 - Loss: -0.034 - Val Los: -0.159
Iter 15/15 - Loss: -0.134 - Val Los: -0.259
```

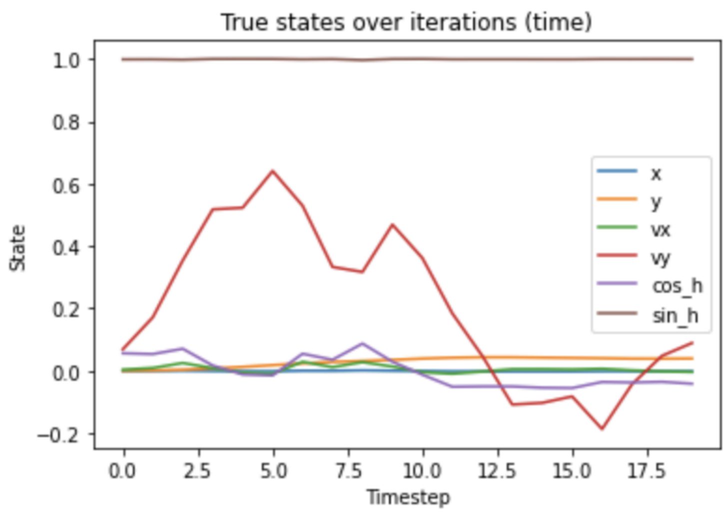
In [34]:

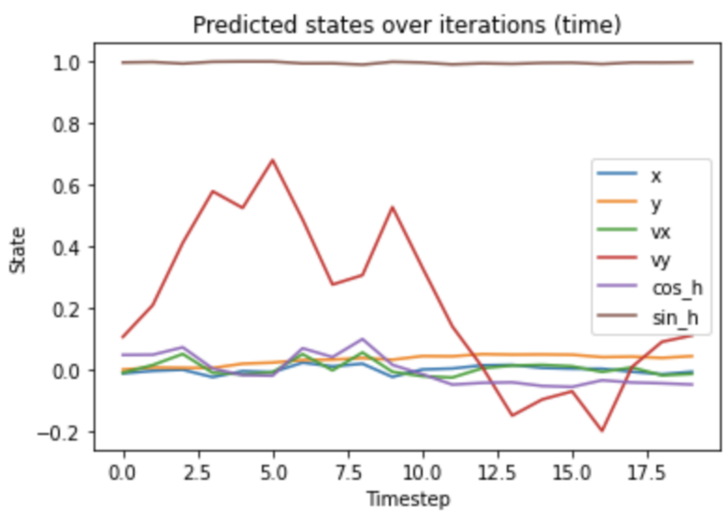
```
# Plot the training and validation losses

plt.plot(losses)
plt.title("Loss vs epochs")
plt.yscale("log")
plt.xlabel("epochs")
plt.ylabel("loss")
plt.legend(["training", "validation"])
plt.show()
```



2. Perform one run of episode again for only n = 20 timesteps and plot the trained GP predictions and the true values of all the states over time!

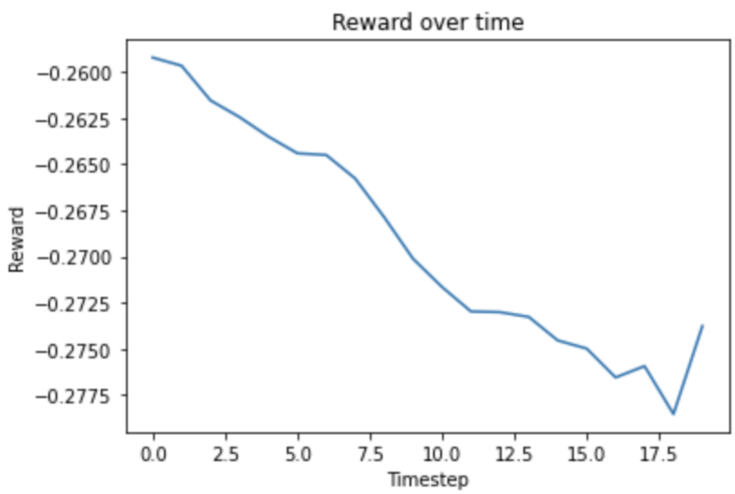
```
plt.title("Predicted states over iterations (time)")
plt.xlabel("Timestep")
plt.ylabel("State")
plt.legend(states_labels)
plt.show()
```



3. Combine the GP model with planning. Evaluate the performance of your model when planning using a time horizon H = 5 and K = 10 possible action sequences only for n = 20 timesteps. Plot the rewards over time!

In [38]:
```python
# Plot the rewards over time step

plt.plot(gp_reward)
plt.title("Reward over time")
plt.xlabel("Timestep")
plt.ylabel("Reward")
plt.show()
```




4. Compare NN and GP with the following metrics: final reward, prediction error, and computation time!

9]:
```python
# Get the final reward (when using NN and GP with CEM)

print(f"NN with CEM final reward: {np.sum(reward_data_nn)} - GP with CEM final reward: {np.sum(gp_reward)}") # R
```

```
NN with CEM final reward: -8.756728981409196 - GP with CEM final reward: -5.384138609503483
```

Computing the prediction error, and computation time between NN and GP

0]:
```python
# Compute the prediction error of NN and GP models during the planning horizon

gp_error = np.mean((np.array(state_data_gp) - np.array(pred_data_gp))**2) # REPLACE THE THREE DOTS WITH YOUR OWN
nn_error = np.mean((np.array(state_data) - np.array(pred_data))**2)
print(f"Mean Squared Error for NN: {nn_error} - Mean Squared Error for GP: {gp_error}")

# Get the computation time of running CEM with NN (from 2.4)and GP (from 3.3)
print(f"Computation time of NN with CEM: {np.sum(nn_time)} seconds - Computation time of GP with CEM: {np.sum(gp
```

```
Mean Squared Error for NN: 0.0027540271064266826 - Mean Squared Error for GP: 0.00034322224687235973
Computation time of NN with CEM: 0.7243785858154297 seconds - Computation time of GP with CEM: 156.52282643318176 s
econds
```